Datenbanksysteme

Einführung in XML-Technologien

Teil 2: XQuery und XSLT

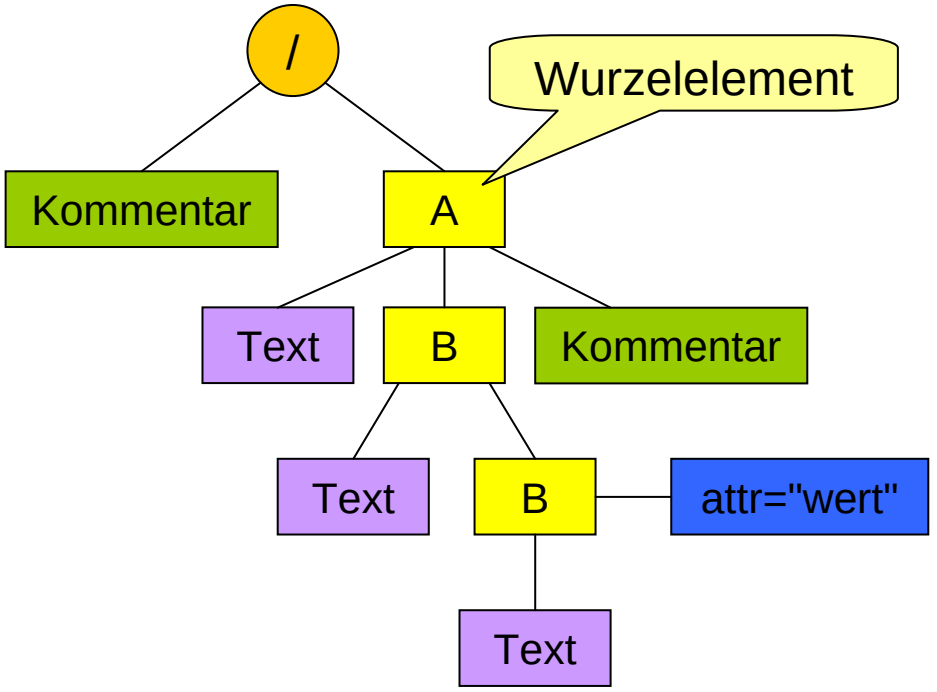
31.5.2011

Martin Gieseking

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>
  <!- zweiter Kommentar -->
</A>
```



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

XQuery-Tools

- 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

Sequenzen

- 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-Audrücke 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)`

XQuery: FLWOR-Ausdrücke

- **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, '&#10;')
```

  - 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 Sequenz-komponente 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, '&#10;')
```

```
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, '&#10;')
```

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

XQuery: Element-Konstruktoren

- 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 Update Facility

- 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 statt dessen 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/letzes 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>
```

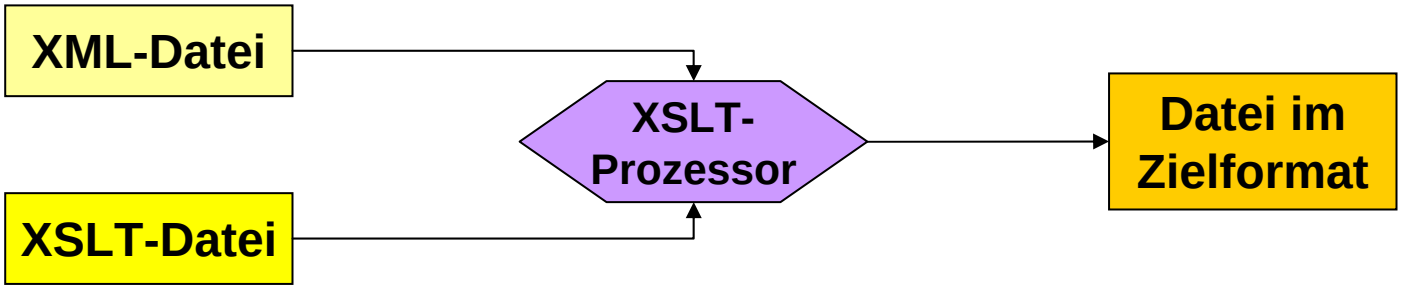
- `replace value of node` *knoten* `with` *ersatz*
  - ersetzt den Inhalt des angegebenen Knotens

```
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>
```

