Contents

[XML Tutorial 7](#_Toc31984568)

[IMPORTANT QUESTIONS 7](#_Toc31984569)

[XML Elements 8](#_Toc31984570)

[Empty XML Elements 8](#_Toc31984571)

[XML Naming Rules 8](#_Toc31984572)

[Best Naming Practices 9](#_Toc31984573)

[XML Attributes 9](#_Toc31984574)

[Avoid XML Attributes? 9](#_Toc31984575)

[XML Namespaces 9](#_Toc31984576)

[Name Conflicts 9](#_Toc31984577)

[XML Namespaces - The xmlns Attribute 10](#_Toc31984578)

[Uniform Resource Identifier (URI) 10](#_Toc31984579)

[Default Namespaces 10](#_Toc31984580)

[XML HttpRequest 10](#_Toc31984581)

[The XMLHttpRequest Object 11](#_Toc31984582)

[Sending an XMLHttpRequest 11](#_Toc31984583)

[Old Versions of Internet Explorer (IE5 and IE6) 11](#_Toc31984584)

[XML Parser 11](#_Toc31984585)

[Parsing a Text String 11](#_Toc31984586)

[The XMLHttpRequest Object 12](#_Toc31984587)

[AJAX Introduction 12](#_Toc31984588)

[The XMLHttpRequest Object 13](#_Toc31984589)

[Access Across Domains 13](#_Toc31984590)

[XMLHttpRequest Object Methods 13](#_Toc31984591)

[XMLHttpRequest Object Properties 13](#_Toc31984592)

[Send a Request To a Server 14](#_Toc31984593)

[GET or POST? 14](#_Toc31984594)

[GET Requests 15](#_Toc31984595)

[POST Requests 15](#_Toc31984596)

[Server Response 15](#_Toc31984597)

[Server Response Properties 16](#_Toc31984598)

[Server Response Methods 16](#_Toc31984599)

[The getAllResponseHeaders() Method 16](#_Toc31984600)

[The getResponseHeader() Method 16](#_Toc31984601)

[XML DOM Tutorial 17](#_Toc31984602)

[Introduction 17](#_Toc31984603)

[Get the Value of an XML Element 17](#_Toc31984604)

[XML DOM Methods 17](#_Toc31984605)

[XML DOM Nodes 17](#_Toc31984606)

[Node Properties 17](#_Toc31984607)

[The nodeName Property 17](#_Toc31984608)

[The nodeValue Property 18](#_Toc31984609)

[Get the Value of an Element 18](#_Toc31984610)

[Change the Value of an Element 18](#_Toc31984611)

[The nodeType Property 18](#_Toc31984612)

[Get Node Values, Attributes, Length 18](#_Toc31984613)

[Navigating DOM Nodes 18](#_Toc31984614)

[Get an Attribute and its Value 19](#_Toc31984615)

[Change the value of nodes 19](#_Toc31984616)

[Change the Value of a Text Node 19](#_Toc31984617)

[Change the Value of an Attribute 19](#_Toc31984618)

[Change an Attribute Using setAttribute() 19](#_Toc31984619)

[Change an Attribute Using nodeValue 19](#_Toc31984620)

[XML DOM Remove Nodes 19](#_Toc31984621)

[Remove an Element Node 19](#_Toc31984622)

[Remove Myself - Remove the Current Node 19](#_Toc31984623)

[Remove a Text Node 19](#_Toc31984624)

[Clear a Text Node 19](#_Toc31984625)

[Remove an Attribute Node by Name 20](#_Toc31984626)

[Remove Attribute Nodes by Object 20](#_Toc31984627)

[XML DOM Replace Nodes 20](#_Toc31984628)

[Replace an Element Node 20](#_Toc31984629)

[Replace Data In a Text Node 20](#_Toc31984630)

[Use the nodeValue Property Instead 21](#_Toc31984631)

[XML DOM Create Nodes 21](#_Toc31984632)

[Create a New Element Node 21](#_Toc31984633)

[Create a New Attribute Node 21](#_Toc31984634)

[Create an Attribute Using setAttribute() 21](#_Toc31984635)

[Create a Text Node 21](#_Toc31984636)

[Create a CDATA Section Node 21](#_Toc31984637)

[Create a Comment Node 21](#_Toc31984638)

[XML DOM Add Nodes 21](#_Toc31984639)

[Add a Node - appendChild() 21](#_Toc31984640)

[Insert a Node - insertBefore() 22](#_Toc31984641)

[Add a New Attribute 22](#_Toc31984642)

[Add Text to a Text Node - insertData() 22](#_Toc31984643)

[XML DOM Clone Nodes 22](#_Toc31984644)

[XPath Tutorial 22](#_Toc31984645)

[XPath Syntax 22](#_Toc31984646)

[Selecting Nodes 23](#_Toc31984647)

[Predicates 23](#_Toc31984648)

[Selecting Unknown Nodes 24](#_Toc31984649)

[Selecting Several Paths 24](#_Toc31984650)

[XPath Axes 25](#_Toc31984651)

[Location Path Expression 25](#_Toc31984652)

[XSLT Introduction 26](#_Toc31984653)

[XSL(T) Languages 26](#_Toc31984654)

[It Started with XSL 26](#_Toc31984655)

[CSS = Style Sheets for HTML 26](#_Toc31984656)

[XSL = Style Sheets for XML 26](#_Toc31984657)

[XSL - More Than a Style Sheet Language 27](#_Toc31984658)

[What is XSLT? 27](#_Toc31984659)

[XSLT = XSL Transformations 27](#_Toc31984660)

[XSLT Uses XPath 27](#_Toc31984661)

[XSLT - Transformation 27](#_Toc31984662)

[Correct Style Sheet Declaration 27](#_Toc31984663)

[Link the XSL Style Sheet to the XML Document 28](#_Toc31984664)

[XSLT <xsl:template> Element 28](#_Toc31984665)

[The <xsl:template> Element 28](#_Toc31984666)

[XSLT <xsl:value-of> Element 29](#_Toc31984667)

[XSLT <xsl:for-each> Element 29](#_Toc31984668)

[Filtering the Output 30](#_Toc31984669)

[XSLT <xsl:sort> Element 30](#_Toc31984670)

[XSLT <xsl:if> Element 30](#_Toc31984671)

[XSLT <xsl:choose> Element 30](#_Toc31984672)

[XSLT <xsl:apply-templates> Element 31](#_Toc31984673)

[XSLT - On the Client 32](#_Toc31984674)

[XSLT - On the Server 33](#_Toc31984675)

[PHP Code: Transform XML to XHTML on the Server 33](#_Toc31984676)

[ASP Code: Transform XML to XHTML on the Server 34](#_Toc31984677)

[XQuery Tutorial 34](#_Toc31984678)

[What is XQuery? 34](#_Toc31984679)

[XQuery FLWOR Expressions 35](#_Toc31984680)

[DTD Tutorial 36](#_Toc31984681)

[IMPORTANT QUESTIONS 36](#_Toc31984682)

[ELEMENTS 36](#_Toc31984683)

[Declaring Elements 36](#_Toc31984684)

[Empty Elements 37](#_Toc31984685)

[Elements with Parsed Character Data 37](#_Toc31984686)

[Elements with any Contents 37](#_Toc31984687)

[Elements with Children (sequences) 37](#_Toc31984688)

[Declaring Only One Occurrence of an Element 37](#_Toc31984689)

[Declaring Minimum One Occurrence of an Element (+) 37](#_Toc31984690)

[Declaring Zero or More Occurrences of an Element(\*) 37](#_Toc31984691)

[Declaring Zero or One Occurrences of an Element (?) 38](#_Toc31984692)

[Declaring either/or Content 38](#_Toc31984693)

[Declaring Mixed Content 38](#_Toc31984694)

[ATTRIBUTES 38](#_Toc31984695)

[Declaring Attributes 38](#_Toc31984696)

[A Default Attribute Value 38](#_Toc31984697)

[The attribute-type can be one of the following: 39](#_Toc31984698)

[The attribute-value can be one of the following: 39](#_Toc31984699)

[#REQUIRED 39](#_Toc31984700)

[#IMPLIED 39](#_Toc31984701)

[#FIXED 40](#_Toc31984702)

[Enumerated Attribute Values 40](#_Toc31984703)

[Avoid using attributes? 40](#_Toc31984704)

[Examples 41](#_Toc31984705)

[Usages of \* + and ? 41](#_Toc31984706)

[Attrubutes, Entities and their usage in elements 41](#_Toc31984707)

[XML Schema Tutorial 42](#_Toc31984708)

[What is an XML Schema? 42](#_Toc31984709)

[Why Learn XML Schema? 42](#_Toc31984710)

[XML Schemas Support Data Types 42](#_Toc31984711)

[XML Schemas use XML Syntax 43](#_Toc31984712)

[XML Schemas Secure Data Communication 43](#_Toc31984713)

[Well-Formed is Not Enough 43](#_Toc31984714)

[XSD How To? 44](#_Toc31984715)

[An XML Schema example 44](#_Toc31984716)

[A Reference to a DTD 44](#_Toc31984717)

[A Reference to an XML Schema 45](#_Toc31984718)

[XSD - The <schema> Element 45](#_Toc31984719)

[XSD Simple Elements 46](#_Toc31984720)

[What is a Simple Element? 46](#_Toc31984721)

[Defining a Simple Element 46](#_Toc31984722)

[Default and Fixed Values for Simple Elements 47](#_Toc31984723)

[XSD Attributes 47](#_Toc31984724)

[What is an Attribute? 47](#_Toc31984725)

[How to Define an Attribute? 47](#_Toc31984726)

[Default and Fixed Values for Attributes 48](#_Toc31984727)

[Optional and Required Attributes 48](#_Toc31984728)

[Restrictions on Content 48](#_Toc31984729)

[XSD Restrictions/Facets 48](#_Toc31984730)

[Restrictions on Values 48](#_Toc31984731)

[Restrictions on a Set of Values 48](#_Toc31984732)

[Restrictions on a Series of Values 49](#_Toc31984733)

[Other Restrictions on a Series of Values 50](#_Toc31984734)

[Restrictions on Whitespace Characters 51](#_Toc31984735)

[Restrictions on Length 52](#_Toc31984736)

[Restrictions for Datatypes 52](#_Toc31984737)

[XSD Complex Elements 53](#_Toc31984738)

[Define complex Element: 53](#_Toc31984739)

[Empty Elements 53](#_Toc31984740)

[Elements Only 53](#_Toc31984741)

[Text-Only Elements 54](#_Toc31984742)

[Mixed Content 54](#_Toc31984743)

[Indicators 54](#_Toc31984744)

[Order indicators: 54](#_Toc31984745)

[Occurrence indicators: 54](#_Toc31984746)

[Group indicators: 54](#_Toc31984747)

[All Indicator 55](#_Toc31984748)

[Choice Indicator 55](#_Toc31984749)

[Sequence Indicator 55](#_Toc31984750)

[maxOccurs Indicator 55](#_Toc31984751)

[minOccurs Indicator 55](#_Toc31984752)

[Element Groups 56](#_Toc31984753)

[Attribute Groups 56](#_Toc31984754)

[The <any> Element 56](#_Toc31984755)

[<anyAttribute> Element 57](#_Toc31984756)

[XSD Data 57](#_Toc31984757)

[String Data Type 57](#_Toc31984758)

[Date and Time Data Types 57](#_Toc31984759)

[Date Data Type 57](#_Toc31984760)

[Time Data Type 58](#_Toc31984761)

[Numeric Data Types 58](#_Toc31984762)

[RDF 58](#_Toc31984763)

[RDF Document Example 58](#_Toc31984764)

[RDF Elements 59](#_Toc31984765)

[The <rdf:RDF> Element 59](#_Toc31984766)

[Properties as Resources 59](#_Toc31984767)

[The <rdf:Bag> Element 59](#_Toc31984768)

[The <rdf:Seq> Element 60](#_Toc31984769)

[The <rdf:Alt> Element 61](#_Toc31984770)

[RDF Collections 61](#_Toc31984771)

[Java Example: 62](#_Toc31984772)

[RDFa 63](#_Toc31984773)

[Introduction 63](#_Toc31984774)

[Attributes 63](#_Toc31984775)

[Vocab 63](#_Toc31984776)

[Typeof 63](#_Toc31984777)

[Property 64](#_Toc31984778)

[Resource 64](#_Toc31984779)

[Prefix 64](#_Toc31984780)

[SPARSQL 65](#_Toc31984781)

[Select Simple 65](#_Toc31984782)

[Select from both sources 65](#_Toc31984783)

[Filter in Query 66](#_Toc31984784)

[OWL 66](#_Toc31984785)

# XML Tutorial

## IMPORTANT QUESTIONS

**What is XML?**

* XML stands for extensible Markup Language.
* XML was designed to store and transport data.
* XML was designed to be both human- and machine-readable.
* XML is a markup language much like HTML
* XML was designed to store and transport data
* XML was designed to be self-descriptive
* XML is a W3C Recommendation

**Why Study XML?**

* XML plays an important role in many different IT systems.
* XML is often used for distributing data over the Internet.
* It is important (for all types of software developers!) to have a good understanding of XML.

