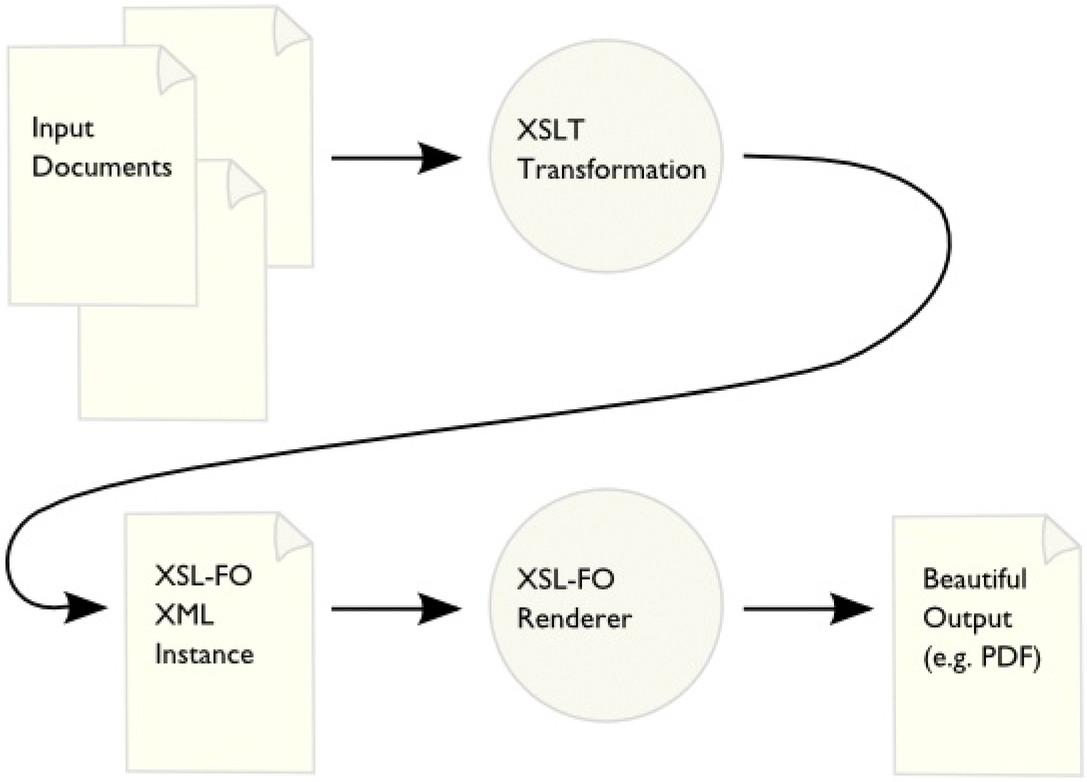
PB138 --Extensible Stylesheet Language Transformation (XSLT)

Extensible Stylesheet Language Transformation (XSLT)

- **XSLT** is a language for specifying transformation of XML documents on the (usually) XML outputs, or text, HTML or other output formats.
- The original application area, the transformation of XML data to XSL:FO (XSL-Formatting Objects), thus rendering XML.
- XSLT specification was therefore part of XSL (eXtensible Stylesheet Language).
- Later, XSL was set aside and XSLT began to be seen as a universal general *declarative language* for XML to XML (text, HTML) transformations.
- XSLT is a Turing-complete language

