Contents

1 DTD

A DTD is a Document Type Definition and defines the structure & the legal elements & attributes of an XML document. By using a DTD independent groups of people can agree on a standard DTD for interchanging data. An application can use a DTD to verify that XML data is valid.
A DTD can be declared inside an XML file or in an external file. Internal:

```
<?xml version="1.0"?>
<!DOCTYPE note [
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
```

```
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

External:

```
<?xml version="1.0"?>
<!DOCTYPE note SYSTEM "note.dtd">
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

note.dtd:

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

Seen from a DTD point of view, all XML documents are made up by the following building blocks:

- Elements = main building blocks
- Attributes = provide extra information about elements
- Entities = some characters in XML have a special meaning, entities are expanded when a document is parsed by an XML parser
- PCDATA = parsed character data (text that will be examined by the parser for entities & markup)
- CDATA = character data that will not be parsed and expanded by a parser

1.1 Elements

- declared via `<!ELEMENT element-name category>` or `<!ELEMENT element-name (element-content)>`
- empty elements are declared with the category keyword EMPTY, eg: `<!ELEMENT br EMPTY>` → `<br />`
- elements with only parsed character data are declared with #PCDATA inside parantheses, eg: `<!ELEMENT recipient (#PCDATA)>`
- elements declared with the category keyword ANY, can contain any combination of parsable data, eg: `<!ELEMENT note ANY>`
- elements with one more children are declared with the name of the children elements inside parantheses, eg: `<!ELEMENT elem-name (child1, child2,...)`

- when children are declared in such a sequence separated by commas, the children *must appear in the same sequence (order)* in the document
- in a full declaration the children must also be declared, and the children can also have children, eg:

```
<!ELEMENT note (to,from,heading,body)>
<!ELEMENT to (#PCDATA)>
<!ELEMENT from (#PCDATA)>
<!ELEMENT heading (#PCDATA)>
<!ELEMENT body (#PCDATA)>
```

- declaring only one exact occurence of an elment: `<!ELEMENT note (message)>`
- declaring minimum one (or more) occurence of an element: `<!ELEMENT note (message+)`
- declaring zero or more occurences of an element: `<!ELEMENT note (message*)>`
- declaring zero or one occurence of an element: `<!ELEMENT note (message?)>`
- declaring either/or content: `<!ELEMENT note (to,from,header,(message|body))>` declares that the "note" element must contain a 'to', 'from', 'header' element and either a 'message' or a 'body' element
- declaring mixed content: `<!ELEMENT note (#PCDATA|to|from|header|message)*>` declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements

1.2 Attributes

In a DTD, attributes declared with an ATTLIST declaration, which has the following syntax: `<!ATTLIST element-name attribute-name attribute-type attribute-value>` for example `<!ATTLIST payment type CDATA "check">` → `<payment type`"check"`>`=
The **attribute type** can be one of the following:

Type	Description
CDATA	value is character data
(en1|en2|..)	value must be one from enumerated list
ID	value is a unique id
IDREF	value is id of another element
IDREFS	value is a list other ids
NMTOKEN	value is a valid XML name
NMTOKENS	value is a list of valid XML names
ENTITY	value is an entity
ENTITIES	value is a list of entities
NOTATION	value is a name of a notation
xml:	value is a predefined xml value

The **attribute value** can be one of the following:

Value	Explanation
value	default value of the attribute
#REQUIRED	attribute is required
#IMPLIED	attribute is optional
#FIXED *value*	attribute value is fixed

Examples:

A default attribute value

DTD:

```
<!ELEMENT square EMPTY>
<!ATTLIST square width CDATA "0">
```

Valid XML:

```
<square width="100" />
```

- in the above example, "square" is defined to be an empty element with a 'width' attribute of type CDATA and a default value of 0 if no width is specified

A required attribute value

DTD:

```
<!ATTLIST person number CDATA #REQUIRED>
```

Valid XML:

```
<person number="5677" />
```

Invalid XML:

```
<person />
```

- use the #REQUIRED keyword if you don't have an option for a default value, but still want to force the attribute to be present.

