Advanced Studies in IT
CT433

Languages for Data Integration of Semi-Structured Data I – Advanced XML: XML Schema, XPath, XSLT: Lecture 3.

sfi science foundation ireland fondúireacht eolaíochta éireann

National University of Ireland, Galway
Ollscoil na hÉireann, Gaillimh

Overview

- Last time: Basics on XML, DTDs

Today:

- **XML Schema**
  - More precise means to describe an XML Grammar than DTDs
- XPath
  - A simple Query language for XML
- XSLT
  - A language to transform between XML formats.

XML Schema

- Another way to describe the structure of an XML document and validate it.
- You use XML Schema similar to DTDs to specify:
  - The allowed **structure** of an XML document
  - The allowed **data types** contained in one (or several) XML documents

DTDs have very limited typing, missing:

- **Cardinalities** in DTDs hardly expressable… at least very inconvenient.
- Reuse, inheritance of **types** is missing, etc.
- Typing, Datatypes…

  … All this is provided in XML Schema!

XML-Schema : Why use schemas instead of DTDs?

- XML Schema Documents (XSDs) themselves are XML documents and therefore use the same syntax and can themselves be validated with schemas! (whereas DTDs are somehow SGML “legacy”)

- XML-schema has more powerful possibilites to define custom **datatypes** (regular expressions, inheritance, cardinalities etc.)

- XML-schema allows to **reuse** element and attribute definitions (by reference)

- XML-schema **uses XML namespaces** (allows to refer to and prescribe certain namespaces)

- *XML-schema allows to define full context-free grammars for expressing arbitrary XML structures!*

- (BTW: Schemas may also be combined with DTDs)

XML-Schema : A Simple XML-Schema


```
<?xml version="1.0" encoding="UTF-8"?>
<marketing
    xmlns:xsi = "http://www.w3c.org/2001/XMLSchema-Instance"
    xsi:noNameSpaceSchemaLocation = "http://www.Dot.com/mySchema.xsd">

  <employee>Gustav Sielmann</employee>
  <employee>Arnold Rummer</employee>
  <employee>Johann Neumeier</employee>

</marketing>
```

Instance

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd = "http://www.w3c.org/2001/XMLSchema">

<xsd:element name = "marketing>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "employee" type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Schema Definition

General features of XSD (XML Schema Definition)

- XML Schema is always a separate, i.e. **external** entity (file)
  - No internal Schema within the XML-instance
- Schemalink via **attribute in Document Element**
  - **schemaLocation** attribute (for Schema with a *targetNamespace*)
  - **noNameSpaceSchemaLocation** (for schema with no *targetNamespace*)

- Schemata can be embedded:
  - <include>, <redefine>, <import>
  - import if different namespaces

    <xsd:import schemaLocation=" http://www.w3.org/1999/xlink xlink.xsd"/>

    with <redefine> single elements can be redefined e.g.
    you can write a second schema which redefines the first one:

    <xsd:redefine schemaLocation="...">
    <xsd:complexType ...../>

    which restricts or extends the type with the same name from the original schema

XSD and namespaces:

- XML Schema
  - *uses namespaces itself* - to distinguish schema instructions from the language we are describing
  - *supports namespace assigning* - by associating a **target namespace** to the **language** we are describing.
- Example:

```
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        xmlns:b="http://businesscard.org"
        targetNamespace="http://businesscard.org">

  <element name="card" type="b:card_type"/>
  ...

  <complexType name="card_type">
    <sequence>
      <element ref="b:name"/>
      ...
    </sequence>
  </complexType>

  ...
</schema>
```

Here:

- the **default** namespace is that of XML Schema (such that e.g. complexType is considered an XML Schema element)
- the **target** namespace is our business card namespace
- the b prefix also denotes our business card namespace (such that we can refer to target language constructs from within the schema)

XSD structure

An XSD is composed of:

- The Schema Element
- Element Definitions
- Attribute Definitions
- Type Definitions
- Annotations

XSD: The Schema Element

- The schema element is the container element where all elements, attributes and data types contained in an XML document are stored
- The schema element refers to the XML-schema definition at W3C and is the **document element (root)** of an XSD:

```
<xsd:schema xmlns:xsd = "http://www.w3c.org/2001/XMLSchema">
          .
          .
          .
          .
          .
</xsd:schema>
```

XSD: The Schema Element

Can have several attributes:

```
<?xml version="1.0"?>
<xsd:schema
 xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.w3schools.com"
  xmlns="http://www.w3schools.com"
  elementFormDefault="qualified">
...
</xsd:schema>
```

Elements and data types used in an XML schema document (schema, element, complexType, sequence, string, boolean, etc.) come from the "http://www.w3.org/2001/XMLSchema" namespace, and must be prefixed with xsd:

XSD: The Schema Element

Can have several attributes:

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.w3schools.com"
  xmlns="http://www.w3schools.com"
  elementFormDefault="qualified">
...
</xsd:schema>
```

The language defined by this schema (note, to, from, heading, body.) is supposed to be in the "http://www.w3schools.com" namespace.

XSD: The Schema Element

Can have several attributes:

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.w3schools.com"
  xmlns="http://www.w3schools.com"
  elementFormDefault="qualified">
...
</xsd:schema>
```

default namespace is "http://www.w3schools.com". Usually makes sense to have the same def.ns and targetns…

XSD: The Schema Element

Can have several attributes:

```
<?xml version="1.0"?>
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.w3schools.com"
  xmlns="http://www.w3schools.com"
  elementFormDefault="qualified">
...
</xsd:schema>
```

Any elements used by an **XML instance document** which were declared in this schema must be namespace qualified by the target namespace (analogous: attributeFormDefault), i.e. the language defined here is bound to the namespace. alternative: "unqualified"

How to reference XSD inside an instance document:

In the instance file:

```
<?xml version="1.0"?> <note
xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com
note.xsd">
```

Again default namespace, (this is now what was the targetns of the schema)

XML Schema Instance namespace

The schemaLocation attribute has two values. The first value is the namespace URI to use. The second value is the location of the XML schema to use for that namespace

How to reference XSD

- XSD can make use of namespaces!