XSLT Original Purpose

- XML to XSL-Formatting Object transformation

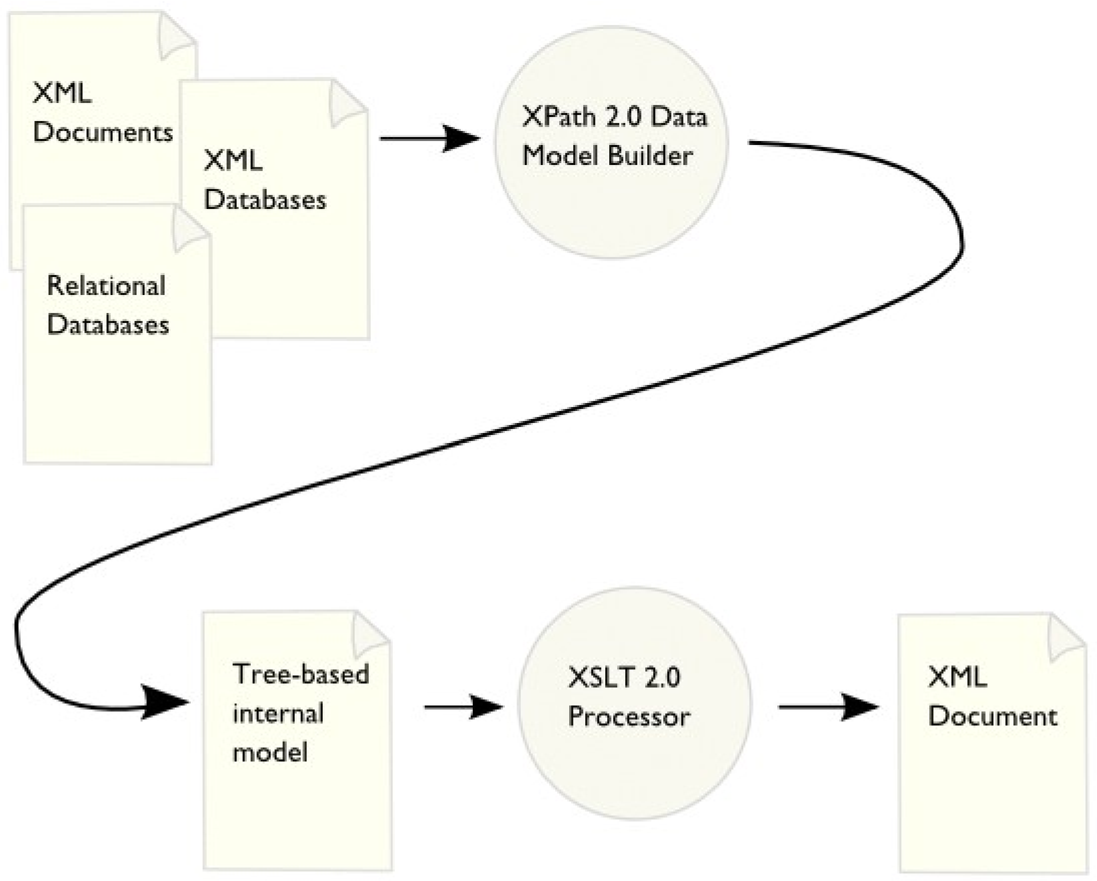


Now the General Goal

- Now since 2.0, the goal is to enable transformations from *any* resources capable of building a XML Document Model from them.
- So XSLT 2.0 processors can operate not only on XML but on anything that can be made to look like XML:
    - relational database tables,

    - geographical information systems,
    - file systems,
    - anything from which your XSLT processor can build an XDM instance.

Now the General Goal



Versions

- The current versions are defined by the XSLT 1.0, 2.0 and 3.0 specifications:
    - XSLT 1.0 XSL Transformations (XSLT) — W3C Recommendation 16 November 1999
    - XSL Transformations (XSLT) Version 2.0 — W3C Recommendation 23 January 2007
    - XSL Transformations (XSLT) Version 3.0 — W3C Last Call Working Draft 2 October 2014
    - The version 1.0 is still widely used, mainly due to lacking (free) implementations of the newer specifications. Also, for many purposes, the features of 1.0 are sufficient.

The main principles

- XSLT is a **functional language**, where reduction rules have the form of templates, which specify how nodes in the source document override output document.
- XSLT transformation specification is contained in a **style file** (**stylesheet**), which is an XML document written in the XSLT syntax. The root element is either xsl:stylesheet or

xsl:transformation (which are synonyms) where xsl: is a prefix for the XSL namespace.

- The XSLT style(sheet) is then processed by an **XSLT processor** and subsequently,
- XML file(s) can be transformed using that stylesheet.

Typical XSLT Workflow

[XSLT workflow] | *../images/XSLT_en.svg*

XSLT style composition

- XSLT stylesheet contains set of **templates**, represented by xsl:template elements.
- Templates have a **selection part** corresponding with the left-hand side of a reduction rule in a functional language and the **construction part** representing the right-hand side of such a rule.