**The Difference between XML and HTML?**

* XML was designed to carry data - with focus on what data is
* HTML was designed to display data - with focus on how data looks
* XML tags are not predefined like HTML tags are

**XML Syntax Rules**

* The syntax rules of XML are very simple and logical. The rules are easy to learn, and easy to use.
* XML Documents Must Have a Root Element
* The XML prolog is optional. If it exists, it must come first in the document.
* XML documents can contain international characters, like Norwegian øæå or French êèé.
* To avoid errors, you should specify the encoding used, or save your XML files as UTF-8.
* UTF-8 is the default character encoding for XML documents.
* All XML Elements Must Have a Closing Tag.
* XML Tags are Case Sensitive.
* XML Elements Must be Properly Nested.
* XML Attribute Values Must Always be Quoted.
* White-space is Preserved in XML.

## XML Elements

**What is an XML Element?**

An XML element is everything from (including) the element's start tag to (including) the element's end tag.

An element can contain:

* text
* attributes
* other elements
* or a mix of the above

### Empty XML Elements

An element with no content is said to be empty.

<element></element>

You can also use a so called self-closing tag:

<element />

### XML Naming Rules

XML elements must follow these naming rules:

* Element names are case-sensitive
* Element names must start with a letter or underscore
* Element names cannot start with the letters xml (or XML, or Xml, etc)
* Element names can contain letters, digits, hyphens, underscores, and periods
* Element names cannot contain spaces
* Any name can be used, no words are reserved (except xml).

### Best Naming Practices

* Create descriptive names, like this: <person>, <firstname>, <lastname>.
* Create short and simple names, like this: <book\_title> not like this: <the\_title\_of\_the\_book>.
* Avoid "-". If you name something "first-name", some software may think you want to subtract "name" from "first".
* Avoid ".". If you name something "first.name", some software may think that "name" is a property of the object "first".
* Avoid ":". Colons are reserved for namespaces (more later).
* Non-English letters like éòá are perfectly legal in XML, but watch out for problems if your software doesn't support them.

## XML Attributes

* XML elements can have attributes, just like HTML.
* Attributes are designed to contain data related to a specific element.
* XML Attributes Must be Quoted
* Attribute values must always be quoted. Either single or double quotes can be used.
* If the attribute value itself contains double quotes you can use single quotes, like in this example:

<gangster name='George "Shotgun" Ziegler'>

You can use character entities:

<gangster name="George &quot;Shotgun&quot; Ziegler">

### Avoid XML Attributes?

Some things to consider when using attributes are:

* attributes cannot contain multiple values (elements can)
* attributes cannot contain tree structures (elements can)
* attributes are not easily expandable (for future changes)

## XML Namespaces

XML Namespaces provide a method to avoid element name conflicts.

### Name Conflicts

In XML, element names are defined by the developer. This often results in a conflict when trying to mix XML documents from different XML applications.

Solving the Name Conflict Using a Prefix

Name conflicts in XML can easily be avoided using a name prefix.

<h:table>

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

<f:table>

<f:name>African Coffee Table</f:name>

<f:width>80</f:width>

<f:length>120</f:length>

</f:table>

### XML Namespaces - The xmlns Attribute

* When using prefixes in XML, a namespace for the prefix must be defined.
* The namespace can be defined by an xmlns attribute in the start tag of an element.
* The namespace declaration has the following syntax. xmlns:prefix="URI".

<h:table xmlns:h="http://www.w3.org/TR/html4/">

<h:tr>

<h:td>Apples</h:td>

<h:td>Bananas</h:td>

</h:tr>

</h:table>

**In the example above:**

* The xmlns attribute in the first <table> element gives the h: prefix a qualified namespace.
* The xmlns attribute in the second <table> element gives the f: prefix a qualified namespace.
* When a namespace is defined for an element, all child elements with the same prefix are associated with the same namespace.

## Uniform Resource Identifier (URI)

A Uniform Resource Identifier (URI) is a string of characters which identifies an Internet Resource.

The most common URI is the Uniform Resource Locator (URL) which identifies an Internet domain address. Another, not so common type of URI is the Uniform Resource Name (URN).

## Default Namespaces

Defining a default namespace for an element saves us from using prefixes in all the child elements. It has the following syntax:

**Example:**

<table xmlns="http://www.w3.org/TR/html4/">

<tr>

<td>Apples</td>

<td>Bananas</td>

</tr>

</table>

## XML HttpRequest

All modern browsers have a built-in XMLHttpRequest object to request data from a server.

### The XMLHttpRequest Object

The XMLHttpRequest object can be used to request data from a web server.

The XMLHttpRequest object is a developers dream, because you can:

* Update a web page without reloading the page
* Request data from a server - after the page has loaded
* Receive data from a server - after the page has loaded
* Send data to a server - in the background

### Sending an XMLHttpRequest

var xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {

if (this.readyState == 4 && this.status == 200) {

// Typical action to be performed when the document is ready:

document.getElementById("demo").innerHTML = xhttp.responseText;

}

};

xhttp.open("GET", "filename", true);

xhttp.send();

### Old Versions of Internet Explorer (IE5 and IE6)

if (window.XMLHttpRequest) {

// code for modern browsers

xmlhttp = new XMLHttpRequest();

} else {

// code for old IE browsers

xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");

}

## XML Parser

* All major browsers have a built-in XML parser to access and manipulate XML.
* The XML DOM (Document Object Model) defines the properties and methods for accessing and editing XML.
* However, before an XML document can be accessed, it must be loaded into an XML DOM object.
* All modern browsers have a built-in XML parser that can convert text into an XML DOM object.

### Parsing a Text String

This example parses a text string into an XML DOM object, and extracts the info from it with JavaScript:

<script>

var text, parser, xmlDoc;

text = "<bookstore><book>" +

"<title>Everyday Italian</title>" +

"<author>Giada De Laurentiis</author>" +

"<year>2005</year>" +

"</book></bookstore>";

parser = new DOMParser();

xmlDoc = parser.parseFromString(text,"text/xml");

document.getElementById("demo").innerHTML =

xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;

</script>

Old Versions of Internet Explorer

Example

if (window.DOMParser) {

// code for modern browsers

parser = new DOMParser();

xmlDoc = parser.parseFromString(text,"text/xml");

} else {

// code for old IE browsers

xmlDoc = new ActiveXObject("Microsoft.XMLDOM");

xmlDoc.async = false;

xmlDoc.loadXML(text);

}

### The XMLHttpRequest Object

* The XMLHttpRequest Object has a built in XML Parser.
* The responseText property returns the response as a string.
* The responseXML property returns the response as an XML DOM object.
* If you want to use the response as an XML DOM object, you can use the responseXML property.

# AJAX Introduction

AJAX is a developer's dream, because you can:

* Update a web page without reloading the page
* Request data from a server - after the page has loaded
* Receive data from a server - after the page has loaded
* Send data to a server - in the background

function loadDoc() {

var xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {

if (this.readyState == 4 && this.status == 200) {

document.getElementById("demo").innerHTML = this.responseText;

}

};

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

}

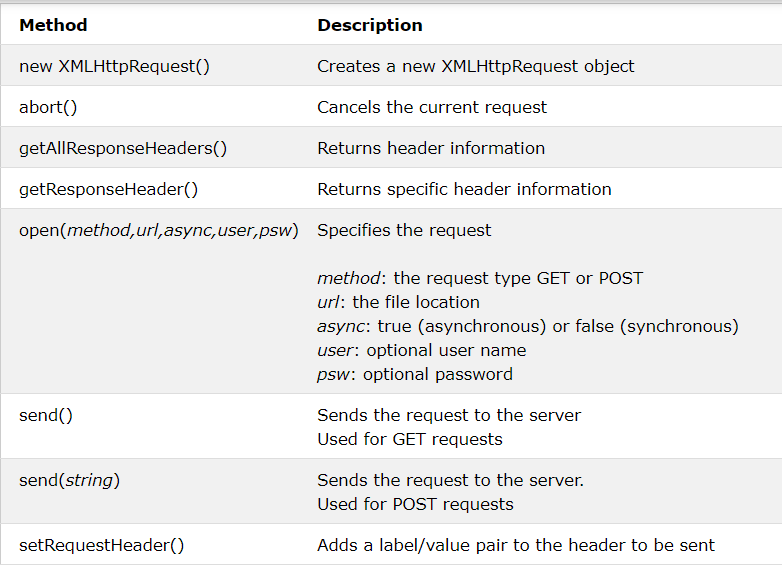
## The XMLHttpRequest Object

* All modern browsers support the XMLHttpRequest object.
* The XMLHttpRequest object can be used to exchange data with a server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

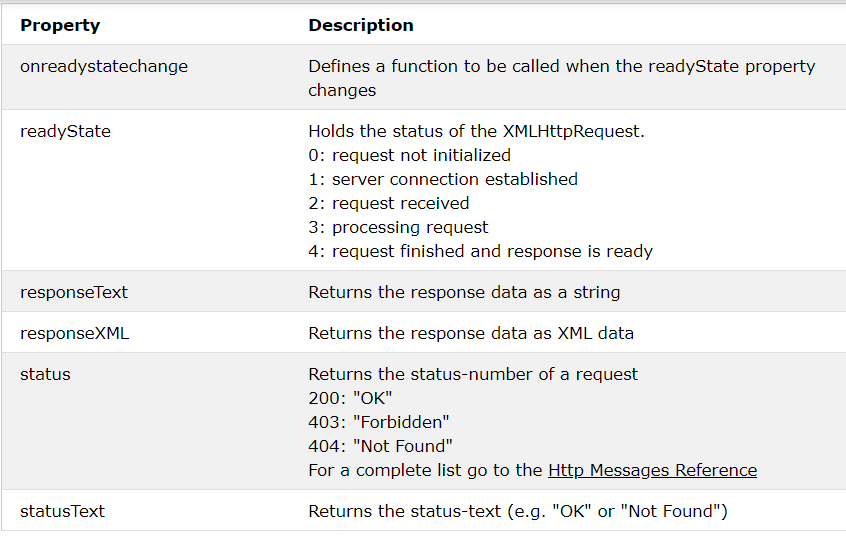
## Access Across Domains

* For security reasons, modern browsers do not allow access across domains.
* This means that both the web page and the XML file it tries to load, must be located on the same server.
* The examples on W3Schools all open XML files located on the W3Schools domain.
* If you want to use the example above on one of your own web pages, the XML files you load must be located on your own server.

