

XML Tutorial

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 Example 1

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

[Display the XML File »](#)

[Display the XML File as a Note »](#)

XML Example 2

```
<?xml version="1.0" encoding="UTF-8"?>
<breakfast_menu>
<food>
  <name>Belgian Waffles</name>
  <price>$5.95</price>
  <description>
    Two of our famous Belgian Waffles with plenty of real maple syrup
  </description>
  <calories>650</calories>
</food>
<food>
  <name>Strawberry Belgian Waffles</name>
  <price>$7.95</price>
  <description>
    Light Belgian waffles covered with strawberries and whipped cream
  </description>
  <calories>900</calories>
</food>
<food>
  <name>Berry-Berry Belgian Waffles</name>
```

```
<price>$8.95</price>
<description>
Belgian waffles covered with assorted fresh berries and whipped cream
</description>
<calories>900</calories>
</food>
<food>
<name>French Toast</name>
<price>$4.50</price>
<description>
Thick slices made from our homemade sourdough bread
</description>
<calories>600</calories>
</food>
<food>
<name>Homestyle Breakfast</name>
<price>$6.95</price>
<description>
Two eggs, bacon or sausage, toast, and our ever-popular hash browns
</description>
<calories>950</calories>
</food>
</breakfast_menu>
```

[Display the XML File »](#)

[Display with XSLT »](#)

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.

What You Will Learn

This tutorial will give you a solid understanding of XML:

What is XML?

How does XML work?

How can I use XML?

What can I use XML for?

Important XML Standards

This tutorial will also dig deep into the following important XML standards:

- [XML AJAX](#)
- [XML DOM](#)
- [XML XPath](#)
- [XML XSLT](#)
- [XML XQuery](#)
- [XML DTD](#)
- [XML Schema](#)
- [XML Services](#)

We recommend reading this tutorial, in the sequence listed in the left menu.

Learn by Examples

Examples are better than 1000 words. Examples are often easier to understand than text explanations.

This tutorial supplements all explanations with clarifying "Try it Yourself" examples.

- [XML Examples](#)
- [AJAX Examples](#)
- [DOM Examples](#)
- [XPath Examples](#)
- [XSLT Examples](#)

If you try all the examples, you will learn a lot about XML in a very short time!

XML Quiz Test

Test your XML skills at W3Schools!

[Start the XML Quiz!](#)

Introduction to XML

XML is a software- and hardware-independent tool for storing and transporting data.

What is XML?

- XML stands for eXtensible Markup Language
- 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

XML Does Not DO Anything

Maybe it is a little hard to understand, but XML does not DO anything.

This note is a note to Tove from Jani, stored as XML:

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

The XML above is quite self-descriptive:

- It has sender information.
- It has receiver information
- It has a heading
- It has a message body.

But still, the XML above does not DO anything. XML is just information wrapped in tags.

Someone must write a piece of software to send, receive, store, or display it:

Note

To: Tove

From: Jani

Reminder

Don't forget me this weekend!

The Difference Between XML and HTML

XML and HTML were designed with different goals:

- 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 Does Not Use Predefined Tags

The XML language has no predefined tags.

The tags in the example above (like `<to>` and `<from>`) are not defined in any XML standard.

These tags are "invented" by the author of the XML document.

HTML works with predefined tags like `<p>`, `<h1>`, `<table>`, etc.

With XML, the author must define both the tags and the document structure.

XML is Extensible

Most XML applications will work as expected even if new data is added (or removed).

Imagine an application designed to display the original version of note.xml (`<to>` `<from>` `<heading>` `<body>`).

Then imagine a newer version of note.xml with added `<date>` and `<hour>` elements, and a removed `<heading>`.

The way XML is constructed, older version of the application can still work:

```
<note>
  <date>2015-09-01</date>
  <hour>08:30</hour>
  <to>Tove</to>
  <from>Jani</from>
  <body>Don't forget me this weekend!</body>
</note>
```

Old Version

Note

To: Tove
From: Jani

Reminder

Don't forget me this weekend!

New Version

Note

To: Tove
From: Jani
Date: 2015-09-01 08:30
Don't forget me this weekend!

XML Simplifies Things

- It simplifies data sharing
- It simplifies data transport
- It simplifies platform changes
- It simplifies data availability

Many computer systems contain data in incompatible formats. Exchanging data between incompatible systems (or upgraded systems) is a time-consuming task for web developers. Large amounts of data must be converted, and incompatible data is often lost.

XML stores data in plain text format. This provides a software- and hardware-independent way of storing, transporting, and sharing data.

XML also makes it easier to expand or upgrade to new operating systems, new applications, or new browsers, without losing data.

With XML, data can be available to all kinds of "reading machines" like people, computers, voice machines, news feeds, etc.

XML is a W3C Recommendation

XML became a W3C Recommendation as early as in February 1998.

How Can XML be Used?

XML is used in many aspects of web development.

XML is often used to separate data from presentation.

XML Separates Data from Presentation

XML does not carry any information about how to be displayed.

The same XML data can be used in many different presentation scenarios.

Because of this, with XML, there is a full separation between data and presentation.

XML is Often a Complement to HTML

In many HTML applications, XML is used to store or transport data, while HTML is used to format and display the same data.

XML Separates Data from HTML

When displaying data in HTML, you should not have to edit the HTML file when the data changes.

With XML, the data can be stored in separate XML files.

With a few lines of JavaScript code, you can read an XML file and update the data content of any HTML page.

[Display Books.xml »](#)

Books.xml

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

    <book category="cooking">
        <title lang="en">Everyday Italian</title>
        <author>Giada De Laurentiis</author>
        <year>2005</year>
        <price>30.00</price>
    </book>

    <book category="children">
        <title lang="en">Harry Potter</title>
        <author>J K. Rowling</author>
        <year>2005</year>
        <price>29.99</price>
    </book>

    <book category="web">
        <title lang="en">XQuery Kick Start</title>
        <author>James McGovern</author>
        <author>Per Bothner</author>
        <author>Kurt Cagle</author>
        <author>James Linn</author>
        <author>Vaidyanathan Nagarajan</author>
        <year>2003</year>
        <price>49.99</price>
    </book>

    <book category="web" cover="paperback">
        <title lang="en">Learning XML</title>
        <author>Erik T. Ray</author>
        <year>2003</year>
        <price>39.95</price>
    </book>

</bookstore>
```

You will learn a lot more about using XML and JavaScript in the DOM section of this tutorial.

Transaction Data

Thousands of XML formats exists, in many different industries, to describe day-to-day data transactions:

Stocks and Shares

Financial transactions

Medical data

Mathematical data

Scientific measurements

News information

Weather services

Example: XML News

XMLNews is a specification for exchanging news and other information.

Using a standard makes it easier for both news producers and news consumers to produce, receive, and archive any kind of news information across different hardware, software, and programming languages.

An example XMLNews document:

```
<?xml version="1.0" encoding="UTF-8"?>
<nitf>
  <head>
    <title>Colombia Earthquake</title>
  </head>
  <body>
    <headline>
      <h1>143 Dead in Colombia Earthquake</h1>
    </headline>
    <byline>
      <bytag>By Jared Kotler, Associated Press Writer</bytag>
    </byline>
    <dateline>
      <location>Bogota, Colombia</location>
      <date>Monday January 25 1999 7:28 ET</date>
    </dateline>
  </body>
</nitf>
```

Example: XML Weather Service

An XML national weather service from NOAA (National Oceanic and Atmospheric Administration):

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

<credit>NOAA's National Weather Service</credit>
<credit_URL>http://weather.gov/</credit_URL>

<image>
  <url>http://weather.gov/images/xml_logo.gif</url>
  <title>NOAA's National Weather Service</title>
  <link>http://weather.gov</link>
</image>

<location>New York/John F. Kennedy Intl Airport, NY</location>
<station_id>KJFK</station_id>
<latitude>40.66</latitude>
<longitude>-73.78</longitude>
<observation_time_rfc822>Mon, 11 Feb 2008 06:51:00 -0500 EST
</observation_time_rfc822>

<weather>A Few Clouds</weather>
<temp_f>11</temp_f>
<temp_c>-12</temp_c>
<relative_humidity>36</relative_humidity>
<wind_dir>West</wind_dir>
<wind_degrees>280</wind_degrees>
<wind_mph>18.4</wind_mph>
<wind_gust_mph>29</wind_gust_mph>
<pressure_mb>1023.6</pressure_mb>
<pressure_in>30.23</pressure_in>
<dewpoint_f>-11</dewpoint_f>
<dewpoint_c>-24</dewpoint_c>
<windchill_f>-7</windchill_f>
<windchill_c>-22</windchill_c>
<visibility_mi>10.00</visibility_mi>

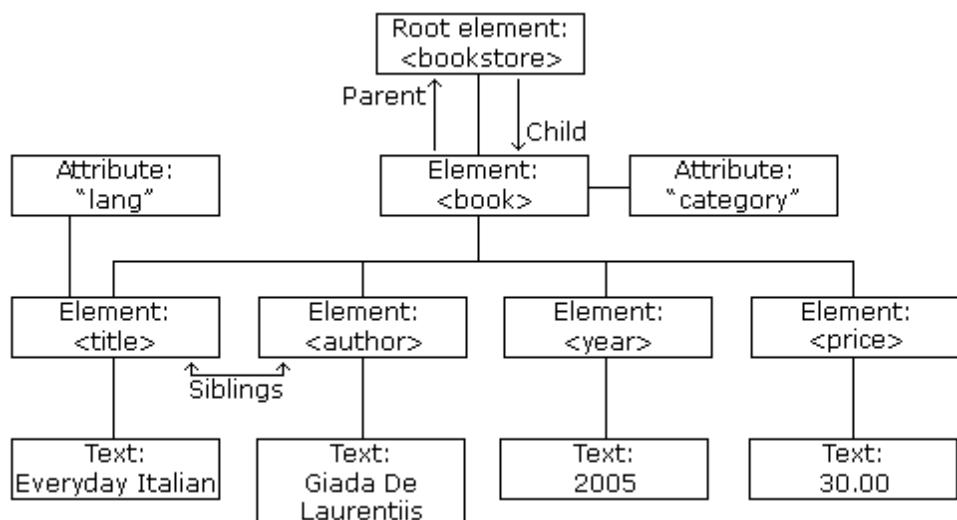
<icon_url_base>http://weather.gov/weather/images/fcicons/</icon_url_base>
<icon_url_name>nfew.jpg</icon_url_name>
<disclaimer_url>http://weather.gov/disclaimer.html</disclaimer_url>
<copyright_url>http://weather.gov/disclaimer.html</copyright_url>

</current_observation>
```

XML Tree

XML documents form a tree structure that starts at "the root" and branches to "the leaves".

XML Tree Structure



An Example XML Document

The image above represents books in this XML:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

XML Tree Structure

XML documents are formed as **element trees**.

An XML tree starts at a **root element** and branches from the root to **child elements**.

All elements can have sub elements (child elements):

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

The terms parent, child, and sibling are used to describe the relationships between elements.

Parents have children. Children have parents. Siblings are children on the same level (brothers and sisters).

All elements can have text content (Harry Potter) and attributes (category="cooking").

Self-Describing Syntax

XML uses a much self-describing syntax.

A prolog defines the XML version and the character encoding:

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

The next line is the **root element** of the document:

```
<bookstore>
```

The next line starts a <book> element:

```
<book category="cooking">
```

The <book> elements have **4 child elements**: <title>, <author>, <year>, <price>.

```
<title lang="en">Everyday Italian</title>
<author>Giada De Laurentiis</author>
<year>2005</year>
<price>30.00</price>
```

The next line ends the book element:

```
</book>
```

You can assume, from this example, that the XML document contains information about books in a bookstore.

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

XML documents must contain one **root** element that is the **parent** of all other elements:

```
<root>
  <child>
    <subchild>.....</subchild>
  </child>
</root>
```

In this example **<note>** is the root element:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

The XML Prolog

This line is called the XML **prolog**:

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

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.

Character encoding can be studied in our [Character Set Tutorial](#).

UTF-8 is also the default encoding for HTML5, CSS, JavaScript, PHP, and SQL.

All XML Elements Must Have a Closing Tag

In XML, it is illegal to omit the closing tag. All elements **must** have a closing tag:

```
<p>This is a paragraph.</p>
<br />
```

Note: The XML prolog does not have a closing tag! This is not an error. The prolog is not a part of the XML document.

XML Tags are Case Sensitive

XML tags are case sensitive. The tag `<Letter>` is different from the tag `<letter>`.

Opening and closing tags must be written with the same case:

```
<message>This is correct</message>
```

"Opening and closing tags" are often referred to as "Start and end tags". Use whatever you prefer. It is exactly the same thing.

XML Elements Must be Properly Nested

In HTML, you might see improperly nested elements:

```
<b><i>This text is bold and italic</b></i>
```

In XML, all elements **must** be properly nested within each other:

```
<b><i>This text is bold and italic</i></b>
```

In the example above, "Properly nested" simply means that since the `<i>` element is opened inside the `` element, it must be closed inside the `` element.

XML Attribute Values Must Always be Quoted

XML elements can have attributes in name/value pairs just like in HTML.

In XML, the attribute values must always be quoted:

```
<note date="12/11/2007">  
  <to>Tove</to>  
  <from>Jani</from>  
</note>
```

Entity References

Some characters have a special meaning in XML.

If you place a character like "<" inside an XML element, it will generate an error because the parser interprets it as the start of a new element.

This will generate an XML error:

```
<message>salary < 1000</message>
```

To avoid this error, replace the "<" character with an **entity reference**:

```
<message>salary &lt; 1000</message>
```

There are 5 pre-defined entity references in XML:

<	<	less than
>	>	greater than
&	&	ampersand
'	'	apostrophe
"	"	quotation mark

Only < and & are strictly illegal in XML, but it is a good habit to replace > with > as well.

Comments in XML

The syntax for writing comments in XML is similar to that of HTML:

```
<!-- This is a comment -->
```

Two dashes in the middle of a comment are not allowed:

```
<!-- This is an invalid -- comment -->
```

White-space is Preserved in XML

XML does not truncate multiple white-spaces (HTML truncates multiple white-spaces to one single white-space):

XML:	Hello	Tove
HTML:	Hello Tove	

XML Stores New Line as LF

Windows applications store a new line as: carriage return and line feed (CR+LF).

Unix and Mac OSX uses LF.

Old Mac systems uses CR.

XML stores a new line as LF.

Well Formed XML

XML documents that conform to the syntax rules above are said to be "Well Formed" XML documents.

XML Elements

An XML document contains 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.

```
<price>29.99</price>
```

An element can contain:

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

```
<bookstore>
  <book category="children">
    <title>Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title>Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

In the example above:

<title>, <author>, <year>, and <price> have **text content** because they contain text (like 29.99).

<bookstore> and <book> have **element contents**, because they contain elements.

<book> has an **attribute** (category="children").

Empty XML Elements

An element with no content is said to be empty.

In XML, you can indicate an empty element like this:

```
<element></element>
```

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

```
<element />
```

The two forms produce identical results in XML software (Readers, Parsers, Browsers).

Empty elements can have attributes.

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.

Naming Styles

There are no naming styles defined for XML elements. But here are some commonly used:

Style	Example	Description
Lower case	<firstname>	All letters lower case
Upper case	<FIRSTNAME>	All letters upper case
Underscore	<first_name>	Underscore separates words
Pascal case	<FirstName>	Uppercase first letter in each word
Camel case	<firstName>	Uppercase first letter in each word except the first

If you choose a naming style, it is good to be consistent!

XML documents often have a corresponding database. A common practice is to use the naming rules of the database for the XML elements.

Camel case is a common naming rule in JavaScripts.

XML Elements are Extensible

XML elements can be extended to carry more information.

Look at the following XML example:

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

Let's imagine that we created an application that extracted the `<to>`, `<from>`, and `<body>` elements from the XML document to produce this output:

MESSAGE

To: Tove **From:** Jani

Don't forget me this weekend!

Imagine that the author of the XML document added some extra information to it:

```
<note>
  <date>2008-01-10</date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Should the application break or crash?

No. The application should still be able to find the `<to>`, `<from>`, and `<body>` elements in the XML document and produce the same output.

This is one of the beauties of XML. It can be extended without breaking applications.

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.

For a person's gender, the <person> element can be written like this:

```
<person gender="female">
```

or like this:

```
<person gender='female'>
```

If the attribute value itself contains double quotes you can use single quotes, like in this example:

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

or you can use character entities:

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

XML Elements vs. Attributes

Take a look at these examples:

```
<person gender="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <gender>female</gender>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example gender is an attribute. In the last, gender is an element. Both examples provide the same information.

There are no rules about when to use attributes or when to use elements in XML.

My Favorite Way

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="2008-01-10">
  <to>Tove</to>
  <from>Jani</from>
</note>
```

A <date> element is used in the second example:

```
<note>
  <date>2008-01-10</date>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

```
<note>
  <date>
    <year>2008</year>
    <month>01</month>
    <day>10</day>
  </date>
  <to>Tove</to>
  <from>Jani</from>
</note>
```

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)

Don't end up like this:

```
<note day="10" month="01" year="2008"
      to="Tove" from="Jani" heading="Reminder"
      body="Don't forget me this weekend!">
</note>
```

XML Attributes for Metadata

Sometimes ID references are assigned to elements. These IDs can be used to identify XML elements in much the same way as the id attribute in HTML. This example demonstrates this:

```
<messages>
  <note id="501">
    <to>Tove</to>
    <from>Jani</from>
    <heading>Reminder</heading>
    <body>Don't forget me this weekend!</body>
  </note>
  <note id="502">
    <to>Jani</to>
    <from>Tove</from>
    <heading>Re: Reminder</heading>
    <body>I will not</body>
  </note>
</messages>
```

The id attributes above are for identifying the different notes. It is not a part of the note itself. What I'm trying to say here is that metadata (data about data) should be stored as attributes, and the data itself should be stored as elements.

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.

This XML carries HTML table information:

```
<table>
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a table (a piece of furniture):

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

If these XML fragments were added together, there would be a name conflict. Both contain a `<table>` element, but the elements have different content and meaning.

A user or an XML application will not know how to handle these differences.

Solving the Name Conflict Using a Prefix

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

This XML carries information about an HTML table, and a piece of furniture:

```
<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>
```

In the example above, there will be no conflict because the two `<table>` elements have different names.

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"`.

```
<root>

<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>

<f:table xmlns:f="https://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>

</root>
```

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.

Namespaces can also be declared in the XML root element:

```
<root xmlns:h="http://www.w3.org/TR/html4/"  
      xmlns:f="https://www.w3schools.com/furniture">  
  
<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>  
  
</root>
```

Note: The namespace URI is not used by the parser to look up information.

The purpose of using an URI is to give the namespace a unique name.

However, companies often use the namespace as a pointer to a web page containing namespace information.

Uniform Resource Identifier (URI)

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:

```
xmlns="namespaceURI"
```

This XML carries HTML table information:

```
<table xmlns="http://www.w3.org/TR/html4/">
  <tr>
    <td>Apples</td>
    <td>Bananas</td>
  </tr>
</table>
```

This XML carries information about a piece of furniture:

```
<table xmlns="https://www.w3schools.com/furniture">
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

Namespaces in Real Use

XSLT is a language that can be used to transform XML documents into other formats. The XML document below, is a document used to transform XML into HTML. The namespace "http://www.w3.org/1999/XSL/Transform" identifies XSLT elements inside an HTML document:

```
<?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>
<th style="text-align:left">Title</th>
<th style="text-align:left">Artist</th>
</tr>
<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>
</table>
</body>
</html>
</xsl:template>

</xsl:stylesheet>
```

If you want to learn more about XSLT, please read our [XSLT Tutorial](#).

Displaying XML

Raw XML files can be viewed in all major browsers.
Don't expect XML files to be displayed as HTML pages.

Viewing XML Files

```
<?xml version="1.0" encoding="UTF-8"?>
- < note >
  <to> Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Look at the XML file above in your browser: [note.xml](#)

Most browsers will display an XML document with color-coded elements.

Often a plus (+) or minus sign (-) to the left of the elements can be clicked to expand or collapse the element structure.

To view raw XML source, try to select "View Page Source" or "View Source" from the browser menu.

Note: In Safari 5 (and earlier), only the element text will be displayed. To view the raw XML, you must right click the page and select "View Source".

Viewing an Invalid XML File

If an erroneous XML file is opened, some browsers will report the error, and some will display it, or display it incorrectly.

```
<?xml version="1.0" encoding="UTF-8"?>
- <note>
  <to>Tove</to>
  <from>Jani</Ffrom>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Try to open the following XML file: [note_error.xml](#)

Other XML Examples

Viewing some XML documents will help you get the XML feeling:

[An XML breakfast menu](#) This is a breakfast food menu from a restaurant, stored as XML.

[An XML CD catalog](#) This is a CD collection, stored as XML.

[An XML plant catalog](#) This is a plant catalog from a plant shop, stored as XML.

Why Does XML Display Like This?

XML documents do not carry information about how to display the data.

Since XML tags are "invented" by the author of the XML document, browsers do not know if a tag like <table> describes an HTML table or a dining table.

Without any information about how to display the data, the browsers can just display the XML document as it is.

Tip: If you want to style an XML document, use [XSLT](#).

XMLHttpRequest

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

XMLHttpRequest Example

When you type a character in the input field below, an XMLHttpRequest is sent to the server, and some name suggestions are returned (from the server):

Example

Start typing a name in the input field below:

Name: Suggestions:

The example above is explained in the AJAX chapters of this tutorial.

Sending an XMLHttpRequest

A common JavaScript syntax for using the XMLHttpRequest object looks much like this:

Example

```
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();
```

Example Explained

The first line in the example above creates an **XMLHttpRequest** object:

```
var xhttp = new XMLHttpRequest();
```

The **onreadystatechange** property specifies a function to be executed every time the status of the XMLHttpRequest object changes:

```
xhttp.onreadystatechange = function()
```

When **readyState** property is 4 and the **status** property is 200, the response is ready:

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

The **responseText** property returns the server response as a text string.

The text string can be used to update a web page:

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

You will learn a lot more about the XMLHttpRequest object in the AJAX chapters of this tutorial.

Old Versions of Internet Explorer (IE5 and IE6)

Old versions of Internet Explorer (IE5 and IE6) do not support the XMLHttpRequest object. To handle IE5 and IE6, check if the browser supports the XMLHttpRequest object, or else create an ActiveXObject:

Example

```
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:

Example

```
<html>
<body>

<p id="demo"></p>

<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>

</body>
</html>
```

Example Explained

A text string is defined:

```
text = "<bookstore><book>" +  
"<title>Everyday Italian</title>" +  
"<author>Giada De Laurentiis</author>" +  
"<year>2005</year>" +  
"</book></bookstore>;"
```

An XML DOM parser is created:

```
parser = new DOMParser();
```

The parser creates a new XML DOM object using the text string:

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

Old Versions of Internet Explorer

Old versions of Internet Explorer (IE5, IE6, IE7, IE8) do not support the `DOMParser` object. To handle older versions of Internet Explorer, check if the browser supports the `DOMParser` object, or else create an `ActiveXObject`:

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.

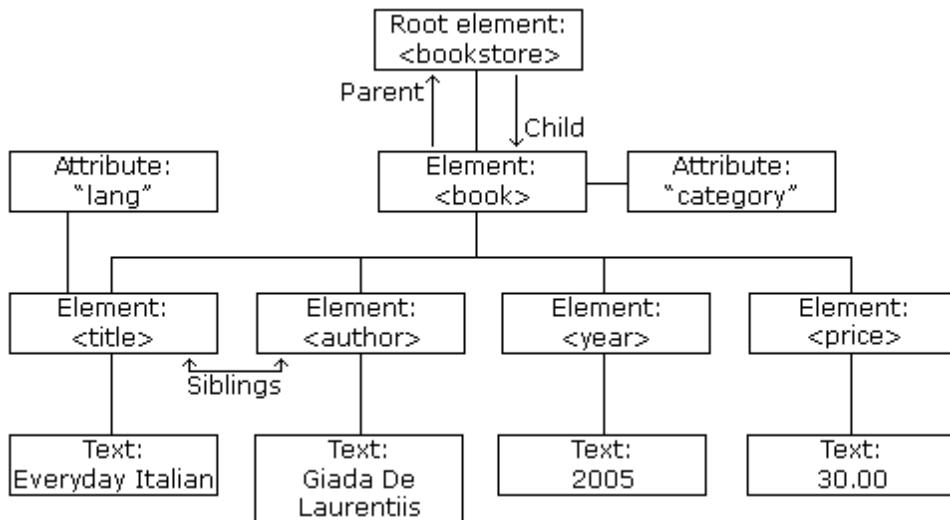
Example

Request the file [cd_catalog.xml](#) and use the response as an XML DOM object:

```
xmlDoc = xmlhttp.responseXML;  
txt = "";  
x = xmlDoc.getElementsByTagName("ARTIST");  
for (i = 0; i < x.length; i++) {  
    txt += x[i].childNodes[0].nodeValue + "<br>";  
}  
document.getElementById("demo").innerHTML = txt;
```

XML DOM

XML DOM



What is the DOM?

The DOM defines a standard for accessing and manipulating documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The HTML DOM defines a standard way for accessing and manipulating HTML documents. It presents an HTML document as a tree-structure.

The XML DOM defines a standard way for accessing and manipulating XML documents. It presents an XML document as a tree-structure.

Understanding the DOM is a must for anyone working with HTML or XML.

The HTML DOM

All HTML elements can be accessed through the HTML DOM.

This example changes the value of an HTML element with id="demo":

Example

```
<h1 id="demo">This is a Heading</h1>

<button type="button"
onclick="document.getElementById('demo').innerHTML = 'Hello World!'">Click
Me!
</button>
```

You can learn a lot more about the HTML DOM in our [JavaScript tutorial](#).

The XML DOM

All XML elements can be accessed through the XML DOM.

Books.xml

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

<book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
</book>

<book category="children">
    <title lang="en">Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
</book>

</bookstore>
```

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

Example

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

The XML DOM is a standard for how to get, change, add, and delete XML elements.

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

Example

```
<html>
<body>

<p id="demo"></p>

<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>

</body>
</html>
```

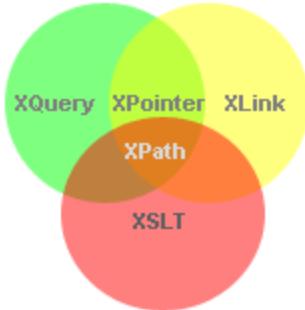
You will learn a lot more about the XML DOM in our [XML DOM Tutorial](#).

XML and XPath

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 is a syntax for defining parts of an XML document
- XPath uses path expressions to navigate in XML documents
- XPath contains a library of standard functions
- XPath is a major element in XSLT and in XQuery
- XPath is a W3C recommendation

XPath Path Expressions

XPath uses path expressions to select nodes or node-sets in an XML document. These path expressions look very much like the expressions you see when you work with a traditional computer file system.

XPath expressions can be used in JavaScript, Java, XML Schema, PHP, Python, C and C++, and lots of other languages.

XPath is Used in XSLT

XPath is a major element in the XSLT standard.

With XPath knowledge you will be able to take great advantage of XSL.

XPath Example

We will use the following XML document:

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

<bookstore>

<book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
</book>

<book category="children">
    <title lang="en">Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
</book>

<book category="web">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
    <year>2003</year>
    <price>49.99</price>
</book>

<book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
</book>

</bookstore>
```

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

XPath Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

XPath Tutorial

You will learn a lot more about XPath in our [XPath Tutorial](#).

XML and XSLT

With XSLT you can transform an XML document into HTML.

Displaying XML with XSLT

XSLT (eXtensible Stylesheet Language Transformations) is the recommended style sheet language for XML.

XSLT is far more sophisticated than CSS. 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.

XSLT uses XPath to find information in an XML document.

XSLT Example

We will use the following XML document:

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

<food>
<name>Belgian Waffles</name>
<price>$5.95</price>
<description>Two of our famous Belgian Waffles with plenty of real maple
syrup</description>
<calories>650</calories>
</food>

<food>
<name>Strawberry Belgian Waffles</name>
<price>$7.95</price>
<description>Light Belgian waffles covered with strawberries and whipped
cream</description>
<calories>900</calories>
</food>

<food>
<name>Berry-Berry Belgian Waffles</name>
<price>$8.95</price>
<description>Light Belgian waffles covered with an assortment of fresh
berries and whipped cream</description>
<calories>900</calories>
```

```
</food>

<food>
<name>French Toast</name>
<price>$4.50</price>
<description>Thick slices made from our homemade sourdough
bread</description>
<calories>600</calories>
</food>

<food>
<name>Homestyle Breakfast</name>
<price>$6.95</price>
<description>Two eggs, bacon or sausage, toast, and our ever-popular hash
browns</description>
<calories>950</calories>
</food>

</breakfast_menu>
```

Use XSLT to transform XML into HTML, before it is displayed in a browser:

Example XSLT Stylesheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<html xsl:version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
<body style="font-family:Arial;font-size:12pt;background-color:#EEEEEE">
<xsl:for-each select="breakfast_menu/food">
  <div style="background-color:teal;color:white;padding:4px">
    <span style="font-weight:bold"><xsl:value-of select="name"/> - </span>
    <xsl:value-of select="price"/>
  </div>
  <div style="margin-left:20px;margin-bottom:1em;font-size:10pt">
    <p>
      <xsl:value-of select="description"/>
      <span style="font-style:italic"> (<xsl:value-of select="calories"/>
calories per serving)</span>
    </p>
  </div>
</xsl:for-each>
</body>
</html>
```

Transform the XML Document with XSLT »

XSLT Tutorial

If you want to learn more about XSLT, go to our [XSLT Tutorial](#).

XML and XQuery

What is XQuery?

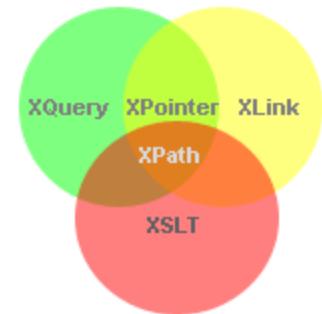
XQuery is to XML what SQL is to databases.

XQuery was 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
```

What is XQuery?



- 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 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 and XPath

XQuery 1.0 and XPath 2.0 share the same data model and support the same functions and operators. If you have already studied XPath you will have no problems with understanding XQuery.

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 is a W3C Recommendation

XQuery is compatible with several W3C standards, such as XML, Namespaces, XSLT, XPath, and XML Schema.

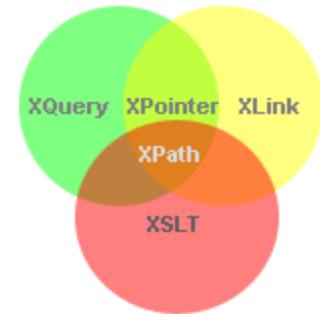
XQuery 1.0 became a W3C Recommendation in 2007.

XQuery Tutorial

You will learn a lot more about XQuery in our [XQuery Tutorial](#).

XML, XLink and XPointer

XLink is used to create hyperlinks in XML documents.



- XLink is used to create hyperlinks within XML documents
- Any element in an XML document can behave as a link
- With XLink, the links can be defined outside the linked files
- XLink is a W3C Recommendation

XLink Browser Support

There is no browser support for XLink in XML documents.

However, all major browsers support XLinks in SVG.

XLink Syntax

In HTML, the `<a>` element defines a hyperlink. However, this is not how it works in XML. In XML documents, you can use whatever element names you want - therefore it is impossible for browsers to predict what link elements will be called in XML documents.

Below is a simple example of how to use XLink to create links in an XML document:

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

<homepages xmlns:xlink="http://www.w3.org/1999/xlink">
    <homepage xlink:type="simple"
xlink:href="https://www.w3schools.com">Visit W3Schools</homepage>
    <homepage xlink:type="simple" xlink:href="http://www.w3.org">Visit
W3C</homepage>
</homepages>
```

To get access to the XLink features we must declare the XLink namespace. The XLink namespace is: "http://www.w3.org/1999/xlink".

The `xlink:type` and the `xlink:href` attributes in the `<homepage>` elements come from the XLink namespace.

The `xlink:type="simple"` creates a simple "HTML-like" link (means "click here to go there").

The `xlink:href` attribute specifies the URL to link to.

XLink Example

The following XML document contains XLink features:

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

<bookstore xmlns:xlink="http://www.w3.org/1999/xlink">

<book title="Harry Potter">
  <description
    xlink:type="simple"
    xlink:href="/images/HPotter.gif"
    xlink:show="new">
    As his fifth year at Hogwarts School of Witchcraft and
    Wizardry approaches, 15-year-old Harry Potter is.....
  </description>
</book>

<book title="XQuery Kick Start">
  <description
    xlink:type="simple"
    xlink:href="/images/XQuery.gif"
    xlink:show="new">
    XQuery Kick Start delivers a concise introduction
    to the XQuery standard.....
  </description>
</book>

</bookstore>
```

Example explained:

- The XLink namespace is declared at the top of the document (`xmlns:xlink="http://www.w3.org/1999/xlink"`)
- The `xlink:type="simple"` creates a simple "HTML-like" link
- The `xlink:href` attribute specifies the URL to link to (in this case - an image)
- The `xlink:show="new"` specifies that the link should open in a new window

XLink - Going Further

In the example above we have demonstrated simple XLinks. XLink is getting more interesting when accessing remote locations as resources, instead of standalone pages.

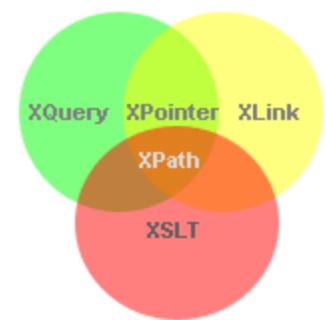
If we set the value of the `xlink:show` attribute to "embed", the linked resource should be processed inline within the page. When you consider that this could be another XML document you could, for example, build a hierarchy of XML documents.

You can also specify WHEN the resource should appear, with the `xlink:actuate` attribute.

XLink Attribute Reference

Attribute	Value	Description
<code>xlink:actuate</code>	onLoad onRequest other none	Defines when the linked resource is read and shown: <ul style="list-style-type: none">• onLoad - the resource should be loaded and shown when the document loads• onRequest - the resource is not read or shown before the link is clicked
<code>xlink:href</code>	<i>URL</i>	Specifies the URL to link to
<code>xlink:show</code>	embed new replace other none	Specifies where to open the link. Default is "replace"
<code>xlink:type</code>	simple extended locator arc resource title none	Specifies the type of link

XPointer



- XPointer allows links to point to specific parts of an XML document
- XPointer uses XPath expressions to navigate in the XML document
- XPointer is a W3C Recommendation

XPointer Browser Support

There is no browser support for XPointer. But XPointer is used in other XML languages.

XPointer Example

In this example, we will use XPointer in conjunction with XLink to point to a specific part of another document.

We will start by looking at the target XML document (the document we are linking to):

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

<dogbreeds>

<dog breed="Rottweiler" id="Rottweiler">
    <picture url="https://dog.com/rottweiler.gif" />
    <history>The Rottweiler's ancestors were probably Roman
    drover dogs.....</history>
    <temperament>Confident, bold, alert and imposing, the Rottweiler
    is a popular choice for its ability to protect....</temperament>
</dog>

<dog breed="FCRetriever" id="FCRetriever">
    <picture url="https://dog.com/fcretriever.gif" />
    <history>One of the earliest uses of retrieving dogs was to
    help fishermen retrieve fish from the water....</history>
    <temperament>The flat-coated retriever is a sweet, exuberant,
    lively dog that loves to play and retrieve....</temperament>
</dog>

</dogbreeds>
```

Note that the XML document above uses id attributes on each element!

