

# Semantic Web

## XML Schema

N. Henze, D. Krause, and F. Abel

IVS Semantic Web Group

23.10.2008



## XML Schema (also called XSD: XML Schema Definition)

- Markup language for structuring the content of XML documents
- based on XML: each XML Schema document is written in XML
- allows to define structural constraints on a particular class of XML documents
  - ELEMENTS and ATTRIBUTES that appear in XML documents
  - number and order of CHILD ELEMENTS
  - empty ELEMENTS
  - DATA TYPES
  - DEFAULT VALUES
  - FIXED VALUES
- allows to define new types by *extending* or restricting already existing types
- set of data-types

All XML schema documents are *extensions* of one XML Schema document which can be found at the W3C Website

- From the W3C Website:  
*XML Schemas express shared vocabularies and allow machines to carry out rules made by people. They provide a means for defining the structure, content [...] of XML documents.*
- XML Schema 1.0: Recommendation, October 2004
- Status of XML Schema 1.1: *still under development*

From the W3C Website:  
"http://www.w3.org/2000/10/XMLSchema" (~ 2000 lines)

```
<!-- XML Schema schema for XML Schemas: Part 1: Structures -->
<!-- Note this schema is NOT the normative structures schema. -->
<!-- The prose copy in the structures REC is the normative -->
<!-- version (which shouldn't differ from this one except for -->
<!-- this comment and entity expansions, but just in case -->

<schema targetNamespace="http://www.w3.org/2000/10/XMLSchema"
        blockDefault="#all" elementFormDefault="qualified" version="1.0">
<annotation>
  <documentation>
    Part 1 version: $Id: XMLSchema.xsd,v 1.26 2000/10/23 08:58:09 ht Exp $
    Part 2 version: $Id: datatypes.xsd,v 1.30 2000/10/24 08:04:50 ht Exp $
  </documentation>
</annotation>
<annotation>
  <documentation xml:lang="en" source="http://www.w3.org/TR/2000/WD-xmlschema-1-20000922/structures...">
    The schema corresponding to this document is normative,
    with respect to the syntactic constraints it expresses in the
    XML Schema language. The documentation (within <documentation> elements)
    below, is not normative, but rather highlights important aspects of
    the W3C Recommendation of which this is a part
  </documentation>
</annotation>
```

Each XML schema document begins with an opening tag referring to the *ultimate XML Schema document*:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

## Abbreviation

"xsd" denotes the namespace of the schema. If this prefix is omitted, elements from this namespace are used by default.

Thus it is possible to write

```
<schema
  xmlns="http://www.w3.org/2000/10/XMLSchema"
  version="1.0">
```

In the ultimate XML Schema:

"<http://www.w3.org/2000/10/XMLSchema>"

```
<complexType name="element" abstract="true">
  <annotation>
    <documentation xml:lang="en">
      The element element can be used either at the toplevel to define an
      element-type binding globally, or within a content model to either
      reference a globally-defined element or type or declare an element-type
      binding locally. The ref form is not allowed at the top level.
    </documentation>
  </annotation>
  <complexContent>
    <extension base="annotated">
      <sequence>
        <choice minOccurs="0">
          <element name="simpleType" type="localSimpleType"/>
          <element name="complexType" type="localComplexType"/>
        </choice>
      </sequence>
    </extension>
  </complexContent>
</complexType>
...

```

Minimum for defining an element:

```
<element name="..">
```

Some optional attributes: type, cardinality constraints.

- type specifies a superclass
- minOccurs="x", x any natural number including zero
- maxOccurs="x", x any natural number including zero
- default (no explicit cardinality constraints):  
minOccurs=maxOccurs=1

## Example

```
<element name="author" type="person" minOccurs="1" maxOccurs="1">
```

# XML Schema: Attribute Types

In the ultimate XML Schema: "<http://www.w3.org/2000/10/XMLSchema>"

```
<complexType name="attribute">
  <complexContent>
    <extension base="annotated">
      <sequence>
        <element name="simpleType" minOccurs="0" type="localSimpleType"/>
      </sequence>
      <attributeGroup ref="defRef"/>
      <attribute name="type" type="QName"/>
      <attribute name="use" use="default" value="optional">
        <simpleType>
          <restriction base="NMTOKEN">
            <enumeration value="prohibited"/>
            <enumeration value="optional"/>
            <enumeration value="required"/>
            <enumeration value="default"/>
            <enumeration value="fixed"/>
          </restriction>
        </simpleType>
      </attribute>
    </extension>
  </complexContent>
</complexType>
```

```
<attribute name="value" use="optional" type="string"/>
<attribute name="form" type="formChoice"/>
</extension>
</complexContent>
</complexType>
```