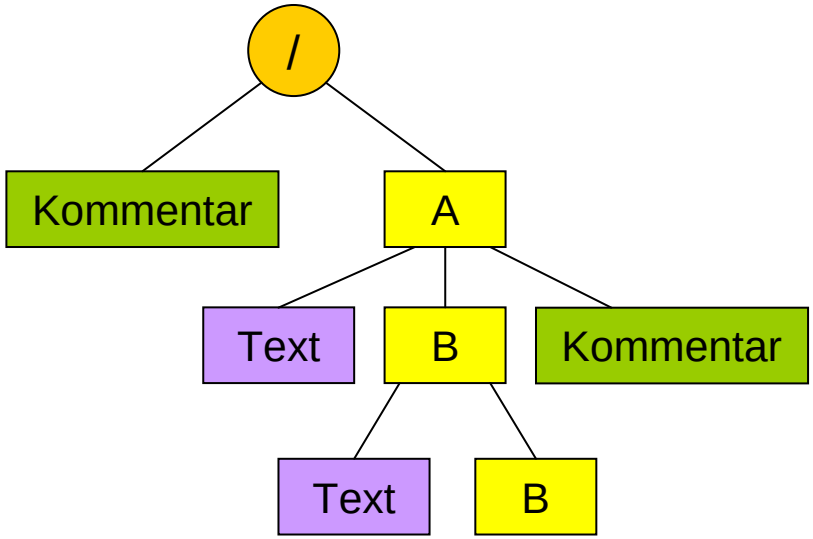
XSLT: Extensible Stylesheet Language Transformations

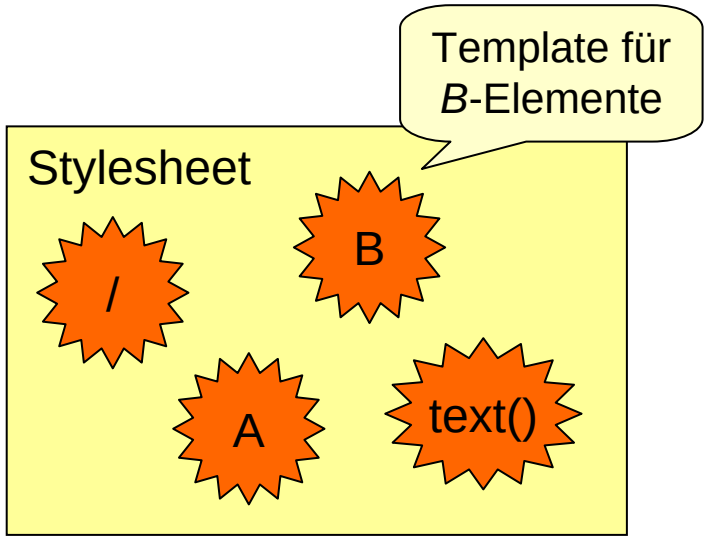
- XSLT ist eine deklarative Programmiersprache im XML-Format, mit der Transformationen von XML-Dateien in andere Formate beschrieben werden können
- XSLT-Stylesheets werden von einem XSLT-Prozessor ausgeführt
  - eine kleine Auswahl kostenloser Prozessoren:
    - *Saxon* von Michael Kay (http://saxon.sourceforge.net)
    - *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





Grundideen von XSLT

- 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

```
<?xml version="1.0" ?>
<name>
  Hans Müller
</name>
```

```
<html>
  <body>
    <h1>
      Hallo
      <font color="blue">
        Hans Müller!
      </font>
    </h1>
  </body>
</html>
```

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 Wurzelelement (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

XSLT-Beispiel

```
<?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

XSLT-Beispiel

