

Datenbanksysteme

Kapitel 9

Einführung in XML, XPath, XQuery und XSLT

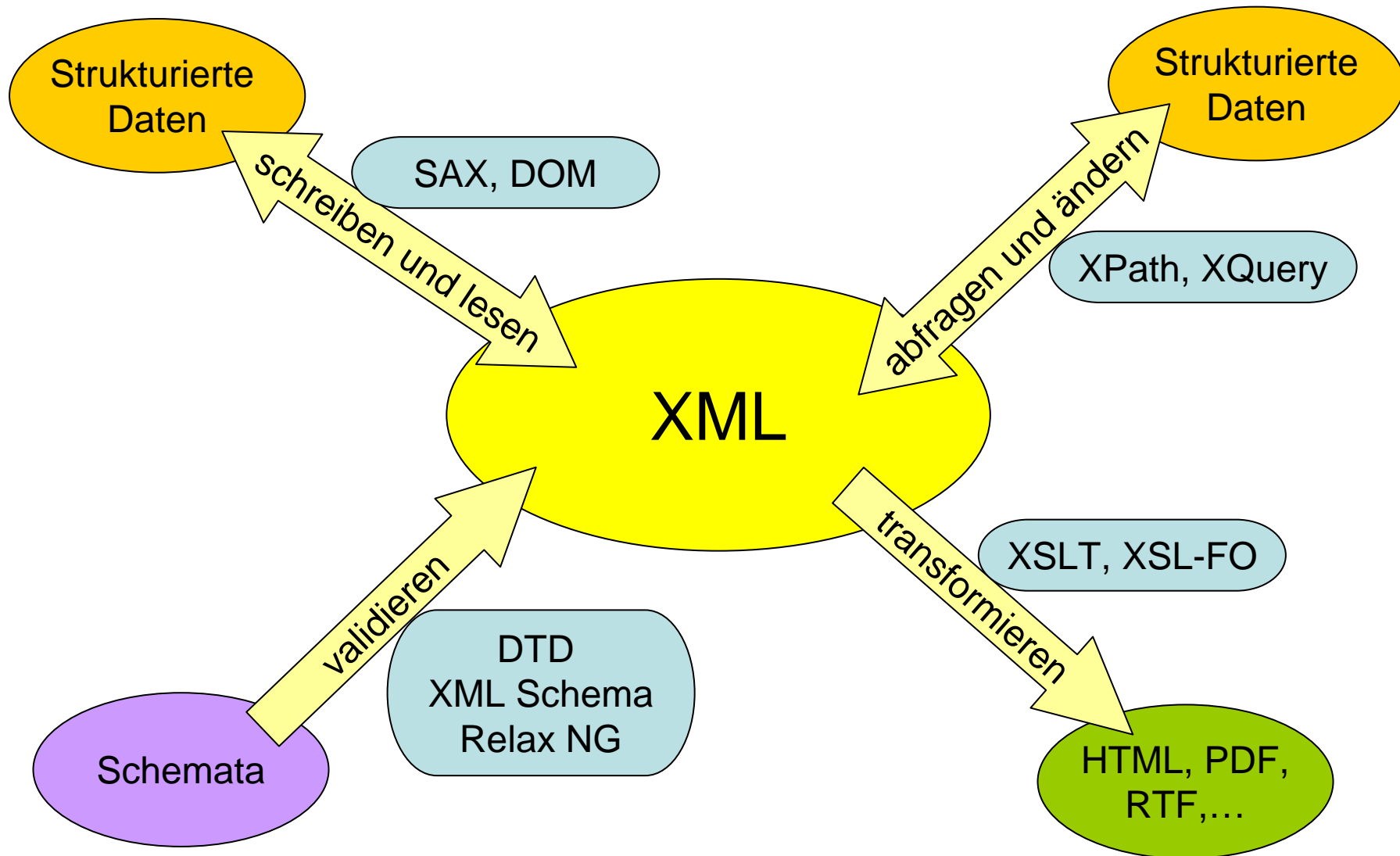
Martin Giesecking

Was ist XML?

- XML (Extensible Markup Language) ist eine Meta-Auszeichnungssprache zur textbasierten Beschreibung hierarchisch strukturierter Daten
 - Spezifikation des World Wide Web Consortiums (W3C)
 - XML definiert kein konkretes Dateiformat sondern legt fest, welche syntaktischen Bestandteile verwendet und wie sie kombiniert werden dürfen
 - auf Grundlage der Regeln können konkrete Dateiformate mit definierter Struktur und Semantik abgeleitet werden
- ein konkretes Dateiformat, das auf Grundlage der XML-Spezifikation festgelegt wurde, wird allgemein **XML-Format** genannt
 - entsprechend nennt man Dateien in diesem Format **XML-Dateien**
 - Beispiele für XML-Formate sind *RSS*, *MathML*, *SVG*, *XHTML*

Beispiel einer XML-Datei

```
<?xml version="1.0"?>
<!-- Vorlesungsverzeichnis Sommersemester 2011 -->
<vorlesungsverzeichnis>
  <abschnitt titel="Mathematik & Informatik">
    <abschnitt titel="Grundstudium">
      <veranstaltung nr="1.234" typ="v">
        <dozent>
          <titel>Prof. Dr.</titel>
          <vname>Klaus</vname>
          <nname>Meier</nname>
        </dozent>
        <titel>Einführung in die monotone Algebra I</titel>
        <zeit>Mo 10:00-12:00</zeit>
        <raum>12/13</raum>
      </veranstaltung>
      <veranstaltung nr="1.247" typ="ü">
        <dozent>
          <vname>Sandra</vname>
          <nname>Schmidt</nname>
        </dozent>
        <titel>Übung zur Einführung in die monotone Algebra I</titel>
        <zeit>Mi 8:00-12:00</zeit>
        <raum>97/E9</raum>
      </veranstaltung>
    </abschnitt>
  </abschnitt>
</vorlesungsverzeichnis>
```



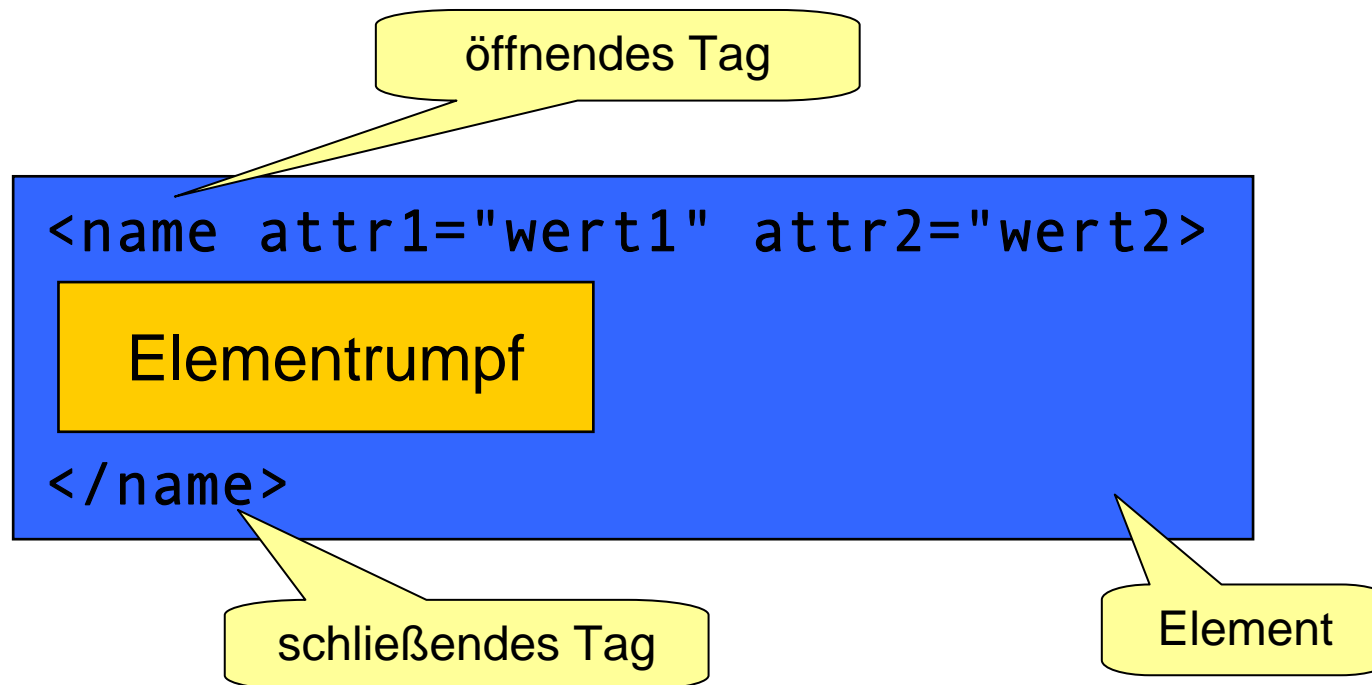
Bausteine einer XML-Datei

- XML-Dateien können aus folgenden Bausteinen zusammengesetzt werden:
 - Elemente
 - Attribute
 - Text und Entities
 - Kommentare
 - Verarbeitungsanweisungen (processing instructions)
 - CDATA-Abschnitte
 - DOCTYPE-Angabe

```
<?xml version="1.0"?>
<!-- Vorlesungsverzeichnis Sommersemester 2011 -->
<vorlesungsverzeichnis>
  <abschnitt titel="Mathematik & Informatik">
    <abschnitt titel="Grundstudium">
      <veranstaltung nr="1.234" typ="v">
        <dozent>
          <titel>Prof. Dr.</titel>
          <vname>Klaus</vname>
          <nname>Meier</nname>
        </dozent>
        <titel>Einführung in die ...</titel>
        <zeit>Mo 10:00-12:00</zeit>
        <raum>12/13</raum>
      </veranstaltung>
    </abschnitt>
  </abschnitt>
</vorlesungsverzeichnis>
```

Elemente

- die strukturierenden Bausteine eines XML-Dokuments werden **Elemente** genannt
- sie bestehen aus einem öffnenden und einem schließenden **Tag** sowie dem **Elementinhalt** (oder **Elementrumpf**)
 - anders als z.B. bei HTML muss jedes Element explizit ein öffnendes und schließendes Tag (Start- und End-Tag) besitzen



- **öffnende Tags** haben immer die Form
`<name attr1="wert1" attr2="wert2" ...>`
 - der Name kann aus Buchstaben, Ziffern und den Zeichen `_ . - ' "` bestehen
 - erstes Zeichen muss ein Buchstabe sein
 - es wird zwischen Groß- und Kleinschreibung unterschieden
 - darf nicht mit *xml*, *Xml*, *xMI*, *xmL*, *XMI*, *XmL*, *xML* oder *XML* beginnen
 - zwischen „<“ und *name* darf sich kein Leerzeichen befinden
 - Attribute werden in der angegebenen Form durch Whitespaces getrennt hinter dem Elementnamen aufgelistet
 - für Attributnamen gelten die gleichen Vorgaben wie für Elementnamen
 - Attributwerte werden wahlweise in einfache oder doppelte Anführungszeichen eingeschlossen
 - die Reihenfolge der Attribute ist nicht signifikant
 - jeder Attributname darf in der Attributliste nur einmal auftauchen
- **schließende Tags** haben immer die Form
`</name>`
 - der Name muss mit dem des zugehörigen öffnenden Tags übereinstimmen
 - schließende Tags enthalten keine weiteren Informationen

Elemente

- der Elementrumpf ist entweder leer oder besteht aus einer Folge von weiteren Elementen, Text, Kommentaren und/oder Verarbeitungsanweisungen
 - Elemente können beliebig tief geschachtelt werden

```
<veranstaltung nr="1.234" typ="v">
  <dozent geschlecht="m">
    <!-- ein Kommentar -->
    <titel>Prof. Dr.</titel>
    <vname>Klaus</vname>
    <nname>Meier</nname>
  </dozent>
  Dies ist ein Text.
  <titel>Einführung in die ...</titel>
  <zeit>Mo 10:00-12:00</zeit>
  <raum>12/13</raum>
</veranstaltung>
```

- für Elemente mit leerem Rumpf gibt es die Kurzschreibweise
`<name attr1="val1" attr2="val2" ... />`

```
<uhrzeit zeitzone="GMT"></uhrzeit>
```

ist identisch mit

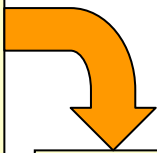
```
<uhrzeit zeitzone="GMT"/>
```


Elemente

- jedes XML-Dokument muss genau ein äußeres Element, das so genannte **Wurzelement** besitzen
 - es stellt quasi den Rahmen oder den Container für die eigentlichen Daten dar
 - außerhalb des Wurzelements sind nur Kommentare, Verarbeitungsanweisungen, eine DOCTYPE-Anweisung und Whitespace erlaubt

Fehler: kein (eindeutiges) Wurzelement

```
<?xml version="1.0"?>  
<veranstaltung nr="1.234">  
  ...  
</veranstaltung>  
  
<veranstaltung nr="4.321">  
  ...  
</veranstaltung>
```



```
<?xml version="1.0"?>  
<veranstaltungen>  
  <veranstaltung nr="1.234">  
    ...  
  </veranstaltung>  
  <veranstaltung nr="4.321">  
    ...  
  </veranstaltung>  
</veranstaltungen>
```

Kommentare und Entities

- neben Text und Elementen spezifiziert XML noch weitere Konstrukte

- Kommentare

`<!-- dies ist ein Kommentar -->`

- die Zeichenfolge - - ist innerhalb von Kommentaren nicht erlaubt

- Entity-Referenzen

`&name; &#dez; &#xhex;`

- bezeichnen ein einzelnes Zeichen oder eine Zeichenfolge
- die Zeichen < und & haben in der XML-Syntax eine besondere Bedeutung (Metazeichen), und dürfen nicht als normaler Textbestandteil verwendet werden
- zahlreiche Zeichen sind nicht im normalen Zeichenvorrat eines Zeichensatzes enthalten, so dass sie nicht direkt eingegeben werden können (z.B. mathematische oder musikalische Zeichen, Ligaturen usw.)

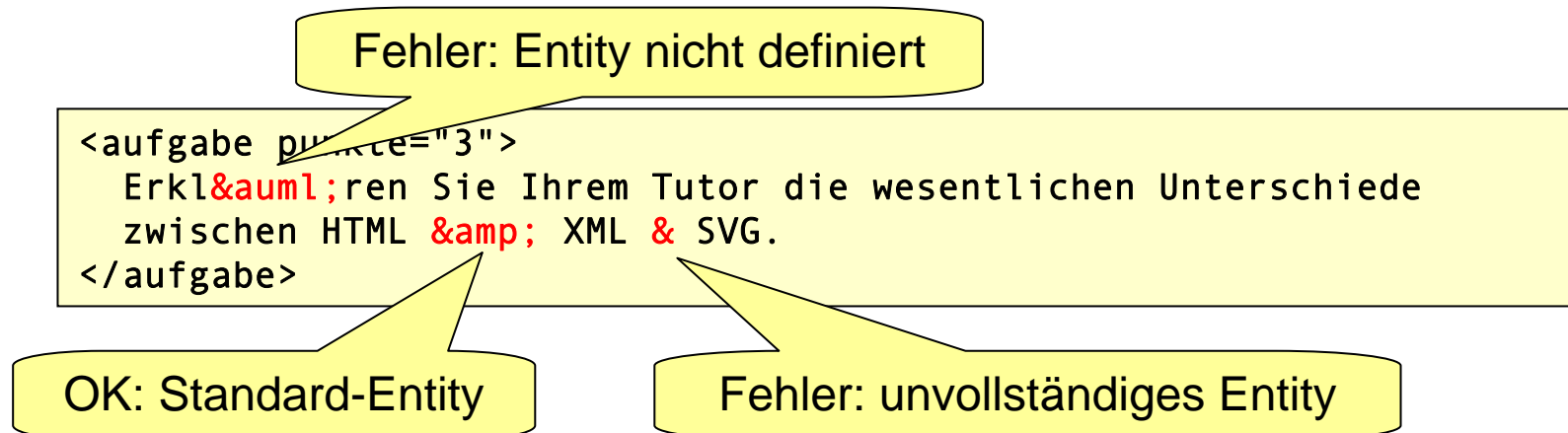
<	<
>	>
'	'
"	"e;
&	&

```
<aufgabe punkte="1">  
  <!-- eine einfache Frage zum Einstieg -->  
  Wofür steht in XML die Zeichenfolge &lt;!- ... -->?  
</aufgabe>
```

Entity-Referenzen

- im Gegensatz zu HTML definiert XML nur fünf benannte Entities
- alle anderen aus HTML bekannten Entities, wie z.B. `ä` `ß` ` ` usw. sind in XML nicht definiert und führen bei Verwendung zu einem Fehler
- es gibt die Möglichkeit, neue benannte Entities zu definieren

<	<
>	>
'	'
"	"e;
&	&



- numerische Entity-Referenzen bezeichnen jeweils ein einzelnes Unicode-Zeichen
 - Eurozeichen: `€` (dezimal), `€` (hexadezimal)