Syntax:

```
<attribute name=".." >
```

Optional attributes:

- *type*
- *use="x"*  $x \in \{optional, required, \dots\}$
- *use="x" value="z"*  $x \in \{default, fixed\}$

## Example

```
<attribute name="id" type="ID" use="required">
```

## Built-in data types

- Numerical data types: *integer, short, byte, long, Float.*
- String data types: *string, ID, IDREF, CDATA, language*
- Date and time: *time, date, month, year*

**User-defined data types:** Simple or Complex data types.

**Complex data types:** Constructed from already existing data types with eventually help of the following elements

- *sequence*: a sequence of existing data type elements (ordered!)
- *all*: a collection of elements where each element must not occur more than once (i.e. minOccurs/maxOccurs can either be 0 or 1)
- *choice*: a collection of elements of which one will be chosen

## Example (ComplexType)

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded" />
    <element name="lastname" type="string" />
  </sequence>
  <attribute name="title" type="string" use="optional" />
</complexType>
```

An element in an XML document that is declared to be of type `lecturerType` may have a `title` attribute, it may also include any number of `firstname` elements, and must include exactly one `lastname` attribute.

Example from "A Semantic Web Primer", G. Antoniou, F. v. Harmelen

# Extending Data Types

Element *extension* with attribute *base*

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded" />
    <element name="lastname" type="string" />
  </sequence>
  <attribute name="title" type="string" use="optional" />
</complexType>
```

## Example (Extension with ComplexType)

```
<complexType name="extendedLecturerType">
  <extension base="lecturerType">
    <sequence>
      <element name="email" type="string"
        minOccurs="0" maxOccurs="1" />
    </sequence>
    <attribute name="rank" type="string" use="required" />
  </extension>
</complexType>
```

# Restricting Data Types

Element *restriction* with attribute *base*

```
<complexType name="lecturerType">
  <sequence>
    <element name="firstname" type="string"
      minOccurs="0" maxOccurs="unbounded" />
    <element name="lastname" type="string" />
  </sequence>
  <attribute name="title" type="string" use="optional">
</complexType>
```

## Example (Restriction with ComplexType)

```
<complexType name="restrictedLecturerType">
  <restriction base="lecturerType">
    <sequence>
      <element name="firstname" type="string"
        minOccurs="1" maxOccurs="2" />
    </sequence>
    <attribute name="title" type="string" use="required">
  </restriction>
</complexType>
```

## Definition (Simple Types)

Simple Types are user-defined types created by restricting built-in types.

## Example (Defining a Simple Type: Restricting the value of a built-in data type)

```
<simpleType name="dayOfMonth">  
  <restriction base="integer">  
    <minInclusive value="1"/>  
    <maxInclusive value="31"/>  
  </restriction>  
</simpleType>
```

Example (Defining a Simple Type: listing all possible values of a built-in type)

```
<simpleType name="dayOfWeek">
  <restriction base="string">
    <enumeration value="Monday"/>
    <enumeration value="Tuesday"/>
    ...
    <enumeration value="Sunday"/>
  </restriction>
</simpleType>
```

Example (Defining the simple type "integer")

```
<simpleType name="integer" id="integer"> ..
  <restriction base="decimal">
    <scale value="0" fixed="true" id="integer.scale"/>
  </restriction>
</simpleType>
```

## Example (Defining an XML Schema for e-Mail)

```
<element name="email" type="emailType"/>
<complexType name="emailType">
  <sequence>
    <element name="head" type="headType" />
    <element name="body" type="bodyType" />
  </sequence>
</complexType>
<complexType name="headType">
  <sequence>
    <element name="from" type="nameAddress" />
    <element name="to" type="nameAddress"
      minOccurs="1" maxOccurs="unbounded" />
    <element name="cc" type="nameAddress"
      minOccurs="0" maxOccurs="unbounded" />
    <element name="subject" type="string" />
  </sequence>
</complexType>
<complexType name="nameAddress">
  <element name="name" type="string" use="optional" />
  <element name="address" type="string" use="required" />
</complexType>
```

## Example (Defining an XML Schema for e-Mail)

```
<complexType name="bodyType">
  <sequence>
    <element name="text" type="string" />
    <element name="attachment" minOccurs="0" maxOccurs="unbounded" >
      <complexType>
        <attribute name="encoding" use="default" value="mime">
          <simpleType>
            <restriction base="string">
              <enumeration value="mime"/>
              <enumeration value="binhex"/>
            </restriction>
          </simpleType>
        </attribute>
        <attribute name="file" type="string" use="required" />
      </complexType>
    </element>
  </sequence>
</complexType>
```