An implied attribute value

DTD:

```
<!ATTLIST contact fax CDATA #IMPLIED>
```

Valid XML:

```
<contact fax="555-667788" />
```

Valid XML:

```
<contact />
```

- use the #IMPLIED keyword if you don't want to force the author to include an attribute, and you don't have an option for a default value

A fixed attribute value

DTD:

```
<!ATTLIST sender company CDATA #FIXED "Microsoft">
```

Valid XML:

```
<sender company="Microsoft" />
```

Invalid XML:

```
<sender company="W3Schools" />
```

- use the #FIXED keyword when you want an attribute to have a fixed value without allowing the author to change it. If an author includes another value, the XML parser will return an error

Enumerated attribute values

DTD:

```
<!ATTLIST payment type (check|cash) "cash">
```

XML example:

```
<payment type="check" />
```

or

```
<payment type="cash" />
```

- use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values

1.3 Entities

Entities are used to define shortcuts to special characters and can be declared internally via `<!ENTITY entity-name "entity-value">` or externally via `<!ENTITY entity-name SYSTEM "URI/URL">`
Example (internal):

```
DTD Example:
<!ENTITY writer "Donald Duck.">
<!ENTITY copyright "Copyright W3Schools.">
XML example:
<author>&writer;&copyright;</author>
```

Note: An entity has three parts: an ampersand (&), an entity name, and a semicolon (;)

Example (external):

```
DTD Example:
<!ENTITY writer SYSTEM "https://www.w3schools.com/entities.dtd">
<!ENTITY copyright SYSTEM "https://www.w3schools.com/entities.dtd">
XML example:
<author>&writer;&copyright;</author>
```

2 XML Schema

An XML Schema describes the structure of an XML document. The XML Schema language is also referred to as XML Schema Definition (XSD).
The purpose of an XML Schema is to define the legal building blocks of an XML document:

- the elements and attributes that can appear in a document
- the number of (and order of) child elements
- data types for elements and attributes
- default and fixed values for elements and attributes

One of the greatest strength of XML Schemas is the support for data types.

- It is easier to describe allowable document content
- It is easier to validate the correctness of data
- It is easier to define data facets (restrictions on data)
- It is easier to define data patterns (data formats)
- It is easier to convert data between different data types

For example look at this simple XML doc called "note.xml":

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

```
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

And the corresponding "note.xsd":

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

  <xs:element name="note">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="to" type="xs:string"/>
        <xs:element name="from" type="xs:string"/>
        <xs:element name="heading" type="xs:string"/>
        <xs:element name="body" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

The note element is a **complex type** because it contains other elements. The other elements (to, from, heading, body) are **simple types** because they do not contain other elements.
The <schema> element is the root of every XML schema and often looks similar to the example above. The fragment `xmlns:xs="http://"xmlns:xs="http://www.w3.org/2001/XMLSchema"` indicates that the elements and datatypes used in the schema come from the `"http://www.w3.org/2001/XMLSchema"` namespace. It also specifies that the elements and data types that come from the `"http://www.w3.org/2001/XMLSchema"` namespace should be prefixed with **xs:**
The fragment `targetNamespace="https://www.w3schools.com"` indicates that the elements defined by this schema (note, to, from, heading, body) come from the `"https://www.w3schools.com"` namespace. This fragment `xmlns="https://www.w3schools.com"` indicates that the default namespace is `"https://www.w3schools.com"`. Last but not least the fragment `elementFormDefault="qualified"` indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.
In the XML document the reference to the XML schema has the following fragment `xmlns="https://www.w3schools.com"` which specifies the default namespace declaration which tells the schema-validator that all the elements used in this XML document are declared in the `"https://www.w3schools.com"` namespace. Once you have the XML Schema Instance available via `xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"` the `schemaLocation` attribute can be used which has two space-separated values which indicate the namespace to use and the location of the XML schema to use for that namespace: `xsi:schemaLocation="https://www.w3schools.com note.xsd"`

2.1 Simple Elements

A simple element is an XML element that contains only text. It cannot contain any other elements or attributes. However, the "only text" restriction is quite misleading. The text can be of many different types. It can be one of the types included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself. You can also add restrictions (facets) to a data type in order to limit its content, or you can require the data to match a specific pattern.

This is the syntax `<xs:element name="xxx" type="yyy"/>` and the most common types are:

- xs:string
- xs:decimal
- xs:integer
- xs:boolean
- xs:date
- xs:time

Here are some examples:

```
<xs:element name="lastname" type="xs:string"/>
<xs:element name="age" type="xs:integer"/>
<xs:element name="dateborn" type="xs:date"/>
and the corresponding XML elements
<lastname>Refsnes</lastname>
<age>36</age>
<dateborn>1970-03-27</dateborn>
```

Simple elements may have a default value OR a fixed value specified. A default value is automatically assigned to the element when no other value is specified `<xs:element name="color" type="xs:string" default="red"/>`. A fixed value is also automatically assigned to the element, and you cannot specify another value `<xs:element name="color" type="xs:string" fixed="red"/>`.

2.2 Attributes

Simple elements cannot have attributes. If an element has attributes, it is considered to be of a complex type. But the attribute itself is always declared as a simple type. `<xs:attribute name="xxx" type="yyy"/>` for example:

```
<xs:attribute name="lang" type="xs:string"/>
<!-- and the corresponding xml -->
<lastname lang="EN">Smith</lastname>
```

Similar to "Simple Elements" default and fixed values are possible. However attributes are optional by default, to specify that an attribute is required use the "use" attribute: `<xs:attribute name="lang" type="xs:string" use="required"/>`

2.3 Restrictions

Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.
The following example defines an element called "age" with a restriction. The value of age cannot be lower than 0 or greater than 120:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
    </xs:restriction>
```

```
  </xs:simpleType>
</xs:element>
```

To limit the content of an XML element to a set of acceptable values, we would use the enumeration constraint. The example below defines an element called "car" with a restriction. The only acceptable values are: Audi, Golf, BMW:

```
<xs:element name="car">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:enumeration value="Audi"/>
      <xs:enumeration value="Golf"/>
      <xs:enumeration value="BMW"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

The example above could also have been written like this:

```
<xs:element name="car" type="carType"/>

<xs:simpleType name="carType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="Audi"/>
    <xs:enumeration value="Golf"/>
    <xs:enumeration value="BMW"/>
  </xs:restriction>
</xs:simpleType>
```

Note: In this case the type "carType" can be used by other elements because it is not a part of the "car" element.

To limit the content of an XML element to define a series of numbers or letters that can be used, we would use the pattern constraint. The example below defines an element called "letter" with a restriction. The only acceptable value is ONE of the LOWERCASE letters from a to z:

```
<xs:element name="letter">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[a-z]"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Other examples:

- three uppercase A-Z letters: `<xs:pattern value="[A-Z][A-Z][A-Z]"/>`
- three upper/lowercase letter: `<xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>`
- one of x,y or z: `<xs:pattern value="[xyz]"/>`
- five digits: `<xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>`
- zero or more occurences of lowercase letters: `<xs:pattern value="([a-z])*"/>`
- one or more pairs of lowercase followed by uppercase letters eg sToP: `<xs:pattern value="([a-z][A-Z])+"/>`

- only male or female: `<xs:pattern value="male|female"/>`
- exactly eight characters in a row and those characters must be lowercase or uppercase letters from a to z, or a number from 0 to 9: `<xs:pattern value="[a-zA-Z0-9]{8}"/>`