## XMLHttpRequest Object Methods



## XMLHttpRequest Object Properties

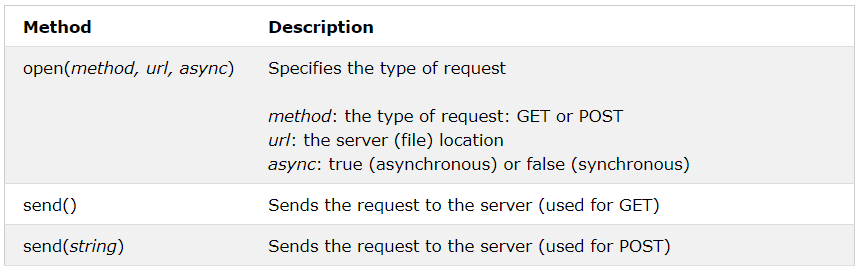


## Send a Request To a Server

To send a request to a server, we use the open() and send() methods of the XMLHttpRequest object:

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();



## GET or POST?

GET is simpler and faster than POST, and can be used in most cases.

However, always use POST requests when:

* A cached file is not an option (update a file or database on the server).
* Sending a large amount of data to the server (POST has no size limitations).
* Sending user input (which can contain unknown characters), POST is more robust and secure than GET.

## GET Requests

**Example 1:**

xhttp.open("GET", "demo\_get.asp", true);

xhttp.send();

**Example 2:**

xhttp.open("GET", "demo\_get.asp?t=" + Math.random(), true);

xhttp.send();

**Example 3:**

xhttp.open("GET", "demo\_get2.asp?fname=Henry&lname=Ford", true);

xhttp.send();

## POST Requests

**Example 1:**

xhttp.open("POST", "demo\_post.asp", true);

xhttp.send();

**Example 2:**

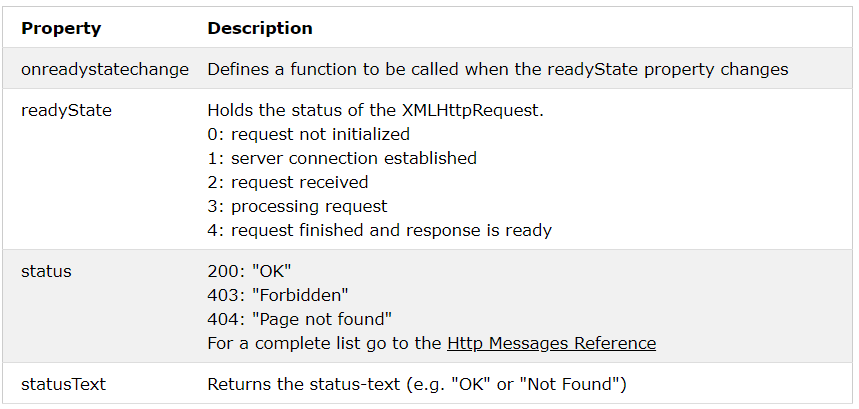
xhttp.open("POST", "demo\_post2.asp", true);

xhttp.setRequestHeader("Content-type", "application/x-www-form-urlencoded");

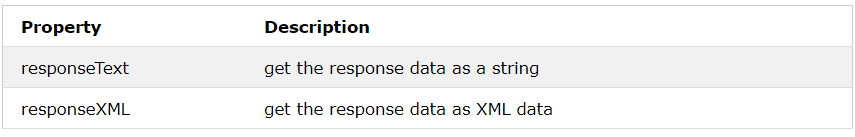
xhttp.send("fname=Henry&lname=Ford");

## Server Response

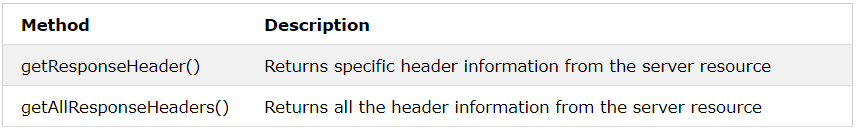
* The onreadystatechange Property
* The readyState property holds the status of the XMLHttpRequest.
* The onreadystatechange property defines a function to be executed when the readyState changes.
* The status property and the statusText property holds the status of the XMLHttpRequest object.



## Server Response Properties



## Server Response Methods



## The getAllResponseHeaders() Method

The getAllResponseHeaders() method returns all header information from the server response.

var xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {

if (this.readyState == 4 && this.status == 200) {

document.getElementById("demo").innerHTML =

this.getAllResponseHeaders();

}

};

## The getResponseHeader() Method

The getResponseHeader() method returns specific header information from the server response.

var xhttp = new XMLHttpRequest();

xhttp.onreadystatechange = function() {

if (this.readyState == 4 && this.status == 200) {

document.getElementById("demo").innerHTML =

this.getResponseHeader("Last-Modified");

}

};

xhttp.open("GET", "ajax\_info.txt", true);

xhttp.send();

# XML DOM Tutorial

## Introduction

**What is XML DOM?**

All XML elements can be accessed through the XML DOM.

The XML DOM is:

* A standard object model for XML
* A standard programming interface for XML
* Platform- and language-independent
* A W3C standard

In other words: The XML DOM is a standard for how to get, change, add, or delete XML elements.

### Get the Value of an XML Element

This code retrieves the text value of the first <title> element in an XML document:

txt = xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;

### XML DOM Methods

* x.getElementsByTagName(name) - get all elements with a specified tag name
* x.appendChild(node) - insert a child node to x
* x.removeChild(node) - remove a child node from x

## XML DOM Nodes

According to the XML DOM, everything in an XML document is a node:

* The entire document is a document node
* Every XML element is an element node
* The text in the XML elements are text nodes
* Every attribute is an attribute node
* Comments are comment nodes

## Node Properties

In the XML DOM, each node is an object.

Objects have methods and properties, that can be accessed and manipulated by JavaScript.

Three important node properties are:

* nodeName
* nodeValue
* nodeType

## The nodeName Property

The nodeName property specifies the name of a node.

* nodeName is read-only
* nodeName of an element node is the same as the tag name
* nodeName of an attribute node is the attribute name
* nodeName of a text node is always #text
* nodeName of the document node is always #document

## The nodeValue Property

The nodeValue property specifies the value of a node.

* nodeValue for element nodes is undefined
* nodeValue for text nodes is the text itself
* nodeValue for attribute nodes is the attribute value

## Get the Value of an Element

The following code retrieves the text node value of the first <title> element:

var x = xmlDoc.getElementsByTagName("title")[0].childNodes[0];

var txt = x.nodeValue;

## Change the Value of an Element

The following code changes the text node value of the first <title> element:

var x = xmlDoc.getElementsByTagName("title")[0].childNodes[0];

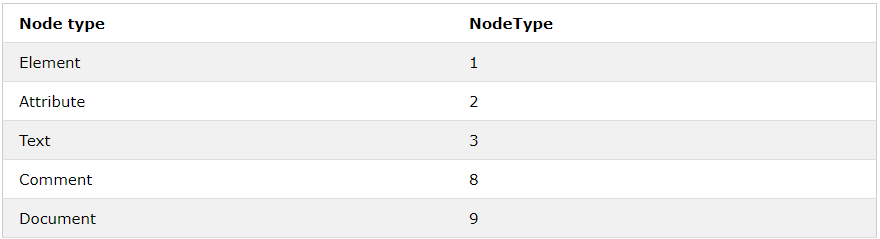
x.nodeValue = "Easy Cooking";

## The nodeType Property

The nodeType property specifies the type of node.

nodeType is read only.

The most important node types are:



## Get Node Values, Attributes, Length

var txt = x[0].childNodes[0].nodeValue;

x = xmlDoc.getElementsByTagName('title').length;

x = xmlDoc.getElementsByTagName('book')[0].attributes;

## Navigating DOM Nodes

Accessing nodes in the node tree via the relationship between nodes, is often called "navigating nodes".

In the XML DOM, node relationships are defined as properties to the nodes:

* parentNode
* childNodes
* firstChild
* lastChild
* nextSibling
* previousSibling

**Example 1:**

document.getElementById("demo").innerHTML = x.parentNode.nodeName;

**Example 2:**

document.getElementById("demo").innerHTML = x.firstChild;

## Get an Attribute and its Value

The getAttributeNode() method returns an attribute node.

x = xmlDoc.getElementsByTagName("title")[0];

y = x.getAttributeNode("lang");

txt = y.nodeValue;

## Change the value of nodes

### Change the Value of a Text Node

xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue = "new content"

## Change the Value of an Attribute

### Change an Attribute Using setAttribute()

xmlDoc.getElementsByTagName("book")[0].setAttribute("category","food");

### Change an Attribute Using nodeValue

xmlDoc.getElementsByTagName("book")[0].getAttributeNode("category").nodeValue = "food";

## XML DOM Remove Nodes

The removeChild() method removes a specified node.

The removeAttribute() method removes a specified attribute.

### Remove an Element Node

y = xmlDoc.getElementsByTagName("book")[0];

xmlDoc.documentElement.removeChild(y);

### Remove Myself - Remove the Current Node

x = xmlDoc.getElementsByTagName("book")[0];

x.parentNode.removeChild(x);

### Remove a Text Node

The removeChild() method can also be used to remove a text node:

x = xmlDoc.getElementsByTagName("title")[0];

y = x.childNodes[0];

x.removeChild(y);

### Clear a Text Node

The nodeValue property can be used to change the value of a text node:

xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue = "";

### Remove an Attribute Node by Name

The removeAttribute() method removes an attribute node by its name.

x = xmlDoc.getElementsByTagName("book");

x[0].removeAttribute("category");

### Remove Attribute Nodes by Object

The removeAttributeNode() method removes an attribute node, using the node object as parameter.

x = xmlDoc.getElementsByTagName("book");

for (i = 0; i < x.length; i++) {

while (x[i].attributes.length > 0) {

attnode = x[i].attributes[0];

old\_att = x[i].removeAttributeNode(attnode);

}

}

## XML DOM Replace Nodes

The replaceChild() method replaces a specified node.

The nodeValue property replaces text in a text node.

### Replace an Element Node

The replaceChild() method is used to replace a node.

**Example**

xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.documentElement;

//create a book element, title element and a text node

newNode=xmlDoc.createElement("book");

newTitle=xmlDoc.createElement("title");

newText=xmlDoc.createTextNode("A Notebook");

//add the text node to the title node,

newTitle.appendChild(newText);

//add the title node to the book node

newNode.appendChild(newTitle);

y=xmlDoc.getElementsByTagName("book")[0]

//replace the first book node with the new node

x.replaceChild(newNode,y);

### Replace Data In a Text Node

The replaceData() method is used to replace data in a text node.

**Example**

xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.getElementsByTagName("title")[0].childNodes[0];

x.replaceData(0,8,"Easy");

### Use the nodeValue Property Instead

It is easier to replace the data in a text node using the nodeValue property.

**Example**

xmlDoc=loadXMLDoc("books.xml");

x=xmlDoc.getElementsByTagName("title")[0].childNodes[0];

x.nodeValue="Easy Italian";

## XML DOM Create Nodes

### Create a New Element Node

The createElement() method creates a new element node:

newElement = xmlDoc.createElement("edition");

xmlDoc.getElementsByTagName("book")[0].appendChild(newElement);

### Create a New Attribute Node

The createAttribute() is used to create a new attribute node:

newAtt = xmlDoc.createAttribute("edition");

newAtt.nodeValue = "first";

xmlDoc.getElementsByTagName("title")[0].setAttributeNode(newAtt);

### Create an Attribute Using setAttribute()

Since the setAttribute() method creates a new attribute if the attribute does not exist, it can be used to create a new attribute.

xmlDoc.getElementsByTagName('book')[0].setAttribute("edition","first");

### Create a Text Node

The createTextNode() method creates a new text node:

newEle = xmlDoc.createElement("edition");

newText = xmlDoc.createTextNode("first");

newEle.appendChild(newText);

xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);

### Create a CDATA Section Node

The createCDATASection() method creates a new CDATA section node.

newCDATA = xmlDoc.createCDATASection("Special Offer & Book Sale");

xmlDoc.getElementsByTagName("book")[0].appendChild(newCDATA);

### Create a Comment Node

The createComment() method creates a new comment node.

newComment = xmlDoc.createComment("Revised March 2015");

xmlDoc.getElementsByTagName("book")[0].appendChild(newComment);

## XML DOM Add Nodes

### Add a Node - appendChild()

The appendChild() method adds a child node to an existing node.