*Example from "A Semantic Web Primer", G. Antoniou, F. v. Harmelen*

## Remark

Some data types are defined separately and given names, while others are defined within other types and defined anonymously (e.g. the types for the attachment element and the encoding attribute). In general, if a type is used once, it makes sense to define it anonymously for local use. However, this approach reaches its limitations quickly if nesting becomes too deep

## The problem

- XML → use of several DTDs or XML schema files
- structuring can be developed independently
- → problem: **name conflicts**: more than one XML schema can define elements of the same name - which schema to apply?

## The solution

- identifying each XML schema or DTD with a prefix – the identifier of the schema / DTD which defines a *space in which names are defined*

## Declaration

```
xmlns:prefix="http://www.example.org/example"
```

## Call

```
prefix:name
```

## Scope

A namespace definition is available within the element it was defined in:

```
<prefix:someElement  
  xmlns:prefix="http://www.example.org/example">  
  <!-- scope of namespace prefix -->  
</prefix:someElement>
```

## Example (Namespaces: Lecturers at a university)

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
  xmlns:vu="http://www.vu.com/empDTD"
  xmlns:gu="http://www.gu.au/empDTD">
  <uky:faculty
    xmlns:uky="http://www.uky.edu/empDTD"
    uky:title="assistant professor"
    uky:name="John Smith"
    uky:department="Computer Science"/>
  <gu:academicStaff
    gu:title="lecturer"
    gu:name="Mate Jones"
    gu:department="Information Technology"/>
</vu:instructors>
```

Omitting the prefix means that the corresponding Namespace is used by default.

## Example (Namespaces: Lecturers at a university II)

```
<?xml version="1.0" encoding="UTF-16"?>
<vu:instructors
  xmlns:vu="http://www.vu.com/empDTD"
  xmlns="http://www.gu.au/empDTD">
  <uky:faculty
    xmlns:uky="http://www.uky.edu/empDTD"
    uky:title="assistant professor"
    uky:name="John Smith"
    uky:department="Computer Science"/>
  <academicStaff
    title="lecturer"
    name="Mate Jones"
    department="Information Technology"/>
</vu:instructors>
```

*Examples derived from "A Semantic Web Primer", G. Antoniou, F. v. Harmelen*

## Example (Book - XMLSchema)

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.publishing.org">
  <xsd:element name="BookCatalogue">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Book" minOccurs="1" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Title" type="xsd:string"/>
              <xsd:element name="Author" type="xsd:string"/>
              <xsd:element name="Date" type="xsd:string"/>
              <xsd:element name="ISBN" type="ISBN-type"/>
              <xsd:element name="Publisher" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

## Example (Book - An XML document using the XMLSchema for Books)

```
<?xml version="1.0"?>
<BookCatalogue xmlns="http://www.publishing.org"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  <Book>
    <Title>Faust</Title>
    <Author>Johann Wolfgang v. Goethe</Author>
    <Date>1999</Date>
    <ISBN>3-406-45214-0</ISBN>
    <Publisher>C.h.Beck'sche Verlagsbuchhandlung</Publisher>
  </Book>
  <Book>
    <Title>G. A. S.</Title>
    <Author>Matt Ruff</Author>
    <Date>2000</Date>
    <ISBN>3-423-12721-X</ISBN>
    <Publisher>DTV</Publisher>
  </Book>
</BookCatalogue>
```

- Stylesheet: style rules that tell a browser how to present a document
- Each rule is made up of a selector (the element on which the style should be applied), and the style to be applied.
- Properties may be defined for an element: Each property takes a value, which together with the property describes how the selector should be presented.

## Syntax:

```
selector { property: value }
```

## Multiple style declarations possible:

```
selector { property1: value1; property2: value2 }
```

## Possibilities to integrate CSS

- external stylesheet file (via @import, or *LINK tag*)
- internal / embedded in the HTML file (via the *STYLE tag* or in HTML tags via the *attribute style*)

## Example

```
BODY                                <-- Selection: denotes for which    -->
                                     <-- HTML this applies;           -->
                                     <-- more than one selection possible -->
{
font-family: "ARIAL"; <-- End each command with a semicolon -->
font-size: 24pt;      <-- default font: Times New Roman      -->
}
```

## Example (Embedding CSS code via the *Style tag*.)