```
<?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 des XSLT-Prozessors

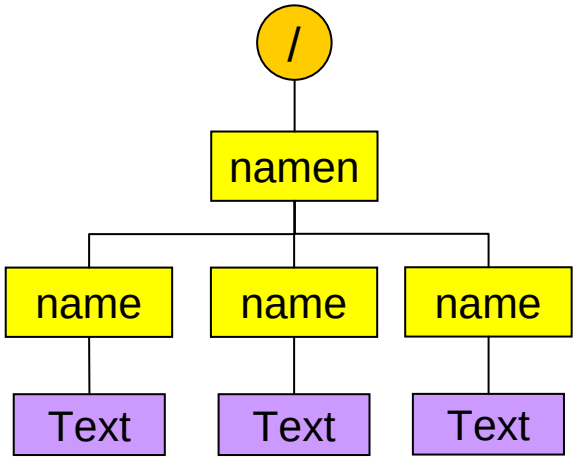
- **Aufruf im Web-Browser**
  - die meisten Web-Browser besitzen einen integrierten XSLT-Prozessor
  - in der zu transformierenden XML-Datei muss hinter der XML-Deklaration folgende Zeile hinzugefügt werden:
    ```
    <?xml-stylesheet type="text/xsl"
                     href="stylesheet.xsl"?>
    ```
  - 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>
```

XSLT-Beispiel

- 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

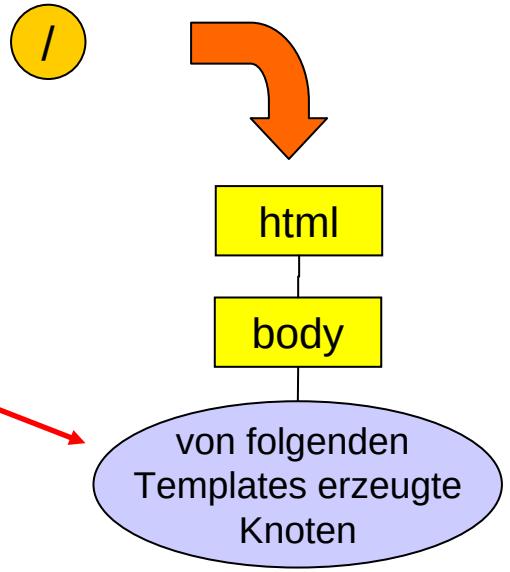
- da sich das Element `xsl:value-of` hier im Kontext von *gruss* befindet, kann zur Ausgabe des *gruss*-Rumpfes jetzt `select="."` geschrieben werden (oder alternativ: `select="text()"`)
- das Template für den Dokumentknoten sieht nun so aus:

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