So, instead of linking to the entire document (as with XLink), XPointer allows you to link to specific parts of the document. To link to a specific part of a page, add a number sign (#) and an XPointer expression after the URL in the xlink:href attribute, like this: `xlink:href="https://dog.com/dogbreeds.xml#xpointer(id('Rottweiler'))"`. The expression refers to the element in the target document, with the id value of "Rottweiler".

XPointer also allows a shorthand method for linking to an element with an id. You can use the value of the id directly, like this: `xlink:href="https://dog.com/dogbreeds.xml#Rottweiler"`.

The following XML document contains links to more information of the dog breed for each of my dogs:

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

<mydogs xmlns:xlink="http://www.w3.org/1999/xlink">

<mydog>
  <description>
    Anton is my favorite dog. He has won a lot of.....
  </description>
  <fact xlink:type="simple"
xlink:href="https://dog.com/dogbreeds.xml#Rottweiler">
    Fact about Rottweiler
  </fact>
</mydog>

<mydog>
  <description>
    Pluto is the sweetest dog on earth.....
  </description>
  <fact xlink:type="simple"
xlink:href="https://dog.com/dogbreeds.xml#FCRetriever">
    Fact about flat-coated Retriever
  </fact>
</mydog>

</mydogs>
```

XML Validator

Use our XML validator to syntax-check your XML.

Well Formed XML Documents

An XML document with correct syntax is called "Well Formed".

The syntax rules were described in the previous chapters:

- XML documents must have a root element
- XML elements must have a closing tag
- XML tags are case sensitive
- XML elements must be properly nested
- XML attribute values must be quoted

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

XML Errors Will Stop You

Errors in XML documents will stop your XML applications.

The W3C XML specification states that a program should stop processing an XML document if it finds an error. The reason is that XML software should be small, fast, and compatible.

HTML browsers are allowed to display HTML documents with errors (like missing end tags).

With XML, errors are not allowed.

Syntax-Check Your XML

To help you syntax-check your XML, we have created an XML validator.

Try to syntax-check correct XML : [Check XML](#)

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Try to syntax-check incorrect XML : [Check XML](#)

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</pheading>
<body>Don't forget me this weekend!</body>
</note>
```

Try to syntax-check your own XML : [Check XML](#)

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

Valid XML Documents

A "well formed" XML document is not the same as a "valid" XML document.

A "valid" XML document must be well formed. In addition, it must conform to a document type definition.

There are two different document type definitions that can be used with XML:

- DTD - The original Document Type Definition
- XML Schema - An XML-based alternative to DTD

A document type definition defines the rules and the legal elements and attributes for an XML document.

XML DTD

An XML document with correct syntax is called "Well Formed".

An XML document validated against a DTD is both "Well Formed" and "Valid".

Valid XML Documents

A "Valid" XML document is a "Well Formed" XML document, which also conforms to the rules of a DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE note SYSTEM "Note.dtd">
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

The DOCTYPE declaration, in the example above, is a reference to an external DTD file. The content of the file is shown in the paragraph below.

XML DTD

The purpose of a DTD is to define the structure of an XML document. It defines the structure with a list of legal elements:

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

The DTD above is interpreted like this:

- !DOCTYPE note defines that the root element of the document is note
- !ELEMENT note defines that the note element must contain the elements: "to, from, heading, body"
- !ELEMENT to defines the to element to be of type "#PCDATA"
- !ELEMENT from defines the from element to be of type "#PCDATA"
- !ELEMENT heading defines the heading element to be of type "#PCDATA"
- !ELEMENT body defines the body element to be of type "#PCDATA"

#PCDATA means parse-able text data.

Using DTD for Entity Declaration

A doctype declaration can also be used to define special characters and character strings, used in the document:

Example

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

<!DOCTYPE note [
<!ENTITY nbsp "&#xA0;">
<!ENTITY writer "Writer: Donald Duck.">
<!ENTITY copyright "Copyright: W3Schools.">
]>

<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
<footer>&writer;&nbsp;&copyright;</footer>
</note>
```

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

When to Use a DTD/Schema?

With a DTD, independent groups of people can agree to use a standard DTD for interchanging data.

With a DTD, you can verify that the data you receive from the outside world is valid.

You can also use a DTD to verify your own data.

If you want to study DTD, please read our [DTD Tutorial](#).

When NOT to Use a DTD/Schema?

XML does not require a DTD/Schema.

When you are experimenting with XML, or when you are working with small XML files, creating DTDs may be a waste of time.

If you develop applications, wait until the specification is stable before you add a document definition. Otherwise, your software might stop working because of validation errors.

XML Schema

An XML Schema describes the structure of an XML document, just like a DTD.

An XML document with correct syntax is called "Well Formed".

An XML document validated against an XML Schema is both "Well Formed" and "Valid".

XML Schema

XML Schema is an XML-based alternative to DTD:

```
<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>
```

The Schema above is interpreted like this:

- `<xs:element name="note">` defines the element called "note"
- `<xs:complexType>` the "note" element is a complex type
- `<xs:sequence>` the complex type is a sequence of elements
- `<xs:element name="to" type="xs:string">` the element "to" is of type string (text)
- `<xs:element name="from" type="xs:string">` the element "from" is of type string
- `<xs:element name="heading" type="xs:string">` the element "heading" is of type string
- `<xs:element name="body" type="xs:string">` the element "body" is of type string

XML Schemas are More Powerful than DTD

- XML Schemas are written in XML
- XML Schemas are extensible to additions
- XML Schemas support data types
- XML Schemas support namespaces

Why Use an XML Schema?

With XML Schema, your XML files can carry a description of its own format.

With XML Schema, independent groups of people can agree on a standard for interchanging data.

With XML Schema, you can verify data.

XML Schemas Support Data Types

One of the greatest strengths of XML Schemas is the support for data types:

- It is easier to describe document content
- It is easier to define restrictions on data
- It is easier to validate the correctness of data
- 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 Schemas with the XML DOM
- You can transform your Schemas with XSLT

If you want to study XML Schema, please read our [XML Schema Tutorial](#).

XML on the Server

XML files are plain text files just like HTML files.

XML can easily be stored and generated by a standard web server.

Storing XML Files on the Server

XML files can be stored on an Internet server exactly the same way as HTML files.

Start Windows Notepad and write the following lines:

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
  <from>Jani</from>
  <to>Tove</to>
  <message>Remember me this weekend</message>
</note>
```

Save the file on your web server with a proper name like "note.xml".

Generating XML with PHP

XML can be generated on a server without any installed XML software.

To generate an XML response from the server using PHP, use following code:

```
<?php
header("Content-type: text/xml");
echo "<?xml version='1.0' encoding='UTF-8'?>";
echo "<note>";
echo "<from>Jani</from>";
echo "<to>Tove</to>";
echo "<message>Remember me this weekend</message>";
echo "</note>";
?>
```

Note that the content type of the response header must be set to "text/xml".

[See how the PHP file will be returned from the server.](#)

If you want to study PHP, you will find our PHP tutorial on our [homepage](#).

Generating XML with ASP

To generate an XML response from the server - simply write the following code and save it as an ASP file on the web server:

```
<%  
response.ContentType="text/xml"  
response.Write("<?xml version='1.0' encoding='UTF-8'?>")  
response.Write("<note>")  
response.Write("<from>Jani</from>")  
response.Write("<to>Tove</to>")  
response.Write("<message>Remember me this weekend</message>")  
response.Write("</note>")  
%>
```

Note that the content type of the response must be set to "text/xml".

[See how the ASP file will be returned from the server.](#)

If you want to study ASP, you will find our ASP tutorial on our [homepage](#).

Generating XML From a Database

XML can be generated from a database without any installed XML software.

To generate an XML database response from the server, simply write the following code and save it as an ASP file on the web server:

```
<%  
response.ContentType = "text/xml"  
set conn=Server.CreateObject("ADODB.Connection")  
conn.provider="Microsoft.Jet.OLEDB.4.0;"  
conn.open server.mappath("/datafolder/database.mdb")  
  
sql="select fname, lname from tblGuestBook"  
set rs=Conn.Execute(sql)  
  
response.write("<?xml version='1.0' encoding='UTF-8'?>")  
response.write("<guestbook>")  
while (not rs.EOF)  
response.write("<guest>")  
response.write("<fname>" & rs("fname") & "</fname>")  
response.write("<lname>" & rs("lname") & "</lname>")  
response.write("</guest>")  
rs.MoveNext()  
wend  
  
rs.close()  
conn.close()  
response.write("</guestbook>")  
%>
```

See the real life database output from the ASP file above.

The example above uses ASP with ADO.

If you want to study ASP and ADO, you will find the tutorials on our [homepage](#).

Transforming XML with XSLT on the Server

This ASP transforms an XML file to XHTML on the server:

```
<%  
'Load XML  
set xml = Server.CreateObject("Microsoft.XMLDOM")  
xml.async = false  
xml.load(Server.MapPath("simple.xml"))  
  
'Load XSL  
set xsl = Server.CreateObject("Microsoft.XMLDOM")  
xsl.async = false  
xsl.load(Server.MapPath("simple.xsl"))  
  
'Transform file  
Response.Write(xml.transformNode(xsl))  
%>
```

Example explained

- The first block of code creates an instance of the Microsoft XML parser (XMLDOM), and loads the XML file into memory.
- The second block of code creates another instance of the parser and loads the XSL file into memory.
- The last line of code transforms the XML document using the XSL document, and sends the result as XHTML to your browser. Nice!

[See how it works.](#)

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

Try it Yourself Examples in Every Chapter

In every chapter, you can edit the examples online, and click on a button to view the result.

AJAX Example

Let AJAX change this text

[Change Content](#)

AJAX Example Explained

HTML Page

```
<!DOCTYPE html>
<html>
<body>

<div id="demo">
  <h2>Let AJAX change this text</h2>
  <button type="button" onclick="loadDoc()">Change Content</button>
</div>

</body>
</html>
```

The HTML page contains a `<div>` section and a `<button>`.

The `<div>` section is used to display information from a server.

The `<button>` calls a function (if it is clicked).

The function requests data from a web server and displays it:

Function loadDoc()

```
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();  
}
```

What is AJAX?

AJAX = **A**synchronous **J**avaScript **A**nd **X**ML.

AJAX is not a programming language.

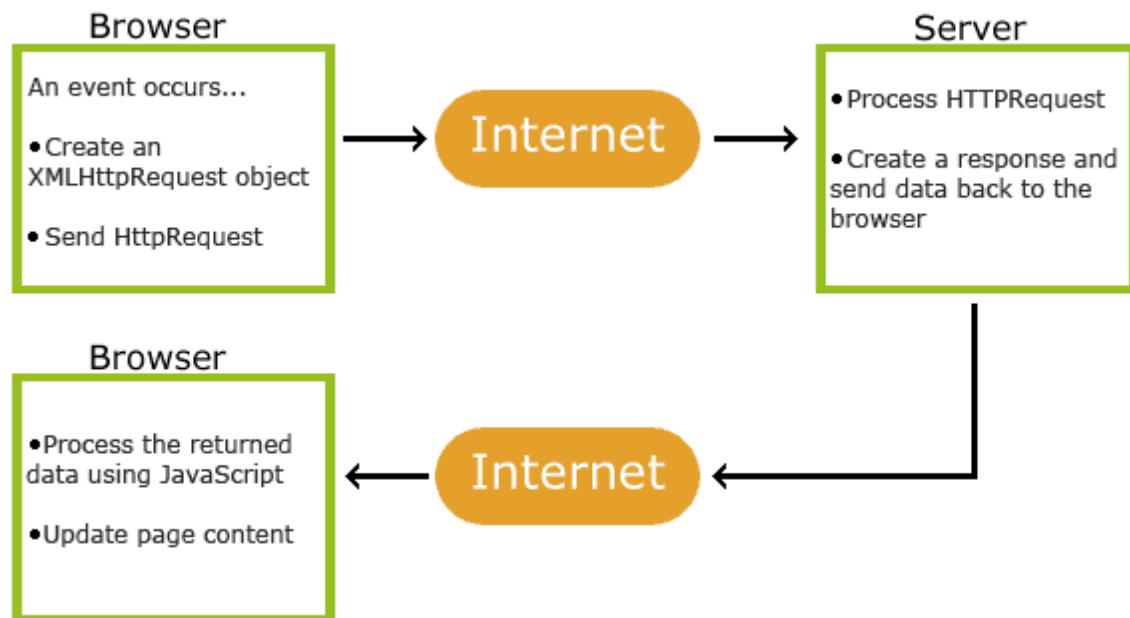
AJAX just uses a combination of:

- A browser built-in XMLHttpRequest object (to request data from a web server)
- JavaScript and HTML DOM (to display or use the data)

AJAX is a misleading name. AJAX applications might use XML to transport data, but it is equally common to transport data as plain text or JSON text.

AJAX allows web pages to be updated asynchronously by exchanging data with a web server behind the scenes. This means that it is possible to update parts of a web page, without reloading the whole page.

How AJAX Works



1. An event occurs in a web page (the page is loaded, a button is clicked)
2. An XMLHttpRequest object is created by JavaScript
3. The XMLHttpRequest object sends a request to a web server
4. The server processes the request
5. The server sends a response back to the web page
6. The response is read by JavaScript
7. Proper action (like page update) is performed by JavaScript

AJAX - The XMLHttpRequest Object

The keystone of AJAX is the XMLHttpRequest object.

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.

Create an XMLHttpRequest Object

All modern browsers (Chrome, Firefox, IE7+, Edge, Safari Opera) have a built-in XMLHttpRequest object.

Syntax for creating an XMLHttpRequest object:

```
variable = new XMLHttpRequest();
```

Example

```
var xhttp = new XMLHttpRequest();
```

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.

Old Versions of Internet Explorer (IE5 and IE6)

Old versions of Internet Explorer (IE5 and IE6) use an ActiveX object instead of the XMLHttpRequest object:

```
variable = new ActiveXObject("Microsoft.XMLHTTP");
```

To handle IE5 and IE6, check if the browser supports the XMLHttpRequest object, or else create an ActiveX object:

Example

```
if (window.XMLHttpRequest) {  
    // code for modern browsers  
    xmlhttp = new XMLHttpRequest();  
} else {  
    // code for old IE browsers  
    xmlhttp = new ActiveXObject("Microsoft.XMLHTTP");  
}
```

XMLHttpRequest Object Methods

Method	Description
new XMLHttpRequest()	Creates a new XMLHttpRequest object
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(<i>method, url, async, user, psw</i>)	Specifies the request <i>method</i> : the request type GET or POST <i>url</i> : the file location <i>async</i> : true (asynchronous) or false (synchronous) <i>user</i> : optional user name <i>psw</i> : optional password
send()	Sends the request to the server.Used for GET requests
send(<i>string</i>)	Sends the request to the server.Used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

XMLHttpRequest Object Properties

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number of a request 200: "OK" 403: "Forbidden" 404: "Not Found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

AJAX - Send a Request To a Server

The XMLHttpRequest object is used to exchange data with a server.

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();
```

Method	Description
<code>open(<i>method</i>, <i>url</i>, <i>async</i>)</code>	Specifies the type of request <i>method</i> : the type of request: GET or POST <i>url</i> : the server (file) location <i>async</i> : true (asynchronous) or false (synchronous)
<code>send()</code>	Sends the request to the server (used for GET)
<code>send(<i>string</i>)</code>	Sends the request to the server (used for POST)

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

A simple GET request:

Example

```
xhttp.open("GET", "demo_get.asp", true);
xhttp.send();
```

Example

```
xhttp.open("GET", "demo_get.asp?t=" + Math.random(), true);  
xhttp.send();
```

If you want to send information with the GET method, add the information to the URL:

Example

```
xhttp.open("GET", "demo_get2.asp?fname=Henry&lname=Ford", true);  
xhttp.send();
```

POST Requests

A simple POST request:

Example

```
xhttp.open("POST", "demo_post.asp", true);  
xhttp.send();
```

To POST data like an HTML form, add an HTTP header with setRequestHeader(). Specify the data you want to send in the send() method:

Example

```
xhttp.open("POST", "demo_post2.asp", true);  
xhttp.setRequestHeader("Content-type", "application/x-www-form-  
urlencoded");  
xhttp.send("fname=Henry&lname=Ford");
```

Method	Description
setRequestHeader(header, value)	Adds HTTP headers to the request <i>header</i> : specifies the header name <i>value</i> : specifies the header value

The url - A File On a Server

The url parameter of the open() method, is an address to a file on a server:

```
xhttp.open("GET", "ajax_test.asp", true);
```

The file can be any kind of file, like .txt and .xml, or server scripting files like .asp and .php (which can perform actions on the server before sending the response back).

Asynchronous - True or False?

Server requests should be sent asynchronously.

The async parameter of the open() method should be set to true:

```
xhttp.open("GET", "ajax_test.asp", true);
```

By sending asynchronously, the JavaScript does not have to wait for the server response, but can instead:

- execute other scripts while waiting for server response
- deal with the response after the response is ready

The onreadystatechange Property

With the XMLHttpRequest object you can define a function to be executed when the request receives an answer.

The function is defined in the **onreadystatechange** property of the XMLHttpRequest object:

Example

```
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();
```

You will learn more about onreadystatechange in a later chapter.

Synchronous Request

To execute a synchronous request, change the third parameter in the open() method to false:

```
xhttp.open("GET", "ajax_info.txt", false);
```

Sometimes `async = false` are used for quick testing. You will also find synchronous requests in older JavaScript code.

Since the code will wait for server completion, there is no need for an `onreadystatechange` function:

Example

```
xhttp.open("GET", "ajax_info.txt", false);
xhttp.send();
document.getElementById("demo").innerHTML = xhttp.responseText;
```

Synchronous XMLHttpRequest (`async = false`) is not recommended because the JavaScript will stop executing until the server response is ready. If the server is busy or slow, the application will hang or stop.

Synchronous XMLHttpRequest is in the process of being removed from the web standard, but this process can take many years.

Modern developer tools are encouraged to warn about using synchronous requests and may throw an `InvalidAccessError` exception when it occurs.

AJAX - 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.

Property	Description
onreadystatechange	Defines a function to be called when the readyState property changes
readyState	Holds the status of the XMLHttpRequest. 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
status	200: "OK" 403: "Forbidden" 404: "Page not found" For a complete list go to the Http Messages Reference
statusText	Returns the status-text (e.g. "OK" or "Not Found")

The onreadystatechange function is called every time the readyState changes.

When readyState is 4 and status is 200, the response is ready:

Example

```
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 onreadystatechange event is triggered four times (1-4), one time for each change in the readyState.

Using a Callback Function

A callback function is a function passed as a parameter to another function.

If you have more than one AJAX task in a website, you should create one function for executing the XMLHttpRequest object, and one callback function for each AJAX task.

The function call should contain the URL and what function to call when the response is ready.

Example

```
loadDoc("url-1", myFunction1);

loadDoc("url-2", myFunction2);

function loadDoc(url, cFunction) {
  var xhttp;
  xhttp=new XMLHttpRequest();
  xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      cFunction(this);
    }
  };
  xhttp.open("GET", url, true);
  xhttp.send();
}

function myFunction1(xhttp) {
  // action goes here
}
function myFunction2(xhttp) {
  // action goes here
}
```

Server Response Properties

Property	Description
responseText	get the response data as a string
responseXML	get the response data as XML data

Server Response Methods

Method	Description
getResponseHeader()	Returns specific header information from the server resource
getAllResponseHeaders()	Returns all the header information from the server resource

The responseText Property

The **responseText** property returns the server response as a JavaScript string, and you can use it accordingly:

Example

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

The responseXML Property

The XML XMLHttpRequest object has an in-built XML parser.

The **responseXML** property returns the server response as an XML DOM object.

Using this property you can parse the response as an XML DOM object:

Example

Request the file [cd_catalog.xml](#) and parse the response:

```
xmlDoc = xhttp.responseXML;
txt = "";
x = xmlDoc.getElementsByTagName("ARTIST");
for (i = 0; i < x.length; i++) {
    txt += x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
xhttp.open("GET", "cd_catalog.xml", true);
xhttp.send();
```

You will learn a lot more about XML DOM in the DOM chapters of this tutorial.

The getAllResponseHeaders() Method

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

Example

```
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.

Example

```
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();
```

AJAX XML Example

AJAX can be used for interactive communication with an XML file.

AJAX XML Example

The following example will demonstrate how a web page can fetch information from an XML file with AJAX:

Example

[Get CD info](#)

Example Explained

When a user clicks on the "Get CD info" button above, the loadDoc() function is executed. The loadDoc() function creates an XMLHttpRequest object, adds the function to be executed when the server response is ready, and sends the request off to the server. When the server response is ready, an HTML table is built, nodes (elements) are extracted from the XML file, and it finally updates the element "demo" with the HTML table filled with XML data:

LoadXMLDoc()

```
function loadDoc() {
    var xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            myFunction(this);
        }
    };
    xhttp.open("GET", "cd_catalog.xml", true);
    xhttp.send();
}

function myFunction(xml) {
    var i;
    var xmlDoc = xml.responseXML;
    var table = "<tr><th>Title</th><th>Artist</th></tr>";
    var x = xmlDoc.getElementsByTagName("CD");
    for (i = 0; i < x.length; i++) {
        table += "<tr><td>" +
        x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +
        "</td><td>" +
        x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +
        "</td></tr>";
    }
    document.getElementById("demo").innerHTML = table;
}
```

The XML File

The XML file used in the example above looks like this: "[cd_catalog.xml](#)".

AJAX PHP Example

AJAX is used to create more interactive applications.

AJAX PHP Example

The following example demonstrates how a web page can communicate with a web server while a user types characters in an input field:

Example

Start typing a name in the input field below:

First name: Suggestions:

Example Explained

In the example above, when a user types a character in the input field, a function called "showHint()" is executed.

The function is triggered by the onkeyup event.

Here is the HTML code:

Example

```
<html>
<head>
<script>
function showHint(str) {
    if (str.length == 0) {
        document.getElementById("txtHint").innerHTML = "";
        return;
    } else {
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                document.getElementById("txtHint").innerHTML =
this.responseText;
            }
        };
        xmlhttp.open("GET", "gethint.php?q=" + str, true);
        xmlhttp.send();
    }
}
</script>
</head>
<body>

<p><b>Start typing a name in the input field below:</b></p>
<form>
First name: <input type="text" onkeyup="showHint(this.value)">
</form>
<p>Suggestions: <span id="txtHint"></span></p>
</body>
</html>
```

Code explanation:

First, check if the input field is empty (str.length == 0). If it is, clear the content of the txtHint placeholder and exit the function.

However, if the input field is not empty, do the following:

- Create an XMLHttpRequest object
- Create the function to be executed when the server response is ready
- Send the request off to a PHP file (gethint.php) on the server
- Notice that q parameter is added gethint.php?q="+str
- The str variable holds the content of the input field

The PHP File - "gethint.php"

The PHP file checks an array of names, and returns the corresponding name(s) to the browser:

```
<?php  
// Array with names  
$a[] = "Anna";  
$a[] = "Brittany";  
$a[] = "Cinderella";  
$a[] = "Diana";  
$a[] = "Eva";  
$a[] = "Fiona";  
$a[] = "Gunda";  
$a[] = "Hege";  
$a[] = "Inga";  
$a[] = "Johanna";  
$a[] = "Kitty";  
$a[] = "Linda";  
$a[] = "Nina";  
$a[] = "Ophelia";  
$a[] = "Petunia";  
$a[] = "Amanda";  
$a[] = "Raquel";  
$a[] = "Cindy";  
$a[] = "Doris";  
$a[] = "Eve";  
$a[] = "Evita";  
$a[] = "Sunniva";  
$a[] = "Tove";  
$a[] = "Unni";  
$a[] = "Violet";  
$a[] = "Liza";  
$a[] = "Elizabeth";
```

```

$a[] = "Ellen";
$a[] = "Wenche";
$a[] = "Vicky";

// get the q parameter from URL
$q = $_REQUEST["q"];

$hint = "";

// lookup all hints from array if $q is different from ""
if ($q !== "") {
    $q = strtolower($q);
    $len=strlen($q);
    foreach($a as $name) {
        if (stristr($q, substr($name, 0, $len))) {
            if ($hint === "") {
                $hint = $name;
            } else {
                $hint .= ", $name";
            }
        }
    }
}

// Output "no suggestion" if no hint was found or output correct values
echo $hint === "" ? "no suggestion" : $hint;
?>

```

AJAX ASP Example

AJAX is used to create more interactive applications.

AJAX ASP Example

The following example will demonstrate how a web page can communicate with a web server while a user type characters in an input field:

Example

Start typing a name in the input field below:

First name: Suggestions:

Example Explained

In the example above, when a user types a character in the input field, a function called "showHint()" is executed.

The function is triggered by the onkeyup event.

Here is the HTML code:

Example

```
<html>
<head>
<script>
function showHint(str) {
    if (str.length == 0) {
        document.getElementById("txtHint").innerHTML = "";
        return;
    } else {
        var xmlhttp = new XMLHttpRequest();
        xmlhttp.onreadystatechange = function() {
            if (this.readyState == 4 && this.status == 200) {
                document.getElementById("txtHint").innerHTML =
this.responseText;
            }
        };
        xmlhttp.open("GET", "gethint.asp?q=" + str, true);
        xmlhttp.send();
    }
}
</script>
</head>
<body>

<p><b>Start typing a name in the input field below:</b></p>
<form>
First name: <input type="text" onkeyup="showHint(this.value)">
</form>
<p>Suggestions: <span id="txtHint"></span></p>
</body>
</html>
```

Code explanation:

First, check if the input field is empty (str.length == 0). If it is, clear the content of the txtHint placeholder and exit the function.

However, if the input field is not empty, do the following:

- Create an XMLHttpRequest object
- Create the function to be executed when the server response is ready
- Send the request off to an ASP file (gethint.asp) on the server
- Notice that q parameter is added gethint.asp?q="+str
- The str variable holds the content of the input field

The ASP File - "gethint.asp"

The ASP file checks an array of names, and returns the corresponding name(s) to the browser:

```
<%
response.expires=-1
dim a(30)
'Fill up array with names
a(1)="Anna"
a(2)="Brittany"
a(3)="Cinderella"
a(4)="Diana"
a(5)="Eva"
a(6)="Fiona"
a(7)="Gunda"
a(8)="Hege"
a(9)="Inga"
a(10)="Johanna"
a(11)="Kitty"
a(12)="Linda"
a(13)="Nina"
a(14)="Ophelia"
a(15)="Petunia"
a(16)="Amanda"
a(17)="Raquel"
a(18)="Cindy"
a(19)="Doris"
a(20)="Eve"
a(21)="Evita"
a(22)="Sunniva"
a(23)="Tove"
a(24)="Unni"
a(25)="Violet"
```

```

a(26)="Liza"
a(27)="Elizabeth"
a(28)="Ellen"
a(29)="Wenche"
a(30)="Vicky"

'get the q parameter from URL
q=ucase(request.querystring("q"))

'lookup all hints from array if length of q>0
if len(q)>0 then
    hint=""
    for i=1 to 30
        if q=ucase(mid(a(i),1,len(q))) then
            if hint="" then
                hint=a(i)
            else
                hint=hint & " , " & a(i)
            end if
        end if
    next
end if

'Output "no suggestion" if no hint were found
'or output the correct values
if hint="" then
    response.write("no suggestion")
else
    response.write(hint)
end if
%>

```

AJAX Database Example

AJAX can be used for interactive communication with a database.

AJAX Database Example

The following example will demonstrate how a web page can fetch information from a database with AJAX:

Example

Select a customer: ▾

Customer info will be listed here...

Example Explained - The showCustomer() Function

When a user selects a customer in the dropdown list above, a function called "showCustomer()" is executed. The function is triggered by the "onchange" event:

showCustomer

```
function showCustomer(str) {
    var xhttp;
    if (str == "") {
        document.getElementById("txtHint").innerHTML = "";
        return;
    }
    xhttp = new XMLHttpRequest();
    xhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            document.getElementById("txtHint").innerHTML = this.responseText;
        }
    };
    xhttp.open("GET", "getcustomer.asp?q="+str, true);
    xhttp.send();
}
```

The showCustomer() function does the following:

- Check if a customer is selected
- Create an XMLHttpRequest object
- Create the function to be executed when the server response is ready
- Send the request off to a file on the server
- Notice that a parameter (q) is added to the URL (with the content of the dropdown list)

The AJAX Server Page

The page on the server called by the JavaScript above is an ASP file called "getcustomer.asp". The server file could easily be rewritten in PHP, or some other server languages.

[Look at a corresponding example in PHP.](#)

The source code in "getcustomer.asp" runs a query against a database, and returns the result in an HTML table:

```
<%
response.expires=-1
sql="SELECT * FROM CUSTOMERS WHERE CUSTOMERID="
sql=sql & "" & request.querystring("q") & ""

set conn=Server.CreateObject("ADODB.Connection")
conn.Provider="Microsoft.Jet.OLEDB.4.0"
conn.Open(Server.MapPath("/datafolder/northwind.mdb"))
set rs=Server.CreateObject("ADODB.recordset")
rs.Open sql,conn

response.write("<table>")
do until rs.EOF
  for each x in rs.Fields
    response.write("<tr><td><b>" & x.name & "</b></td>")
    response.write("<td>" & x.value & "</td></tr>")
  next
  rs.MoveNext
loop
response.write("</table>")
%>
```

XML Applications

This chapter demonstrates some HTML applications using XML, HTTP, DOM, and JavaScript.

The XML Document Used

In this chapter we will use the XML file called ["cd_catalog.xml"](#).

Display XML Data in an HTML Table

This example loops through each <CD> element, and displays the values of the <ARTIST> and the <TITLE> elements in an HTML table:

Example

```
<html>
<head>
<style>
table, th, td {
  border: 1px solid black;
  border-collapse: collapse;
}
th, td {
  padding: 5px;
}
</style>
</head>
<body>

<table id="demo"></table>

<script>
function loadXMLDoc() {
  var xmlhttp = new XMLHttpRequest();
  xmlhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
      myFunction(this);
    }
  };
  xmlhttp.open("GET", "cd_catalog.xml", true);
  xmlhttp.send();
}

myFunction = function(xml) {
  var table, tr, td, cd, artist, title, i;
  table = document.createElement("table");
  tr = document.createElement("tr");
  td = document.createElement("td");
  cd = xml.documentElement.getElementsByTagName("CD");
  for (i = 0; i < cd.length; i++) {
    artist = cd[i].getElementsByTagName("ARTIST");
    title = cd[i].getElementsByTagName("TITLE");
    td.innerHTML = artist[0].childNodes[0].nodeValue + " - " + title[0].childNodes[0].nodeValue;
    tr.appendChild(td);
  }
  table.appendChild(tr);
  document.getElementById("demo").appendChild(table);
}

loadXMLDoc();
```

```
}

function myFunction(xml) {
    var i;
    var xmlDoc = xml.responseXML;
    var table=<tr><th>Artist</th><th>Title</th></tr>;
    var x = xmlDoc.getElementsByTagName("CD");
    for (i = 0; i <x.length; i++) {
        table += "<tr><td>" +
        x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +
        "</td><td>" +
        x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +
        "</td></tr>";
    }
    document.getElementById("demo").innerHTML = table;
}
</script>

</body>
</html>
```

For more information about using JavaScript and the XML DOM, go to [DOM Intro](#).

Display the First CD in an HTML div Element

This example uses a function to display the first CD element in an HTML element with id="showCD":

Example

```
displayCD(0);

function displayCD(i) {
    var xmlhttp = new XMLHttpRequest();
    xmlhttp.onreadystatechange = function() {
        if (this.readyState == 4 && this.status == 200) {
            myFunction(this, i);
        }
    };
    xmlhttp.open("GET", "cd_catalog.xml", true);
    xmlhttp.send();
}

function myFunction(xml, i) {
    var xmlDoc = xml.responseXML;
    x = xmlDoc.getElementsByTagName("CD");
    document.getElementById("showCD").innerHTML =
    "Artist: " +
    x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +
    "<br>Title: " +
    x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +
    "<br>Year: " +
    x[i].getElementsByTagName("YEAR")[0].childNodes[0].nodeValue;
}
```

Navigate Between the CDs

To navigate between the CDs, in the example above, add a next() and previous() function:

Example

```
function next() {  
    // display the next CD, unless you are on the last CD  
    if (i < x.length-1) {  
        i++;  
        displayCD(i);  
    }  
}  
  
function previous() {  
    // display the previous CD, unless you are on the first CD  
    if (i > 0) {  
        i--;  
        displayCD(i);  
    }  
}
```

Show Album Information When Clicking On a CD

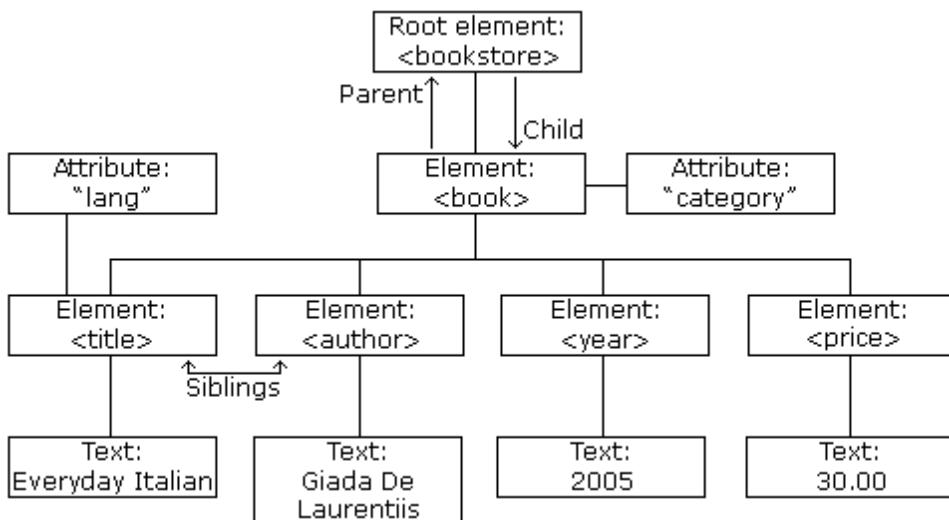
The last example shows how you can display album information when the user clicks on a CD:

Example

```
function displayCD(i) {  
    document.getElementById("showCD").innerHTML =  
        "Artist: " +  
        x[i].getElementsByTagName("ARTIST")[0].childNodes[0].nodeValue +  
        "<br>Title: " +  
        x[i].getElementsByTagName("TITLE")[0].childNodes[0].nodeValue +  
        "<br>Year: " +  
        x[i].getElementsByTagName("YEAR")[0].childNodes[0].nodeValue;  
}
```

XML DOM Tutorial

XML DOM



What is the DOM?

The DOM defines a standard for accessing and manipulating documents:

"The W3C Document Object Model (DOM) is a platform and language-neutral interface that allows programs and scripts to dynamically access and update the content, structure, and style of a document."

The HTML DOM defines a standard way for accessing and manipulating HTML documents. It presents an HTML document as a tree-structure.

The XML DOM defines a standard way for accessing and manipulating XML documents. It presents an XML document as a tree-structure.

Understanding the DOM is a must for anyone working with HTML or XML.

The HTML DOM

All HTML elements can be accessed through the HTML DOM.

This example changes the value of an HTML element with id="demo":

Example

```
<h1 id="demo">This is a Heading</h1>

<script>
```

```
document.getElementById("demo").innerHTML = "Hello World!";
</script>
```

This example changes the value of the first <h1> element in an HTML document:

Example

```
<h1>This is a Heading</h1>

<h1>This is a Heading</h1>

<script>
document.getElementsByTagName("h1")[0].innerHTML = "Hello World!";
</script>
```

Note: Even if the HTML document contains only ONE <h1> element you still have to specify the array index [0], because the getElementsByTagName() method always returns an array. You can learn a lot more about the HTML DOM in our [JavaScript tutorial](#).

The 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:

Example

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

Loading an XML File

The XML file used in the examples below is `books.xml`.

This example reads "books.xml" into `xmlDoc` and retrieves the text value of the first `<title>` element in `books.xml`:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myFunction(this);
    }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
    var xmlDoc = xml.responseXML;
    document.getElementById("demo").innerHTML =
        xmlDoc.getElementsByTagName("title")[0].childNodes[0].nodeValue;
}
</script>

</body>
</html>
```

Example Explained

- **`xmlDoc`** - the XML DOM object created by the parser.
- **`getElementsByTagName("title")[0]`** - get the first `<title>` element
- **`childNodes[0]`** - the first child of the `<title>` element (the text node)
- **`nodeValue`** - the value of the node (the text itself)

Loading an XML String

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

Example

```
<html>
<body>

<p id="demo"></p>

<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>

</body>
</html>
```

Programming Interface

The DOM models XML as a set of node objects. The nodes can be accessed with JavaScript or other programming languages. In this tutorial we use JavaScript.

