

# INFO-H-509 XML TECHNOLOGIES

## Lecture 5: XSLT

---

Stijn Vansummeren

March 20, 2019

## LECTURE OUTLINE

---

1. How XML may be rendered in Web Browsers
2. Syntax and Semantics of XSLT
3. How XPath is used in XSLT

## OUR STORY SO FAR ...

---

- XML is a standard notation for documents and data
- We can select parts of XML with XPath
- We can describe schemas for XML languages by DTDs and XSDs

Often we want to **transform** an XML document into another document:

- Transform the input into (X)HTML for presentation in a browser
- Provide a different view of the input data

## EXAMPLE: PRESENTING A BUSINESS CARD IN A BROWSER

---

### Business Card:

```
<card xmlns="http://businesscard.org">
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

### Firefox:

```
-<card>
  <name>John Doe</name>
  <title>CEO, Widget Inc.</title>
  <email>john.doe@widget.com</email>
  <phone>(202) 555-1414</phone>
  <logo uri="widget.gif"/>
</card>
```

## EXAMPLE: PRESENTING A BUSINESS CARD USING CSS

---

- Add the following processing instruction to card.xml:

```
<?xml-stylesheet type="text/css" href="card.css"?>
```

- Where card.css contains the following cascading stylesheet:

```
card { background-color: #cccccc; border: none; width: 300px; }
name { display: block; font-size: 20pt; margin-left: 0pt; }
title { display: block; margin-left: 20pt; }
email { display: block; font-family: monospace; margin-left: 20pt; }
phone { display: block; margin-left: 20pt; }
```

## EXAMPLE: PRESENTING A BUSINESS CARD USING CSS

---

- Add the following processing instruction to card.xml:

```
<?xml-stylesheet type="text/css" href="card.css"?>
```

- Where card.css contains the following cascading stylesheet:

```
card { background-color: #cccccc; border: none; width: 300px; }
name { display: block; font-size: 20pt; margin-left: 0pt; }
title { display: block; margin-left: 20pt; }
email { display: block; font-family: monospace; margin-left: 20pt; }
phone { display: block; margin-left: 20pt; }
```

**John Doe**

CEO, Widget Inc.  
john.doe@widget.com  
(202) 555-1414

## EXAMPLE: PRESENTING A BUSINESS CARD USING CSS

---

- Add the following processing instruction to card.xml:

```
<?xml-stylesheet type="text/css" href="card.css"?>
```

- Where card.css contains the following cascading stylesheet:

```
card { background-color: #cccccc; border: none; width: 300px; }
name { display: block; font-size: 20pt; margin-left: 0pt; }
title { display: block; margin-left: 20pt; }
email { display: block; font-family: monospace; margin-left: 20pt; }
phone { display: block; margin-left: 20pt; }
```

**John Doe**

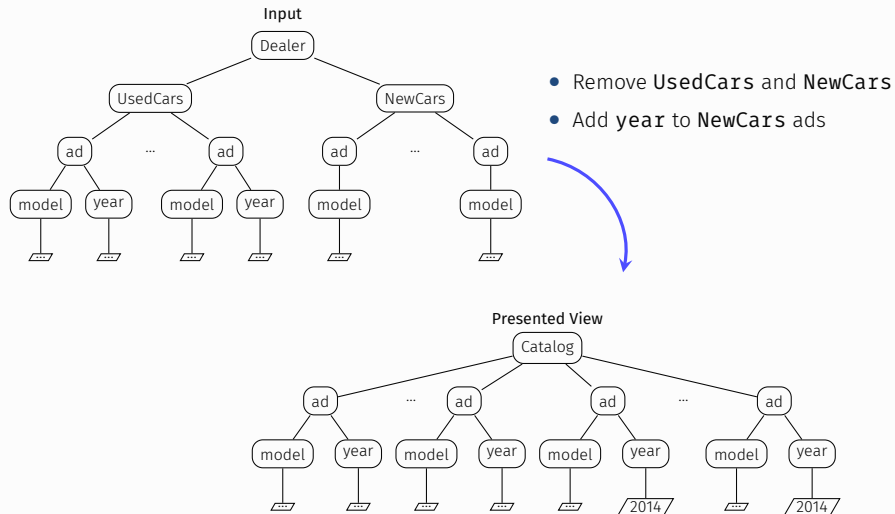
CEO, Widget Inc.  
john.doe@widget.com  
(202) 555-1414

### Limitations:

- Order of information cannot be changed
- Information in attributes (i.e., the logo) cannot be displayed
- We can't add new content

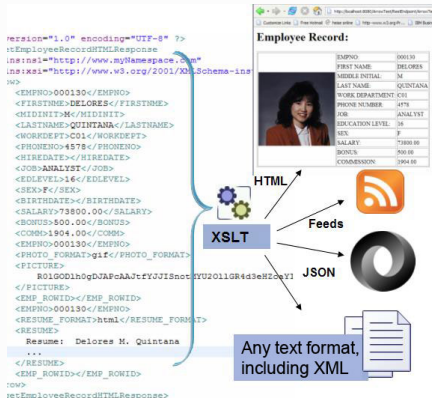
## EXAMPLE: PROVIDING A DIFFERENT VIEW ON INPUT DATA

---



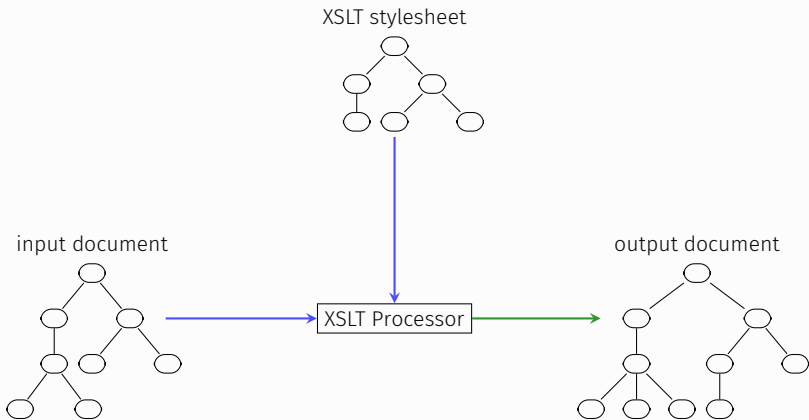


- XSLT is a **domain-specific** language for writing **XML transformations**
- Started out as a generalization of CSS
- But is now a full-fledged programming language



## XSLT: SEMANTIC OVERVIEW

---



## XSLT: SYNTACTIC STRUCTURE

---

- Each XSLT stylesheet is written in XML

### Stylesheet structure:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"  
                version="2.0">  
  template rules  
</xsl:stylesheet>
```

## XSLT: SYNTACTIC STRUCTURE

---

- Each XSLT stylesheet is written in XML

### Stylesheet structure:

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
                version="2.0">
  template rules
</xsl:stylesheet>
```

- Each stylesheet consists of a set of **template rules**

### Template rule structure:

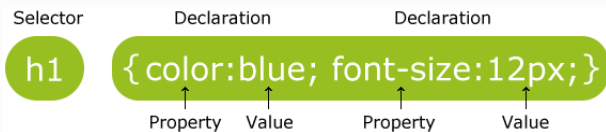
```
<xsl:template match="xpath expression">
  sequence constructor
</xsl:template>
```

## XSLT: TEMPLATE RULES

---

Remember how CSS worked?

- A CSS consists of a sequence of **rules**
- Each rule two parts: a **selector** and one or more **declarations**
- A declaration assigns a **value** to a **property**

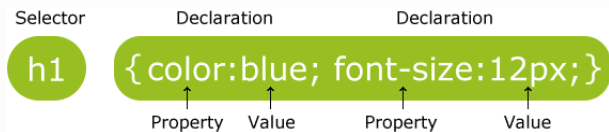


## XSLT: TEMPLATE RULES

---

Remember how CSS worked?

- A CSS consists of a sequence of **rules**
- Each rule two parts: a **selector** and one or more **declarations**
- A declaration assigns a **value** to a **property**



XSLT templates generalize this idea:

```
<xsl:template match="xpath expression">  
  sequence constructor  
</xsl:template>
```

- the XPath expression **selects** the nodes where the rule is **applicable**
- the sequence constructor **constructs the output**

## XSLT: SEQUENCE CONSTRUCTORS

- In its simplest form: an XML fragment to be output, with embedded XSLT commands
- Conceptually similar to e.g. PHP, Java Server Pages, ...

input document

```
<marriage conducted="1862">
  <woman>Salambo</woman>
  <man>Matho</man>
</marriage>
```

template rule

```
<xsl:template match="marriage">
  <b><xsl:value-of select="woman"/></b>
  and
  <b><xsl:value-of select="man"/></b>
  married in
  <xsl:value-of select="@conducted"/>
</xsl:template>
```

output

```
<b>Salombo</b> and <b>Matho</b> married in 1862
```

## XSLT: PROCESSING MODEL

---

- Evaluation always happens with respect to a current **context**, which contains the context node, the context position, the context size, ...
- Initially, the **context node** is set to the root node



- Evaluation always happens with respect to a current **context**, which contains the context node, the context position, the context size, ...
- Initially, the **context node** is set to the root node

### Evaluation algorithm:

1. Find the template rule(s) that **match** the current context node
2. If there is more than one, select the **most specific one**
3. Evaluate the rule's sequence constructor:
  - Output any literally constructed elements
  - All XPath expressions in the sequence constructor are evaluated with respect to the current context node
  - The commands `<xsl:apply-templates>` or `<xsl:call-templates>` change the context node, after which we restart from step 1.

## XSLT: PROCESSING MODEL

---

- Evaluation always happens with respect to a current **context**, which contains the context node, the context position, the context size, ...
- Initially, the **context node** is set to the root node

### Evaluation algorithm:

1. Find the template rule(s) that **match** the current context node
2. If there is more than one, select the **most specific one**
3. Evaluate the rule's sequence constructor:
  - Output any literally constructed elements
  - All XPath expressions in the sequence constructor are evaluated with respect to the current context node
  - The commands `<xsl:apply-templates>` or `<xsl:call-templates>` change the context node, after which we restart from step 1.

Careful: there are some implicitly defined rules!!!!

XPath expressions are used in XSLT to:

- Specify patterns for template rules
- Select nodes for processing
- Compute boolean conditions
- Generate text contents for the output document

1. A general introduction to XSLT
2. **Patterns in detail**
3. Sequence constructors in detail
4. Using XSLT

## PATTERNS IN DETAIL

---

A **pattern** is a restricted XPath expression:

- it is a union of path expressions in abbreviated syntax
- each path expression contains a number of steps separated by / or //
- each step may only use the **child** or **attribute** axes
- Node tests and predicates may use arbitrary XPath

- **Ok** `rcp:recipe/rcp:ingredient//rcp:preparation`
- **Not ok** `rcp:recipe/ancestor::node()/descendant::text()`
- **Ok**  
`rcp:recipe[parent::node()[@id=50]]/rcp:ingredient//rcp:preparation`

## MATCHING IN DETAIL

---

A pattern  $P$  **matches** a node  $n$  if:

- there exists **some** node  $m$  in the input tree
- such that  $n$  is present in the result of evaluating  $P$  starting from  $m$

A template  $T$  with pattern  $P$  matches  $n$  if  $P$  matches  $n$

- The following pattern matches all **preparation** elements that are descendants of some **ingredient** element that is a child of some **recipe** element:

```
rcp:recipe/rcp:ingredient//rcp:preparation
```

## A NOTE ABOUT NAMESPACES

---

During XSLT processing, we normally deal with elements that live in **three different namespaces**:

- The XSLT commands live in the `http://www.w3.org/1999/XSL/Transform` namespace, usually bound to the `xsl` prefix
- The elements/attributes of the input document live in their own namespace
- The elements/attributes of the output document live in the **default** namespace specified

### Stylesheet structure:

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns:rcp="http://www.brics.dk/ixwt/recipes"
    xmlns="http://www.w3.org/1999/xhtml">
  template rules
</xsl:stylesheet>
```

## WHEN MULTIPLE TEMPLATE RULES MATCH

---

When the patterns of multiple template rules match a given node, the **most specific** rule is chosen to be applied.

**The most specific rule is the one that has the highest **priority**:**

- The priority of a rule is computed based on the syntax of the pattern
- The computation itself is a bit complicated (so we won't deal with it here)

**Rule of thumb:**

- “the most specific rule is the one that has the most complicated pattern”



## OVERVIEW

---

1. A general introduction to XSLT
2. Patterns in detail
3. **Sequence constructors in detail**
4. Using XSLT

### Example:

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <html>
      <head>
        <title>Hello World</title>
      </head>
      <body bgcolor="green">
        <b>Hello World</b>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

## EXPLICIT CONSTRUCTORS

---

### Example:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <xsl:element name="html">
      <xsl:element name="head">
        <xsl:element name="title">
          Hello World
        </xsl:element>
      </xsl:element>
      <xsl:element name="body">
        <xsl:attribute name="bgcolor" select="'green'"/>
        <xsl:element name="b">
          Hello World
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

## COMPUTED ATTRIBUTE VALUES (1/2)

---

### Example:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <xsl:element name="html">
      <xsl:element name="head">
        <xsl:element name="title">
          Hello World
        </xsl:element>
      </xsl:element>
      <xsl:element name="body">
        <xsl:attribute name="bgcolor" select="//@bgcolor"/>
        <xsl:element name="b">
          Hello World
        </xsl:element>
      </xsl:element>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

## COMPUTED ATTRIBUTE VALUES (2/2)

---

### Example:

```
<xsl:stylesheet version="2.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
    xmlns="http://www.w3.org/1999/xhtml">
  <xsl:template match="/">
    <html>
      <head>
        <title>Hello World</title>
      </head>
      <body bgcolor="{//@bgcolor}">
        <b>Hello World</b>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

## TEXT CONSTRUCTORS

---

- Literal text becomes character data in the output

```
here is some chardata
```

- Whitespace control requires a constructor:

```
<xsl:text>    </xsl:text>
```

- The (atomized) value of an XPath expression:

```
<xsl:value-of select="//*[@unit]"/>
```

## RECURSIVE APPLICATION

---

Recursive application happens by means of

```
<xsl:apply-templates select="xpath-expr"/>
```

- This evaluates *xpath-expr* within the current context node
- Applies the entire stylesheet to the resulting nodes
- Concatenates the resulting sequences

### Remarks:

- The *xpath-expr* is an **arbitrary** XPath 2.0 expression (not restricted!)
- If the **select** attribute is omitted, the default is **child::node()**
- Processing is often (**but not necessarily!**) a simple recursive traversal down the input XML tree

## RECURSIVE APPLICATION: EXAMPLE (1/2)

---

Input document

```
<students>
  <student id="100026">
    <name">Joe Average</name>
    <age>21</age>
    <major>Biology</major>
    <results>
      <result course="Math 101" grade="C-"/>
      <result course="Biology 101" grade="C+"/>
      <result course="Statistics 101" grade="D"/>
    </results>
  </student>
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```

Desired Output

```
<summary>
  <grades id="100026">
    <grade>C-</grade>
    <grade>C+</grade>
    <grade>D</grade>
  </grades>
  <grades id="100078">
    <grade>A</grade>
    <grade>A-</grade>
    <grade>B+</grade>
    <grade>A</grade>
  </grades>
</summary>
```