The new node is added (appended) after any existing child nodes.

**Example**

newEle = xmlDoc.createElement("edition");

xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);

### Insert a Node - insertBefore()

The insertBefore() method inserts a node before a specified child node.

**Example**

newNode = xmlDoc.createElement("book");

x = xmlDoc.documentElement;

y = xmlDoc.getElementsByTagName("book")[3];

x.insertBefore(newNode,y);

### Add a New Attribute

The setAttribute() method sets the value of an attribute.

**Example**

xmlDoc.getElementsByTagName('book')[0].setAttribute("edition","first");

### Add Text to a Text Node - insertData()

The insertData() method inserts data into an existing text node.

**Example**

xmlDoc.getElementsByTagName("title")[0].childNodes[0].insertData(0,"Easy ");

## XML DOM Clone Nodes

Copy a Node

The cloneNode() method creates a copy of a specified node.

**Example**

oldNode = xmlDoc.getElementsByTagName('book')[0];

newNode = oldNode.cloneNode(true);

xmlDoc.documentElement.appendChild(newNode);

# XPath Tutorial

**What is XPath?**

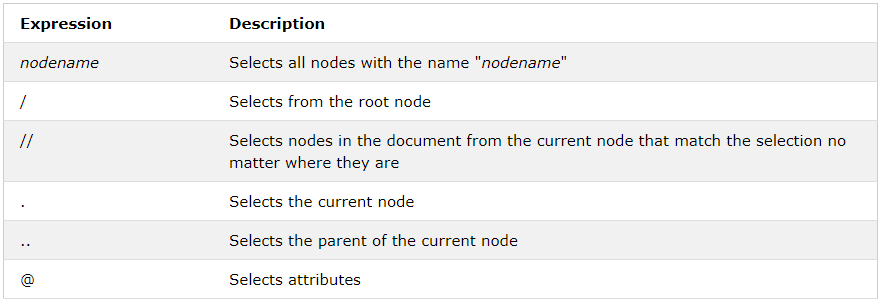
* XPath is a major element in the XSLT standard.
* XPath can be used to navigate through elements and attributes in an XML document.
* XPath stands for XML Path Language
* XPath uses "path like" syntax to identify and navigate nodes in an XML document
* XPath contains over 200 built-in functions
* XPath is a major element in the XSLT standard
* XPath is a W3C recommendation

## XPath Syntax

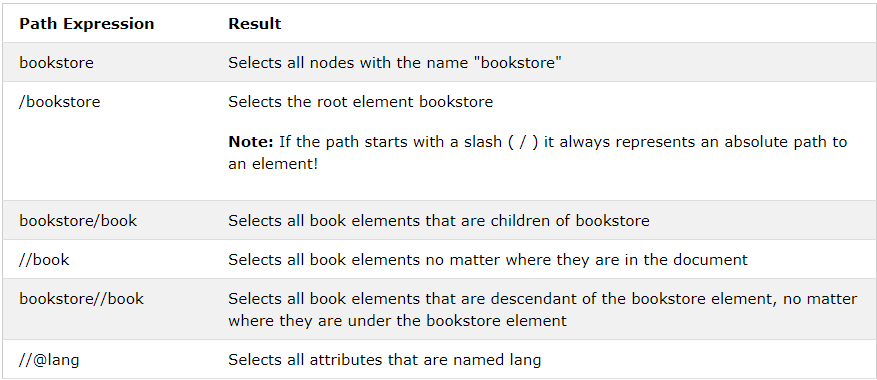
XPath uses path expressions to select nodes or node-sets in an XML document. The node is selected by following a path or steps.

## Selecting Nodes

XPath uses path expressions to select nodes in an XML document. The node is selected by following a path or steps. The most useful path expressions are listed below:



In the table below we have listed some path expressions and the result of the expressions:

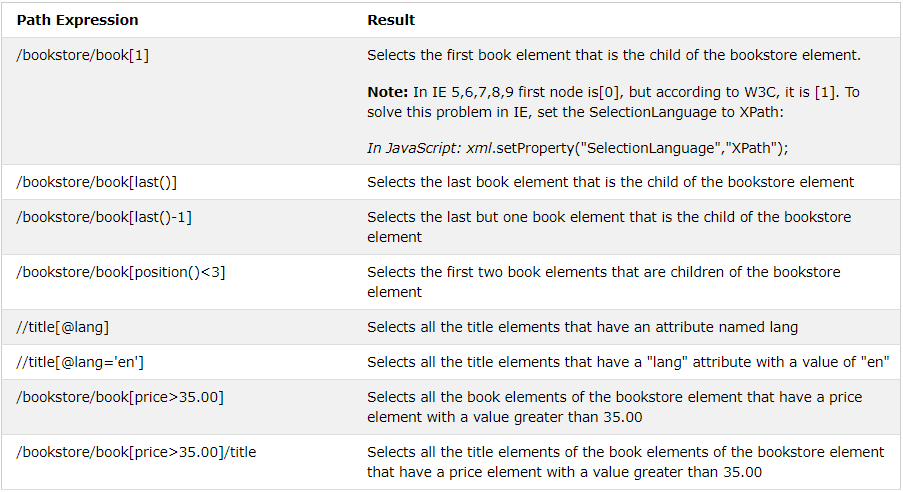


## Predicates

Predicates are used to find a specific node or a node that contains a specific value.

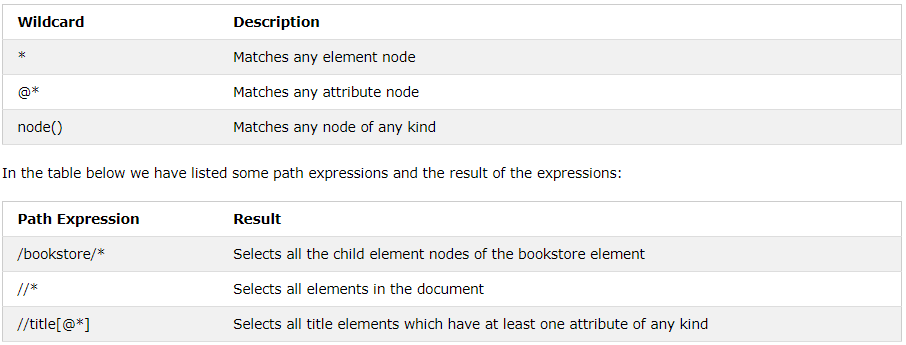
Predicates are always embedded in square brackets.

In the table below we have listed some path expressions with predicates and the result of the expressions:



## Selecting Unknown Nodes

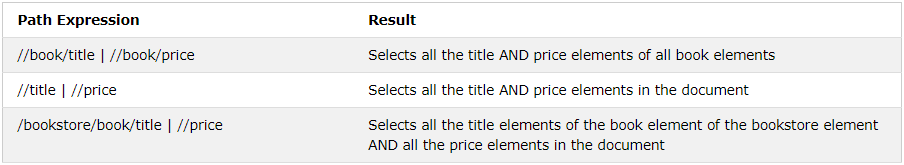
XPath wildcards can be used to select unknown XML nodes.



## Selecting Several Paths

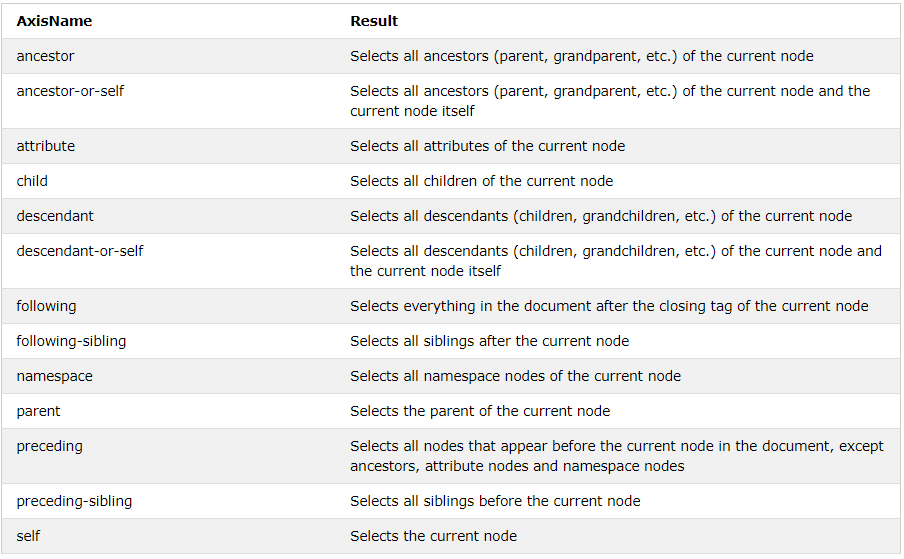
By using the | operator in an XPath expression you can select several paths.

In the table below we have listed some path expressions and the result of the expressions:



## XPath Axes

An axis represents a relationship to the context (current) node, and is used to locate nodes relative to that node on the tree.



## Location Path Expression

A location path can be absolute or relative.

An absolute location path starts with a slash ( / ) and a relative location path does not. In both cases the location path consists of one or more steps, each separated by a slash:

An absolute location path:

/step/step/...

A relative location path:

step/step/...

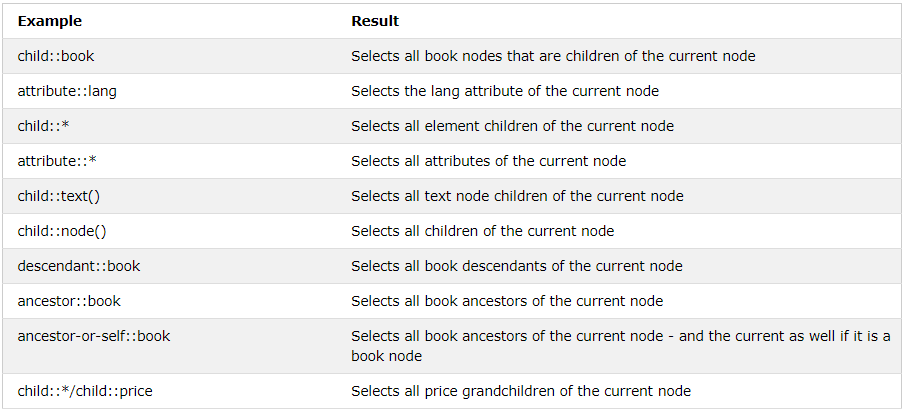
Each step is evaluated against the nodes in the current node-set.

A step consists of:

* an axis (defines the tree-relationship between the selected nodes and the current node)
* a node-test (identifies a node within an axis)
* zero or more predicates (to further refine the selected node-set)

The syntax for a location step is:

axisname::nodetest[predicate]



# XSLT Introduction

* XSL (eXtensible Stylesheet Language) is a styling language for XML.
* XSLT stands for XSL Transformations.
* This tutorial will teach you how to use XSLT to transform XML documents into other formats (like transforming XML into HTML).

## XSL(T) Languages

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

### It Started with XSL

* XSL stands for EXtensible Stylesheet Language.
* The World Wide Web Consortium (W3C) started to develop XSL because there was a need for an XML-based Stylesheet Language.

### CSS = Style Sheets for HTML

HTML uses predefined tags. The meaning of, and how to display each tag is well understood.