XML-Namensräume

- Namensräume ermöglichen es, zusammengehörige Elemente zu bündeln und von „fremden“ Elementen abzugrenzen
- Namensräume werden in XML durch eindeutige URIs (Uniform Resource Identifiers) benannt
 - Zeichenkette, die einen Namensraum eindeutig identifiziert
 - haben meist die Form einer URL (muss nicht existieren)
- In XML-Dokumenten werden die Namensräume mit frei definierbaren Präfixen verknüpft, die den Elementen durch einen Doppelpunkt getrennt vorangestellt werden

```
<buch xmlns:pers="http://xyz.de/person">  
  <titel>Das Leben der Affen</titel>  
  <autor>  
    <pers:person>  
      <pers:titel>Prof. Dr.</pers:titel>  
      <pers:vorname>Jim</pers:vorname>  
      <pers:nachname>Panse</pers:nachname>  
    </pers:person>  
  </autor>  
</buch>
```

```
<buch>  
  <titel>Das Leben der Affen</titel>  
  <autor>  
    <person xmlns="http://xyz.de/person">  
      <titel>Prof. Dr.</titel>  
      <vorname>Jim</vorname>  
      <nachname>Panse</nachname>  
    </person>  
  </autor>  
</buch>
```

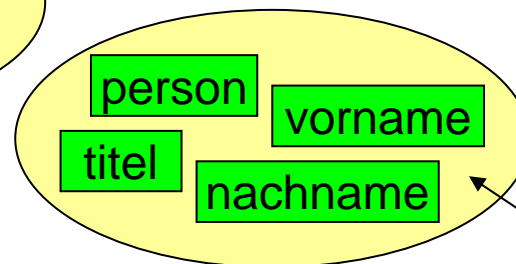
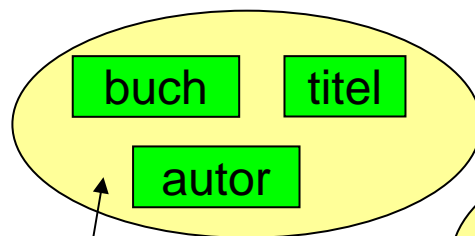
XML-Namensräume

```
<buch xmlns:pers="http://xyz.de/person">  
  <titel>Das Leben der Affen</titel>  
  <autor>  
    <pers:person>  
      <pers:titel>Prof. Dr.</pers:titel>  
      <pers:vorname>Jim</pers:vorname>  
      <pers:nachname>Panse</pers:nachname>  
    </pers:person>  
  </autor>  
</buch>
```

```
<buch>  
  <titel>Das Leben der Affen</titel>  
  <autor>  
    <person xmlns="http://xyz.de/person">  
      <titel>Prof. Dr.</titel>  
      <vorname>Jim</vorname>  
      <nachname>Panse</nachname>  
    </person>  
  </autor>  
</buch>
```

- definiert den Präfix *pers* für den Namensraum *http://xyz.de/person*
- ist nur innerhalb des Elements bekannt, in dem er definiert wird

- setzt den Standard-Namensraum auf *http://xyz.de/person*
- das Element und die Kindelemente werden dem Namensraum zugeordnet
- ein expliziter Präfix vor den Elementnamen ist hier nicht erforderlich



Wohlgeformte XML-Dokumente

- ein XML-Dokument heißt **wohlgeformt**, wenn es die Syntax- und Strukturvorgaben der XML-Spezifikation einhält
 - über 100 Regeln, die größtenteils intuitiv aus den bisher beschriebenen Aspekten hervorgehen
- die Wohlgeformtheit kann ohne Kenntnis der in einem Dokument zulässigen Elemente überprüft werden

nicht wohlgeformt

```
<?xml version="1.0"?>
<blatt nr="1">
  <aufgabe punkte="4">
    <list>
      <li>Punkt 1</li>
      <li>Punkt 2
    </aufgabe>
    </li>
  </list>
</Blatt>

<blatt nr="2">
  <aufgabe Punkte="6"/>
</blatt>
```

wohlgeformt

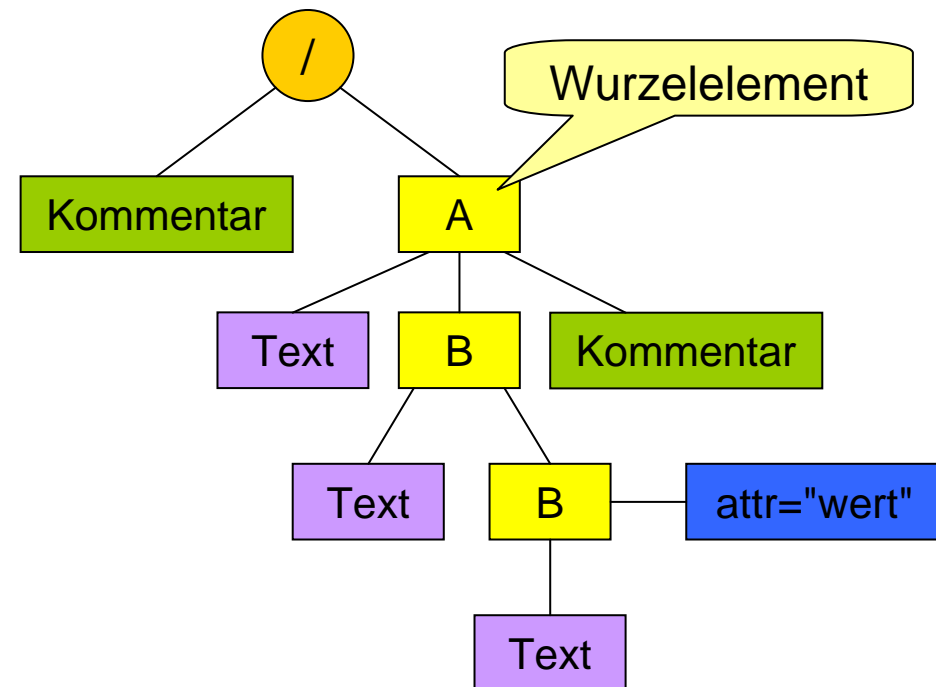
```
<?xml version="1.0"?>
<blattsammlung>
  <blatt nr="1">
    <aufgabe punkte="4">
      <list>
        <li>Punkt 1</li>
        <li>Punkt 2</li>
      </list>
    </aufgabe>
  </blatt>

  <blatt nr="2">
    <aufgabe PUNKTE="6"/>
  </blatt>
</blattsammlung>
```

Struktur von XML-Dokumenten

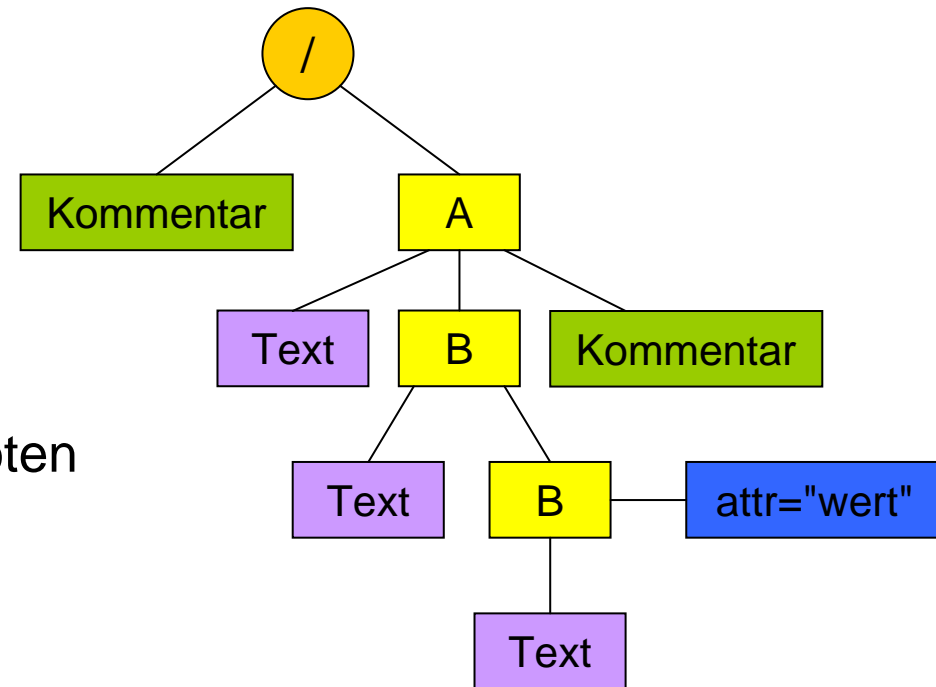
- XML-Dokumente besitzen grundsätzlich eine Baumstruktur
 - es gibt genau einen Wurzelknoten vom Typ *Document*
 - bis auf den **Dokumentknoten** haben alle Knoten einen eindeutigen Elternknoten
 - Attribute sind sog. **assoziierte Knoten**, die zwar ein Elternelement besitzen, selbst aber keine Kinder sind
- der Dokumentknoten muss genau einen Element-Kindknoten, das **Wurzelelement**, besitzen

```
<?xml version="1.0"?>
<!-- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B attr="wert">
      dritter Textteil
    </B>
  </B>
</A>
<!-- zweiter Kommentar -->
```



XML-Knotentypen

- entsprechend der syntaktischen Elemente einer XML-Datei wird zwischen verschiedenen Knotentypen unterschieden
 - Dokumentknoten
 - Elementknoten
 - Textknoten
 - Kommentarknoten
 - Attributknoten
 - Namensraumknoten
 - Verarbeitungsanweisungsknoten



XPath: Navigieren in XML-Bäumen

- eine zentrale Operation auf XML-Bäumen ist das Auswählen von einzelnen Knoten oder Knotenmengen nach bestimmten Kriterien
- die Lokatorsprache **XPath** stellt einen Formalismus zur Adressierung von Knoten in XML-Bäumen bereit
 - zusätzlich werden mathematische und logische Operatoren sowie eine überschaubare Anzahl von Funktionen bereitgestellt (XPath 1.0)
- XPath ist integraler Bestandteil von weiteren XML-Technologien, wie XSLT und XQuery (Obermenge von XPath 2.0)

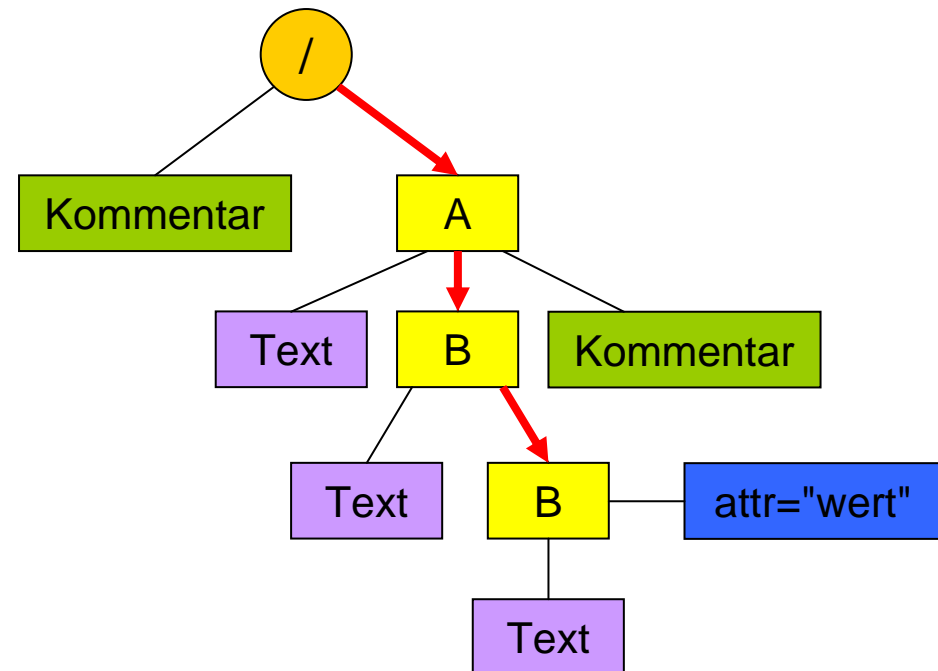
XPath-Software (Open Source)

- Firebug- und FirePath-Erweiterung für Firefox
 - Firefox unterstützt wie alle anderen Browser nur XPath 1.0
- Libxml2
 - <http://xmlsoft.org>
 - Kommandozeilen-Programm `xml lint` mit Option `--xpath`
- BaseX
 - <http://www.inf.uni-konstanz.de/dbis/basex>
 - XML-Datenbank mit grafischer Oberfläche und XQuery-Schnittstelle
 - XQuery ist eine Obermenge von XPath 2.0
- XQilla
 - <http://xqilla.sourceforge.net/HomePage>
 - XQuery-Prozessor

Beispiel: einfache Pfadangabe

- Knoten werden in XPath mit Hilfe von **Pfadangaben** ausgewählt
 - einfache XPath-Pfadangaben erinnern an Unix-Pfade zur Navigation im Dateisystem
 - ausgehend von der Dokumentwurzel können die Kindknoten schrittweise ausgewählt werden
- das Ergebnis einer Pfadangabe ist immer eine **Knotenmenge**
 - eine Liste mit Referenzen auf alle passenden Knoten ohne Dopplungen
- Beispiel: /A/B/B

```
<?xml version="1.0"?>
<!-- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B attr="wert">
      dritter Textteil
    </B>
  </B>
  <!-- zweiter Kommentar -->
</A>
```



- jede Pfadangabe besteht aus einer Folge sogenannter **Location-Steps**
 - werden durch Slashes (/) voneinander getrennt
 - Beispiel: `step1/step2/step3`
 - jeder Location-Step wählt relativ zum vorangehenden Location-Step bzw. zum Kontextknoten eine Knotenmenge des XML-Baums aus
- jeder Location-Step besteht aus maximal drei Komponenten
 - einem optionalen **Achsenbezeichner**
 - einem **Knotentest**
 - einem oder mehreren optionalen **Prädikat(en)**
 - Syntax: `achsenbezeichner::knotentest[prädikat]`
 - Beispiel: `ancestor::A[B]`

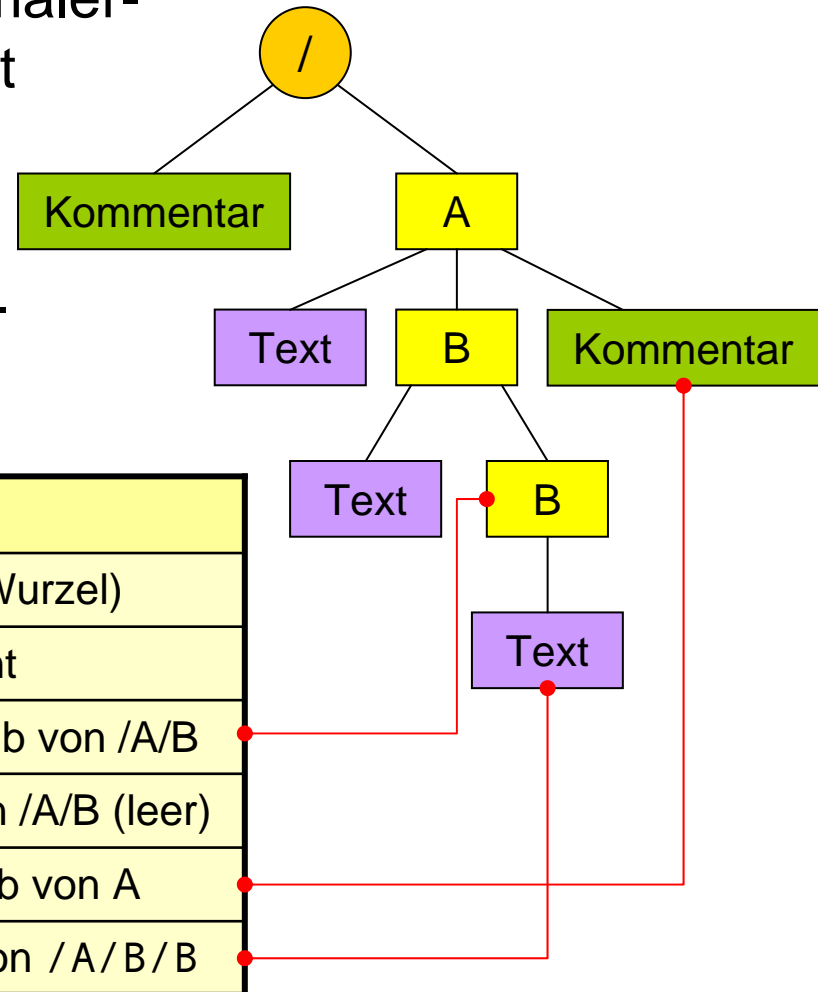
Knotentests

- im einfachsten Fall besteht ein Location-Step lediglich aus einem **Knotentest**
 - prüft, ob es an der aktuellen Position Knoten eines bestimmten Typs gibt
 - das Ergebnis eines Knotentests ist immer eine Menge aller passenden Knoten
 - falls es keine passenden Knoten gibt, ist die Ergebnismenge leer

Knotentyp	Syntax des Knotentests
Dokumentwurzel	/
Element <i>A</i>	<i>A</i>
Attribut <i>attr</i>	@ <i>attr</i>
Text	text()
Kommentar	comment()
alle Knotentypen	node()

Knotentests

- Die Knotentests `text()` und `comment()` selektieren Text- bzw. Kommentarknoten (unabhängig von ihrem Inhalt)
- Elemente und Attribute werden normalerweise über ihren Namen ausgewählt
- der Knotentest `*` selektiert Elementknoten unabhängig vom Namen
- der Knotentest `@*` selektiert Attributknoten unabhängig vom Namen