## RECURSIVE APPLICATION: EXAMPLE (2/2)

---

### Stylesheet

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <summary>
      <xsl:apply-templates select="student"/>
    </summary>
  </xsl:template>

  <xsl:template match="student">
    <grades>
      <xsl:attribute name="id" select="@id"/>
      <xsl:apply-templates select="./@grade"/>
    </grades>
  </xsl:template>

  <xsl:template match="@grade">
    <grade>
      <xsl:value-of select="."/>
    </grade>
  </xsl:template>
</xsl:stylesheet>
```

## RECURSIVE APPLICATION WITH MODES: EXAMPLE (1/2)

---

Input document

```
<students>
  <student id="100026">
    <name">Joe Average</name>
    <age>21</age>
    <major>Biology</major>
    <results>
      <result course="Math 101" grade="C-"/>
      <result course="Biology 101" grade="C+"/>
      <result course="Statistics 101" grade="D"/>
    </results>
  </student>
  <student id="100078">
    <name>Jack Doe</name>
    <age>18</age>
    <major>Physics</major>
    <major>XML Science</major>
    <results>
      <result course="Math 101" grade="A"/>
      <result course="XML 101" grade="A-"/>
      <result course="Physics 101" grade="B+"/>
      <result course="XML 102" grade="A"/>
    </results>
  </student>
</students>
```

Desired Output

```
<summary>
  <name id="100026">Joe Average</name>
  <name id="100027">Jack Doe</name>
  <grades id="100026">
    <grade>C-</grade>
    <grade>C+</grade>
    <grade>D</grade>
  </grades>
  <grades id="100078">
    <grade>A</grade>
    <grade>A-</grade>
    <grade>B+</grade>
    <grade>A</grade>
  </grades>
</summary>
```

## RECURSIVE APPLICATION WITH MODES: EXAMPLE (2/2)

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <summary>
      <xsl:apply-templates mode="names" select="student"/>
      <xsl:apply-templates mode="grades" select="student"/>
    </summary>
  </xsl:template>

  <xsl:template mode="names" match="student">
    <name>
      <xsl:attribute name="id" select="@id"/>
      <xsl:value-of select="name"/>
    </name>
  </xsl:template>

  <xsl:template mode="grades" match="student">
    <grades>
      <xsl:attribute name="id" select="@id"/>
      <xsl:apply-templates select="./@grade"/>
    </grades>
  </xsl:template>

  <xsl:template match="@grade">
    <grade><xsl:value-of select="."/></grade>
  </xsl:template>
</xsl:stylesheet>
```

Template rules with a **mode** attribute only match when **apply-templates** is called with the correct mode value.

## AN ALTERNATIVE: FOR-LOOP

---

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <summary>
      <xsl:apply-templates select="student"/>
    </summary>
  </xsl:template>

  <xsl:template match="student">
    <grades>
      <xsl:attribute name="id" select="@id"/>
      <xsl:for-each select="//*[@grade]">
        <grade>
          <xsl:value-of select="."/>
        </grade>
      </xsl:for-each>
    </grades>
  </xsl:template>
</xsl:stylesheet>
```

- `<for-each>` first evaluates the XPath expression in the `select` attribute
- for each resulting node it evaluates the body **with that node as context node**

## CONDITIONALS: IF

---

Select all grades different from F

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <summary>
      <xsl:apply-templates select="student"/>
    </summary>
  </xsl:template>

  <xsl:template match="student">
    <grades>
      <xsl:attribute name="id" select="@id"/>
      <xsl:for-each select="//@grade">
        <xsl:if test=". ne 'F'">
          <grade><xsl:value-of select="."/></grade>
        </xsl:if>
      </xsl:for-each>
    </grades>
  </xsl:template>
</xsl:stylesheet>
```

- `<if>` elements first evaluate the XPath expression in their `test` attribute
- The body is evaluated only if the test returns `true` after conversion to a boolean
- There is never an “else” branch