Example: XML Source

(Wikipedia, XSLT)

```
<?xml version="1.0" encoding="UTF-8"?>
<persons>
  <person username="JS1">
    <name>John</name>
    <family-name>Smith</family-name>
  </person>
  <person username="MI1">
     <name>Morka</name>
     <family-name>Ismincius</family-name>
  </person>
</persons>
```

Example: XSLT Stylesheet

(Wikipedia, XSLT)

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
            version="1.0">
  <xsl:output method="xml" indent="yes"/>
  <xsl:template match="/persons">
    <root>
      <xsl:apply-templates select="person"/>
    </root>
  </xsl:template>
  <xsl:template match="person">
    <name username="{@username}">
      <xsl:value-of select="name" />
    </name>
  </xsl:template>
</xsl:stylesheet>
```

Example: Resulting XML

(Wikipedia, XSLT)

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
  <name username="JS1">John</name>
  <name username="MI1">Morka</name>
</root>
```

XSLT Processing

1. XSLT processor (interpreter) takes the *stylesheet* (XSLT code)
2. Usually compiles it into an internal form.
3. Then it takes the nodes from the *input document*, looks for an appropriate template and select it.
4. Then it produces a *result fragment* corresponding to the construction part of the selected template.
5. Recursive takes next nodes from the input document and applies the same procedure for them.

Open-source XSLT Processors

- libxslt: is a free library released under the MIT License that can be reused in commercial applications. It is based on libxml and implemented in C. It can be used at the command line via xsltproc which is included in OS X and many Linux distributions.
- **WebKit**, **Blink**: used for example in the Safari and Chrome web browsers respectively, uses the libxslt library to do XSL transformations.

- **Saxon**: an XSLT (2.0 and partial 3.0) and XQuery 3.0 processor with open-source and proprietary commercial versions for stand-alone operation and for Java, JavaScript and .NET. The open-source version does not support XSLT 3.0.

Open-source XSLT Processors

- **Xalan**: an open source XSLT 1.0 processor from the Apache Software Foundation available stand-alone and for Java and C++. Integrated into Java SE.

  Web browsers

  > Safari, Chrome, Firefox, Opera and Internet Explorer all support XSLT 1.0. None supports XSLT 2.0 natively, although the third party products like *Saxon-CE (Saxon-Client Edition)* and *Frameless* can provide this functionality. Browsers can perform on-the-fly transformations of XML files and display the transformation output in the browser window. This is done either by embedding the XSL in the XML document or by referencing a file containing XSL instructions from the XML document. The latter may not work with Chrome because of its security model.

- **XMLStarlet**: XMLStarlet is a command line XML toolkit which can be used to transform, query, validate, and edit XML documents and files using simple set of shell commands in similar way it is done for plain text files using grep/sed/awk/tr/diff/patch. It does not require Java. Available for Windows and Linux. Supports XSLT 1.0.

Commercial XSLT Processors

- **MSXML** and .NET: includes an XSLT 1.0 processor. From *MSXML 4.0* it includes the command line utility msxsl.exe.
- **QuiXSLT**: an XSLT 3.0 processor doing streaming implemented in Java by Innovimax and INRIA.
- **Saxon**: commercial versions support the newest standards such as XSLT 3.0.

Information Resources

- W3C XSLT 1.0 Recommendation: XSLT 1.0 is still the most used version.
- What is XSLT? on XML.COM: http://www.xml.com/pub/a/2000/08/holman/index.html
- Mulberrytech.com XSLT Quick Reference (2xA4, PDF): http://www.mulberrytech.com/quickref/XSLTquickref.pdf
- Dr. Pawson XSLT FAQ: http://www.dpawson.co.uk/xsl/xslfaq.html
- Zvon XSLT Tutorial: http://zvon.org/xxl/XSLTutorial/Books/Book1/index.html
- Safari online XSLT Reference: https://www.safaribooksonline.com/library/view/xslt-10-pocket/9780596801434/ch04.html

XSLT Syntax

Basic XSLT Elements

'(From http://en.wikipedia.org/wiki/XSLT_elements)'

- xsl:stylesheet: (or xsl:transform) is the top-level element. Occurs only once in a stylesheet document. The attribute version specifies which XSLT version is being used. The NS declaration xmlns:xsl specifies the URL, which is always http://www.w3.org/1999/XSL/Transform regardless of the XSLT version.
- xsl:output: Child element of stylesheet. It describes how data will be returned. The attribute method designates what kind of data is returned (such as xml, text, html). The attribute omit-xml-declaration indicates if the initial <?xml heading should be included. The attribute encoding designates the encoding used for output.
- xsl:template: Specifies processing templates by the attribute “match” - when the template should be used. “name” gives the template a name which xsl:call-template can use to call this template.