The programming interface to the DOM is defined by a set standard properties and methods.

Properties are often referred to as something that is (i.e. nodename is "book").

Methods are often referred to as something that is done (i.e. delete "book").

XML DOM Properties

These are some typical DOM properties:

- `x.nodeName` - the name of x
- `x.nodeValue` - the value of x
- `x.parentNode` - the parent node of x
- `x.childNodes` - the child nodes of x
- `x.attributes` - the attributes nodes of x

Note: In the list above, x is a node object.

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

Note: In the list above, x is a node object.

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

DOM Example

Look at the following XML file (`books.xml`):

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">XQuery Kick Start</title>
    <author>James McGovern</author>
    <author>Per Bothner</author>
    <author>Kurt Cagle</author>
    <author>James Linn</author>
    <author>Vaidyanathan Nagarajan</author>
    <year>2003</year>
    <price>49.99</price>
  </book>
  <book category="web" cover="paperback">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

The root node in the XML above is named `<bookstore>`.

All other nodes in the document are contained within `<bookstore>`.

The root node `<bookstore>` holds 4 `<book>` nodes.

The first `<book>` node holds the child nodes: `<title>`, `<author>`, `<year>`, and `<price>`.

The child nodes contain one text node each, "Everyday Italian", "Giada De Laurentiis", "2005", and "30.00".

Text is Always Stored in Text Nodes

A common error in DOM processing is to expect an element node to contain text.

However, the text of an element node is stored in a text node.

In this example: <year>2005</year>, the element node <year> holds a text node with the value "2005".

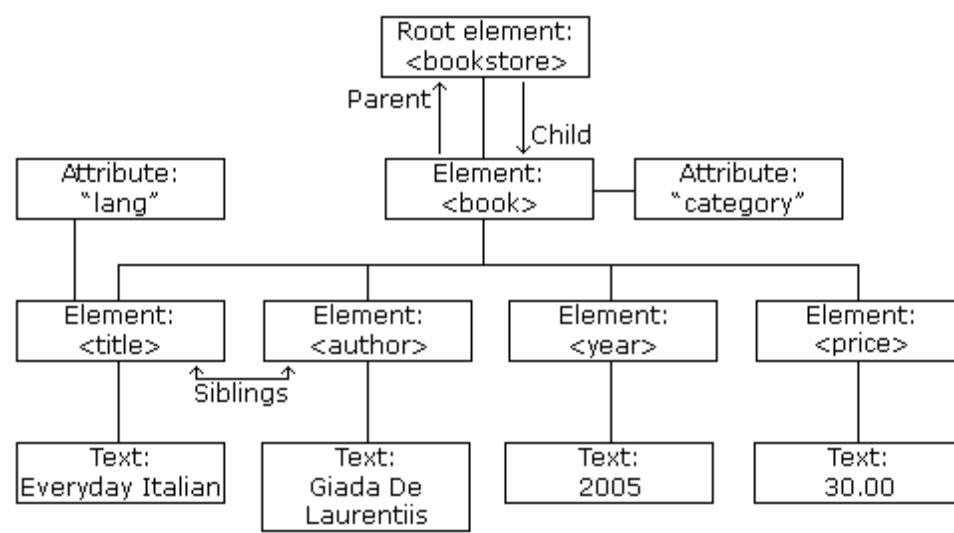
"2005" is **not** the value of the <year> element!

The XML DOM Node Tree

The XML DOM views an XML document as a tree-structure. The tree structure is called a **node-tree**.

All nodes can be accessed through the tree. Their contents can be modified or deleted, and new elements can be created.

The node tree shows the set of nodes, and the connections between them. The tree starts at the root node and branches out to the text nodes at the lowest level of the tree:



The image above represents the XML file `books.xml`.

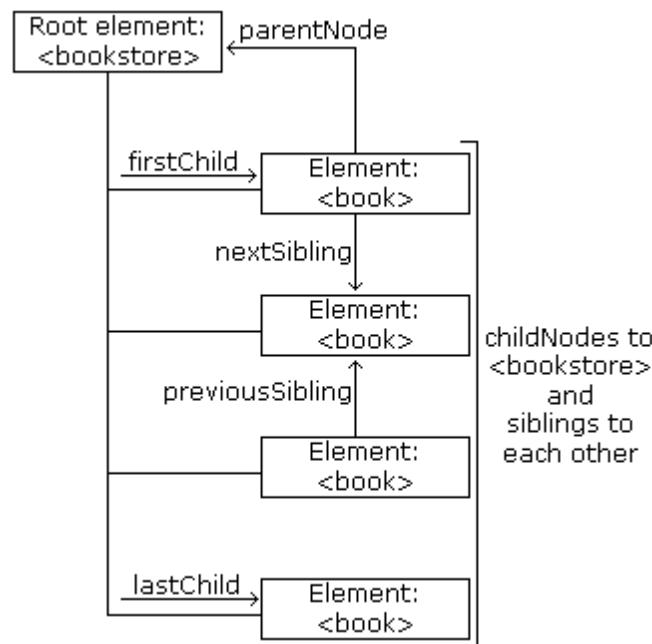
Node Parents, Children, and Siblings

The nodes in the node tree have a hierarchical relationship to each other.

The terms parent, child, and sibling are used to describe the relationships. Parent nodes have children. Children on the same level are called siblings (brothers or sisters).

- In a node tree, the top node is called the root
- Every node, except the root, has exactly one parent node
- A node can have any number of children
- A leaf is a node with no children
- Siblings are nodes with the same parent

The following image illustrates a part of the node tree and the relationship between the nodes:



Because the XML data is structured in a tree form, it can be traversed without knowing the exact structure of the tree and without knowing the type of data contained within.

You will learn more about traversing the node tree in a later chapter of this tutorial.

First Child - Last Child

Look at the following XML fragment:

```
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
</bookstore>
```

In the XML above, the `<title>` element is the first child of the `<book>` element, and the `<price>` element is the last child of the `<book>` element.

Furthermore, the `<book>` element is the parent node of the `<title>`, `<author>`, `<year>`, and `<price>` elements.

XML DOM - Accessing Nodes

With the DOM, you can access every node in an XML document.

Try it Yourself - Examples

The examples below use the XML file `books.xml`.

Access a node using its index number in a node list Use the `getElementsByTagName()` method to get the third `<title>` element in "books.xml"

Loop through nodes using the length property Use the `length` property to loop through all `<title>` elements in "books.xml"

See the node type of an element Use the `nodeType` property to get node type of the root element in "books.xml".

Loop through element nodes Use the `nodeType` property to only process element nodes in "books.xml".

Loop through element nodes using node relationships Use the `nodeType` property and the `nextSibling` property to process element nodes in "books.xml".

Accessing Nodes

You can access a node in three ways:

1. By using the `getElementsByTagName()` method
2. By looping through (traversing) the nodes tree.
3. By navigating the node tree, using the node relationships.

The `getElementsByTagName()` Method

`getElementsByTagName()` returns all elements with a specified tag name.

Syntax

```
node.getElementsByTagName( "tagname" );
```

Example

The following example returns all `<title>` elements under the `x` element:

```
x.getElementsByTagName( "title" );
```

Note that the example above only returns <title> elements under the x node. To return all <title> elements in the XML document use:

```
xmlDoc.getElementsByTagName("title");
```

where xmlDoc is the document itself (document node).

DOM Node List

The getElementsByTagName() method returns a node list. A node list is an array of nodes.

```
x = xmlDoc.getElementsByTagName("title");
```

The <title> elements in x can be accessed by index number. To access the third <title> you can write::

```
y = x[2];
```

Note: The index starts at 0.

You will learn more about node lists in a later chapter of this tutorial.

DOM Node List Length

The length property defines the length of a node list (the number of nodes).

You can loop through a node list by using the length property:

Example

```
var x = xmlDoc.getElementsByTagName("title");

for (i = 0; i <x.length; i++) {
    // do something for each node
}
```

Node Types

The **documentElement** property of the XML document is the root node.

The **nodeName** property of a node is the name of the node.

The **nodeType** property of a node is the type of the node.

You will learn more about the node properties in the next chapter of this tutorial.

Try it Yourself

Traversing Nodes

The following code loops through the child nodes, that are also element nodes, of the root node:

Example

```
txt = "";
x = xmlDoc.documentElement.childNodes;

for (i = 0; i <x.length; i++) {
    // Process only element nodes (type 1)
    if (x[i].nodeType == 1) {
        txt += x[i].nodeName + "<br>";
    }
}
```

Example explained:

1. Suppose you have loaded "[books.xml](#)" into xmlDoc
2. Get the child nodes of the root element (xmlDoc)
3. For each child node, check the node type. If the node type is "1" it is an element node
4. Output the name of the node if it is an element node

Navigating Node Relationships

The following code navigates the node tree using the node relationships:

Example

```
x = xmlDoc.getElementsByTagName("book")[0];
xlen = x.childNodes.length;
y = x.firstChild;

txt = "";
for (i = 0; i <xlen; i++) {
    // Process only element nodes (type 1)
    if (y.nodeType == 1) {
        txt += y.nodeName + "<br>";
    }
    y = y.nextSibling;
}
```

Example explained:

1. Suppose you have loaded "[books.xml](#)" into xmlDoc
2. Get the child nodes of the first book element
3. Set the "y" variable to be the first child node of the first book element
4. For each child node (starting with the first child node "y"):
5. Check the node type. If the node type is "1" it is an element node
6. Output the name of the node if it is an element node
7. Set the "y" variable to be the next sibling node, and run through the loop again

XML DOM Node Information

The `nodeName`, `nodeValue`, and `nodeType` properties contain information about nodes.

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Get the node name of an element node](#) This example uses the `nodeName` property to get the node name of the root element in "books.xml".

[Get the text from a text node](#) This example uses the `nodeValue` property to get the text of the first `<title>` element in "books.xml".

[Change the text in a text node](#) This example uses the `nodeValue` property to change the text of the first `<title>` element in "books.xml".

[Get the node name and type of an element node](#) This example uses the `nodeName` and `nodeType` property to get node name and type of the root element in "books.xml".

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`

[Try it Yourself.](#)

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:

Example

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

Result: txt = "Everyday Italian"

Example explained:

1. Suppose you have loaded "[books.xml](#)" into xmlDoc
2. Get text node of the first <title> element node
3. Set the txt variable to be the value of the text node

Change the Value of an Element

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

Example

```
var x = xmlDoc.getElementsByTagName("title")[0].childNodes[0];
x.nodeValue = "Easy Cooking";
```

Example explained:

1. Suppose you have loaded "[books.xml](#)" into xmlDoc
2. Get text node of the first <title> element node
3. Change the value of the text node to "Easy Cooking"

The nodeType Property

The `nodeType` property specifies the type of node.

`nodeType` is read only.

The most important node types are:

Node type	NodeType
Element	1
Attribute	2
Text	3
Comment	8
Document	9

[Try it Yourself.](#)

XML DOM Node List

A list of nodes is returned by the `getElementsByName()` method and the `childNodes` property.

Try it Yourself - Examples

The examples below use the XML file `books.xml`.

Get the text from the first <title> element This example uses the `getElementsByName()` method to get the text from the first `<title>` element in "books.xml".

Loop through nodes using the length property This example uses node list and the `length` property to loop through all `<title>` elements in "books.xml"

Get the attribute of an element This example uses a attribute list to get attribute from the first `<book>` element in "books.xml".

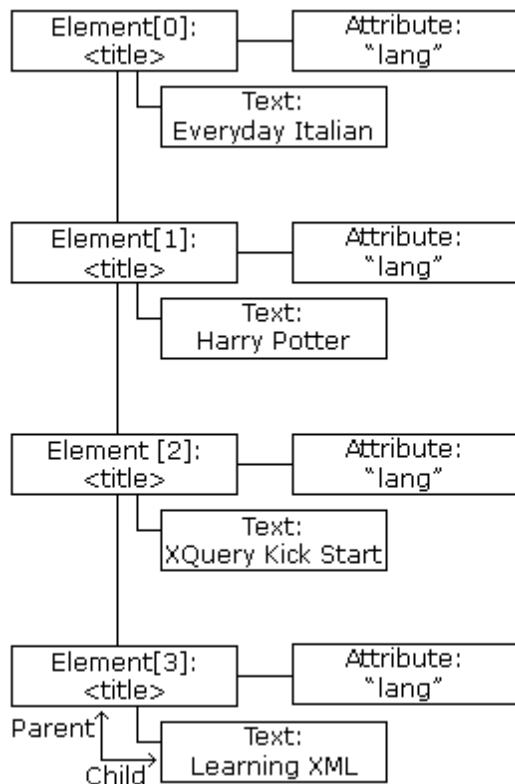
DOM Node List

When using properties or methods like `childNodes` or `getElementsByName()`, a node list object is returned.

A node list object represents a list of nodes, in the same order as in the XML.

Nodes in the node list are accessed with index numbers starting from 0.

The following image represents a node list of the `<title>` elements in "[books.xml](#)":



Suppose "books.xml" is loaded into the variable `xmlDoc`.

This code fragment returns a node list of title elements in "books.xml":

```
x = xmlDoc.getElementsByTagName("title");
```

After the execution of the statement above, `x` is a node list object.

The following code fragment returns the text from the first `<title>` element in the node list (`x`):

Example

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

After the execution of the statement above, `txt = "Everyday Italian"`.

Node List Length

A node list object keeps itself up-to-date. If an element is deleted or added, the list is automatically updated.

The length property of a node list is the number of nodes in the list.

This code fragment returns the number of <title> elements in "books.xml":

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

After the execution of the statement above, the value of x will be 4.

The length of the node list can be used to loop through all the elements in the list.

This code fragment uses the length property to loop through the list of <title> elements:

Example

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

for (i = 0; i <xLen; i++) {
    txt += x[i].childNodes[0].nodeValue + " ";
}
```

Output:

```
Everyday Italian
Harry Potter
XQuery Kick Start
Learning XML
```

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Set the x variable to hold a node list of all title elements
3. Collect the text node values from <title> elements

DOM Attribute List (Named Node Map)

The attributes property of an element node returns a list of attribute nodes.

This is called a named node map, and is similar to a node list, except for some differences in methods and properties.

An attribute list keeps itself up-to-date. If an attribute is deleted or added, the list is automatically updated.

This code fragment returns a list of attribute nodes from the first <book> element in "books.xml":

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

After the execution of the code above, x.length = is the number of attributes and x.getNamedItem() can be used to return an attribute node.

This code fragment gets the value of the "category" attribute, and the number of attributes, of a book:

Example

```
x = xmlDoc.getElementsByTagName("book")[0].attributes;  
  
txt = x.getNamedItem("category").nodeValue + " " + x.length;
```

Output:

```
cooking 1
```

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Set the x variable to hold a list of all attributes of the first <book> element
3. Get the value of the "category" attribute and the length of the attribute list

XML DOM Traverse Node Tree

Traversing means looping through or traveling across the node tree.

Traversing the Node Tree

Often you want to loop an XML document, for example: when you want to extract the value of each element.

This is called "Traversing the node tree"

The example below loops through all child nodes of <book>, and displays their names and values:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var x, i ,xmlDoc;
var txt = "";
var text = "<book>" +
"<title>Everyday Italian</title>" +
"<author>Giada De Laurentiis</author>" +
"<year>2005</year>" +
"</book>";

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

// documentElement always represents the root node
x = xmlDoc.documentElement.childNodes;
for (i = 0; i < x.length ;i++) {
    txt += x[i].nodeName + ": " + x[i].childNodes[0].nodeValue + "<br>";
}
document.getElementById("demo").innerHTML = txt;
</script>

</body>
</html>
```

Output:

```
title: Everyday Italian  
author: Giada De Laurentiis  
year: 2005
```

Example explained:

1. Load the XML string into xmlDoc
2. Get the child nodes of the root element
3. For each child node, output the node name and the node value of the text node

Browser Differences in DOM Parsing

All modern browsers support the W3C DOM specification.

However, there are some differences between browsers. One important difference is:

- The way they handle white-spaces and new lines

DOM - White Spaces and New Lines

XML often contains new line, or white space characters, between nodes. This is often the case when the document is edited by a simple editor like Notepad.

The following example (edited by Notepad) contains CR/LF (new line) between each line and two spaces in front of each child node:

```
<book>
  <title>Everyday Italian</title>
  <author>Giada De Laurentiis</author>
  <year>2005</year>
  <price>30.00</price>
</book>
```

Internet Explorer 9 and earlier do NOT treat empty white-spaces, or new lines as text nodes, while other browsers do.

The following example will output the number of child nodes the root element (of [books.xml](#)) has. IE9 and earlier will output 4 child nodes, while IE10 and later versions, and other browsers will output 9 child nodes:

Example

```
function myFunction(xml) {
var xmlDoc = xml.responseXML;
x = xmlDoc.documentElement.childNodes;
document.getElementById("demo").innerHTML =
"Number of child nodes: " + x.length;
}
```

PCDATA - Parsed Character Data

XML parsers normally parse all the text in an XML document.

When an XML element is parsed, the text between the XML tags is also parsed:

```
<message>This text is also parsed</message>
```

The parser does this because XML elements can contain other elements, as in this example, where the `<name>` element contains two other elements (first and last):

```
<name><first>Bill</first><last>Gates</last></name>
```

and the parser will break it up into sub-elements like this:

```
<name>
  <first>Bill</first>
  <last>Gates</last>
</name>
```

Parsed Character Data (PCDATA) is a term used about text data that will be parsed by the XML parser.

CDATA - (Unparsed) Character Data

The term CDATA is used about text data that should not be parsed by the XML parser.

Characters like "<" and "&" are illegal in XML elements.

"<" will generate an error because the parser interprets it as the start of a new element.

"&" will generate an error because the parser interprets it as the start of an character entity.

Some text, like JavaScript code, contains a lot of "<" or "&" characters. To avoid errors script code can be defined as CDATA.

Everything inside a CDATA section is ignored by the parser.

A CDATA section starts with "<![CDATA[" and ends with "]]>":

```
<script>
<![CDATA[
function matchwo(a,b) {
    if (a < b && a < 0) {
        return 1;
    } else {
        return 0;
    }
}]
</script>
```

In the example above, everything inside the CDATA section is ignored by the parser.

Notes on CDATA sections:

A CDATA section cannot contain the string "]]>". Nested CDATA sections are not allowed.

The "]]>" that marks the end of the CDATA section cannot contain spaces or line breaks.

XML DOM - Navigating Nodes

Nodes can be navigated using node relationships.

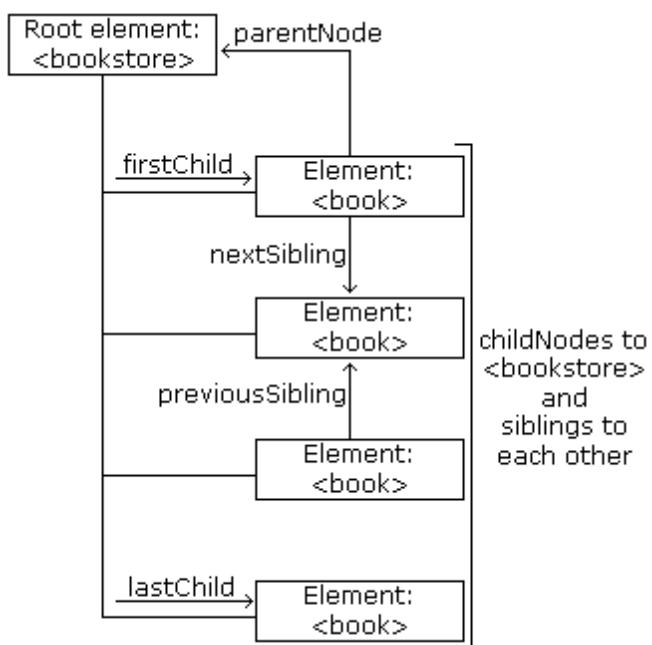
Navigating DOM Nodes

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

The following image illustrates a part of the node tree and the relationship between nodes in [books.xml](#):



DOM - Parent Node

All nodes have exactly one parent node. The following code navigates to the parent node of <book>:

Example

```
function myFunction(xml) {  
    var xmlDoc = xml.responseXML;  
    var x = xmlDoc.getElementsByTagName("book")[0];  
    document.getElementById("demo").innerHTML = x.parentNode.nodeName;  
}
```

Example explained:

1. Load "books.xml" into xmlDoc
2. Get the first <book> element
3. Output the node name of the parent node of "x"

Avoid Empty Text Nodes

Firefox, and some other browsers, will treat empty white-spaces or new lines as text nodes, Internet Explorer will not.

This causes a problem when using the properties: firstChild, lastChild, nextSibling, previousSibling.

To avoid navigating to empty text nodes (spaces and new-line characters between element nodes), we use a function that checks the node type:

```
function get_nextSibling(n) {  
    var y = n.nextSibling;  
    while (y.nodeType! = 1) {  
        y = y.nextSibling;  
    }  
    return y;  
}
```

The function above allows you to use `get_nextSibling(node)` instead of the property `node.nextSibling`.

Code explained:

Element nodes are type 1. If the sibling node is not an element node, it moves to the next nodes until an element node is found. This way, the result will be the same in both Internet Explorer and Firefox.

Get the First Child Element

The following code displays the first element node of the first <book>:

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myFunction(this);
    }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
    var xmlDoc = xml.responseXML;
    var x = get.firstChild(xmlDoc.getElementsByTagName("book")[0]);
    document.getElementById("demo").innerHTML = x.nodeName;
}

//check if the first node is an element node
function getFirstChild(n) {
    var y = n.firstChild;
    while (y.nodeType != 1) {
        y = y.nextSibling;
    }
    return y;
}
</script>

</body>
</html>
```

Output:

title

Example explained:

1. Load "books.xml" into xmlDoc
2. Use the get.firstChild function on the first <book> element node to get the first child node that is an element node
3. Output the node name of first child node that is an element node

More Examples

lastChild() This example uses the lastChild() method and a custom function to get the last child node of a node

nextSibling() This example uses the nextSibling() method and a custom function to get the next sibling node of a node

previousSibling() This example uses the previousSibling() method and a custom function to get the previous sibling node of a node

XML DOM Get Node Values

The `nodeValue` property is used to get the text value of a node.

The `getAttribute()` method returns the value of an attribute.

Get the Value of an Element

In the DOM, everything is a node. Element nodes do not have a text value.

The text value of an element node is stored in a child node. This node is called a text node.

To retrieve the text value of an element, you must retrieve the value of the elements' text node.

The `getElementsByTagName` Method

The `getElementsByTagName()` method returns a **node list of all elements**, with the specified tag name, in the same order as they appear in the source document.

Suppose "[books.xml](#)" has been loaded into `xmlDoc`.

This code retrieves the first `<title>` element:

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

The `childNodes` Property

The `childNodes` property returns a **list of an element's child nodes**.

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

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

The nodeValue Property

The **nodeValue** property returns the **text value of a text node**.

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

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.childNodes[0];
z = y.nodeValue;
```

Result in z: "Everyday Italian"

Complete Example

Example

```
<!DOCTYPE html>
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
    if (this.readyState == 4 && this.status == 200) {
        myFunction(this);
    }
};
xhttp.open("GET", "books.xml", true);
xhttp.send();

function myFunction(xml) {
    var xmlDoc = xml.responseXML;
    var x = xmlDoc.getElementsByTagName('title')[0];
    var y = x.childNodes[0];
    document.getElementById("demo").innerHTML = y.nodeValue;
}
</script>

</body>
</html>
```

Loop through all <title> elements: [Try it Yourself](#)

Get the Value of an Attribute

In the DOM, attributes are nodes. Unlike element nodes, attribute nodes have text values.

The way to get the value of an attribute, is to get its text value.

This can be done using the `getAttribute()` method or using the `nodeValue` property of the attribute node.

Get an Attribute Value - `getAttribute()`

The `getAttribute()` method returns an **attribute's value**.

The following code retrieves the text value of the "lang" attribute of the first `<title>` element:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
txt = x.getAttribute("lang");
```

Result in txt: "en"

Loop through all `<book>` elements and get their "category" attributes: [Try it yourself](#)

Get an Attribute Value - `getAttributeNode()`

The `getAttributeNode()` method returns an **attribute node**.

The following code retrieves the text value of the "lang" attribute of the first `<title>` element:

Example

```
x = xmlDoc.getElementsByTagName("title")[0];
y = x.getAttributeNode("lang");
txt = y.nodeValue;
```

Result in txt = "en"

Loop through all `<book>` elements and get their "category" attributes: [Try it Yourself](#)

XML DOM Change Node Values

The `nodeValue` property is used to change a node value.

The `setAttribute()` method is used to change an attribute value.

Try it Yourself - Examples

The examples below use the XML file `books.xml`.

[Change an element's text node](#) This example uses the `nodeValue` property to change the text node of the first `<title>` element in "books.xml".

[Change an attribute's value using `setAttribute`](#) This example uses the `setAttribute()` method to change the value of the "category" attribute of the first `<book>`.

[Change an attribute's value using `nodeValue`](#) This example use the `nodeValue` property to change the value of the "category" attribute of the first `<book>`.

Change the Value of an Element

In the DOM, everything is a node. Element nodes do not have a text value.

The text value of an element node is stored in a child node. This node is called a text node.

To change the text value of an element, you must change the value of the elements's text node.

Change the Value of a Text Node

The **`nodeValue`** property can be used to change **the value of a text node**.

Suppose "`books.xml`" has been loaded into `xmlDoc`.

This code changes the text node value of the first `<title>` element:

Example

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

Example explained:

1. Suppose "`books.xml`" is loaded into `xmlDoc`
2. Get the first child node of the `<title>` element
3. Change the node value to "new content"

Loop through and change the text node of all `<title>` elements: [Try it yourself](#)

Change the Value of an Attribute

In the DOM, attributes are nodes. Unlike element nodes, attribute nodes have text values. The way to change the value of an attribute, is to change its text value. This can be done using the `setAttribute()` method or setting the `nodeValue` property of the attribute node.

Change an Attribute Using `setAttribute()`

The **`setAttribute()`** method **changes the value of an attribute**.

If the attribute does not exist, a new attribute is created.

This code changes the category attribute of the `<book>` element:

Example

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

Example explained:

1. Suppose "books.xml" is loaded into `xmlDoc`
2. Get the first `<book>` element
3. Change the "category" attribute value to "food"

Loop through all `<title>` elements and add a new attribute: [Try it yourself](#)

Note: If the attribute does not exist, a new attribute is created (with the name and value specified).

Change an Attribute Using `nodeValue`

The **`nodeValue`** property is **the value of a attribute node**.

Changing the `value` property changes the value of the attribute.

Example

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

Example explained:

1. Suppose "books.xml" is loaded into `xmlDoc`
2. Get the "category" attribute of the first `<book>` element
3. Change the attribute node value to "food"

XML DOM Remove Nodes

The `removeChild()` method removes a specified node.

The `removeAttribute()` method removes a specified attribute.

Try it Yourself - Examples

The examples use the XML file [books.xml](#).

[Remove an element node](#) This example uses `removeChild()` to remove the first `<book>` element.

[Remove the current element node](#) This example uses `parentNode` and `removeChild()` to remove the current `<book>` element.

[Remove a text node](#) This example uses `removeChild()` to remove the text node from the first `<title>` element.

[Clear the text of a text node](#) This example uses the `nodeValue()` property to clear the text node of the first `<title>` element.

[Remove an attribute by name](#) This example uses `removeAttribute()` to remove the "category" attribute from the first `<book>` element.

[Remove attributes by object](#) This example uses `removeAttributeNode()` to remove all attributes from all `<book>` elements.

Remove an Element Node

The **`removeChild()`** method removes a specified node.

When a node is removed, all its child nodes are also removed.

This code will remove the first `<book>` element from the loaded xml:

Example

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

Example explained:

1. Suppose "[books.xml](#)" is loaded into `xmlDoc`
2. Set the variable `y` to be the element node to remove
3. Remove the element node by using the `removeChild()` method from the parent node

Remove Myself - Remove the Current Node

The `removeChild()` method is the only way to remove a specified node.

When you have navigated to the node you want to remove, it is possible to remove that node using the `parentNode` property and the `removeChild()` method:

Example

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

Example explained:

1. Suppose "books.xml" is loaded into `xmlDoc`
2. Set the variable `y` to be the element node to remove
3. Remove the element node by using the `parentNode` property and the `removeChild()` method

Remove a Text Node

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

Example

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

Example explained:

1. Suppose "books.xml" is loaded into `xmlDoc`
2. Set the variable `x` to be the first title element node
3. Set the variable `y` to be the text node to remove
4. Remove the element node by using the `removeChild()` method from the parent node

It is not very common to use `removeChild()` just to remove the text from a node. The `nodeValue` property can be used instead. See next paragraph.

Clear a Text Node

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

Example

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

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Get the first title element's first child node.
3. Use the `nodeValue` property to clear the text from the text node

Remove an Attribute Node by Name

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

Example: `removeAttribute('category')`

This code removes the "category" attribute in the first <book> element:

Example

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

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Use `getElementsByTagName()` to get book nodes
3. Remove the "category" attribute form the first book element node

Loop through and remove the "category" attribute of all <book> elements: [Try it yourself](#)

Remove Attribute Nodes by Object

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

Example: removeAttributeNode(x)

This code removes all the attributes of all <book> elements:

Example

```
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);
    }
}
```

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Use `getElementsByTagName()` to get all book nodes
3. For each book element check if there are any attributes
4. While there are attributes in a book element, remove the attribute

XML DOM Replace Nodes

The `replaceChild()` method replaces a specified node.

The `nodeValue` property replaces text in a text node.

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Replace an element node](#) This example uses `replaceChild()` to replace the first `<book>` node.

[Replace data in a text node](#) This example uses the `nodeValue` property to replace data in a text node.

Replace an Element Node

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

The following code fragment replaces the first `<book>` element:

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);
```

Example explained:

1. Load "[books.xml](#)" into xmlDoc
2. Create a new element node <book>
3. Create a new element node <title>
4. Create a new text node with the text "A Notebook"
5. Append the new text node to the new element node <title>
6. Append the new element node <title> to the new element node <book>
7. Replace the first <book> element node with the new <book> element node

Replace Data In a Text Node

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

The replaceData() method has three parameters:

- offset - Where to begin replacing characters. Offset value starts at zero
- length - How many characters to replace
- string - The string to insert

Example

```
xmlDoc=loadXMLDoc("books.xml");

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

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

Example explained:

Use the nodeValue Property Instead

The following code fragment will replace the text node value in the first <title> element with "Easy Italian":

Example

```
xmlDoc=loadXMLDoc("books.xml");

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

x.nodeValue="Easy Italian";
```

Example explained:

You can read more about changing node values in the [Change Node chapter](#).

XML DOM Create Nodes

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Create an element node](#) This example uses createElement() to create a new element node, and appendChild() to add it to a node.

[Create an attribute node using createAttribute](#) This example uses createAttribute() to create a new attribute node, and setAttributeNode() to insert it to an element.

[Create an attribute node using setAttribute](#) This example uses setAttribute() to create a new attribute for an element.

[Create a text node](#) This example uses createTextNode() to create a new text node, and appendChild() to add it to an element.

[Create a CDATA section node](#) This example uses createCDATAsection() to create a CDATA section node, and appendChild() to add it to an element.

[Create a comment node](#) This example uses createComment() to create a comment node, and appendChild() to add it to an element.

Create a New Element Node

The **createElement()** method creates a new element node:

Example

```
newElement = xmlDoc.createElement("edition");
xmlDoc.getElementsByTagName("book")[0].appendChild(newElement);
```

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Create a new element node <edition>
3. Append the element node to the first <book> element

Loop through and add an element to all <book> elements: [Try it yourself](#)

Create a New Attribute Node

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

Example

```
newAtt = xmlDoc.createAttribute("edition");
newAtt.nodeValue = "first";

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

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Create a new attribute node "edition"
3. Set the value of the attribute node to "first"
4. Add the new attribute node to the first <title> element

Loop through all <title> elements and add a new attribute node: [Try it yourself](#)

If the attribute already exists, it is replaced by the new one.

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.

Example

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

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Set the attribute "edition" value to "first" for the first <book> element

Loop through all <title> elements and add a new attribute: [Try it yourself](#)

Create a Text Node

The **createTextNode()** method creates a new text node:

Example

```
newEle = xmlDoc.createElement("edition");
newText = xmlDoc.createTextNode("first");
newEle.appendChild(newText);

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

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Create a new element node <edition>
3. Create a new text node with the text "first"
4. Append the new text node to the element node
5. Append the new element node to the first <book> element

Add an element node, with a text node, to all <book> elements: [Try it yourself](#)

Create a CDATA Section Node

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

Example

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

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

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Create a new CDATA section node
3. Append the new CDATA node to the first <book> element

Loop through, and add a CDATA section, to all <book> elements: [Try it yourself](#)

Create a Comment Node

The **createComment()** method creates a new comment node.

Example

```
newComment = xmlDoc.createComment("Revised March 2015");  
  
xmlDoc.getElementsByTagName("book")[0].appendChild(newComment);
```

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc using
2. Create a new comment node
3. Append the new comment node to the first <book> element

Loop through, and add a comment node, to all <book> elements: [Try it yourself](#)

XML DOM Add Nodes

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Add a node after the last child node](#) This example uses appendChild() to add a child node to an existing node.

[Add a node before a specified child node](#) This example uses insertBefore() to insert a node before a specified child node.

[Add a new attribute](#) This example uses the setAttribute() method to add a new attribute.

[Add data to a text node](#) This example uses insertData() to insert data into an existing text node.

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.

Note: Use insertBefore() if the position of the node is important.

This code fragment creates an element (<edition>), and adds it after the last child of the first <book> element:

Example

```
newEle = xmlDoc.createElement("edition");
xmlDoc.getElementsByTagName("book")[0].appendChild(newEle);
```

Example explained:

1. Suppose "[books.xml](#)" is loaded into xmlDoc
2. Create a new node <edition>
3. Append the node to the first <book> element

This code fragment does the same as above, but the new element is added with a value:

Example

```
newEle = xmlDoc.createElement("edition");
newText=xmlDoc.createTextNode("first");
newEle.appendChild(newText);

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

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Create a new node <edition>
3. Create a new text node "first"
4. Append the text node to the <edition> node
5. Append the <addition> node to the <book> element

Insert a Node - insertBefore()

The **insertBefore()** method inserts a node before a specified child node. This method is useful when the position of the added node is important:

Example

```
newNode = xmlDoc.createElement("book");

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

x.insertBefore(newNode,y);
```

Example explained:

1. Suppose "books.xml" is loaded into xmlDoc
2. Create a new element node <book>
3. Insert the new node in front of the last <book> element node

If the second parameter of insertBefore() is null, the new node will be added after the last existing child node.

x.insertBefore(newNode,null) and **x.appendChild(newNode)** will both append a new child node to x.

Add a New Attribute

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

Example

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

Example explained:

1. Suppose "books.xml" has been loaded into xmlDoc
2. Set the value of the attribute "edition" to "first" for the first <book> element

There is no method called add Attribute() The setAttribute() will create a new attribute if the attribute does not exist.

