

Intro to JSON

Why JSON?

You have learned XML which has been used a lot to transfer data between different platforms. The topic of JSON is not covered in the book but this is another important topic to learn as JSON is another transfer data technology that has been increasingly used.

Introducing JSON

JSON is short for JavaScript Object Notation.

Some usage and basic characteristics are:

- Lightweight, text-based open standard for data interchange
- It has been extended from the JavaScript language
- The extension of the file is .json
- The internet media type is application/json and the uniform type identifier is public.json
- It is used when writing JavaScript based application which includes browser extension and websites
- It is used for transmitting structured data over network connection
- Primarily used to transmit data between server and web application
- Easy to read and write and language independent

Although JSON is an extension of JavaScript, it is also available in many different languages such as: C, Ruby, Python, etc.

Example of JSON

```
{
  "book": [
    {
      "id": "01",
      "language": "Java",
      "edition": "third",
      "author": "Lucy Smith"
    },
    {
      "id": "03",
      "language": "VB.NET",
      "edition": "first",
      "author": "Janet Wilde"
    }
  ]
}
```

Can you imagine how this same structure would be written in XML?

Syntax

Remember that JSON is a subset of JavaScript syntax and it includes:

- Data which is represented in name/value pairs
- Curly braces hold objects
- Name is followed by : (colon)
- Name/value pairs are separated by , (comma)
- Square brackets hold arrays and values are separated by , (comma)
- JSON supports the following data structures:
- Collection of name/value pairs
- Ordered list of values (array, list, vector, or sequence)

Take a look at the example presented in the previous chapter (here below again) and try to recognize the elements written above:

```
{
  "book": [
    {
      "id": "01",
      "language": "Java",
      "edition": "third",
      "author": "Lucy Smith"
    },
    {
      "id": "03",
      "language": "VB.NET",
      "edition": "first",
      "author": "Janet Wilde"
    }
  ]
}
```

Data Types

Type	Description
Number	Double-precision floating-point format
String	Double-quoted Unicode with backslash escaping
Boolean	True or False
Value	A string, a number, true or false, null, etc.
Array	Ordered sequence of values
Whitespace	Used between any pair of tokens
Object	Unordered collection of key:value pairs
Null	empty

Number

- Octal and hexadecimal formats are not used
- No NaN or Infinity is used in Number

Syntax: `var json-object-name = {string: number_value,}`

Example: `var obj = {marks: 97}`

String

- Sequence of zero or more double quoted characters with backslash escaping

Syntax: `var json-object-name = {string: number_value,}`

Example: `var obj = {name: 'Joseph'}`

String Types

Type	Description
"	Double quote
\	Reverse solidus
/	Solidus
b	Backspace
f	Form feed
n	New line
r	Carriage return
t	Horizontal tab
u	Four hexadecimal digits

Boolean

- Includes true or false values

Syntax: `var json-object-name = {string: true/false,}`

Example: `var obj = {name: 'Josh', marks: 97, grad: true}`

Array

- Begins with [and ends with]. Values are separated by , (comma). Indexing can start at 0 or 1

Syntax: `[value,]`

Example: `{ "books": [{ "language": "C", "edition": "first" }, { "language": "C++", "edition": "third" }] }`

Object

- Enclosed in curly braces
- Each name followed by : (colon) and name/value pairs separated by , (comma)
- Keys must be strings and should be different from each other

Syntax: {string: value,}

Example showing Object:

```
{
  "id": "1A",
  "language": "C++",
  "price": 150
}
```

Whitespace

- Can be inserted between any pair of tokens

Syntax: {string: " ",}

Example: var a = " bold"; var b = " lab"

Null

- It means empty type

Syntax: null

Example: var i = null;

Creating Simple Objects

JSON Objects can be created with JavaScript. Here are some examples:

Creating empty Object

```
var JSONObj = {};
```

Creating a new Object

```
var JSONObj = new Object();
```

Creating an object with attributes prodname and price

```
var JSONObj = {"prodname": "Orange", "price": 1 };
```

Take a look at the code in file [jsonobject.htm](#) in the JSONFiles folder - open it in a browser

Creating Array Objects

Take a look at the code in file [jsonarray.htm](#) in the JSONFiles folder - open it in a browser

JSON Schema

It's an specification for defining JSON data - remember XML Schema?!? It describes existing data format; creates a clear human - and machine-readable documentation. It's a complete structural validation, useful for automated testing, validating client-submitted data.