Declarations in xsl:stylesheet

- xsl:param: parameter declarations (and their implicit value). Such parameters can then be set when calling XSLT processing, e.g. java net.sf.saxon.Transform -o outfile.xml infile.xml style.xsl -Dparam=paramvalue
- xsl:variable: similarly to parameters, it declares and initializes variables. They however cannot be set from outside. It should also be noted that XSLT (without processor-specific extension) is a pure functional language, i.e. applications of templates do not have side effects → variables (or parameters) can be assigned just once, then just read!

XSLT Templates

Template (xsl:template) is a specification *which node* should be rewriten and how (into what):

- Which node is to be *processed* (i.e. rewritten into output), is described in the attribute match.
- The resulting fragment (into *what* it is rewritten) is stated in the body of the template.
- After processing of the source node, the processing continues at the nodes selected by xsl:apply-templates select=" <*xpath expression*> ".
  - The template can also be explicitly *named* (named template) using the name attribute, in which case it can be called directly / explicitly using xsl:call-template.

Modularization

- xsl:import: Retrieves another XSLT file addressed by the href (URI of the file). The templates in the linking (originating) stylesheet have priority over the imported ones.

- xsl:include: Similarly, but works as a textual (verbatim) include, so no prioritization of the linking stylesheet is done.

Processing modes

- A template can specify a (processing) *mode* is which it can be activated.
- The mode is indicated using the mode attribute at the xsl:template element.
- Processing starts in *no mode* and
- can be switched into another mode by using the attribute mode in the xsl:apply-templates or xsl:call-templates.

Where to use processing modes?

- Motivation: Modes allow a parallel set of templates with the same patterns match, but used for different purposes, for example:
    - one set of templates for generating *table of contents* (index) from the document
    - one for the *full text* of the document itself

Transformation Process in Detail

1. First, the processor selects the root (document node) as the *current node*, i.e. the node corresponding to the XPath expression /
2. Then, it finds the template matching it. If it is found, the template is used for processing this node.
3. Otherwise, a default (implicit) template is used to process the document node.
4. The processing recursively continues at nodes selected by the template that has been used for processing in the previous step.

Template Priority

- If there are two or more template matching, then ambiguity occurs and an error is emmited.
- This situation can be avoided by distinguishing the templates by setting their priorities using their priority attribute. The priority can be an integer, greater number means higher priority.
- Implicit templates have lower priority than explicit ones.

Template Invocation

- **Direct** invocation (call): xsl:call-template (possibly using xsl:with-params)
- With **implicit node selection**: by applying a matching template (again may be with parameters)

using xsl:apply-templates without explicit selection of nodes for further processing. Then, all child elements of the current (context) node will be selected and processed. Equivalent to xsl:apply-templates select="*".

- With **explicit node selection**: when using xsl:apply-templates with explicit selection of nodes for further processing by using the select attribute.
    - In general, the preferred way is to avoid *direct invocation* because the other ways correspond better to the functional nature of the XSLT language.

Outputting text nodes

- Type in text *directly* (as a literal) to output (body part) of the template.
- Be careful with the whitespace characters (spaces, CR/LF) as everything gets into the output.
- When the whitespace handling is important, eg. no unnecessary whitespaces should be produced, use the special element xsl:text.
- Inside of it, whitespaces are always maintained!

Implicit/Default templates

- Purpose: provide at least some standard "fallback" way to process basic structures such as traverse the document tree structure
- to "save typing" frequently used templates (ignoring comments and PI).
- These default templates have low priority and can be overriden by specifying explicit templates the same (or overlapping) match clause.
- The following default templates are implicitly "embedded" in each correct XSLT processor.

Default tree (do-nothing) traversal

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

- Selects any element and the root.
- Produces nothing for it but
- traverses its all child elements.