  `schemaLocation` attribute in instance

  (value: pair namespaces-xsd)
- w/o targetNamespace:

  `noNamespaceSchemaLocation` attribute in instance

… sophisticated… don't bother too much for the moment. Find more in the references:

e.g.

http://www.w3.org/XML/Schema (official specification)
http://ww.w3schools.com (3rd party tutorial)

XML-Schema : Defining Elements

Elements

- Must have a name and a type (attributes)
- Either globally defined: as child of <schema>
- or locally defined: in context of other elements

  (can also have the same name as globally defined)
- Elements can refer to other (global) element or type definitions
- Elements can have an associated cardinality
- Complex or simple type:
  - simple: (refinements of) built-in types (e.g. strings, numbers, dates, etc.)
  - complex: can have nested elements, etc.

Example for a simple type element referring to a built-in type:

```
<xsd:element name = "employee" type = "xsd:string"/>
```

Example for a complex type element which does not refer to a global or built-in type declaration:

```
<xsd:element name = "marketing">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name = "employee" type = "xsd:string"
                   maxOccurs = "unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Element order in XSD

- In general, there is no preferred order within a schema document – you generally have to infer the "containing" element of a schema based upon which <xsd:element> references the highest elements in the tree.

  (unlike DTDs where you strictly define the document element)

XML-Schema : Defining Attributes

- Like elements, attributes must have a name and type
- Attributes can use custom data types (but **only simple types***!)
- Attributes can be restricted similar like in DTDs (`optional, fixed, required, default` values)

```
<xsd:attribute name = "country" type = "xsd:string" fixed="Austria"/>
```

```
<xsd:attribute ref="xml:lang" use="optional" />
```

- Attributes can refer to other (global) attribute definitions:

```
<xsd:attribute name="partNum" type="ipo:SKU" use="required"/>
```

* More on what exactly are simple types in the following slides….

XML-Schema : Annotations

- XML-schema provides several tags for annotating a schema:
  `documentation` (intended for human readers),
  `appInfo` (intended for applications) and `annotation`
- `Documentation` and `appInfo` usually appear as subelements of `annotation`

```
<xsd:annotation>
  <xsd:documentation xml:lang = "en">
    here goes the documentation text for the schema
  </xsd:documentation>
</xsd:annotation>
```

- *Remark: Different from normal comments in XML, since these annotations can be used in e.g. XML Schema editors, etc., difference is mainly conceptually*

XML-Schema : Data Types

XML-schema data types are either:

- Pre-built Simple Types
- Derived from Simple Types
- Complex Types

XML-Schema : Simple Types

- Simple types are elements that contain **data**
- Simple types **may not** contain attributes or sub-elements
- New simple types are defined by deriving them from built-in simple types

```
<xsd:simpleType name = "mySimpleDayOfMonth">
  <xsd:restriction base = "xsd:positiveInteger">
    <!--positiveInteger defines the minimum to be 1-->
    <xsd:maxInclusion value = "31"/>
  </xsd:restriction>
</xsd:simpleType>
```

XML-Schema : Some Built-in Simple Types

Simple Type	Example
xsd:string	Man, this day is long!
xsd:hexBinary	0FB7
xsd:integer	-12834, 0, 235
xsd:decimal	-1.23, 0, 5.256
xsd:boolean	TRUE, FALSE, 0, 1
xsd:date	1977-10-03
xsd:anyURI	http://www.Dot.com/my.html#a3

XML-Schema : Complex Types

- Complex types are elements that allow sub-elements and/or attributes
- Complex types are defined by listing the elements and/or attributes nested within
- Complex types are used if one wants to define sequences, groups or choices of elements

```
<xsd:complexType name = "myAdressType">
  <xsd:sequence>
    <xsd:element name = "Name" type = "xsd:string">
    <xsd:element name = "Email" type = "xsd:string">
    <xsd:element name = "Tel" type = "xsd:string">
  </xsd:sequence>
</xsd:complexType>
```

XML-Schema : Complex Types

In complex types elements can be combined using the following constructs:

- **Sequence:** all the named elements must appear in the sequence listed
- **Choice:** one and only one of the elements must appear
- **All:** all the named elements must appear, but in no specific order
- **Group:** collection of elements, usually used to refer to a common group of elements (only usable for global declarations and for references!), groups other sequences, choices or alls, for reuse.

XML-Schema : Complex Types


```
<xsd:complexType name = "myAdressType">
  <xsd:sequence>
    <xsd:element name = "Name" type = "xsd:string">
    <xsd:element name = "Email" type = "xsd:string">
    <xsd:element name = "Tel" type = "xsd:string">
  </xsd:sequence>
</xsd:complexType>
```

Cardinality can also be restricted (for choice and sequence!)