To specify how whitespace characters should be handled, we would use the whiteSpace constraint. This example defines an element called "address" with a restriction. The whiteSpace constraint is set to "preserve", which means that the XML processor WILL NOT remove any white space characters:

```
<xs:element name="address">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:whiteSpace value="preserve"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

- other options would be `replace` (replaces all whitespace with space) and `collapse` (collapse to single space)

To limit the length of a value in an element, we would use the length (`<xs:length value="8"/>`), maxLength(`<xs:maxLeng` `value="8"/>`), and minLength(`<xs:minLength value="5"/>`) constraints.

Additional restrictions are `fractionDigits`, `minExclusive`, `maxExclusive`, `totalDigits`. Unique id Attribute:

```
<xsd:element name="root" type="myList">
   <xsd:unique name="myId">
     <xsd:selector xpath="./person"/>
     <xsd:field xpath="@id"/>
   </xsd:unique>
 </xsd:element>
```

2.4 Complex Elements

A complex element is an XML element that contains other elements and/or attributes. There are four kinds of complex elements:

- empty elements
- elements that contain only other elements
- elements that contain only text
- elements that contain both other elements and text

Note: Each of these elements may contain attributes as well!
Example:

```
<xs:element name="employee">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="firstname" type="xs:string"/>
      <xs:element name="lastname" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
```

```
</xs:element

<employee>
  <firstname>John</firstname>
  <lastname>Smith</lastname>
</employee>
```

- `sequence` means that the elements must appear in that order inside employee

It is also possible to base a complex element on an existing complex element and add some elements:

```
<xs:element name="employee" type="fullpersoninfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
    <xs:element name="lastname" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="fullpersoninfo">
  <xs:complexContent>
    <xs:extension base="personinfo">
      <xs:sequence>
        <xs:element name="address" type="xs:string"/>
        <xs:element name="city" type="xs:string"/>
        <xs:element name="country" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

To define an empty type, define a type without declaring any elements inside it:

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

A **complex text-only element** can contain text and attributes. This type contains only simple content (text and attributes), therefore we add a simpleContent element around the content. When using simple content, you must define an extension OR a restriction within the simpleContent element, like this:

```
<xs:element name="shoesize">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:integer">
        <xs:attribute name="country" type="xs:string" />
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>

<shoesize country="france">35</shoesize>
```

A mixed complex type element can contain attributes, elements, and text. To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true". The <xs:sequence> tag means that the elements defined (name, orderid and shipdate) must appear in that order inside a "letter" element:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
      <xs:element name="orderid" type="xs:positiveInteger"/>
      <xs:element name="shipdate" type="xs:date"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<letter>
  Dear Mr. <name>John Smith</name>.
  Your order <orderid>1032</orderid>
  will be shipped on <shipdate>2001-07-13</shipdate>.
</letter>
```

2.4.1 Indicators

1. Order Indicators Order indicators are used to define the order of the elements. The `<all>` indicator specifies that the child elements can appear in any order, and that each child element must occur only once (When using the <all> indicator you can set the <minOccurs> indicator to 0 or 1 and the <maxOccurs> indicator can only be set to 1).
   The `<choice>` indicator specifies that either one child element or another can occur. The `<sequence>` indicator specifies that the child elements must appear in a specific order:

   ```
   <xs:element name="person">
     <xs:complexType>
     <xs:all>
       <xs:element name="firstname" type="xs:string"/>
       <xs:element name="lastname" type="xs:string"/>
      </xs:all>
      <xs:choice>
       <xs:element name="employee" type="employee"/>
       <xs:element name="member" type="member"/>
      </xs:choice>
      <xs:sequence>
       <xs:element name="firstname" type="xs:string"/>
       <xs:element name="lastname" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
   </xs:element>
   ```

2. Occurence Indicators Occurrence indicators are used to define how often an element can occur (Note: For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) the default value for maxOccurs and minOccurs is 1). The `<maxOccurs>` indicator specifies the maximum number of times an element can occur. The `<minOccurs>` indicator specifies the minimum number of times an element can occur (by default is 1).
   Example:
   family.xsd

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">

<xs:element name="persons">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="person" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="full_name" type="xs:string"/>
            <xs:element name="child_name" type="xs:string"
            minOccurs="0" maxOccurs="5"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
  </xs:complexType>
</xs:element>

</xs:schema>
```

and family.xml

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

<persons xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="family.xsd">

<person>
  <full_name>Hege Refsnes</full_name>
  <child_name>Cecilie</child_name>
</person>