Default tree (do-nothing) traversal for specified mode

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

- Does the same but only for the specified mode.

Copy text nodes and attributes

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

- Copies text nodes and attributes to the result

Ignore PIs and comments

```
<xsl:template match="processing-instruction()|comment()" />
```

- Ignores (does not include the results of the PI and comments)

Generating values programmatically

- Not only elements, attributes and texts from the source are copied to the output.
- All can be programmatically dynamically generated.

Generation of element with calculated attribute value

- Objective: Generate the output of a predetermined element (with pre-known name), but with attributes with values with calculated during transformation.
- Solution: Use the normal procedure - literal result element - attributes and values specified as the attribute value templates (AVT)

Example

```
<link ref="a_link_href">
   ...
</link>
```

- Template

```
<xsl:template match="link">
   <a href="#{@ref}"> ... </a>
</xsl: template>
```

Explanation

- Transforms the link to a (possibly HTML) a element, the href attribute value is composed of # and the value of the original ref attribute.

  Output

```
<a href="#a_link_href"> ... </a>
```

Generating with calculated element- or attribute name

- Objective: Generate the output element whose name, attributes and content is NOT known in advance when writing the style. So it must be determined (calculated) in runtime (when transforming).
- Solution: Use a template to component xsl:element

Example

- Input:

```
<generate element-name="myelement"> ... </generate>
```

- Template:

```
<xsl:template match="generate">
   <xsl:element name="{@element-name}">
      <xsl:attribute name="id">ID1</xsl:attribute>
   </xsl:element>
</xsl:template>
```

- Result: Creates an element with the name myelement, equipping it with the attribute id="ID1". Also the attribute name could be generated if we wished so.

XSLT Conditional processing

- Objective: To influence the output based on a condition.
- Solution: Use branching in the template - either
    - xsl:if for single *then* / *else* branches or
    - multiway xsl:choose / xsl:when / xsl:otherwise

Example xsl:if

```
<bread price="50"> ... </bread>
```

```
<xsl:template match="bread">
   <p>
      <xsl:if test="@price > 30">
         <span class="expensive">Expensive </span>
      </xsl:if>bread - price <xsl:value-of select="@price"/> CZK</p>
</xsl:template>
```

- Creates an element p with a record about the bread. If the bread was expensive, also the "Expensive" indication is produced.

Example xsl:choose

```
<bread price="12"> ... </bread>
<bread price="19"> ... </bread>
<bread price="30"> ... </bread>
```

```
<xsl:template match="bread">
   <xsl:choose>
      <xsl:when test="@price > 30">
         <span class="expensive">Expensive</span>
      </xsl:when>
      <xsl:when test="@price < 10">
         <span class="strangely-cheap">Suspiciously cheap</span>
      </xsl:when>
      <xsl:otherwise>
         <span class="normal-price">Normal</span>
      </xsl:otherwise>
  </xsl:choose> bread - price <xsl:value-of select="@price"/> CZK
</xsl:template>
```

- Filters out the two extreme price level — normal prices remain for xsl:otherwise.

Loops

```
<grocery>
  <bread price="12"> ... </bread>
  <bread price="19"> ... </bread>
  <bread price="30"> ... </bread>
</grocery>
```

```
<xsl:template match="grocery">
   <xsl:for-each select="bread">
      <p> bread - price <xsl:value-of select="@price"/> CZK </p>
   </xsl:for-each>
</xsl:template>
```

- Result: Creates series of elements p with bread prices.
- Caution: Construction xsl:for-each typically has procedural nature, which is *generally not recommended* for XSLT as it namely gives minimum flexibility to iterate through the contents of a set of nodes — we must know its exact structures beforehand. The style is also more difficult to modify if the structure changes (eg. new or altered element names).

Template calls and parameters

- **Named template declaration**: <xsl:template name="generate_list">. The template may contain declarations of parameters: <xsl:param name="list_format"/> (parameter type is not specified — i.e. dynamic typing)
- **Template call**: using <xsl:call-template name="_atemplatename_"/>
- The call can also specify the parameters if declared at definition:
  - <xsl:with-param name="_parametername_" select="_parametervalueexpression_"/>
  - or <xsl:with-param name="_parametername_">_parametervalue_</xsl:with-param>
- **Default parameter value**: can also be specified using <xsl:param name="_parametername_" select="_defaultvalueexpression_"/>

Automatic (generated) numbering

- Achieved by using xsl:number element
- For either (or both):
  - *counting elements* in input to allow automatic numbering (eg. number book chapters), or *formatting numbers*, eg. writing them in Arabic or Roman numbers.
  - Resembles part of the internationalization support seen in java.text.