There are different **JSON validator** libraries for different programming languages:

- WJElement (LGPLv3) for C;
- Json.NET (MIT) for .NET;
- Jschema for Python;
- php-json-schema (MIT), json-schema (Berkeley) for PHP;
- schema.js, Orderly (BSD), JSV, Matic (MIT) for JavaScript;
- etc.

Take a look at file [jsonschema.txt](#) where you will see a basic JSON schema. You will find this file inside the JSON folder.

Important keywords you will see in jsonschema.txt:

- **\$schema** - states that the schema is written according to the draft v4 specification
- **title** - gives a title to the schema
- **description** - description of the schema
- **type** - defines first constraint (has to be JSON Object)
- **properties** - defines various keys and the value types
- **required** - list of required properties
- **minimum** - minimum acceptable value
- **exclusiveMinimum** - if this has value of true, the instance is valid if strictly greater than the minimum value
- **maximum** - maximum acceptable value
- **exclusiveMaximum** - if this is true, the instance is valid if strictly lower than maximum value
- **multipleOf** - numeric instance valid if result of division of instance by this keyword's value is integer
- **maxLength** - length of string instance defined as maximum number of its characters
- **minLength** - length of a string instance defined as minimum number of its characters
- **pattern** - string instance considered valid if the regular expression matches the instance successfully

You can check the site <http://json-schema.org> for complete list of keywords that can be used when defining a JSON schema

Comparing JSON with XML

- Both have support for creation, reading and decoding in real situations.
- XML is more verbose than JSON, so, it's faster to write JSON for humans and produces smaller files
- Support has grown among major companies - Example: Twitter with API's that supports JSON
- XML is used to describe structured data which does not include arrays while JSON includes arrays

Take a look at file [jsonxml.txt](#) where it shows the same data described using JSON and XML.

Based on this example, you should try to get one of the first XML files we used in the course and try to rewrite it in JSON.

Example with JavaScript

```
{
  "name": "Douglas Jason",
  "position": "Programmer",
  "courses": ["C", "VB.NET", "Java"]
}
var info = JSON.parse(data);
info.courses[1]
```

Once you have the JSON data declared, you can access it using the **JSON.parse** method, declaring a variable (in the example above, the variable is **info**).

Refer to the values using the dot notation shown in the example. The **info.courses[1]** will then refer to "VB.NET" (indexes start by zero).

Notice the array type used to define keyword **courses**.

Accessing Values

The language - example shown was JavaScript - will not store the data in the order as it comes. If you want to make sure that it stores in a pre-defined order, you should declare an array.

The previous example showed the value being retrieved using the normal dot notation:

```
var info = {"full_name" : "Claudia Da Silva"};
console.log(info.full_name);
```

You can also access the value using the **[]** notation:

```
var info = {"full_name" : "Claudia Da Silva"};
console.log(info["full_name"]);
```

Validating JSON

Take a look at file [jsonexample.html](#)

Notice the way the JSON data is declared - the double quotes, colon, comma, etc.

You can validate your code using, for example JS Hint - <http://www.jshint.com> - note that depending on how you set up the rules on this site, you may get errors that are not true errors.

Another validator is JSONLint - <http://www.jsonlint.com> - it's a **JSON validator**, in this one, you should not include the JavaScript code or you will get an error.

JSON Editors

A good JSON Editor is **JSON Editor Online** - <http://jsoneditoronline.org> - on the right side of the screen, you will be able to look at the object as you would on the console and then modify the object values on that side and transfer (using the arrows) to the code (that is shown on the left side).