```
<HTML>
  <HEAD>
    <STYLE>
      BODY
      {
        font-family: "ARIAL";
        font-size: 24pt;
      }
    </STYLE>
    <title> Embedding CSS Example </title>
  </HEAD>

  <BODY>
    ...
  </BODY>
</HTML>
```

The *Style tag* has the attribute *media* which allows to define which device should follow the style sheet. Values

- all - all devices
- print - printers
- braille - devices supporting braille
- screen - screen
- speech - devices supporting speech
- tv - television-type devices
- ...

## Example ( Embedding CSS code via the attribute "style" )

```
<HTML>
  <HEAD>
    <title> Embedding CSS Example </title>
  </HEAD>

  <BODY style="font-family: 'Arial'; font-size: 24pt">
    ...
  </BODY>
</HTML>
```

Let mystyle.css contain the following code:

```
BODY
{
  font-family: "Arial";
  font-size: 24pt;
}
```

Example (Embedding CSS code via @import)

```
<HTML>
  <HEAD>
    <STYLE type="text/css">
      @import url(mystyle.css);
    </STYLE>
    <title> Embedding CSS Example </title>
  </HEAD>

  <BODY> ... </BODY>
</HTML>
```

## Example (Embedding CSS code via the *Link* tag)

```
<HTML>
  <HEAD>
    <Link rel=Stylesheet href="mystyle.css"
          type="text/css">
    <title> Embedding CSS Example </title>
  </HEAD>

  <BODY>
    . . .
  </BODY>
</HTML>
```

## Pseudo classes for manipulating the state of a link

:ACTIVE                      :HOVER  
:LINK                         :VISITED

## Further pseudo classes:

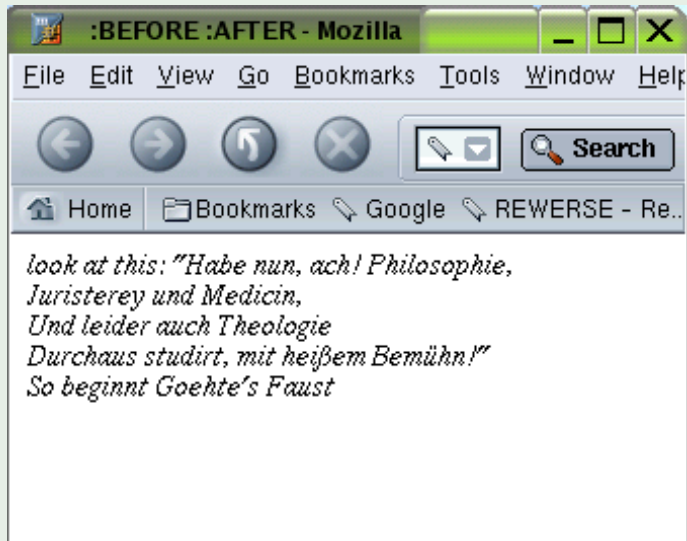
:FIRST-LETTER                :FIRST-LINE  
:BEFORE                       :AFTER

":BEFORE" and ":AFTER" are used with the "CITE"-tag of HTML, and require a "context" attribute: (Caution: not all browser support this)

## Example (Use of :BEFORE and :AFTER)

```
<html>
  <head> <title> :BEFORE :AFTER </title>
    <style type="text/css">
      CITE:BEFORE
      { content: "look at this: "}
      CITE:AFTER
      { content: "So beginnt Goethes Faust"}
    </style>
  </head>
  <body>
    <cite>
      "Habe nun, ach! Philosophie, <br>
      Juristerey und Medicin, <br>
      Und leider auch Theologie <br>
      Durchaus studirt, mit heis&szlig;em Bem&uuml;hn!"<br>
    </cite>
  </body>
</html>
```

## Example (Use of :BEFORE and :AFTER)



## Example (A stylesheet for links)

BODY

```
{ font-family: "Verdana";  
  font-size: 10pt;      }
```

A:LINK

```
{ color: blue;  
  text-decoration: none;  }
```

A:VISITED

```
{ color: red;  
  font-style: oblique;   }
```

A:ACTIVE

```
{ color: yellow;  
  text-decoration: overline; }
```

A:HOVER

```
{ color: green;  
  text-decoration: overline; }
```

## Example (A stylesheet for links)

An HTML file which uses this stylesheet: A1.html

```
<HTML>
  <HEAD>
    <TITLE>Pseudo-Klassen für Hyperlinks</TITLE>
    <LINK rel=Stylesheet href="A.css" type="text/css">
  </HEAD>