```
<xsd:complexType name = "myAtMost2AdressType">
  <xsd:sequence minOccurs=0 maxOccurs=2>
    <xsd:element name = "Name" type = "xsd:string">
    <xsd:element name = "Email" type = "xsd:string">
    <xsd:element name = "Tel" type = "xsd:string">
  </xsd:sequence>
</xsd:complexType>
```

XML-Schema : Mixed, Empty and Any Content

- **Mixed content** is used if you want to model elements that includes both subelements and character data `<xs:complexType mixed="true">`
- **Empty content** is used to define elements that must not include any subelements and character data, implicit in the definition:

```
<xs:element name="product">
    <xs:complexType>
            <xs:attribute name="prodid" type="xs:positiveInteger"/>
     </xs:complexType>
</xs:element>
```

- **Any content** (the most basic data type) does not constrain the content in any way. The <any> element enables us to extend the XML document with elements not specified by the schema:

```
<xs:element name="person">
   <xs:complexType>
            <xs:sequence>
            <xs:element name="firstname" type="xs:string"/>
            <xs:element name="lastname" type="xs:string"/>
            <xs:any minOccurs="0"/>
</xs:element>
```

XML-Schema : Inheritance

- XML-Schema provides a „pseudo“ inheritance via type-derivations
- In XML-Schema all inheritance has to be defined explicitly
- New types can only be created by extending or restricting existing types
- Types can only be derived from one type – multiple inheritance is not supported

XML-Schema : Restricting a Type

- New simple types can be derived by constraining facets of a simple type
- The **XSD:RESTRICTION** element is used to state the base type

```
<xsd:simpleType name = "AustrianPostalCode">

  <xsd:restriction base = "xsd:integer">

    <xsd:minInclusive value = "1000">
    <xsd:maxInclusive value = "9999">

  </xsd:restriction>

</xsd:simpleType>
```

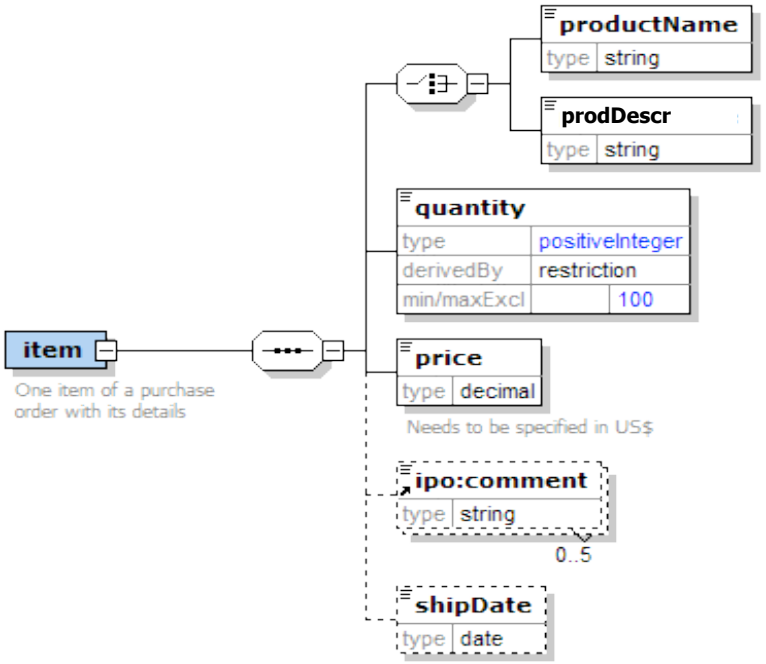
XML-Schema : Extending a Type

- New complex types can be derived by extending other types (simpleContent/complexContent)
- The **XSD:EXTENSION** element is used to state the base type

```
<xsd:complexType name = "myPrice">

  <xsd:simpleContent>

    <xsd:extension base = "xsd:decimal">
      <xsd:attribute name = "currency" type="xsd:string"/>
    </xsd:extension>

  </xsd:simpleContent>

</xsd:complexType>
```

When do you need this? E.g. if you want to attach attributes to a simple type element

Yet another complex example…