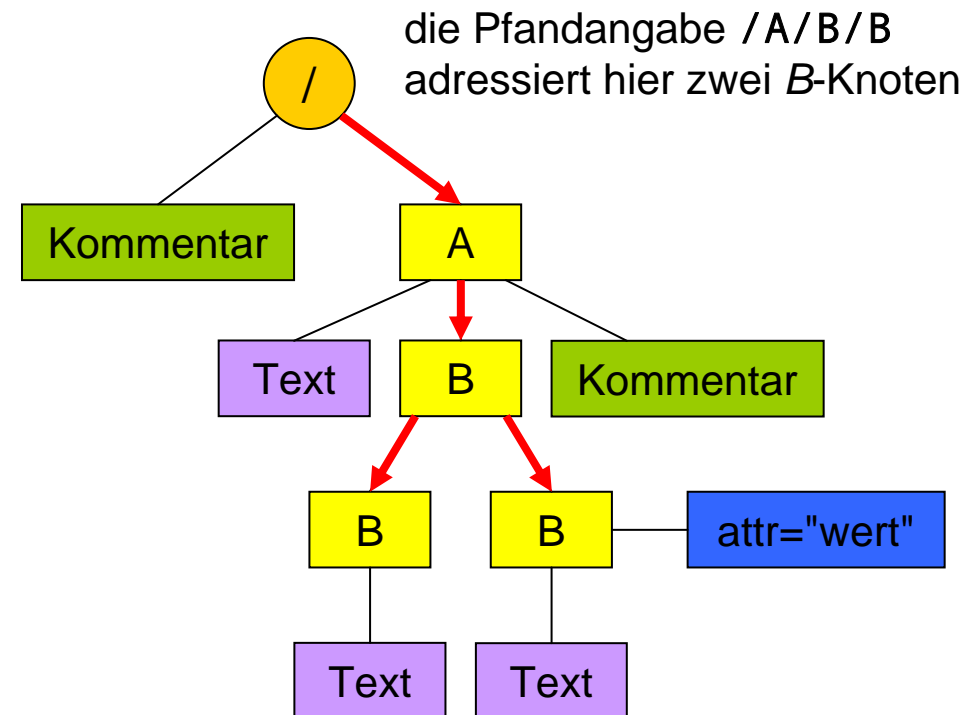


XPath-Ausdruck	Bedeutung
<code>/</code>	Dokumentknoten (Wurzel)
<code>/*</code>	Wurzelelement
<code>/A/B/*</code>	Elementknoten unterhalb von <code>/A/B</code>
<code>/A/B/A</code>	A-Element unterhalb von <code>/A/B</code> (leer)
<code>/A/comment()</code>	Kommentar unterhalb von A
<code>/A/B/B/text()</code>	Textknoten unterhalb von <code>/A/B/B</code>

Knotenmengen

- da Elementknoten mehrere gleichnamige und/oder gleichartige Knoten enthalten können, kann auch das Resultat eines Knotentests aus mehreren Knoten bestehen
 - das Resultat ist eine Knotenmenge
 - die Ergebnisknoten werden in Dokumentreihenfolge angeordnet (gilt nicht für alle Achsen)

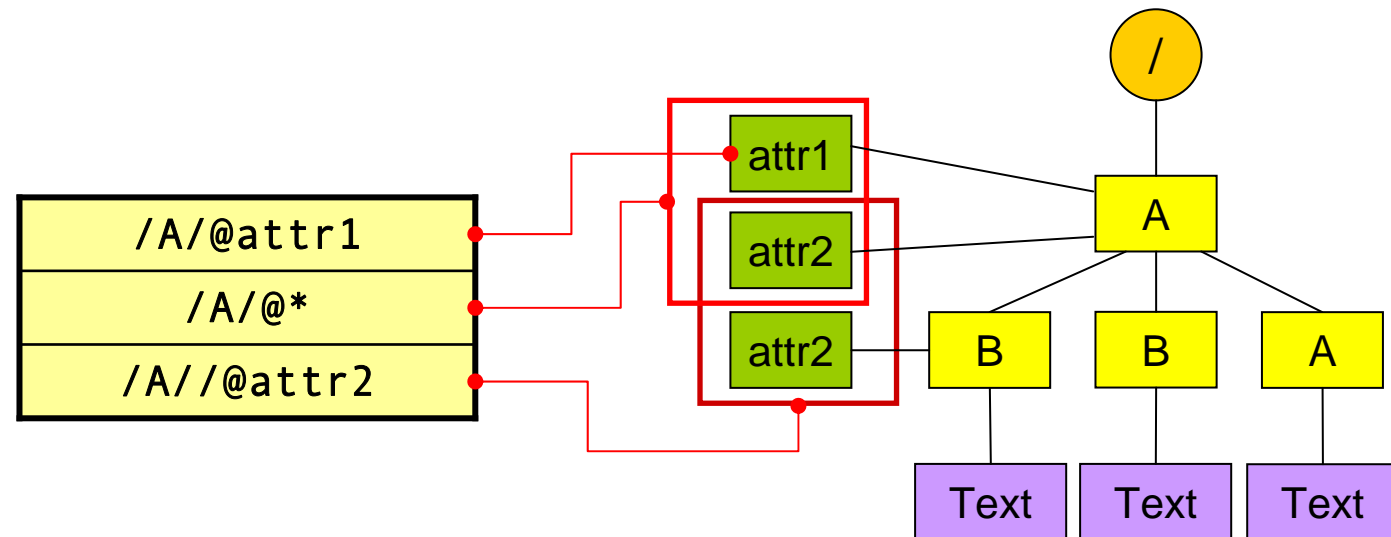
```
<?xml version="1.0"?>
<!-- erster Kommentar -->
<A>
  erster Textteil
  <B>
    zweiter Textteil
    <B>
      dritter Textteil
    </B>
    <B attr="wert">
      vierter Textteil
    </B>
  </B>
<!-- zweiter Kommentar -->
</A>
```



Attributknoten

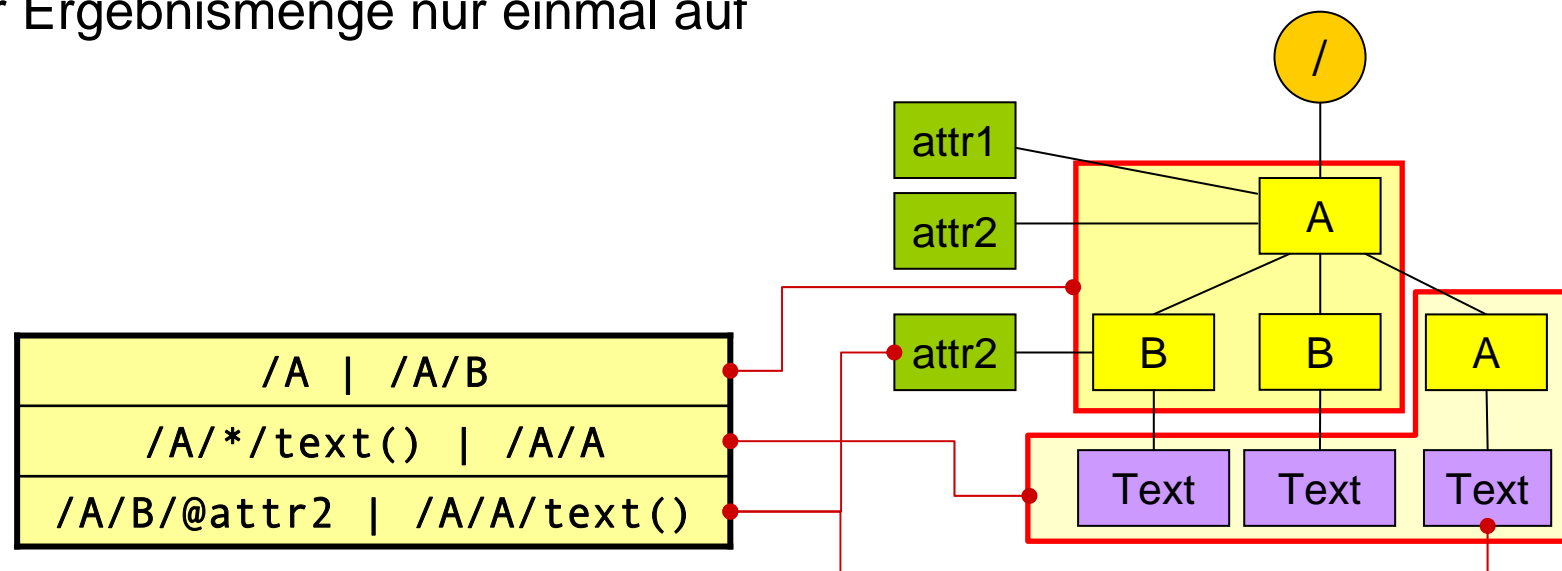
- die Attributknoten eines Elements werden durch einen Klammeraffen (@) gefolgt vom Attributnamen adressiert
 - mit @* erhält man alle Attributknoten der spezifizierten Elemente
- Attribute, die Namensraum-Präfixe definieren, können nicht mit @ adressiert werden
 - Zugriff muss mit Hilfe der *namespace*-Achse erfolgen

```
<?xml version="1.0"?>  
<A attr1="val1" attr2="val2">  
  <B attr2="val3">  
    Text  
  </B>  
  <B>Text</B>  
  <A>Text</A>  
</A>
```



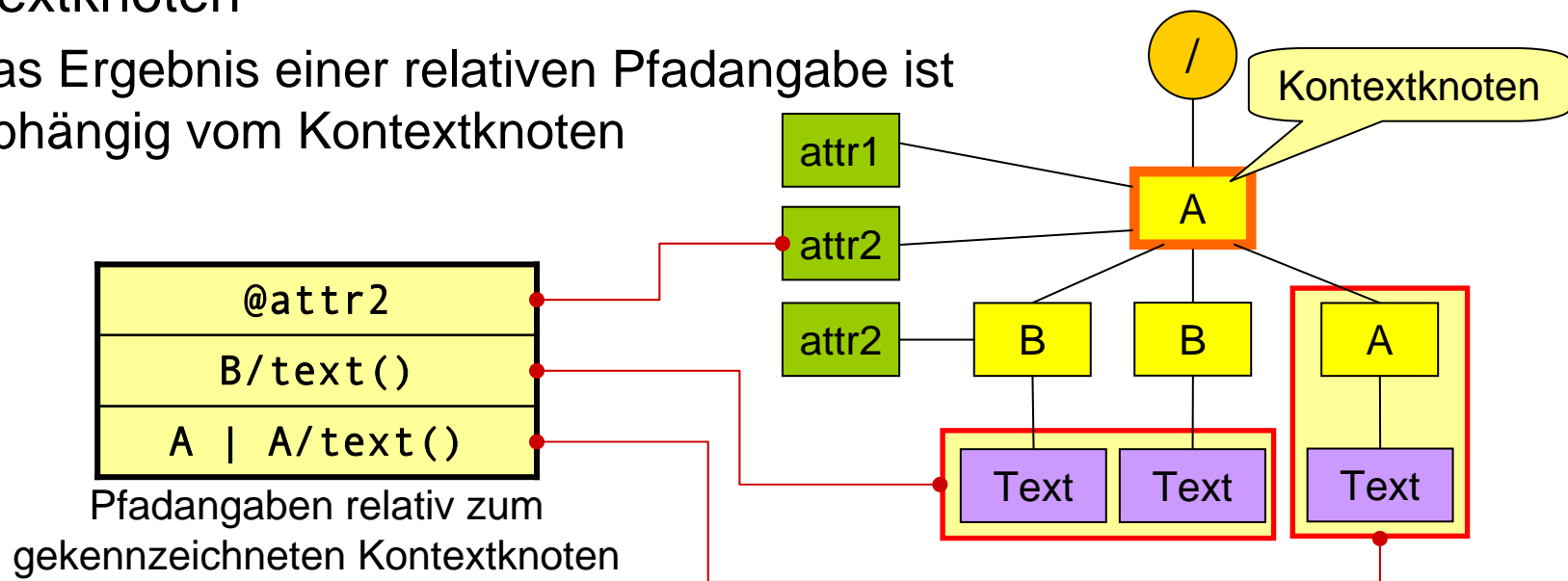
Vereinigung von Knotenmengen

- mit Hilfe des Kompositionsoperators | (senkrechter Strich) lassen sich zwei Knotenmengen vereinigen
 - auf beiden Seiten des Operators steht eine Pfadangabe
 - der Kompositionsoperator bindet schwächer als alle anderen Pfadoperatoren
 - die Knoten der Ergebnismenge sind immer in Dokumentreihenfolge angeordnet
 - Knoten, die von beiden Knotentests ausgewählt werden, tauchen in der Ergebnismenge nur einmal auf



Absolute und relative Pfadangaben

- XPath unterscheidet zwischen absoluten und relativen Pfadangaben
- **absolute Pfadangaben** beginnen immer bei der Dokumentwurzel
 - da die Dokumentwurzel mit einem Slash adressiert wird, beginnen absolute Pfadangaben immer mit einem Slash
 - das Ergebnis einer absoluten Pfadangabe ist für dasselbe XML-Dokument immer gleich
- **relative Pfadangaben** beziehen sich auf zuvor ausgewählte Kontextknoten
 - das Ergebnis einer relativen Pfadangabe ist abhängig vom Kontextknoten

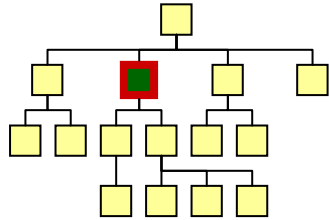


Achsen: Suchbereich von Knotentests angeben

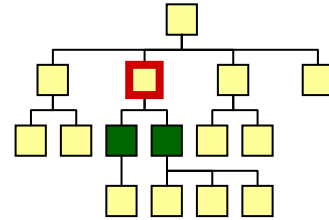
- wenn nicht anders angegeben, beziehen sich alle Knotentests immer auf die Kinder der aktuell adressierten Knoten
 - /A/B/C bedeutet:
 - suche alle *A*-Kindelemente des Dokumentknotens
 - suche in den gefundenen *A*-Elementen nach *B*-Kindelementen
 - suche in den gefundenen *B*-Elementen nach *C*-Kindelementen und liefere sie als Ergebnis zurück
 - jeder Location-Step bewirkt hier also einen Abstieg um eine Ebene im Baum
- mit Hilfe einer **Achsenangabe** kann der Bereich, auf den ein Knotentest angewendet werden soll, geändert werden, z.B. um
 - in Eltern- oder Geschwisterknoten zu suchen
 - rekursiv in allen Vorgänger- oder allen Folgeknoten zu suchen
 - Attribut- oder Namensraumknoten auszuwählen
- XPath 1.0 definiert 13 verschiedene Achsen