## CONDITIONALS: CHOOSE

---

Show email if available, otherwise phone, otherwise error message

```
<xsl:stylesheet version="2.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
xmlns:b="http://businesscard.org"
  <xsl:template match="b:card">
    <contact>
      <xsl:choose>
        <xsl:when test="b:email">
          <xsl:value-of select="b:email"/>
        </xsl:when>
        <xsl:when test="b:phone">
          <xsl:value-of select="b:phone"/>
        </xsl:when>
        <xsl:otherwise>
          No information available
        </xsl:otherwise>
      </xsl:choose>
    </contact>
  </xsl:template>
</xsl:stylesheet>
```

- each **<when>** branch is tried in order until a **test** evaluates to **true**
- if no branch evaluates to **true**, the **<otherwise>** element is evaluated

## CALLING TEMPLATES BY NAME

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <summary>
      <xsl:apply-templates select="student"/>
    </summary>
  </xsl:template>

  <xsl:template match="student">
    <grades>
      <xsl:attribute name="id" select="@id"/>
      <xsl:for-each select="./@grade">
        <xsl:call-template name="listgrade"/>
      </xsl:for-each>
    </grades>
  </xsl:template>

  <xsl:template name="list-grade">
    <grade><xsl:value-of select="."/></grade>
  </xsl:template>

</xsl:stylesheet>
```

- Template rules can have a `name` attribute **instead** of a `match` attribute
- This allows them to be called explicitly through `<xsl:call-template>` **on the current context node**
- (Think of them as functions)

```
int fib(int n)
{
  if (n <= 1) return 1;

  int f1 = fib(n-1);
  int f2 = fib(n-2);
  return f1 + f2;
}
```

- Remember how we write Fibonacci in Java/C/C++?
- We can do something similar in XSLT using **named templates** and **parameters**



```
<xsl:template name="fib">
  <xsl:param name="n"/>
  <xsl:choose>
    <xsl:when test="$n le 1"> <xsl:value-of select="1"/> </xsl:when>
    <xsl:otherwise>
      <xsl:variable name="f1">
        <xsl:call-template name="fib">
          <xsl:with-param name="n" select="$n -1"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:variable name="f2">
        <xsl:call-template name="fib">
          <xsl:with-param name="n" select="$n -2"/>
        </xsl:call-template>
      </xsl:variable>
      <xsl:value-of select="$f1+$f2"/>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>

<xsl:template match="/">
  <xsl:call-template name="fib">
    <xsl:with-param name="n" select="10"/>
  </xsl:call-template>
</xsl:template>
```

- Template rules can declare **parameters**, which can be assigned a value by the `<xsl:with-param>` subelement of `<xsl:call-template>`
- `<xsl:variable>` elements have a sequence constructor as subelement. They get bound to the result of evaluating that sequence constructor.
- Variables that have already been defined can be used later XPath expressions.

## BUILT-IN TEMPLATE RULES

---

If a node is not matched by any template rule, XSLT applies a **default template rule**:

- text is copied to the output
- otherwise the stylesheet is applied recursively to the children of the context node (with parameters passed on recursively)

This means that the following rules are always implicitly defined

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

<xsl:template match="text()|@" mode="#all">
  <xsl:value-of select="."/>
</xsl:template>

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

<xsl:template match="node()" mode="#all"/>
  <xsl:apply-templates/>
</xsl:template>
```

- But here, parameters are recursively passed on

## SORTING

---

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="students">
    <enrolled>
      <xsl:apply-templates select="student">
        <xsl:sort select="age" data-type="number" order="descending"/>
        <xsl:sort select="name"/>
      </xsl:apply-templates>
    </enrolled>
  </xsl:template>

  <xsl:template match="student">
    <student name="{name}" age="{age}"/>
  </xsl:template>
</xsl:stylesheet>
```

- First sort the nodes returned by the **select** attribute according to (1) age (older first) and (2) name
- Then call **<apply-templates>** for each node **in this sorted order**

Output