<person>
  <full_name>Tove Refsnes</full_name>
  <child_name>Hege</child_name>
  <child_name>Stale</child_name>
  <child_name>Jim</child_name>
  <child_name>Borge</child_name>
</person>

<person>
  <full_name>Stale Refsnes</full_name>
</person>

</persons>
```

3. Group Indicators Group indicators are used to define related sets of elements which must be all, choice or sequence elements, eg:

```
<xs:group name="persongroup">
  <xs:sequence>
    <xs:element name="firstname" type="xs:string"/>
```

```
    <xs:element name="lastname" type="xs:string"/>
    <xs:element name="birthday" type="xs:date"/>
  </xs:sequence>
</xs:group>
```

can then be referenced in another definiton like this:

```
<xs:element name="person" type="personinfo"/>

<xs:complexType name="personinfo">
  <xs:sequence>
    <xs:group ref="persongroup"/>
    <xs:element name="country" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

This is also possible for attribute groups, just replace `xs:group` with `xs:attributeGroup` and `xs:element` with `xs:attribute`.

2.4.2 Misc

The <any> element enables us to extend the XML document with elements not specified by the schema (eg `xs:any minOccurs="0"/>`).
The <anyAttribute> element enables us to extend the XML document with attributes not specified by the schema (eg `<xs:anyAttribute/>`).
Common data types:

- String: `string`, `ID`, `IDREF`, `language`
- Date: `date`, `dateTime`, `time`
- Numeric: `decimal`, `integer`

3 XSLT

XSL (eXtensible Stylesheet Language) is a styling language for XML. XSLT stands for XSL Transformations. XSLT can be used to transform XML documents into other formats like for example HTML. XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library. There are over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, and more.

- **XSLT** is a language for transforming XML documents using XPath
- **XPath** is a language for navigating in XML documents
- **XQuery** is a language for querying XML documents

3.1 Basics

An XSL style sheet consists of one or more set of rules that are called templates. A template contains rules to apply when a specified node is matched.
The `<xsl:template>` element is used to build templates. The match attribute is used to associate a template with an XML element. The match attribute can also be used to define a template for the entire XML document.

The value of the match attribute is an XPath expression (i.e. `match="/"` defines the whole document). Look at the previous example in the "Basics" section to see this in action.
The `xsl:value-of>` is used to extract the value of an XML element and add it to the output stream of the transformation. It is used in conjuction with an `select=XPATH`.
The `<xslLfor-each>` element can be used to select every XML element of a specified node-set via `select=XPATH`.
The output can also be filtered, eg `<xsl:for-each select="catalog/cd[artist='Bob Dylan']">`, legal filter operators are `, !`, <, >
The `<xsl:sort` element is used to sort the output - the **select** attribbute indicates which XML element to sort on.
To add a conditional test, add the `<xsl:if>` element with a `test="expression"` . The `<xsl:choose>` element is used in conjunction with `<xsl:when test="expression">` and `xsl:otherwise>` to express multiple conditional tests:

```
<xsl:choose>
  <xsl:when test="expression">
    ... some output ...
  </xsl:when>
  <xsl:otherwise>
    ... some output ....
  </xsl:otherwise>
</xsl:choose>
```