Some offline tools:

- Cocoa JSON Editor - for Mac computers - <http://www.cocoajsoneditor.com>

- Free JSON Editor from CNET - for PC Windows computers - http://download.cnet.com/Free-JSON-Editor/3000-2213_4-76170769.html
- JSON Editor Online - <https://www.jsoneditoronline.org/>

JSON Add ons

For Firefox - **JSONView** - <https://addons.mozilla.org/en-US/firefox/addon/jsonview> or at the Chrome Web Store at <http://goo.gl/PZaa2>

For WordPress - **JSON API** - <http://goo.gl/hldXy> - has not been updated recently but it's a great tool to pull your JSON data out of WordPress such as articles, categories, posts and convert it into JSON data that can be used in other applications.

Note about Web Browser Tools

When using the Web Developer Tools in Browsers - example Chrome - you can modify the values of the elements of your JSON structure but this is a temporary modification and you are not modifying the file. This might be a good way to debug and "play around" with the data structure.

To delete an element from an array, use splice and to insert an element in an array, use push such as:

`info.links.splice(1,1)` - deleting 1 element from the links array that was received by the info variable - the element being deleted is the one with index = 1

```
info.links.push({"courses":"http://www.lynda.com"})
```

Example using JSON with HTML and JavaScript - a simple alert box

Take a look at the file in your XMLFiles, inside the JSONFiles folder, called **jsonalert.html**. Take a look at the code and recognize the JSON format inside the small JavaScript code! It's just one simple example on how you would be receiving a JSON file and then display its data in a web page.

Example using JSON with HTML and JavaScript - JSON data as an array

Take a look at the file in your XMLFiles, inside the JSONFiles folder, called **jsonanimals.html**. Take a look at the code and recognize the JSON format inside the small JavaScript code! It's just one simple example on how you would be receiving a JSON file and then display its data in a web page.

Another example using JSON

Take a look at the web page http://iviewsource.com/exercises/json_finished/

This example used jQuery code from <http://jquery.com>, also used Mustache JS <http://mustache.github.com>, the rotator cycle from <http://jquery.malsup.com/cycle>

Steps to create the example

- Start with a simple HTML document as the one shown in [jsonrotating.html](#) file
- Insert a div with id="speakerbox"
- Inside the speakerbox div, insert another div with id="carousel"
- You will need to copy the 3 libraries you will be using (jQuery, jQuery cycle, and Mustache JS) right after the div tags you inserted, in the body section of your HTML document - you can copy these libraries from the cdnjs website - <http://cdnjs.com>
- Templating with Mustache JS - you can have as many different templates you want in the same page. The template can be seen in the data.json file - you need to make sure that the JSON file is created perfectly (validate it using, for example, the JSON Editor Online), otherwise, you will not be able to work with it
- Between the <script> and </div> tags, insert the template in your HTML document with <script id="speakerstpl" type="text/template"></script>
- Between the <script> and </script> tags you just inserted, paste the code you see in template.txt - do not include the comments you see at the bottom of the file starting with Comments: (these comments are just for you to understand the code that you are inserting in your HTML document)
- After all the CDN scripts you inserted, insert another pair of script tags as <script type="text/javascript"></script>
- Between these tags you will insert the jQuery code found in script.txt without the Comments:

Up to this point, if you upload the files to a server and show the html document in a browser, you will see the 3 speakers shown in sequence -To apply the library that can cycle and show only one speaker at a time, you should proceed with the steps that follow below.

The example used jQuery cycle - a plugin found at <http://jquery.malsup.com/cycle> - the CDN was already inserted in our HTML document, remember?