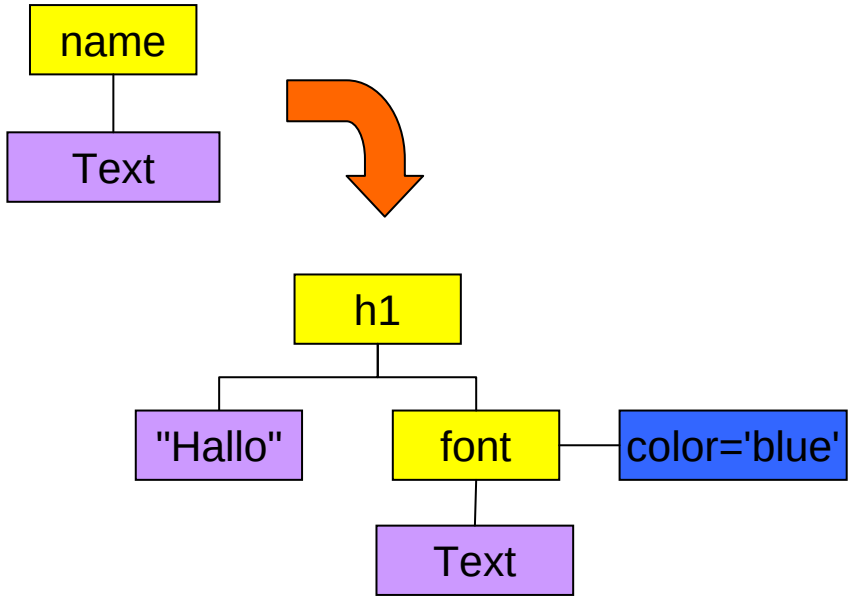
- `<xsl:apply-templates/>` weist den XSLT-Prozessor an, die Verarbeitung des XML-Dokuments bei den Kindknoten (hier das *gruesse*-Element) fortzusetzen
- ohne Aufruf von `apply-templates` findet keine Verarbeitung der Kindknoten statt

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

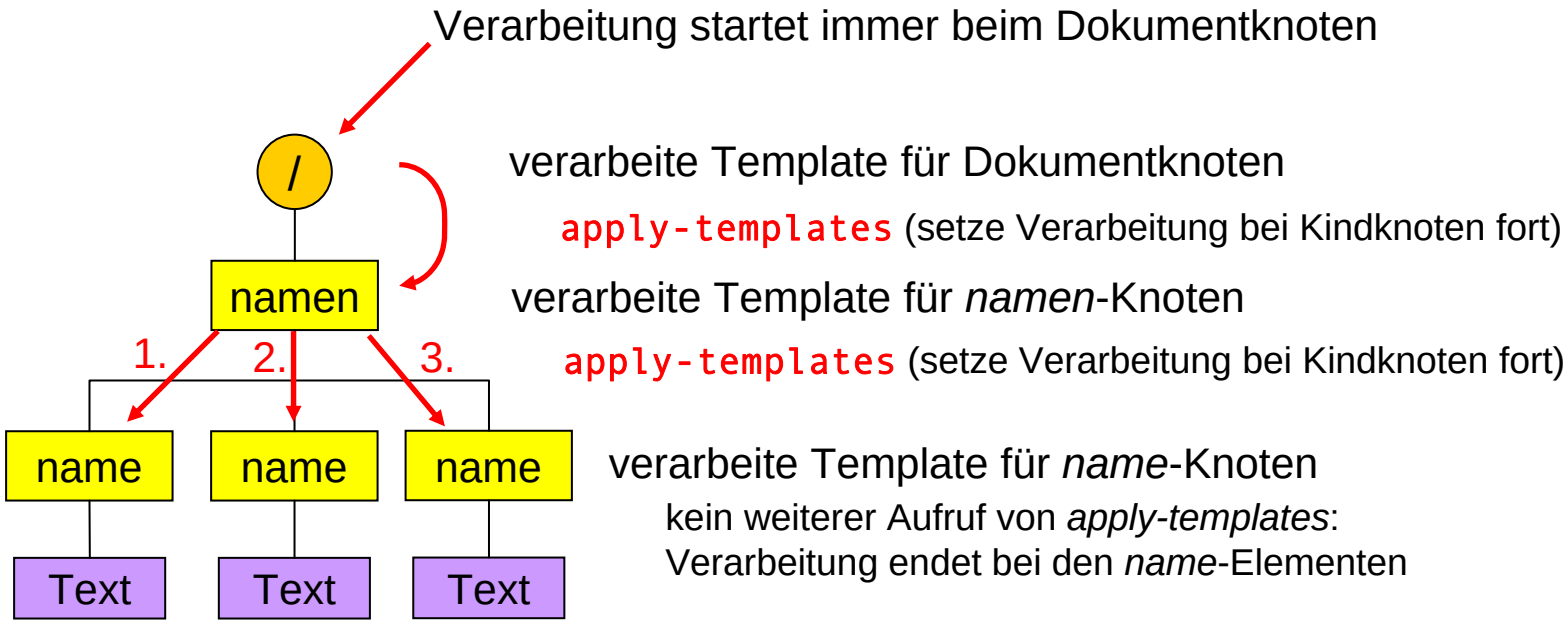
- Wenn der XSLT-Prozessor im Stylesheet kein passendes Template für einen Knoten findet, verwendet er eines der vordefinierten **Standard-Templates**:
  - Dokument- und Elementknoten
    - setzt die Verarbeitung bei den Kindknoten fort
      Achtung: Attributknoten liegen nicht auf der *child*-Achse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

Standard-Templates von XSLT

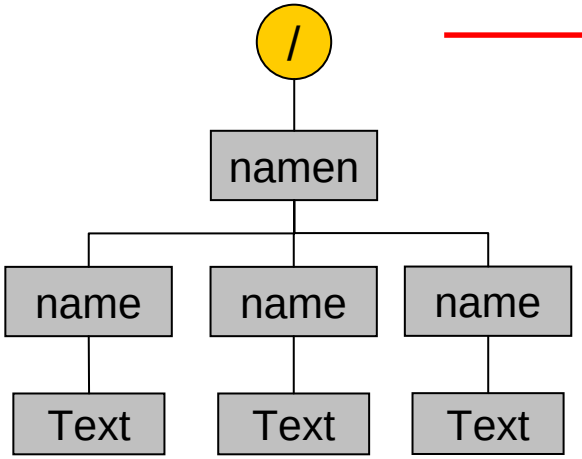
```
<xsl:template match="/">
  <xsl:apply-templates/>
</xsl:template>