   - The autonumbering can also be *multilevel* eg. (sub)chapter numbers like *1.1* etc.

Example - XSLT style

- Plenty of variants shown in XSLT Cookbook - Recipe of the Day

```
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="text"/>
  <xsl:strip-space elements="*"/>
  <xsl:template match="group">
    <xsl:text>Group </xsl:text>
    <xsl:number count="group" level="multiple"/>
    <xsl:text>&#xa;</xsl:text>
    <xsl:apply-templates/>
  </xsl:template>
  <xsl:template match="person">
    <xsl:number count="group" level="multiple" format="1.1.1."/>
    <xsl:number count="person" level="single" format="1 "/>
    <xsl:value-of select="@name"/>
    <xsl:text>&#xa;</xsl:text>
  </xsl:template>
</xsl:stylesheet>
```

Example - XML source

```
<people>
  <group>
    <person name="Al Zehtooney" age="33" sex="m" smoker="no"/>
    <person name="Brad York" age="38" sex="m" smoker="yes"/>
  </group>
  <group>
    <person name="Greg Sutter" age="40" sex="m" smoker="no"/>
    <person name="Harry Rogers" age="37" sex="m" smoker="no"/>
    <group>
      <person name="John Quincy" age="43" sex="m" smoker="yes"/>
      <person name="Kent Peterson" age="31" sex="m" smoker="no"/>
    </group>
    <person name="John Frank" age="24" sex="m" smoker="no"/>
  </group>
</people>
```

Example - text result

```
Group 1
1.1 Al Zehtooney
1.2 Brad York
Group 2
2.1 Greg Sutter
2.2 Harry Rogers
Group 2.1
2.1.1 John Quincy
2.1.2 Kent Peterson
2.3 John Frank
```

Namespace Handling

- XSLT allows to select and produce nodes (elements, attributes) in namespaces.
- However, it has some pitfalls, see Namespaces in XSLT issues
- Multiple namespaces in XSLT
- What can be done with Namespaces in XSLT 2.0
- Avoid Namespace mistakes in XSLT at Developerworks

Where to do XSLT?

- Online (just for fun)
- In all XML professional editors and many programmers' IDE such as *NetBeans*
- Command-line tools, such as xsltproc or xmlstarlet
- From within Java programs using Java Core API (javax.xml.transform package)
- Using specialized tools programmatically (via API) or command-line, such as *Saxon*
- Similarly for other languages, almost all now have XML/XSLT support

Online Tools

- Good for simple try-and-see:
  - Online XSLT engine @Freeformatter.com plus a demo XML input/XSLT/XML output
  - W3Schools online XSLT engine maybe even better :-)
- With XSLT 2.0 support:
  - http://xslttest.appspot.com/

XSLT in NetBeans - steps

- All recent NetBeans version allow to launch XSLT just by:

◦ Opening an XSLT (or source XML) file

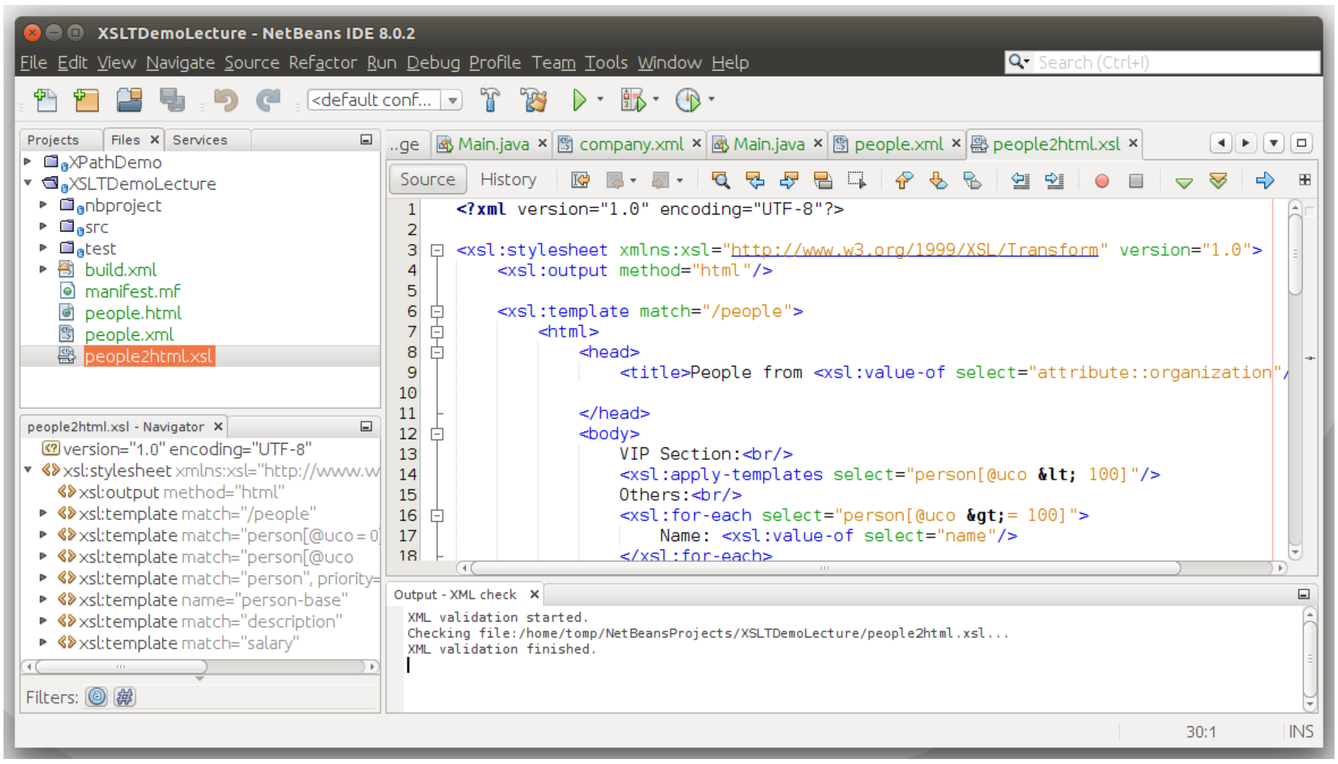
◦ Click on the blue right arrow on the toolbar right

◦ Specify source XML, XSLT, and output files

◦ Run the transformation

◦ Inspect the resulting file directly in the IDE

XSLT in NetBeans - screenshot


XSLTDemoLecture - NetBeans IDE 8.0.2
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
Search (Ctrl+I)
<default conf...
Projects
Files
Services
XPathDemo
XSLTDemoLecture
nbproject
src
test
build.xml
manifest.mf
people.html
people.xml
people2html.xsl
..ge
Main.java
company.xml
Main.java
people.xml
people2html.xsl
Source
History
people2html.xsl - Navigator
version="1.0" encoding="UTF-8"
xsl:stylesheet xmlns:xsl="http://www.w
xsl:output method="html"
xsl:template match="/people"
xsl:template match="person[@uco = 0]
xsl:template match="person[@uco
xsl:template match="person", priority=
xsl:template name="person-base"
xsl:template match="description"
xsl:template match="salary"
Filters:
Output - XML check
30:1
INS

```
<?xml version="1.0" encoding="UTF-8"?>

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

    <xsl:template match="/people">
        <html>
            <head>
                <title>People from <xsl:value-of select="attribute::organization"/

            </head>
            <body>
                VIP Section:<br/>
                <xsl:apply-templates select="person[@uco &lt; 100]"/>
                Others:<br/>
                <xsl:for-each select="person[@uco &gt;= 100]">
                    Name: <xsl:value-of select="name"/>
                </xsl:for-each>
```

```
XML validation started.
Checking file:/home/tomp/NetBeansProjects/XSLTDemoLecture/people2html.xsl...
XML validation finished.
```

Using XSLT in Java (Core API)

- See Using the XSLT Processor for Java from Oracle