Example showcasing the above basics:
The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>.
We want to transform the following XML document "cdcatalog.xml" into XHTML:

```
<?xml version="1.0" encoding="UTF-8"?>
<catalog>
  <cd>
    <title>Empire Burlesque</title>
    <artist>Bob Dylan</artist>
    <country>USA</country>
    <company>Columbia</company>
    <price>10.90</price>
    <year>1985</year>
  </cd>
...
...
</catalog>
```

First create an XSL stylesheet "cdcatalog.xsl" with a transformation table:

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

<xsl:stylesheet version="1.0"
                xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <h2>My CD Collection</h2>
        <table border="1">
          <tr bgcolor="#9acd32">
            <th>Title</th>
            <th>Artist</th>
```

```
            </tr>
            <xsl:for-each select="catalog/cd">
              <xsl:sort select="artist"/>
              <xsl:if test="price &amp;gt; 10">
                <tr>
                  <td><xsl:value-of select="title"/></td>
                  <td><xsl:value-of select="artist"/></td>
                </tr>
              </xsl:if>
            </xsl:for-each>
          </table>
        </body>
      </html>
    </xsl:template>
</xsl:stylesheet>
```

And then link the XSL stylesheet into the XML document from before by putting `<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>` before the root element.

3.2 Template Example

XML:

```
<!xml version="1.0"?>
<deliveries>
  <article id="3526">
    <name>apple</name>
    <price unitprice="true">8.97</price>
    <suppler>Fa. Krause</supplier>
  </article>
  <article id="7866">
    <name>cherries</name>
    <price unitprice="true">10.45</price>
    <suppler>Fa. Helbig</supplier>
  </article>
  <article id="3526">
    <name>apple</name>
    <price unitprice="true">12.67</price>
    <suppler>Fa. Liebig</supplier>
  </article>
  (...)
  <article id="7789">
    <name>pineapple</name>
    <price unitprice="true">8.60</price>
    <suppler>Fa. Richard</supplier>
  </article>
</deliveries>
```

Write an XSL transformation which produces the following result:

- Fa. Helbig supplies: cherries
- Fa. Liebig supplies: apple

- Fa. Krause supplies: apple cherries
- Fa. Hoeller supplies: cherries cabbage banana
- Fa. Reinhardt supplies: cabbage
- Fa. Richard supplies: cherries pineapple

Solution:

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="utf-8"/>
  <xsl:template match="seite">
    <xsl:apply-templates select="document('deliveries.xml')"/>
  </xsl:template>

  <xsl:template match="deliveries">
    <html>
      <head><title><xsl:text>suppliers</xsl:text></title></head>
      <body bgcolor="#ffffff">
        <xsl:for-each select="//supplier[not(preceding::supplier/. = .)]">
          <xsl:apply-templates select=".">
          </xsl:apply-templates>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>

  <xsl:template match="supplier">
    <p>
      <xsl:value-of select="text()"/><xsl:text> supplies </xsl:text>
      <xsl:for-each select="//article[supplier/text() = current()/text()]">
        <xsl:value-of select="name/text()"/><xsl:text> <xsl:text>
      </xsl:for-each>
    </p>
  </xsl:template>

</xsl:stylesheet>
```

4 XPath Reference

4.1 Selecting nodes

In XPath, there are seven kinds of nodes: element, attribute, text, namespace, processing-instruction, comment, and document nodes.

Examples:

Table 1:

Expression	Description
nodename	selects all nodes with the name "nodename"
/	selects from the root node
//	selects nodes in the document from the current node that match the selection no matter where they are
.	selects the current node
..	selects the parent of the current node
@	selects attributes

Path Expression	Result
bookstore	selects all nodes with the name "bookstore"
/bookstore	selects the root element bookstore (absolute path)
bookstore/book	selects all book elements that are children of bookstore
//book	selects all book elements no matter where they are in the document
bookstore/book	selects all book elements that are descendant of the bookstore element,
//@lang	selects all attributes that are named lang
//Sales.Customer[starts-with(Name, 'Jans')]	all customers from which the name starts with 'Jans'