```
<element name="item">
  <annotation>
    <documentation>One item of a purchase order with its
                   details</documentation>
  </annotation>
  <complexType>
    <sequence>
      <choice>
        <element name="productName" type="string"/>
        <element name="productDescr" type="string"/>
      </choice>
      <element name="quantity">
        <simpleType>
          <restriction base="positiveInteger">
            <maxExclusive value="100"/>
          </restriction>
        </simpleType>
      </element>
      <element name="price" type="decimal">
        <annotation>
          <documentation>Needs to be specified in
                         US$</documentation>
        </annotation>
      </element>
      <element ref="ipo:comment" minOccurs="0" maxOccurs="5"/>
      <element name="shipDate" type="date" minOccurs="0"/>
    </sequence>
    <attribute name="partNum" type="ipo:Sku"/>
  </complexType>
</element>
```

Inheritance in XSD compared with “real” type systems/inheritance:

- Note that this inheritance is not FULL inheritance in the sense of the semantics defined for OOP or Ontologies, for instance there is no Bottom-up inheritance of instances:

  E.g. element employee is a subclass (restriction/extension) of complextype person, but in XSD no means to say that an employee is a person then…)

These were some but not ALL features of DTDs and XSD…

Some examples and exercices in the zip file I will pt on the lecture homepage might help!

Play around with it and find out more!

Overview

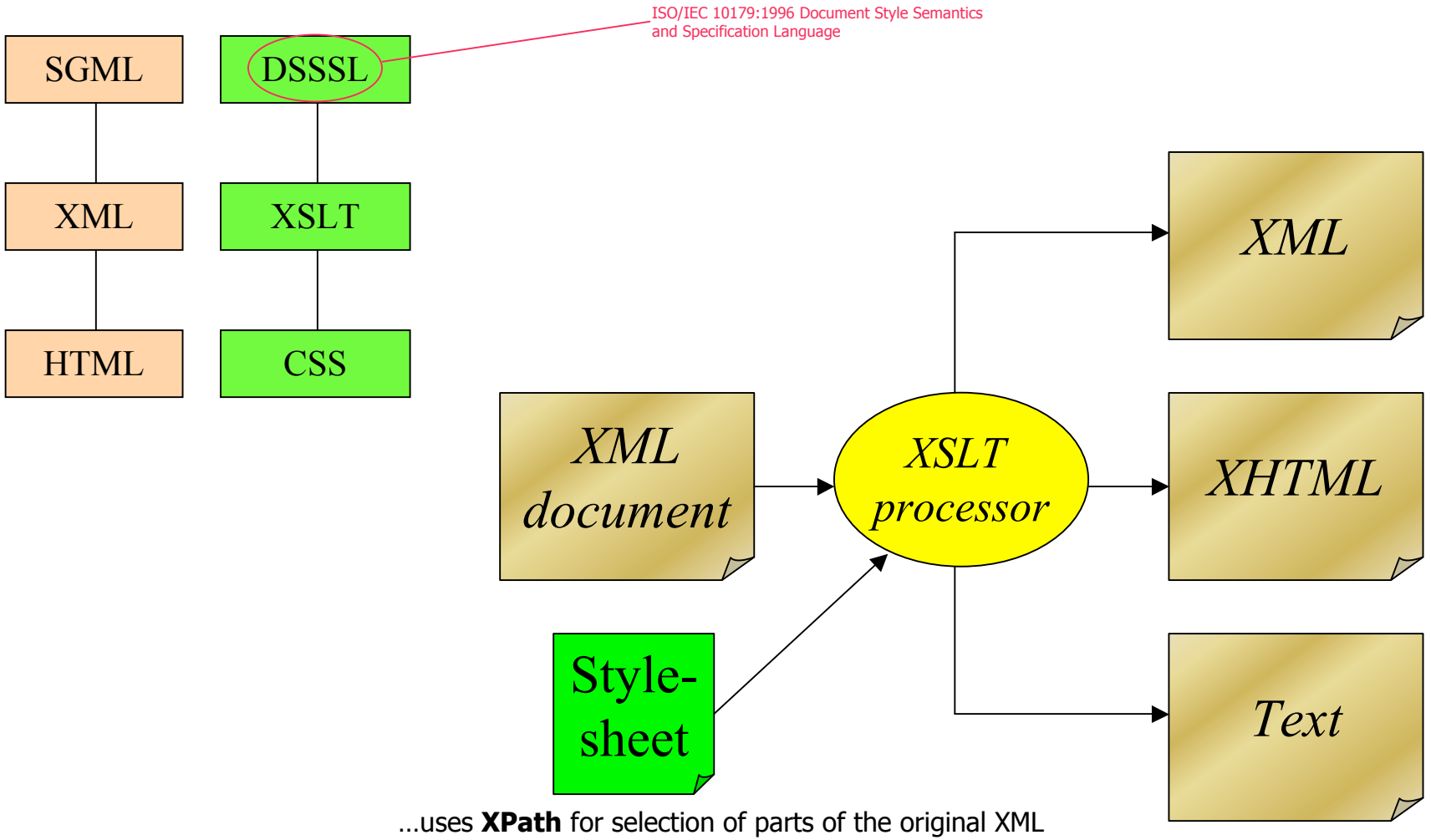
- Last time: Basics on XML, DTDs

Today:

- XML Schema
  - More precise means to describe an XML Grammar than DTDs
- **XPath**
  - A simple Query language for XML
- **XSLT**
  - A language to transform between XML formats.

- XPath: A Language for navigating through and querying/selecting parts of the XML tree
- XSL(T): XML Stylesheet Language (Transformations), define rules for transforming a source tree into a result tree, using XPath to select the parts to transform, also allows combining XML documents
- An application domain on the Web: For the (current) Web we can use tools for generating XML out of existing HTML content to extract information→ *Wrappers*
- *More generally:* Transformations from one XML format into another (XML) format.

XML Stylesheets:



...uses **XPath** for selection of parts of the original XML

The Extensible Stylesheet Language Family (XSL)

Definitions from http://www.w3.org/Style/XSL/

- XSL Transformations (XSLT)
  - a language for transforming XML
- the XML Path Language (XPath)
  - an expression language used by XSLT to access or refer to parts of an XML document. (XPath is also used by the XML Linking specification and the newer Xquery standard.)
- XSL Formatting Objects (XSL-FO)
  - an XML vocabulary for specifying formatting semantics, rather for output formats/rendering than for XML to XML transformations.

Examples:

- In our examples, we will use XSLT mainly to extract from/generate XHTML, but there are lots of more general applications to transform from/to aribtrary (not necessarily XML) formats!

A first example:

Input:

```
<?xml version="1.0" encoding="UTF-8"?>
<lehre>
   <veranstaltung>Telecooperation</veranstaltung>
</lehre>
```

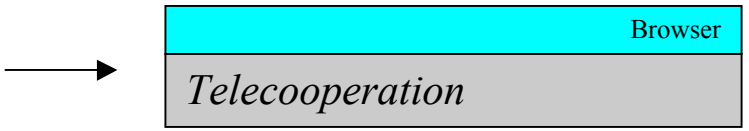
Stylesheet: stylesheets consist of templates:

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

XPath expressions

Result:

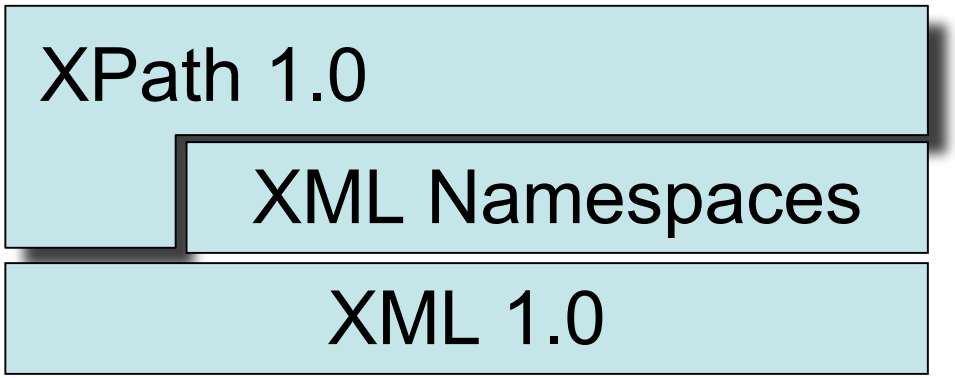
```
<i>Telecooperation</i>
```



→ Files: exa1.xml, exa1.xsl

Before we get to Stylesheets…

XPath:



XPath is a major element in the W3C XSLT standard. Without XPath knowledge you will not be able to create XSLT documents.

XPath:

- XPath is the basis for most XML query languages
  - Selection of document parts
  - Search context: ordered set of nodes
- XPath is being used extensively in XSLT
  - Disadvantage: XPath itself is not XML notation
  - Also basis of XPointer und XQuery
- Navigate through the XML Tree
  - Similar to a filesystem (“/“, “../“, “ ./“, etc.)
  - Query result is the final search context, usually a set of nodes
  - Filters can modify the search context
  - Selection of nodes by
    element names, attribut names, type, content, value, relations
  - several pre-defined functions
- Version 1.0 Recommendation, Version 2.0 Working Draft

XPath:


```
<?xml version="1.0" encoding="UTF-8"?>
<lehre>
  <responsible>
    <title>Dr.</title>
    <name>Ying Ding</name>
  </responsible>
  <veranstaltung sine-tempore="no" year="2003" type="lecture">
    <title>Telecooperation</title>
    <lecturer>Dieter Fensel</lecturer>
    <exam>
      <date>2.7.</date>
      <room>HS A</room>
      <max>150</max>
    </exam>
    <exam>
      <room>SR 13</room>
      <date>2.10.</date>
      <max>20</max>
    </exam>
  </veranstaltung>
  <veranstaltung sine-tempore="no" year="2004" type="lecture">
    <title>Telecooperation</title>
    <lecturer>Axel Polleres</lecturer>
    <exam>
      <date>1.7.</date>
      <room>HS A</room>
      <max>150</max>
    </exam>
  </veranstaltung>
  <veranstaltung sine-tempore="yes" year="2004" type="seminar">
    <title>Next Web Generation</title>
    <lecturer>Dieter Fensel</lecturer>
  </veranstaltung>
</lehre>
```

Select particular parts of this document, some possible queries:

- **Give me all exams of lectures in 2003**
- **All titles**
- **All exams which allow for more than 100 participants**

etc.

All this is (and some more) is expressible in XPath!

XPath: How to reference paths

- "/" … document root (before doc element)
- "/lehre" … selects the document element.

- **Absolute Paths**:
  "/lehre/veranstaltung/title" … returns a set of nodes

(all title nodes which are children of veranstaltung nodes directly below lehre)

- Arbitrary descendands, wildcards:

- "/lehre//datum" … all datum nodes at any depth below lehre
- "/lehre/*/title" … all title nodes at depth 2 below lehre (any node in between).

XPath: How to reference paths

- **Relative paths:** if you do not start the path with "/", the path is viewed relative to the current search context (important for instance in stylesheets, the context where the template is applied):

  `"./"` … refers to the current search context

  `"../"` … refers to the parent node oft the current search context.

`title/..` … selects the parents of title (relative path)

`//date/../*` … selects all elements on the same level as date, in the document (absolute path).

Filters: use Predicates in []

Filters allow us to specify conditions under which a certain part is selected.

Examples:

- `//veranstaltung/exam[2]` … selects only the second `exam` for each `veranstaltung`
- `//veranstaltung[@sine-tempore="yes"]` … selects all `veranstaltung` nodes with the attribute `sine-tempore` set true.
- `//veranstaltung[@sine-tempore]` … selects all `veranstaltung` nodes with the attribute `sine-tempore` defined.

Can be combined with relative paths and comparison operators:

- `//veranstaltung[./exam/max>100]` … selects only the veranstaltung nodes which have an exam with more than 100 participants

XPath Axes

The XPath syntax for a location step is:

```
axisname::nodetest[predicate]
```

- We have used axes already (but hidden):

Abbr.	Meaning	Example
none	child::	`veranstaltung` is short for `child::veranstaltung`
@	attribute::	`veranstaltung[@type="lecture"]` is short for `veranstaltung ::node()[attribute::type="lecture"]`
.	self::node()	`.//exam` is short for `self::node()/descendant-or-self::node()/child::exam`
..	parent::node()	`../exam` is short for `parent::node()/child::exam`
//	/descendant-or-self::node()/	`//exam` is short for `/descendant-or-self::node()/child::exam`

Other axes:

AxisName	Description
ancestor	Contains all ancestors (parent, grandparent, etc.) of the current node
ancestor-or self	Like above, but also contains the current node
attribute	Contains all attributes of the current node
child	Contains children of the current node
descendant, descentant-or self	analogous ancestor, but contains all descendants instead.
following following-sibling	Everything in the document after the current node (in depth-first parsing of the tree) All siblings after the current node
preceding, preceding-sibling	analogous following, following-sibling
self	The current node
namespace	All namespace nodes of the current node

Test your examples:

→ Files:

teaching.xml, teaching2.xml, xpathtest.xsl

Node-Tests:

The XPath syntax for a location step is:

```
axisname::nodetest[predicate]
```

- Nodetests:

Function	Description	Examples
text()	returns all textnodes (could be multiple in case of mixed content)	//title/text() /descendant::text()
node()	Returns the node itself	//exam/self::node()
*	Returns all element nodes	//exam/date/* *(empty set!)* /*/*/self::node() (all nodes at depth 2)

Node Set functions (usable in filter predicates):

The XPath syntax for a location step is:

```
axisname::nodetest[predicate]
```

- In predicates you can use relative paths and node set functions, some of which are:

position()	Position in the node set, more precisely: position wrt. to its sibling!	//@*[position()=2] //exam[position()=1] ***short***: //exam[1]
count(nodeset)	Counts number of elements of the node set	//*[count(./child::node())>3] ***short***: //*[count(./*)>3]
sum(noteset)	Sum of numerical value-node set	//veranstaltung[sum(.//max) > 150]
last()	Returns the position of the last element in the node set	//veranstaltung[title="Telecooperation"]/exam[position()=last()]
local-name(), name(), namespace-uri()	Returns the local part of a node name Returns the full node-name Resolves and returns the namespace-uri	//*[namespace-uri()="http://www.xyz.org"]

Some arithmetics, Boolean & String Functions:

+,-, *, div, mod	Simple Arithmetics	//veranstaltung[@year - 3 = 2000]
=, !=, <, >, <=, >=	Comparison operators, <, >, >=, <= always numberic (no lexicographic comparison)!	//*[count(./child::node())>3]
and, or, not(expression)	Returns the position of the last element in the node set	//veranstaltung[title="Telecooperation“ and @year=2004]
floor(number), ceiling(number), number(value), round(number),		… some more arthimetic functions

Some String functions:

string(value)	String conversion	
normalize-space()	Like trim() in JAVA	
String-length(string)	Returns length	//veranstaltung[string-length(title) >15]
Starts-with(string, substr)	String starts with substring	//veranstaltung[starts-with(title,"Tele")]

… and many more functions, check Standard for the details!

Filter-lists: If multiple filters are specified the semantics depends on order!

//exam[max > 50][1]

choose the first among those exams which have max > 50 for each veranstaltung.*

//exam[1][max > 50]

choose the first exam for each veranstaltung*, but only if max > 50.

Remark: *The position is viewed wrt. the siblings in the tree not wrt. all elements, cf. slide 14. In order to select* ***the first among ALL exam nodes which have max>50****, you have to write something like:* /descendant::exam[max>50][1]

-> Let's try this out!

XPath: Summary

- Simple but powerful query & navigation language for XML trees
- Allows to almost arbitraryly select parts of the XML Tree.
- Many useful built-in functions!

- *Time allowed, we can test some more examples…*

XML Stylesheet Language

- Stylesheet consists of a list of templates, example:

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

- Output not necessarily HTML or XML, could be any text!

- XSLT *Recommendation: Version 1.0, Working Draft: V2.0*

Stylesheets: Building Blocks:

- A list of templates:

```
<xsl:template match="XPATH pattern">
     substitution part
</xsl:template>
```

- A **pattern** is matched with match and in the **substitution part** it is described how it is processed:
- The **substitution part** contains Markup, Text and XSLT elements.

Stylesheets: Building Blocks

- Important elements in the **substitution part**:

  - `</xsl:value-of select="XPath expression">`
    chooses textual value of the XPath expression, left-to-right concatenation.

  - `</xsl:apply-templates [select="XPath expression"]>`
    determines whether and which templates will be further applied at the current position:
    - w/o `select` : all child nodes with all templates
    - with `select` : all matching descendants with all templates
    - only some templates: by use of `call-templates`

Details: see below, slide 31ff.

Stylesheets: How to use

Stylesheet overall structure:

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

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

          <xsl:template …>
                    …
          </xsl:template>
          …
          List of templates
          …
</xsl:stylesheet>
```

Link the stylesheet in a (source) document:

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

Stylesheets: Processing

- Documents are processed depth-first
- Check whether current node matches a template
- Execute template (specific templates first)
- If “apply-templates” then process child nodes, otherwise backtrack

A simple example:


```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:output method="xml" version="1.0" encoding="UTF-8" indent="yes"/>

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

<xsl:template match="test1">
    <test>
       <att1>
          <xsl:value-of select="@att1"/>
       </att1>
       <xsl:apply-templates/>
    </test>
</xsl:template>

<xsl:template match="test2">
   <xsl:element name="test123">
     <xsl:attribute name="att2"><xsl:value-of select="."/></xsl:attribute>
   </xsl:element>
</xsl:template>
</xsl:stylesheet>
```

Create new document element

Make element from attribute

Make attribute from element

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl"
href="test.xslt"?>
<document>
  <test1 att1="AnAttribute">
      <test2>AnElement</test2>
  </test1>
</document>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<testoutput>
  <test>
   <att1>AnAttribute</att1>
   <test123 att2="AnElement"/>
  </test>
</testoutput>
```

XSLT: How to match patterns

<xsl:template match="**XPATH pattern**">

- a pattern is a set of XPath location paths separated by **|** (union)
- restrictions: only the **child** (default) and **attribute** (**@**) axes are allowed here
- extensions: the location paths may start with **id(..)** or **key(..)**

XSLT: Template

There are many different kinds of template constructs:

- literal result fragments
- recursive processing
- computed result fragments
- conditional processing
- sorting
- numbering
- variables and parameters
- keys

Literal result fragments

A literal result fragment is:

- a text constant (character data)
- an element not belonging to the XSL namespace
- **<xsl:text ...> ... </...>** as raw text, but with keeps white-spaces and character escaping control, e.g.:

```
<xsl:text disable-output-escaping="yes">    write &lt; </xsl:text>
```

- **<xsl:comment> ... </...>** (inserts a comment **<!--...-->**)

Recursive processing of templates:


```
<xsl:apply-templates select= "XPath expression" .../>
```
apply pattern matching and template instantiation on selected nodes (default: all children)

```
<xsl:call-template name="..."/>
```
invoke template by name (where xsl:template has name="..." attribute, a named template does not need a match condition)

```
<xsl:for-each select="XPath expression"> template </xsl:for-each >
```
instantiate inlined template for each node in node-set (document order by default)

```
<xsl:copy> template </xsl:copy>
```
copy current node to output and apply template,

(shallow copy, without child nodes, attributes or other content!)