- In the jQuery code you pasted from [script.txt](#), right after `$('#carousel').html(html);` line, insert the code from [scriptcycle.txt](#) - this code will be responsible for cycling the pictures and info of the speakers one by one on the web page
- Remember that the part starting at Comments: should not be included in your HTML code
- Some options were used and the different options in this cycle can be seen at <http://jquery.malsup.com/cycle/options.html>
- Right after the opening <div id="speakerbox">, insert two anchor tags (links) for the buttons you will be using (the prev and next buttons) as

```
<a href="#" id="prev_btn">&laquo;</a>
```

```
<a href="#" id="next_btn">&raquo;</a>
```

- The « and &; represent the arrows that will be seen on the buttons (left and right arrows)
- If you upload now this new HTML and show in a browser, you will notice that the elements are being shown once at a time
- You can style the way these elements are shown using CSS. The CSS can be found in [mystyle.css](#).
- Include in the head section of your HTML document the link to the CSS file as

```
<link rel="stylesheet" href="mystyle.css" />
```

In order to see this example working, you will need to transfer your HTML document to a live server (you can use your Hills account for example) - AJAX is the communication that happens between a client and a server - it will not work with local files.

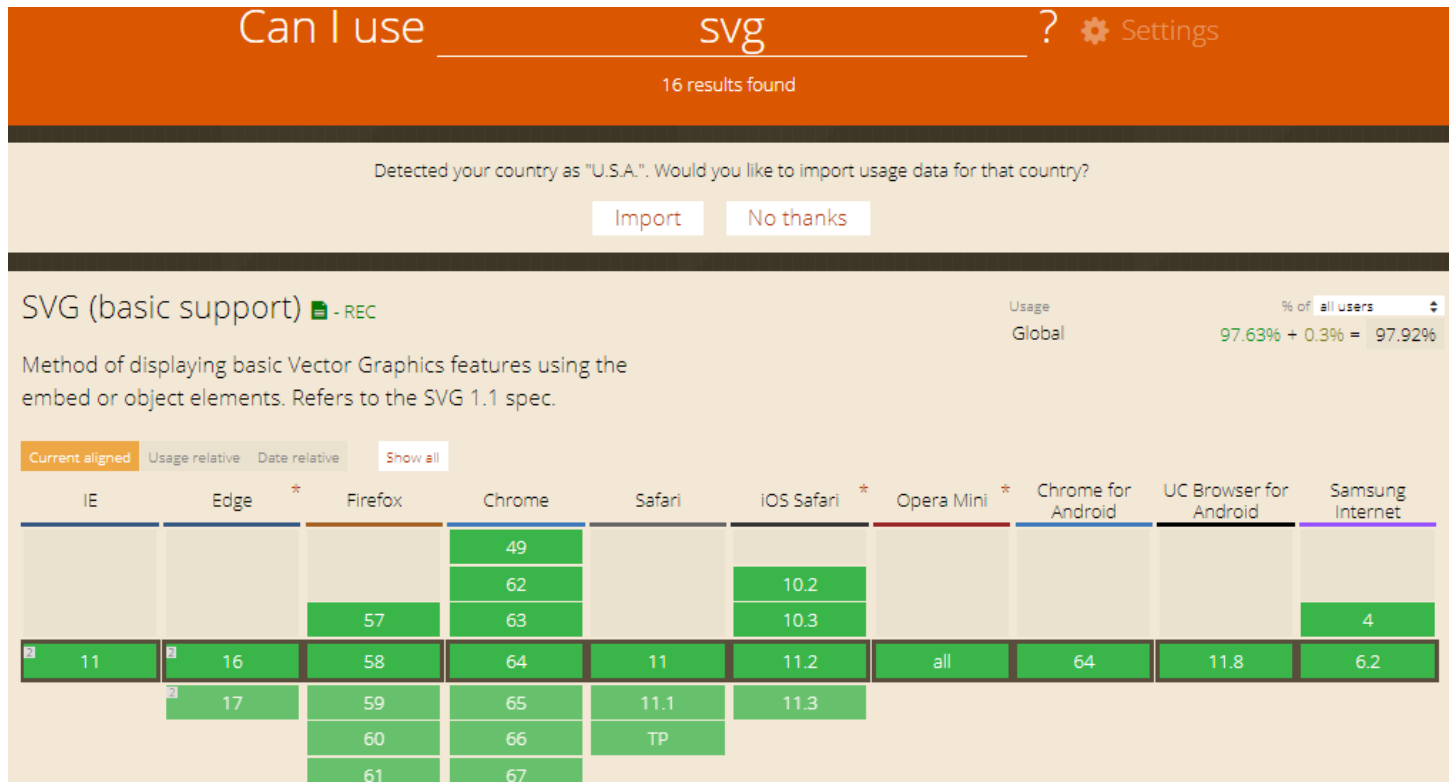
All files need to be uploaded to the server (the data.json, the images folder (with the 3 pictures), and the HTML document