CSS is used to add styles to HTML elements.

### XSL = Style Sheets for XML

* XML does not use predefined tags, and therefore the meaning of each tag is not well understood.
* A <table> element could indicate an HTML table, a piece of furniture, or something else - and browsers do not know how to display it!
* So, XSL describes how the XML elements should be displayed.

### XSL - More Than a Style Sheet Language

XSL consists of four parts:

* XSLT - a language for transforming XML documents
* XPath - a language for navigating in XML documents
* XSL-FO - a language for formatting XML documents (discontinued in 2013)
* XQuery - a language for querying XML documents

### What is XSLT?

* XSLT stands for XSL Transformations
* XSLT is the most important part of XSL
* XSLT transforms an XML document into another XML document
* XSLT uses XPath to navigate in XML documents
* XSLT is a W3C Recommendation

### XSLT = XSL Transformations

XSLT is the most important part of XSL.

XSLT is used to transform an XML document into another XML document, or another type of document that is recognized by a browser, like HTML and XHTML. Normally XSLT does this by transforming each XML element into an (X)HTML element.

With XSLT you can add/remove elements and attributes to or from the output file. You can also rearrange and sort elements, perform tests and make decisions about which elements to hide and display, and a lot more.

A common way to describe the transformation process is to say that XSLT transforms an XML source-tree into an XML result-tree.

### XSLT Uses XPath

XSLT uses XPath to find information in an XML document. XPath is used to navigate through elements and attributes in XML documents.

## XSLT - Transformation

Example study: How to transform XML into XHTML using XSLT?

The details of this example will be explained in the next chapter.

### Correct Style Sheet Declaration

The root element that declares the document to be an XSL style sheet is <xsl:stylesheet> or <xsl:transform>.

**Note:** <xsl:stylesheet> and <xsl:transform> are completely synonymous and either can be used!

The correct way to declare an XSL style sheet according to the W3C XSLT Recommendation is:

**Example 1:**

<xsl:stylesheet version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

**Example 2:**

<xsl:transform version="1.0"

xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

### Link the XSL Style Sheet to the XML Document

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

<?xml-stylesheet type="text/xsl" href="cdcatalog.xsl"?>

<catalog>

.

.

</catalog>

## XSLT <xsl:template> Element

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

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).

**Example:**

<?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>

<tr>

<td>.</td>

<td>.</td>

</tr>

</table>

</body>

</html>

</xsl:template>

</xsl:stylesheet>

**Example Explained**

* Since an XSL style sheet is an XML document, it always begins with the XML declaration: <?xml version="1.0" encoding="UTF-8"?>.
* The next element, <xsl:stylesheet>, defines that this document is an XSLT style sheet document (along with the version number and XSLT namespace attributes).
* The <xsl:template> element defines a template. The match="/" attribute associates the template with the root of the XML source document.
* The content inside the <xsl:template> element defines some HTML to write to the output.
* The last two lines define the end of the template and the end of the style sheet.
* The result from this example was a little disappointing, because no data was copied from the XML document
* to the output. In the next chapter you will learn how to use the <xsl:value-of> element to select values from the XML elements.

## XSLT <xsl:value-of> Element

The <xsl:value-of> element is used to extract the value of a selected node.

The <xsl:value-of> element can be used to extract the value of an XML element and add it to the output stream of the transformation:

**Example:**

<td><xsl:value-of select="catalog/cd/title"/></td>

Example Explained

Note: The select attribute, in the example above, contains an XPath expression. An XPath expression works like navigating a file system; a forward slash (/) selects subdirectories.

The result from the example above was a little disappointing; only one line of data was copied from the XML document to the output. In the next chapter you will learn how to use the <xsl:for-each> element to loop through the XML elements, and display all of the records.

## XSLT <xsl:for-each> Element

The <xsl:for-each> element allows you to do looping in XSLT.

The XSL <xsl:for-each> element can be used to select every XML element of a specified node-set:

<xsl:for-each select="catalog/cd">

<tr>

<td><xsl:value-of select="title"/></td>

<td><xsl:value-of select="artist"/></td>

</tr>

</xsl:for-each>

**Note:** The value of the select attribute is an XPath expression. An XPath expression works like navigating a file system; where a forward slash (/) selects subdirectories.

### Filtering the Output

We can also filter the output from the XML file by adding a criterion to the select attribute in the <xsl:for-each> element.

<xsl:for-each select="catalog/cd[artist='Bob Dylan']">

Legal filter operators are:

* = (equal)
* != (not equal)
* &lt; less than
* &gt; greater than

## XSLT <xsl:sort> Element

Where to put the Sort Information

To sort the output, simply add an <xsl:sort> element inside the <xsl:for-each> element in the XSL file:

**Note:** The select attribute indicates what XML element to sort on.

## XSLT <xsl:if> Element

The <xsl:if> element is used to put a conditional test against the content of the XML file.

Syntax:

<xsl:if test="expression">

...some output if the expression is true...

</xsl:if>

**Note:** The value of the required test attribute contains the expression to be evaluated.

## XSLT <xsl:choose> Element

The <xsl:choose> element is used in conjunction with <xsl:when> and <xsl:otherwise> to express multiple conditional tests.

Syntax:

The <xsl:choose> Element

<xsl:choose>

<xsl:when test="expression">

... some output ...

</xsl:when>

<xsl:otherwise>

... some output ....

</xsl:otherwise>

</xsl:choose>

Example:

<xsl:for-each select="catalog/cd">

<tr>

<td><xsl:value-of select="title"/></td>

<xsl:choose>

<xsl:when test="price &gt; 10">

<td bgcolor="#ff00ff">

<xsl:value-of select="artist"/></td>

</xsl:when>

<xsl:otherwise>

<td><xsl:value-of select="artist"/></td>

</xsl:otherwise>

</xsl:choose>

</tr>

</xsl:for-each>

## XSLT <xsl:apply-templates> Element

The <xsl:apply-templates> element applies a template rule to the current element or to the current element's child nodes.

The <xsl:apply-templates> Element

The <xsl:apply-templates> element applies a template to the current element or to the current element's child nodes.

If we add a "select" attribute to the <xsl:apply-templates> element, it will process only the child elements that matches the value of the attribute. We can use the "select" attribute to specify in which order the child nodes are to be processed

Example:

<?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>

<xsl:apply-templates/>

</body>

</html>

</xsl:template>

<xsl:template match="cd">

<p>

<xsl:apply-templates select="title"/>

<xsl:apply-templates select="artist"/>

</p>

</xsl:template>

<xsl:template match="title">

Title: <span style="color:#ff0000">

<xsl:value-of select="."/></span>

<br />

</xsl:template>

<xsl:template match="artist">

Artist: <span style="color:#00ff00">

<xsl:value-of select="."/></span>

<br />

</xsl:template>

## XSLT - On the Client

XSLT can be used to transform the document to XHTML in your browser.

A JavaScript Solution

In the previous chapters we have explained how XSLT can be used to transform a document from XML to XHTML. We did this by adding an XSL style sheet to the XML file and let the browser do the transformation.

Even if this works fine, it is not always desirable to include a style sheet reference in an XML file (e.g. it will not work in a non XSLT aware browser.)

A more versatile solution would be to use a JavaScript to do the transformation.

By using a JavaScript, we can:

do browser-specific testing

use different style sheets according to browser and user needs

That is the beauty of XSLT! One of the design goals for XSLT was to make it possible to transform data from one format to another, supporting different browsers and different user needs.

The XML File and the XSL File

Look at the XML document that you have seen in the previous chapters:

Example:

<!DOCTYPE html>

<html>

<head>

<script>

function loadXMLDoc(filename)

{

if (window.ActiveXObject)

{

xhttp = new ActiveXObject("Msxml2.XMLHTTP");

}

else

{

xhttp = new XMLHttpRequest();

}

xhttp.open("GET", filename, false);

try {xhttp.responseType = "msxml-document"} catch(err) {} // Helping IE11

xhttp.send("");

return xhttp.responseXML;

}

function displayResult()

{

xml = loadXMLDoc("cdcatalog.xml");

xsl = loadXMLDoc("cdcatalog.xsl");

// code for IE

if (window.ActiveXObject || xhttp.responseType == "msxml-document")

{

ex = xml.transformNode(xsl);

document.getElementById("example").innerHTML = ex;

}

// code for Chrome, Firefox, Opera, etc.

else if (document.implementation && document.implementation.createDocument)

{

xsltProcessor = new XSLTProcessor();

xsltProcessor.importStylesheet(xsl);

resultDocument = xsltProcessor.transformToFragment(xml, document);

document.getElementById("example").appendChild(resultDocument);

}

}

</script>

</head>

<body onload="displayResult()">

<div id="example" />

</body>

</html>

## XSLT - On the Server

To make XML data available to all kind of browsers, we can transform the XML document on the SERVER and send it back to the browser as XHTML.

A Cross Browser Solution

In the previous chapter we explained how XSLT can be used to transform a document from XML to XHTML in the browser. We used a JavaScript and an XML parser for the transformation. However, this will not work in a browser that doesn't have an XML parser.

To make XML data available to all kind of browsers, we can transform the XML document on the server and send back to the browser as XHTML.

That's another beauty of XSLT. One of the design goals for XSLT was to make it possible to transform data from one format to another on a server, returning readable data to all kinds of browsers.

### PHP Code: Transform XML to XHTML on the Server

Example 1:

<?php

// Load XML file

$xml = new DOMDocument;

$xml->load('cdcatalog.xml');

// Load XSL file

$xsl = new DOMDocument;

$xsl->load('cdcatalog.xsl');

// Configure the transformer

$proc = new XSLTProcessor;

// Attach the xsl rules

$proc->importStyleSheet($xsl);

echo $proc->transformToXML($xml);

?>

### ASP Code: Transform XML to XHTML on the Server

Example 2:

<%

'Load XML file

set xml = Server.CreateObject("Microsoft.XMLDOM")

xml.async = false

xml.load(Server.MapPath("cdcatalog.xml"))

'Load XSL file

set xsl = Server.CreateObject("Microsoft.XMLDOM")

xsl.async = false

xsl.load(Server.MapPath("cdcatalog.xsl"))

'Transform file

Response.Write(xml.transformNode(xsl))

%>

# XQuery Tutorial

## What is XQuery?

* XQuery is to XML what SQL is to databases.
* XQuery is the language for querying XML data
* XQuery for XML is like SQL for databases
* XQuery is built on XPath expressions
* XQuery is supported by all major databases
* XQuery is a W3C Recommendation

XQuery is designed to query XML data.

**XQuery Example**

for $x in doc("books.xml")/bookstore/book

where $x/price>30

order by $x/title

return $x/title

**XQuery is About Querying XML**

XQuery is a language for finding and extracting elements and attributes from XML documents.

Here is an example of what XQuery could solve:

"Select all CD records with a price less than $10 from the CD collection stored in cd\_catalog.xml"

XQuery - Examples of Use

XQuery can be used to:

Extract information to use in a Web Service

Generate summary reports

Transform XML data to XHTML

Search Web documents for relevant information

## XQuery FLWOR Expressions

What is FLWOR?

FLWOR (pronounced "flower") is an acronym for "For, Let, Where, Order by, Return".

For - selects a sequence of nodes

Let - binds a sequence to a variable

Where - filters the nodes

Order by - sorts the nodes

Return - what to return (gets evaluated once for every node)

How to Select Nodes From "books.xml" With FLWOR

doc("books.xml")/bookstore/book[price>30]/title

Example:

for $x in doc("books.xml")/bookstore/book

where $x/price>30

return $x/title

Result:

<title lang="en">XQuery Kick Start</title>

<title lang="en">Learning XML</title>

* The **for clause** selects all book elements under the bookstore element into a variable called $x.
* The **where clause** selects only book elements with a price element with a value greater than 30.
* The **order by clause** defines the sort-order. Will be sort by the title element.
* The **return clause** specifies what should be returned. Here it returns the title elements.
* The result of the XQuery expression above will be:

# DTD Tutorial

## IMPORTANT QUESTIONS

**Q1: What is a DTD?**

A DTD is a Document Type Definition.