  <BODY>
    Für ein Beispiel, das sich um Hyperlinks dreht, braucht man
    selbstverständlich auch einen <A href="A2.html">solchen</A>.
    Dieser demonstriert :ACTIVE.<BR>
    Dies ist ein <A href="A1.htm">Link</A> zur selben Datei.
  </BODY>
</HTML>
```

The referred file A2.html:

```
<HTML>
  <HEAD>
    <TITLE>Für 'nen Hyperlink</TITLE>
  </HEAD>
  <BODY>
    Das ist lediglich eine Datei, auf die ein Hyperlink aus <I>A1.css</I> verweist.
    Sie ist ohne CSS!
  </BODY>
</HTML>
```

*Note: Example from "Daniel Jokat: Jetzt lerne ich Cascading Stylesheets, Markt+Technik Verlag, 2002".*

A collection of examples of CSS can be found at SELFHTML by Stefan Münz, online available at <http://de.selfhtml.org/layouts/index.htm>

"Kurzreferenz: CSS" from SELFHTML by Stefan Münz, online available at <http://de.selfhtml.org/navigation/css.htm>

More on css for example at <http://www.htmlhelp.com/reference/css/quick-tutorial.html>.

*Note: Slides in this section are based on the W3Schools' XPATH Tutorial,*

*[http://www.w3schools.com/xpath/xpath\\_intro.asp](http://www.w3schools.com/xpath/xpath_intro.asp)*

- XPath: string-based syntax
- XPath uses path expressions to identify nodes in an XML document
- These path expressions are similar to path expressions on a file system
- XPath defines a library of standard functions
- XPath was released as a W3C Recommendation November 16th 1999 as a language for addressing parts of an XML document.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<catalog>
  <cd country="USA">
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <price>10.90</price>
  </cd>
  <cd country="UK">
    <title>Hide your heart</title>
    <artist>Bonnie Tyler</artist>
    <price>9.90</price>
  </cd>
  <cd country="USA">
    <title>Greatest Hits</title>
    <artist>Dolly Parton</artist>
    <price>9.90</price>
  </cd>
</catalog>
```

- XPath uses a pattern expression to identify nodes in an XML document
- An XPath pattern is a slash-separated list of child element names that describe a path through the XML document.
- path starts with `"/`: absolute path, starting from the root element
- path starts with `//`: matching of elements to the path expression, starting at arbitrary level
- path starts with some string `"a_string"`: matching of elements to the path expression, starting at the element named `"a_string"`
- wildcards ( `*` ) can be used to select unknown XML elements.

## Remark

Often there exists more than one solution. Try your solutions with:  
<http://b-cage.net/code/web/xpath-evaluator.html> or:  
<http://www.personal-reader.de/semweb08/slides/xsd-coding.zip>

## Example (Locating Nodes in XPATH)

- select all price elements of all the cd elements of the catalog element
- select all the cd elements in the document
- select all the child elements of all the cd elements of the catalog element
- select all the price elements that have as a grandparent element the catalog element
- select all price elements which have 2 ancestors
- select all elements in the document

## Example (Solution: Locating Nodes in XPATH)

- select all price elements of all the cd elements of the catalog element: **/catalog/cd/price**.
- select all the cd elements in the document: **//cd**
- select all the child elements of all the cd elements of the catalog element: **/catalog/cd/\***
- select all the price elements that have as a grandparent element the catalog element: **/catalog/\*/price**
- select all price elements which have 2 ancestors: **//\*\*/price**
- select all elements in the document: **//\***

square brackets in an XPath expression allow to specify an element further.

## Example (Selecting Branches)

- select the first cd child element of the catalog element
- select the last cd child element of the catalog element (Note: There is no function named first())
- select all the cd elements of the catalog element that have a price element
- select all the cd elements of the catalog element that have a price element with a value of 10.90
- select all the price elements of all the cd elements of the catalog element that have a price element with a value of 10.90

## Example (Solution: Selecting Branches)

- select the first cd child element of the catalog element:  
`/catalog/cd[1]`
- select the last cd child element of the catalog element:  
`/catalog/cd[last()]`  
(Note: There is no function named first()):
- select all the cd elements of the catalog element that have a price element: `/catalog/cd[price]`
- select all the cd elements of the catalog element that have a price element with a value of 10.90:  
`/catalog/cd[price=10.90]`
- select all the price elements of all the cd elements of the catalog element that have a price element with a value of 10.90: `/catalog/cd[price=10.90]/price`

The "|" operator in an XPath expression allows to select several paths.