Achsen

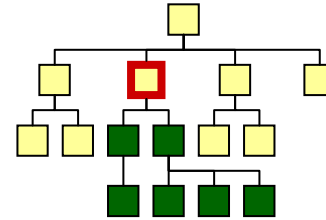
self



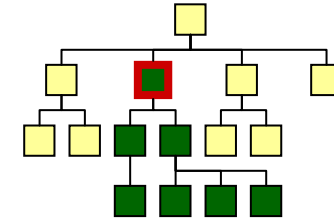
child



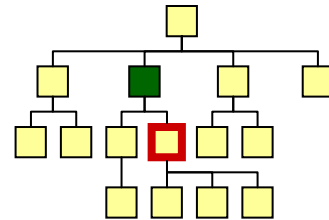
descendant



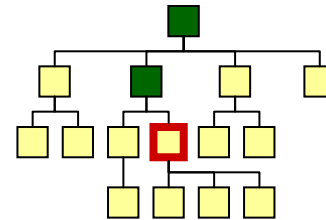
descendant-or-self



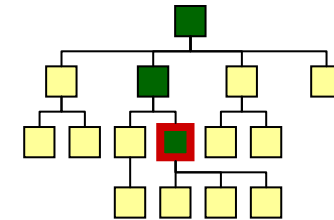
parent



ancestor

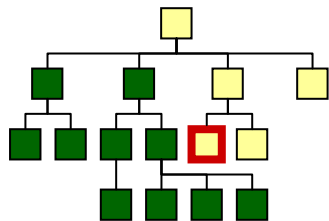


ancestor-or-self

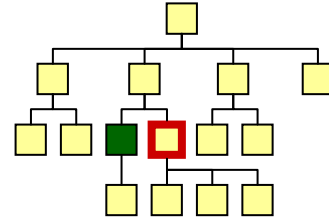


□ Kontextknoten
■ Knoten der Achse

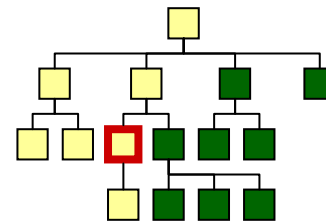
preceding



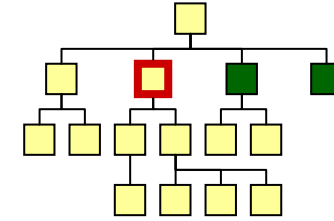
preceding-sibling



following



following-sibling

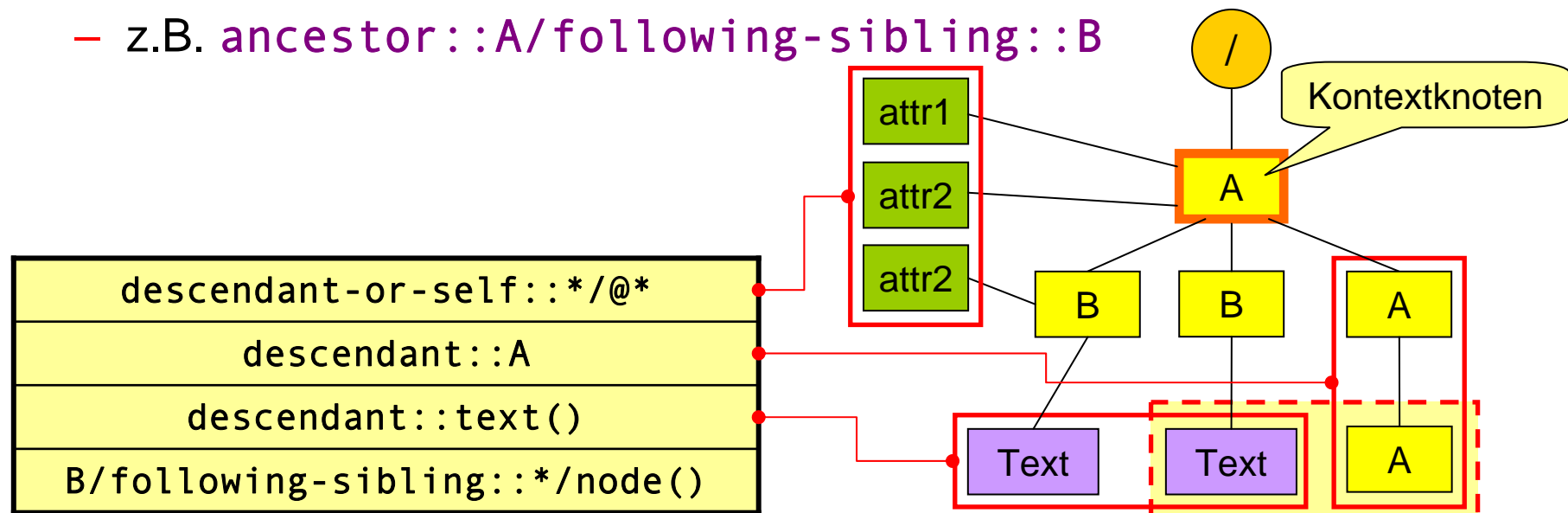


Achse	Beschreibung
self	aktueller Knoten
child	alle unmittelbaren Kinder des aktuellen Knotens
descendant	alle Nachfahren des aktuellen Knotens
descendant-or-self	wie <i>descendant</i> aber zuzüglich des aktuellen Knotens
parent	Elternknoten des aktuellen Knotens
ancestor	alle Vorfahren des aktuellen Knotens
ancestor-or-self	wie <i>ancestor</i> aber zuzüglich des aktuellen Knotens
preceding	alle vorausgehenden Knoten ohne Vorfahren des aktuellen Knotens
preceding-sibling	alle vorausgehenden Geschwisterknoten
following	alle nachfolgenden Knoten ohne Nachfahren des aktuellen Knotens
following-sibling	alle nachfolgenden Geschwisterknoten
attribute	alle Attributknoten des aktuellen Elements

- die preceding-Achse umfasst alle Knoten, die in der XML-Datei vor dem aktuellen Knoten stehen und keine Vorfahren des aktuellen Knotens sind
- die following-Achse umfasst alle Knoten, die in der XML-Datei hinter dem aktuellen Knoten stehen und keine Nachfahren des aktuellen Knotens sind

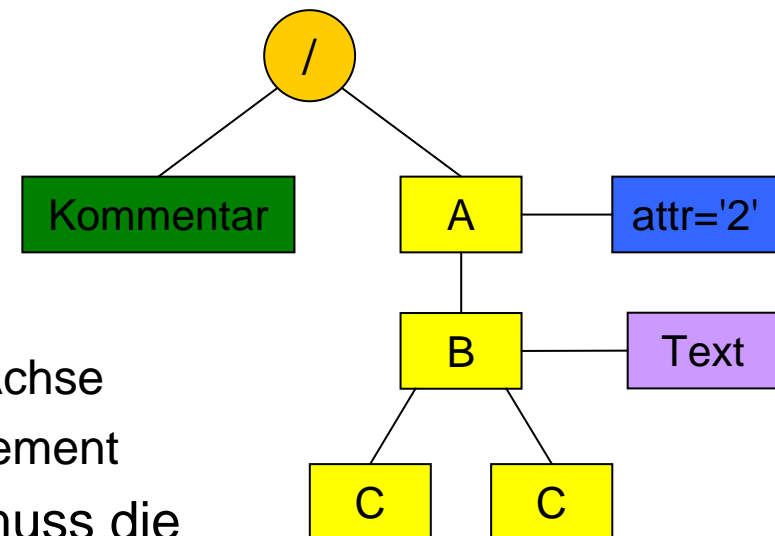
Auswahl einer Achse

- Achsenangaben werden einem Knotentest zusammen mit zwei Doppelpunkten vorangestellt
 - z.B. `parent::A`
- Achsenangaben gelten nur für den aktuellen Location-Step
 - in der Pfadangabe `ancestor::A/B` bezieht sich die *ancestor*-Achse nur auf die Auswahl der *A*-Knoten; die *B*-Knoten werden in den Kindelementen der gefundenen *A*-Knoten gesucht (bei fehlender Achsenangabe wird die *child*-Achse verwendet)
- jeder Location-Step kann eine Achsenangabe enthalten
 - z.B. `ancestor::A/following-sibling::B`



- außer *self* berücksichtigen die gerade erwähnten Achsen nur Dokument-, Element-, Text- und Kommentar-Knoten (sowie Processing-Instructions)
- Attribut- und Namespace-Knoten sind nur über die *attribute*- bzw. *namespace*-Achse erreichbar

- für nebenstehenden XML-Baum liefert `/A/node()` nur den *B*-Knoten
 - `node()` sucht hier auf der *child*-Achse von Element *A*
 - Attributknoten gehören nicht zur *child*-Achse
 - Knotentest `node()` findet nur das *B*-Element
- zur Adressierung des Attributknotens muss die *attribute*-Achse gewählt werden, z.B.
 - `/A/attribute::node()`
 - `/A/attribute::attr`
 - `/A/attribute::*`



Kurzformen für häufig verwendete Location-Steps

- in der Praxis werden einige Achsen zusammen mit bestimmten Knotentests besonders häufig benötigt
 - für fünf ausgewählte Location-Steps stellt XPath Kurzschreibweisen zur Verfügung

Kurzform	ausführliche Form
.	<code>self::node()</code>
..	<code>parent::node()</code>
//	<code>/descendant-or-self::node()/</code>
<i>name</i>	<code>child::<i>name</i></code>
@ <i>name</i>	<code>attribute::<i>name</i></code>

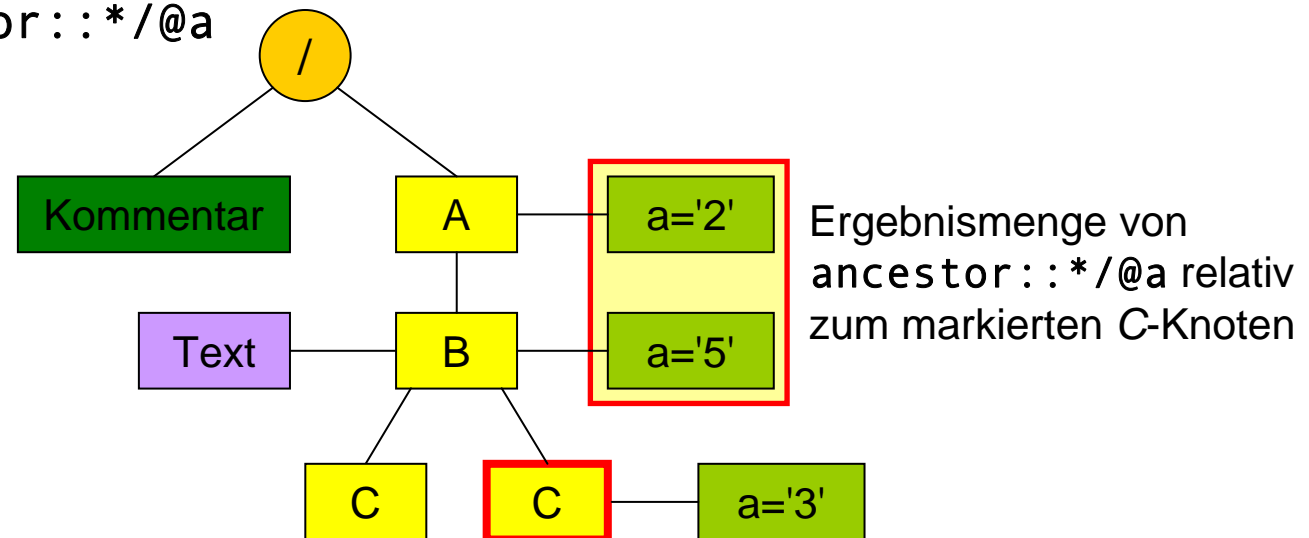
Beispiel:

`A//B/@attr` ist eine Kurzform von

`child::A/descendant-or-self::node()/child::B/attribute::attr`

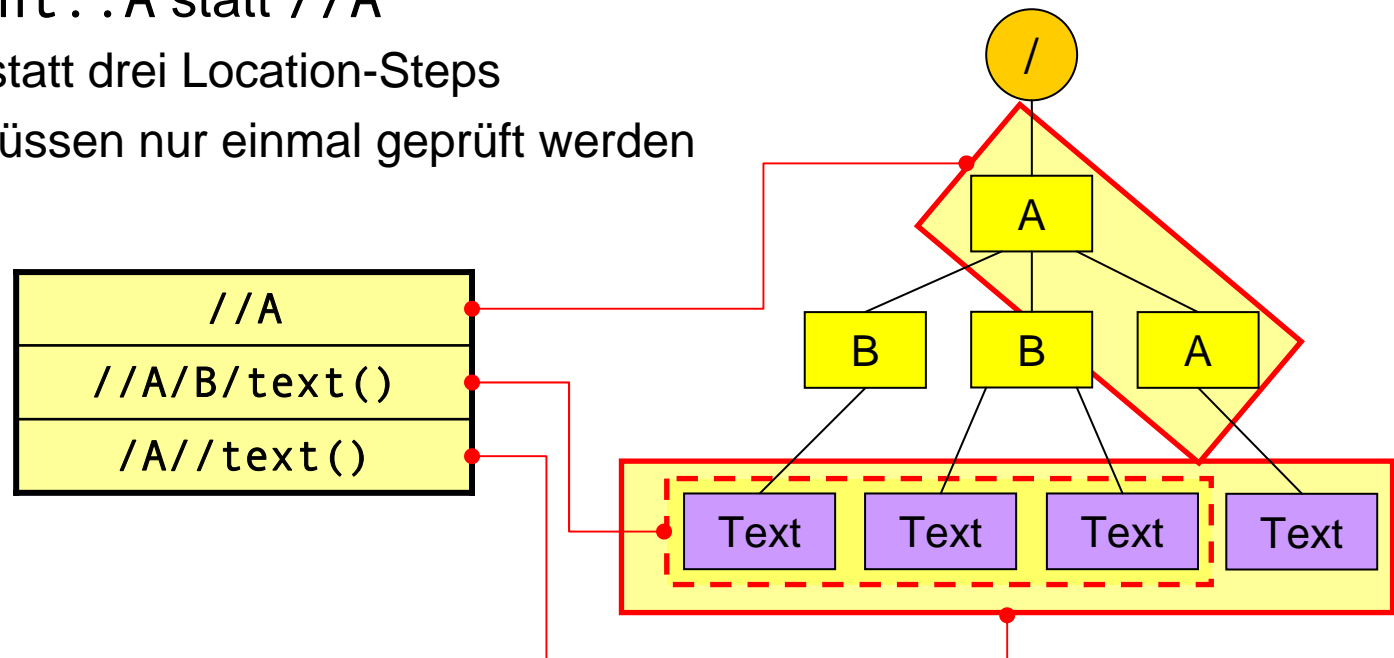
Anmerkungen zur *attribute*-Achse

- Wichtig: @a ist eine Kurzform für `attribute::a`
 - die Kurzform kann nicht direkt mit anderen Achsen kombiniert werden
 - etwas wie `ancestor::@a` ist syntaktisch falsch
 - die Langform wäre hier: `ancestor::attribute::a`
 - da jeder Location-Step nur eine Achsenangabe enthalten kann, ist der vorangehende Ausdruck nicht definiert
 - zur Auswahl von Attributen der Vorgängerelemente müssen zunächst die gewünschten Elemente ausgewählt werden; im zweiten Schritt sind dann die Attribute erreichbar
 - z.B. `ancestor::*/@a`



Rekursiver Abstieg mit //

- die Zeichenfolge // ist eine Kurzschreibweise für `/descendant-or-self::node()`
 - kann anstelle einfacher Slashes verwendet werden, um anschließende Knotentests auf alle Nachfahren anzuwenden
 - sollte sorgsam verwendet werden, da die Auswertung abhängig von der Größe des Teilbaums zeitaufwendig sein kann
 - abhängig vom XPath-Prozessor kann ein direkter Knotentest auf der *descendant*-Achse etwas effizienter sein, z.B. `/descendant::A` statt `//A`
 - nur zwei statt drei Location-Steps
 - Knoten müssen nur einmal geprüft werden



Datentypen in XPath 1.0

- XPath 1.0 unterstützt vier verschiedene Datentypen
 - **boolean** (Wahrheitswerte)
 - die beiden booleschen Konstanten lauten in XPath `true()` und `false()`
 - **number** (Gleitkommazahlen)
 - **string**
 - String-Konstanten können in einfachen oder doppelten Anführungszeichen angegeben werden, z.B. `'Hallo'` oder `"Hallo"`
 - **node-set** (Knotenmengen)
 - Knotenmengen werden üblicherweise durch Pfadangaben erzeugt

XPath-Operatoren

$a = b$	<i>a</i> gleich <i>b</i>
$a \neq b$	<i>a</i> ungleich <i>b</i>
$a < b$	<i>a</i> kleiner <i>b</i>
$a > b$	<i>a</i> größer <i>b</i>
$a \leq b$	<i>a</i> kleiner oder gleich <i>b</i>
$a \geq b$	<i>a</i> größer oder gleich <i>b</i>

$a + b$	Addition
$a - b$	Subtraktion
$a * b$	Multiplikation
$a \text{ div } b$	Gleitkomma-Division
$a \text{ mod } b$	Gleitkomma-Modulo
$a \text{ or } b$	logisches ODER
$a \text{ and } b$	logisches UND
$a b$	Vereinigung von Knotenmengen

Ausgewählte XPath-Funktionen

Funktion	Returntyp	Beschreibung
<code>concat(<i>s1</i>, ..., <i>sn</i>)</code>	string	verkettet die Strings <i>s1</i> bis <i>sn</i> zu einem neuen String
<code>contains(<i>s1</i>, <i>s2</i>)</code>	boolean	prüft, ob String <i>s1</i> den String <i>s2</i> enthält
<code>count(<i>K</i>)</code>	number	Anzahl der Knoten in Menge <i>K</i>
<code>name(<i>K</i>)</code>	string	Name des ersten Knotens in Menge <i>K</i>
<code>normalize-space(<i>s</i>)</code>	string	entfernt Whitespace am Anfang und Ende von <i>s</i> und ersetzt die restlichen Whitespace-Folgen durch einfache Leerzeichen
<code>not(<i>b</i>)</code>	boolean	boolsche Negation
<code>round(<i>x</i>)</code>	number	rundet <i>x</i>
<code>starts-with(<i>s1</i>, <i>s2</i>)</code>	boolean	prüft, ob String <i>s1</i> mit <i>s2</i> beginnt
<code>string-length(<i>s</i>)</code>	number	Länge von String <i>s</i>
<code>substring(<i>s</i>, <i>n</i>, <i>l</i>)</code>	string	Teilstring von <i>s</i> , beginnend beim <i>n</i> -ten Zeichen und Länge <i>l</i>
<code>substring-after(<i>s1</i>, <i>s2</i>)</code>	string	sucht <i>s2</i> in <i>s1</i> und liefert alle Zeichen hinter dem ersten Treffer zurück (liefert Leerstring zurück falls <i>s2</i> in <i>s1</i> nicht vorkommt)
<code>substring-before(<i>s1</i>, <i>s2</i>)</code>	string	sucht <i>s2</i> in <i>s1</i> und liefert alle Zeichen vor dem ersten Treffer zurück (liefert Leerstring zurück falls <i>s2</i> in <i>s1</i> nicht vorkommt)
<code>sum(<i>K</i>)</code>	number	Addiert die Werte der Knoten in Menge <i>K</i>
<code>translate(<i>s1</i>, <i>s2</i>, <i>s3</i>)</code>	string	sucht in <i>s1</i> der Reihe nach die Zeichen von <i>s2</i> und ersetzt in <i>s1</i> das <i>i</i> -te Zeichen von <i>s2</i> durch das <i>i</i> -te Zeichen von <i>s3</i>

- neben Pfadangaben sind auch Funktionsaufrufe sowie Ausdrücke mit mathematischen und logischen Operatoren gültige XPath-Ausdrücke
- Beispiele:
 - `1+2*3`
 - `concat('XML', 'und', 'XSLT')`
 - `contains('Blumentopferde', 'pferd')`
 - `count(//A)`
 - `translate('Hello', 'eo', 'ae')`
 - `(1+2*3 > (1+2)*3) or starts-with('Hallo', 'ha')`
 - `not(count(ancestor::A) > 5)`

Automatische Typumwandlung

- Falls ein Operand oder ein Funktionsparameter nicht den erwarteten Typ besitzt, führt XPath eine implizite Typkonvertierung durch, z.B.

