

Semistrukturierte Daten

Sommersemester 2008

Teil 8: XSLT 1.0

- 8.1. Einführung
- 8.2. Aufbau von XSLT-Stylesheets
- 8.3. Abarbeitung von XSLT-Stylesheets
- 8.4. Result Tree
- 8.5. Kontrollstrukturen
- 8.6. Variablen, Parameter
- 8.7. weitere Features
- 8.8. Java-API TrAX



8.1. Einführung

- Stylesheets
- Recommendations
- XSLT + XSL-FO
- XSLT-Prozessoren

Stylesheets

- Aufgaben von Stylesheets:
 - Layout
 - Transformation
 - Informationsextraktion/Abfragen
- Konvertieren von XML Dokumenten
 - in andere XML Dokumente
 - in HTML und andere Formate
 - in Text
- XML-Stylesheets:
 - CSS: Cascading Stylesheets
 - XSL: Extensible Stylesheet Language

Recommendations

■ Bestandteile von XSL:

- XSLT (XSL Transformations)
- XSL-FO (XSL Formatting Objects)
- baut sehr stark auf XPath auf

■ XSLT:

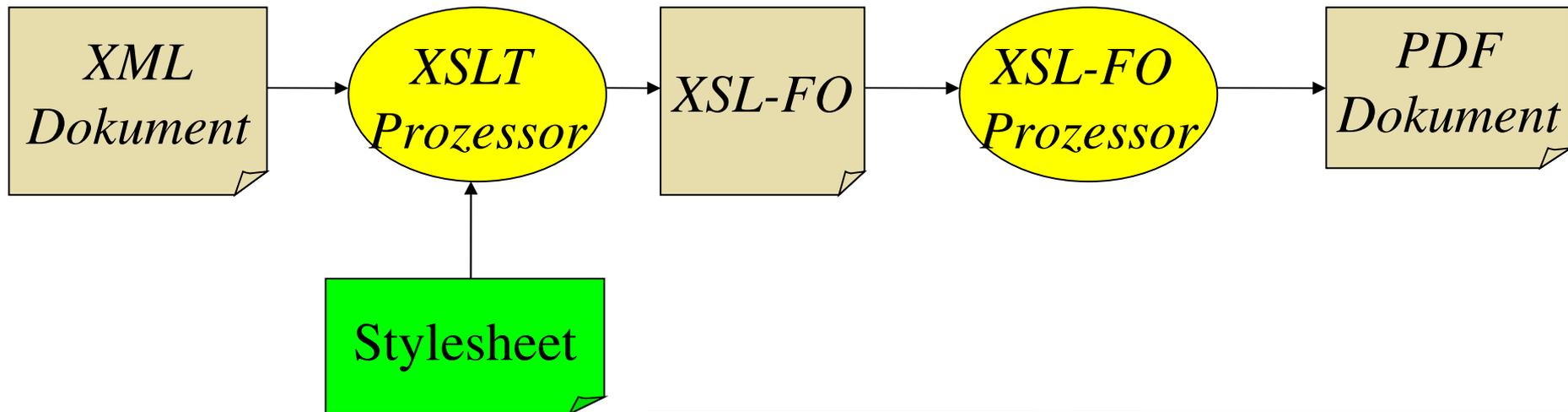
- Version 1.0: W3C Recommendation seit 1999
- Version 2.0: sehr große Erweiterung, z.B.: verwendet XPath 2.0 statt XPath 1.0. (Recommendation seit Jänner 2007)

■ XSL-FO:

- W3C Recommendation seit Dezember 2006
- Ursprünglich als Hauptteil von XSL gedacht
- Hat wesentlich geringere Bedeutung als XSLT

XSLT + XSL-FO

- XSLT benötigt XSLT Prozessor
 - z.B. Xalan, Saxon, die meisten Browser
 - meist ohne XSL-FO: Erstellung von HTML, Text, SVG, etc.
- XSL-FO benötigt FO Prozessor
 - z.B.: Apache FOP für **PDF Generierung**
 - Formatierte Ausgabe (= Rendering)



XSLT Prozessoren

- **Xalan** (Teil des Apache Projekts)

- <http://xml.apache.org/xalan-j/index.html>

```
set CLASSPATH=xalan.jar;xercesImpl.jar
java org.apache.xalan.xslt.Process -IN test.xml
    -XSL test.xsl [-OUT out.xml]
```

- **Saxon** (von Michael Kay)

- <http://saxon.sourceforge.net/>

```
set CLASSPATH=saxon.jar;saxon-jdom.jar
java com.icl.saxon.StyleSheet test.xml test.xsl
    > out.xml
```