Note: If the attribute already exists, the setAttribute() method will overwrite the existing value.

Add Text to a Text Node - insertData()

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

The insertData() method has two parameters:

- offset - Where to begin inserting characters (starts at zero)
- string - The string to insert

The following code fragment will add "Easy" to the text node of the first <title> element of the loaded XML:

Example

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

XML DOM Clone Nodes

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Copy a node and append it to an existing node](#) This example uses `cloneNode()` to copy a node and append it to the root node of the XML document

Copy a Node

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

The `cloneNode()` method has a parameter (true or false). This parameter indicates if the cloned node should include all attributes and child nodes of the original node.

The following code fragment copies the first `<book>` node and appends it to the root node of the document:

Example

```
oldNode = xmlDoc.getElementsByTagName('book')[0];
newNode = oldNode.cloneNode(true);
xmlDoc.documentElement.appendChild(newNode);
```

Result:

```
Everyday Italian
Harry Potter
XQuery Kick Start
Learning XML
Everyday Italian
```

Example explained:

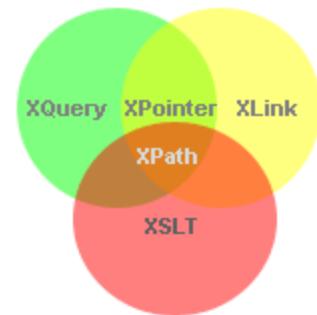
1. Suppose "[books.xml](#)" is loaded into `xmlDoc`
2. Get the node to copy (`oldNode`)
3. Clone the node into "`newNode`"
4. Append the new node to the root node of the XML document

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 Path Expressions

XPath uses path expressions to select nodes or node-sets in an XML document.

These path expressions look very much like the path expressions you use with traditional computer file systems:



XPath Standard Functions

XPath includes over 200 built-in functions.

There are functions for string values, numeric values, booleans, date and time comparison, node manipulation, sequence manipulation, and much more.

Today XPath expressions can also be used in JavaScript, Java, XML Schema, PHP, Python, C and C++, and lots of other languages.

XPath is Used in XSLT

XPath is a major element in the XSLT standard.

With XPath knowledge you will be able to take great advantage of your XSLT knowledge.

XPath is a W3C Recommendation

XPath 1.0 became a W3C Recommendation on November 16, 1999.

XPath 2.0 became a W3C Recommendation on January 23, 2007.

XPath 3.0 became a W3C Recommendation on April 8, 2014.

XPath Nodes

XPath Terminology

Nodes

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

XML documents are treated as trees of nodes. The topmost element of the tree is called the root element.

Look at the following XML document:

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

<bookstore>
  <book>
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
</bookstore>
```

Example of nodes in the XML document above:

```
<bookstore> (root element node)

<author>J K. Rowling</author> (element node)

lang="en" (attribute node)
```

Atomic values

Atomic values are nodes with no children or parent.

Example of atomic values:

J K. Rowling

"en"

Items

Items are atomic values or nodes.

Relationship of Nodes

Parent

Each element and attribute has one parent.

In the following example; the book element is the parent of the title, author, year, and price:

```
<book>
  <title>Harry Potter</title>
  <author>J. K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Children

Element nodes may have zero, one or more children.

In the following example; the title, author, year, and price elements are all children of the book element:

```
<book>
  <title>Harry Potter</title>
  <author>J. K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Siblings

Nodes that have the same parent.

In the following example; the title, author, year, and price elements are all siblings:

```
<book>
  <title>Harry Potter</title>
  <author>J. K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>
```

Ancestors

A node's parent, parent's parent, etc.

In the following example; the ancestors of the title element are the book element and the bookstore element:

```
<bookstore>

<book>
  <title>Harry Potter</title>
  <author>J. K. Rowling</author>
  <year>2005</year>
  <price>29.99</price>
</book>

</bookstore>
```

Descendants

A node's children, children's children, etc.

In the following example; descendants of the bookstore element are the book, title, author, year, and price elements:

```
<bookstore>

  <book>
    <title>Harry Potter</title>
    <author>J. K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>

</bookstore>
```

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.

The XML Example Document

We will use the following XML document in the examples below.

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

<bookstore>

  <book>
    <title lang="en">Harry Potter</title>
    <price>29.99</price>
  </book>

  <book>
    <title lang="en">Learning XML</title>
    <price>39.95</price>
  </book>

</bookstore>
```

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:

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

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

Path Expression	Result
bookstore	Selects all nodes with the name "bookstore"
/bookstore	Selects the root element bookstore Note: If the path starts with a slash (/) it always represents an absolute path to an element!
bookstore/book	Selects all book elements that are children of bookstore
//book	Selects all book elements no matter where they are in the document
bookstore//book	Selects all book elements that are descendant of the bookstore element, no matter where they are under the bookstore element
//@lang	Selects all attributes that are named lang

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:

Path Expression	Result
/bookstore/book[1]	Selects the first book element that is the child of the bookstore element. Note: In IE 5,6,7,8,9 first node is[0], but according to W3C, it is [1]. To solve this problem in IE, set the SelectionLanguage to XPath: <i>In JavaScript:</i> <code>xml.setProperty("SelectionLanguage","XPath");</code>
/bookstore/book[last()]	Selects the last book element that is the child of the bookstore element
/bookstore/book[last()-1]	Selects the last but one book element that is the child of the bookstore element
/bookstore/book[position()<3]	Selects the first two book elements that are children of the bookstore element
//title[@lang]	Selects all the title elements that have an attribute named lang
//title[@lang='en']	Selects all the title elements that have a "lang" attribute with a value of "en"
/bookstore/book[price>35.00]	Selects all the book elements of the bookstore element that have a price element with a value greater than 35.00
/bookstore/book[price>35.00]/title	Selects all the title elements of the book elements of the bookstore element that have a price element with a value greater than 35.00

Selecting Unknown Nodes

XPath wildcards can be used to select unknown XML nodes.

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

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

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

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:

Path Expression	Result
//book/title //book/price	Selects all the title AND price elements of all book elements
//title //price	Selects all the title AND price elements in the document
/bookstore/book/title //price	Selects all the title elements of the book element of the bookstore element AND all the price elements in the document

XPath Axes

The XML Example Document

We will use the following XML document in the examples below.

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

<bookstore>

<book>
  <title lang="en">Harry Potter</title>
  <price>29.99</price>
</book>

<book>
  <title lang="en">Learning XML</title>
  <price>39.95</price>
</book>

</bookstore>
```

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.

AxisName	Result
ancestor	Selects all ancestors (parent, grandparent, etc.) of the current node
ancestor-or-self	Selects all ancestors (parent, grandparent, etc.) of the current node and the current node itself
attribute	Selects all attributes of the current node
child	Selects all children of the current node
descendant	Selects all descendants (children, grandchildren, etc.) of the current node
descendant-or-self	Selects all descendants (children, grandchildren, etc.) of the current node and the current node itself
following	Selects everything in the document after the closing tag of the current node
following-sibling	Selects all siblings after the current node
namespace	Selects all namespace nodes of the current node
parent	Selects the parent of the current node
preceding	Selects all nodes that appear before the current node in the document, except ancestors, attribute nodes and namespace nodes
preceding-sibling	Selects all siblings before the current node
self	Selects the current node

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]
```

Examples

Example	Result
child::book	Selects all book nodes that are children of the current node
attribute::lang	Selects the lang attribute of the current node
child::*	Selects all element children of the current node
attribute::*	Selects all attributes of the current node
child::text()	Selects all text node children of the current node
child::node()	Selects all children of the current node
descendant::book	Selects all book descendants of the current node
ancestor::book	Selects all book ancestors of the current node
ancestor-or-self::book	Selects all book ancestors of the current node - and the current as well if it is a book node
child::*/child::price	Selects all price grandchildren of the current node

XPath Operators

An XPath expression returns either a node-set, a string, a Boolean, or a number.

XPath Operators

Below is a list of the operators that can be used in XPath expressions:

Operator	Description	Example
	Computes two node-sets	//book //cd
+	Addition	6 + 4
-	Subtraction	6 - 4
*	Multiplication	6 * 4
div	Division	8 div 4
=	Equal	price=9.80
!=	Not equal	price!=9.80
<	Less than	price<9.80
<=	Less than or equal to	price<=9.80
>	Greater than	price>9.80
>=	Greater than or equal to	price>=9.80
or	or	price=9.80 or price=9.70
and	and	price>9.00 and price<9.90
mod	Modulus (division remainder)	5 mod 2

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).

Online XSLT Editor

With our online editor, you can edit XML and XSLT code, and click on a button to view the result.

XSLT Example

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

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

<xsl:template match="/">
  <html>
    <body>
      <h2>My CD Collection</h2>
      <table border="1">
        <tr bgcolor="#9acd32">
          <th>Title</th>
          <th>Artist</th>
        </tr>
        <xsl:for-each select="catalog/cd">
          <tr>
            <td><xsl:value-of select="title"/></td>
            <td><xsl:value-of select="artist"/></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

What You Should Already Know

Before you continue you should have a basic understanding of the following:

- HTML
- XML

If you want to study these subjects first, find the tutorials on our [Home page](#).

XSLT References

XSLT Elements

Description of all the XSLT elements from the W3C Recommendation, and information about browser support.

XSLT, XPath, and XQuery Functions

XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library. There are over 100 built-in functions. There are functions for string values, numeric values, date and time comparison, node and QName manipulation, sequence manipulation, and more.

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

With the **CSS3 Paged Media Module**, W3C has delivered a new standard for document formatting. So, since 2013, CSS3 is proposed as an XSL-FO replacement.

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.

If you want to study XPath first, please read our [XPath Tutorial](#).

How Does it Work?

In the transformation process, XSLT uses XPath to define parts of the source document that should match one or more predefined templates. When a match is found, XSLT will transform the matching part of the source document into the result document.

XSLT Browser Support

All major browsers support XSLT and XPath.

XSLT is a W3C Recommendation

XSLT became a W3C Recommendation 16. November 1999.

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:

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

or:

```
<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
```

To get access to the XSLT elements, attributes and features we must declare the XSLT namespace at the top of the document.

The `xmlns:xsl="http://www.w3.org/1999/XSL/Transform"` points to the official W3C XSLT namespace. If you use this namespace, you must also include the attribute `version="1.0"`.

Start with a Raw XML Document

We want to **transform** the following XML document ("cdcatalog.xml") into XHTML:

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

Viewing XML Files in IE, Chrome, Firefox, Safari, and Opera: Open the XML file (click on the link below) - The XML document will be displayed with color-coded root and child elements (except in Safari). Often, there is a plus (+) or minus sign (-) to the left of the elements that can be clicked to expand or collapse the element structure. **Tip: To view the raw XML source, right-click in XML file and select "View Source"!**

[View "cdcatalog.xml"](#)

Create an XSL Style Sheet

Then you create an XSL Style Sheet ("cdcatalog.xsl") with a transformation template:

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

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

<xsl:template match="/">
  <html>
    <body>
      <h2>My CD Collection</h2>
      <table border="1">
        <tr bgcolor="#9acd32">
          <th>Title</th>
          <th>Artist</th>
        </tr>
        <xsl:for-each select="catalog/cd">
          <tr>
            <td><xsl:value-of select="title"/></td>
            <td><xsl:value-of select="artist"/></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

[View "cdcatalog.xsl"](#)

Link the XSL Style Sheet to the XML Document

Add the XSL style sheet reference to your XML document ("cdcatalog.xml"):

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

If you have an XSLT compliant browser it will nicely **transform** your XML into XHTML.

[View the result](#)

The details of the example above will be explained in the next chapters.

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).

Ok, let's look at a simplified version of the XSL file from the previous chapter:

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

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

```
<?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><xsl:value-of select="catalog/cd/title"/></td>
          <td><xsl:value-of select="catalog/cd/artist"/></td>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

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:for-each> Element

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

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>
        <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>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

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)
- < less than
- > greater than

Take a look at the adjusted XSL style sheet:

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>
          <xsl:for-each select="catalog/cd[artist='Bob Dylan']">
            <tr>
              <td><xsl:value-of select="title"/></td>
              <td><xsl:value-of select="artist"/></td>
            </tr>
          </xsl:for-each>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

XSLT <xsl:sort> Element

The <xsl:sort> element is used to sort the output.

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:

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>
        <xsl:for-each select="catalog/cd">
          <xsl:sort select="artist"/>
          <tr>
            <td><xsl:value-of select="title"/></td>
            <td><xsl:value-of select="artist"/></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

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.

The <xsl:if> Element

To put a conditional if test against the content of the XML file, add an <xsl:if> element to the XSL document.

Syntax

```
<xsl:if test="expression">  
    ...some output if the expression is true...  
</xsl:if>
```

Where to Put the <xsl:if> Element

To add a conditional test, add the <xsl:if> element inside the <xsl:for-each> element in the XSL file:

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>
          <th>Price</th>
        </tr>
        <xsl:for-each select="catalog/cd">
          <xsl:if test="price > 10">
            <tr>
              <td><xsl:value-of select="title"/></td>
              <td><xsl:value-of select="artist"/></td>
              <td><xsl:value-of select="price"/></td>
            </tr>
          </xsl:if>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

Note: The value of the required **test** attribute contains the expression to be evaluated. The code above will only output the title and artist elements of the CDs that has a price that is higher than 10.

XSLT <xsl:choose> Element

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

The <xsl:choose> Element

Syntax

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

Where to put the Choose Condition

To insert a multiple conditional test against the XML file, add the `<xsl:choose>`, `<xsl:when>`, and `<xsl:otherwise>` elements to the XSL file:

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>
        <xsl:for-each select="catalog/cd">
          <tr>
            <td><xsl:value-of select="title"/></td>
            <xsl:choose>
              <xsl:when test="price > 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>
        </table>
      </body>
    </html>
  </xsl:template>

</xsl:stylesheet>
```

The code above will add a pink background-color to the "Artist" column WHEN the price of the CD is higher than 10.

Another Example

Here is another example that contains two <xsl:when> elements:

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>
        <xsl:for-each select="catalog/cd">
          <tr>
            <td><xsl:value-of select="title"/></td>
            <xsl:choose>
              <xsl:when test="price > 10">
                <td bgcolor="#ff00ff">
                  <xsl:value-of select="artist"/></td>
                </xsl:when>
              <xsl:when test="price > 9">
                <td bgcolor="#cccccc">
                  <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>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

The code above will add a pink background color to the "Artist" column WHEN the price of the CD is higher than 10, and a grey background-color WHEN the price of the CD is higher than 9 and lower or equal to 10.

XSLT <xsl:apply-templates> Element

The <xsl:apply-templates> element applies a template 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 element that matches the value of the attribute. We can use the select attribute to specify the order in which the child nodes are processed.

Look at the following XSL style sheet:

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>

</xsl:stylesheet>
```

XSLT - On the Client

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

A JavaScript Solution

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:

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

View the XML file.

And the accompanying XSL style sheet:

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

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

<xsl:template match="/">
  <h2>My CD Collection</h2>
  <table border="1">
    <tr bgcolor="#9acd32">
      <th style="text-align:left">Title</th>
      <th style="text-align:left">Artist</th>
    </tr>
    <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>
  </table>
</xsl:template>

</xsl:stylesheet>
```

View the XSL file.

Notice that the XML file does not have a reference to the XSL file.

IMPORTANT: The above sentence indicates that an XML file could be transformed using many different XSL style sheets.

Transforming XML to XHTML in the Browser

Here is the source code needed to transform the XML file to XHTML on the client:

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>

```

Tip: If you don't know how to write JavaScript, please study our [JavaScript tutorial](#).

Example Explained:

The **loadXMLDoc()** function does the following:

- Create an XMLHttpRequest object
- Use the open() and send() methods of the XMLHttpRequest object to send a request to a server
- Get the response data as XML data

The **displayResult()** function is used to display the XML file styled by the XSL file:

- Load XML and XSL files
- Test what kind of browser the user has
- If Internet Explorer:
 - Use the transformNode() method to apply the XSL style sheet to the xml document
 - Set the body of the current document (id="example") to contain the styled xml document
- If other browsers:
 - Create a new XSLTProcessor object and import the XSL file to it
 - Use the transformToFragment() method to apply the XSL style sheet to the xml document
 - Set the body of the current document (id="example") to contain the styled xml document

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

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.

The XML File and the XSLT File

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

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

View the XML file.

And the accompanying XSL style sheet:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

  <xsl:template match="/">
    <h2>My CD Collection</h2>
    <table border="1">
      <tr bgcolor="#9acd32">
        <th style="text-align:left">Title</th>
        <th style="text-align:left">Artist</th>
      </tr>
      <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>
    </table>
  </xsl:template>

</xsl:stylesheet>
```

View the XSL file.

Notice that the XML file does not have a reference to the XSL file.

IMPORTANT: The above sentence indicates that an XML file could be transformed using many different XSL style sheets.

PHP Code: Transform XML to XHTML on the Server

Here is the PHP source code needed to transform the XML file to XHTML on the server:

```
<?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);  
?>
```

Tip: If you don't know how to write PHP, please study our [PHP tutorial](#). See [how it works with PHP](#).

ASP Code: Transform XML to XHTML on the Server

Here is the ASP source code needed to transform the XML file to XHTML on the server:

```
<%
'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))
%>
```

[See how it works with ASP.](#)

XSLT - Editing XML

Data stored in XML files can be edited from an Internet browser.

Open, Edit and Save XML

Now, we will show how to open, edit, and save an XML file that is stored on the server. We will use XSL to transform the XML document into an HTML form. The values of the XML elements will be written to HTML input fields in an HTML form. The HTML form is editable. After editing the data, the data is going to be submitted back to the server and the XML file will be updated (we will show code for both PHP and ASP).

The XML File and the XSL File

First, take a look at the XML document ("tool.xml"):

```
<?xml version="1.0" encoding="UTF-8"?>
<tool>
  <field id="prodName">
    <value>HAMMER HG2606</value>
  </field>
  <field id="prodNo">
    <value>32456240</value>
  </field>
  <field id="price">
    <value>$30.00</value>
  </field>
</tool>
```

View the XML file.

Then, take a look at the following style sheet ("tool.xsl"):

```
<?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>
      <form method="post" action="edittool.asp">
        <h2>Tool Information (edit):</h2>
        <table border="0">
          <xsl:for-each select="tool/field">
            <tr>
              <td><xsl:value-of select="@id"/></td>
              <td>
                <input type="text">
                <xsl:attribute name="id">
                  <xsl:value-of select="@id" />
                </xsl:attribute>
                <xsl:attribute name="name">
                  <xsl:value-of select="@id" />
                </xsl:attribute>
                <xsl:attribute name="value">
                  <xsl:value-of select="value" />
                </xsl:attribute>
                </input>
              </td>
            </tr>
          </xsl:for-each>
        </table>
        <br />
        <input type="submit" id="btn_sub" name="btn_sub" value="Submit" />
        <input type="reset" id="btn_res" name="btn_res" value="Reset" />
      </form>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

View the XSL file.

The XSL file above loops through the elements in the XML file and creates one input field for each XML "field" element. The value of the XML "field" element's "id" attribute is added to both the "id" and "name" attributes of each HTML input field. The value of each XML "value" element is added to the "value" attribute of each HTML input field. The result is an editable HTML form that contains the values from the XML file.

Then, we have a second style sheet: "tool_updated.xsl". This is the XSL file that will be used to display the updated XML data. This style sheet will not result in an editable HTML form, but a static HTML table:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">

<xsl:template match="/">
  <html>
    <body>
      <h2>Updated Tool Information:</h2>
      <table border="1">
        <xsl:for-each select="tool/field">
          <tr>
            <td><xsl:value-of select="@id" /></td>
            <td><xsl:value-of select="value" /></td>
          </tr>
        </xsl:for-each>
      </table>
    </body>
  </html>
</xsl:template>

</xsl:stylesheet>
```

View the XSL file.

The PHP File

In the "tool.xsl" file above, change the HTML form's action attribute to "edittool.php". The "edittool.php" page contains two functions: The loadFile() function loads and transforms the XML file for display and the updateFile() function applies the changes to the XML file:

```
<?php
function loadFile($xml, $xsl)
```

```

{
$xmlDoc = new DOMDocument();
$xmlDoc->load($xml);

$xslDoc = new DOMDocument();
$xslDoc->load($xsl);

$proc = new XSLTProcessor();
$proc->importStyleSheet($xslDoc);
echo $proc->transformToXML($xmlDoc);
}

function updateFile($xml)
{
$xmlLoad = simplexml_load_file($xml);
$postKeys = array_keys($_POST);

foreach($xmlLoad->children() as $x)
{
    foreach($_POST as $key=>$value)
    {
        if($key == $x->attributes())
        {
            $x->value = $value;
        }
    }
}

$xmlLoad->asXML($xml);
loadFile($xml,"tool_updated.xsl");
}

if($_POST["btn_sub"] == "")
{
    loadFile("tool.xml", "tool.xsl");
}
else
{
    updateFile("tool.xml");
}
?>
```

Tip: If you don't know how to write PHP, please study our [PHP tutorial](#).

Note: We are doing the transformation and applying the changes to the XML file on the server. This is a cross-browser solution. The client will only get HTML back from the server - which will work in any browser.

The ASP File

The HTML form in the "tool.xsl" file above has an action attribute with a value of "edittool.asp".

The "edittool.asp" page contains two functions: The loadFile() function loads and transforms the XML file for display and the updateFile() function applies the changes to the XML file:

```
<%
function loadFile(xmlfile,xslfile)
Dim xmlDoc,xslDoc
'Load XML and XSL file
set xmlDoc = Server.CreateObject("Microsoft.XMLDOM")
xmlDoc.async = false
xmlDoc.load(xmlfile)
set xslDoc = Server.CreateObject("Microsoft.XMLDOM")
xslDoc.async = false
xslDoc.load(xslfile)
'Transform file
Response.Write(xmlDoc.transformNode(xslDoc))
end function

function updateFile(xmlfile)
Dim xmlDoc,rootEl,f
Dim i
'Load XML file
set xmlDoc = Server.CreateObject("Microsoft.XMLDOM")
xmlDoc.async = false
xmlDoc.load(xmlfile)

'Set the rootEl variable equal to the root element
Set rootEl = xmlDoc.documentElement

'Loop through the form collection
for i = 1 To Request.Form.Count
  'Eliminate button elements in the form
  if instr(1,Request.Form.Key(i),"btn_")=0 then
    'The selectSingleNode method queries the XML file for a single node
    'that matches a query. This query requests the value element that is
    'the child of a field element that has an id attribute which matches
```

```

'the current key value in the Form Collection. When there is a match -
'set the text property equal to the value of the current field in the
'Form Collection.
set f = rootEl.selectSingleNode("field[@id=''" & _
Request.Form.Key(i) & "']/value")
f.Text = Request.Form(i)
end if
next

'Save the modified XML file
xmlDoc.save xmlfile

'Release all object references
set xmlDoc=nothing
set rootEl=nothing
set f=nothing

'Load the modified XML file with a style sheet that
'allows the client to see the edited information
loadFile xmlfile,server.MapPath("tool_updated.xsl")
end function

'If form is submitted, update the XML file and display result
' - if not, transform the XML file for editing
if Request.Form("btn_sub")="" then
    loadFile server.MapPath("tool.xml"),server.MapPath("tool.xsl")
else
    updateFile server.MapPath("tool.xml")
end if
%>

```

DTD Tutorial

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.

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.

An Internal DTD Declaration

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

XML document with an internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE note [
  <!ELEMENT note (to,from,heading,body)>
  <!ELEMENT to (#PCDATA)>
  <!ELEMENT from (#PCDATA)>
  <!ELEMENT heading (#PCDATA)>
  <!ELEMENT body (#PCDATA)>
]>
<note>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend</body>
</note>
```

[View XML file »](#)

In the XML file, select "view source" to view the DTD.

The DTD above is interpreted like this:

- **!DOCTYPE note** defines that the root element of this document is note
- **!ELEMENT note** defines that the note element must contain four elements: "to,from,heading,body"
- **!ELEMENT to** defines the to element to be of type "#PCDATA"
- **!ELEMENT from** defines the from element to be of type "#PCDATA"
- **!ELEMENT heading** defines the heading element to be of type "#PCDATA"
- **!ELEMENT body** defines the body element to be of type "#PCDATA"

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 document with a reference to an external DTD

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

[View XML file »](#)

And here is the file "note.dtd", which contains the DTD:

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

DTD - XML Building Blocks

The main building blocks of both XML and HTML documents are elements.

The Building Blocks of XML Documents

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

- Elements
- Attributes
- Entities
- PCDATA
- CDATA

Elements

Elements are the **main building blocks** of both XML and HTML documents.

Examples of HTML elements are "body" and "table". Examples of XML elements could be "note" and "message". Elements can contain text, other elements, or be empty. Examples of empty HTML elements are "hr", "br" and "img".

Examples:

```
<body>some text</body>

<message>some text</message>
```

Attributes

Attributes provide **extra information about elements**.

Attributes are always placed inside the opening tag of an element. Attributes always come in name/value pairs. The following "img" element has additional information about a source file:

```

```

The name of the element is "img". The name of the attribute is "src". The value of the attribute is "computer.gif". Since the element itself is empty it is closed by a "/".

Entities

Some characters have a special meaning in XML, like the less than sign (<) that defines the start of an XML tag.

Most of you know the HTML entity: " ". This "no-breaking-space" entity is used in HTML to insert an extra space in a document. Entities are expanded when a document is parsed by an XML parser.

The following entities are predefined in XML:

Entity References	Character
<	<
>	>
&	&
"	"
'	'

PCDATA

PCDATA means parsed character data.

Think of character data as the text found between the start tag and the end tag of an XML element.

PCDATA is text that WILL be parsed by a parser. The text will be examined by the parser for entities and markup.

Tags inside the text will be treated as markup and entities will be expanded.

However, parsed character data should not contain any &, <, or > characters; these need to be represented by the & < and > entities, respectively.

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.

DTD - 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>  
or  
<!ELEMENT element-name (element-content)>
```

Empty Elements

Empty elements are declared with the category keyword EMPTY:

```
<!ELEMENT element-name EMPTY>
```

Example:

```
<!ELEMENT br EMPTY>
```

XML example:

```
<br />
```

Elements with Parsed Character Data

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

```
<!ELEMENT element-name (#PCDATA)>
```

Example:

```
<!ELEMENT from (#PCDATA)>
```

Elements with any Contents

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

```
<!ELEMENT element-name ANY>
```

Example:

```
<!ELEMENT note 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)>
```

or

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

Example:

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

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)>
```

Example:

```
<!ELEMENT note (message)>
```

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+)>
```

Example:

```
<!ELEMENT note (message+)>
```

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*)>
```

Example:

```
<!ELEMENT note (message*)>
```

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?)>
```

Example:

```
<!ELEMENT note (message?)>
```

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))>
```

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)*>
```

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

DTD - 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" />
```

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

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

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

Value	Explanation
<i>value</i>	The default value of the attribute
#REQUIRED	The attribute is required
#IMPLIED	The attribute is optional
#FIXED <i>value</i>	The attribute value is fixed

A Default Attribute Value

DTD:

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

Valid XML:

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

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.

#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 />
```

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 />
```

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" />
```

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" />
```

or

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

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

XML Elements vs. Attributes

In XML, there are no rules about when to use attributes, and when to use child elements.

Use of Elements vs. Attributes

Data can be stored in child elements or in attributes.

Take a look at these examples:

```
<person sex="female">
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

```
<person>
  <sex>female</sex>
  <firstname>Anna</firstname>
  <lastname>Smith</lastname>
</person>
```

In the first example sex is an attribute. In the last, sex is a child element. Both examples provide the same information.

There are no rules about when to use attributes, and when to use child elements. My experience is that attributes are handy in HTML, but in XML you should try to avoid them. Use child elements if the information feels like data.

My Favorite Way

I like to store data in child elements.

The following three XML documents contain exactly the same information:

A date attribute is used in the first example:

```
<note date="12/11/2002">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

A date element is used in the second example:

```
<note>
  <date>12/11/2002</date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

An expanded date element is used in the third: (THIS IS MY FAVORITE):

```
<note>
  <date>
    <day>12</day>
    <month>11</month>
    <year>2002</year>
  </date>
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>
```

Avoid using attributes?

Should you 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

If you use attributes as containers for data, you end up with documents that are difficult to read and maintain. Try to use **elements** to describe data. Use attributes only to provide information that is not relevant to the data.

Don't end up like this (this is not how XML should be used):

```
<note day="12" month="11" year="2002"  
to="Tove" from="Jani" heading="Reminder"  
body="Don't forget me this weekend!">  
</note>
```

An Exception to my Attribute Rule

Rules always have exceptions.

My rule about attributes has one exception:

Sometimes I assign ID references to elements. These ID references can be used to access XML elements in much the same way as the NAME or ID attributes in HTML. This example demonstrates this:

```
<messages>
<note id="p501">
  <to>Tove</to>
  <from>Jani</from>
  <heading>Reminder</heading>
  <body>Don't forget me this weekend!</body>
</note>

<note id="p502">
  <to>Jani</to>
  <from>Tove</from>
  <heading>Re: Reminder</heading>
  <body>I will not!</body>
</note>
</messages>
```

The ID in these examples is just a counter, or a unique identifier, to identify the different notes in the XML file, and not a part of the note data.

What I am trying to say here is that metadata (data about data) should be stored as attributes, and that data itself should be stored as elements.

DTD - Entities

Entities are used to define shortcuts to special characters.

Entities can be declared internal or external.

An Internal Entity Declaration

Syntax

```
<!ENTITY entity-name "entity-value">
```

Example

DTD Example:

```
<!ENTITY writer "Donald Duck.">
<!ENTITY copyright "Copyright W3Schools.">
```

XML example:

```
<author>&writer;&copyright;</author>
```

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

An External Entity Declaration

Syntax

```
<!ENTITY entity-name SYSTEM "URI/URL">
```

Example

DTD Example:

```
<!ENTITY writer SYSTEM "https://www.w3schools.com/entities.dtd">
<!ENTITY copyright SYSTEM "https://www.w3schools.com/entities.dtd">
```

XML example:

```
<author>&writer;&copyright;</author>
```

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.

A Simple XML Document

Look at this simple XML document called "note.xml":

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

A DTD File

The following example is a DTD file called "note.dtd" that defines the elements of the XML document above ("note.xml"):

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

The first line defines the note element to have four child elements: "to, from, heading, body". Line 2-5 defines the to, from, heading, body elements to be of type "#PCDATA".

An XML Schema

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"?>
<xss: schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">

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

</xss: 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

```
<?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.

The <schema> Element

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

```
<?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.

Referencing a Schema in an XML Document

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 note.xsd">

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

The following fragment:

```
xmlns="https://www.w3schools.com"
```

specifies the default namespace declaration. This declaration tells the schema-validator that all the elements used in this XML document are declared in the "https://www.w3schools.com" namespace.

Once you have the XML Schema Instance namespace available:

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

you can use the schemaLocation attribute. This attribute has two values, separated by a space. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace:

```
xsi:schemaLocation="https://www.w3schools.com note.xsd"
```

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.

- 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"/>
      <xsmaxLength 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

A complex element contains other elements and/or attributes.

What is a Complex Element?

A complex element is an XML element that contains other elements and/or attributes.

There are four kinds of complex elements:

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

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

Examples of Complex Elements

A complex XML element, "product", which is empty:

```
<product pid="1345"/>
```

A complex XML element, "employee", which contains only other elements:

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

A complex XML element, "food", which contains only text:

```
<food type="dessert">Ice cream</food>
```

A complex XML element, "description", which contains both elements and text:

```
<description>
It happened on <date lang="norwegian">03.03.99</date> ....
</description>
```

How to Define a Complex Element

Look at this complex XML element, "employee", which contains only other elements:

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

We can define a complex element in an XML Schema two different ways:

1. The "employee" element can be declared directly by naming the element, like this:

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

If you use the method described above, only the "employee" element can use the specified complex type. Note that the child elements, "firstname" and "lastname", are surrounded by the <sequence> indicator. This means that the child elements must appear in the same order as they are declared. You will learn more about indicators in the XSD Indicators chapter.

2. The "employee" element can have a type attribute that refers to the name of the complex type to use:

```
<xs:element name="employee" 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>
```

If you use the method described above, several elements can refer to the same complex type, like this:

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

You can also base a complex element on an existing complex element and add some elements, like this:

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

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

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

XSD Empty Elements

An empty complex element cannot have contents, only attributes.

Complex Empty Elements

An empty XML element:

```
<product prodid="1345" />
```

The "product" element above has no content at all. To define a type with no content, we must define a type that allows elements in its content, but we do not actually declare any elements, like this:

```
<xss:element name="product">
  <xss:complexType>
    <xss:complexContent>
      <xss:restriction base="xss:integer">
        <xss:attribute name="prodid" type="xss:positiveInteger"/>
      </xss:restriction>
    </xss:complexContent>
  </xss:complexType>
</xss:element>
```

In the example above, we define a complex type with a complex content. The `complexContent` element signals that we intend to restrict or extend the content model of a complex type, and the restriction of integer declares one attribute but does not introduce any element content. However, it is possible to declare the "product" element more compactly, like this:

```
<xss:element name="product">
  <xss:complexType>
    <xss:attribute name="prodid" type="xss:positiveInteger"/>
  </xss:complexType>
</xss:element>
```

Or you can give the complexType element a name, and let the "product" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

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

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

XSD Elements Only

An "elements-only" complex type contains an element that contains only other elements.

Complex Types Containing Elements Only

An XML element, "person", that contains only other elements:

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

You can define the "person" element in a schema, like this:

```
<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>
```

Notice the `<xs:sequence>` tag. It means that the elements defined ("firstname" and "lastname") must appear in that order inside a "person" element.

Or you can give the complexType element a name, and let the "person" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

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

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

XSD Text-Only Elements

A complex text-only element can contain text and attributes.

Complex Text-Only Elements

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

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

OR

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

Tip: Use the extension/restriction element to expand or to limit the base simple type for the element.

Here is an example of an XML element, "shoesize", that contains text-only:

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

The following example declares a complexType, "shoesize". The content is defined as an integer value, and the "shoesize" element also contains an attribute named "country":

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

We could also give the complexType element a name, and let the "shoesize" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="shoesize" type="shoetype"/>

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

XSD Mixed Content

A mixed complex type element can contain attributes, elements, and text.

Complex Types with Mixed Content

An XML element, "letter", that contains both text and other elements:

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

The following schema declares the "letter" element:

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

Note: To enable character data to appear between the child-elements of "letter", the mixed attribute must be set to "true". The `<xss:sequence>` tag means that the elements defined (name, orderid and shipdate) must appear in that order inside a "letter" element.

We could also give the complexType element a name, and let the "letter" element have a type attribute that refers to the name of the complexType (if you use this method, several elements can refer to the same complex type):

```
<xs:element name="letter" type="lettertype"/>

<xs:complexType name="lettertype" 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>
```

XSD Indicators

We can control HOW elements are to be used in documents with indicators.

Indicators

There are seven indicators:

Order indicators:

- All
- Choice
- Sequence

Occurrence indicators:

- maxOccurs
- minOccurs

Group indicators:

- Group name
- attributeGroup name

Order Indicators

Order indicators are used to define the order of the elements.

All Indicator

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

```
<xss:element name="person">
  <xss:complexType>
    <xss:all>
      <xss:element name="firstname" type="xs:string"/>
      <xss:element name="lastname" type="xs:string"/>
    </xss:all>
  </xss:complexType>
</xss: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>
```

Occurrence Indicators

Occurrence indicators are used to define how often an element can occur.