"5.5" + true() * 2 → 7.5

von \ nach	boolean	number	string	node-set
boolean		false → 0 true → 1	false → 'false' true → 'true'	<i>nicht erlaubt</i>
number	0 → false sonst true		Dezimalzahl als String	<i>nicht erlaubt</i>
string	' ' → false sonst true	interpretiert den String als Zahl		<i>nicht erlaubt</i>
node-set	{ } → false sonst true	interpretiert den aus erstem Knoten gebildeten String als Zahl	hängt die Texte descendant::text() des ersten Knotens aneinander	

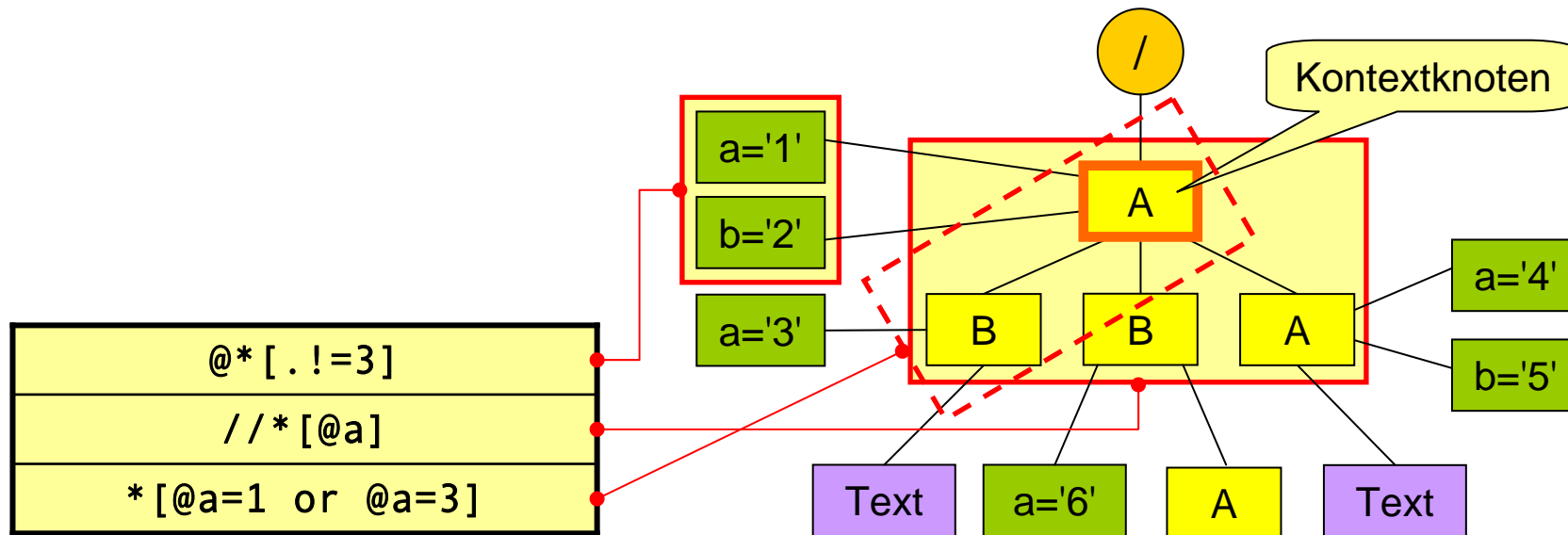
- **Prädikate** sind optionale Bestandteile eines Location-Steps und grenzen Knotenmengen durch Boolesche Ausdrücke weiter ein
 - nur Knoten, auf die der angegebene Ausdruck zutrifft, werden in die Ergebnismenge aufgenommen
- Prädikate werden in eckigen Klammern hinter einem Knotentest angegeben
 - `A[.='Hallo']`
liefert alle *A*-Kindknoten, die den Text "Hallo" enthalten
 - `A[@attrib = 'Hallo']`
liefert alle *A*-Kindknoten, die einen Attributknoten *attrib* mit Wert "Hallo" enthalten
 - `A[B]`
liefert alle *A*-Kindknoten, die mindestens einen *B*-Kindknoten enthalten
- relative Pfadangaben in Prädikaten beziehen sich immer auf die durch den vorangehenden Pfadausdruck ausgewählte Knotenmenge

Prädikate: Beispiele

```

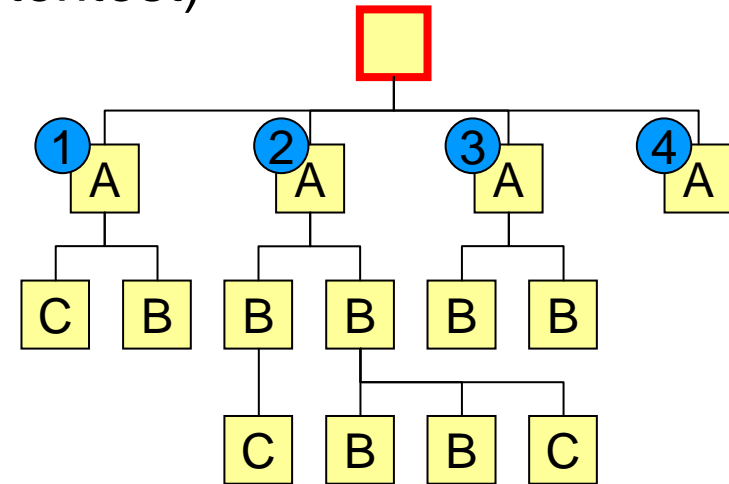
<?xml version="1.0"?>
<A a="5" b="8">
  <B a="20">Guten</B>
  <B>
    <A a="1">Morgen</A>
    <C>Heute</C>
  </B>
  <B a="50">Tag</B>
  <A>Abend</A>
</A>
  
```

//A[not(@a)]
/A/B[@a < 50]
//B[.='Tag']



Knotennummerierung und Positionstests

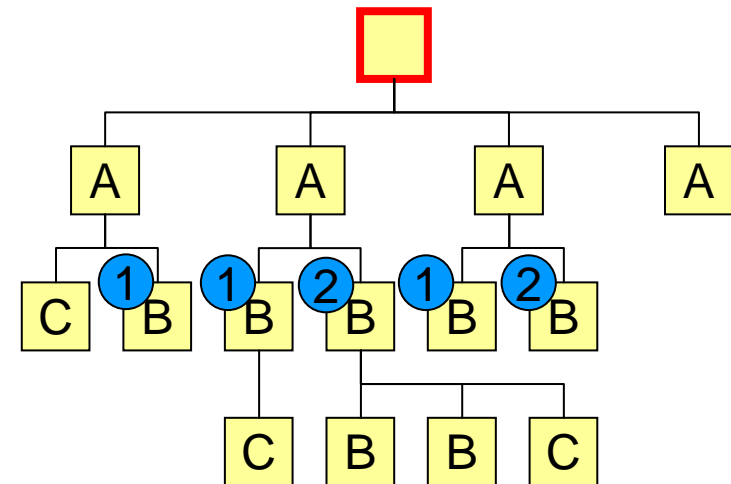
- die von einem Location-Step ausgewählten Knoten werden implizit nummeriert (relativ zu Achse und Knotentest)
 - Nummerierung beginnt immer bei 1
- die XPath-Funktion `position()` liefert die Positionsnummer des aktuellen Kontextknotens
 - der Ausdruck `A[position()=1]` liefert alle A-Knoten mit Positionsnummer 1
- Prädikate der Form `position()=n`, wobei n eine Zahl (Typ *number*) ist, werden **Positionstests** genannt
- Prädikate, die nur aus einer Zahl bestehen, werden immer als Positionstests interpretiert
 - `A[1]` ist eine Kurzform von `A[position()=1]`



Nummerierung der Ergebnisknoten beim Location-Step A relativ zum rot markierten Kontextknoten

Knotennummerierung und Positionstests

- die Knotennummerierung bezieht sich immer auf die gewählte Achse
 - im Fall der *child*-Achse werden die Kinder des Kontextknotens in Dokumentreihenfolge durchnummeriert
 - wenn ein Knotentest Kindknoten von unterschiedlichen Eltern zurückliefert, beginnt die Nummerierung der Kinder für jeden Elternknoten erneut bei 1
- für den abgebildeten Baum liefert die Pfadangabe `A/B[1]` eine Menge mit drei *B*-Knoten zurück
- zur gezielten Auswahl eines *B*-Knotens muss zuvor der gewünschte *A*-Knoten selektiert werden, z.B. liefert `A[3]/B[1]` den ersten *B*-Kindknoten des dritten *A*-Elements

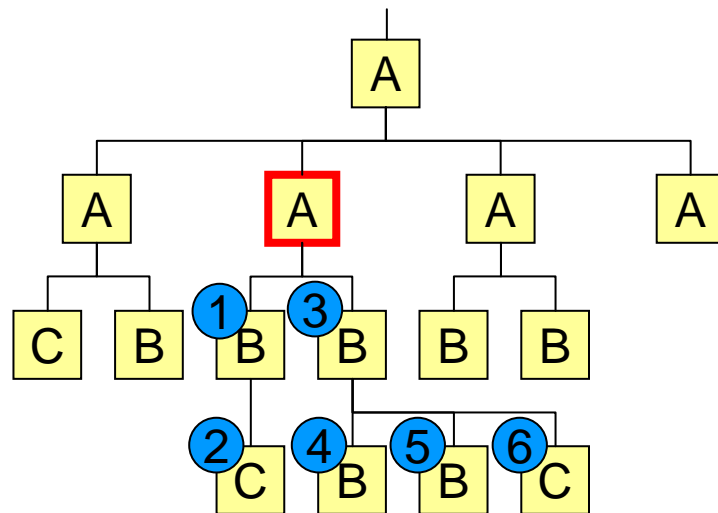


Knotennummerierung und Positionstests

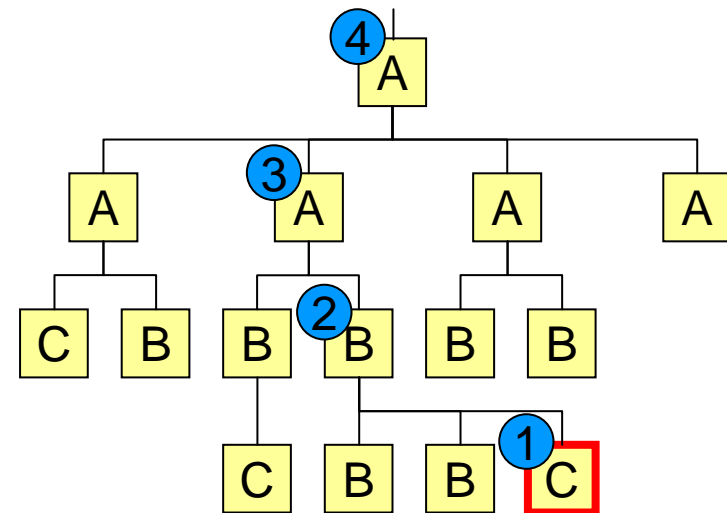
- die Knotennummerierung ist abhängig von
 - dem/den jeweiligen Kontextknoten
 - der gewählten Achse
 - dem Knotentest
- durch Änderung eines dieser Parameter ändert sich ggf. die Nummerierung

Knotennummerierung und Positionstests

- bei allen Achsen, die Dokumentbereiche hinter dem Kontextknoten beschreiben, wächst die Nummerierung in Richtung des Dokumentendes
- bei allen Achsen, die Dokumentbereiche vor dem Kontextknoten beschreiben, wächst die Nummerierung in Richtung des Dokumentanfangs



descendant::*



ancestor-or-self::*

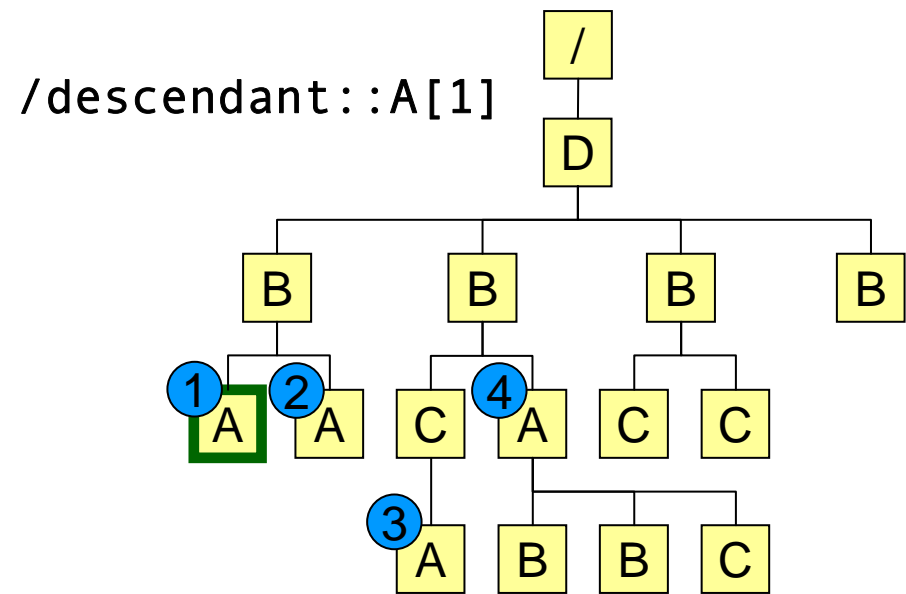
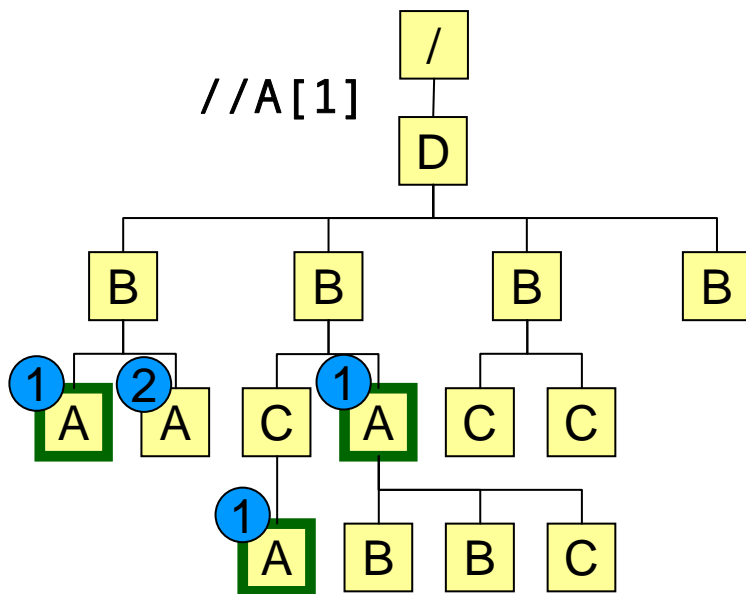
Knotennummerierung und Positionstests

```
<adressbuch>
  <adresse>
    <name geschlecht="m">
      <vorname>Jim</vorname>
      <nachname>Panse</nachname>
    </name>
    <strasse>Primatenring 16</strasse>
    <ort plz="12345">Affenbach</ort>
  </adresse>
  <adresse>
    <name geschlecht="m">
      <vorname>Bernhard</vorname>
      <nachname>Diener</nachname>
    </name>
    <strasse>Dackelgasse 9</strasse>
    <ort plz="54321">Köteringen</ort>
  </adresse>
  <adresse>
    <name geschlecht="w">
      <vorname>Wanda</vorname>
      <nachname>Düne</nachname>
    </name>
    <strasse>Küstenstraße 1</strasse>
    <ort plz="77777">Süddeich</ort>
  </adresse>
</adressbuch>
```

/*/adresse[2]
/*/descendant::vorname[2]
//vorname[2]

Unterschied //A[1] und /descendant::A[1]

- //A und /descendant::A sammeln dieselben Knoten ein
- //A[1] und /descendant::A[1] beschreiben hingegen unterschiedliche Knotenmengen. Warum?
- die Pfadangabe //A[1] besteht aus drei Location-Steps
 - Langform: /descendant-or-self::node()/A[1]
 - das Prädikat bezieht sich auf einen Knotentest der *child*-Achse
 - bei /descendant::A[1] bezieht sich das Prädikat auf einen Knotentest der *descendant*-Achse



Knotennummerierung und Positionstests

- soll sich ein Positionstest nicht auf die Position eines Knotens im XML-Baum sondern auf die Position in der Ergebnismenge beziehen, muss die Pfadangabe geklammert werden:
(pfadangabe) [position() = n] oder kurz *(pfadangabe) [n]*
- `//A[1]` liefert alle Knoten des XML-Dokuments, die erstes A-Kinderelement ihres Elternelements sind
- `(//A)[1]` liefert das erste im XML-Dokument vorkommende A-Element

XPath vs. XQuery

- mit XPath kann man:
 - in XML-Bäumen navigieren
 - einzelne Knoten und Knotenmengen aus XML-Bäumen auswählen
- mit XPath kann man nicht:
 - Daten sortieren
 - Daten neu gruppieren
 - Variablen und Funktionen definieren
 - neue XML-Knoten erzeugen
 - XML-Dokumente ändern
- XQuery ist eine deklarative Programmiersprache mit XPath 2.0 als Untermenge
 - ermöglicht komplexe Abfragen von Daten aus XML-Dokumenten
 - bietet Konstrukte zur Erzeugung neuer XML-Bestandteile
 - kann für bestimmte Aufgaben als Alternative zu XSLT dienen
 - Ändern von XML-Daten nur mit *XQuery Update Facility* möglich