If you followed the steps correctly, your final HTML file should be equal to [jsonrotatingdone.html](#)

SVG

Introduction to SVG

SVG (Scalable Vector Graphics) language is one of the most well-known XML applications. It's a W3C specification that provides markup elements that you can use to describe 2D graphics. Notice in the image below the wide support from browsers (this image was made in Feb/2018):



There are some important advantages of SVG over other graphic formats:

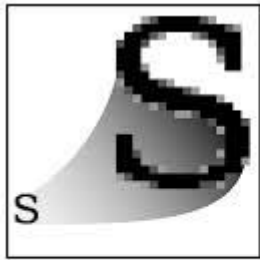
- a) SVG graphics can be animated
- b) SVG is text-based (while other formats such as GIF and PNG are stored in machine-readable binary code)
- c) SVG is an official W3C standard
- d) SVG supports transparency and filters
- e) SVG graphics can be scaled
- f) SVG graphics are searchable

The standard of SVG is designed in a way that it works very well with CSS and JavaScript which means that you can use CSS to control the appearance of the graphic and you can use JavaScript to apply some functionality to the graphic.

Because SVG is text-based, you can easily edit it using any plain text editor and also they can be ranked by search engines according to their content.

Vector formats describe graphics as a series of shapes that are drawn from one point to another – for example, a rectangle is defined by the coordinates of two points on the opposite corners. This is a more efficient way than the bitmap format that describes the color of each pixel that composes the graphic. This all mean that SVG files tend to be smaller in file-size especially for larger graphics.

SVG graphics can also be scaled up without any loss in quality which is a huge difference from a JPEG-format



Raster
.jpeg .gif .png



Vector
.svg

graphic – this means a lot especially in a world where graphics are developed to be shown on big screens but also on the screens of mobile devices – notice the example in the image below where the black S on the left side is all “pixelated” (especially when scaled up) as it was built using raster technology (or it’s a jpg, or a gif, or a png file) while the blue S on the right side that remains perfect no matter how much it’s scaled up.

Making some graphic shapes

SVG files will have the .svg extension. You need to start the SVG file as you would start any XML file – declaring the XML. Also, to draw the shapes, you need first to declare the <svg> element which will contain the shapes you want to draw.

The SVG specifications provide elements such as: <rect>, <circle>, and <ellipse> that are used to draw simple shapes. These elements have some attributes such as: x and y to determine the position of the graphic on the page; the width and height to determine the size of the shape, and the fill to determine the color of the shape.

If you need to draw more complex shapes, you can specify a list of x,y coordinates to the points attribute of the <polygon> element.

All the shapes elements support further optional attributes such as: opacity, stroke-width (the same of border-width that we know from CSS), and stroke (the border-color).

Let’s then create some shapes as an exercise:

1) Open your web editor with an empty file – save the file as **svg1.svg**

2) As we will start a standard XML file, then we need to have, in the first line, the following – look at the code in the image below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2
```

3) We will add the SVG root element with the version, namespace reference, and overall size – see the image below now with the SVG element declared:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4
5
6 </svg>
7
```

Note: The **baseProfile** attribute is used to describe the minimum SVG language profile that the developer believes is necessary to render the content. If the attribute is not specified, the effect is as if a value of none is specified. The values of this attribute can be: **full** – represents a normal profile and generally used for most computers (desktops, laptops); **basic** – represents a light weight profile and was mostly used for PDAs (rarely seen); **tiny** – represents an even more light weight profile and is generally used for cell phones. In reality, the **baseProfile** attribute will work together with the version attribute.