4.2 Predicates

Predicates are used to find a specific node or a node that contains a specific value. Predicates are always embedded in square brackets. In the table beloware some path expressions with predicates and the result of the expressions:

Path Expression	Result
/bookstore/book[1]	selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	selects the second to last book element that is the child of the bookstore element
/bookstore/book[position()<3]	selects the first two book elements that are child of bookstore
/title[@lang='en']	selects all title elements that have a "lang" attr with val "en"
/bookstore/book[price>35.00]	selects all the book elements of the bookstore that have a price val higher than 3
/bookstore/book[price>35.00]/title	selects all the title elements of book elements of the bookstore that have a price

4.3 Selecting unknown nodes

XPath wildcards can be used to select unknown XML nodes:

Wildcard	Description
*	matches any element node
@*	matches any attribute node
node()	matches any node of any kind

Examples:

Path Expression	Result
/bookstore/*	selects all the child element nodes of the bookstore element
//*	selects all elements in the document
//title[@*]	selects all title elements which have at least one attribute of any kind

By using the "|" operator to separate XPath expressions multiple paths can be selected eg `//book/title | //book/price` selects all the title AND price elements of all book elements

5 RDF

RDF triple: Describes a statement in form of a relationship (P) between a subject (S) and an object (O).

- Statement describes a thing S, where a property P is provided with a value of O
- RDF triple (S,P,O)
  - Subject - Predicate - Object, or technically Subject - Property - Object
- P connects S and O
- S and P are URIs, O is a URI or a literal
- conceptually P connects S and O

First example statement: "Gaedke is gay"

- S = `http://gaedke.com/`, P = `http://.../is`, O = `http://.../gay`
- statements as an RDF triple: `http://gaedke.com/`,`http://.../is`,`http://.../gay`

RDF is a data model... machine-readable implementations include RDF/XML, Notation 3 (N3), Turtle, etc.

5.1 RDF/XML

- `rdf:Description` elements describe resources (**Subjects**)
- nested elements are properties/predicates (**P**)
- attributes *or* contents of a property element describe **Objects** (O)

For example:

```
<?xml version="1.0" encoding="utf-8"?>
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:zt="http://example.org/evkonzept#">
  <rdf:Description about="http://zoo.../tiere#t12">
    <zt:Vatervon resource="http://zoo.../tiere#t34"/>
    <zt:Name>Teddy</zt:Name>
  </rdf:Description>
</rdf:RDF>
```

- Tier 12 (`t12`) is the subject, `zt:Vatervon` is the property and Tier 34 (`t34`) is the object (URI)
- Tier 12 (`t12`) is the subject, `zt:Name` is the property and "Teddy" is the object (literal)

5.2 RDF Schema

RDF Vocabulary Description Language to defined the vocabulary similar to XML Schema is defined via `xmlns:rdfs="http`
RDF Schema does not provide actual application-specific classes and properties. Instead RDF Schema provides the framework to describe application-specific classes and properties. Classes in RDF Schema are much like classes in object oriented programming languages. This allows resources to be defined as instances of classes, and subclasses of classes.
Example 1:

```
<?xml version="1.0"?>
<rdf:RDF
     xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
     xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
     xml:base="http://www.animals.fake/animals#">

  <rdf:Description rdf:ID="animal">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
  </rdf:Description>

  <rdf:Description rdf:ID="horse">
    <rdf:type rdf:resource="http://www.w3.org/2000/01/rdf-schema#Class"/>
    <rdfs:subClassOf rdf:resource="#animal"/>
  </rdf:Description>

</rdf:RDF>
```

In the example above, the resource "horse" is a subclass of the class "animal". Since an RDFS class is an RDF resource we can abbreviate the example above by using rdfs:Class instead of rdf:Description, and drop the rdf:type information:

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

<rdf:RDF
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xml:base="http://www.animals.fake/animals#">