Note: For all "Order" and "Group" indicators (any, all, choice, sequence, group name, and group reference) the default value for maxOccurs and minOccurs is 1.

maxOccurs Indicator

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

```
<xss:element name="person">
  <xss:complexType>
    <xss:sequence>
      <xss:element name="full_name" type="xs:string"/>
      <xss:element name="child_name" type="xs:string" maxOccurs="10"/>
    </xss:sequence>
  </xss:complexType>
</xss:element>
```

The example above indicates that the "child_name" element can occur a minimum of one time (the default value for minOccurs is 1) and a maximum of ten times in the "person" element.

minOccurs Indicator

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

```
<xss:element name="person">
  <xss:complexType>
    <xss:sequence>
      <xss:element name="full_name" type="xs:string"/>
      <xss:element name="child_name" type="xs:string"
        maxOccurs="10" minOccurs="0"/>
    </xss:sequence>
  </xss:complexType>
</xss:element>
```

The example above indicates that the "child_name" element can occur a minimum of zero times and a maximum of ten times in the "person" element.

Tip: To allow an element to appear an unlimited number of times, use the maxOccurs="unbounded" statement:

A working example:

An XML file called "Myfamily.xml":

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

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

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

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

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

</persons>
```

The XML file above contains a root element named "persons". Inside this root element we have defined three "person" elements. Each "person" element must contain a "full_name" element and it can contain up to five "child_name" elements.

Here is the schema file "family.xsd":

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

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

</xs:schema>
```

Group Indicators

Group indicators are used to define related sets of elements.

Element Groups

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

```
<xs:group name="groupname">
  ...
</xs:group>
```

You must define an all, choice, or sequence element inside the group declaration. The following example defines a group named "persongroup", that defines a group of elements that must occur in an exact sequence:

```
<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>
```

After you have defined a group, you can reference it in another definition, 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="groupname">
  ...
</xs:attributeGroup>
```

The following example defines an attribute group named "personattrgroup":

```
<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>
```

After you have defined an attribute group, you can reference it in another definition, 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>

<xs:element name="person">
  <xs:complexType>
    <xs:attributeGroup ref="personattrgroup"/>
  </xs:complexType>
</xs:element>
```

XSD The <any> Element

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

The <any> Element

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

The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <any> element we can extend (after <lastname>) the content of "person" with any element:

```
<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>
```

Now we want to extend the "person" element with a "children" element. In this case we can do so, even if the author of the schema above never declared any "children" element.

Look at this schema file, called "children.xsd":

```
<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">

<xss:element name="children">
  <xss:complexType>
    <xss:sequence>
      <xss:element name="childname" type="xs:string"
        maxOccurs="unbounded"/>
    </xss:sequence>
  </xss:complexType>
</xss:element>

</xss:schema>
```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "children.xsd":

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

<persons xmlns="http://www.microsoft.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.microsoft.com family.xsd
  https://www.w3schools.com children.xsd">

  <person>
    <firstname>Hege</firstname>
    <lastname>Rfsnes</lastname>
    <children>
      <childname>Cecilie</childname>
    </children>
  </person>

  <person>
    <firstname>Stale</firstname>
    <lastname>Rfsnes</lastname>
  </person>

</persons>
```

The XML file above is valid because the schema "family.xsd" allows us to extend the "person" element with an optional element after the "lastname" element.

The `<any>` and `<anyAttribute>` elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSD The <anyAttribute> Element

The <anyAttribute> element enables us to extend the XML document with attributes not specified by the schema!

The <anyAttribute> Element

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

The following example is a fragment from an XML schema called "family.xsd". It shows a declaration for the "person" element. By using the <anyAttribute> element we can add any number of attributes to the "person" 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>
```

Now we want to extend the "person" element with a "eyecolor" attribute. In this case we can do so, even if the author of the schema above never declared any "eyecolor" attribute.

Look at this schema file, called "attribute.xsd":

```

<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema"
targetNamespace="https://www.w3schools.com"
xmlns="https://www.w3schools.com"
elementFormDefault="qualified">

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

</xss:schema>

```

The XML file below (called "Myfamily.xml"), uses components from two different schemas; "family.xsd" and "attribute.xsd":

```

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

<persons xmlns="http://www.microsoft.com"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:SchemaLocation="http://www.microsoft.com family.xsd
  https://www.w3schools.com attribute.xsd">

  <person eyecolor="green">
    <firstname>Hege</firstname>
    <lastname>Rfsnes</lastname>
  </person>

  <person eyecolor="blue">
    <firstname>Stale</firstname>
    <lastname>Rfsnes</lastname>
  </person>

</persons>

```

The XML file above is valid because the schema "family.xsd" allows us to add an attribute to the "person" element.

The <any> and <anyAttribute> elements are used to make EXTENSIBLE documents! They allow documents to contain additional elements that are not declared in the main XML schema.

XSD Element Substitution

With XML Schemas, one element can substitute another element.

Element Substitution

Let's say that we have users from two different countries: England and Norway. We would like the ability to let the user choose whether he or she would like to use the Norwegian element names or the English element names in the XML document.

To solve this problem, we could define a **substitutionGroup** in the XML schema. First, we declare a head element and then we declare the other elements which state that they are substitutable for the head element.

```
<xs:element name="name" type="xs:string"/>
<xs:element name="navn" substitutionGroup="name"/>
```

In the example above, the "name" element is the head element and the "navn" element is substitutable for "name".

Look at this fragment of an XML schema:

```
<xs:element name="name" type="xs:string"/>
<xs:element name="navn" substitutionGroup="name"/>

<xs:complexType name="custinfo">
  <xs:sequence>
    <xs:element ref="name"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="customer" type="custinfo"/>
<xs:element name="kunde" substitutionGroup="customer"/>
```

A valid XML document (according to the schema above) could look like this:

```
<customer>
  <name>John Smith</name>
</customer>
```

```
<kunde>
  <navn>John Smith</navn>
</kunde>
```

Blocking Element Substitution

To prevent other elements from substituting with a specified element, use the `block` attribute:

```
<xs:element name="name" type="xs:string" block="substitution"/>
```

Look at this fragment of an XML schema:

```
<xs:element name="name" type="xs:string" block="substitution"/>
<xs:element name="navn" substitutionGroup="name"/>

<xs:complexType name="custinfo">
  <xs:sequence>
    <xs:element ref="name"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="customer" type="custinfo" block="substitution"/>
<xs:element name="kunde" substitutionGroup="customer"/>
```

A valid XML document (according to the schema above) looks like this:

```
<customer>
  <name>John Smith</name>
</customer>
```

BUT THIS IS NO LONGER VALID:

```
<kunde>
  <navn>John Smith</navn>
</kunde>
```

Using substitutionGroup

The type of the substitutable elements must be the same as, or derived from, the type of the head element. If the type of the substitutable element is the same as the type of the head element you will not have to specify the type of the substitutable element.

Note that all elements in the substitutionGroup (the head element and the substitutable elements) must be declared as global elements, otherwise it will not work!

What are Global Elements?

Global elements are elements that are immediate children of the "schema" element! Local elements are elements nested within other elements.

XSD String Data Types

String data types are used for values that contains character strings.

String Data Type

The string data type can contain characters, line feeds, carriage returns, and tab characters. The following is an example of a string declaration in a schema:

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

An element in your document might look like this:

```
<customer>John Smith</customer>
```

Or it might look like this:

```
<customer> John Smith </customer>
```

Note: The XML processor will not modify the value if you use the string data type.

NormalizedString Data Type

The normalizedString data type is derived from the String data type.

The normalizedString data type also contains characters, but the XML processor will remove line feeds, carriage returns, and tab characters.

The following is an example of a normalizedString declaration in a schema:

```
<xs:element name="customer" type="xs:normalizedString"/>
```

An element in your document might look like this:

```
<customer>John Smith</customer>
```

Or it might look like this:

```
<customer>      John Smith      </customer>
```

Note: In the example above the XML processor will replace the tabs with spaces.

Token Data Type

The token data type is also derived from the String data type.

The token data type also contains characters, but the XML processor will remove line feeds, carriage returns, tabs, leading and trailing spaces, and multiple spaces.

The following is an example of a token declaration in a schema:

```
<xss:element name="customer" type="xs:token"/>
```

An element in your document might look like this:

```
<customer>John Smith</customer>
```

Or it might look like this:

```
<customer>      John Smith      </customer>
```

Note: In the example above the XML processor will remove the tabs.

String Data Types

Note that all of the data types below derive from the String data type (except for string itself)!

Name	Description
ENTITIES	
ENTITY	
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)
IDREFS	
language	A string that contains a valid language id
Name	A string that contains a valid XML name
NCName	
NMTOKEN	A string that represents the NMTOKEN attribute in XML (only used with schema attributes)
NMTOKENS	
normalizedString	A string that does not contain line feeds, carriage returns, or tabs
QName	
string	A string
token	A string that does not contain line feeds, carriage returns, tabs, leading or trailing spaces, or multiple spaces

Restrictions on String Data Types

Restrictions that can be used with String data types:

- enumeration
- length
- maxLength
- minLength
- pattern (NMTOKENS, IDREFS, and ENTITIES cannot use this constraint)
- whiteSpace

XSD Date and Time Data Types

Date and time data types are used for values that contain date and time.

Date Data Type

The date data type is used to specify a date.

The date is specified in the following form "YYYY-MM-DD" where:

- YYYY indicates the year
- MM indicates the month
- DD indicates the day

Note: All components are required!

The following is an example of a date declaration in a schema:

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

An element in your document might look like this:

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

Time Zones

To specify a time zone, you can either enter a date in UTC time by adding a "Z" behind the date - like this:

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

or you can specify an offset from the UTC time by adding a positive or negative time behind the date - like this:

```
<start>2002-09-24-06:00</start>
```

or

```
<start>2002-09-24+06:00</start>
```

Time Data Type

The time data type is used to specify a time.

The time is specified in the following form "hh:mm:ss" where:

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

Note: All components are required!

The following is an example of a time declaration in a schema:

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

An element in your document might look like this:

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

Or it might look like this:

```
<start>09:30:10.5</start>
```

Time Zones

To specify a time zone, you can either enter a time in UTC time by adding a "Z" behind the time - like this:

```
<start>09:30:10Z</start>
```

or you can specify an offset from the UTC time by adding a positive or negative time behind the time - like this:

```
<start>09:30:10-06:00</start>
```

or

```
<start>09:30:10+06:00</start>
```

DateTime Data Type

The dateTime data type is used to specify a date and a time.

The dateTime is specified in the following form "YYYY-MM-DDThh:mm:ss" where:

- YYYY indicates the year
- MM indicates the month
- DD indicates the day
- T indicates the start of the required time section
- hh indicates the hour
- mm indicates the minute
- ss indicates the second

Note: All components are required!

The following is an example of a dateTime declaration in a schema:

```
<xss:element name="startdate" type="xs:dateTime"/>
```

An element in your document might look like this:

```
<startdate>2002-05-30T09:00:00</startdate>
```

Or it might look like this:

```
<startdate>2002-05-30T09:30:10.5</startdate>
```

Time Zones

To specify a time zone, you can either enter a dateTime in UTC time by adding a "Z" behind the time - like this:

```
<startdate>2002-05-30T09:30:10Z</startdate>
```

or you can specify an offset from the UTC time by adding a positive or negative time behind the time - like this:

```
<startdate>2002-05-30T09:30:10-06:00</startdate>
```

or

```
<startdate>2002-05-30T09:30:10+06:00</startdate>
```

Duration Data Type

The duration data type is used to specify a time interval.

The time interval is specified in the following form "PnYnMnDTnHnMnS" where:

- P indicates the period (required)
- nY indicates the number of years
- nM indicates the number of months
- nD indicates the number of days
- T indicates the start of a time section (required if you are going to specify hours, minutes, or seconds)
- nH indicates the number of hours
- nM indicates the number of minutes
- nS indicates the number of seconds

The following is an example of a duration declaration in a schema:

```
<xss:element name="period" type="xss:duration"/>
```

An element in your document might look like this:

```
<period>P5Y</period>
```

The example above indicates a period of five years.

Or it might look like this:

```
<period>P5Y2M10D</period>
```

The example above indicates a period of five years, two months, and 10 days.
Or it might look like this:

```
<period>P5Y2M10DT15H</period>
```

The example above indicates a period of five years, two months, 10 days, and 15 hours.
Or it might look like this:

```
<period>PT15H</period>
```

The example above indicates a period of 15 hours.

Negative Duration

To specify a negative duration, enter a minus sign before the P:

```
<period>-P10D</period>
```

The example above indicates a period of minus 10 days.

Date and Time Data Types

Name	Description
date	Defines a date value
dateTime	Defines a date and time value
duration	Defines a time interval
gDay	Defines a part of a date - the day (DD)
gMonth	Defines a part of a date - the month (MM)
gMonthDay	Defines a part of a date - the month and day (MM-DD)
gYear	Defines a part of a date - the year (YYYY)
gYearMonth	Defines a part of a date - the year and month (YYYY-MM)
time	Defines a time value

Restrictions on Date Data Types

Restrictions that can be used with Date data types:

- enumeration
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- pattern
- whiteSpace

XSD Numeric Data Types

Decimal data types are used for numeric values.

Decimal Data Type

The decimal data type is used to specify a numeric value.

The following is an example of a decimal declaration in a schema:

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

An element in your document might look like this:

```
<prize>999.50</prize>
```

Or it might look like this:

```
<prize>+999.5450</prize>
```

Or it might look like this:

```
<prize>-999.5230</prize>
```

Or it might look like this:

```
<prize>0</prize>
```

Or it might look like this:

```
<prize>14</prize>
```

Integer Data Type

The integer data type is used to specify a numeric value without a fractional component.

The following is an example of an integer declaration in a schema:

```
<xss:element name="prize" type="xs:integer"/>
```

An element in your document might look like this:

```
<prize>999</prize>
```

Or it might look like this:

```
<prize>+999</prize>
```

Or it might look like this:

```
<prize>-999</prize>
```

Or it might look like this:

```
<prize>0</prize>
```

Numeric Data Types

Note that all of the data types below derive from the Decimal data type (except for decimal itself)!

Name	Description
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

Restrictions on Numeric Data Types

Restrictions that can be used with Numeric data types:

- enumeration
- fractionDigits
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- pattern
- totalDigits
- whiteSpace

XSD Miscellaneous Data Types

Other miscellaneous data types are boolean, base64Binary, hexBinary, float, double, anyURI, QName, and NOTATION.

Boolean Data Type

The boolean data type is used to specify a true or false value.

The following is an example of a boolean declaration in a schema:

```
<xs:attribute name="disabled" type="xs:boolean"/>
```

An element in your document might look like this:

```
<prize disabled="true">999</prize>
```

Note: Legal values for boolean are true, false, 1 (which indicates true), and 0 (which indicates false).

Binary Data Types

Binary data types are used to express binary-formatted data.

We have two binary data types:

- base64Binary (Base64-encoded binary data)
- hexBinary (hexadecimal-encoded binary data)

The following is an example of a hexBinary declaration in a schema:

```
<xs:element name="blobsrc" type="xs:hexBinary"/>
```

AnyURI Data Type

The anyURI data type is used to specify a URI.

The following is an example of an anyURI declaration in a schema:

```
<xs:attribute name="src" type="xs:anyURI"/>
```

```
<pic src="https://www.w3schools.com/images/smiley.gif" />
```

Note: If a URI has spaces, replace them with %20.

Miscellaneous Data Types

Name	Description
anyURI	
base64Binary	
boolean	
double	
float	
hexBinary	
NOTATION	
QName	

Restrictions on Miscellaneous Data Types

Restrictions that can be used with the other data types:

- enumeration (a Boolean data type cannot use this constraint)
- length (a Boolean data type cannot use this constraint)
- maxLength (a Boolean data type cannot use this constraint)
- minLength (a Boolean data type cannot use this constraint)
- pattern
- whiteSpace

XML Schema Reference

XSD Elements

Element	Explanation
<u>all</u>	Specifies that the child elements can appear in any order. Each child element can occur 0 or 1 time
<u>annotation</u>	Specifies the top-level element for schema comments
<u>any</u>	Enables the author to extend the XML document with elements not specified by the schema
<u>anyAttribute</u>	Enables the author to extend the XML document with attributes not specified by the schema
<u>appinfo</u>	Specifies information to be used by the application (must go inside annotation)
<u>attribute</u>	Defines an attribute
<u>attributeGroup</u>	Defines an attribute group to be used in complex type definitions
<u>choice</u>	Allows only one of the elements contained in the <choice> declaration to be present within the containing element
<u>complexContent</u>	Defines extensions or restrictions on a complex type that contains mixed content or elements only
<u>complexType</u>	Defines a complex type element
<u>documentation</u>	Defines text comments in a schema (must go inside annotation)
<u>element</u>	Defines an element
<u>extension</u>	Extends an existing simpleType or complexType element
<u>field</u>	Specifies an XPath expression that specifies the value used to define an identity constraint
<u>group</u>	Defines a group of elements to be used in complex type definitions
<u>import</u>	Adds multiple schemas with different target namespace to a document

Element	Explanation
<u>include</u>	Adds multiple schemas with the same target namespace to a document
<u>key</u>	Specifies an attribute or element value as a key (unique, non-nullable, and always present) within the containing element in an instance document
<u>keyref</u>	Specifies that an attribute or element value correspond to those of the specified key or unique element
<u>list</u>	Defines a simple type element as a list of values
<u>notation</u>	Describes the format of non-XML data within an XML document
<u>redefine</u>	Redefines simple and complex types, groups, and attribute groups from an external schema
<u>restriction</u>	Defines restrictions on a simpleType, simpleContent, or a complexContent
<u>schema</u>	Defines the root element of a schema
<u>selector</u>	Specifies an XPath expression that selects a set of elements for an identity constraint
<u>sequence</u>	Specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times
<u>simpleContent</u>	Contains extensions or restrictions on a text-only complex type or on a simple type as content and contains no elements
<u>simpleType</u>	Defines a simple type and specifies the constraints and information about the values of attributes or text-only elements
<u>union</u>	Defines a simple type as a collection (union) of values from specified simple data types
<u>unique</u>	Defines that an element or an attribute value must be unique within the scope

XSD Restrictions/Facets for Datatypes

Look at XSD Restrictions!

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 maximum number of digits allowed. Must be greater than zero
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

XML Schema all Element

[« Complete XML Schema Reference](#)

Definition and Usage

The all element specifies that the child elements can appear in any order and that each child element can occur zero or one time.

Element Information

- **Parent elements:** group, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<all  
id=ID  
maxOccurs=1  
minOccurs=0|1  
any attributes  
>  
  
(annotation?,element*)  
  
</all>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the all element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
maxOccurs	Optional. Specifies the maximum number of times the element can occur. The value must be 1.
minOccurs	Optional. Specifies the minimum number of times the element can occur. The value can be 0 or 1. Default value is 1
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

```
<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>
```

The example above indicates that the "firstname" and the "lastname" elements can appear in any order but both elements MUST occur once and only once!

Example 2

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

The example above indicates that the "firstname" and the "lastname" elements can appear in any order and each element CAN appear zero or one time!

[« Complete XML Schema Reference](#)

XML Schema annotation Element

[« Complete XML Schema Reference](#)

Definition and Usage

The annotation element is a top level element that specifies schema comments. The comments serve as inline documentation.

Element Information

- **Parent elements:** Any element

Syntax

```
<annotation  
id=ID  
any attributes  
>  
  
(appinfo|documentation)*  
  
</annotation>
```

(The * sign declares that the element can occur zero or more times inside the annotation element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

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

<xs:annotation>
  <xs:appinfo>W3Schools Note</xs:appinfo>
  <xs:documentation xml:lang="en">
    This Schema defines a W3Schools note!
  </xs:documentation>
</xs:annotation>

.

.

.

</xs:schema>
```

[« Complete XML Schema Reference](#)

XML Schema any Element

[« Complete XML Schema Reference](#)

Definition and Usage

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

Element Information

- **Parent elements:** choice, sequence

Syntax

```
<any
id=ID
maxOccurs=nonNegativeInteger|unbounded
minOccurs=nonNegativeInteger
namespace=namespace
processContents=lax|skip|strict
any attributes
>

(annotation?)

</any>
```

(The ? sign declares that the element can occur zero or one time inside the any element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
maxOccurs	Optional. Specifies the maximum number of times the any element can occur in the parent element. The value can be any number ≥ 0 , or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1
minOccurs	Optional. Specifies the minimum number of times the any element can occur in the parent element. The value can be any number ≥ 0 . Default value is 1
namespace	Optional. Specifies the namespaces containing the elements that can be used. Can be set to one of the following: <ul style="list-style-type: none"> • ##any - elements from any namespace is allowed (this is default) • ##other - elements from any namespace that is not the namespace of the parent element can be present • ##local - elements must come from no namespace • ##targetNamespace - elements from the namespace of the parent element can be present • List of {URI references of namespaces, ##targetNamespace, ##local} - elements from a space-delimited list of the namespaces can be present
processContents	Optional. Specifies how the XML processor should handle validation against the elements specified by this any element. Can be set to one of the following: <ul style="list-style-type: none"> • strict - the XML processor must obtain the schema for the required namespaces and validate the elements (this is default) • lax - same as strict but; if the schema cannot be obtained, no errors will occur • skip - The XML processor does not attempt to validate any elements from the specified namespaces
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a declaration for an element called "person". By using the <any> element the author can extend (after <lastname>) the content of "person" with any element:

```
<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>
```

[« Complete XML Schema Reference](#)

XML Schema anyAttribute Element

[« Complete XML Schema Reference](#)

Definition and Usage

The anyAttribute element enables the author to extend the XML document with attributes not specified by the schema.

Element Information

- **Parent elements:** complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent), attributeGroup

Syntax

```
<anyAttribute  
id=ID  
namespace=namespace  
processContents=lax|skip|strict  
any attributes  
>  
  
(annotation?)  
  
</anyAttribute>
```

(The ? sign declares that the element can occur zero or one time inside the anyAttribute element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
namespace	Optional. Specifies the namespaces containing the attributes that can be used. Can be set to one of the following: <ul style="list-style-type: none"> ##any - attributes from any namespace is allowed (this is default) ##other - attributes from any namespace that is not the namespace of the parent element can be present ##local - attributes must come from no namespace ##targetNamespace - attributes from the namespace of the parent element can be present List of {URI references of namespaces, ##targetNamespace, ##local} - attributes from a space-delimited list of the namespaces can be present
processContents	Optional. Specifies how the XML processor should handle validation against the elements specified by this any element. Can be set to one of the following: <ul style="list-style-type: none"> strict - the XML processor must obtain the schema for the required namespaces and validate the elements (this is default) lax - same as strict but; if the schema cannot be obtained, no errors will occur skip - The XML processor does not attempt to validate any elements from the specified namespaces
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a declaration for an element called "person". By using the <anyAttribute> element the author can add any number of attributes to the "person" 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>
```

[« Complete XML Schema Reference](#)

XML Schema appinfo Element

[« Complete XML Schema Reference](#)

Definition and Usage

The appinfo element specifies information to be used by the application. This element must go within an annotation element.

Element Information

- **Parent elements:** annotation

Syntax

```
<appinfo  
source=anyURL  
>  
  
Any well-formed XML content  
  
</appinfo>
```

Attribute Description

source	Optional. A URI reference that specifies the source of the application information
--------	--

Example 1

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

<xs:annotation>
  <xs:appinfo>W3Schools Note</xs:appinfo>
  <xs:documentation xml:lang="en">
    This Schema defines a W3Schools note!
  </xs:documentation>
</xs:annotation>

.

.

.

</xs:schema>
```

[« Complete XML Schema Reference](#)

XML Schema attribute Element

[« Complete XML Schema Reference](#)

Definition and Usage

The attribute element defines an attribute.

Element Information

- **Parent elements:** attributeGroup, schema, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<attribute
default=string
fixed=string
form=qualified|unqualified
id=ID
name=NCName
ref=QName
type=QName
use=optional|prohibited|required
any attributes
>

(annotation?,(simpleType?))

</attribute>
```

(The ? sign declares that the element can occur zero or one time inside the attribute element)

Attribute	Description
default	Optional. Specifies a default value for the attribute. Default and fixed attributes cannot both be present
fixed	Optional. Specifies a fixed value for the attribute. Default and fixed attributes cannot both be present
form	Optional. Specifies the form for the attribute. The default value is the value of the attributeFormDefault attribute of the element containing the attribute. Can be set to one of the following: <ul style="list-style-type: none"> • "qualified" - indicates that this attribute must be qualified with the namespace prefix and the no-colon-name (NCName) of the attribute • unqualified - indicates that this attribute is not required to be qualified with the namespace prefix and is matched against the (NCName) of the attribute
id	Optional. Specifies a unique ID for the element
name	Optional. Specifies the name of the attribute. Name and ref attributes cannot both be present
ref	Optional. Specifies a reference to a named attribute. Name and ref attributes cannot both be present. If ref is present, simpleType element, form, and type cannot be present
type	Optional. Specifies a built-in data type or a simple type. The type attribute can only be present when the content does not contain a simpleType element
use	Optional. Specifies how the attribute is used. Can be one of the following values: <ul style="list-style-type: none"> • optional - the attribute is optional (this is default) • prohibited - the attribute cannot be used • required - the attribute is required
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

```

<xs:attribute name="code">

  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z]" />
    </xs:restriction>
  </xs:simpleType>

</xs:attribute>

```

The example above indicates that the "code" attribute has a restriction. The only acceptable value is two of the uppercase letters from a to z.

Example 2

To declare an attribute using an existing attribute definition within a complex type, use the ref attribute:

```

<xs:attribute name="code">
  <xs:simpleType>
    <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z]" />
    </xs:restriction>
  </xs:simpleType>
</xs:attribute>

<xs:complexType name="someComplexType">
  <xs:attribute ref="code"/>
</xs:complexType>

```

Example 3

Attributes can have either 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 when no other value is specified. But unlike default values; if you specify another value than the fixed, the document is considered invalid. In the following example the fixed value is "EN":

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

Example 4

All attributes are optional by default. To explicitly specify that the attribute is optional, use the "use" attribute:

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

To make an attribute required:

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

[« Complete XML Schema Reference](#)

XML Schema attributeGroup Element

[« Complete XML Schema Reference](#)

Definition and Usage

The attributeGroup element is used to group a set of attribute declarations so that they can be incorporated as a group into complex type definitions.

Element Information

- **Parent elements:** attributeGroup, complexType, schema, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<attributeGroup  
id=ID  
name=NCName  
ref=QName  
any attributes  
>  
  
(annotation?),((attribute|attributeGroup)*,anyAttribute?))  
  
</attributeGroup>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the attributeGroup element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Optional. Specifies the name of the attribute group. Name and ref attributes cannot both be present
ref	Optional. Specifies a reference to a named attribute group. Name and ref attributes cannot both be present
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

```
<xs:attributeGroup name="personattr">
  <xs:attribute name="attr1" type="string"/>
  <xs:attribute name="attr2" type="integer"/>
</xs:attributeGroup>

<xs:complexType name="person">
  <xs:attributeGroup ref="personattr"/>
</xs:complexType>
```

The example above defines an attribute group named "personattr" which is used in a complex type named "person".

[« Complete XML Schema Reference](#)

XML Schema choice Element

[« Complete XML Schema Reference](#)

Definition and Usage

XML Schema choice element allows only one of the elements contained in the <choice> declaration to be present within the containing element.

Element Information

- **Parent elements:** group, choice, sequence, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<choice
  id=ID
  maxOccurs=nonNegativeInteger|unbounded
  minOccurs=nonNegativeInteger
  any attributes
>

(annotation?,(element|group|choice|sequence|any)*)

</choice>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the choice element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
maxOccurs	Optional. Specifies the maximum number of times the choice element can occur in the parent element. The value can be any number ≥ 0 , or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1
minOccurs	Optional. Specifies the minimum number of times the choice element can occur in the parent element. The value can be any number ≥ 0 . Default value is 1
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example

```

<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>

```

The example above defines an element named "person" which must contain either a "employee" element or a "member" element.

[« Complete XML Schema Reference](#)

XML Schema complexContent Element

[« Complete XML Schema Reference](#)

Definition and Usage

The complexContent element defines extensions or restrictions on a complex type that contains mixed content or elements only.

Element Information

- **Parent elements:** complexType

Syntax

```
<complexContent
id=ID
mixed=true|false
any attributes
>

(annotation?,(restriction|extension))

</complexContent>
```

(The ? sign declares that the element can occur zero or one time inside the complexContent element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
mixed	Optional. Specifies whether character data is allowed to appear between the child elements of this complexType element. Default is false
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example has a complex type, "fullpersoninfo", that derives from another complex type, "personinfo", by extending the inherited type with three additional elements (address, city and country):

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

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

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

In the example above the "employee" element must contain, in sequence, the following elements: "firstname", "lastname", "address", "city", and "country".

[« Complete XML Schema Reference](#)

XML Schema complexType Element

[« Complete XML Schema Reference](#)

Definition and Usage

The complexType element defines a complex type. A complex type element is an XML element that contains other elements and/or attributes.

Element Information

- **Parent elements:** element, redefine, schema

Syntax

```
<complexType
  id=ID
  name=NCName
  abstract=true|false
  mixed=true|false
  block=(#all|list of (extension|restriction))
  final=(#all|list of (extension|restriction))
  any attributes
>

(annotation?,(simpleContent|complexContent|((group|all|
choice|sequence)?,((attribute|attributeGroup)*,anyAttribute?)))))

</complexType>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the complexType element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Optional. Specifies a name for the element
abstract	Optional. Specifies whether the complex type can be used in an instance document. True indicates that an element cannot use this complex type directly but must use a complex type derived from this complex type. Default is false
mixed	Optional. Specifies whether character data is allowed to appear between the child elements of this complexType element. Default is false. If a simpleContent element is a child element, the mixed attribute is not allowed!
block	Optional. Prevents a complex type that has a specified type of derivation from being used in place of this complex type. This value can contain #all or a list that is a subset of extension or restriction: <ul style="list-style-type: none"> • extension - prevents complex types derived by extension • restriction - prevents complex types derived by restriction • #all - prevents all derived complex types
final	Optional. Prevents a specified type of derivation of this complex type element. Can contain #all or a list that is a subset of extension or restriction. <ul style="list-style-type: none"> • extension - prevents derivation by extension • restriction - prevents derivation by restriction • #all - prevents all derivation
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example has an element named "note" that is of a complex type:

```

<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>

```

Example 2

The following example has a complex type, "fullpersoninfo", that derives from another complex type, "personinfo", by extending the inherited type with three additional elements (address, city and country):

```

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

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

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

```

In the example above the "employee" element must contain, in sequence, the following elements: "firstname", "lastname", "address", "city", and "country".

[« Complete XML Schema Reference](#)

XML Schema documentation Element

[« Complete XML Schema Reference](#)

Definition and Usage

The documentation element is used to enter text comments in a schema. This element must go inside an annotation element.

Element Information

- **Parent elements:** annotation

Syntax

```
<documentation  
source=URI reference  
xml:lang=language>  
  
Any well-formed XML content  
  
</documentation>
```

Attribute	Description
source	Optional. Specifies the source of the application information
xml:lang	Optional. Specifies the language used in the contents

Example 1

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

<xs:annotation>
  <xs:appinfo>W3Schools Note</xs:appinfo>
  <xs:documentation xml:lang="en">
    This Schema defines a W3Schools note!
  </xs:documentation>
</xs:annotation>

.

.

.

</xs:schema>
```

[« Complete XML Schema Reference](#)

XML Schema element Element

[« Complete XML Schema Reference](#)

Definition and Usage

The element element defines an element.

Element Information

- **Parent elements:** schema, choice, all, sequence, group

Syntax

```
<element
  id=ID
  name=NCName
  ref=QName
  type=QName
  substitutionGroup=QName
  default=string
  fixed=string
  form=qualified|unqualified
  maxOccurs=nonNegativeInteger|unbounded
  minOccurs=nonNegativeInteger
  nillable=true|false
  abstract=true|false
  block=(#all|list of (extension|restriction))
  final=(#all|list of (extension|restriction))
  any attributes
>

annotation?,(simpleType|complexType)?,(unique|key|keyref)*

</element>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the element element)

Attribute	Description
-----------	-------------

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Optional. Specifies a name for the element. This attribute is required if the parent element is the schema element
ref	Optional. Refers to the name of another element. The ref attribute can include a namespace prefix. This attribute cannot be used if the parent element is the schema element
type	Optional. Specifies either the name of a built-in data type, or the name of a simpleType or complexType element
substitutionGroup	Optional. Specifies the name of an element that can be substituted with this element. This attribute cannot be used if the parent element is not the schema element
default	Optional. Specifies a default value for the element (can only be used if the element's content is a simple type or text only)
fixed	Optional. Specifies a fixed value for the element (can only be used if the element's content is a simple type or text only)
form	Optional. Specifies the form for the element. "unqualified" indicates that this element is not required to be qualified with the namespace prefix. "qualified" indicates that this element must be qualified with the namespace prefix. The default value is the value of the elementFormDefault attribute of the schema element. This attribute cannot be used if the parent element is the schema element
maxOccurs	Optional. Specifies the maximum number of times this element can occur in the parent element. The value can be any number ≥ 0 , or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1. This attribute cannot be used if the parent element is the schema element
minOccurs	Optional. Specifies the minimum number of times this element can occur in the parent element. The value can be any number ≥ 0 . Default value is 1. This attribute cannot be used if the parent element is the schema element
nillable	Optional. Specifies whether an explicit null value can be assigned to the element. True enables an instance of the element to have the null attribute set to true. The null attribute is defined as part of the XML Schema namespace for instances. Default is false

Attribute	Description
abstract	Optional. Specifies whether the element can be used in an instance document. True indicates that the element cannot appear in the instance document. Instead, another element whose substitutionGroup attribute contains the qualified name (QName) of this element must appear in this element's place. Default is false
block	Optional. Prevents an element with a specified type of derivation from being used in place of this element. This value can contain #all or a list that is a subset of extension, restriction, or equivClass: <ul style="list-style-type: none"> • extension - prevents elements derived by extension • restriction - prevents elements derived by restriction • substitution - prevents elements derived by substitution • #all - prevents all derived elements
final	Optional. Sets the default value of the final attribute on the element element. This attribute cannot be used if the parent element is not the schema element. This value can contain #all or a list that is a subset of extension or restriction: <ul style="list-style-type: none"> • extension - prevents elements derived by extension • restriction - prevents elements derived by restriction • #all - prevents all derived elements
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example is a schema with four simple elements named "fname", "lname", "age", and "dateborn", which are of type string, nonNegativeInteger, and date:

```

<?xml version="1.0"?>
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:element name="fname" type="xs:string"/>
<xs:element name="lname" type="xs:string"/>
<xs:element name="age" type="xs:nonNegativeInteger"/>
<xs:element name="dateborn" type="xs:date"/>

</xsschema>

```

Example 2

The following example is a schema with an element named "note" that is of a complex type. The "note" element contains four other simple elements; "to", "from", "heading", and "body":

```
<?xml version="1.0"?>
<xsschema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xselement name="note">
<xsccomplexType>
<xsssequence>
<xselement name="to" type="xs:string"/>
<xselement name="from" type="xs:string"/>
<xselement name="heading" type="xs:string"/>
<xselement name="body" type="xs:string"/>
</xsssequence>
</xsccomplexType>
</xselement>

</xsschema>
```

Example 3

This example is quite equal to Example 2, but here we have chosen to use the ref attribute to refer to the element names:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

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

<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:schema>
```

[« Complete XML Schema Reference](#)

XML Schema extension Element

[« Complete XML Schema Reference](#)

Definition and Usage

The extension element extends an existing simpleType or complexType element.