```
<enrolled>
  <student name="Joe Average" age="21"/>
  <student name="Jack Doe" age="18"/>
</enrolled>
```

## COPYING NODES

---

- The `<copy>` element make a **shallow** copy: only the node itself is copied, but not its children
- The `<copy-of>` element make a **deep** copy: the node and all of its children are copied

## COPYING NODES

---

- The `<copy>` element make a **shallow** copy: only the node itself is copied, but not its children
- The `<copy-of>` element make a **deep** copy: the node and all of its children are copied

### An identity transformation:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/|@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

## COPYING NODES

---

- The `<copy>` element make a **shallow** copy: only the node itself is copied, but not its children
- The `<copy-of>` element make a **deep** copy: the node and all of its children are copied

### An identity transformation:

```
<xsl:stylesheet version="2.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/|@*|node()">
    <xsl:copy>
      <xsl:apply-templates select="@*|node()"/>
    </xsl:copy>
  </xsl:template>
</xsl:stylesheet>
```

### Example: give top-most HTML lists square bullets:

```
<xsl:template match="ol|ul">
  <xsl:copy>
    <xsl:attribute name="style" select="'list-style-type: square;'/>
    <xsl:copy-of select="*" />
  </xsl:copy>
</xsl:template>
```

## OVERVIEW

---

1. A general introduction to XSLT
2. Patterns in detail
3. Sequence constructors in detail
4. **Using XSLT**

## XSLT 1.0 RESTRICTIONS

---

- Most browsers only support XSLT 1.0
- Can only use XPath 1.0
- There are differences in default priorities
- No sequence values, only **result tree fragments** that can only be output, not subjected to further computation.
- ...





Example  
By means of  
Online demonstration

## BUILDING A NUTRITION TABLE

---

```
<xsl:stylesheet version="2.0"
xmlns:rcp="http://www.brics.dk/ixwt/recipes"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="rcp:collection">
    <nutrition>
      <xsl:apply-templates select="rcp:recipe"/>
    </nutrition>
  </xsl:template>

  <xsl:template match="rcp:recipe">
    <dish name="{rcp:title/text()}"
      calories="{rcp:nutrition/@calories}"
      fat="{rcp:nutrition/@fat}"
      carbohydrates="{rcp:nutrition/@carbohydrates}"
      protein="{rcp:nutrition/@protein}"
      alcohol="{if (rcp:nutrition/@alcohol)
        then rcp:nutrition/@alcohol else '0%'}"/>
  </xsl:template>
</xsl:stylesheet>
```

## BUILDING A NUTRITION TABLE

---

```
<xsl:stylesheet version="2.0"
xmlns:rcp="http://www.brics.dk/ixwt/recipes"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

```
<xsl:template match="rcp:collection">
  <nutrition>
    <xsl:apply-templates select="rcp:recipe"/>
  </nutrition>
</xsl:template>
```

```
<xsl:template match="rcp:recipe">
  <dish name="{rcp:title/text()}"
```

Output

```
<nutrition>
  <dish name="Beef Parmesan with Garlic Angel Hair Pasta"
    calories="1167"
    fat="23%" carbohydrates="45%" protein="32%" alcohol="0%"/>
  <dish name="Ricotta Pie"
    calories="349"
    fat="18%" carbohydrates="64%" protein="18%" alcohol="0%"/>
  ...
</nutrition>
```

## OTHER LANGUAGE FEATURES

---

- Multiple input/output documents
- Numbering
- Functions
- Sequence types
- Dividing a stylesheet into several files
- Stylesheets that generate stylesheets as output

**See the book!**

- XSLT 3.0 was standardized in 2017
- Main modification is support for **streaming** transformations.
- XSLT 3.0 processors can, but need not, support all of XPath 3.1.
- All XSLT 3.0 processors need to support the expressions for manipulating **maps** present in XPath 3.1 (increased JSON support).
- Support for transforming JSON documents by means of the `json-to-xml()` and `xml-to-json()` de-serialization and serialization functions.