<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

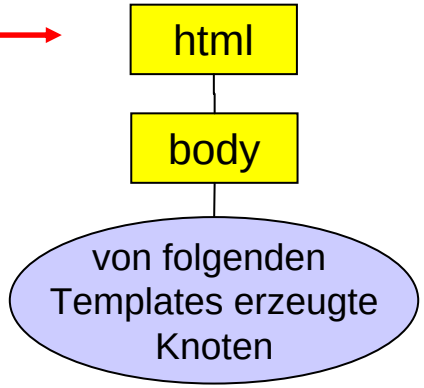


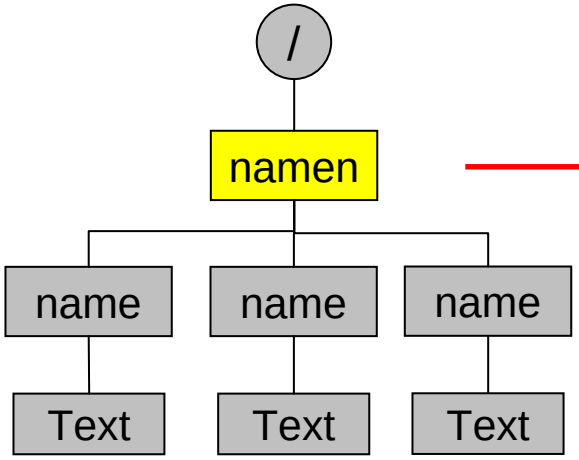
Push-Processing



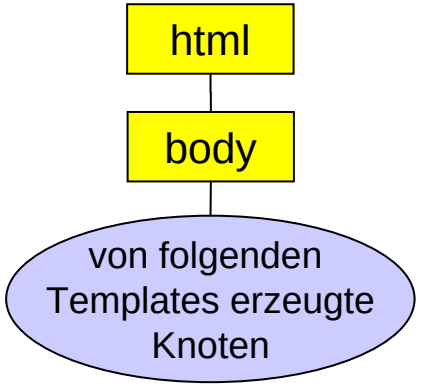
eigenes Template für /

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

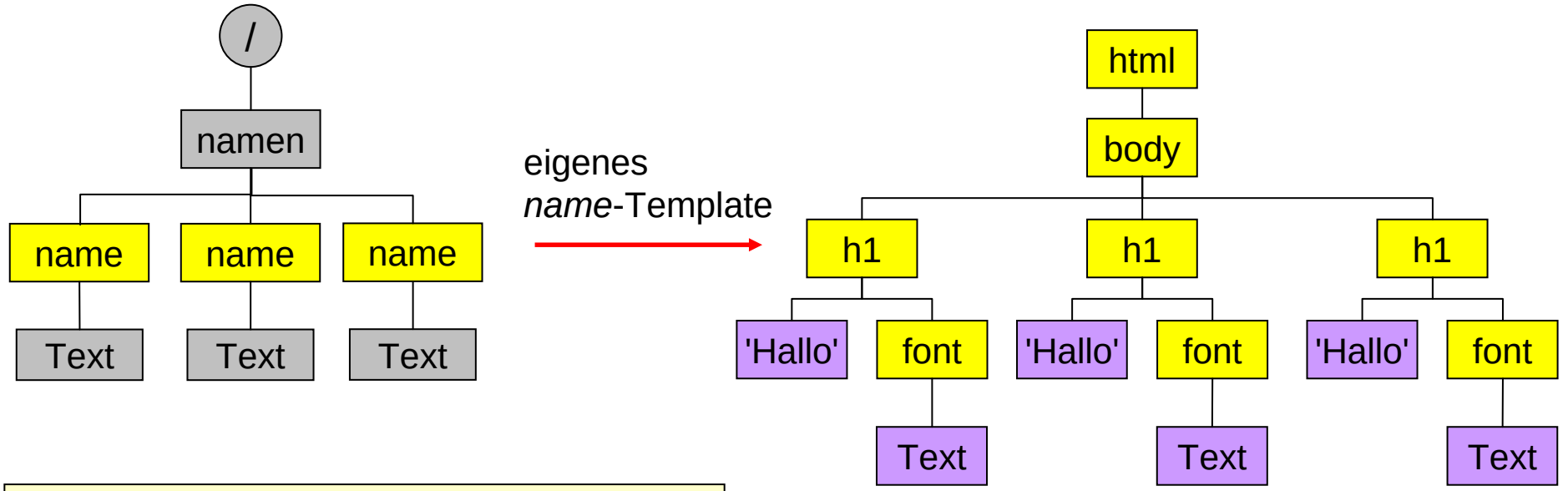




Standard-Template für Elemente
(Matching bei Kindknoten fortsetzen)