```
<xsl:copy-of select="..."/>
```
copy selected nodes to output (deep copy, includes descendants)

Computed result fragments

Result fragments can be computed using XPath expressions:

<xsl:element name="..." namespace="..."> ... </...>
construct an element with the given name, attributes, and contents

<xsl:attribute name="..." namespace="..."> ... </...>
construct an attribute (This has to occur inside xsl:element !!!)

<xsl:value-of select="..."/>
construct character data or attribute value (expression converted to string)

<xsl:processing-instruction name="..."> ... </...>
construct a processing instruction

The attributes may contain {*expression*}: XPath expressions which are **evaluated** (and converted to string) on instantiation. Example (from slide 8):

```
<xsl:template match="veranstaltung">
 <xsl:element name="{@type}{@year}">
   <xsl:attribute name="titel">
     <xsl:value-of select="title"/>
   </xsl:attribute>
 </xsl:element>
</xsl:template>
```

→

```
<lecture2003 title="Telecooperation"/>
<lecture2004 title="Telecooperation"/>
<seminar2004 title="Next Web Generation"/>
```

Conditional processing

Processing can be conditional:

`<xsl:if test="expression"> ... </xsl:if>`
apply template if expression (converted to boolean) evaluates to true (like in XPath predicates and filters)

```
<xsl:choose>
   <xsl:when test="XPath condition"> ... </...>
   ...
   <xsl:otherwise> ... </...>
</ xsl:choose >
```
test conditions in turn, apply template for the first that is true

```
<xsl:template match="veranstaltung">
 <course>
   <xsl:if test="@year = 2004">
   <xsl:attribute name="thisyear">yes</xsl:attribute>
 </course>
</xsl:template>
```

→

```
<course/>
<course thisYear="yes"/>
<course thisYear="yes"/>
```

Sorting:

`<xsl:sort select="expression" .../>`

a sequence of `xsl:sort` elements placed inside `xsl:apply-templates` or `xsl:for-each` defines a lexicographic order (default: document order)

- Some extra attributes:

```
order="ascending/descending"
lang="..."
data-type="text/number"
case-order="upper-first/lower-first"
```

Example: Sort by lecturer name

```
<xsl:template match="/">
  <teaching>
  <xsl:apply-templates select="./*/veranstaltung">
    <xsl:sort select="lecturer"/>
  </xsl:apply-templates>/>
  </teaching>
</xsl:template>
```

Numbering

- For numbering of lists, sections, items, etc.:

```
<xsl:number      value="expression"          converted to number
                 format="... "               (as ol in HTML, default: "1. ")
                 level="..."                 any / single / multiple
                 count="expression"          select what to count
                 from="..."                  select where to start counting
                 lang="..." letter-value="..."
                 grouping-separator="..." grouping-size="..."/>
```

level="single"	(default) numbering with respect to sibling nodes
level="all"	all nodes of the same type in the tree
level="multiple"	like "all", but numbering respects depth (1.1, 1.1.2, 2.3.4)

```
<xsl:template match="exam">
          <xsl:number count="exam" format="1."/> Exam: <xsl:value-of select="date"/>
</xsl:template>
```

Variables & Parameters

- Rudimentary variable definitions are possible (cannot be updated)
- Global or local in templates

Declaration:

```
<xsl:variable name="..." select="expression"/>
```

variable declaration, value given by XPath expression

or

```
<xsl:variable name="...">
      template
</ xsl:variable >
```

variable declaration, where the value is a result tree fragment.

Refer to variables by $*name*

Very similar: `xsl:param`

`xsl:with-param` allows to pass parameters in `xsl:call-template` and `xsl:apply-templates`

Keys

`<xsl:key match="pattern" name="..." use="node set expression"/>`

declares set of keys - one for each node matching the pattern and for each node in the node set

Extra XPath function:

`key(name expression, value expression)`

returns nodes with given key name and value

Eample: `key('course','T')` returns all `veranstaltung` nodes which title starts with a 'T', with the following key definition:

```
<xsl:key name="course" match="veranstaltung"
use="substring(titel, 1, 1), 'ABCDEFGHIJKLMNOPQRSTUVWXYZ')"/>
```

Another example: Combining XML-documents:

- `document()` … another special XPAth function:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl=
   "http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml"/>

<xsl:variable name="emps" select="document('merge2.xml')"/>

<xsl:template match="/">
   <employees>
   <xsl:for-each select="employees/employee">
      <xsl:copy-of select="."/>
   </xsl:for-each>

   <xsl:for-each select="$emps/employees/employee">
      <xsl:copy-of select="."/>
   </xsl:for-each>
   </employees>
</xsl:template>

</xsl:stylesheet>
```

Attribute-Sets

Globally defined set of attributes:

```
<xsl:attribute-set name="myAtts">
          <xsl:attribute name="href">
                    <xsl:value-of select="@link"/>
          </xsl:attribute>
          <xsl:attribute name="font-size">
                    +1
          </xsl:attribute>
</xsl:template>

<xsl:template match="/url">
          <a xsl:use-attribute-set="myAtts">
                    <xsl:apply-templates/>
          </a>
</xsl:template>
```