A DTD defines the structure and the legal elements and attributes of an XML document.

**Q2: Why Use a DTD?**

With 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.

**Q3: An Internal DTD Declaration**

If the DTD is declared inside the XML file, it must be wrapped inside the <!DOCTYPE> definition:

Example:

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

*<!DOCTYPE note [*

*<!ELEMENT note (to,from,heading,body)>*

*<!ELEMENT to (#PCDATA)>*

*<!ELEMENT from (#PCDATA)>*

*<!ELEMENT heading (#PCDATA)>*

*<!ELEMENT body (#PCDATA)>*

*]>*

**Q4: An External DTD Declaration**

If the DTD is declared in an external file, the <!DOCTYPE> definition must contain a reference to the DTD file:

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

*<!DOCTYPE note SYSTEM "note.dtd">*

*<note>*

*...*

*</note>*

**Q5: Predefined entities in XML**

&lt; < &gt; > &amp; & &quot; " &apos; '

**Q6: what is PCDATA?**

It is parsed character data which should not contain any &, <, or > characters; these need to be represented by the &amp; &lt; and &gt; entities, respectively.

**Q6: What is CDATA?**

CDATA means character data. CDATA is text that will NOT be parsed by a parser. Tags inside the text will NOT be treated as markup and entities will not be expanded.

## ELEMENTS

In a DTD, elements are declared with an ELEMENT declaration.

### Declaring Elements

In a DTD, XML elements are declared with the following syntax:

<!ELEMENT element-name category>

<!ELEMENT element-name (element-content)>

### Empty Elements

Empty elements are declared with the category keyword EMPTY:

<!ELEMENT element-name EMPTY>

<br />

### Elements with Parsed Character Data

Elements with only parsed character data are declared with #PCDATA inside parentheses:

<!ELEMENT element-name (#PCDATA)>

### Elements with any Contents

Elements declared with the category keyword ANY, can contain any combination of parsable data:

<!ELEMENT element-name ANY>

### Elements with Children (sequences)

Elements with one or more children are declared with the name of the children elements inside parentheses:

<!ELEMENT element-name (child1,child2,...)>

**Note:** When children are declared in a sequence separated by commas, the children must appear in the same sequence in the document. In a full declaration, the children must also be declared, and the children can also have children. The full declaration of the "note" element is:

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

<!ELEMENT to (#PCDATA)>

<!ELEMENT from (#PCDATA)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT body (#PCDATA)>

### Declaring Only One Occurrence of an Element

<!ELEMENT element-name (child-name)>

**Note:** The example above declares that the child element "message" must occur once, and only once inside the "note" element.

### Declaring Minimum One Occurrence of an Element (+)

<!ELEMENT element-name (child-name+)>

**Note:** The + sign in the example above declares that the child element "message" must occur one or more times inside the "note" element.

### Declaring Zero or More Occurrences of an Element(\*)

<!ELEMENT element-name (child-name\*)>

**Note:** The \* sign in the example above declares that the child element "message" can occur zero or more times inside the "note" element.

### Declaring Zero or One Occurrences of an Element (?)

<!ELEMENT element-name (child-name?)>

**Note:** The ? sign in the example above declares that the child element "message" can occur zero or one time inside the "note" element.

### Declaring either/or Content

<!ELEMENT note (to,from,header,(message|body))>

**Note:** The example above declares that the "note" element must contain a "to" element, a "from" element, a "header" element, and either a "message" or a "body" element.

### Declaring Mixed Content

<!ELEMENT note (#PCDATA|to|from|header|message)\*>

**Note:** The example above declares that the "note" element can contain zero or more occurrences of parsed character data, "to", "from", "header", or "message" elements.

## ATTRIBUTES

In a DTD, attributes are declared with an ATTLIST declaration.

### Declaring Attributes

An attribute declaration has the following syntax:

<!ATTLIST element-name attribute-name attribute-type attribute-value>

DTD example:

<!ATTLIST payment type CDATA "check">

XML example:

<payment type="check" />

### A Default Attribute Value

DTD:

<!ELEMENT square EMPTY>

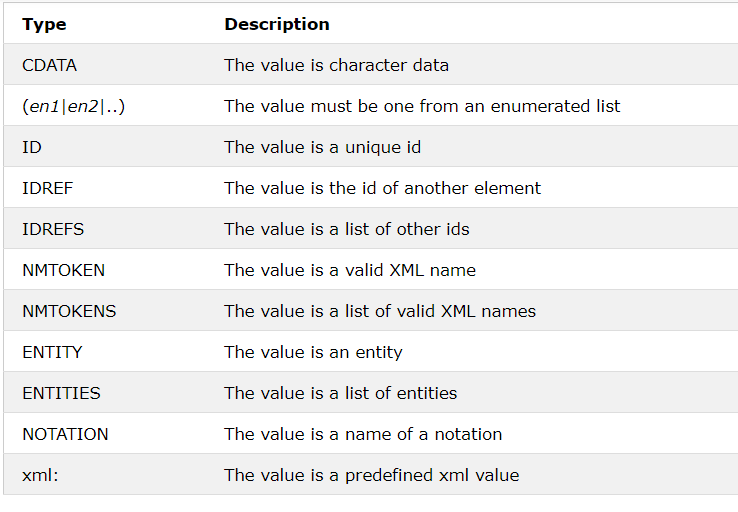
<!ATTLIST square width CDATA "0">

Valid XML:

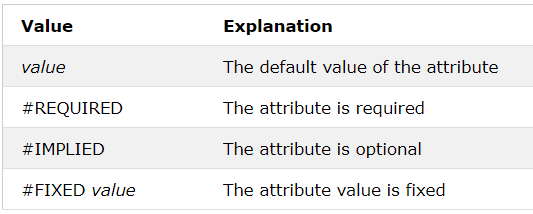
<square width="100" />

**Note:** In the example above, the "square" element is defined to be an empty element with a "width" attribute of type CDATA. If no width is specified, it has a default value of 0.

### The attribute-type can be one of the following:



### The attribute-value can be one of the following:



### #REQUIRED

Syntax

<!ATTLIST element-name attribute-name attribute-type #REQUIRED>

Example

DTD:

<!ATTLIST person number CDATA #REQUIRED>

Valid XML:

<person number="5677" />

Invalid XML:

<person />

**Note:** 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.

### #IMPLIED

Syntax

<!ATTLIST element-name attribute-name attribute-type #IMPLIED>

Example

DTD:

<!ATTLIST contact fax CDATA #IMPLIED>

Valid XML:

<contact fax="555-667788" />

Valid XML:

<contact />

**Note:** 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.

### #FIXED

Syntax

<!ATTLIST element-name attribute-name attribute-type #FIXED "value">

Example

DTD:

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

Valid XML:

<sender company="Microsoft" />

Invalid XML:

<sender company="W3Schools" />

**Note:** 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

Syntax

<!ATTLIST element-name attribute-name (en1|en2|..) default-value>

Example

DTD:

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

XML example:

<payment type="check" />

<payment type="cash" />

**Note:** Use enumerated attribute values when you want the attribute value to be one of a fixed set of legal values.

### Avoid using attributes?

Some of the problems with attributes are:

* attributes cannot contain multiple values (child elements can)
* attributes are not easily expandable (for future changes)
* attributes cannot describe structures (child elements can)
* attributes are more difficult to manipulate by program code
* attribute values are not easy to test against a DTD

## Examples

### Usages of \* + and ?

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

<!DOCTYPE note [

<!ELEMENT note (to+, from\*, heading?, body)>

<!ELEMENT to ANY>

<!ELEMENT from (#PCDATA)>

<!ELEMENT heading (#PCDATA)>

<!ELEMENT body EMPTY>

]>

<note>

<!-- + one or more times -->

<to>Sadiq</to>

<to>Sadiq</to>

<to>Sadiq</to>

<!-- zero or more times -->

<!-- <from>Muhammad</from> -->

<!-- Zero or one only occurances -->

<heading>Greetings</heading>

<!-- <heading>Greetings</heading> -->

<body/>

</note>

### Attrubutes, Entities and their usage in elements

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

<!DOCTYPE student [

<!ELEMENT student (name, class)>

<!ELEMENT name (#PCDATA)>

<!ELEMENT class (#PCDATA)>

<!ATTLIST name id CDATA "0">

<!ATTLIST name rollNo CDATA #REQUIRED>

<!ATTLIST name height CDATA #IMPLIED>

<!ATTLIST name type CDATA #FIXED "student">

<!ATTLIST name gender (male|female) #REQUIRED>

<!ENTITY writer "Muhammad Sadiq Odho">

<!ENTITY copyrights "Copyrights STutorials">

]>

<student>

<!-- id attribute is not required but it have default value -->

<!-- rollNo attribute is required -->

<!-- height attribute is implied which does not have default value

and not required attribute -->

<!-- type is a fixed value which can be changed it is not required -->

<!-- Enumerated value for gender male or female -->

<name rollNo="99" height="5.6" type="student" gender="male">Muhammad Sadiq</name>

<!-- Invalid because tyoe can not be changed -->

<!-- <name rollNo="99" height="5.6" type="teacher">Muhammad Sadiq</name> -->

<class>&writer;&copyrights;</class>

</student>

# XML Schema Tutorial

## What is an 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).

**XSD Example**

<?xml version="1.0"?>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<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 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

### Why Learn XML Schema?

In the XML world, hundreds of standardized XML formats are in daily use.

Many of these XML standards are defined by XML Schemas.

XML Schema is an XML-based (and more powerful) alternative to DTD.

### XML Schemas Support Data Types

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

### XML Schemas use XML Syntax

Another great strength about XML Schemas is that they are written in XML.

* You don't have to learn a new language
* You can use your XML editor to edit your Schema files
* You can use your XML parser to parse your Schema files
* You can manipulate your Schema with the XML DOM
* You can transform your Schema with XSLT

XML Schemas are extensible, because they are written in XML.

With an extensible Schema definition you can:

Reuse your Schema in other Schemas

Create your own data types derived from the standard types

Reference multiple schemas in the same document

## XML Schemas Secure Data Communication

When sending data from a sender to a receiver, it is essential that both parts have the same "expectations" about the content.

With XML Schemas, the sender can describe the data in a way that the receiver will understand.

A date like: "03-11-2004" will, in some countries, be interpreted as 3.November and in other countries as 11.March.

**However, an XML element with a data type like this:**

<date type="date">2004-03-11</date>

ensures a mutual understanding of the content, because the XML data type "date" requires the format "YYYY-MM-DD".

## Well-Formed is Not Enough

A well-formed XML document is a document that conforms to the XML syntax rules, like:

* it must begin with the XML declaration
* it must have one unique root element
* start-tags must have matching end-tags
* elements are case sensitive
* all elements must be closed
* all elements must be properly nested
* all attribute values must be quoted
* entities must be used for special characters

Even if documents are well-formed they can still contain errors, and those errors can have serious consequences.

Think of the following situation: you order 5 gross of laser printers, instead of 5 laser printers. With XML Schemas, most of these errors can be caught by your validating software.

## XSD How To?

XML documents can have a reference to a DTD or to an XML Schema.

### An XML Schema example

The following example is an XML Schema file called "note.xsd" that defines the elements of the XML document above ("note.xml"):

<?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. You will learn more about simple and complex types in the following chapters.