Element Information

- **Parent elements:** simpleContent, complexContent

Syntax

```
<extension  
id=ID  
base=QName  
any attributes  
>  
  
(annotation?,((group|all|choice|sequence)?,  
((attribute|attributeGroup)*,anyAttribute?)))  
  
</extension>
```

(The ? sign declares that the element can occur zero or one time, and the * sign declares that the element can occur zero or more times inside the extension element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
base	Required. Specifies the name of a built-in data type, a simpleType element, or a complexType element
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example extends an existing simpleType by adding an attribute:

```
<?xml version="1.0"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xss:simpleType name="size">
  <xss:restriction base="xss:string">
    <xss:enumeration value="small" />
    <xss:enumeration value="medium" />
    <xss:enumeration value="large" />
  </xss:restriction>
</xss:simpleType>

<xss:complexType name="jeans">
  <xss:simpleContent>
    <xss:extension base="size">
      <xss:attribute name="sex">
        <xss:simpleType>
          <xss:restriction base="xss:string">
            <xss:enumeration value="male" />
            <xss:enumeration value="female" />
          </xss:restriction>
        </xss:simpleType>
      </xss:attribute>
    </xss:extension>
  </xss:simpleContent>
</xss:complexType>

</xss:schema>
```

Example 2

The following example extends an existing complexType element by adding three elements:

```
<?xml version="1.0"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xss:element name="employee" type="fullpersoninfo"/>

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

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

</xss:schema>
```

[« Complete XML Schema Reference](#)

XML Schema field Element

[« Complete XML Schema Reference](#)

Definition and Usage

The field element specifies an XPath expression that specifies the value used to define an identity constraint.

Element Information

- **Parent elements:** key, keyref, unique

Syntax

```
<field  
id=ID  
xpath=XPath expression  
any attributes  
>  
  
(annotation?)  
  
</field>
```

(The ? sign declares that the element can occur zero or one time inside the field element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
xpath	Required. Identifies a single element or attribute whose content or value is used for the constraint
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a field element that specifies the "userID" attribute as the field to use for the identity constraint:

```
<xs:field xpath="@userID" />
```

◀ Complete XML Schema Reference

XML Schema group Element

[« Complete XML Schema Reference](#)

Definition and Usage

The group element is used to define a group of elements to be used in complex type definitions.

Element Information

- **Parent elements:** schema, choice, sequence, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<group  
id=ID  
name=NCName  
ref=QName  
maxOccurs=nonNegativeInteger|unbounded  
minOccurs=nonNegativeInteger  
any attributes  
>  
  
(annotation?,(all|choice|sequence)?)  
  
</group>
```

(The ? sign declares that the element can occur zero or one time inside the group element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Optional. Specifies a name for the group. This attribute is used only when the schema element is the parent of this group element. Name and ref attributes cannot both be present
ref	Optional. Refers to the name of another group. Name and ref attributes cannot both be present
maxOccurs	Optional. Specifies the maximum number of times the group element can occur in the parent element. The value can be any number ≥ 0 , or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1
minOccurs	Optional. Specifies the minimum number of times the group element can occur in the parent element. The value can be any number ≥ 0 . Default value is 1
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example defines a group containing a sequence of four elements and uses the group element in a complex type definition:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:group name="custGroup">
  <xs:sequence>
    <xs:element name="customer" type="xs:string"/>
    <xs:element name="orderdetails" type="xs:string"/>
    <xs:element name="billto" type="xs:string"/>
    <xs:element name="shipto" type="xs:string"/>
  </xs:sequence>
</xs:group>

<xs:element name="order" type="ordertype"/>

<xs:complexType name="ordertype">
  <xs:group ref="custGroup"/>
  <xs:attribute name="status" type="xs:string"/>
</xs:complexType>

</xs:schema>
```

[« Complete XML Schema Reference](#)

XML Schema import Element

[« Complete XML Schema Reference](#)

Definition and Usage

The import element is used to add multiple schemas with different target namespace to a document.

Element Information

- **Parent elements:** schema

Syntax

```
<import  
id=ID  
namespace=anyURI  
schemaLocation=anyURI  
any attributes  
>  
  
(annotation?)  
  
</import>
```

(The ? sign declares that the element can occur zero or one time inside the import element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
namespace	Optional. Specifies the URI of the namespace to import
schemaLocation	Optional. Specifies the URI to the schema for the imported namespace
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows how to import a namespace:

```
<?xml version="1.0"?>
<xss: schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xss:import namespace="https://www.w3schools.com/schema"/>

..
..
..

</xss: schema>
```

◀ Complete XML Schema Reference

XML Schema include Element

[« Complete XML Schema Reference](#)

Definition and Usage

The include element is used to add multiple schemas with the same target namespace to a document.

Element Information

- **Parent elements:** schema

Syntax

```
<include  
id=ID  
schemaLocation=anyURI  
any attributes  
>  
  
(annotation?)  
  
</include>
```

(The ? sign declares that the element can occur zero or one time inside the include element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
schemaLocation	Required. Specifies the URI to the schema to include in the target namespace of the containing schema
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

With included schemas, the included files must all reference the same target namespace. If the schema target namespace don't match, the include won't work:

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

<xs:include schemaLocation="https://www.w3schools.com/xml/customer.xsd"/>
<xs:include schemaLocation="https://www.w3schools.com/xml/company.xsd"/>

..
..
..
</xss:schema>
```

[« Complete XML Schema Reference](#)

XML Schema key Element

[« Complete XML Schema Reference](#)

Definition and Usage

The key element specifies an attribute or element value as a key (unique, non-nullable, and always present) within the containing element in an instance document.

The key element MUST contain the following (in order):

- one and only one selector element (contains an XPath expression that specifies the set of elements across which the values specified by field must be unique)
- one or more field elements (contains an XPath expression that specifies the values that must be unique for the set of elements specified by the selector element)

Element Information

- **Parent elements:** element

Syntax

```
<key  
id=ID  
name=NCName  
any attributes  
>  
  
(annotation?,(selector,field+))  
  
</key>
```

(The ? sign declares that the element can occur zero or one time, and the + sign declares that the element must occur one or more times inside the key element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Required. Specifies the name of the key element
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

XML Schema keyref Element

[« Complete XML Schema Reference](#)

Definition and Usage

The keyref element specifies that an attribute or element value correspond to those of the specified key or unique element.

The keyref element MUST contain the following (in order):

- one and only one selector element (contains an XPath expression that specifies the set of elements across which the values specified by field must be unique)
- one or more field elements (contains an XPath expression that specifies the values that must be unique for the set of elements specified by the selector element)

Element Information

- **Parent elements:** element

Syntax

```
<keyref
  id=ID
  name=NCName
  refer=QName
  any attributes
>

(annotation?,(selector,field+))

</keyref>
```

(The ? sign declares that the element can occur zero or one time, and the + sign declares that the element must occur one or more times inside the keyref element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Required. Specifies the name of the keyref element
refer	Required. Specifies the name of a key or unique element defined in this or another schema
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

[« Complete XML Schema Reference](#)

XML Schema list Element

[« Complete XML Schema Reference](#)

Definition and Usage

The list element defines a simple type element as a list of values of a specified data type.

Element Information

- **Parent elements:** simpleType

Syntax

```
<list  
id=ID  
itemType=QName  
any attributes  
>  
  
(annotation?,(simpleType?))  
  
</list>
```

(The ? sign declares that the element can occur zero or one time inside the list element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
itemType	Specifies the name of a built-in data type or simpleType element defined in this or another schema. This attribute is not allowed if the content contains a simpleType element, otherwise it is required
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a simple type that is a list of integers:

```

<?xml version="1.0"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xss:element name="intvalues" type="valuelist"/>

<xss:simpleType name="valuelist">
  <xss:list itemType="xss:integer"/>
</xss:simpleType>

</xss:schema>

```

The "intvalues" element in a document could look like this (notice that the list will have five list items):

```
<intvalues>100 34 56 -23 1567</intvalues>
```

Note: White space is treated as the list item separator!

Example 2

The following example shows a simple type that is a list of strings:

```

<?xml version="1.0"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xss:element name="stringvalues" type="valuelist"/>

<xss:simpleType name="valuelist">
  <xss:list itemType="xss:string"/>
</xss:simpleType>

</xss:schema>

```

The "stringvalues" element in a document could look like this (notice that the list will have four list items):

```
<stringvalues>I love XML Schema</stringvalues>
```

XML Schema notation Element

[« Complete XML Schema Reference](#)

Definition and Usage

The notation element describes the format of non-XML data within an XML document.

Element Information

- **Parent elements:** schema

Syntax

```
<notation  
id=ID  
name=NCName  
public=anyURI  
system=anyURI  
any attributes  
>  
  
(annotation?)  
  
</notation>
```

(The ? sign declares that the element can occur zero or one time inside the notation element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Required. Specifies a name for the element
public	Required. Specifies a URI corresponding to the public identifier
system	Optional. Specifies a URI corresponding to the system identifier
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a gif and a jpeg notation using a viewer application, view.exe:

```
<?xml version="1.0"?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">

<xss:notation name="gif" public="image/gif" system="view.exe"/>
<xss:notation name="jpeg" public="image/jpeg" system="view.exe"/>

<xss:element name="image">
  <xss:complexType>
    <xss:simpleContent>
      <xss:attribute name="type">
        <xss:simpleType>
          <xss:restriction base="xss:NOTATION">
            <xss:enumeration value="gif"/>
            <xss:enumeration value="jpeg"/>
          </xss:restriction>
        </xss:simpleType>
      </xss:attribute>
    </xss:simpleContent>
  </xss:complexType>
</xss:element>

</xss:schema>
```

The "image" element in a document could look like this:

```
<image type="gif"></image>
```

[« Complete XML Schema Reference](#)

XML Schema redefine Element

[« Complete XML Schema Reference](#)

Definition and Usage

The redefine element redefines simple and complex types, groups, and attribute groups from an external schema.

Element Information

- **Parent elements:** schema

Syntax

```
<redefine
  id=ID
  schemaLocation=anyURI
  any attributes
>

(annotation|(simpleType|complexType|group|attributeGroup))*

</redefine>
```

Attribute	Description
id	Optional. Specifies a unique ID for the element
schemaLocation	Required. A URI to the location of a schema document
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

The following example shows a schema, Myschema2.xsd, with elements specified by the Myschema1.xsd. The pname type is redefined. According to this schema, elements constrained by the pname type must end with a "country" element:

Myschema1.xsd:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:complexType name="pname">
  <xs:sequence>
    <xs:element name="firstname"/>
    <xs:element name="lastname"/>
  </xs:sequence>
</xs:complexType>

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

</xs:schema>
```

Myschema2.xsd:

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

<xs:redefine schemaLocation="Myschema1.xsd">
  <xs:complexType name="pname">
    <xs:complexContent>
      <xs:extension base="pname">
        <xs:sequence>
          <xs:element name="country"/>
        </xs:sequence>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
</xs:redefine>

<xs:element name="author" type="pname"/>

</xs:schema>
```

XML Schema restriction Element

[« Complete XML Schema Reference](#)

Definition and Usage

The restriction element defines restrictions on a simpleType, simpleContent, or complexContent definition.

Element Information

- **Parent elements:** simpleType, simpleContent, complexContent

Syntax

```
<restriction  
id=ID  
base=QName  
any attributes  
>
```

Content for simpleType:

```
(annotation?,(simpleType?,(minExclusive|minInclusive|  
maxExclusive|maxInclusive|totalDigits|fractionDigits|  
length|minLength|maxLength|enumeration|whiteSpace|pattern)*))
```

Content for simpleContent:

```
(annotation?,(simpleType?,(minExclusive |minInclusive|  
maxExclusive|maxInclusive|totalDigits|fractionDigits|  
length|minLength|maxLength|enumeration|whiteSpace|pattern)*?),  
((attribute|attributeGroup)*,anyAttribute?))
```

Content for complexContent:

```
(annotation?,(group|all|choice|sequence)?,  
((attribute|attributeGroup)*,anyAttribute?))
```

```
</restriction>
```

(The ? sign declares that the element can occur zero or one time inside the restriction element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
base	Required. Specifies the name of a built-in data type, simpleType element, or complexType element defined in this schema or another schema
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

This example defines an element called "age" with a restriction. The value of age can NOT be lower than 0 or greater than 100:

```
<xs:element name="age">    <xs:simpleType>      <xs:restriction>
base="xs:integer">          <xs:minInclusive value="0"/>      <xs:maxInclusive>
value="100"/>        </xs:restriction>  </xs:simpleType> </xs:element>
```

Example 2

This example also defines an element called "initials". The "initials" element is a simple type 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>
```

Example 3

This example defines an element called "password". The "password" element is a simple type 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"/>
      <xsmaxLength value="8"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>

```

Example 4

This example shows a complex type definition using restriction. The complex type "Norwegian_customer" is derived from a general customer complex type and its country element is fixed to "Norway":

```

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

<xs:complexType name="Norwegian_customer">
  <xs:complexContent>
    <xs:restriction base="customer">
      <xs:sequence>
        <xs:element name="firstname" type="xs:string"/>
        <xs:element name="lastname" type="xs:string"/>
        <xs:element name="country" type="xs:string" fixed="Norway"/>
      </xs:sequence>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>

```

XML Schema Element

[« Complete XML Schema Reference](#)

Definition and Usage

The schema element defines the root element of a schema.

Element Information

- **Parent elements:** NONE

Syntax

```
<schema
  id=ID
  attributeFormDefault=qualified|unqualified
  elementFormDefault=qualified|unqualified
  blockDefault=(#all|list of (extension|restriction|substitution))
  finalDefault=(#all|list of (extension|restriction|list|union))
  targetNamespace=anyURI
  version=token
  xmlns=anyURI
  any attributes
>

((include|import|redefine|annotation)*,(((simpleType|complexType|
group|attributeGroup)|element|attribute|notation),annotation*)*)

</schema>
```

Attribute	Description
id	Optional. Specifies a unique ID for the element
attributeFormDefault	Optional. The form for attributes declared in the target namespace of this schema. The value must be "qualified" or "unqualified". Default is "unqualified". "unqualified" indicates that attributes from the target namespace are not required to be qualified with the namespace prefix. "qualified" indicates that attributes from the target namespace must be qualified with the namespace prefix

Attribute	Description
elementFormDefault	<p>Optional. The form for elements declared in the target namespace of this schema. The value must be "qualified" or "unqualified". Default is "unqualified". "unqualified" indicates that elements from the target namespace are not required to be qualified with the namespace prefix. "qualified" indicates that elements from the target namespace must be qualified with the namespace prefix</p>
blockDefault	<p>Optional. Specifies the default value of the block attribute on element and complexType elements in the target namespace. The block attribute prevents a complex type (or element) that has a specified type of derivation from being used in place of this complex type. This value can contain #all or a list that is a subset of extension, restriction, or substitution:</p> <ul style="list-style-type: none"> • extension - prevents complex types derived by extension • restriction - prevents complex types derived by restriction • substitution - prevents substitution of elements • #all - prevents all derived complex types
finalDefault	<p>Optional. Specifies the default value of the final attribute on element, simpleType, and complexType elements in the target namespace. The final attribute prevents a specified type of derivation of an element, simpleType, or complexType element. For element and complexType elements, this value can contain #all or a list that is a subset of extension or restriction. For simpleType elements, this value can additionally contain list and union:</p> <ul style="list-style-type: none"> • extension - prevents derivation by extension • restriction - prevents derivation by restriction • list - prevents derivation by list • union - prevents derivation by union • #all - prevents all derivation
targetNamespace	Optional. A URI reference of the namespace of this schema
version	Optional. Specifies the version of the schema
xmlns	A URI reference that specifies one or more namespaces for use in this schema. If no prefix is assigned, the schema components of the namespace can be used with unqualified references
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">

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

</xs:schema>
```

Example 2

In this example, the schema components (element name, type) in the <http://www.w3.org/2001/XMLSchema> namespace are unqualified and those for <https://www.w3schools.com/w3schoolsschema> (mystring) are qualified with the wsc prefix:

```
<?xml version="1.0"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:wsc="https://www.w3schools.com/w3schoolsschema">

<element name="fname" type="wsc:mystring"/>

</schema>
```

[« Complete XML Schema Reference](#)

XML Schema selector Element

[« Complete XML Schema Reference](#)

Definition and Usage

The selector element specifies an XPath expression that selects a set of elements for an identity constraint (unique, key, and keyref elements).

Element Information

- **Parent elements:** key, keyref, unique

Syntax

```
<selector  
id=ID  
xpath=a subset of XPath expression  
any attributes  
>  
  
(annotation?)  
  
</selector>
```

(The ? sign declares that the element can occur zero or one time inside the selector element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
xpath	Required. Specifies an XPath expression, relative to the element being declared, that identifies the child elements to which the identity constraint applies
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

[« Complete XML Schema Reference](#)

XML Schema sequence Element

[« Complete XML Schema Reference](#)

Definition and Usage

The sequence element specifies that the child elements must appear in a sequence. Each child element can occur from 0 to any number of times.

Element Information

- **Parent elements:** group, choice, sequence, complexType, restriction (both simpleContent and complexContent), extension (both simpleContent and complexContent)

Syntax

```
<sequence
  id=ID
  maxOccurs=nonNegativeInteger|unbounded
  minOccurs=nonNegativeInteger
  any attributes
>

(annotation?,(element|group|choice|sequence|any)*)

</sequence>
```

(The ? sign declares that the element can occur zero or one time inside the sequence element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
maxOccurs	Optional. Specifies the maximum number of times the sequence element can occur in the parent element. The value can be any number ≥ 0 , or if you want to set no limit on the maximum number, use the value "unbounded". Default value is 1
minOccurs	Optional. Specifies the minimum number of times the sequence element can occur in the parent element. The value can be any number ≥ 0 . Default value is 1
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

This example shows a declaration for an element called "personinfo", which must contain the following five elements in order; "firstname", "lastname", "address", "city", and "country":

```

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

```

Example 2

This example shows a declaration for an element called "pets" that can have zero or more of the following elements, dog and cat, in the sequence element:

```
<xs:element name="pets">
  <xs:complexType>
    <xs:sequence minOccurs="0" maxOccurs="unbounded">
      <xs:element name="dog" type="xs:string"/>
      <xs:element name="cat" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

◀ Complete XML Schema Reference

XML Schema simpleContent Element

[« Complete XML Schema Reference](#)

Definition and Usage

The simpleContent element contains extensions or restrictions on a text-only complex type or on a simple type as content and contains no elements.

Element Information

- **Parent elements:** complexType

Syntax

```
<simpleContent  
id=ID  
any attributes  
>  
  
(annotation?,(restriction|extension))  
  
</simpleContent>
```

(The ? sign declares that the element can occur zero or one time inside the simpleContent element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

Here is an example of an XML element (`<shoesize>`) that contains text-only:

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

The following example declares a complexType, "shoesize", with its content defined as a integer data type and with a country attribute:

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

[« Complete XML Schema Reference](#)

XML Schema simpleType Element

[« Complete XML Schema Reference](#)

Definition and Usage

The simpleType element defines a simple type and specifies the constraints and information about the values of attributes or text-only elements.

Element Information

- **Parent elements:** attribute, element, list, restriction, schema, union

Syntax

```
<simpleType  
id=ID  
name=NCName  
any attributes  
>  
  
(annotation?,(restriction|list|union))  
  
</simpleType>
```

(The ? sign declares that the element can occur zero or one time inside the simpleType element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Specifies a name for the element. This attribute is required if the simpleType element is a child of the schema element, otherwise it is not allowed
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

Example 1

This example defines an element called "age" that is a simple type with a restriction. The value of age can NOT be lower than 0 or greater than 100:

```
<xs:element name="age">
  <xs:simpleType>
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="100"/>
    </xs:restriction>
  </xs:simpleType>
</xs:element>
```

◀ Complete XML Schema Reference

XML Schema union Element

[« Complete XML Schema Reference](#)

Definition and Usage

The union element defines a simple type as a collection (union) of values from specified simple data types.

Element Information

- **Parent elements:** simpleType

Syntax

```
<union
  id=ID
  memberTypes="list of QNames"
  any attributes
>

(annotation?,(simpleType*))

</union>
```

(The ? sign declares that the element can occur zero or one time inside the union element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
memberTypes	Optional. Specifies a list of built-in data types or simpleType elements defined in a schema
any attributes	Optional. Specifies any other attributes with non-schema namespace

Example 1

This example shows a simple type that is a union of two simple types:

```
<xs:element name="jeans_size">
  <xs:simpleType>
    <xs:union memberTypes="sizebyno sizebystring" />
  </xs:simpleType>
</xs:element>

<xs:simpleType name="sizebyno">
  <xs:restriction base="xs:positiveInteger">
    <xs:maxInclusive value="42"/>
  </xs:restriction>
</xs:simpleType>

<xs:simpleType name="sizebystring">
  <xs:restriction base="xs:string">
    <xs:enumeration value="small"/>
    <xs:enumeration value="medium"/>
    <xs:enumeration value="large"/>
  </xs:restriction>
</xs:simpleType>
```

[« Complete XML Schema Reference](#)

XML Schema unique Element

[« Complete XML Schema Reference](#)

Definition and Usage

The unique element defines that an element or an attribute value must be unique within the scope.

The unique element MUST contain the following (in order):

- one and only one selector element (contains an XPath expression that specifies the set of elements across which the values specified by field must be unique)
- one or more field elements (contains an XPath expression that specifies the values that must be unique for the set of elements specified by the selector element)

Element Information

- **Parent elements:** element

Syntax

```
<unique
  id=ID
  name=NCName
  any attributes
>

(annotation?,(selector,field+))

</unique>
```

(The ? sign declares that the element can occur zero or one time inside the unique element)

Attribute	Description
id	Optional. Specifies a unique ID for the element
name	Required. Specifies a name for the element
<i>any attributes</i>	Optional. Specifies any other attributes with non-schema namespace

XML Web Services

Web services are web application components.

Web services can be published, found, and used on the Web.

This tutorial introduces WSDL, SOAP, RDF, and RSS.

WSDL

- WSDL stands for Web Services Description Language
- WSDL is an XML-based language for describing Web services.
- WSDL is a W3C recommendation

SOAP

- SOAP stands for Simple Object Access Protocol
- SOAP is an XML based protocol for accessing Web Services.
- SOAP is based on XML
- SOAP is a W3C recommendation

RDF

- RDF stands for Resource Description Framework
- RDF is a framework for describing resources on the web
- RDF is written in XML
- RDF is a W3C Recommendation

RSS

- RSS stands for Really Simple Syndication
- RSS allows you to syndicate your site content
- RSS defines an easy way to share and view headlines and content
- RSS files can be automatically updated
- RSS allows personalized views for different sites
- RSS is written in XML

What You Should Already Know

Before you study web services you should have a basic understanding of XML and XML Namespaces.

If you want to study these subjects first, please read our [XML Tutorial](#).

Web Services

- Web services are application components
- Web services communicate using open protocols
- Web services are self-contained and self-describing
- Web services can be discovered using UDDI
- Web services can be used by other applications
- HTTP and XML is the basis for Web services

Interoperability has Highest Priority

When all major platforms could access the Web using Web browsers, different platforms couldn't interact. For these platforms to work together, Web-applications were developed. Web-applications are simply applications that run on the web. These are built around the Web browser standards and can be used by any browser on any platform.

Web Services take Web-applications to the Next Level

By using Web services, your application can publish its function or message to the rest of the world.

Web services use XML to code and to decode data, and SOAP to transport it (using open protocols).

With Web services, your accounting department's Win 2k server's billing system can connect with your IT supplier's UNIX server.

Web Services have Two Types of Uses

Reusable application-components.

There are things applications need very often. So why make these over and over again?

Web services can offer application-components like: currency conversion, weather reports, or even language translation as services.

Connect existing software.

Web services can help to solve the interoperability problem by giving different applications a way to link their data.

With Web services you can exchange data between different applications and different platforms.

Any application can have a Web Service component.

Web Services can be created regardless of programming language.

A Web Service Example

In the following example we will use ASP.NET to create a simple Web Service that converts the temperature from Fahrenheit to Celsius, and vice versa:

```
<%@ WebService Language="VBScript" Class="TempConvert" %>

Imports System
Imports System.Web.Services

Public Class TempConvert :Inherits WebService

<WebMethod()> Public Function FahrenheitToCelsius
(ByVal Fahrenheit As String) As String
    dim fahr
    fahr=trim(replace(Fahrenheit,",",".""))
    if fahr="" or IsNumeric(fahr)=false then return "Error"
    return (((fahr) - 32) / 9) * 5
end function

<WebMethod()> Public Function CelsiusToFahrenheit
(ByVal Celsius As String) As String
    dim cel
    cel=trim(replace(Celsius,",",".""))
    if cel="" or IsNumeric(cel)=false then return "Error"
    return (((cel) * 9) / 5) + 32
end function

end class
```

This document is saved as an .asmx file. This is the ASP.NET file extension for XML Web Services.

Example Explained

Note: To run this example, you will need a .NET server.

The first line in the example states that this is a Web Service, written in VBScript, and has the class name "TempConvert":

```
<%@ WebService Language="VBScript" Class="TempConvert" %>
```

```
Imports System  
Imports System.Web.Services
```

The next line defines that the "TempConvert" class is a WebService class type:

```
Public Class TempConvert :Inherits WebService
```

The next steps are basic VB programming. This application has two functions. One to convert from Fahrenheit to Celsius, and one to convert from Celsius to Fahrenheit.

The only difference from a normal application is that this function is defined as a "WebMethod()".

Use "WebMethod()" to convert the functions in your application into web services:

```
<WebMethod()> Public Function FahrenheitToCelsius  
(ByVal Fahrenheit As String) As String  
    dim fahr  
    fahr=trim(replace(Fahrenheit,",","."))  
    if fahr="" or IsNumeric(fahr)=false then return "Error"  
    return (((fahr) - 32) / 9) * 5  
end function  
  
<WebMethod()> Public Function CelsiusToFahrenheit  
(ByVal Celsius As String) As String  
    dim cel  
    cel=trim(replace(Celsius,",","."))  
    if cel="" or IsNumeric(cel)=false then return "Error"  
    return (((cel) * 9) / 5) + 32  
end function
```

Then, end the class:

```
end class
```

Publish the .asmx file on a server with .NET support, and you will have your first working Web Service.

Put the Web Service on Your Web Site

Using a form and the HTTP POST method, you can put the web service on your site, like this:

Fahrenheit to Celsius:

Celsius to Fahrenheit:

How To Do It

Here is the code to add the Web Service to a web page:

```
<form action='tempconvert.asmx/FahrenheitToCelsius'  
method="post" target="_blank">  
<table>  
  <tr>  
    <td>Fahrenheit to Celsius:</td>  
    <td>  
      <input class="frmInput" type="text" size="30" name="Fahrenheit">  
    </td>  
  </tr>  
  <tr>  
    <td></td>  
    <td align="right">  
      <input type="submit" value="Submit" class="button">  
    </td>  
  </tr>  
</table>  
</form>  
  
<form action='tempconvert.asmx/CelsiusToFahrenheit'  
method="post" target="_blank">  
<table>  
  <tr>  
    <td>Celsius to Fahrenheit:</td>  
    <td>  
      <input class="frmInput" type="text" size="30" name="Celsius">  
    </td>  
  </tr>  
  <tr>  
    <td></td>  
    <td align="right">  
      <input type="submit" value="Submit" class="button">  
    </td>  
  </tr>  
</table>  
</form>
```

Substitute the "tempconvert.asmx" with the address of your web service like:

<http://www.example.com/xml/tempconvert.asmx>

XML WSDL

- WSDL stands for Web Services Description Language
- WSDL is used to describe web services
- WSDL is written in XML
- WSDL is a W3C recommendation from 26. June 2007

WSDL Documents

An WSDL document describes a web service. It specifies the location of the service, and the methods of the service, using these major elements:

Element	Description
<types>	Defines the (XML Schema) data types used by the web service
<message>	Defines the data elements for each operation
<portType>	Describes the operations that can be performed and the messages involved.
<binding>	Defines the protocol and data format for each port type

The main structure of a WSDL document looks like this:

```

<definitions>

<types>
    data type definitions.....
</types>

<message>
    definition of the data being communicated....
</message>

<portType>
    set of operations.....
</portType>

<binding>
    protocol and data format specification....
</binding>

</definitions>

```

WSDL Example

This is a simplified fraction of a WSDL document:

```

<message name="getTermRequest">
    <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
    <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
    <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
    </operation>
</portType>

```

In this example the **<portType>** element defines "glossaryTerms" as the name of a **port**, and "getTerm" as the name of an **operation**.

The "getTerm" operation has an **input message** called "getTermRequest" and an **output message** called "getTermResponse".

The **<message>** elements define the **parts** of each message and the associated data types.

The **<portType>** Element

The **<portType>** element defines **a web service**, the **operations** that can be performed, and the **messages** that are involved.

The request-response type is the most common operation type, but WSDL defines four types:

Type	Definition
One-way	The operation can receive a message but will not return a response
Request-response	The operation can receive a request and will return a response
Solicit-response	The operation can send a request and will wait for a response
Notification	The operation can send a message but will not wait for a response

WSDL One-Way Operation

A one-way operation example:

```
<message name="newTermValues">
  <part name="term" type="xs:string"/>
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="setTerm">
    <input name="newTerm" message="newTermValues"/>
  </operation>
</portType >
```

In the example above, the portType "glossaryTerms" defines a one-way operation called "setTerm".

The "setTerm" operation allows input of new glossary terms messages using a "newTermValues" message with the input parameters "term" and "value". However, no output is defined for the operation.

WSDL Request-Response Operation

A request-response operation example:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>
```

In the example above, the portType "glossaryTerms" defines a request-response operation called "getTerm".

The "getTerm" operation requires an input message called "getTermRequest" with a parameter called "term", and will return an output message called "getTermResponse" with a parameter called "value".

WSDL Binding to SOAP

WSDL bindings defines the message format and protocol details for a web service.
A request-response operation example:

```
<message name="getTermRequest">
  <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
  <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
  <operation name="getTerm">
    <input message="getTermRequest"/>
    <output message="getTermResponse"/>
  </operation>
</portType>

<binding type="glossaryTerms" name="b1">
  <soap:binding style="document"
    transport="http://schemas.xmlsoap.org/soap/http" />
  <operation>
    <soap:operation soapAction="http://example.com/getTerm"/>
    <input><soap:body use="literal"/></input>
    <output><soap:body use="literal"/></output>
  </operation>
</binding>
```

The **binding** element has two attributes - name and type.

The name attribute (you can use any name you want) defines the name of the binding, and the type attribute points to the port for the binding, in this case the "glossaryTerms" port.

The **soap:binding** element has two attributes - style and transport.

The style attribute can be "rpc" or "document". In this case we use document. The transport attribute defines the SOAP protocol to use. In this case we use HTTP.

The **operation** element defines each operation that the portType exposes.

For each operation the corresponding SOAP action has to be defined. You must also specify how the input and output are encoded. In this case we use "literal".

XML Soap

- SOAP stands for **S**imple **O**bject **A**ccess **P- SOAP is an application communication protocol
- SOAP is a format for sending and receiving messages
- SOAP is platform independent
- SOAP is based on XML
- SOAP is a W3C recommendation**

Why SOAP?

It is important for web applications to be able to communicate over the Internet. The best way to communicate between applications is over HTTP, because HTTP is supported by all Internet browsers and servers. SOAP was created to accomplish this. SOAP provides a way to communicate between applications running on different operating systems, with different technologies and programming languages.

SOAP Building Blocks

A SOAP message is an ordinary XML document containing the following elements:

- An Envelope element that identifies the XML document as a SOAP message
- A Header element that contains header information
- A Body element that contains call and response information
- A Fault element containing errors and status information

All the elements above are declared in the default namespace for the SOAP envelope:

<http://www.w3.org/2003/05/soap-envelope/>

and the default namespace for SOAP encoding and data types is:

<http://www.w3.org/2003/05/soap-encoding>

Syntax Rules

Here are some important syntax rules:

- A SOAP message MUST be encoded using XML
- A SOAP message MUST use the SOAP Envelope namespace
- A SOAP message MUST use the SOAP Encoding namespace
- A SOAP message must NOT contain a DTD reference
- A SOAP message must NOT contain XML Processing Instructions

Skeleton SOAP Message

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Header>
        ...
    </soap:Header>

    <soap:Body>
        ...
        <soap:Fault>
            ...
        </soap:Fault>
    </soap:Body>

</soap:Envelope>
```

The SOAP Envelope Element

The required SOAP Envelope element is the root element of a SOAP message. This element defines the XML document as a SOAP message.

Example

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    ...
    Message information goes here
    ...
</soap:Envelope>
```

The xmlns:soap Namespace

Notice the xmlns:soap namespace in the example above. It should always have the value of: "http://www.w3.org/2003/05/soap-envelope/".

The namespace defines the Envelope as a SOAP Envelope.

If a different namespace is used, the application generates an error and discards the message.

The encodingStyle Attribute

The encodingStyle attribute is used to define the data types used in the document. This attribute may appear on any SOAP element, and applies to the element's contents and all child elements.

A SOAP message has no default encoding.

Syntax

```
soap:encodingStyle="URI"
```

Example

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    ...
    Message information goes here
    ...
</soap:Envelope>
```

The SOAP Header Element

The optional SOAP Header element contains application-specific information (like authentication, payment, etc) about the SOAP message.

If the Header element is present, it must be the first child element of the Envelope element.

Note: All immediate child elements of the Header element must be namespace-qualified.

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Header>
        <m:Trans xmlns:m="https://www.w3schools.com/transaction/"
            soap:mustUnderstand="1">234
        </m:Trans>
    </soap:Header>
    ...
    ...
</soap:Envelope>
```

The example above contains a header with a "Trans" element, a "mustUnderstand" attribute with a value of 1, and a value of 234.

SOAP defines three attributes in the default namespace. These attributes are: mustUnderstand, actor, and encodingStyle.

The attributes defined in the SOAP Header defines how a recipient should process the SOAP message.

The mustUnderstand Attribute

The SOAP mustUnderstand attribute can be used to indicate whether a header entry is mandatory or optional for the recipient to process.

If you add `mustUnderstand="1"` to a child element of the Header element it indicates that the receiver processing the Header must recognize the element. If the receiver does not recognize the element it will fail when processing the Header.

Syntax

```
soap:mustUnderstand="0|1"
```

Example

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Header>
        <m:Trans xmlns:m="https://www.w3schools.com/transaction/"
            soap:mustUnderstand="1">234
        </m:Trans>
    </soap:Header>
    ...
    ...
</soap:Envelope>
```

The actor Attribute

A SOAP message may travel from a sender to a receiver by passing different endpoints along the message path. However, not all parts of a SOAP message may be intended for the ultimate endpoint, instead, it may be intended for one or more of the endpoints on the message path. The SOAP actor attribute is used to address the Header element to a specific endpoint.

Syntax

```
soap:actor="URI"
```

Example

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Header>
        <m:Trans xmlns:m="https://www.w3schools.com/transaction/"
            soap:actor="https://www.w3schools.com/code/">234
        </m:Trans>
    </soap:Header>
    ...
    ...
</soap:Envelope>
```

The encodingStyle Attribute

The `encodingStyle` attribute is used to define the data types used in the document. This attribute may appear on any SOAP element, and it will apply to that element's contents and all child elements.

A SOAP message has no default encoding.

Syntax

```
soap:encodingStyle="URI"
```

The SOAP Body Element

The required SOAP Body element contains the actual SOAP message intended for the ultimate endpoint of the message.

Immediate child elements of the SOAP Body element may be namespace-qualified.

Example

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Body>
        <m:GetPrice xmlns:m="https://www.w3schools.com/prices">
            <m:Item>Apples</m:Item>
        </m:GetPrice>
    </soap:Body>

</soap:Envelope>
```

The example above requests the price of apples. Note that the m:GetPrice and the Item elements above are application-specific elements. They are not a part of the SOAP namespace. A SOAP response could look something like this:

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

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Body>
        <m:GetPriceResponse xmlns:m="https://www.w3schools.com/prices">
            <m:Price>1.90</m:Price>
        </m:GetPriceResponse>
    </soap:Body>

</soap:Envelope>
```

The SOAP Fault Element

The optional SOAP Fault element is used to indicate error messages.