4) The svg element will then contain the shapes we want to draw. So, let's start by drawing a rectangle. We will define the rectangle using the <rect> element that will be coded between the <svg> and </svg> like the image below is showing the code (notice that we are drawing a square as both height and width have the same value = 120px):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5
6 </svg>
```

5) We will now use the <rect> element again but now to draw a rectangle (not a square) and with rounded borders – see the code for this new shape in the image below, right after the first <rect> element we coded:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="140" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 </svg>
```

6) Now, we will add a <rect> element that will be pink but with only 30% opacity – look at the third <rect> element that was added in the code below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="140" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 <rect x="10" y="150" opacity="0.3" fill="pink" width="100px" height="80px" />
7 </svg>
```

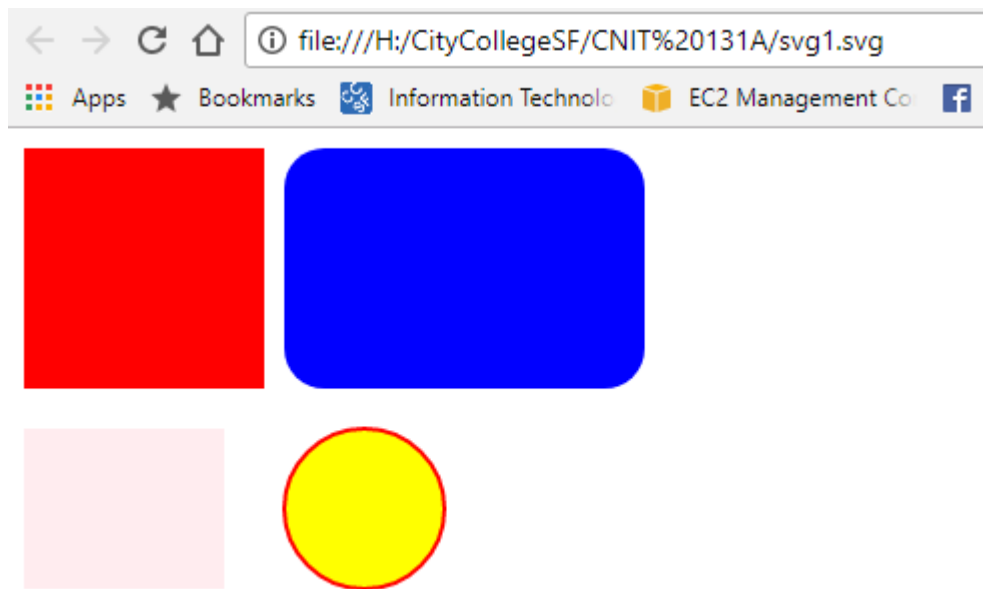
Note: Remember that the x and y attributes are used to position the shape on the page (on the browser window).

7) We will now add a perfect circle that will be color yellow and will have a red border around it. See the <circle> element coded below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="140" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 <rect x="10" y="150" opacity="0.3" fill="pink" width="100px" height="80px" />
7 <circle fill="yellow" cx="180" cy="190" r="40px" stroke="red" stroke-width="2px" />
8 </svg>
```

Note: instead of using x and y attributes, we are using cx and cy and this is to position, in the browser, the center point of the circle. The r attribute is defining the size of the radius of the circle

If you save this file up to this point and show in the browser, you should see something as: (you barely can see the rectangle with opacity 30% that is below the red rectangle element in the image below, but it's there!!!)