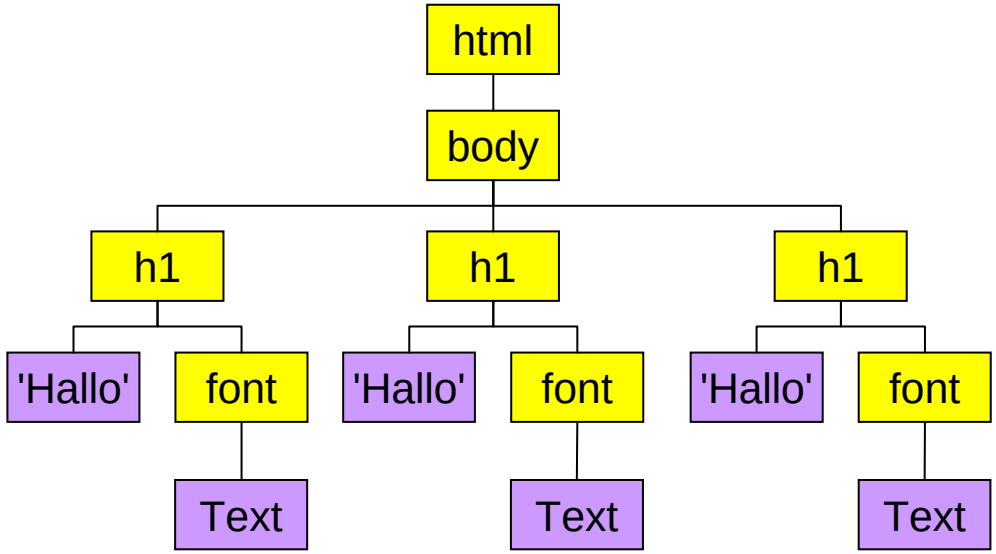


Push-Processing

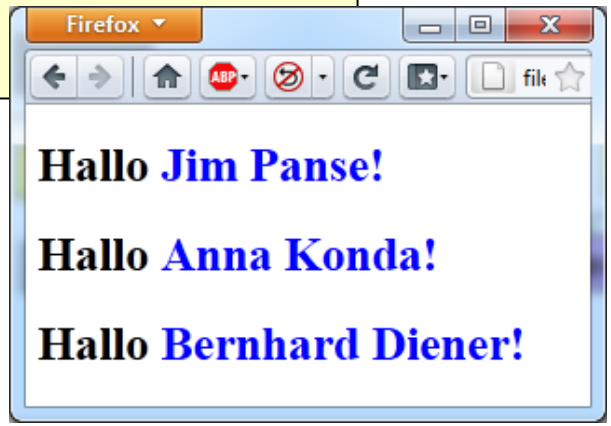



```
<xsl:template match="name">
  <h1>
    Hallo
    <font color="blue">
      <xsl:value-of select="."/>!
    </font>
  </h1>
</xsl:template>
```

Push-Processing



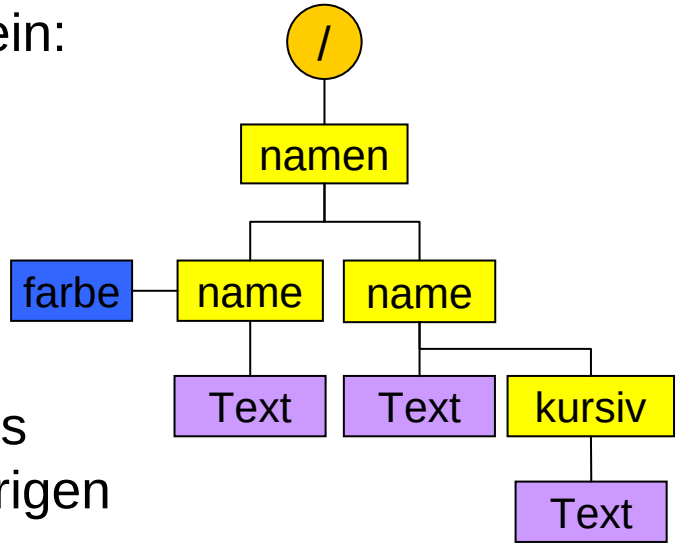

```
<html>
  <body>
    <h1>Hallo <font color="blue">Jim Panse!</font></h1>
    <h1>Hallo <font color="blue">Anna Konda!</font></h1>
    <h1>Hallo <font color="blue">Bernhard Diener!</font></h1>