## Example (Selecting Several Paths)

- select all the title and artist elements of the cd element of the catalog element
- select all the title and artist elements in the document
- select all the title, artist and price elements in the document
- select all the title elements of the cd element of the catalog element, and all the artist elements in the document

## Example (Solution: Selecting Several Paths)

- select all the title and artist elements of the cd element of the catalog element: **`/catalog/cd/title | /catalog/cd/artist`**
- select all the title and artist elements in the document:  
**`//title | //artist`**
- select all the title, artist and price elements in the document:  
**`//title | //artist | //price`**
- select all the title elements of the cd element of the catalog element, and all the artist elements in the document:  
**`/catalog/cd/title | //artist`**

In XPath all attributes are specified by the @ prefix.

## Example (Selecting Attributes)

- select all attributes named country
- select all cd elements which have an attribute named country
- select all cd elements which have any attribute
- select all cd elements which have an attribute named country with a value of 'UK'

## Example (Solution: Selecting Attributes)

- select all attributes named country: `//@country`
- select all cd elements which have an attribute named country: `//cd[@country]`
- select all cd elements which have any attribute: `//cd[@*]`
- select all cd elements which have an attribute named country with a value of 'UK': `//cd[@country='UK']`

# Location Path Expression

- A location path can be absolute or relative.
  - An absolute location path starts with a slash ( / )
  - An relative location path starts with the name of an element
  - In both cases the location path consists of one or more location steps, each separated by a slash.  
absolute **/step/step/...**  
relative **step/step/...**
- location steps are evaluated in order one at a time, from left to right
- Each step is evaluated against the nodes in the current node-set
- Location steps consist of:
  - an axis (specifies the tree relationship between the nodes selected by the location step and the current node)
  - a node test (specifies the node type and expanded-name of the nodes selected by the location step)
  - zero or more predicates (use expressions to further refine the set of nodes selected by the location step)
- The syntax for a location step is: **axisname::nodetest[predicate]**

## Example (Location Step)

```
child::price[price=9.90]
```

- An axis defines a node-set relative to the current node.
- A node test is used to identify a node within an axis. We can perform a node test by name or by type.

## Axes

**ancestor** Contains all ancestors (parent, grandparent, etc.) of the current node

Note: This axis will always include the root node, unless the current node is the root node

**ancestor-or-self** Contains the current node plus all its ancestors (parent, grandparent, etc.)

**attribute** Contains all attributes of the current node

**child** Contains all children of the current node

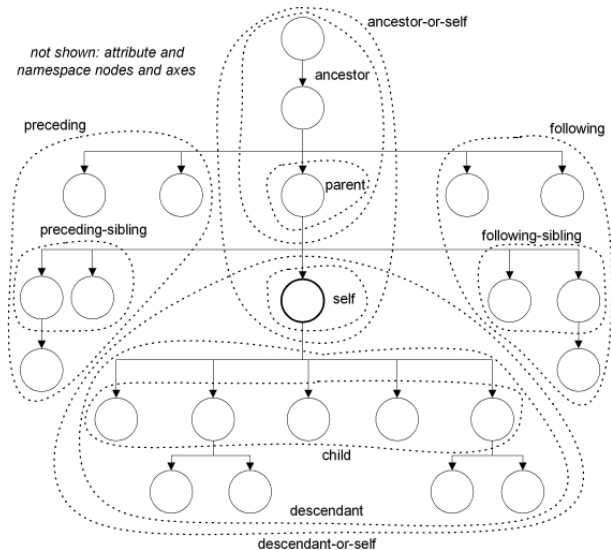
**descendant** Contains all descendants (children, grandchildren, etc.) of the current node

Note: This axis never contains attribute or namespace nodes

- descendant-or-self** Contains the current node plus all its descendants (children, grandchildren, etc.)
- following** Contains everything in the document after the closing tag of the current node
- following-sibling** Contains all siblings after the current node  
Note: If the current node is an attribute node or namespace node, this axis will be empty
- namespace** Contains all namespace nodes of the current node
- parent** Contains the parent of the current node
- preceding** Contains everything in the document that is before the starting tag of the current node
- preceding-sibling** Contains all siblings before the current node  
Note: If the current node is an attribute node or namespace node, this axis will be empty
- self** Contains the current node

# Axes

Note: the following graphic is courtesy of Crane Softwrights, found at P. Grosso, N. Walsh: *XSL Concepts and Practical Use*, online at <http://nwalsh.com/docs/tutorials/xsl/xsl/slides.html>.