- **XML-Spy**

- enthält unter anderem: XSLT Prozessor, XSLT Debugger
- Gratisversion für 30 Tage erhältlich
- http://www.altova.com/products_ide.html

8.2. Aufbau von XSLT-Stylesheets

- Überblick
- Elemente im XSLT-Namespace
- Templates
- Kurzschreibweise mittels LRE

Überblick

- XSLT-Stylesheets sind selbst XML-Dokumente

```
<?xml version="1.0"?>
```

```
<xsl:stylesheet
```

```
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```
  <!-- Hier kommen die top-level Elemente -->
```

```
</xsl:stylesheet>
```

- Aufbau:

- Namespace: "http://www.w3.org/1999/XSL/Transform"

- Präfix: üblicherweise xsl

- Dokumentelement: `<xsl:stylesheet>` (oder `<xsl:transform>`)

- Verknüpfung XML-Dokument/Stylesheet:

- mittels PI: `<?xml-stylesheet type="text/xsl" href="lva.xsl"?>`

- oder als Parameter beim Aufruf des XSLT-Prozessors

Top-level Elemente

<code>xsl:template</code>	definiert Regeln für die Transformation source tree -> result tree
<code>xsl:output</code>	Ausgabemethode: XML, HTML, Text
<code>xsl:attribute-set</code>	Attributmengen für das Output-Dokument
<code>xsl:variable</code>	globale Variablen
<code>xsl:param</code>	Parameter

Weitere Top-level Elemente:

`xsl:import`, `xsl:include`, `xsl:key`, `xsl:decimal-format`,
`xsl:strip-space`, `xsl:preserve-space`, etc.

Weitere XSLT Elemente

xsl:apply-templates	definiert die Knotenmenge, bei der die Abarbeitung weitergeht.
xsl:value-of	definiert String-Wert mittels XPath-Ausdruck
xsl:copy	kopiert den momentanen Knoten
xsl:copy-of	kopiert den ganzen Sub-Baum
xsl:for-each	Schleife über eine Knotenmenge
xsl:if	bedingte Anweisung
xsl:choose / xsl:when	
xsl:otherwise	Alternativen
xsl:sort	sortiert eine Knotenmenge

Weitere Elemente:

xsl:call-template, xsl:element, xsl:attribute, xsl:text, etc.

Beispiel

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40" version="1.0">
  <xsl:output method="html"/>
  <xsl:template match="/">
    <html><xsl:apply-templates/></html>
  </xsl:template>
  <xsl:template match="veranstaltung/titel">
    <i><xsl:value-of select="."/></i><br/>
  </xsl:template>
  <xsl:template match="text()">
  </xsl:template>
</xsl:stylesheet>
```

Elementinhalt wird in den Output geschrieben

Zusätzliches Template, das den Textinhalt der anderen Elemente nicht schreibt.

Beispiel 1

Browser

Semistrukturierte Daten
Webdatenextraktion
Datenbanksysteme



Templates

- Attribute von `xsl:template`:

- `match="pattern"`: Aktion wird ausgeführt für alle Knoten, für die dieser Patternpfad erfüllt ist, d.h.: Alle Knoten, für die es einen context-node gibt, von dem aus sie mit diesem Patternpfad selektiert würden.
- `name`: benanntes template
(Aufruf `<xsl:call-template name="name"/>`)
- `priority`: Ändern der Default-Priorität

- Attribute von `xsl:apply-templates`:

- `select`: suche für die selektierten Knoten nach einem anwendbaren Template.

Kurzschreibweise mittels LRE

LRE (= literal result element):

- LRE = Element **außerhalb des XSLT-Namespaces**: wird inklusive Attributen und Namespaces in den Output durchgereicht.
- LRE darf selbst beliebigen Inhalt haben (insbes. XSLT-Elemente außer top-level Elementen).
- **Kurzschreibweise mit LRE als Dokument-Element** erlaubt, z.B.:

```
<?xml version = "1.0" encoding = "ISO-8859-1" ?>
<html xsl:version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/TR/REC-html40">
  <xsl:for-each select="//veranstaltung/titel">
    <i><xsl:value-of select="."/></i><br/>
  </xsl:for-each>
</html>
```

8.3. Abarbeitung von XSLT-Stylesheets

- Ablaufsteuerung
- Programmiersprache XSLT
- Select vs. Match
- Built-in Templates
- Template-Auswahl

Ablaufsteuerung (1)

- XSLT Prozessor durchläuft den **Source Tree** und erzeugt einen **Result Tree**:
 - Durchlaufen Source Tree:
 - Falls nichts anderes festgelegt wird (z.B. mittels **select**-Attribut in **xsl:apply-templates**) => Depth-First Traversal
 - Es gibt immer einen **current node** und eine **current node list**
 - => daraus ergibt sich z.B. der Kontext für XPath-Auswertung
 - XSLT Prozessor sucht passendes Template für den current node
 - Wenn current node das XPath-Pattern eines Template 'matched', wird dieses Template ausgeführt.
 - Immer nur 1 Template pro current node => Konfliktauflösung (s.u.)

Ablaufsteuerung (2)

- Stylesheet ist deklarative Regelmenge (templates)

```
<xsl:template match="XPath-Ausdruck">
```

```
  Substitutionsausdruck
```

```
</xsl:template>
```

- apply-templates Element:

- **strukturelle Rekursion**: wende Templates rekursiv auf Kinder (bzw. Nachfahren) im Source Tree an.
- Ohne "apply-templates": Nach Abarbeitung eines Templates würde die Bearbeitung des aktuellen Pfades enden und der XSLT-Prozessor mit der current node list bzw. mit dem parent weiter machen.



Programmiersprache XSLT

- Imperative Programmiersprachen:
 - Fallunterscheidungen mittels IF, IF ELSE, etc.
 - Wiederholung mittels Schleifen (oder Rekursion)
- XSLT (analog zu **deklarativen Programmiersprachen** wie PROLOG und LISP)
 - **pattern matching**
 - **(structural) recursion**
- Kontrollstrukturen in XSLT:
 - if, for-each, choose "vereinfachen" das Leben
 - XSLT ist Turing-vollständig (auch ohne Kontrollstrukturen)

Select vs. Match

■ select-Attribut in xsl:apply-templates

- **Aufgabe des location path:** Bestimme Menge aller Knoten Y , die man vom context node X aus mit dem select-Pfad selektieren kann. Für jedes Y wird dann ein passendes Template gesucht.
- auch absolute Pfade möglich
- beliebige Achsen im Pfad erlaubt => Endlosschleifen möglich

■ match-Attribut in xsl:template

- **Aufgabe des location path:** Für einen bestimmten Knoten Y (der mittels apply-templates selektiert wurde) wird getestet, ob dieser von irgendeinem context-node X aus mit dem match-Pfad selektiert würde.
- nur child und attribute Achse sowie Abkürzung // erlaubt!
- absolute Pfade erlaubt.

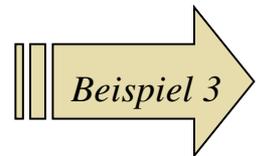
Built-In Templates

- **Idee:** Falls es für einen Knoten kein Template gibt, dann werden vom XSLT-Prozessor built-in Templates verwendet.
- **Folgende Templates sind (im XSLT Standard) vordefiniert:**
 - **Elemente:** built-in template tut selbst nichts, aber ruft rekursiv die Kinder des current node auf (damit Rekursion weitergeht).

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

- **Attribute und Text-Knoten:** Wert wird standardmäßig einfach in den Output geschrieben. Allerdings: Attribute werden beim Dokumentendurchlauf nicht durchlaufen, d.h.: "apply-templates" erforderlich, damit built-in Template greift!

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



Template-Auswahl

■ Prinzip:

- Auf jeden Knoten wird immer exakt ein Template angewandt
- Wenn mehrere Templates den current node matchen, muss ein Template ausgewählt werden.
- Konfliktauflösung: Template mit der höchsten **Priorität** wird ausgeführt.

■ Prioritäten:

- Je nach Struktur des match-Attributs hat jedes Template eine default-Priorität
- Benutzer kann Priorität eines Template mittels **priority**-Attribut selbst festlegen.
- Falls dadurch keine eindeutige Auswahl erfolgt: Verhalten prozessorabhängig, z.B.: Abbruch mit Fehler, Auswahl des letzten Templates im Stylesheet

Default-Priorität

■ Idee:

- Je spezifischer eine Regel ist, umso höher ist die Priorität.
- Der "Normalfall" (d.h.: child/attribute-Achse + qualifizierter Name) bekommt Priorität 0.
- Spezifischere Patterns haben positive Priorität.
- Weniger spezifische Patterns haben negative Priorität.

■ Berechnung der Default-Priorität:

- Mit | getrennte Patterns werden wie 2 getrennte Templates behandelt
- child/attribute-Achse + qualifizierter Name: Priority = 0
- child/attribute-Achse + unqualifizierter Name: Priority = -0.25
- child/attribute-Achse + NodeTest: Priority = -0.5
- sonst (insbes.: > 1 location step) Priority = 0.5

8.4. Result Tree

- `xsl:element`
- `xsl:attribute`
- `xsl:attribute-set`
- weitere Knoten-Typen
- `xsl:value-of`
- `xsl:copy`, `xsl:copy-of`
- Whitespace

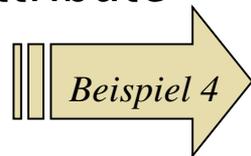
xsl:element

- Erzeugt einen Element-Knoten im Result Tree
- Attribute von `xsl:element`:
 - `name` (mandatory): Name des Elements
 - `namespace`: URI-reference
(Behandlung des Präfix ist prozessorabhängig)
 - `use-attribute-sets`: Liste von Attributmengen
- Vorteile gegenüber LRE:
 - Name kann mittels XPath-Ausdruck berechnet werden
- Beispiel:

```
<xsl:element name="lehrveranstaltungen"  
  namespace="http://www.dbai.tuwien.ac.at/education">  
  <xsl:apply-templates/>  
</xsl:element>
```

xsl:attribute

- Erzeugt ein Attribut im Result Tree
 - Als (erstes) Sub-Element des zugehörigen Elements
 - Sowohl bei LRE als auch bei xsl:element erlaubt
- Attribute von **xsl:attribute**:
 - **name** (mandatory): Name des Attributs
 - **namespace**: URI-reference
(Behandlung des Präfix ist prozessorabhängig)
- Attributwert: als Inhalt des xsl:attribute-Elements
- Vorteile gegenüber LRE mit Attribut:
 - Name kann mittels XPath-Ausdruck berechnet werden
 - Beliebig komplexer Content (insbes. xsl-Elemente) des xsl:attribute-Elements zur Berechnung des Attributwerts,
z.B.: **xsl:value-of**, **xsl:apply-templates**.



Attributwerte berechnen

- Problem:

- Markup innerhalb des Markup ist nicht möglich.
- => folgender Ausdruck ist nicht erlaubt:

```
<elem attr="<xsl:value-of select ='expr' .....>
```

- Zwei Möglichkeiten:

- AVT (= Attribute Value Template) = XPath Ausdruck in { }:

```
<elem attr="{expr}" .....>
```

z.B. Element-Inhalt in Attribut-Wert umwandeln:

```
<xsl:template match="elem">  
  <new-elem attr="{text()}"> ... </new-elem>  
</xsl:template>
```

- `xsl:attribute`-Element:

Attributwert = Inhalt des `xsl:attribute`-Elements

xsl:attribute-set

- **xsl:attribute-set**
 - Top-level Element im XSLT-Stylesheet
 - um Gruppe von Attributen zu definieren
 - Inhalt: **xsl:attribute**-Elemente
- Attribute von **xsl:attribute-set**
 - **name** (mandatory): Name der Attribut-Menge
 - **use-attribute-sets**: referenziert andere Attribut-Mengen
- Verwendung (in Elementen oder anderen Attribut-Mengen)
 - mittels Attribut use-attribute-sets
- Typische Anwendung:
 - verschiedene attribute-sets für unterschiedliche Format-Anweisungen im Output

Beispiel

```
<xsl:attribute-set name="plain_table">
  <xsl:attribute name="border">0</xsl:attribute>
  <xsl:attribute name="cellpadding">3</xsl:attribute>
  <xsl:attribute name="cellspacing">4</xsl:attribute>
</xsl:attribute-set>
<xsl:attribute-set name="bordered_table">
  <xsl:attribute name="border">1</xsl:attribute>
  <xsl:attribute name="cellpadding">2</xsl:attribute>
  <xsl:attribute name="cellspacing">3</xsl:attribute>
</xsl:attribute-set>
<xsl:template match ="buecher">
  <xsl:element name="table" use-attribute-sets ="plain_table">
    <xsl:apply-templates/>
    <!-- z.B.: Tabelleneintrag fuer jedes buch-Element -->
  </xsl:element>
</xsl:template>
```



Weitere Knoten-Typen

■ xsl:text

- Erzeugt einen Text-Knoten, z.B.: `<xsl:text disable-output-escaping="yes">< & </xsl:text>`

■ xsl:comment

- Erzeugt einen Kommentar, z.B.:
`<xsl:comment>Hier steht der Kommentar</xsl:comment>`

■ xsl:processing-instruction

- Erzeugt eine processing instruction, z.B.:
`<xsl:processing-instruction name="xml-stylesheet">
"text/css" href="lehre.css"
</xsl:processing-instruction>`

■ Inhalt dieser Elemente:

- xsl:text: PCDATA
- Die anderen 2: beliebig komplexer Inhalt.

xsl:value-of

■ `xsl:value-of`

- Erzeugt einen Textknoten im Result Tree (der mit angrenzenden Text-Knoten verschmolzen wird).

■ Attribute von `xsl:value-of`

- `select` (mandatory): XPath-Ausdruck mit Result-Type String (andere Typen werden zu String konvertiert)
- `disable-output-escaping`: (nur für & < > " ')

■ Beispiel:

```
<xsl:template match="person">
  <p>
    <xsl:value-of select="@vorname"/>
    <xsl:text> </xsl:text>
    <xsl:value-of select="@nachname"/>
  </p>
</xsl:template>
```



xsl:copy

■ xsl:copy

- Kopiert den current node in den Output ("shallow copying")
- Bei Element-Knoten: NS-Knoten werden ebenfalls kopiert, aber Attribute und Inhalt werden nicht kopiert!
- Das xsl:copy-Element selbst kann beliebigen Inhalt haben (ist nur relevant, wenn der current node ein Element-Knoten oder der root node ist)
- Optionales Attribut: use-attribute-sets

■ Beispiel (identity transformation):

```
<xsl:template match="@*|node() ">  
  <xsl:copy>  
    <xsl:apply-templates select="@*|node()" />  
  </xsl:copy>  
</xsl:template>
```

xsl:copy-of

- **xsl:copy-of** hat immer leeren Inhalt
- Einziges Attribut: **select**
 - Wenn XPath-Ausdruck des select-Attributs ein **node-set** ergibt:
 - => Kopiert alle Bäume, deren Wurzel in der Knotenmenge liegt, in den Output (insbes. auch Attribute und Inhalt von Element-Knoten)
 - Bei Result-Type **string/boolean/number**:
 - => xsl:copy-of verhält sich gleich wie xsl:value-of
- Unterschiede zw. **xsl:copy-of** und **xsl:copy**:
 - xsl:copy kopiert nur den Markup, xsl:copy-of auch den Inhalt
 - xsl:copy kann selbst beliebig komplexen Inhalt haben, um den Element-Inhalt im Result Tree zu erzeugen. xsl:copy-of ist ein leeres Element.



Whitespace

Whitespace-Verhalten ist steuerbar mit:

- `<xsl:text>`
- `<xsl:output indent="yes"/>`
- `normalize-space (expr)`
- `<xsl:strip-space elements="el1 el2"/>` definiert Elemente, bei denen Whitespaces getrimmt werden
- Gegenteil: `<xsl:preserve-space>` (ist der Default)
- Attribut `xml:space` in jedem Element erlaubt:
mögliche Werte `default|preserve`

8.5. Kontrollstrukturen

- `xsl:for-each`
- `xsl:if`
- `xsl:choose`

xsl:for-each

■ `xsl:for-each`:

- Schleife über eine Knotenmenge, die mittels XPath-Ausdruck des `select`-Attributs bestimmt wird.
- `current node` und `current node list` innerhalb der Schleife entsprechen dieser selektierten Knotenmenge
- Inhalt von `xsl:for-each`: Anweisungen der Schleife

■ Beispiel:

```
<xsl:template match="lehre">
  <xsl:value-of select="veranstaltung[1]/titel"/>
  <xsl:for-each select="veranstaltung[position() > 1]">
    <xsl:text>, </xsl:text>
    <xsl:value-of select="titel" />
  </xsl:for-each>
</xsl:template>
```

-> *Liste aller Lehrveranstaltungstitel*

for-each vs. apply-templates

- Üblicherweise austauschbar, z.B.:

```
<xsl:template match="lehre">
  <xsl:value-of select="veranstaltung[1]/titel"/>
  <xsl:apply-templates select="veranstaltung[position() > 1]"/>
</xsl:template>

<xsl:template match="veranstaltung[position() > 1]">
  <xsl:text>, </xsl:text>
  <xsl:value-of select="titel" />
</xsl:template>
```

- Vorteile von for-each:

- Einfache Formulierung von **Joins** (z.B.: mit **Variablen**)
- Auch in einfachen Stylesheets (mit LRE als top-level Element) möglich (mit absolutem Pfad im select-Ausdruck)



xsl:if

■ xsl:if

- Definition von bedingten Anweisungen
- **test**-Attribut enthält die Bedingung als boolean Expression
- Kein else-Zweig möglich

■ Beispiel:

```
<xsl:template match="veranstaltung">  
  <xsl:value-of select="titel"/>  
  <xsl:if test="not(position()=last())">, </xsl:if>  
</xsl:template>
```

xsl:choose

■ xsl:choose

- Zur Unterscheidung beliebig vieler Fälle im Stil von
if ... then ... else if ... then else if ... then ... else
- Keine Attribute; Inhalt enthält ein oder mehrere xsl:when-
Elemente und optional ein xsl:otherwise-Element
- Ausgeführt wird Inhalt des ersten **xsl:when**-Elements, für das das
test-Attribut true ergibt. Falls alle false liefern, wird (falls
vorhanden) der Inhalt des **xsl:otherwise** Elements ausgeführt.

■ xsl:when: **test**-Attribut enthält Bedingung als boolean Expression

■ xsl:otherwise: (optional) keine Attribute

z.B.: **<xsl:choose>**

```
<xsl:when test="expr1">...das  
geschieht...</xsl:when>  
<xsl:when test="expr2">...bzw. das...</xsl:when>  
<xsl:otherwise> und ansonsten das <xsl:otherwise>  
</xsl:choose>
```



8.6. Variablen, Parameter

- Variablen
- Parameter



Variablen

- Definition:
 - Mittels `xsl:variable` Element
 - Global (als top-level Element) oder lokal innerhalb eines Template
 - Lokale Definition kann globale Definition überlagern
- Variablen-Bindung:
 - Mittels `select`-Attribut oder als Inhalt von `xsl:variable`, d.h.:
`<xsl:variable name = "var" select="expr"/>` oder
`<xsl:variable name = "var"/>....</xsl:variable>`
 - einmalige Zuweisung: Variablenwert wird nie mehr verändert.
- Verwendung von Variablen:
 - mit \$ vor dem Variablen-Namen: `$var`
 - z.B.: `<veranstaltung jahr="{ $thisyear }">`

Variablen

- Typische Anwendung:
bei **Joins**: um **unterschiedliche Kontexte** zu verbinden.
- Beispiel:

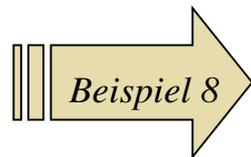
```
<xsl:for-each select="/lehre/veranstaltung">  
  <xsl:variable name="x" select="."/>  
  <lva>  
    <xsl:copy-of select="titel"/>  
    <xsl:copy-of  
      select="//mitarbeiter/veranstaltung[@nr=$x/@nr]/../name"/>  
  </lva>  
</xsl:for-each>
```

Parameter

- Definition/Parameter-Bindung:
 - Mittels `xsl:param` Element
 - Bindung exakt wie bei Variablen.
 - Zugewiesener Wert bei Parameter ist nur der Default für den Fall, dass kein anderer Parameterwert übergeben wird.
- Parameter-Übergabe:
 - Globaler Parameter: von XSLT-Prozessor versorgt
 - Lokaler Parameter in benannten Templates: `xsl:with-param`
- Beispiel:

```
<xsl:template name="vielfaerbig">
  <xsl:param name="farbe">green</xsl:param>
  <font color="{ $farbe }"><xsl:value-of select="."/></font>
</xsl:template>

...<xsl:call-template name="vielfaerbig">
  <xsl:with-param name="farbe">red</xsl:with-param>
</xsl:call-template>
```



8.7. weitere Features

- Sortieren
- document() Funktion
- weitere Funktionen

xsl:sort

- **xsl:sort:**
 - als Kind von **xsl:for-each** oder **xsl:apply-templates**
 - bewirkt Veränderung der Ordnung der selektierten Knoten
 - Mehrere **xsl:sort** Elemente können aufeinander folgen
=> primäre, sekundäre, ... Sortierung
- Attribute von **xsl:sort**
 - **select:** string-expression als Sortier-Schlüssel
 - **data-type:** text, number
 - **order:** ascending, descending
 - **case-order:** lower-first, upper-first (d.h.: ob Groß- oder Kleinbuchstaben Vorrang haben)

Beispiel

- `xsl:sort` und `xsl:for-each`:

```
<xsl:for-each select="//mitarbeiter">  
  <xsl:sort select="name" order="ascending"/>  
  <xsl:copy-of select="."/>  
</xsl:for-each>
```

- `xsl:sort` und `xsl:apply-templates`:

```
<xsl:apply-templates select="employee">  
  <xsl:sort select="name/family"/>  
  <xsl:sort select="name/given"/>  
</xsl:apply-templates>
```



document()

- *node-set* **document** (*object*, *node-set*?)
erlaubt den Zugriff auf Knoten(mengen) mehrerer Dokumente
- Parameter beim Aufruf von document():
 - `document(string)`
string = URI des zusätzlichen Input-Dokuments,
Spezialfall: `string=""` referenziert das XSLT-Stylesheet;
Ergebnis = root node des Dokuments
 - `document(string, node-set)`
string = URI des zusätzlichen Input-Dokuments;
Ergebnis = Knotenmenge, die mit dem **node-set** Ausdruck im
zusätzlichen Input-Dokument selektiert wird
- Ordnung von Knoten unterschiedlicher Dokumente:
 - prozessorabhängig

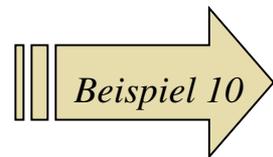
document()

■ Default-Dokument:

- Man kann mittels document() Funktion auf beliebig viele "weitere" Dokumente zugreifen.
- Man braucht aber für den Aufruf des XSLT-Prozessors immer ein default-Dokument (kann auch ein "dummy" Dokument sein, das gar keinen Einfluss auf das Ergebnis hat).

■ Beispiel:

```
<xsl:variable name="emps" select="document('merge2.xml')"/>
<xsl:template match="/">
  <employees>
    <xsl:for-each select="$emps/employees/employee">
      <xsl:copy-of select="."/>
    </xsl:for-each>
  </employees>
</xsl:template>
```



Weitere Funktionen

■ *node-set* **current** ()

- Auswahl des current node
- auf äußerster Ebene wie ".", z.B.: `<xsl:value-of select="."/>`
ist dasselbe wie `<xsl:value-of select="current()"/>`
- Typische Anwendung: Joins, z.B.
`veranstaltung[@nr=current()]/@nr]`

■ *string* **generate-id**(*node-set*?)

- erstellt eindeutige id für den ersten Knoten in der Knotenmenge
- Wenn Argument fehlt, wird die id für den current node berechnet
- Die id ist prozessorabhängig, muss aber für einen best. Knoten bei wiederholten Aufrufen von generate-id immer dieselbe id sein!
- Beispiel: Hyperlinks in einem HTML-Dokument:

Definition des Ankers: ``

Definition eines Links: `...`

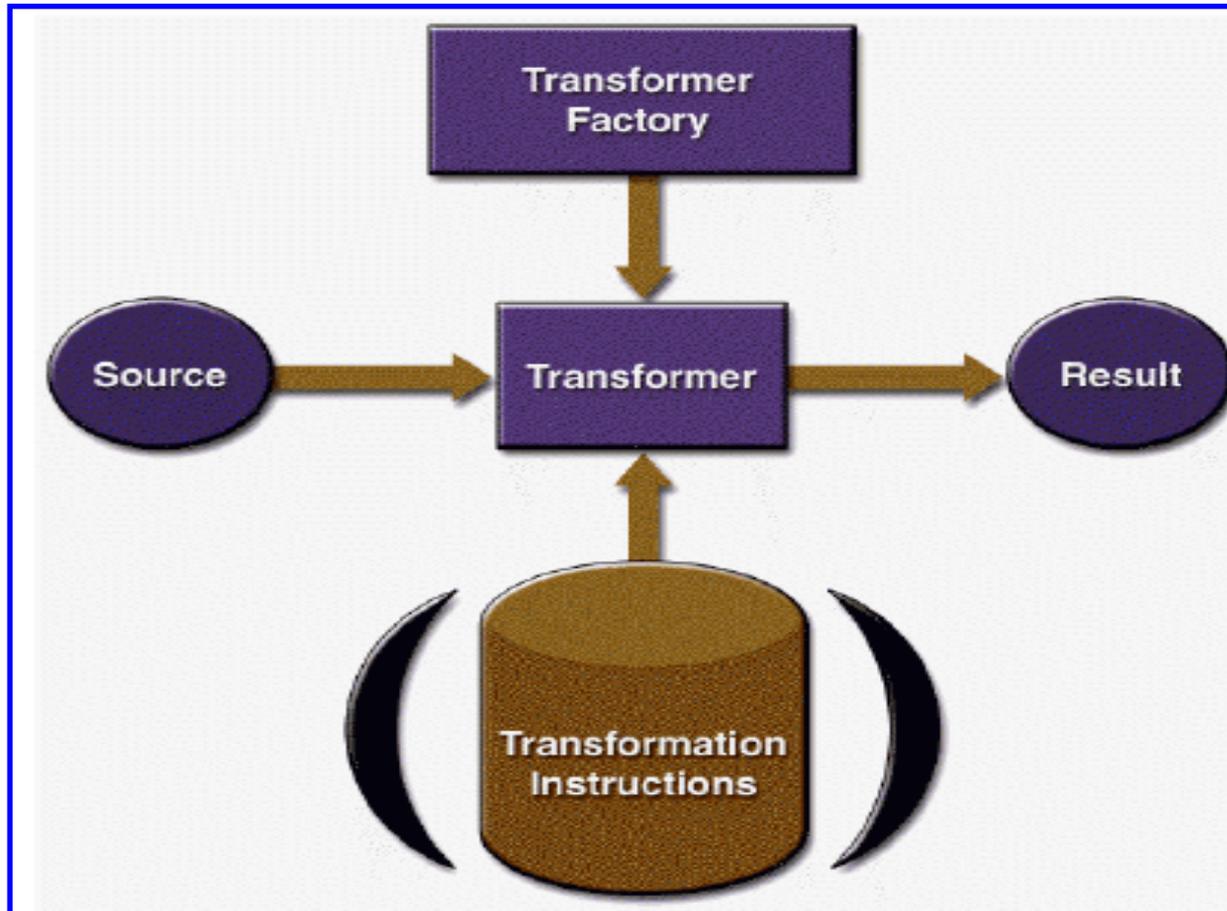
8.8. Java-API TrAX

- Überblick
- Packages von TrAX
- Verwendung von TrAX
- DOM- und SAX-Input
- XML-Filter

Überblick

- TrAX: Transformations API for XML
- Teil von JAXP (und somit von JDK ab Version 1.4)
- Die wichtigsten Klassen bzw. Interfaces
 - TransformerFactory: wählt Implementierung aus und erzeugt neuen Transformer mittels XSLT Stylesheet
 - Transformer: die eigentliche Transformation
 - Source: Datei, SAX, DOM
 - Result: Datei, SAX, DOM

Überblick



Packages von TrAX

- `javax.xml.transform`:
Enthält die Klassen `TransformerFactory`, `Transformer`, `Source`, `Result`:
- `javax.xml.transform.dom`:
Enthält die Klassen `DOMSource` und `DOMResult` (= DOM-spezifische Implementierung der Interfaces `Source` und `Result`)
- `javax.xml.transform.sax`:
Enthält die Klassen `SAXSource` und `SAXResult`
- `javax.xml.transform.stream`:
Enthält die Klassen `StreamSource` und `StreamResult`

Mindest-Code

■ Importe:

```
import javax.xml.transform.*;  
import javax.xml.transform.stream.*;
```

■ Erzeugung der Sources und Results:

```
Source source = new StreamSource(in);  
Result result = new StreamResult(out);  
Source xslsource = new StreamSource(xsl);
```

■ Factory/Transformer-Instanzierung, Transformation:

```
TransformerFactory tFactory = TransformerFactory.newInstance();  
Transformer transformer = tFactory.newTransformer(xslsource);  
transformer.transform(source, result);
```

DOM- und SAX-Input

■ DOM:

- DOMSource ist eine der 3 möglichen Implementierungen des Source-Interface; package javax.xml.transform.dom
- Üblicherweise: `Source source = new DOMSource(doc);`
- Ebenso möglich: DOMSource aus einem Sub-Tree des DOM-Baums erzeugen, d.h.: `Source source = new DOMSource(node);`

■ SAX:

- SAXSource ist eine der 3 möglichen Implementierungen des Source-Interface; package javax.xml.transform.sax
- Erzeugung der SAXSource: benötigt InputSource und Reader:
`inputSource isource = new InputSource(in);`
`SAXSource ssource = new SAXSource(reader, isource);`

XML-Filter

- Erzeugung eines XML-Filters mittels XSLT-Stylesheet:
 - Dazu ist eine SAXTransformerFactory (d.h.: Subclass von TransformerFactory) erforderlich.
 - Aber mit `TransformerFactory.newInstance()`; wird in Wirklichkeit ohnehin eine SAXTransformerFactory erzeugt => Cast auf SAXTransformerFactory genügt.
 - Mit `newXMLFilter()`; anstelle von `newTransformer()`; wird der XML-Filter erzeugt.
- Beispiel:
 - SAX-Parser: Liest XML-Dokument
 - XML-Filter1: konsumiert Events des Parsers
 - XML-Filter2: konsumiert Events des XML-Filter1
 - Transformer: Wandelt den XML-Output des XML-Filter2 (als SAX-Events) in einen Stream um, der auf Datei geschrieben wird.