<rdfs:Class rdf:ID="animal" />

<rdfs:Class rdf:ID="horse">
  <rdfs:subClassOf rdf:resource="#animal"/>
</rdfs:Class>

</rdf:RDF>
```

6 SPARQL

SPARQL is a query language and a protocol for accessing RDF designed by the W3C RDF Data Access Working Group. As a query language, SPARQL is "data-oriented" in that it only queries the information held in the models; there is no inference in the query language itself. Of course, the Jena model may be 'smart' in that it provides the impression that certain triples exist by creating them on-demand, including OWL reasoning. SPARQL does not do anything other than take the description of what the application wants, in the form of a query, and returns that information, in the form of a set of bindings or an RDF graph.

6.1 SPARQL Examples

List all persons whose last name is "Smith"

```
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>

SELECT ?person
```

```
WHERE
{
  ?person vcard:family-name "Smith".
}
```

- in this query the triple pattern will match against triples whose predicate is the `family-name` property from the `vcard` vocabulary whose object is the string `"Smith"` and whose subject is anything (because `?person` is just a variable/wildcard)
- `SELECT` indicates which values should be listed after the query executes

The result of such query would be for example `emp1, emp2`. Because this does not tell much, let's extend the query. Let's add a second triple pattern that matches on the `?givenName` (just a wildcard) for everything that matched in the previous query (first line in `WHERE` statement) and whose predicate is a `given-name` property from the vcard vocab:

```
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>

SELECT ?person ?givenName
WHERE
{
  ?person vcard:family-name "Smith".
  ?person vcard:given-name ?givenName.
}
```

Now the result could look like:

?person	?givenName
emp1	Heidi
emp2	John

Let's retrieve the given name, family name and hire date of *all* employees:

```
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>

SELECT ?givenName ?familyName ?hd
WHERE
{
  ?person vcard:given-name ?givenName.
  ?person vcard:family-name ?familyName.
  ?person vcard:hireDate ?hd.
}
```

Result:

?givenName	?familyName	?hd
Jane	Berger	2019-03-10
Francis	Jones	2019-02-13
John	Smith	2019-01-28
Heidi	Smith	2019-01-13

To narrow down the above results via some condition we could use filter:

```
PREFIX vcard: <http://www.w3.org/2006/vcard/ns#>
```

```
SELECT ?givenName ?familyName ?hd
WHERE
{
  ?person vcard:given-name ?givenName.
  ?person vcard:family-name ?familyName.
  ?person vcard:hireDate ?hd.
  FILTER(?hd < "2019-03-01")
}
```

Result:

?givenName	?familyName	?hd
Francis	Jones	2019-02-13
John	Smith	2019-01-28
Heidi	Smith	2019-01-13

Let's say we add another triple pattern `?person vcard:completedOrientation ?oDate.`. Now only employees who have an `vcard:completedOrientation` predicate association would be listed. But we still want to show all employees, so in this case the triple pattern should be made optional `OPTIONAL {?person vcard:completedOrientation ?oDate.}`.
On the other hand if you want to list every person who has not completed the orientation yet, you would use `NOT EXISTS {?person vcard:completedOrientation ?oDate.}`

7 OWL by Erfan

OWL (Web Ontology Language)

- Enables (by means of additional vocabulary bound to formal semantics) stronger interpretation possibilities of Web contents such as XML, RDF and RDFS

OWL consists of three sub-languages with increasing expressive power: OWL Lite, OWL DL , OWL Full OWL Lite • Easy to implement and made for simple taxonomies • equivalence, i.e. owl:sameAs • Property characteristics, i.e. owl:InverseOf • Many further aspects: property restrictions, cardinality, intersections OWL DL • Allows a decidable set of first-order logic OWL Full • Same constructs as in OWL DL, but without restrictions; not decidable statements can also be described. Ontology is undecidable, but it, thus, enables higher order predicate logic