- **name** matches `<name>` element nodes
- **\*** matches any element node
- **namespace:name** matches `<name>` element nodes in the specified namespace
- **namespace:\*** matches any element node in the specified namespace
- **comment()** matches comment-nodes
- **node()** matches all node-types
- **text()** matches text-nodes
- **processing-instruction()** matches a node of type processing instruction

## Example (Axes and Node Tests)

- select all cd elements that are children of the current node (if the current node has no cd children, it will select an empty node-set)
- selects the src attribute of the current node (if the current node has no src attribute, it will select an empty node-set)
- select all child elements of the current node
- select all attributes of the current node
- select the text node children of the current node
- select all the children of the current node
- select all the cd element descendants of the current node
- select all cd ancestors of the current node
- select all cd ancestors of the current node and, if the current node is a cd element, the current node as well
- select all price grandchildren of the current node
- select the document root

## Example (Solution: Axes and Node Tests)

- select all cd elements that are children of the current node (if the current node has no cd children, it will select an empty node-set) **child::cd**
- selects the src attribute of the current node (if the current node has no src attribute, it will select an empty node-set) **attribute::src**
- select all child elements of the current node **child::\***
- select all attributes of the current node **attribute::\***
- select the text node children of the current node **child::text()**
- select all the children of the current node **child::node()**
- select all the cd element descendants of the current node **descendant::cd**
- select all cd ancestors of the current node **ancestor::cd**
- select all cd ancestors of the current node and, if the current node is a cd element, the current node as well **ancestor-or-self::cd**
- Selects all price grandchildren of the current node **child::\*/\*/child::price**
- select the document root /

- A predicate filters a node-set into a new node-set.
- A predicate is placed inside square brackets ( [ ] ).

## Example (Predicates)

- select all price elements that are children of the current node with a price element that equals 9.90
- select the first cd child of the current node
- select the last cd child of the current node
- select the last but one cd child of the current node
- select the first five cd children of the current node
- select the seventh cd element in the document
- select all cd children of the current node that have a type attribute with value classic

## Example (Solution: Predicates)

- select all price elements that are children of the current node with a price element that equals 9.90 **child::price[price=9.90]**
- select the first cd child of the current node **child::cd[position()=1]**
- select the last cd child of the current node  
**child::cd[position()=last()]**
- select the last but one cd child of the current node  
**child::cd[position()=last()-1]**
- select the first five cd children of the current node  
**child::cd[position()<6]**
- select the seventh cd element in the document  
**/descendant::cd[position()=7]**
- select all cd children of the current node that have a type attribute with value classic **child::cd[attribute::type="classic"]**

- Abbreviations can be used when describing a location path.
- most important abbreviation is that **child::** can be omitted from a location step.

## Abbreviations:

**none** abbreviation for "child::" Example: "cd" is short for "child::cd"

**@** abbreviation for "attribute::"

Example: `cd[@type="classic"]` is short for `child::cd[attribute::type="classic"]`

**.** abbreviation for "self::node()"

Example: `"./cd"` is short for `"self::node()/descendant-or-self::node()/child::cd"`

**..** abbreviation for "parent::node()"

Example: `"../cd"` is short for `"parent::node()/child::cd"`

**//** abbreviation for "/descendant-or-self::node()/"

Example `"//cd"` is short for `"/descendant-or-self::node()/child::cd"`

## Example (Location Path Abbreviated Syntax)

- select all the cd elements that are children of the current node
- select all child elements of the current node
- select all text node children of the current node
- select the src attribute of the current node
- select all the attributes of the current node
- select the first cd child of the current node
- select the last cd child of the current node
- select all cd grandchildren of the current node
- select the first para of the third chapter of the book
- select all the cd descendants of the document root and thus selects all cd elements in the same document as the current node
- select the current node

## Example (Location Path Abbreviated Syntax 2)

- select the cd element descendants of the current node
- select the parent of the current node
- select the src attribute of the parent of the current node
- select all cd children of the current node that have a type attribute with value classic
- select the fifth cd child of the current node that has a type attribute with value classic
- select the fifth cd child of the current node if that child has a type attribute with value classic
- select all the cd children of the current node that have both a type attribute and a country attribute

## Example (Solution: Location Path Abbreviated Syntax)

- select all the cd elements that are children of the current node: **cd**
- select all child elements of the current node: **\***
- select all text node children of the current node: **text()**
- select the src attribute of the current node: **@src**
- select all the attributes of the current node: **@\***
- select the first cd child of the current node: **cd[1]**
- select the last cd child of the current node: **cd[last()]**
- select all cd grandchildren of the current node: **\*/cd**
- select the first para of the third chapter of the book:  
**/book/chapter[3]/para[1]**
- select all the cd descendants of the document root and thus selects all cd elements in the same document as the current node: **//cd**
- select the current node: **.**