- BaseX, XML-Datenbank mit GUI
 - <http://www.inf.uni-konstanz.de/dbis/basex>
 - Java
- Saxon-HE von Michael Kay
 - <http://saxon.sourceforge.net>
 - Java, .NET, kostenlose Home Edition (Open Source)
- AltovaXML
 - <http://www.altova.com/de/altovaxml.html>
 - proprietär, nur für Windows
- XQilla
 - <http://xqilla.sourceforge.net>
 - C++
- Zorba XQuery-Prozessor
 - <http://www.zorba-xquery.com>
 - C++

XQuery: Datentypen und -strukturen

- XQuery (und XPath 2.0) verwendet das Typensystem von XML Schema
 - **atomare (einfache elementare) Typen:**
xs:boolean, xs:integer, xs:double, xs:string, xs:date, ...
 - **Knotentypen:**
Element, Attribut, Text, Kommentar, Verarbeitungsanweisung, Namensraum
- als einzige eingebaute, komplexe Datenstruktur gibt es die **Sequenz**
 - geordnete Liste von Objekten beliebigen Typs
 - Sequenzen dürfen aus beliebig vielen Komponenten bestehen
 - jede Komponente darf einen anderen Typ haben
 - die Reihenfolge der Komponenten ist signifikant
- Sequenzen können u.a. durch Auflistung der Komponenten oder als Ergebnis von Pfadausdrücken erzeugt werden
 - (1, 2, 'Hallo', @wert)
 - //person/vorname

XQuery: neues Typensystem

- mit der Einführung des neuen Typensystems findet eine strengere Prüfung der Typen statt
 - im Objekte mit atomarem Typ werden nicht mehr automatisch in andere Typen konvertiert
 - der Ausdruck `1+'2'+true()` ist in XQuery nicht erlaubt
 - Typkonvertierungen müssen explizit notiert werden
 - z.B. `1+xs:integer('2')+xs:integer(true())`
 - Knoten ohne zugewiesenen Schema-Typ bekommen den Typ *xs:untyped* und werden bei der Auswertung ggf. automatisch konvertiert
 - im Ausdruck `1+@wert` wird das *wert*-Attribut des Kontextknotens in eine Zahl (*xs:decimal*) konvertiert
 - im Ausdruck `concat('Hallo', @wert)` wird das *wert*-Attribut des Kontextknotens in einen String (*xs:string*) konvertiert

- in XPath 1.0 erzeugen Pfadausdrücke Knotenmengen, die keinen, einen oder mehrere Knoten enthalten können
- in XQuery sind Sequenzen endliche Listen, die beliebige Objekte mit beliebigen Typen enthalten dürfen
- Sequenzen können explizit durch Aufzählung der Komponenten in runden Klammern erzeugt werden
 - `(1, 2, 'Hallo', 3.0)`
 - `(1 to 10)` ist eine Kurzform für `(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)`
- Sequenzen dürfen auch Sequenzen enthalten
 - geschachtelte Sequenzen werden immer aufgelöst, so dass eine eindimensionale Liste entsteht („innere Klammern werden entfernt“)
 - `(1, (2, (3, 4)), ('Hallo', 'Welt'))` wird zu `(1, 2, 3, 4, 'Hallo', 'Welt')`
 - `(//bundesland[@typ='stadtstaat'], //name)` kombiniert die Ergebnissequenzen der beiden Pfadausdrücke

XQuery: Variablen definieren mit `let`

- in XPath kann zwar auf Variablen zugegriffen werden, es gibt aber keine Möglichkeit, Variablen zu definieren
- XQuery stellt die Anweisung **let** zur Variablendefinition bereit
 - Variablen dürfen beliebige Objekte oder Sequenzen zugewiesen werden
 - Variablen können nachträglich nicht mehr geändert werden
 - Variablen können optional typisiert werden
- `let $var := ausdruck`
 - `let $zahl := 5`
 - `let $namen := //person/name`
 - `let $seq := (1,2,6,3,'Hallo')`
- `let $var as typ := ausdruck`
 - `let $zahl as xs:double := 5`
 - `let $namen as xs:string* := //person/name`

XQuery: Ergebnis angeben mit return

- jeder XPath-Ausdruck produziert ein Ergebnis, das anschließend weiterverarbeitet werden kann
 - das Ergebnis kann aus einem Leerstring oder einer leeren Sequenz bestehen
 - es gibt keinen XPath-Ausdruck vom Typ „void“
- in XQuery gilt im Prinzip das gleiche:
jeder XQuery-Ausdruck muss ein Ergebnis produzieren
- da eine Variablendefinition kein Ergebnis erzeugt, darf sie nicht isoliert verwendet werden
- das gewünschte Query-Ergebnis muss mit einer folgenden **return**-Anweisung festgelegt werden

```
let $personen := //personen[vorname='Maria']  
return $personen/nachname
```

```
let $vornamen := //personen/vorname  
return for $vn in $vornamen return concat('Hallo', $vn)
```

XQuery: Definition mehrerer Variablen mit gleichem Namen

- in XQuery gibt es keinen Zuweisungsoperator, so dass Variablen nach ihrer Definition nicht mehr geändert werden können
- trotzdem ist folgendes Query erlaubt:

```
let $n := 1
let $n := $n+1
return $n
```

– Ergebnis des Querys ist 2

– wie kann das sein, wenn Variablen nicht geändert werden können?

- jede Variablendefinition erzeugt eine neue Variable
 - gleichnamige Variablen verdecken die jeweils vorangehende Definition
 - auf der rechten Seite von `:=` ist die vorangehende Definition noch sichtbar, deshalb wird dem zweiten `n` der Wert des ersten zugewiesen
 - in der `return`-Anweisung ist nur noch die zweite Variable sichtbar, die erste existiert aber noch
- das Query ist identisch zu folgendem:

```
let $n := 1
let $m := $n+1
return $m
```

– hier kann im `return`-Statement sowohl auf `n` als auch auf `m` zugegriffen werden

XQuery: Bedingte Ausdrücke mit `if-then-else`

- XQuery erlaubt fallunterscheidende Ausdrücke mit Hilfe einer `if-then-else`-Konstruktion:
- Syntax: `if (<bedingung>) then <ausdruck1> else <ausdruck2>`
 - da XQuery-Ausdrücke immer ein Ergebnis produzieren müssen, ist der `else`-Teil verpflichtend, kann also nicht weggelassen werden
 - ist vergleichbar mit dem ternären Operator `?:` in C/C++/Java
 - Java: `(b > c) ? b : c;`
 - XPath: `if ($b > $c) then $b else $c`
 - die Datentypen vom `then`- und `else`-Teil müssen nicht identisch sein
 - `if (adressen) then adressen/adresse else "Hallo Welt"` ist erlaubt
 - der `then`-Ausdruck wird nur ausgewertet, wenn die Bedingung wahr ist und der `else`-Ausdruck nur, wenn die Bedingung falsch ist
 - `if ($a > 0) then 1 div $a else $a` produziert also keinen Fehler falls `$a=0`

XQuery: Iterieren über Sequenzen mit `for - return`

- um einen Ausdruck auf jede Komponente einer Sequenz anzuwenden wird **for-return** verwendet
 - das Resultat ist eine neue Sequenz mit den Ergebnissen der einzelnen Iterationsschritte
- Syntax: `for $var in <sequenz> return <ausdruck>`
- Beispiele:
 - `for $i in (1 to 10) return $i*0.1`
erzeugt die Sequenz (0.1, 0.2, .0.3, ..., 1.0)
 - `for $i in //name return concat('Hallo ', $i)`
iteriert über alle *name*-Elemente des aktuellen Dokuments und erzeugt daraus eine String-Sequenz der Form ('Hallo Jim', 'Hallo Anna', ...)
- *for-return* darf überall dort verwendet werden, wo Sequenzausdrücke erlaubt sind, z.B.

```
let $grüße := if (//name) then
  for $i in //name return concat('Hallo ', $i)
else
  'niemand zum Begrüßen da'
return $grüße
```

XQuery: FLWOR – erweiterte **for-return**-Anweisung

- XQuery erlaubt zwischen *for* und *return* zusätzlich die folgenden optionalen Angaben:
 - beliebig viele **let**-Anweisungen zur Variablendefinition
 - eine **where**-Anweisung der Form *where bedingung* zum Filtern von Sequenzkomponenten
 - eine **order by**-Anweisung der Form *order by ausdruck richtung* zum Ändern der Iterationsreihenfolge

XQuery: FLWOR – erweiterte for-return-Anweisung

- **let:** Definition von Variablen bei jedem Iterationsschritt

```
for $n in (4,7,1,1)
let $m := 2*$n
return $m
```

- die Variable m wird bei jedem Iterationsschritt neu definiert
- das Query liefert die Sequenz (8, 14, 2, 2)

- auch hier gilt, dass eine Variable mit gleichem Namen die vorangehende verdeckt

```
let $m := 5
for $n in (4,7,1,1)
let $m := $n*$m
return $m
```

- im Ausdruck $\$m*\n bezeichnet $\$m$ die erste Variable m (mit Wert 5), im *return*-Statement wird die zweite verwendet
- liefert die Sequenz (20, 35, 5, 5)

- **where:** Iteration beschränken
 - es werden nur Sequenzelemente berücksichtigt, die die angegebene Bedingung erfüllen

```
for $person in //person
let $vname := $person/name/vorname
where starts-with($vname, 'M')
return $person
```

- hat den gleichen Effekt wie ein Prädikat
 - im *where*-Ausdruck können zusätzlich "innere" Variablen verwendet werden
- **order by:** Verarbeitungsreihenfolge ändern
 - vor der Iteration wird für jede Komponente der Sortierausdruck ausgewertet und die Sequenz anhand dieser Werte sortiert
 - anschließend wird über die sortierte Sequenz iteriert

```
for $person in //person
order by $person/name/nachname, $person/name/vorname
return $person
```

XQuery: FLWOR-Ausdrücke

- die optionalen Bestandteile von *for-return* müssen immer in der Reihenfolge **let – where – order by** angegeben werden
- ein kompletter *for-return*-Ausdruck hat also die Form **for – let – where – order by – return**
- die Anweisung wird deshalb auch **FLWOR-Ausdruck** genannt, unabhängig davon, ob alle Bestandteile verwendet werden
 - FLWOR wird wie engl. *flower* ausgesprochen

```
for $person in //person
let $vorname := $person/name/vorname
let $nachname := $person/name/nachname
where $person/wohnort = 'Osnabrück'
order by $nachname, $vorname
return concat('Hallo ', $vorname, ' ', $nachname, '
')
```

- trotz der verschiedenen Schlüsselwörter handelt es sich hier um nur einen Ausdruck
- liefert als Resultat hier eine Sequenz mit String-Objekten

XQuery: FLWOR-Ausdrücke

- ein FLWOR-Ausdruck darf mehrere *for*-Anweisungen enthalten
 - werden wie ineinander geschachtelte *for*-Schleifen behandelt

```
for $i in ('a','e')
for $j in ('b','k','t')
return concat($i, $j)
```

oder

```
for $i in ('a','e'),
    $j in ('b','k','t')
return concat($i, $j)
```

- liefert die Sequenz ('ab', 'ak', 'at', 'eb', 'ek', 'et')
- zwischen zwei *for*-Anweisungen dürfen beliebig viele *let*-Anweisungen stehen
 - *order by*, *where* und *return* sind nur hinter dem letzten *for* erlaubt

```
for $i in ('a','e')
let $r := concat($i,'r')
for $j in ('b','k','e')
let $s := concat($r,$j)
where contains($s, 'e')
return $s
```

- liefert die Sequenz ('are', 'erb', 'erk', 'ere')

XQuery: Positionsvariablen in FLWOR-Ausdrücken

- zur Abfrage der Position einer Sequenzkomponente werden in FLWOR-Ausdrücken **Positionsvariablen** verwendet
 - zusätzliche Variable, die in der *for*-Anweisung hinter der Iterationsvariablen mit dem Präfix **at** angegeben wird

```
for $person at $i in //personen[vorname='Maria']  
return concat($i, ' Maria ', $person/nachname, '&#10;')
```

- eine Positionsvariable enthält einen Integer-Wert, der die Position der aktuellen Komponente in der angegebenen Sequenz enthält
- **order by** ändert die Zuordnung von Position zu Sequenzkomponente nicht!

```
for $z at $i in (5,3,1,2,4)  
order by $z  
return ('Pos.', $i, ':', $z)
```



```
Pos. 3 : 1  
Pos. 4 : 2  
Pos. 2 : 3  
Pos. 5 : 4  
Pos. 1 : 5
```

XQuery: Positionsvariablen in FLWOR-Ausdrücken

- sollen die Komponenten der sortierten Sequenz nummeriert ausgegeben werden, benötigt man einen zweiten FLWOR-Ausdruck

```
(: Sequenz sortieren und in Variable ablegen:)  
let $sortiert :=  
  for $z1 in (5,3,1,2,4)  
  order by $z1  
  return $z1  
  
(: sortierte Sequenz nummeriert ausgeben :)  
for $z2 at $i in $sortiert  
return ('Pos.', $i, ':', $z2, '
')
```



```
Pos. 1 : 1  
Pos. 2 : 2  
Pos. 3 : 3  
Pos. 4 : 4  
Pos. 5 : 5
```

– dasselbe ohne zusätzliche Variable:

```
for $z2 at $i in  
  (: sortierte Sequenz erzeugen :)  
  for $z1 in (5,3,1,2,4)  
  order by $z1  
  return $z1  
return ('Pos.', $i, ':', $z2, '
')
```



```
Pos. 1 : 1  
Pos. 2 : 2  
Pos. 3 : 3  
Pos. 4 : 4  
Pos. 5 : 5
```


XQuery: Element-Konstrukturen

- XQuery-Skripte können literale XML-Elemente enthalten
- der Inhalt von Attributen und Elementrümpfen wird als literaler Text interpretiert und nicht weiter ausgewertet
- XQuery-Anweisungen innerhalb von Attributen oder Elementen müssen mit {...} geklammert werden

```
let $personen :=
  <personen>
    <person geschlecht="m">Jim Panse</person>
    <person geschlecht="w">Anna Konda</person>
  </personen>
return
  <personen-neu>{
    for $person at $pos in $personen/person
    return
      <person pos="{ $pos }">
        { $person/@geschlecht }
        <vname>{substring-before($person, ' ')}</vname>
        <nname>{substring-after($person, ' ')}</nname>
      </person>
    }</personen-neu>
```

- XQuery ist von Haus aus eine reine Abfragesprache ohne Seiteneffekte
 - d.h. es gibt keine Möglichkeit, Daten in XML-Dokumenten zu ändern
- bei Verwendung von XML als Datenbankformat ist es allerdings erforderlich, auch Daten ändern, einfügen oder löschen zu können
- daher gibt es die XQuery-Erweiterung **XQuery Update Facility**, kurz: **XUF**
 - separate W3C-Spezifikation neuer XQuery-Sprachbestandteile
 - wird inzwischen von nahezu allen XQuery-Prozessoren unterstützt
- XUF stellt fünf neue XQuery-Anweisungen zur Änderung von Daten zur Verfügung
 - **insert, delete, replace, rename, copy**
 - die Anweisungen produzieren im Gegensatz zu allen anderen XQuery-Ausdrücken keinen Rückgabewert, sondern ändern stattdessen das XML-Dokument

XQuery Update Facility: insert

- `insert node(s) knoten into ziel`
 - fügt den/die angegebenen Knoten als Kind in den Zielknoten ein
 - Position innerhalb des Zielknotens ist implementationsabhängig
- `insert node(s) knoten as first|last into ziel`
 - fügt den/die angegebenen Knoten als erstes/letztes Kind in den Zielknoten ein
- `insert node(s) knoten after|before ziel`
 - fügt den/die angegebenen Knoten als Geschwister vor/hinter dem Zielknoten ein

```
insert node <plz>49080</plz>  
as last into //adresse[1]
```

```
insert node <plz>49080</plz>  
after //adresse[1]/ort
```

```
<adressen>  
  <adresse>  
    <name>Hans Meier</name>  
    <ort>Osnabrück</ort>  
    <plz>49080</plz>  
  </adresse>  
  <adresse>...</adresse>  
  ...  
</adressen>
```

XQuery Update Facility: delete und rename

- **delete node(s) *knoten***

- entfernt den/die angegebenen Knoten aus dem XML-Dokument

```
delete node //adresse[1]/plz
```

```
delete nodes //adresse/plz
```

```
<adressen>
  <adresse>
    <name>Hans Meier</name>
    <ort>Osnabrück</ort>
    <plz>49080</plz>
  </adresse>
  <adresse>...</adresse>
  ...
</adressen>
```

- **rename node *knoten* as *neuer_name***

- weist dem angegebenen Knoten einen neuen Namen zu

```
for $a in //adresse
return
  rename node $a as 'anschrift'
```

```
<adressen>
  <anschrift>
    <name>Hans Meier</name>
    <ort>Osnabrück</ort>
    <plz>49080</plz>
  </anschrift>
  <anschrift>...</anschrift>
  ...
</adressen>
```

XQuery Update Facility: replace

- **replace node** *knoten* with *ersatz*
 - ersetzt den angegebenen Knoten durch einen anderen

```
replace node //adresse[1]/plz  
with <PLZ>49078</PLZ>
```

```
<adressen>  
  <adresse>  
    <name>Hans Meier</name>  
    <ort>Osnabrück</ort>  
    <PLZ>49078</PLZ>  
  </adresse>  
  <adresse>...</adresse>  
  ...  
</adressen>
```