The SOAP Fault element holds errors and status information for a SOAP message.

If a Fault element is present, it must appear as a child element of the Body element. A Fault element can only appear once in a SOAP message.

The SOAP Fault element has the following sub elements:

Sub Element	Description
<faultcode>	A code for identifying the fault
<faultstring>	A human readable explanation of the fault
<faultactor>	Information about who caused the fault to happen
<detail>	Holds application specific error information related to the Body element

SOAP Fault Codes

The faultcode values defined below must be used in the faultcode element when describing faults:

Error	Description
VersionMismatch	Found an invalid namespace for the SOAP Envelope element
MustUnderstand	An immediate child element of the Header element, with the mustUnderstand attribute set to "1", was not understood
Client	The message was incorrectly formed or contained incorrect information
Server	There was a problem with the server so the message could not proceed

The HTTP Protocol

HTTP communicates over TCP/IP. An HTTP client connects to an HTTP server using TCP. After establishing a connection, the client can send an HTTP request message to the server:

```
POST /item HTTP/1.1
Host: 189.123.255.239
Content-Type: text/plain
Content-Length: 200
```

The server then processes the request and sends an HTTP response back to the client. The response contains a status code that indicates the status of the request:

```
200 OK
Content-Type: text/plain
Content-Length: 200
```

In the example above, the server returned a status code of 200. This is the standard success code for HTTP.

If the server could not decode the request, it could have returned something like this:

```
400 Bad Request
Content-Length: 0
```

SOAP Binding

The SOAP specification defines the structure of the SOAP messages, not how they are exchanged. This gap is filled by what is called "SOAP Bindings". SOAP bindings are mechanisms which allow SOAP messages to be effectively exchanged using a transport protocol.

Most SOAP implementations provide bindings for common transport protocols, such as HTTP or SMTP.

HTTP is synchronous and widely used. A SOAP HTTP request specifies at least two HTTP headers: Content-Type and Content-Length.

SMTP is asynchronous and is used in last resort or particular cases.

Java implementations of SOAP usually provide a specific binding for the JMS (Java Messaging System) protocol.

Content-Type

The Content-Type header for a SOAP request and response defines the MIME type for the message and the character encoding (optional) used for the XML body of the request or response.

Syntax

```
Content-Type: MIMETYPE; charset=character-encoding
```

Example

```
POST /item HTTP/1.1
Content-Type: application/soap+xml; charset=utf-8
```

Content-Length

The Content-Length header for a SOAP request and response specifies the number of bytes in the body of the request or response.

Syntax

```
Content-Length: bytes
```

Example

```
POST /item HTTP/1.1
Content-Type: application/soap+xml; charset=utf-8
Content-Length: 250
```

A SOAP Example

In the example below, a GetStockPrice request is sent to a server. The request has a StockName parameter, and a Price parameter that will be returned in the response. The namespace for the function is defined in "http://www.example.org/stock".

A SOAP request:

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
  xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

  <soap:Body xmlns:m="http://www.example.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>

</soap:Envelope>
```

The SOAP response:

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

    <soap:Body xmlns:m="http://www.example.org/stock">
        <m:GetStockPriceResponse>
            <m:Price>34.5</m:Price>
        </m:GetStockPriceResponse>
    </soap:Body>

</soap:Envelope>
```

XML RDF

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>
```

What is RDF?

- RDF stands for **R**esource **D**escription **F**ramework
- 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 - Examples of Use

- Describing properties for shopping items, such as price and availability
- Describing time schedules for web events
- Describing information about web pages (content, author, created and modified date)
- Describing content and rating for web pictures
- Describing content for search engines
- Describing electronic libraries

RDF is Designed to be Read by Computers

RDF was designed to provide a common way to describe information so it can be read and understood by computer applications.

RDF descriptions are not designed to be displayed on the web.

RDF is Written in XML

RDF documents are written in XML. The XML language used by RDF is called RDF/XML. By using XML, RDF information can easily be exchanged between different types of computers using different types of operating systems and application languages.

RDF and "The Semantic Web"

The RDF language is a part of the W3C's Semantic Web Activity. W3C's "Semantic Web Vision" is a future where:

- Web information has exact meaning
- Web information can be understood and processed by computers
- Computers can integrate information from the web

RDF uses Web identifiers (URIs) to identify resources.

RDF describes resources with properties and property values.

RDF Resource, Property, and Property Value

RDF identifies things using Web identifiers (URIs), and describes resources with properties and property values.

Explanation of Resource, Property, and Property value:

- A **Resource** is anything that can have a URI, such as "<https://www.w3schools.com/rdf>"
- A **Property** is a Resource that has a name, such as "author" or "homepage"
- A **Property value** is the value of a Property, such as "Jan Egil Refsnes" or "<https://www.w3schools.com>" (note that a property value can be another resource)

The following RDF document could describe the resource "<https://www.w3schools.com/rdf>":

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

<RDF>
  <Description about="https://www.w3schools.com/rdf">
    <author>Jan Egil Refsnes</author>
    <homepage>https://www.w3schools.com</homepage>
  </Description>
</RDF>
```

The example above is simplified. Namespaces are omitted.

RDF Statements

The combination of a Resource, a Property, and a Property value forms a **Statement** (known as the **subject, predicate and object** of a Statement).

Let's look at some example statements to get a better understanding:

Statement: "The author of <https://www.w3schools.com/rdf> is Jan Egil Refsnes".

- The subject of the statement above is: <https://www.w3schools.com/rdf>
- The predicate is: author
- The object is: Jan Egil Refsnes

Statement: "The homepage of <https://www.w3schools.com/rdf> is <https://www.w3schools.com>".

- The subject of the statement above is: <https://www.w3schools.com/rdf>
- The predicate is: homepage
- The object is: <https://www.w3schools.com>

RDF Example

Here are two records from a CD-list:

Title	Artist	Country	Company	Price	Year
Empire Burlesque	Bob Dylan	USA	Columbia	10.90	1985
Hide your heart	Bonnie Tyler	UK	CBS Records	9.90	1988

Below is a few lines from an RDF document:

```
<?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>Bob Dylan</cd:artist>
      <cd:country>USA</cd:country>
      <cd:company>Columbia</cd:company>
      <cd:price>10.90</cd:price>
      <cd:year>1985</cd:year>
    </rdf:Description>

    <rdf:Description
      rdf:about="http://www.recshop.fake/cd/Hide your heart">
        <cd:artist>Bonnie Tyler</cd:artist>
        <cd:country>UK</cd:country>
        <cd:company>CBS Records</cd:company>
        <cd:price>9.90</cd:price>
        <cd:year>1988</cd:year>
    </rdf:Description>

  .
  .
  .
</rdf:RDF>
```

The first line of the RDF document is the XML declaration. The XML declaration is followed by the root element of RDF documents: **<rdf:RDF>**.

The **xmlns:rdf** namespace, specifies that elements with the rdf prefix are from the namespace "http://www.w3.org/1999/02/22-rdf-syntax-ns#".

The **xmlns:cd** namespace, specifies that elements with the cd prefix are from the namespace "http://www.recshop.fake/cd#".

The **<rdf:Description>** element contains the description of the resource identified by the **rdf:about** attribute.

The elements: **<cd:artist>**, **<cd:country>**, **<cd:company>**, etc. are properties of the resource.

RDF Online Validator

W3C's RDF Validation Service is useful when learning RDF. Here you can experiment with RDF files.

The online RDF Validator parses your RDF document, checks your syntax, and generates tabular and graphical views of your RDF document.

Copy and paste the example below into W3C's RDF validator:

```
<?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.com</si:title>
    <si:author>Jan Egil Refsnes</si:author>
  </rdf:Description>
</rdf:RDF>
```

When you parse the example above, the result will look something like this.

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:

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

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#">
  ...Description goes here...
</rdf:RDF>
```

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:

```
<?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>Bob Dylan</cd:artist>
      <cd:country>USA</cd:country>
      <cd:company>Columbia</cd:company>
      <cd:price>10.90</cd:price>
      <cd:year>1985</cd:year>
    </rdf:Description>

  </rdf:RDF>
```

The elements, artist, country, company, price, and year, are defined in the <http://www.recshop.fake/cd#> namespace. This namespace is outside RDF (and not a part of RDF). RDF defines only the framework. The elements, artist, country, company, price, and year, must be defined by someone else (company, organization, person, etc).

Properties as Attributes

The property elements can also be defined as attributes (instead of elements):

```
<?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="Bob Dylan" cd:country="USA"
    cd:company="Columbia" cd:price="10.90"
    cd:year="1985" />

</rdf:RDF>
```

Properties as Resources

The property elements can also be defined as resources:

```
<?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>
```

In the example above, the property artist does not have a value, but a reference to a resource containing information about the artist.

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 Terms

In the examples above we have talked about "list of values" when describing the container elements. In RDF these "list of values" are called members.

So, we have the following:

- A container is a resource that contains things
- The contained things are called members (not list of values)

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".

Example

```
<?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>
```

RDF Schema and Application Classes

RDF Schema (RDFS) is an extension to RDF.

RDF describes resources with classes, properties, and values.

In addition, RDF also needs a way to define application-specific classes and properties.

Application-specific classes and properties must be defined using extensions to RDF.

One such extension is RDF Schema.

RDF Schema (RDFS)

RDF Schema does not provide actual application-specific classes and properties.

Instead RDF Schema provides the framework to describe application-specific classes and properties.

Classes in RDF Schema are much like classes in object oriented programming languages. This allows resources to be defined as instances of classes, and subclasses of classes.

RDFS Example

The following example demonstrates some of the RDFS facilities:

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

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

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

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

</rdf:RDF>
```

In the example above, the resource "horse" is a subclass of the class "animal".

Example Abbreviated

Since an RDFS class is an RDF resource we can abbreviate the example above by using rdfs:Class instead of rdf:Description, and drop the rdf:type information:

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

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

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

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

</rdf:RDF>
```

That's it!

The Dublin Core

The Dublin Core Metadata Initiative (DCMI) has created some predefined properties for describing documents.

RDF is metadata (data about data). RDF is used to describe information resources.

The Dublin Core is a set of predefined properties for describing documents.

The first Dublin Core properties were defined at the **Metadata Workshop in Dublin, Ohio** in 1995 and is currently maintained by the [Dublin Core Metadata Initiative](#).

Property	Definition
Contributor	An entity responsible for making contributions to the content of the resource
Coverage	The extent or scope of the content of the resource
Creator	An entity primarily responsible for making the content of the resource
Format	The physical or digital manifestation of the resource
Date	A date of an event in the lifecycle of the resource
Description	An account of the content of the resource
Identifier	An unambiguous reference to the resource within a given context
Language	A language of the intellectual content of the resource
Publisher	An entity responsible for making the resource available
Relation	A reference to a related resource
Rights	Information about rights held in and over the resource
Source	A Reference to a resource from which the present resource is derived
Subject	A topic of the content of the resource
Title	A name given to the resource
Type	The nature or genre of the content of the resource

A quick look at the table above indicates that RDF is ideal for representing Dublin Core information.

RDF Example

The following example demonstrates the use of some of the Dublin Core properties in an RDF document:

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

<rdf:RDF
  xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:dc= "http://purl.org/dc/elements/1.1/">

  <rdf:Description rdf:about="https://www.w3schools.com">
    <dc:description>W3Schools - Free tutorials</dc:description>
    <dc:publisher>Refsnes Data as</dc:publisher>
    <dc:date>2008-09-01</dc:date>
    <dc:type>Web Development</dc:type>
    <dc:format>text/html</dc:format>
    <dc:language>en</dc:language>
  </rdf:Description>

</rdf:RDF>
```

RDF Reference

The RDF namespace (xmlns:rdf) is: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>

The RDFS namespace (xmlns:rdfs) is: <http://www.w3.org/2000/01/rdf-schema#>

The recommended file extension for RDF files is **.rdf**. However, the extension **.xml** is often used to provide compatibility with old xml parsers.

The MIME type should be "**application/rdf+xml**".

RDFS / RDF Classes

Element	Class of	Subclass of
rdfs:Class	All classes	
rdfs:Datatype	Data types	Class
rdfs:Resource	All resources	Class
rdfs:Container	Containers	Resource
rdfs:Literal	Literal values (text and numbers)	Resource
rdf:List	Lists	Resource
rdf:Property	Properties	Resource
rdf:Statement	Statements	Resource
rdf:Alt	Containers of alternatives	Container
rdf:Bag	Unordered containers	Container
rdf:Seq	Ordered containers	Container
rdfs:ContainerMembershipProperty	Container membership properties	Property
rdf:XMLLiteral	XML literal values	Literal

RDFS / RDF Properties

Element	Domain	Range	Description
rdfs:domain	Property	Class	The domain of the resource
rdfs:range	Property	Class	The range of the resource
rdfs:subPropertyOf	Property	Property	The property is a sub property of a property
rdfs:subClassOf	Class	Class	The resource is a subclass of a class
rdfs:comment	Resource	Literal	The human readable description of the resource
rdfs:label	Resource	Literal	The human readable label (name) of the resource
rdfs:isDefinedBy	Resource	Resource	The definition of the resource
rdfs:seeAlso	Resource	Resource	The additional information about the resource
rdfs:member	Resource	Resource	The member of the resource
rdf:first	List	Resource	
rdf:rest	List	List	
rdf:subject	Statement	Resource	The subject of the resource in an RDF Statement
rdf:predicate	Statement	Resource	The predicate of the resource in an RDF Statement
rdf:object	Statement	Resource	The object of the resource in an RDF Statement
rdf:value	Resource	Resource	The property used for values
rdf:type	Resource	Class	The resource is an instance of a class

RDF Attributes

Attribute	Description
rdf:about	Defines the resource being described
rdf:Description	Container for the description of a resource
rdf:resource	Defines a resource to identify a property
rdf:datatype	Defines the data type of an element
rdf:ID	Defines the ID of an element
rdf:li	Defines a list
rdf:_n	Defines a node
rdf:nodeID	Defines the ID of an element node
rdf:parseType	Defines how an element should be parsed
rdf:RDF	The root of an RDF document
xml:base	Defines the XML base
xml:lang	Defines the language of the element content

XML RSS

With RSS it is possible to distribute up-to-date web content from one web site to thousands of other web sites around the world.
RSS allows fast browsing for news and updates.

RSS Document Example

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
    <title>W3Schools Home Page</title>
    <link>https://www.w3schools.com</link>
    <description>Free web building tutorials</description>
    <item>
        <title>RSS Tutorial</title>
        <link>https://www.w3schools.com/xml/xml_rss.asp</link>
        <description>New RSS tutorial on W3Schools</description>
    </item>
    <item>
        <title>XML Tutorial</title>
        <link>https://www.w3schools.com/xml</link>
        <description>New XML tutorial on W3Schools</description>
    </item>
</channel>

</rss>
```

What is RSS?

- RSS stands for **Really Simple Syndication**
- RSS allows you to syndicate your site content
- RSS defines an easy way to share and view headlines and content
- RSS files can be automatically updated
- RSS allows personalized views for different sites
- RSS is written in XML

Why use RSS?

RSS was designed to show selected data.

Without RSS, users will have to check your site daily for new updates. This may be too time-consuming for many users. With an RSS feed (RSS is often called a News feed or RSS feed) they can check your site faster using an RSS aggregator (a site or program that gathers and sorts out RSS feeds).

Since RSS data is small and fast-loading, it can easily be used with services like cell phones or PDA's.

Web-rings with similar information can easily share data on their web sites to make them better and more useful.

Who Should use RSS?

Webmasters who seldom update their web sites do not need RSS!

RSS is useful for web sites that are updated frequently, like:

- News sites - Lists news with title, date and descriptions
- Companies - Lists news and new products
- Calendars - Lists upcoming events and important days
- Site changes - Lists changed pages or new pages

Benefits of RSS

Here are some benefits of using RSS:

Choose your news With RSS you can choose to view the news you want, the news that interest you and are relevant to your work.

Remove unwanted information With RSS you can (finally) separate wanted information from unwanted information (spam)!

Increase your site traffic With RSS you can create your own news channel, and publish it to the Internet!

The History of RSS

- 1997 - Dave Winer at UserLand develops scriptingNews. RSS was born
- 1999 - Netscape develops RSS 0.90 (which supported scriptingNews)
- 1999 - Dave Winer develops scriptingNews 2.0b1 (which included RSS 0.90 features)
- 1999 - Netscape develops RSS 0.91 (which included most features from scriptingNews 2.0b1)
- 1999 - UserLand gets rid of scriptingNews and uses only RSS 0.91
- 1999 - Netscape stops their RSS development
- 2000 - UserLand releases the official RSS 0.91 specification
- 2000 - O'Reilly develops RSS 1.0. This format uses RDF and namespaces.
- 2000 - Dave Winer at UserLand develops RSS 0.92
- 2002 - Dave Winer develops RSS 2.0 after leaving UserLand
- 2003 - The official RSS 2.0 specification is released

RSS 1.0 is the only version that was developed using the W3C RDF (Resource Description Framework) standard.

The idea behind RDF was to help create a Semantic Web. However, this does not matter too much for ordinary users, but by using web standards it will be easier for persons and applications to exchange data.

What RSS Version Should I Use?

RSS 0.91 and RSS 2.0 are easier to understand than RSS 1.0. Our tutorial is based on RSS 2.0.

The syntax rules of RSS 2.0 are very simple and very strict.

Is RSS a Web Standard?

There is no official standard for RSS.

- About 50 % of all RSS feeds use RSS 0.91
- About 25 % use RSS 1.0
- The last 25 % is split between RSS 0.9x versions and RSS 2.0

How RSS Works

RSS is used to share content between websites.

With RSS, you register your content with companies called aggregators.

So, to be a part of it: First, create an RSS document and save it with an .xml extension. Then, upload the file to your website. Next, register with an RSS aggregator. Each day the aggregator searches the registered websites for RSS documents, verifies the link, and displays information about the feed so clients can link to documents that interests them.

Tip: Read our RSS Publishing chapter to view free RSS aggregation services.

RSS Example

RSS documents use a self-describing and simple syntax.

Here is a simple RSS document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
    <title>W3Schools Home Page</title>
    <link>https://www.w3schools.com</link>
    <description>Free web building tutorials</description>
    <item>
        <title>RSS Tutorial</title>
        <link>https://www.w3schools.com/xml/xml_rss.asp</link>
        <description>New RSS tutorial on W3Schools</description>
    </item>
    <item>
        <title>XML Tutorial</title>
        <link>https://www.w3schools.com/xml</link>
        <description>New XML tutorial on W3Schools</description>
    </item>
</channel>

</rss>
```

The first line in the document - the XML declaration - defines the XML version and the character encoding used in the document. In this case the document conforms to the 1.0 specification of XML and uses the UTF-8 character set.

The next line is the RSS declaration which identifies that this is an RSS document (in this case, RSS version 2.0).

The next line contains the `<channel>` element. This element is used to describe the RSS feed. The `<channel>` element has three required child elements:

- `<title>` - Defines the title of the channel (e.g. W3Schools Home Page)
- `<link>` - Defines the hyperlink to the channel (e.g. <https://www.w3schools.com>)
- `<description>` - Describes the channel (e.g. Free web building tutorials)

Each `<channel>` element can have one or more `<item>` elements.
Each `<item>` element defines an article or "story" in the RSS feed.
The `<item>` element has three required child elements:

- `<title>` - Defines the title of the item (e.g. RSS Tutorial)
- `<link>` - Defines the hyperlink to the item (e.g. https://www.w3schools.com/xml/xml_rss.asp)
- `<description>` - Describes the item (e.g. New RSS tutorial on W3Schools)

Finally, the two last lines close the `<channel>` and `<rss>` elements.

Comments in RSS

The syntax for writing comments in RSS is similar to that of HTML:

```
| <!-- This is an RSS comment -->
```

RSS is Written in XML

Because RSS is XML, keep in mind that:

- All elements must have a closing tag
- Elements are case sensitive
- Elements must be properly nested
- Attribute values must always be quoted

The RSS <channel> Element

The RSS <channel> element describes the RSS feed.

Look at the following RSS document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
    <title>W3Schools Home Page</title>
    <link>https://www.w3schools.com</link>
    <description>Free web building tutorials</description>
    <item>
        <title>RSS Tutorial</title>
        <link>https://www.w3schools.com/xml/xml_rss.asp</link>
        <description>New RSS tutorial on W3Schools</description>
    </item>
</channel>

</rss>
```

As mentioned before, the <channel> element describes the RSS feed, and has three required child elements:

- <title> - Defines the title of the channel (e.g. W3Schools Home Page)
- <link> - Defines the hyperlink to the channel (e.g. https://www.w3schools.com)
- <description> - Describes the channel (e.g. Free web building tutorials)

The <channel> element usually contains one or more <item> elements. Each <item> element defines an article or "story" in the RSS feed.

Furthermore, there are several optional child elements of <channel>. We will explain the most important ones below.

The <category> Element

The <category> child element is used to specify a category for your feed.

The <category> element makes it possible for RSS aggregators to group sites based on category.

The category for the RSS document above could be:

```
<category>Web development</category>
```

The <copyright> Element

The <copyright> child element notifies about copyrighted material.

The copyright for the RSS document above could be:

```
<copyright>2006 Refsnes Data as. All rights reserved.</copyright>
```

The <image> Element

The <image> child element allows an image to be displayed when aggregators present a feed.

The <image> element has three required child elements:

- <url> - Defines the URL to the image
- <title> - Defines the text to display if the image could not be shown
- <link> - Defines the hyperlink to the website that offers the channel

The image for the RSS document above could be:

```
<image>
  <url>https://www.w3schools.com/images/logo.gif</url>
  <title>W3Schools.com</title>
  <link>https://www.w3schools.com</link>
</image>
```

The <language> Element

The <language> child element is used to specify the language used to write your document.

The <language> element makes it possible for RSS aggregators to group sites based on language.

The language for the RSS document above could be:

```
<language>en-us</language>
```

The <item> Element

Each <item> element defines an article or "story" in an RSS feed.

Look at the following RSS document:

```
<?xml version="1.0" encoding="UTF-8" ?>
<rss version="2.0">

<channel>
    <title>W3Schools Home Page</title>
    <link>https://www.w3schools.com</link>
    <description>Free web building tutorials</description>
    <item>
        <title>RSS Tutorial</title>
        <link>https://www.w3schools.com/xml/xml_rss.asp</link>
        <description>New RSS tutorial on W3Schools</description>
    </item>
</channel>

</rss>
```

As mentioned before, each <item> element defines an article or "story" in the RSS feed.

The <item> element has three required child elements:

- <title> - Defines the title of the item (e.g. RSS Tutorial)
- <link> - Defines the hyperlink to the item (e.g. https://www.w3schools.com/xml/xml_rss.asp)
- <description> - Describes the item (e.g. New RSS tutorial on W3Schools)

Furthermore, there are several optional child elements of <item>. We will explain the most important ones below.

The <author> Element

The <author> child element is used to specify the e-mail address of the author of an item.

Note: To prevent spam e-mails, some developers do not include the <author> element.

The author of the item in the RSS document above could be:

```
<author>hege@refsnesdata.no</author>
```

The <comments> Element

The <comments> child element allows an item to link to comments about that item.
A comment of the item in the RSS document above could be:

```
<comments>https://www.w3schools.com/comments</comments>
```

The <enclosure> Element

The <enclosure> child element allows a media-file to be included with an item.
The <enclosure> element has three required attributes:

- url - Defines the URL to the media file
- length - Defines the length (in bytes) of the media file
- type - Defines the type of media file

A media-file included in the item in the RSS document above could be:

```
<enclosure url="https://www.w3schools.com/xml/rss.mp3"
length="5000" type="audio/mpeg" />
```

Get Your RSS Feed Up On The Web

Having an RSS document is not useful if other people cannot reach it.

Now it's time to get your RSS file up on the web. Here are the steps:

1. Name your RSS file. Notice that the file must have an .xml extension.
2. Validate your RSS file (a good validator can be found at <http://www.feedvalidator.org>).
3. Upload the RSS file to your web directory on your web server.
4. Copy the little orange **RSS** or **XML** button to your web directory.
5. Put the little orange "RSS" or "XML" button on the page where you will offer RSS to the world (e.g. on your home page). Then add a link to the button that links to the RSS file. The code will look something like this: ` .`
6. Submit your RSS feed to the RSS Feed Directories (you can Google or Yahoo for "RSS Feed Directories"). Note! The URL to your feed is not your home page, it is the URL to your feed, like "https://www.w3schools.com/xml/myfirstrss.xml". Here is a free RSS aggregation service:
 - [Newsisfree: Register here](#)

7. Register your feed with the major search engines:

- Google - <http://www.google.com/submityourcontent/website-owner>
- Bing - <http://www.bing.com/toolbox/submit-site-url>

8. Update your feed - After registering your RSS feed, you must make sure that you update your content frequently and that your RSS feed is constantly available.

Can I Manage my RSS Feed Myself?

The best way to ensure your RSS feed works the way you want, is to manage it yourself. However, this can be very time consuming, especially for pages with lot of updates. An alternative is to use a third-party automated RSS.

Automated RSS

For users who only need an RSS feed for their personal website, some of the most popular blog (Web Log) managers that offer built-in RSS services are:

- [Wordpress](#)
- [Blogger](#)
- [Radio](#)

RSS Readers

An RSS Reader is used to read RSS Feeds!

RSS readers are available for many different devices and OS.

There are a lot of different RSS readers. Some work as web services, and some are limited to windows (or Mac, PDA or UNIX):

- [QuiteRSS](#) - FREE! QuiteRSS is an open-source, cross-platform RSS/Atom news reader. It is versatile, and has a full set of options. QuietRSS has a rich set of social sharing options (Email/Twitter/Facebook/.../Pocket/Etc). QuietRSS is fast starting, and navigation is quick
- [FeedReader](#) - FREE! Simple, straightforward feed reader that easily handles large number of feeds. Has the essential options (not a lot of confusing ones). Does not require Java. Import or export OPML files. Option to open links in an external browser
- [RssReader](#) - FREE! Windows-based RSS reader. Supports RSS versions 0.9x, 1.0 and 2.0 and Atom 0.1, 0.2 and 0.3
- [blogbotrss](#) - FREE! An RSS reader plug-in for Internet Explorer and Microsoft Outlook

Tip: Most browsers have a built-in RSS Reader. If you go to a web site that offers RSS feeds, you will see an RSS icon  in the address bar, or toolbar. Click on the icon to view a list of the different feeds. Choose the feed you want to read.

I have an RSS Reader. Now what?

Click on the little **RSS** or **XML** button next to the RSS feed you want to read. Copy The URL you get in the browser window and paste it in your RSS reader.

RSS Examples

These examples demonstrate RSS using our RSS reader to view the results.

RSS <channel> Element

Setting the required channel elements (<title>, <link>, and <description>). Specify a category for the RSS Specify the program used to generate the RSS Add an image to the RSS Specify the language of the RSS Add an text input field to the RSS Specify days that the RSS should not be updated

RSS <item> element

Setting the required item elements (<title>, <link>, and <description>). Add a link to comments about the RSS item Add a media file to the RSS item Specify a unique identifier for the item Specify the publication date for the RSS item Specify a third-party source for the RSS item

RSS Reference

RSS <channel> Element

The links in the "Element" column point to more information about each specific element.

Element	Description
<u><category></u>	Optional. Defines one or more categories for the feed
<u><cloud></u>	Optional. Register processes to be notified immediately of updates of the feed
<u><copyright></u>	Optional. Notifies about copyrighted material
<u><description></u>	Required. Describes the channel
<u><docs></u>	Optional. Specifies a URL to the documentation of the format used in the feed
<u><generator></u>	Optional. Specifies the program used to generate the feed
<u><image></u>	Optional. Allows an image to be displayed when aggregators present a feed
<u><language></u>	Optional. Specifies the language the feed is written in
<u><lastBuildDate></u>	Optional. Defines the last-modified date of the content of the feed
<u><link></u>	Required. Defines the hyperlink to the channel
<u><managingEditor></u>	Optional. Defines the e-mail address to the editor of the content of the feed
<u><pubDate></u>	Optional. Defines the last publication date for the content of the feed
<u><rating></u>	Optional. The PICS rating of the feed
<u><skipDays></u>	Optional. Specifies the days where aggregators should skip updating the feed
<u><skipHours></u>	Optional. Specifies the hours where aggregators should skip updating the feed
<u><textInput></u>	Optional. Specifies a text input field that should be displayed with the feed
<u><title></u>	Required. Defines the title of the channel
<u><ttl></u>	Optional. Specifies the number of minutes the feed can stay cached before refreshing it from the source
<u><webMaster></u>	Optional. Defines the e-mail address to the webmaster of the feed

RSS <item> Element

Element	Description
<u><author></u>	Optional. Specifies the e-mail address to the author of the item
<u><category></u>	Optional. Defines one or more categories the item belongs to
<u><comments></u>	Optional. Allows an item to link to comments about that item
<u><description></u>	Required. Describes the item
<u><enclosure></u>	Optional. Allows a media file to be included with the item
<u><guid></u>	Optional. Defines a unique identifier for the item
<u><link></u>	Required. Defines the hyperlink to the item
<u><pubDate></u>	Optional. Defines the last-publication date for the item
<u><source></u>	Optional. Specifies a third-party source for the item
<u><title></u>	Required. Defines the title of the item

XML DOM Node Types

The DOM presents a document as a hierarchy of node objects.

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

[Display nodeName and nodeType of all elements](#)

[Display nodeName and nodeValue of all elements](#)

Node Types

The following table lists the different W3C node types, and which node types they may have as children:

Node Type	Description	Children
Document	Represents the entire document (the root-node of the DOM tree)	Element (max. one), ProcessingInstruction, Comment, DocumentType
DocumentFragment	Represents a "lightweight" Document object, which can hold a portion of a document	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
DocumentType	Provides an interface to the entities defined for the document	None
ProcessingInstruction	Represents a processing instruction	None
EntityReference	Represents an entity reference	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Element	Represents an element	Element, Text, Comment, ProcessingInstruction, CDATASection, EntityReference
Attr	Represents an attribute	Text, EntityReference
Text	Represents textual content in an element or attribute	None

Node Type	Description	Children
CDATASection	Represents a CDATA section in a document (text that will NOT be parsed by a parser)	None
Comment	Represents a comment	None
Entity	Represents an entity	Element, ProcessingInstruction, Comment, Text, CDATASection, EntityReference
Notation	Represents a notation declared in the DTD	None

Node Types - Return Values

The following table lists what the nodeName and the nodeValue properties will return for each node type:

NodeType	Named Constant
1	ELEMENT_NODE
2	ATTRIBUTE_NODE
3	TEXT_NODE
4	CDATA_SECTION_NODE
5	ENTITY_REFERENCE_NODE
6	ENTITY_NODE
7	PROCESSING_INSTRUCTION_NODE
8	COMMENT_NODE
9	DOCUMENT_NODE
10	DOCUMENT_TYPE_NODE
11	DOCUMENT_FRAGMENT_NODE
12	NOTATION_NODE

XML DOM - The Node Object

The Node Object

The Node object represents a single node in the document tree.

A node can be an element node, an attribute node, a text node, or any other of the node types explained in the [Node Types](#) chapter.

Notice that while all objects inherits the Node properties / methods for dealing with parents and children, not all objects can have parents or children. For example, Text nodes may not have children, and adding children to such nodes results in a DOM error.

Node Object Properties

Property	Description
attributes	A NamedNodeMap containing the attributes of this node (if it is an Element)
<u>baseURI</u>	Returns the absolute base URI of a node
<u>childNodes</u>	Returns a NodeList of child nodes for a node
<u>firstChild</u>	Returns the first child of a node
<u>lastChild</u>	Returns the last child of a node
<u>nextSibling</u>	Returns the node immediately following a node
<u>nodeName</u>	Returns the name of a node, depending on its type
<u>nodeType</u>	Returns the type of a node
<u>nodeValue</u>	Sets or returns the value of a node, depending on its type
<u>ownerDocument</u>	Returns the root element (document object) for a node
<u>parentNode</u>	Returns the parent node of a node
<u>prefix</u>	Sets or returns the namespace prefix of a node
<u>previousSibling</u>	Returns the node immediately before a node
<u>textContent</u>	Sets or returns the textual content of a node and its descendants

Node Object Methods

Method	Description
<u>appendChild()</u>	Appends a new child node to the end of the list of children of a node
<u>cloneNode()</u>	Clones a node
<u>compareDocumentPosition()</u>	Compares the placement of two nodes in the DOM hierarchy (document)
<u>getFeature(feature,version)</u>	Returns a DOM object which implements the specialized APIs of the specified feature and version
<u>getUserData(key)</u>	Returns the object associated to a key on a this node. The object must first have been set to this node by calling <code>setUserData</code> with the same key
<u>hasAttributes()</u>	Returns <code>true</code> if the specified node has any attributes, otherwise <code>false</code>
<u>hasChildNodes()</u>	Returns <code>true</code> if the specified node has any child nodes, otherwise <code>false</code>
<u>insertBefore()</u>	Inserts a new child node before an existing child node
<u>isDefaultNamespace(URI)</u>	Returns whether the specified namespaceURI is the default
<u>isEqualNode()</u>	Tests whether two nodes are equal
<u>isSameNode()</u>	Tests whether the two nodes are the same node
<u>lookupNamespaceURI()</u>	Returns the namespace URI associated with a given prefix
<u>lookupPrefix()</u>	Returns the prefix associated with a given namespace URI
<u>normalize()</u>	Puts all Text nodes underneath a node (including attribute nodes) into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates Text nodes, i.e., there are neither adjacent Text nodes nor empty Text nodes
<u>removeChild()</u>	Removes a specified child node from the current node
<u>replaceChild()</u>	Replaces a child node with a new node
<u>setUserData(key,data,handler)</u>	Associates an object to a key on a node

XML DOM - The NodeList Object

The NodeList object represents an ordered list of nodes.

The NodeList object

The nodes in the node list can be accessed through their index number (starting from 0).

The node list keeps itself up-to-date. If an element is deleted or added, in the node list or the XML document, the list is automatically updated.

Note: In a node list, the nodes are returned in the order in which they are specified in the XML document.

NodeList Object Properties

Property	Description
<u>length</u>	Returns the number of nodes in a node list

NodeList Object Methods

Method	Description
<u>item()</u>	Returns the node at the specified index in a node list

XML DOM - The NamedNodeMap Object

The NamedNodeMap object represents an unordered list of nodes.

The NamedNodeMap object

The nodes in the NamedNodeMap can be accessed through their name.

The NamedNodeMap keeps itself up-to-date. If an element is deleted or added, in the node list or the XML document, the list is automatically updated.

Note: In a named node map, the nodes are not returned in any particular order.

NamedNodeMap Object Properties

Property	Description
<u>length</u>	Returns the number of nodes in the list

NamedNodeMap Object Methods

Method	Description
<u>getNamedItem()</u>	Returns the node with the specific name
<u>getNamedItemNS()</u>	Returns the node with the specific name and namespace
<u>item()</u>	Returns the node at the specified index
<u>removeNamedItem()</u>	Removes the node with the specific name
<u>removeNamedItemNS()</u>	Removes the node with the specific name and namespace
<u>setNamedItem()</u>	Sets the specified node (by name)
<u>setNamedItemNS()</u>	Sets the specified node (by name and namespace)

XML DOM - The Document Object

The Document object represents the entire XML document.

The XML Document Object

The Document object is the root of an XML document tree, and gives us the primary access to the document's data.

Since element nodes, text nodes, comments, processing instructions, etc. cannot exist outside the document, the Document object also contains methods to create these objects. The Node objects have a ownerDocument property which associates them with the Document where they were created.

Document Object Properties

Property	Description
<u>childNodes</u>	Returns a NodeList of child nodes for the document
<u>doctype</u>	Returns the Document Type Declaration associated with the document
<u>documentElement</u>	Returns the root node of the document
<u>documentURI</u>	Sets or returns the location of the document
<u>domConfig</u>	Returns the configuration used when normalizeDocument() is invoked
<u>firstChild</u>	Returns the first child node of the document
<u>implementation</u>	Returns the DOMImplementation object that handles this document
<u>inputEncoding</u>	Returns the encoding used for the document (when parsing)
<u>lastChild</u>	Returns the last child node of the document
<u>nodeName</u>	Returns the name of a node (depending on its type)
<u>nodeType</u>	Returns the node type of a node
<u>nodeValue</u>	Sets or returns the value of a node (depending on its type)
<u>xmlEncoding</u>	Returns the XML encoding of the document

Property	Description
<u>xmlStandalone</u>	Sets or returns whether the document is standalone
<u>xmlVersion</u>	Sets or returns the XML version of the document

Document Object Methods

Method	Description
adoptNode(sourcenode)	Adopts a node from another document to this document, and returns the adopted node
createAttribute(name)	Creates an attribute node with the specified name, and returns the new Attr object
createAttributeNS(uri,name)	Creates an attribute node with the specified name and namespace, and returns the new Attr object
<u>createCDATASection()</u>	Creates a CDATA section node
<u>createComment()</u>	Creates a comment node
createDocumentFragment()	Creates an empty DocumentFragment object, and returns it
<u>createElement()</u>	Creates an element node
<u>createElementNS()</u>	Creates an element node with a specified namespace
createEntityReference(name)	Creates an EntityReference object, and returns it
createProcessingInstruction(target,data)	Creates a ProcessingInstruction object, and returns it
<u>createTextNode()</u>	Creates a text node
getElementById(id)	Returns the element that has an ID attribute with the given value. If no such element exists, it returns null
<u>getElementsByName()</u>	Returns a NodeList of all elements with a specified name
<u>getElementsByNameNS()</u>	Returns a NodeList of all elements with a specified name and namespace

Method	Description
importNode(nodetoimport,deep)	Imports a node from another document to this document. This method creates a new copy of the source node. If the deep parameter is set to true, it imports all children of the specified node. If set to false, it imports only the node itself. This method returns the imported node
normalizeDocument()	
<u>renameNode()</u>	Renames an element or attribute node

DocumentType Object Properties

Each document has a DOCTYPE attribute that whose value is either null or a DocumentType object.

The DocumentType object provides an interface to the entities defined for an XML document.

Property	Description
<u>name</u>	Returns the name of the DTD
publicId	Returns the public identifier of the DTD
<u>systemId</u>	Returns the system identifier of the external DTD

DocumentImplementation Object Methods

The DOMImplementation object performs operations that are independent of any particular instance of the document object model.

Method	Description
createDocument(nsURI, name, doctype)	Creates a new DOM Document object of the specified doctype
createDocumentType(name, pubId, systemId)	Creates an empty DocumentType node
getFeature(feature, version)	Returns an object which implements the APIs of the specified feature and version, if the is any
hasFeature(feature, version)	Checks whether the DOM implementation implements a specific feature and version

ProcessingInstruction Object Properties

The ProcessingInstruction object represents a processing instruction.

A processing instruction is used as a way to keep processor-specific information in the text of the XML document.

Property	Description
data	Sets or returns the content of this processing instruction
target	Returns the target of this processing instruction

XML DOM - The Element Object

The Element object

The Element object represents an element in an XML document. Elements may contain attributes, other elements, or text. If an element contains text, the text is represented in a text-node.

IMPORTANT! Text is always stored in text nodes. A common error in DOM processing is to navigate to an element node and expect it to contain the text. However, even the simplest element node has a text node under it. For example, in `<year>2005</year>`, there is an element node (year), and a text node under it, which contains the text (2005).

Because the Element object is also a Node, it inherits the Node object's properties and methods.

Element Object Properties

Property	Description
<u>attributes</u>	Returns a NamedNodeMap of attributes for the element
<u>baseURI</u>	Returns the absolute base URI of the element
<u>childNodes</u>	Returns a NodeList of child nodes for the element
<u>firstChild</u>	Returns the first child of the element
<u>lastChild</u>	Returns the last child of the element
<u>localName</u>	Returns the local part of the name of the element
<u>namespaceURI</u>	Returns the namespace URI of the element
<u>nextSibling</u>	Returns the node immediately following the element
<u>nodeName</u>	Returns the name of the node, depending on its type
<u>nodeType</u>	Returns the type of the node
<u>ownerDocument</u>	Returns the root element (document object) for an element
<u>parentNode</u>	Returns the parent node of the element
<u>prefix</u>	Sets or returns the namespace prefix of the element
<u>previousSibling</u>	Returns the node immediately before the element
<u>schemaTypeInfo</u>	Returns the type information associated with the element

Property	Description
<u>tagName</u>	Returns the name of the element
<u>textContent</u>	Sets or returns the text content of the element and its descendants

Element Object Methods

Method	Description
<u>appendChild()</u>	Adds a new child node to the end of the list of children of the node
<u>cloneNode()</u>	Clones a node
<u>compareDocumentPosition()</u>	Compares the document position of two nodes
<u>getAttribute()</u>	Returns the value of an attribute
<u>getAttributeNS()</u>	Returns the value of an attribute (with a namespace)
<u>getAttributeNode()</u>	Returns an attribute node as an Attribute object
<u>getAttributeNodeNS()</u>	Returns an attribute node (with a namespace) as an Attribute object
<u>getElementsByTagName()</u>	Returns a NodeList of matching element nodes, and their children
<u>getElementsByTagNameNS()</u>	Returns a NodeList of matching element nodes (with a namespace), and their children
<u>getFeature(feature,version)</u>	Returns a DOM object which implements the specialized APIs of the specified feature and version
<u>getUserData(key)</u>	Returns the object associated to a key on this node. The object must first have been set to this node by calling <u>setUserData</u> with the same key
<u>hasAttribute()</u>	Returns whether an element has any attributes matching a specified name
<u>hasAttributeNS()</u>	Returns whether an element has any attributes matching a specified name and namespace
<u>hasAttributes()</u>	Returns whether the element has any attributes
<u>hasChildNodes()</u>	Returns whether the element has any child nodes

Method	Description
<u>insertBefore()</u>	Inserts a new child node before an existing child node
<u>isDefaultNamespace(URI)</u>	Returns whether the specified namespaceURI is the default
<u>isEqualNode()</u>	Checks if two nodes are equal
<u>lookupNamespaceURI()</u>	Returns the namespace URI matching a specified prefix
<u>lookupPrefix()</u>	Returns the prefix matching a specified namespace URI
<u>normalize()</u>	Puts all text nodes underneath this element (including attributes) into a "normal" form where only structure (e.g., elements, comments, processing instructions, CDATA sections, and entity references) separates Text nodes, i.e., there are neither adjacent Text nodes nor empty Text nodes
<u>removeAttribute()</u>	Removes a specified attribute
<u>removeAttributeNS()</u>	Removes a specified attribute (with a namespace)
<u>removeAttributeNode()</u>	Removes a specified attribute node
<u>removeChild()</u>	Removes a child node
<u>replaceChild()</u>	Replaces a child node
<u>setUserData(key,data,handler)</u>	Associates an object to a key on the element
<u>setAttribute()</u>	Adds a new attribute
<u>setAttributeNS()</u>	Adds a new attribute (with a namespace)
<u>setAttributeNode()</u>	Adds a new attribute node
<u>setAttributeNodeNS(attrnode)</u>	Adds a new attribute node (with a namespace)
<u>setIdAttribute(name,isId)</u>	If the <code>isId</code> property of the <code>Attribute</code> object is true, this method declares the specified attribute to be a user-determined ID attribute
<u>setIdAttributeNS(uri,name,isId)</u>	If the <code>isId</code> property of the <code>Attribute</code> object is true, this method declares the specified attribute (with a namespace) to be a user-determined ID attribute
<u>setIdAttributeNode(idAttr,isId)</u>	If the <code>isId</code> property of the <code>Attribute</code> object is true, this method declares the specified attribute to be a user-determined ID attribute

XML DOM - The Attr Object

The Attr object

The Attr object represents an attribute of an Element object. The allowable values for attributes are usually defined in a DTD.

Because the Attr object is also a Node, it inherits the Node object's properties and methods. However, an attribute does not have a parent node and is not considered to be a child node of an element, and will return null for many of the Node properties.

Attr Object Properties

Property	Description
<u>baseURI</u>	Returns the absolute base URI of the attribute
<u>isId</u>	Returns true if the attribute is known to be of type ID, otherwise it returns false
<u>localName</u>	Returns the local part of the name of the attribute
<u>name</u>	Returns the name of the attribute
<u>namespaceURI</u>	Returns the namespace URI of the attribute
<u>nodeName</u>	Returns the name of the node, depending on its type
<u>nodeType</u>	Returns the type of the node
<u>nodeValue</u>	Sets or returns the value of the node, depending on its type
<u>ownerDocument</u>	Returns the root element (document object) for an attribute
<u>ownerElement</u>	Returns the element node the attribute is attached to
<u>prefix</u>	Sets or returns the namespace prefix of the attribute
<u>schemaTypeInfo</u>	Returns the type information associated with this attribute
<u>specified</u>	Returns true if the attribute value is set in the document, and false if it's a default value in a DTD/Schema.
<u>textContent</u>	Sets or returns the textual content of an attribute
<u>value</u>	Sets or returns the value of the attribute

XML DOM - The Text Object

The Text object

The Text object represents the textual content of an element or attribute.

Text Object Properties

Property	Description
<u>data</u>	Sets or returns the text of the element or attribute
isElementContentWhitespace	Returns true if the text node contains content whitespace, otherwise it returns false
<u>length</u>	Returns the length of the text of the element or attribute
<u>wholeText</u>	Returns all text of text nodes adjacent to this node, concatenated in document order

Text Object Methods

Method	Description
<u>appendData()</u>	Appends data to the node
<u>deleteData()</u>	Deletes data from the node
<u>insertData()</u>	Inserts data into the node
<u>replaceData()</u>	Replaces data in the node
<u>replaceWholeText(text)</u>	Replaces the text of this node and all adjacent text nodes with the specified text
<u>splitText()</u>	Splits this node into two nodes at the specified offset, and returns the new node that contains the text after the offset
<u>substringData()</u>	Extracts data from the node

XML DOM - The CDATASEction Object

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

The CDATASEction object

The CDATASEction object represents a CDATA section in a document.

A CDATA section contains text that will NOT be parsed by a parser. Tags inside a CDATA section will NOT be treated as markup and entities will not be expanded. The primary purpose is for including material such as XML fragments, without needing to escape all the delimiters.

The only delimiter that is recognized in a CDATA section is "]]>" - which indicates the end of the CDATA section. CDATA sections cannot be nested.

CDATASEction Object Properties

Property	Description
<u>data</u>	Sets or returns the text of this node
<u>length</u>	Returns the length of the CDATA section

CDATASEction Object Methods

Property	Description
<u>appendData()</u>	Appends data to the node
<u>deleteData()</u>	Deletes data from the node
<u>insertData()</u>	Inserts data into the node
<u>replaceData()</u>	Replaces data in the node
<u>splitText()</u>	Splits the CDATA node into two nodes
<u>substringData()</u>	Extracts data from the node

XML DOM - The Comment Object

Try it Yourself - Examples

The examples below use the XML file [books.xml](#).

The Comment object

The Comment object represents the content of comment nodes in a document.

Comment Object Properties

Property	Description
<u>data</u>	Sets or returns the text of this node
<u>length</u>	Returns the length of the text of this node

Comment Object Methods

Property	Description
<u>appendData()</u>	Appends data to the node
<u>deleteData()</u>	Deletes data from the node
<u>insertData()</u>	Inserts data into the node
<u>replaceData()</u>	Replaces data in the node
<u>substringData()</u>	Extracts data from the node

The XMLHttpRequest Object

With the XMLHttpRequest object you can update parts of a web page, without reloading the whole page.

Try it Yourself - Examples

[A simple XMLHttpRequest example](#) Create a simple XMLHttpRequest, and retrieve data from a TXT file.

[Retrieve header information with getAllResponseHeaders\(\)](#) Retrieve header information of a resource (file).

[Retrieve specific header information with getResponseHeader\(\)](#) Retrieve specific header information of a resource (file).

[Retrieve the content of an ASP file](#) How a web page can communicate with a web server while a user type characters in an input field.

[Retrieve content from a database](#) How a web page can fetch information from a database with the XMLHttpRequest object.

[Retrieve the content of an XML file](#) Create an XMLHttpRequest to retrieve data from an XML file and display the data in an HTML table.

The XMLHttpRequest Object

The XMLHttpRequest object is used to exchange data with a server behind the scenes.

The XMLHttpRequest object is **the 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

XMLHttpRequest Object Methods

Method	Description
abort()	Cancels the current request
getAllResponseHeaders()	Returns header information
getResponseHeader()	Returns specific header information
open(method,url,async,uname,pswd)	Specifies the type of request, the URL, if the request should be handled asynchronously or not, and other optional attributes of a request method: the type of request: GET or POST url: the location of the file on the server async: true (asynchronous) or false (synchronous)
send(string)	send(string) Sends the request off to the server. string: Only used for POST requests
setRequestHeader()	Adds a label/value pair to the header to be sent

XMLHttpRequest Object Properties

Property	Description
onreadystatechange	Stores a function (or the name of a function) to be called automatically each time the readyState property changes
readyState	Holds the status of the XMLHttpRequest. Changes from 0 to 4: 0: request not initialized 1: server connection established 2: request received 3: processing request 4: request finished and response is ready
responseText	Returns the response data as a string
responseXML	Returns the response data as XML data
status	Returns the status-number (e.g. "404" for "Not Found" or "200" for "OK")
statusText	Returns the status-text (e.g. "Not Found" or "OK")

XML DOM Parser Errors

XML Parser Error

When trying to open an XML document, a parser-error may occur.

If the parser encounters an error, it may load an XML document containing the error description.

The code example below tries to load an XML document that is not well-formed.

You can read more about well-formed XML in [XML Syntax](#).

Example

```
<html>
<body>

<p id="demo"></p>

<script>
var xhttp = new XMLHttpRequest();
xhttp.onreadystatechange = function() {
  if (xhttp.this == 4 && this.status == 200) {
    myFunction(this);
  }
};
xhttp.open("GET", "note_error.xml", true);
xhttp.send();

function myFunction(xml) {
  var parser, xmlDoc;
  parser = new DOMParser();
  xmlDoc = parser.parseFromString(xml.responseText,"text/xml");
  document.getElementById("demo").innerHTML =
  myLoop(xmlDoc.documentElement);
}

function myLoop(x) {
  var i, y, xLen, txt;
  txt = "";
  x = x.childNodes;
  xLen = x.length;
  for (i = 0; i < xLen ;i++) {
```

```
y = x[i];
if (y.nodeType != 3) {
    if (y.childNodes[0] != undefined) {
        txt += myLoop(y);
    }
} else {
    txt += y.nodeValue + "<br>";
}
}
return txt;
}
</script>

</body>
</html>
```

Look at the XML file: [note_error.xml](#)

XSLT Elements Reference

The XSLT elements from the W3C Recommendation (XSLT Version 1.0).

XSLT Elements

The links in the "Element" column point to attributes and more useful information about each specific element.

Element	Description
apply-imports	Applies a template rule from an imported style sheet
apply-templates	Applies a template rule to the current element or to the current element's child nodes
attribute	Adds an attribute
attribute-set	Defines a named set of attributes
call-template	Calls a named template
choose	Used in conjunction with <when> and <otherwise> to express multiple conditional tests
comment	Creates a comment node in the result tree
copy	Creates a copy of the current node (without child nodes and attributes)
copy-of	Creates a copy of the current node (with child nodes and attributes)
decimal-format	Defines the characters and symbols to be used when converting numbers into strings, with the format-number() function
element	Creates an element node in the output document
fallback	Specifies an alternate code to run if the processor does not support an XSLT element
for-each	Loops through each node in a specified node set
if	Contains a template that will be applied only if a specified condition is true
import	Imports the contents of one style sheet into another. Note: An imported style sheet has lower precedence than the importing style sheet

Element	Description
<u>include</u>	Includes the contents of one style sheet into another. Note: An included style sheet has the same precedence as the including style sheet
<u>key</u>	Declares a named key that can be used in the style sheet with the key() function
<u>message</u>	Writes a message to the output (used to report errors)
<u>namespace-alias</u>	Replaces a namespace in the style sheet to a different namespace in the output
<u>number</u>	Determines the integer position of the current node and formats a number
<u>otherwise</u>	Specifies a default action for the <choose> element
<u>output</u>	Defines the format of the output document
<u>param</u>	Declares a local or global parameter
<u>preserve-space</u>	Defines the elements for which white space should be preserved
<u>processing-instruction</u>	Writes a processing instruction to the output
<u>sort</u>	Sorts the output
<u>strip-space</u>	Defines the elements for which white space should be removed
<u>stylesheet</u>	Defines the root element of a style sheet
<u>template</u>	Rules to apply when a specified node is matched
<u>text</u>	Writes literal text to the output
<u>transform</u>	Defines the root element of a style sheet
<u>value-of</u>	Extracts the value of a selected node
<u>variable</u>	Declares a local or global variable
<u>when</u>	Specifies an action for the <choose> element
<u>with-param</u>	Defines the value of a parameter to be passed into a template

XSLT, XPath, and XQuery Functions

XSLT 2.0, XPath 2.0, and XQuery 1.0, share the same functions library.

Functions Reference

- [Accessor](#)
- [Error and Trace](#)
- [Numeric](#)
- [String](#)
- [AnyURI](#)
- [Boolean](#)
- [Duration/Date/Time](#)
- [QName](#)
- [Node](#)
- [Sequence](#)
- [Context](#)

The default prefix for the function namespace is fn: The URI of the function namespace is: <http://www.w3.org/2005/xpath-functions>

Tip: Functions are often called with the fn: prefix, such as fn:string(). However, since fn: is the default prefix of the namespace, the function names do not need to be prefixed when called.

Accessor Functions

Name	Description
fn:node-name(<i>node</i>)	Returns the node-name of the argument node
fn:nilled(<i>node</i>)	Returns a Boolean value indicating whether the argument node is nilled
fn:data(<i>item.item,...</i>)	Takes a sequence of items and returns a sequence of atomic values
fn:base-uri() fn:base-uri(<i>node</i>)	Returns the value of the base-uri property of the current or specified node
fn:document-uri(<i>node</i>)	Returns the value of the document-uri property for the specified node

Error and Trace Functions

Name	Description
fn:error() fn:error(error) fn:error(error,description) fn:error(error,description,error-object)	Example: error(fn:QName('http://example.com/test', 'err:toohigh'), 'Error: Price is too high') Result: Returns http://example.com/test#toohigh and the string "Error: Price is too high" to the external processing environment
fn:trace(value,label)	Used to debug queries

Functions on Numeric Values

Name	Description
fn:number(arg)	Returns the numeric value of the argument. The argument could be a boolean, string, or node-set Example: number('100') Result: 100
fn:abs(num)	Returns the absolute value of the argument Example: abs(3.14) Result: 3.14 Example: abs(-3.14) Result: 3.14
fn:ceiling(num)	Returns the smallest integer that is greater than the number argument Example: ceiling(3.14) Result: 4
fn:floor(num)	Returns the largest integer that is not greater than the number argument Example: floor(3.14) Result: 3
fn:round(num)	Rounds the number argument to the nearest integer Example: round(3.14) Result: 3
fn:round-half-to-even()	Example: round-half-to-even(0.5) Result: 0 Example: round-half-to-even(1.5) Result: 2 Example: round-half-to-even(2.5) Result: 2

Functions on Strings

Name	Description
------	-------------

Name	Description
fn:string(<i>arg</i>)	Returns the string value of the argument. The argument could be a number, boolean, or node-set Example: string(314) Result: "314"
fn:codepoints-to-string((<i>int,int,...</i>))	Creates a string from a sequence of the Unicode Standard code points Example: codepoints-to-string((84, 104, 233, 114, 232, 115, 101)) Result: 'Thérèse'
fn:string-to-codepoints(<i>string</i>)	Returns the sequence of the Unicode standard code points from a string Example: string-to-codepoints("Thérèse") Result: (84, 104, 233, 114, 232, 115, 101)
fn:codepoint-equal(<i>comp1,comp2</i>)	Returns true if the value of comp1 is equal to the value of comp2, according to the Unicode code point collation (http://www.w3.org/2005/02/xpath-functions/collation/codepoint), otherwise it returns false
fn:compare(<i>comp1,comp2</i>) fn:compare(<i>comp1,comp2,collation</i>)	Returns -1 if comp1 is less than comp2, 0 if comp1 is equal to comp2, or 1 if comp1 is greater than comp2 (according to the rules of the collation that is used) Example: compare('ghi', 'ghi') Result: 0
fn:concat(<i>string,string,...</i>)	Returns the concatenation of the strings Example: concat('XPath ','is ','FUN!') Result: 'XPath is FUN!'
fn:string-join((<i>string,string,...</i>), <i>sep</i>)	Returns a string created by concatenating the string arguments and using the sep argument as the separator Example: string-join('We', 'are', 'having', 'fun!'), ' ') Result: ' We are having fun! ' Example: string-join('We', 'are', 'having', 'fun!')) Result: 'Wearehavingfun!' Example: string-join((), 'sep') Result: ''

Name	Description
fn:substring(<i>string,start,len</i>) fn:substring(<i>string,start</i>)	Returns the substring from the start position to the specified length. Index of the first character is 1. If length is omitted it returns the substring from the start position to the end Example: substring('Beatles',1,4) Result: 'Beat' Example: substring('Beatles',2) Result: 'eatles'
fn:string-length(<i>string</i>) fn:string-length()	Returns the length of the specified string. If there is no string argument it returns the length of the string value of the current node Example: string-length('Beatles') Result: 7
fn:normalize-space(<i>string</i>) fn:normalize-space()	Removes leading and trailing spaces from the specified string, and replaces all internal sequences of white space with one and returns the result. If there is no string argument it does the same on the current node Example: normalize-space(' The XML ') Result: 'The XML'
fn:normalize-unicode()	
fn:upper-case(<i>string</i>)	Converts the string argument to upper-case Example: upper-case('The XML') Result: 'THE XML'
fn:lower-case(<i>string</i>)	Converts the string argument to lower-case Example: lower-case('The XML') Result: 'the xml'
fn:translate(<i>string1,string2,string3</i>)	Converts string1 by replacing the characters in string2 with the characters in string3 Example: translate('12:30','30','45') Result: '12:45' Example: translate('12:30','03','54') Result: '12:45' Example: translate('12:30','0123','abcd') Result: 'bc:da'

Name	Description
fn:escape-uri(<i>stringURI,esc-res</i>)	<p>Example: escape-uri("http://example.com/test#car", true()) Result: "http%3A%2F%2Fexample.com%2Ftest#car"</p> <p>Example: escape-uri("http://example.com/test#car", false()) Result: "http://example.com/test#car"</p> <p>Example: escape-uri ("http://example.com/~bébé", false()) Result: "http://example.com/~b%C3%A9b%C3%A9"</p>
fn:contains(<i>string1,string2</i>)	Returns true if string1 contains string2, otherwise it returns false Example: contains('XML','XM') Result: true
fn:starts-with(<i>string1,string2</i>)	Returns true if string1 starts with string2, otherwise it returns false Example: starts-with('XML','X') Result: true
fn:ends-with(<i>string1,string2</i>)	Returns true if string1 ends with string2, otherwise it returns false Example: ends-with('XML','X') Result: false
fn:substring-before(<i>string1,string2</i>)	Returns the start of string1 before string2 occurs in it Example: substring-before('12/10','/') Result: '12'
fn:substring-after(<i>string1,string2</i>)	Returns the remainder of string1 after string2 occurs in it Example: substring-after('12/10','/') Result: '10'
fn:matches(<i>string,pattern</i>)	Returns true if the string argument matches the pattern, otherwise, it returns false Example: matches("Merano", "ran") Result: true
fn:replace(<i>string,pattern,replace</i>)	Returns a string that is created by replacing the given pattern with the replace argument Example: replace("Bella Italia", "l", "*") Result: 'Be**a Ita*ia' Example: replace("Bella Italia", "l", "") Result: 'Bea Itaia'
fn:tokenize(<i>string,pattern</i>)	Example: tokenize("XPath is fun", "\s+ ") Result: ("XPath", "is", "fun")

Functions for anyURI

Name	Description
fn:resolve-uri(<i>relative,base</i>)	

Functions on Boolean Values

Name	Description
fn:boolean(<i>arg</i>)	Returns a boolean value for a number, string, or node-set
fn:not(<i>arg</i>)	The argument is first reduced to a boolean value by applying the boolean() function. Returns true if the boolean value is false, and false if the boolean value is true Example: not(true()) Result: false
fn:true()	Returns the boolean value true Example: true() Result: true
fn:false()	Returns the boolean value false Example: false() Result: false

Functions on Durations, Dates and Times

Component Extraction Functions on Durations, Dates and Times

Name	Description
fn:dateTime(<i>date,time</i>)	Converts the arguments to a date and a time
fn:years-from-duration(<i>datetimedur</i>)	Returns an integer that represents the years component in the canonical lexical representation of the value of the argument
fn:months-from-duration(<i>datetimedur</i>)	Returns an integer that represents the months component in the canonical lexical representation of the value of the argument
fn:days-from-duration(<i>datetimedur</i>)	Returns an integer that represents the days component in the canonical lexical representation of the value of the argument

Name	Description
fn:hours-from-duration(<i>datetimedur</i>)	Returns an integer that represents the hours component in the canonical lexical representation of the value of the argument
fn:minutes-from-duration(<i>datetimedur</i>)	Returns an integer that represents the minutes component in the canonical lexical representation of the value of the argument
fn:seconds-from-duration(<i>datetimedur</i>)	Returns a decimal that represents the seconds component in the canonical lexical representation of the value of the argument
fn:year-from-datetime(<i>datetime</i>)	Returns an integer that represents the year component in the localized value of the argument Example: year-from-datetime(xs:dateTime("2005-01-10T12:30-04:10")) Result: 2005
fn:month-from-datetime(<i>datetime</i>)	Returns an integer that represents the month component in the localized value of the argument Example: month-from-datetime(xs:dateTime("2005-01-10T12:30-04:10")) Result: 01
fn:day-from-datetime(<i>datetime</i>)	Returns an integer that represents the day component in the localized value of the argument Example: day-from-datetime(xs:dateTime("2005-01-10T12:30-04:10")) Result: 10
fn:hours-from-datetime(<i>datetime</i>)	Returns an integer that represents the hours component in the localized value of the argument Example: hours-from-datetime(xs:dateTime("2005-01-10T12:30-04:10")) Result: 12
fn:minutes-from-datetime(<i>datetime</i>)	Returns an integer that represents the minutes component in the localized value of the argument Example: minutes-from-datetime(xs:dateTime("2005-01-10T12:30-04:10")) Result: 30

Name	Description
fn:seconds-from-dateTime(<i>datetime</i>)	Returns a decimal that represents the seconds component in the localized value of the argument Example: seconds-from-dateTime(xs:dateTime("2005-01-10T12:30:00-04:10")) Result: 0
fn:timezone-from-dateTime(<i>datetime</i>)	Returns the time zone component of the argument if any
fn:year-from-date(<i>date</i>)	Returns an integer that represents the year in the localized value of the argument Example: year-from-date(xs:date("2005-04-23")) Result: 2005
fn:month-from-date(<i>date</i>)	Returns an integer that represents the month in the localized value of the argument Example: month-from-date(xs:date("2005-04-23")) Result: 4
fn:day-from-date(<i>date</i>)	Returns an integer that represents the day in the localized value of the argument Example: day-from-date(xs:date("2005-04-23")) Result: 23
fn:timezone-from-date(<i>date</i>)	Returns the time zone component of the argument if any
fn:hours-from-time(<i>time</i>)	Returns an integer that represents the hours component in the localized value of the argument Example: hours-from-time(xs:time("10:22:00")) Result: 10
fn:minutes-from-time(<i>time</i>)	Returns an integer that represents the minutes component in the localized value of the argument Example: minutes-from-time(xs:time("10:22:00")) Result: 22
fn:seconds-from-time(<i>time</i>)	Returns an integer that represents the seconds component in the localized value of the argument Example: seconds-from-time(xs:time("10:22:00")) Result: 0
fn:timezone-from-time(<i>time</i>)	Returns the time zone component of the argument if any

Name	Description
fn:adjust-dateTime-to-timezone(<i>datetime,timezone</i>)	If the timezone argument is empty, it returns a dateTime without a timezone. Otherwise, it returns a dateTime with a timezone
fn:adjust-date-to-timezone(<i>date,timezone</i>)	If the timezone argument is empty, it returns a date without a timezone. Otherwise, it returns a date with a timezone
fn:adjust-time-to-timezone(<i>time,timezone</i>)	If the timezone argument is empty, it returns a time without a timezone. Otherwise, it returns a time with a timezone

Functions Related to QNames

Name	Description
fn:QName()	
fn:local-name-from-QName()	
fn:namespace-uri-from-QName()	
fn:namespace-uri-for-prefix()	
fn:in-scope-prefixes()	
fn:resolve-QName()	

Functions on Nodes

Name	Description
fn:name() fn:name(<i>nodeset</i>)	Returns the name of the current node or the first node in the specified node set
fn:local-name() fn:local-name(<i>nodeset</i>)	Returns the name of the current node or the first node in the specified node set - without the namespace prefix
fn:namespace-uri() fn:namespace-uri(<i>nodeset</i>)	Returns the namespace URI of the current node or the first node in the specified node set

Name	Description
fn:lang(<i>lang</i>)	Returns true if the language of the current node matches the language of the specified language Example: Lang("en") is true for <p xml:lang="en">...</p> Example: Lang("de") is false for <p xml:lang="en">...</p>
fn:root() fn:root(<i>node</i>)	Returns the root of the tree to which the current node or the specified belongs. This will usually be a document node

Functions on Sequences

General Functions on Sequences

Name	Description
fn:index-of((<i>item</i> , <i>item</i> ,...), <i>searchitem</i>)	Returns the positions within the sequence of items that are equal to the searchitem argument Example: index-of ((15, 40, 25, 40, 10), 40) Result: (2, 4) Example: index-of (("a", "dog", "and", "a", "duck"), "a") Result (1, 4) Example: index-of ((15, 40, 25, 40, 10), 18) Result: ()
fn:remove((<i>item</i> , <i>item</i> ,...), <i>position</i>)	Returns a new sequence constructed from the value of the item arguments - with the item specified by the position argument removed Example: remove(("ab", "cd", "ef"), 0) Result: ("ab", "cd", "ef") Example: remove(("ab", "cd", "ef"), 1) Result: ("cd", "ef") Example: remove(("ab", "cd", "ef"), 4) Result: ("ab", "cd", "ef")
fn:empty(<i>item</i> , <i>item</i> ,...)	Returns true if the value of the arguments IS an empty sequence, otherwise it returns false Example: empty(remove(("ab", "cd"), 1)) Result: false

Name	Description
fn:exists(<i>item, item, ...</i>)	Returns true if the value of the arguments IS NOT an empty sequence, otherwise it returns false Example: exists(remove(("ab"), 1)) Result: false
fn:distinct-values(<i>(item, item, ...), collation</i>)	Returns only distinct (different) values Example: distinct-values((1, 2, 3, 1, 2)) Result: (1, 2, 3)
fn:insert-before(<i>(item, item, ...), pos, inserts</i>)	Returns a new sequence constructed from the value of the item arguments - with the value of the inserts argument inserted in the position specified by the pos argument Example: insert-before(("ab", "cd"), 0, "gh") Result: ("gh", "ab", "cd") Example: insert-before(("ab", "cd"), 1, "gh") Result: ("gh", "ab", "cd") Example: insert-before(("ab", "cd"), 2, "gh") Result: ("ab", "gh", "cd") Example: insert-before(("ab", "cd"), 5, "gh") Result: ("ab", "cd", "gh")
fn:reverse(<i>(item, item, ...)</i>)	Returns the reversed order of the items specified Example: reverse(("ab", "cd", "ef")) Result: ("ef", "cd", "ab") Example: reverse(("ab")) Result: ("ab")
fn:subsequence(<i>(item, item, ...), start, len</i>)	Returns a sequence of items from the position specified by the start argument and continuing for the number of items specified by the len argument. The first item is located at position 1 Example: subsequence((\$item1, \$item2, \$item3,...), 3) Result: (\$item3, ...) Example: subsequence((\$item1, \$item2, \$item3, ...), 2, 2) Result: (\$item2, \$item3)
fn:unordered(<i>(item, item, ...)</i>)	Returns the items in an implementation dependent order

Functions That Test the Cardinality of Sequences

Name	Description
fn:zero-or-one(<i>item, item, ...</i>)	Returns the argument if it contains zero or one items, otherwise it raises an error
fn:one-or-more(<i>item, item, ...</i>)	Returns the argument if it contains one or more items, otherwise it raises an error
fn:exactly-one(<i>item, item, ...</i>)	Returns the argument if it contains exactly one item, otherwise it raises an error

Equals, Union, Intersection and Except

Name	Description
fn:deep-equal(<i>param1, param2, collation</i>)	Returns true if param1 and param2 are deep-equal to each other, otherwise it returns false

Aggregate Functions

Name	Description
fn:count(<i>(item, item, ...)</i>)	Returns the count of nodes
fn:avg(<i>(arg, arg, ...)</i>)	Returns the average of the argument values Example: avg((1,2,3)) Result: 2
fn:max(<i>(arg, arg, ...)</i>)	Returns the argument that is greater than the others Example: max((1,2,3)) Result: 3 Example: max('a', 'k') Result: 'k'
fn:min(<i>(arg, arg, ...)</i>)	Returns the argument that is less than the others Example: min((1,2,3)) Result: 1 Example: min('a', 'k') Result: 'a'
fn:sum(<i>arg, arg, ...</i>)	Returns the sum of the numeric value of each node in the specified node-set

Functions that Generate Sequences

Name	Description
------	-------------

Name	Description
fn:id((<i>string, string, ...</i>), <i>node</i>)	Returns a sequence of element nodes that have an ID value equal to the value of one or more of the values specified in the string argument
fn:idref((<i>string, string, ...</i>), <i>node</i>)	Returns a sequence of element or attribute nodes that have an IDREF value equal to the value of one or more of the values specified in the string argument
fn:doc(<i>URI</i>)	
fn:doc-available(<i>URI</i>)	Returns true if the doc() function returns a document node, otherwise it returns false
fn:collection() fn:collection(<i>string</i>)	

Context Functions

Name	Description
fn:position()	Returns the index position of the node that is currently being processed Example: //book[position()<=3] Result: Selects the first three book elements
fn:last()	Returns the number of items in the processed node list Example: //book[last()] Result: Selects the last book element
fn:current-dateTime()	Returns the current dateTime (with timezone)
fn:current-date()	Returns the current date (with timezone)
fn:current-time()	Returns the current time (with timezone)
fn:implicit-timezone()	Returns the value of the implicit timezone
fn:default-collation()	Returns the value of the default collation
fn:static-base-uri()	Returns the value of the base-uri

XSLT Functions

In addition, there are the following built-in XSLT functions:

Name	Description
<u>current()</u>	Returns the current node
<u>document()</u>	Used to access the nodes in an external XML document
<u>element-available()</u>	Tests whether the element specified is supported by the XSLT processor
<u>format-number()</u>	Converts a number into a string
<u>function-available()</u>	Tests whether the function specified is supported by the XSLT processor
<u>generate-id()</u>	Returns a string value that uniquely identifies a specified node
<u>key()</u>	Returns a node-set using the index specified by an <xsl:key> element
<u>system-property()</u>	Returns the value of the system properties
<u>unparsed-entity-uri()</u>	Returns the URI of an unparsed entity