Overview over XSL elements:

- xsl:attribute — produces an attribute with a given name
- xsl:use-attribute-set — for attribute-sets
- xsl:value-of — returns text (extraction of source tree or variables)
- xsl:element — produces an element with a given name (e.g. useful if element name produced by variable name)
- xsl:text — produces text node
- xsl:processing-instruction — produce PI
- xsl:comment — produce a comment
- xsl:copy-of — copy the subtree at a certain node
- xsl:copy — copy current context tag
- xsl:if — conditional
- xsl:choose / xsl:when — alternatives
- xsl:number — produce a formated number
- xsl:key — set a key

- xsl:variable — define variables
- xsl:param,xsl:with-param — defune, use parameters

Default-Templates:

The following templates are pre-defined:

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

Guarantees that per default, children are recursively processed without producing a child element.

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

In general, for text nodes write the text out to the result document.

If you want to override these you have to create an empty template, e.g.:
`<xsl:template match="text()|@*"/>`
or you can particularly suppress certain elements by empty templates:
`<xsl:template match=„younot | youneither" />`

XSLT & Browsers & Tools

- All current browsers should contain an XSLT processor.
- Other tools, e.g.
  - Oxygen, XMLSpy
  - Sablotron http://www.gingerall.com/charlie/ga/xml/p_sab.xml
  - Simply xsltproc on the commandline (Linux)
- APIs: e.g. Apache XALAN
  http://xml.apache.org

Other query languages: XQuery

- XML Query (XQuery): More powerful than XPath, W3C recommendation since 23 January 2007 only!
  - derived from an XML query language called Quilt [Quilt], which in turn
  - borrowed features from several other languages, including XPath 1.0 [XPath 1.0], XQL [XQL], XML-QL [XML-QL], SQL [SQL], and OQL [ODMG].

- XCerpt - a more academic one http://www.xcerpt.org/

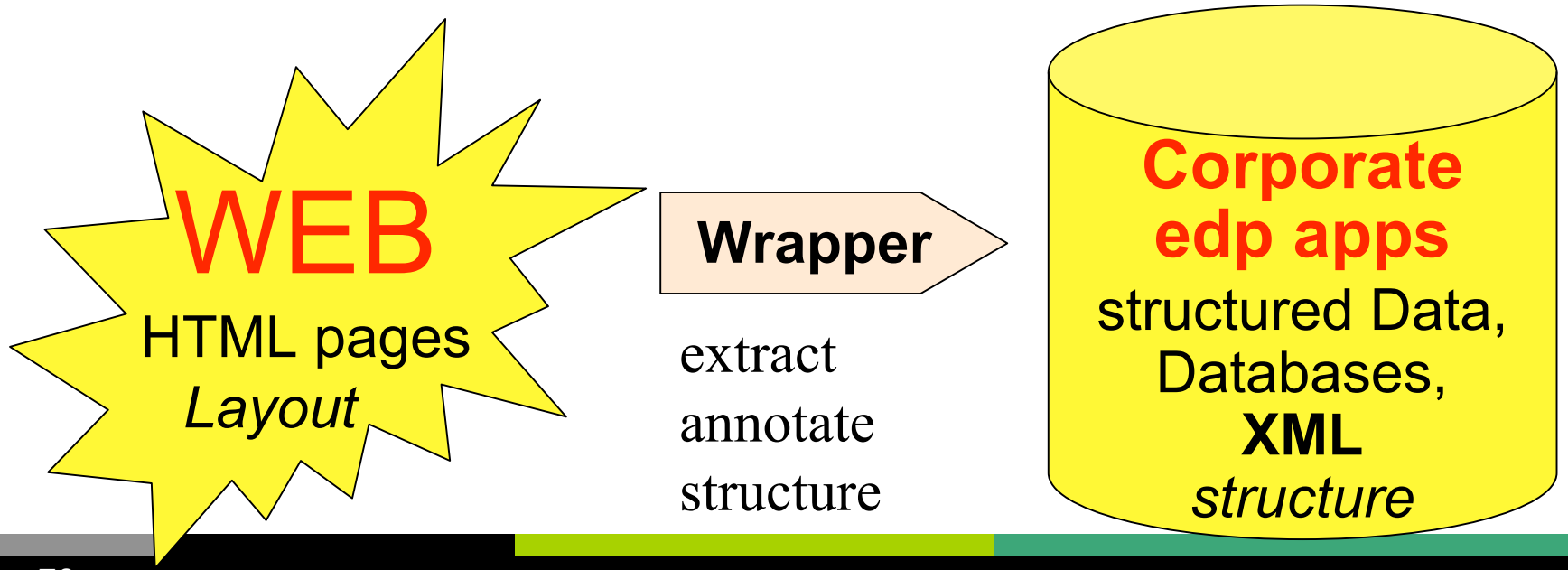
Q: Now how do we get the XML Data to be transformed from current Web or other Data?

A: Wrapper Tools

- A wrapper is an extraction program
- Takes information from Web Pages and turns it into more meaningful structured data, XML.
- Many tools around, cf.
  http://www.wifo.uni-mannheim.de/%7Ekuhlins/wrappertools/

Motivation: Bridge the Gap

Goal: Make Web content accessible for electronic data exchange.



References

- **XPath:** http://www.w3.org/TR/xpath
- **XSLT:** http://www.w3.org/Style/XSL/

Exercises/Assignment:

Will be made available online today, solve it until next time, send to me by mail until March 7th

Next Lecture:

March 10th !!! No lecture next week!!