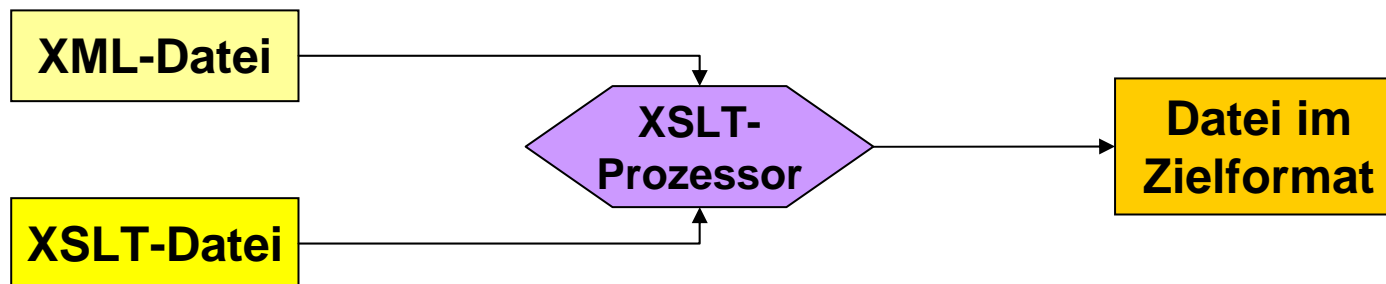
- **replace value of node** *knoten* with *ersatz*
 - ersetzt den Inhalt des angegebenen Knotens

```
replace value of  
node //adresse[1]/plz  
with 49090
```

```
<adressen>  
  <anschrift>  
    <name>Hans Meier</name>  
    <ort>Osnabrück</ort>  
    <plz>49090</plz>  
  </anschrift>  
  <anschrift>...</anschrift>  
  ...  
</adressen>
```

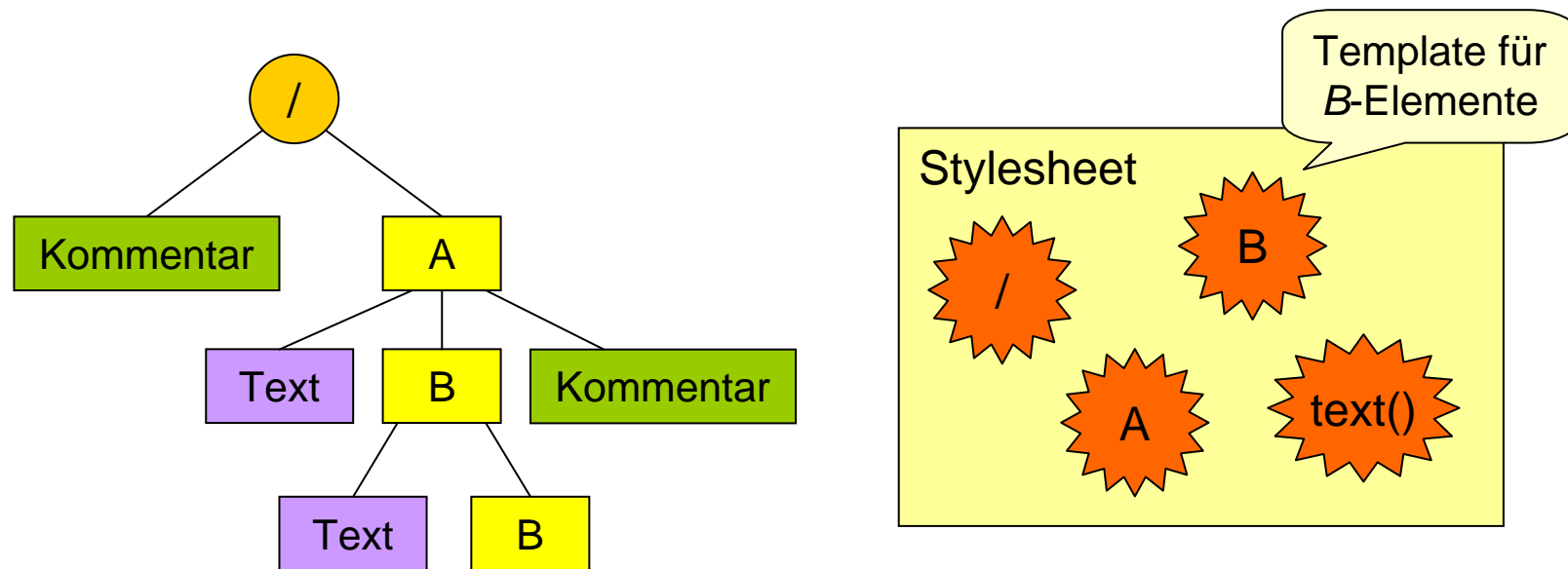
XSLT: Extensible Stylesheet Language Transformations

- XSLT ist eine deklarative Programmiersprache im XML-Format, mit der Transformationen von XML-Dateien in andere Formate beschrieben werden können
- XSLT-Stylesheets werden von einem XSLT-Prozessor ausgeführt
 - eine kleine Auswahl kostenloser Prozessoren:
 - *Saxon* von Michael Kay (<http://saxon.sourceforge.net>)
 - *Xalan* von der Apache Group (<http://xml.apache.org/xalan-j>)
 - *libxslt* des Gnome Projekts (<http://xmlsoft.org/XSLT>)
 - *AltovaXML for Windows* (<http://www.altova.com/de/altovaxml.html>)
 - alle gängigen Web-Browser (Firefox, Opera, Chrome, IE, Safari, ...)



Grundideen von XSLT

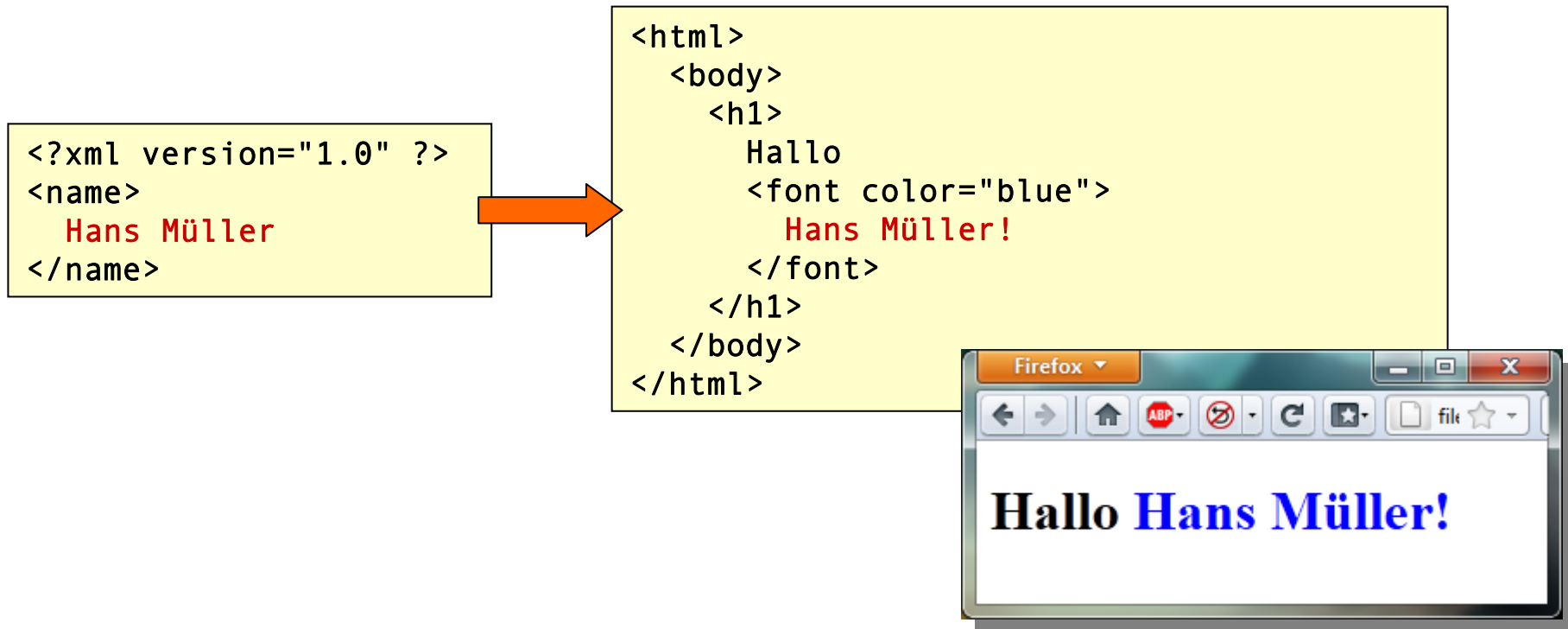
- XSLT-Programme, auch XSLT-**Stylesheets** genannt, erzeugen aus einem XML-Dokument ein neues Dokument
 - das XML-Ausgangsdokument wird dabei nicht verändert
- XSLT-Stylesheets enthalten **Templates**, die beschreiben, was der XSLT-Prozessor mit einzelnen Knoten des XML-Dokuments machen soll



- für jeden Knoten des XML-Dokuments muss es ein passendes Template geben
 - XSLT definiert Standard-Templates für jeden Knotentyp
 - nur vom Standardverhalten abweichende Templates müssen implementiert werden
- die Reihenfolge, in der die Templates aufgerufen werden, wird normalerweise nicht im Stylesheet festgelegt
 - die Verarbeitung beginnt immer mit der Dokumentwurzel
 - das zu verarbeitende XML-Dokument bestimmt, welches Template wann verwendet wird (dokumentgetriebene Verarbeitung)
 - das Stylesheet legt fest, wie anschließend die Knoten des XML-Baums durchwandert werden
 - Standardverhalten: Knoten werden in Dokumentreihenfolge verarbeitet
 - abhängig vom aktuellen Knoten wird ein passendes Template ausgewählt und ausgeführt

XSLT-Beispiel

- Gegeben ist eine XML-Datei, die in eine HTML-Datei transformiert werden soll.
 - die XML-Datei enthält nur ein Element *name* mit dem ein Begrüßungstext festgelegt wird



XSLT-Beispiel

- das zugehörige XSLT-Stylesheet, das die Transformation beschreibt, sieht wie folgt aus:

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body>
        <h1>
          Hallo
          <font color="blue">
            <xsl:value-of select="name"/>!
          </font>
        </h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- jede XSLT-Datei beginnt mit der üblichen XML-Deklaration
- danach folgt das Wurzelement (immer *xsl:stylesheet*) mit Versionsnummer und Angabe zum Namensraum *xsl*
- das Format der Zieldatei wird mit dem Element *xsl:output* festgelegt
 - mögliche Formate: xml, html, text

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body>
        <h1>
          Hallo
          <font color="blue">
            <xsl:value-of select="name"/>!
          </font>
        </h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- die Beschreibung der eigentlichen Transformation folgt im Anschluss in Form eines Templates
 - das Attribut **match** enthält einen XPath-Ausdruck, der festlegt, für welche Knoten des XML-Dokuments das Template gelten soll
 - hier handelt es sich also um ein Template für den Dokumentknoten
 - relative XPath-Ausdrücke innerhalb des Template-Rumpfs beziehen sich auf den ausgewählten XML-Knoten

```
<?xml version="1.0" ?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html>
      <body>
        <h1>
          Hallo
          <font color="blue">
            <xsl:value-of select="name"/>!
          </font>
        </h1>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

- der Rumpf des Templates enthält das, was in der Zieldatei bei Verarbeitung des ausgewählten Knotens eingefügt werden soll
 - Text und XML-Elemente, die nicht zum XSL-Namensraum gehören, werden unverändert in das Ausgabedokument kopiert
 - `<xsl:value-of select="name"/>` konvertiert das *name*-Element in einen String, d.h. der Text im Elementrumpf wird ausgegeben
 - das Attribut **select** enthält einen XPath-Ausdruck relativ zum aktuellen Kontextknoten (hier also "/")

- **Aufruf im Web-Browser**

- die meisten Web-Browser besitzen einen integrierten XSLT-Prozessor
- in der zu transformierenden XML-Datei muss hinter der XML-Deklaration folgende Zeile hinzugefügt werden:

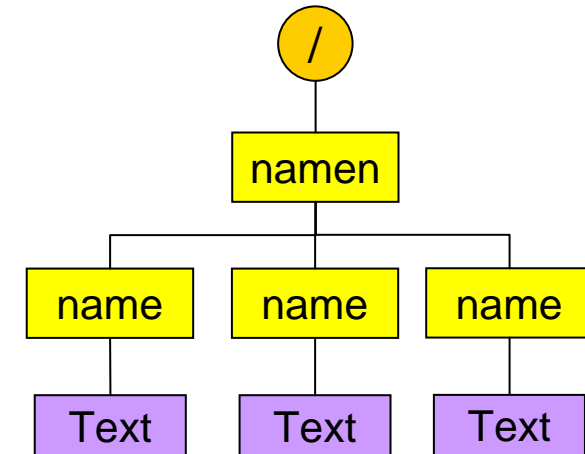
```
<?xml-stylesheet type="text/xsl" href="stylesheet.xsl"?>
```
- beim Öffnen der XML-Datei im Browser wird sie mit dem angegebenen Stylesheet transformiert und das Resultat angezeigt

```
<?xml version="1.0" ?>
<?xml-stylesheet type="text/xsl" href="gruss.xsl"?>

<gruss>
  Hans Müller
</gruss>
```

- nun soll das Format der XML-Datei so erweitert werden, dass mehrere Namen abgelegt werden können:

```
<?xml version="1.0"?>
<namen>
  <name>Jim Panse</name>
  <name>Anna Konda</name>
  <name>Bernhard Diener</name>
</namen>
```



- jeder Name soll wie im vorangegangenen Beispiel transformiert werden
 - vor jedem Namen soll "Hallo" stehen
 - jeder Name soll in blauer Schrift ausgegeben werden
- man muss dafür sorgen, dass für jedes *name*-Element der gleiche HTML-Ausschnitt erzeugt wird
- Lösung: das *name*-Element bekommt sein eigenes Template

XSLT-Templates

```
<xsl:template match="name">
  <h1>
    Hallo
    <font color="blue">
      <xsl:value-of select="."/>!
    </font>
  </h1>
</xsl:template>
```

- dieses Template wird immer dann automatisch angewendet, wenn ein *gruss*-Element transformiert werden soll

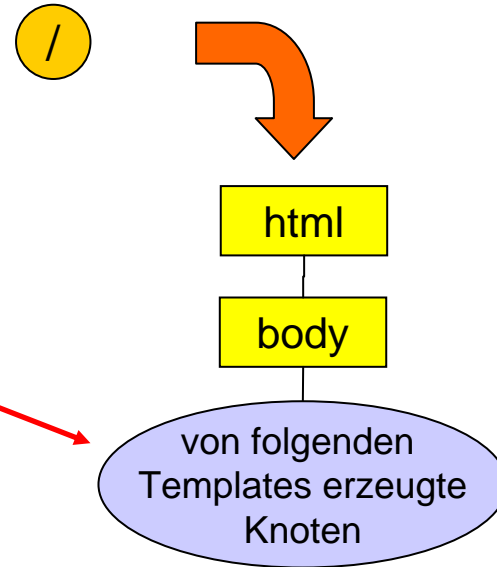
- da sich das Element `xsl:value-of` hier im Kontext von *gruss* befindet, kann zur Ausgabe des *gruss*-Rumpfes jetzt `select="."` geschrieben werden (oder alternativ: `select="text()"`)
- das Template für den Dokumentknoten sieht nun so aus:

```
<xsl:template match="/">
  <html>
    <body>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

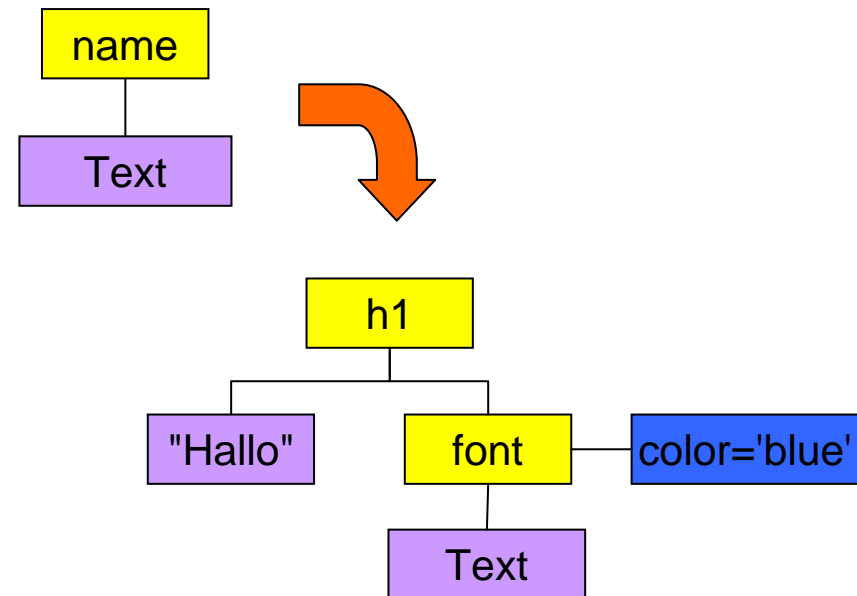
- `<xsl:apply-templates/>` weist den XSLT-Prozessor an, die Verarbeitung des XML-Dokuments bei den Kindknoten (hier das *gruesse*-Element) fortzusetzen
- ohne Aufruf von `apply-templates` findet keine Verarbeitung der Kindknoten statt