## A Reference to a DTD

This XML document has a reference to a DTD:

<?xml version="1.0"?>

<!DOCTYPE note SYSTEM

"https://www.w3schools.com/xml/note.dtd">

<note>

<to>Tove</to>

<from>Jani</from>

<heading>Reminder</heading>

<body>Don't forget me this weekend!</body>

</note>

### A Reference to an XML Schema

This XML document has a reference to an XML Schema:

<?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>

## XSD - The <schema> Element

The <schema> element is the root element of every XML Schema.

Syntax:

<?xml version="1.0"?>

<xs:schema>

...

...

</xs:schema>

The <schema> element may contain some attributes. A schema declaration often looks something like this:

<?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:schema>

**The following fragment:**

*xmlns:xs="http://www.w3.org/2001/XMLSchema"*

indicates that the elements and data types 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:

**This 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".

**This fragment:**

elementFormDefault="qualified"

indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

## XSD Simple Elements

XML Schemas define the elements of your XML files.

A simple element is an XML element that contains only text. It cannot contain any other elements or attributes.

### What is a Simple Element?

A simple element is an XML element that can contain 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.

### Defining a Simple Element

The syntax for defining a simple element is:

<xs:element name="xxx" type="yyy"/>

where xxx is the name of the element and yyy is the data type of the element.

XML Schema has a lot of built-in data types. The most common types are:

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

**Example**

Here are some XML elements:

<lastname>Refsnes</lastname>

<age>36</age>

<dateborn>1970-03-27</dateborn>

And here are the corresponding simple element definitions:

<xs:element name="lastname" type="xs:string"/>

<xs:element name="age" type="xs:integer"/>

<xs:element name="dateborn" type="xs:date"/>

### Default and Fixed Values for Simple Elements

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.

In the following example the default value is "red":

<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.

In the following example the fixed value is "red":

<xs:element name="color" type="xs:string" fixed="red"/>

## XSD Attributes

All attributes are declared as simple types.

### What is an Attribute?

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.

### How to Define an Attribute?

The syntax for defining an attribute is:

<xs:attribute name="xxx" type="yyy"/>

where xxx is the name of the attribute and yyy specifies the data type of the attribute.

XML Schema has a lot of built-in data types. The most common types are:

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

Example

Here is an XML element with an attribute:

<lastname lang="EN">Smith</lastname>

And here is the corresponding attribute definition:

<xs:attribute name="lang" type="xs:string"/>

### Default and Fixed Values for Attributes

Attributes may have a default value OR a fixed value specified.

A default value is automatically assigned to the attribute when no other value is specified.

In the following example the default value is "EN":

<xs:attribute name="lang" type="xs:string" default="EN"/>

A fixed value is also automatically assigned to the attribute, and you cannot specify another value.

In the following example the fixed value is "EN":

<xs:attribute name="lang" type="xs:string" fixed="EN"/>

### Optional and Required Attributes

Attributes are optional by default. To specify that the attribute is required, use the "use" attribute:

<xs:attribute name="lang" type="xs:string" use="required"/>

## Restrictions on Content

When an XML element or attribute has a data type defined, it puts restrictions on the element's or attribute's content.

If an XML element is of type "xs:date" and contains a string like "Hello World", the element will not validate.

With XML Schemas, you can also add your own restrictions to your XML elements and attributes. These restrictions are called facets. You can read more about facets in the next chapter.

### XSD Restrictions/Facets

Restrictions are used to define acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.

### Restrictions on Values

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>

### Restrictions on a Set of Values

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.

### Restrictions on a Series of Values

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>

The next example defines an element called "initials" with a restriction. The only acceptable value is THREE of the UPPERCASE letters from a to z:

<xs:element name="initials">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="[A-Z][A-Z][A-Z]"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The next example also defines an element called "initials" with a restriction. The only acceptable value is THREE of the LOWERCASE OR UPPERCASE letters from a to z:

<xs:element name="initials">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="[a-zA-Z][a-zA-Z][a-zA-Z]"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The next example defines an element called "choice" with a restriction. The only acceptable value is ONE of the following letters: x, y, OR z:

<xs:element name="choice">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="[xyz]"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The next example defines an element called "prodid" with a restriction. The only acceptable value is FIVE digits in a sequence, and each digit must be in a range from 0 to 9:

<xs:element name="prodid">

<xs:simpleType>

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

<xs:pattern value="[0-9][0-9][0-9][0-9][0-9]"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

### Other Restrictions on a Series of Values

The example below defines an element called "letter" with a restriction. The acceptable value is zero or more occurrences of 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>

The next example also defines an element called "letter" with a restriction. The acceptable value is one or more pairs of letters, each pair consisting of a lower case letter followed by an upper case letter. For example, "sToP" will be validated by this pattern, but not "Stop" or "STOP" or "stop":

<xs:element name="letter">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="([a-z][A-Z])+"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The next example defines an element called "gender" with a restriction. The only acceptable value is male OR female:

<xs:element name="gender">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="male|female"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

The next example defines an element called "password" with a restriction. There must be 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:element name="password">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:pattern value="[a-zA-Z0-9]{8}"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

### Restrictions on Whitespace Characters

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>

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "replace", which means that the XML processor WILL REPLACE all white space characters (line feeds, tabs, spaces, and carriage returns) with spaces:

<xs:element name="address">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:whiteSpace value="replace"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

This example also defines an element called "address" with a restriction. The whiteSpace constraint is set to "collapse", which means that the XML processor WILL REMOVE all white space characters (line feeds, tabs, spaces, carriage returns are replaced with spaces, leading and trailing spaces are removed, and multiple spaces are reduced to a single space):

<xs:element name="address">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:whiteSpace value="collapse"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

### Restrictions on Length

To limit the length of a value in an element, we would use the length, maxLength, and minLength constraints.

This example defines an element called "password" with a restriction. The value must be exactly eight characters:

<xs:element name="password">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:length value="8"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

This example defines another element called "password" with a restriction. The value must be minimum five characters and maximum eight characters:

<xs:element name="password">

<xs:simpleType>

<xs:restriction base="xs:string">

<xs:minLength value="5"/>

<xs:maxLength value="8"/>

</xs:restriction>

</xs:simpleType>

</xs:element>

### Restrictions for Datatypes

* Constraint Description
* enumeration Defines a list of acceptable values
* fractionDigits Specifies the maximum number of decimal places allowed. Must be equal to or greater than zero
* length Specifies the exact number of characters or list items allowed. Must be equal to or greater than zero
* maxExclusive Specifies the upper bounds for numeric values (the value must be less than this value)
* maxInclusive Specifies the upper bounds for numeric values (the value must be less than or equal to this value)
* maxLength Specifies the maximum number of characters or list items allowed. Must be equal to or greater than zero
* minExclusive Specifies the lower bounds for numeric values (the value must be greater than this value)
* minInclusive Specifies the lower bounds for numeric values (the value must be greater than or equal to this value)
* minLength Specifies the minimum number of characters or list items allowed. Must be equal to or greater than zero
* pattern Defines the exact sequence of characters that are acceptable
* totalDigits Specifies the exact number of digits allowed. Must be greater than zero
* whiteSpace Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

# XSD Complex Elements

## Define complex Element:

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

## Empty Elements

<xs:element name="product" type="prodtype"/>

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

## Elements Only

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

## Text-Only Elements

<xs:element name="somename">  
  <xs:complexType>  
    <xs:simpleContent>  
      <xs:extension base="basetype">  
        ....  
        ....  
      </xs:extension>  
    </xs:simpleContent>  
  </xs:complexType>  
</xs:element>

## Mixed Content

<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>

**Example:**

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

## Indicators

There are seven indicators:

### Order indicators:

* All
* Choice
* Sequence

### Occurrence indicators:

* maxOccurs
* minOccurs

### Group indicators:

* Group name
* attributeGroup name

## All Indicator

The <all> indicator specifies that the child elements can appear in any order, and that each child element must occur only once:

<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:complexType>  
</xs:element>

Note: 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 <minOccurs> and <maxOccurs> are described later).

## Choice Indicator

The <choice> indicator specifies that either one child element or another can occur:

<xs:element name="person">  
  <xs:complexType>  
    <xs:choice>  
      <xs:element name="employee" type="employee"/>  
      <xs:element name="member" type="member"/>  
    </xs:choice>  
  </xs:complexType>  
</xs:element>

## Sequence Indicator

The <sequence> indicator specifies that the child elements must appear in a specific order:

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

## maxOccurs Indicator

The <maxOccurs> indicator specifies the maximum number of times an element can occur:

<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="full\_name" type="xs:string"/>  
      <xs:element name="child\_name" type="xs:string" maxOccurs="10"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>

## minOccurs Indicator

The <minOccurs> indicator specifies the minimum number of times an element can occur:

<xs:element name="person">  
  <xs:complexType>  
    <xs:sequence>  
      <xs:element name="full\_name" type="xs:string"/>  
      <xs:element name="child\_name" type="xs:string"  
      maxOccurs="10" minOccurs="0"/>  
    </xs:sequence>  
  </xs:complexType>  
</xs:element>

## Element Groups

Element groups are defined with the group declaration, like this:

<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>  
  
<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

## Attribute Groups

Attribute groups are defined with the attributeGroup declaration, like this:

<xs:attributeGroup name="personattrgroup">  
  <xs:attribute name="firstname" type="xs:string"/>  
  <xs:attribute name="lastname" type="xs:string"/>  
  <xs:attribute name="birthday" type="xs:date"/>  
</xs:attributeGroup>

## The <any> Element

The <any> element enables us to extend the XML document with elements not specified by the schema.

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

## <anyAttribute> Element

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

**Example:**

<xs:attribute name="eyecolor">  
  <xs:simpleType>  
    <xs:restriction base="xs:string">  
      <xs:pattern value="blue|brown|green|grey"/>  
    </xs:restriction>  
  </xs:simpleType>  
</xs:attribute>  
  
</xs:schema>

# XSD Data

## String Data Type

<xs:element name="customer" type="xs:string"/>

* **ID:** A string that represents the ID attribute in XML (only used with schema attributes)
* **IDREF:** A string that represents the IDREF attribute in XML (only used with schema attributes)
* **Language:** A string that contains a valid language id
* **Name:** A string that contains a valid XML name
* **NMTOKEN:** A string that represents the NMTOKEN attribute in XML (only used with schema attributes)
* **normalizedString:** A string that does not contain line feeds, carriage returns, or tabs
* **string:** A string
* **token:** A string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces

## Date and Time Data Types

### Date Data Type

<xs:element name="start" type="xs:date"/>

<start>2002-09-24</start>

### Time Data Type

* hh indicates the hour
* mm indicates the minute
* ss indicates the second

<xs:element name="start" type="xs:time"/>

<start>09:00:00</start>

## Numeric Data Types

<xs:element name="prize" type="xs:decimal"/>

<prize>999.50</prize>

* **Byte:** A signed 8-bit integer
* **decimal:** A decimal value
* **int:** A signed 32-bit integer
* **integer:** An integer value
* **long:** A signed 64-bit integer
* **negativeInteger:** An integer containing only negative values (..,-2,-1)
* **nonNegativeInteger:** An integer containing only non-negative values (0,1,2,..)
* **nonPositiveInteger:** An integer containing only non-positive values (..,-2,-1,0)
* **positiveInteger:** An integer containing only positive values (1,2,..)
* **short:** A signed 16-bit integer
* **unsignedLong:** An unsigned 64-bit integer
* **unsignedInt:** An unsigned 32-bit integer
* **unsignedShort:** An unsigned 16-bit integer
* **unsignedByte:** An unsigned 8-bit integer

# RDF

**What is RDF?**

* RDF stands for Resource Description Framework
* RDF is a framework for describing resources on the web
* RDF is designed to be read and understood by computers
* RDF is not designed for being displayed to people
* RDF is written in XML
* RDF is a part of the W3C's Semantic Web Activity
* RDF is a W3C Recommendation from 10. February 2004

## RDF Document Example

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:si="https://www.w3schools.com/rdf/">  
  
<rdf:Description rdf:about="https://www.w3schools.com">  
  <si:title>W3Schools</si:title>  
  <si:author>Jan Egil Refsnes</si:author>  
</rdf:Description>  
  
</rdf:RDF>

## RDF Elements

The main elements of RDF are the root element, <RDF>, and the <Description> element, which identifies a resource.