  </body>
</html>
```



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>
```



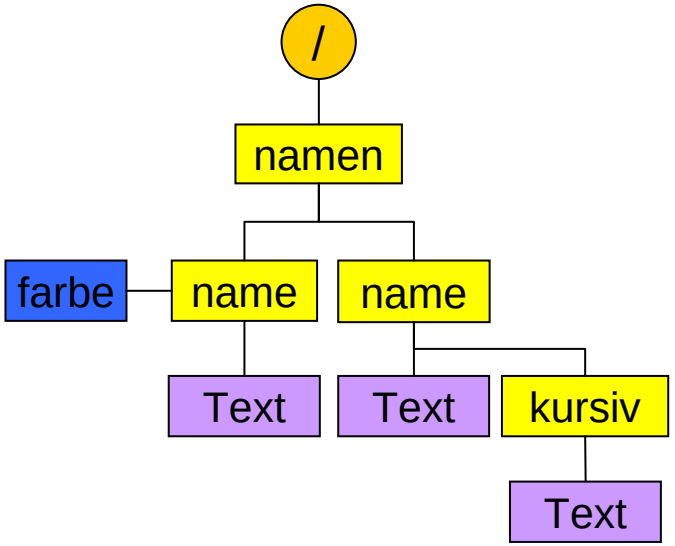
- das Attribut *farbe* soll die Farbe des Grußtextes festlegen; das Element *kursiv* soll den zugehörigen Text kursiv darstellen
- das "/"-Template bleibt unverändert, der Rest sieht wie folgt aus:

```
<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

XSLT-Templates

```
<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