XSLT-Templates

```
<xsl:template match="/">  
  <html>  
    <body>  
      <xsl:apply-templates/>  
    </body>  
  </html>  
</xsl:template>
```



```
<xsl:template match="name">  
  <h1>  
    Hallo  
    <font color="blue">  
      <xsl:value-of select="."/>  
    </font>  
  </h1>  
</xsl:template>
```



Standard-Templates

- Wenn der XSLT-Prozessor im Stylesheet kein passendes Template für einen Knoten findet, verwendet er eines der vordefinierten **Standard-Templates**:
 - Dokument- und Elementknoten
 - setzt die Verarbeitung bei den Kindknoten fort
Achtung: Attributknoten liegen nicht auf der *child*-Achse
 - Attributknoten
 - gib den Attributwert aus
 - Textknoten
 - gib den Text aus
 - Kommentarknoten
 - keine Aktion

Standard-Templates
von XSLT

```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<xsl:template match="*">
  <xsl:apply-templates/>
</xsl:template>

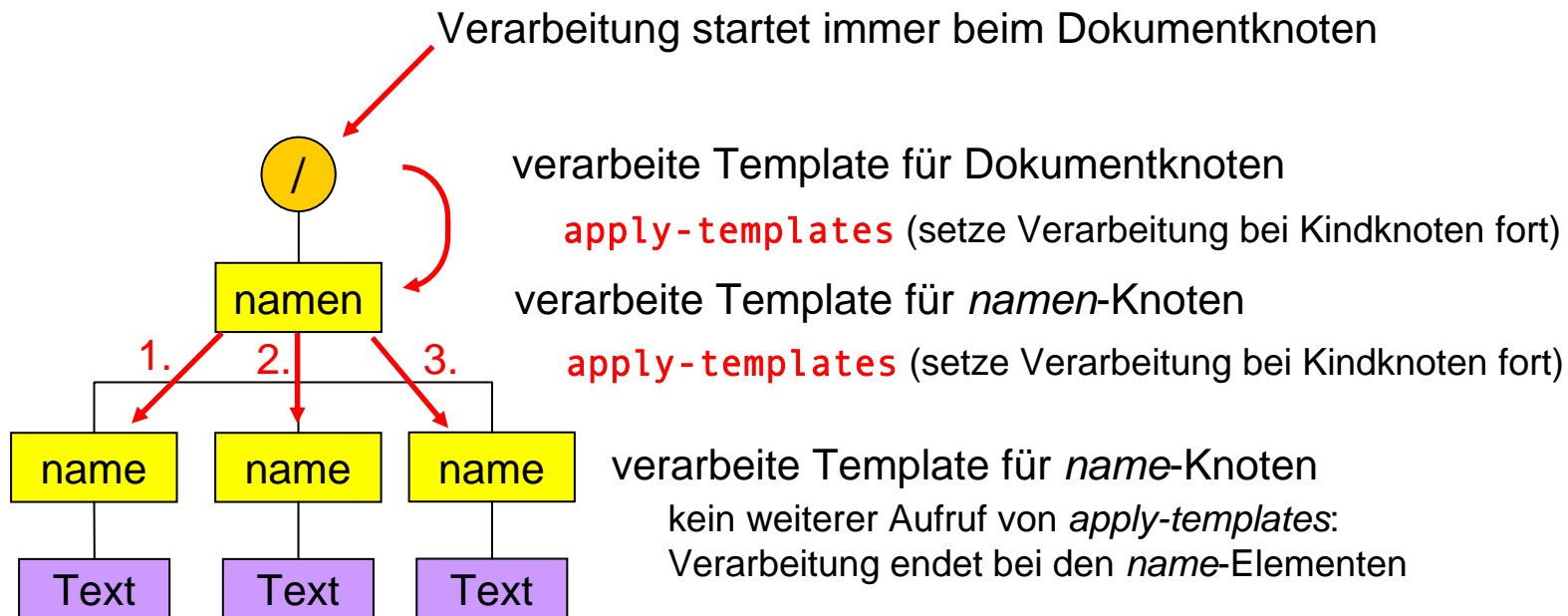
<xsl:template match="@*">
  <xsl:value-of select="."/>
</xsl:template>

<xsl:template match="text()">
  <xsl:value-of select="."/>
</xsl:template>

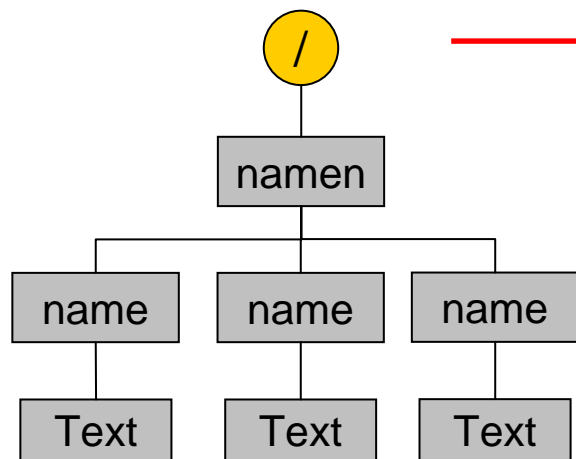
<xsl:template match="comment()"/>
```

Push-Processing

- durch den wiederholten Aufruf von *apply-templates* werden die Knoten des XML-Dokuments sukzessive von der Wurzel bis zu den Blättern verarbeitet
 - nicht das Stylesheet sondern das XML-Dokument bestimmt die Reihenfolge der Verarbeitung
 - diese Art der rekursiven Verarbeitung wird **Push-Processing** genannt

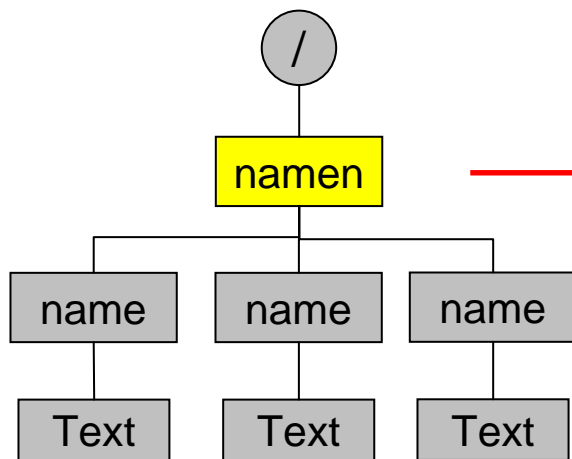
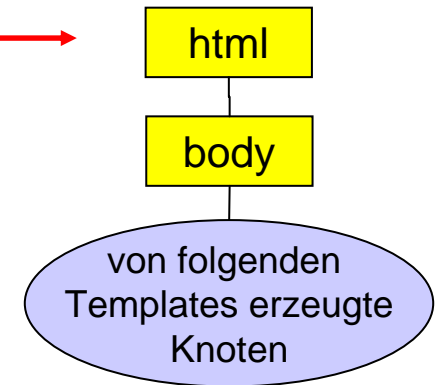


Push-Processing

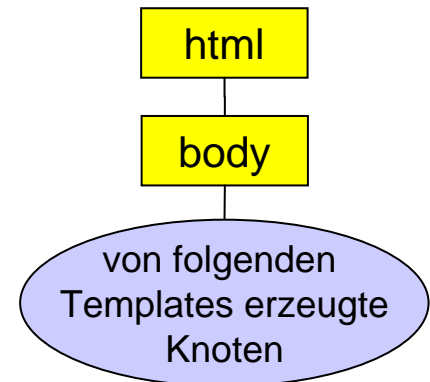


eigenes Template für /

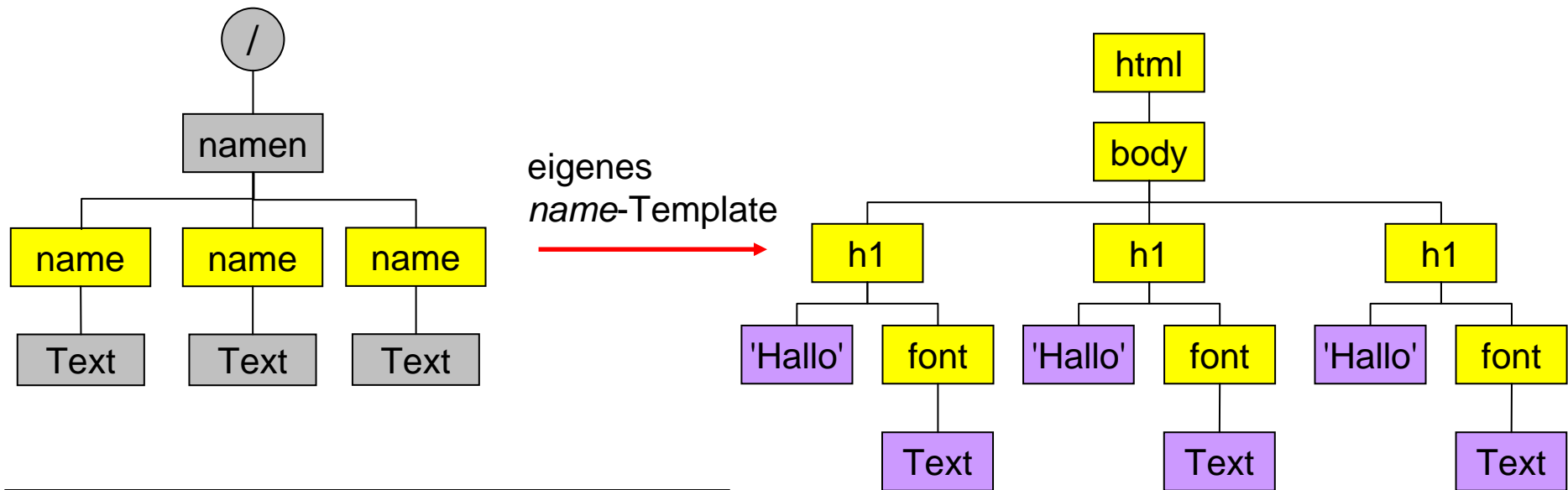
```
<xsl:template match="/">  
  <html>  
    <body>  
      <xsl:apply-templates/>  
    </body>  
  </html>  
</xsl:template>
```



Standard-Template für Elemente
(Matching bei Kindknoten fortsetzen)

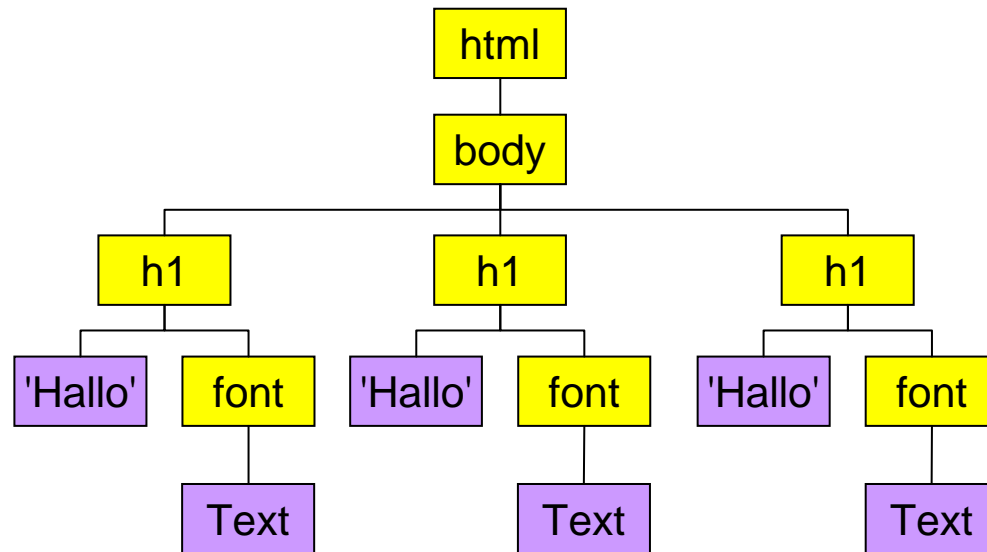


Push-Processing

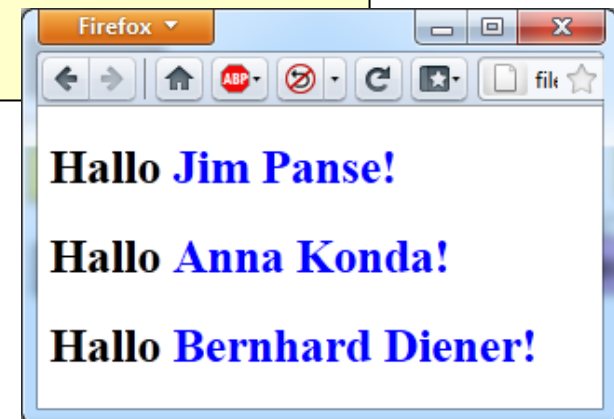


```
<xsl:template match="name">
  <h1>
    Hallo
    <font color="blue">
      <xsl:value-of select="."/>!
    </font>
  </h1>
</xsl:template>
```

Push-Processing



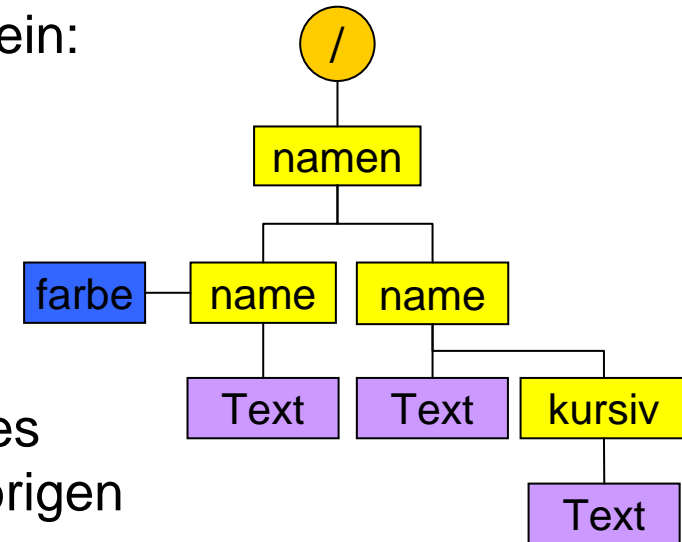
```
<html>  
  <body>  
    <h1>Hallo <font color="blue">Jim Panse!</font></h1>  
    <h1>Hallo <font color="blue">Anna Konda!</font></h1>  
    <h1>Hallo <font color="blue">Bernhard Diener!</font></h1>  
  </body>  
</html>
```



XSLT-Templates

- nun sollen folgende Konstruktionen möglich sein:

```
<?xml version="1.0"?>
<namen>
  <name farbe="green">Anna Konda</name>
  <name>Bernhard <kursiv>Diener</kursiv></name>
</namen>
```



- das Attribut *farbe* soll die Farbe des Grußtextes festlegen; das Element *kursiv* soll den zugehörigen Text kursiv darstellen
- das "/"-Template bleibt unverändert, der Rest sieht wie folgt aus:

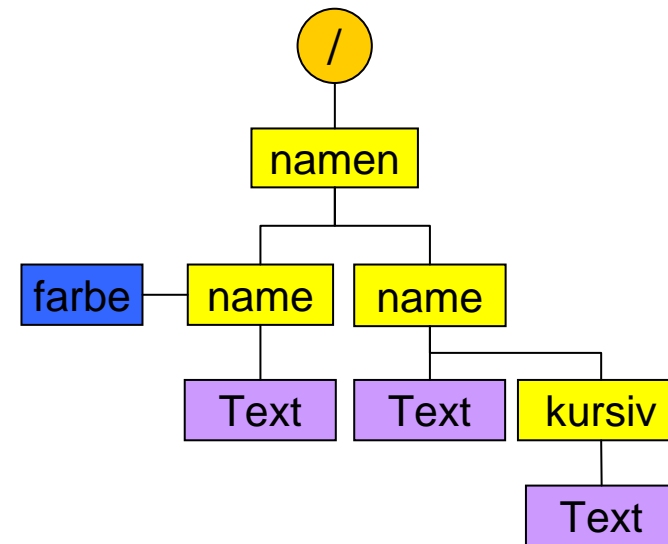
```
<xsl:template match="name">
  Hallo
  <font color="{@farbe}">
    <xsl:apply-templates/>
  </font>
</xsl:template>

<xsl:template match="kursiv">
  <i>
    <xsl:value-of select="."/>
  </i>
</xsl:template>
```

- da `xsl:value-of` innerhalb von Attributwerten nicht verwendet werden kann, gibt es für diesen Fall die Variante `{...}`
 - das Ergebnis des in Klammern stehenden XPath-Ausdrucks wird in einen String umgewandelt und an der angegebenen Stelle eingefügt

```
<xsl:template match="name">
  Hallo
  <font color="{@farbe}">
    <xsl:apply-templates/>!
  </font>
</xsl:template>

<xsl:template match="kursiv">
  <i>
    <xsl:value-of select="."/>
  </i>
</xsl:template>
```



- *xsl:apply-templates* sorgt hier dafür, dass die Verarbeitung bei den Kindknoten der *name*-Elemente fortgesetzt wird
 - wird ein Textknoten gefunden, greift das Standard-Template für *text* () und gibt den Text aus
 - wird ein *kursiv*-Element gefunden, wird das selbst definierte *kursiv*-Template ausgeführt
 - dieses erzeugt ein *i*-Element mit darin enthaltenem Text aus dem *kursiv*-Element der XML-Datei