### The <rdf:RDF> Element

**<rdf:RDF>** is the root element of an RDF document. It defines the XML document to be an RDF document. It also contains a reference to the RDF namespace:

* The <rdf:Description> Element
* The <rdf:Description> element identifies a resource with the about attribute.
* The <rdf:Description> element contains elements that describe the resource:

## Properties as Resources

The property elements can also be defined as resources:

Example:

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:cd="http://www.recshop.fake/cd#">  
  
<rdf:Description  
rdf:about="http://www.recshop.fake/cd/Empire Burlesque">  
  <cd:artist rdf:resource="http://www.recshop.fake/cd/dylan" />  
  ...  
  ...  
</rdf:Description>  
  
</rdf:RDF>

RDF Containers

RDF containers are used to describe group of things.

The following RDF elements are used to describe groups: <Bag>, <Seq>, and <Alt>.

## The <rdf:Bag> Element

The <rdf:Bag> element is used to describe a list of values that do not have to be in a specific order.

The <rdf:Bag> element may contain duplicate values.

**Example:**

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:cd="http://www.recshop.fake/cd#">  
  
<rdf:Description  
rdf:about="http://www.recshop.fake/cd/Beatles">  
  <cd:artist>  
    <rdf:Bag>  
      <rdf:li>John</rdf:li>  
      <rdf:li>Paul</rdf:li>  
      <rdf:li>George</rdf:li>  
      <rdf:li>Ringo</rdf:li>  
    </rdf:Bag>  
  </cd:artist>  
</rdf:Description>  
  
</rdf:RDF>

## The <rdf:Seq> Element

The <rdf:Seq> element is used to describe an ordered list of values (For example, in alphabetical order).

The <rdf:Seq> element may contain duplicate values.

**Example**

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:cd="http://www.recshop.fake/cd#">  
  
<rdf:Description  
rdf:about="http://www.recshop.fake/cd/Beatles">  
  <cd:artist>  
    <rdf:Seq>  
      <rdf:li>George</rdf:li>  
      <rdf:li>John</rdf:li>  
      <rdf:li>Paul</rdf:li>  
      <rdf:li>Ringo</rdf:li>  
    </rdf:Seq>  
  </cd:artist>  
</rdf:Description>  
  
</rdf:RDF>

## The <rdf:Alt> Element

The <rdf:Alt> element is used to describe a list of alternative values (the user can select only one of the values).

**Example**

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:cd="http://www.recshop.fake/cd#">  
  
<rdf:Description  
rdf:about="http://www.recshop.fake/cd/Beatles">  
  <cd:format>  
    <rdf:Alt>  
      <rdf:li>CD</rdf:li>  
      <rdf:li>Record</rdf:li>  
      <rdf:li>Tape</rdf:li>  
    </rdf:Alt>  
  </cd:format>  
</rdf:Description>  
  
</rdf:RDF>

## RDF Collections

RDF collections describe groups that can ONLY contain the specified members.

The rdf:parseType="Collection" Attribute

As seen in the previous chapter, a container says that the containing resources are members - it does not say that other members are not allowed.

RDF collections are used to describe groups that can ONLY contain the specified members.

A collection is described by the attribute rdf:parseType="Collection".

<?xml version="1.0"?>  
  
<rdf:RDF  
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"  
xmlns:cd="http://recshop.fake/cd#">  
  
<rdf:Description  
rdf:about="http://recshop.fake/cd/Beatles">  
  <cd:artist rdf:parseType="Collection">  
    <rdf:Description rdf:about="http://recshop.fake/cd/Beatles/George"/>  
    <rdf:Description rdf:about="http://recshop.fake/cd/Beatles/John"/>  
    <rdf:Description rdf:about="http://recshop.fake/cd/Beatles/Paul"/>  
    <rdf:Description rdf:about="http://recshop.fake/cd/Beatles/Ringo"/>  
  </cd:artist>  
</rdf:Description>  
  
</rdf:RDF>

## Java Example:

import java.io.\*;

import org.apache.jena.rdf.model.\*;

import org.apache.jena.riot.lang.\*;

import org.apache.jena.riot.Lang;

import org.apache.jena.riot.RDFDataMgr;

public class Tutorial7\_Task2 extends Object{

public static void main(String[] args) {

Model authorModel = RDFDataMgr.loadModel("authors.rdf");

Model booksModel = RDFDataMgr.loadModel("books.rdf");

Model all = booksModel.union(authorModel);

ResIterator iterator = all.listResourcesWithProperty(Litera.hasWritten);

if(iterator.hasNext()){

while(iterator.hasNext()){

Resource author = (Resource) iterator.nextResource();

System.out.print(author.getProperty(Litera.authorName).getString());

System.out.println(" wrote: ");

StmtIterator bookList = author.listProperties(Litera.hasWritten);

if(bookList.hasNext()){

while(bookList.hasNext()){

Resource book = (Resource) bookList.nextStatement().getObject();

System.out.println("\t"+book.getProperty(Litera.bookName).getString());

}

} else {

System.out.println("No book");

}

}

} else {

System.out.println("No Author");

}

// RDFDataMgr.write(System.out, authorModel, Lang.RDFXML);

}}

# RDFa

Introduction:

The full RDFa syntax [[rdfa-core](https://www.w3.org/TR/rdfa-lite/#bib-rdfa-core)] provides a number of basic and advanced features that enable authors to express fairly complex structured data, such as relationships among people, places, and events in an HTML or XML document. Some of these advanced features may make it difficult for authors, who may not be experts in structured data, to use RDFa. This lighter version of RDFa is a gentler introduction to the world of structured data, intended for authors that want to express fairly simple data in their web pages. The goal is to provide a minimal subset that is easy to learn and will work for 80% of authors doing simple data markup.

## Attributes

RDFa Lite consists of five simple attributes; **vocab**, **typeof**, **property**, **resource**, and **prefix**. RDFa 1.1 Lite is completely upwards compatible with the full set of RDFa 1.1 attributes. This means that if an author finds that RDFa Lite isn't powerful enough, transitioning to the full version of RDFa is just a matter of adding the more powerful RDFa attributes into the existing RDFa Lite markup.

### Vocab

RDFa, like Microformats [[microformats](https://www.w3.org/TR/rdfa-lite/#bib-microformats)] and Microdata [[microdata](https://www.w3.org/TR/rdfa-lite/#bib-microdata)], enables us to talk about things on the Web such that a machine can understand what we are saying. Typically when we talk about a thing, we use a particular vocabulary to talk about it. So, if you wanted to talk about People, the vocabulary that you would use would specify terms like name and telephone number. When we want to mark up things on the Web, we need to do something very similar, which is specify which vocabulary that we are going to be using. Here is a simple example that specifies a vocabulary that we intend to use to markup things in the paragraph:

EXAMPLE 1

<p **vocab="http://schema.org/"**>

My name is Manu Sporny and you can give me a ring via 1-800-555-0199.

</p>

### Typeof

In this example we have specified that we are going to be using the vocabulary that can be found at http://schema.org/. This is a vocabulary that has been released by major search engine companies to talk about common things on the Web that Search Engines care about – things like People, Places, Reviews, Recipes, and Events. Once we have specified the vocabulary, we need to specify the type of the thing that we're talking about. In this particular case we are talking about a Person, which can be marked up like so:

EXAMPLE 2

<p vocab="http://schema.org/" **typeof="Person"**>

My name is Manu Sporny and you can give me a ring via 1-800-555-0199.

</p>

## Property

Now all we need to do is specify which properties of that person we want to point out to the search engine. In the following example, we mark up the person's name, phone number and web page. Both text and URLs can be marked up with RDFa Lite. In the following example, pay particular attention to the types of data that are being pointed out to the search engine, which are highlighted in blue:

EXAMPLE 3

<p vocab="http://schema.org/" typeof="Person">

My name is

<span property="name">Manu Sporny</span>

and you can give me a ring via

<span **property="telephone"**>1-800-555-0199</span>

or visit

<a property="url" href="http://manu.sporny.org/">my homepage</a>.

</p>

## Resource

If you want Web authors to be able to talk about each thing on your page, you need to create an identifier for each of these things. Just like we create identifiers for parts of a page using the id attribute in HTML, you can create identifiers for things described on a page using the resource attribute:

EXAMPLE 4

<p vocab="http://schema.org/" **resource="#manu"** typeof="Person">

My name is

<span property="name">Manu Sporny</span>

and you can give me a ring via

<span property="telephone">1-800-555-0199</span>.

<img property="image" src="http://manu.sporny.org/images/manu.png" />

</p>

If we assume that the markup above can be found at http://example.org/people, then the identifier for the thing is the address, plus the value in the resource attribute. Therefore, the identifier for the thing on the page would be: **http://example.org/people#manu**. This identifier is also useful if you want to talk about that same thing on another Web page. By identifying all things on the Web using a unique Uniform Resource Locator (URL), we can start building a Web of things. Companies building software for the Web can use this Web of things to answer complex questions like: "What is Manu Sporny's phone number and what does he look like?"

### Prefix

In some cases, a vocabulary may not have all of the terms an author needs when describing their thing. The last feature in RDFa 1.1 Lite that some authors might need is the ability to specify more than one vocabulary. For example, if we are describing a Person and we need to specify that they have a favorite animal, we could do something like the following:

EXAMPLE 5

<p vocab="http://schema.org/" **prefix="ov: http://open.vocab.org/terms/"** resource="#manu" typeof="Person">

My name is

<span property="name">Manu Sporny</span>

and you can give me a ring via

<span property="telephone">1-800-555-0199</span>.

<img property="image" src="http://manu.sporny.org/images/manu.png" />

My favorite animal is the <span **property="ov:preferredAnimal"**>Liger</span>.

</p>

The example assigns a short-hand prefix to the Open Vocabulary (ov) and uses that prefix to specify the preferredAnimal vocabulary term. Since schema.org doesn't have a clear way of expressing a favorite animal, the author instead depends on this alternate vocabulary to get the job done.

RDFa 1.1 Lite also pre-defines a number of [useful and popular prefixes](http://www.w3.org/2011/rdfa-context/rdfa-1.1), such as dc, foaf, and schema. This ensures that even if authors forget to declare the popular prefixes, that their structured data will continue to work. A full list of pre-declared prefixes can be found in the [initial context document for RDFa 1.1](http://www.w3.org/2011/rdfa-context/rdfa-1.1).

# SPARSQL

## Select Simple

# Select capital name of Dresden

PREFIX dbo: <http://dbpedia.org/ontology/>

SELECT ?capital

WHERE{

?capital dbo:capital ?Dresden .

}

## Select from both sources

# Select personalities who born in Dresden or Leipzig

PREFIX dbo: <http://dbpedia.org/ontology/>

PREFIX dbr: <http://dbpedia.org/resource/>

SELECT ?who ?city

FROM <http://dbpedia.org/data/Dresden.rdf/>

FROM <http://dbpedia.org/data/Leipzig.rdf/>

{

{

?who dbo:birthPlace dbr:Dresden .

}

UNION

{

?who dbo:birthPlace dbr:Leipzig .

}

}

## Filter in Query

# Select personalities who died in Dresden or Leipzig which have english label names

PREFIX dbo: <http://dbpedia.org/ontology/>

PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>

SELECT ?who ?labelname

#FROM <http://dbpedia.org/data/Dresden.rdf/>

#FROM <http://dbpedia.org/data/Leipzig.rdf/>

{

?who dbo:deathPlace ?city .

?city rdfs:label ?labelname .

FILTER(lang(?labelname) = "en")

}

# OWL