## Example (Solution: Location Path Abbreviated Syntax 2)

- select the cd element descendants of the current node: `./cd`
- select the parent of the current node: `..`
- select the src attribute of the parent of the current node:  
`../@src`
- select all cd children of the current node that have a type attribute with value classic: `cd[@type="classic"]`
- select the fifth cd child of the current node that has a type attribute with value classic: `cd[@type="classic"] [5]`
- select the fifth cd child of the current node if that child has a type attribute with value classic: `cd[5][@type="classic"]`
- select all the cd children of the current node that have both a type attribute and a country attribute: `cd[@type and @country]`

## Numerical Expressions

- Numerical expressions are used to perform arithmetic operations on numbers.
- supported operators: "+", "-", "\*", "div" (division), "mod" (modulus)
- Note: XPath always converts each operand to a number before performing an arithmetic expression.

## Equality Expressions

- Equality expressions are used to test the equality between two values.
- supported operators: "=" (equal), "!=" (not equal)

## Testing Against a Node-Set

- If the test value is tested for equality against a node-set, the result is true if the node-set contains any node with a value that matches the test value.
- If the test value is tested for not equal against a node-set, the result is true if the node-set contains any node with a value that is different from the test value.
- The result is that the node-set can be equal and not equal at the same time !!!

## Relational Expressions

- Relational expressions are used to compare two values.
- supported operators: " $<$ ", " $<=$ ", " $>$ ", " $>=$ "
- Note: XPath always converts each operand to a number before performing the evaluation.

## Boolean Expressions

- Boolean expressions are used to compare two values.
- supported operators: "or" "and"

## XPath Function Library

- The XPath function library contains a set of core functions for converting and translating data.

`count()` Returns the number of nodes in a node-set  
Syntax: `number=count(node-set)`

`id()` Selects elements by their unique ID  
Syntax: `node-set=id(value)`

`last()` Returns the position number of the last node in the processed node list  
Syntax: `number=last()`

- `local-name()` Returns the local part of a node. A node usually consists of a prefix, a colon, followed by the local name  
Syntax: `string=local-name(node)`
- `name()` Returns the name of a node  
Syntax: `string=name(node)`
- `namespace-uri()` Returns the namespace URI of a specified node  
Syntax: `uri=namespace-uri(node)`
- `position()` Returns the position in the node list of the node that is currently being processed  
Syntax: `number=position()`

## String Functions

`concat()` Returns the concatenation of all its arguments  
Syntax: `concat(val1, val2, ..)`

`contains()` Returns true if the second string is contained within the first string, otherwise it returns false  
Syntax: `contains(val,substr)`

`normalize-space()` Removes leading and trailing spaces from a string  
Syntax: `normalize-space(string)`

`starts-with()` Returns true if the first string starts with the second string, otherwise it returns false  
Syntax: `starts-with(string,substr)`

`string()` Converts the value argument to a string  
Syntax: `string(value)`

`string-length()` Returns the number of characters in a string  
Syntax: `string-length(string)`

`substring()` Returns a part of the string in the string argument

Syntax: `substring(string,start,length)`

Example: `substring('Beatles',1,4)` Result: 'Beat'

`substring-after()` Returns the part of the string in the string argument that occurs after the substring in the substr argument

Syntax: `substring-after(string,substr)`

`substring-before()` Returns the part of the string in the string argument that occurs before the substring in the substr argument

Syntax: `substring-before(string,substr)`

`translate()` Performs a character by character replacement. It looks in the value argument for characters contained in string1, and replaces each character for the one in the same position in the string2

Syntax: `translate(value,string1,string2)`

Example: `translate('12:30','30','45')` Result: '12:45'

Example: `translate('12:30','03','54')` Result: '12:45'

Example: `translate('12:30','0123','abcd')` Result: 'bc:da'

## Number Functions

- `ceiling()` Returns the smallest integer that is not less than the number argument
- `floor()` Returns the largest integer that is not greater than the number argument
- `number()` Converts the value argument to a number
- `round()` Rounds the number argument to the nearest integer
- `sum()` Returns the total value of a set of numeric values in a node-set

## Boolean Functions

`boolean()` Converts the value argument to Boolean and returns true or false

`false()` Returns false

`true()` Returns true

`lang()` Returns true if the language argument matches the language of the the `xsl:lang` element, otherwise it returns false

`not()` Returns true if the condition argument is false, and false if the condition argument is true

More examples to play with:

<http://www.w3schools.com/XPath/>