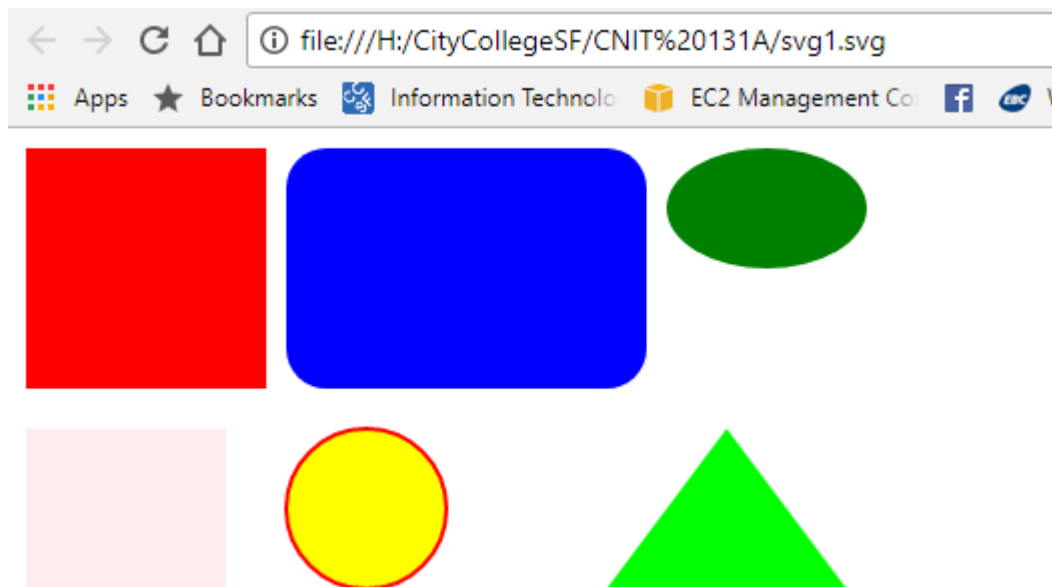
8) We will now draw an ellipse that will be beside the blue rectangle with rounded corners – notice that although the ellipse will be the last one coded, you determine the position of the element by the cx and cy attributes. Notice also that the ellipse has two values for radius the rx and ry, otherwise it would be a circle:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="140" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 <rect x="10" y="150" opacity="0.3" fill="pink" width="100px" height="80px" />
7 <circle fill="yellow" cx="180" cy="190" r="40px" stroke="red" stroke-width="2px" />
8 <ellipse fill="green" cx="380" cy="40" rx="50px" ry="30px" />
9 </svg>
```

9) We will finally use a polygon element to draw a lime triangle that will be shown beside the yellow circle – again, notice that now, we will be using the points attribute to set the x and y position of each pair of points that make the corner of the triangle:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="140" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 <rect x="10" y="150" opacity="0.3" fill="pink" width="100px" height="80px" />
7 <circle fill="yellow" cx="180" cy="190" r="40px" stroke="red" stroke-width="2px" />
8 <ellipse fill="green" cx="380" cy="40" rx="50px" ry="30px" />
9 <polygon fill="lime" points="300,230,360,150,420,230" />
10 </svg>
```

Now, if you save the **svg1.svg** file again and refresh the browser, you should see the following:



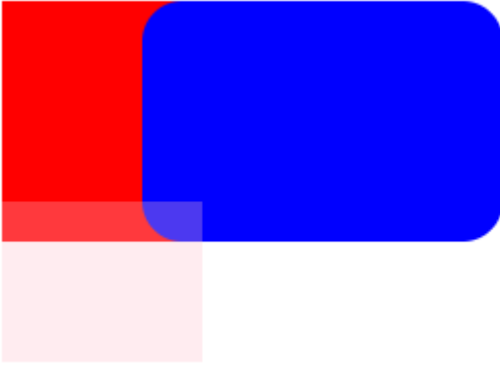
DO NOT FORGET!!!

The x and y attributes specify the top left corner position of box elements while the cx and cy attributes specify the center position (point) of circles.

Another important note: the shapes listed last in the code will be pasted over the aforementioned shapes – this means that if you want a shape to appear in the background, it should be the first one in your code as SVG shapes are “drawn” in the browser in order from top to bottom – if you would change the svg1.svg code to be like the image below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <rect x="10" y="10" fill="red" width="120px" height="120px" />
5 <rect x="80" y="10" rx="20" ry="20" fill="blue" width="180px" height="120px" />
6 <rect x="10" y="110" opacity="0.3" fill="pink" width="100px" height="80px" />
7 </svg>
```

You would see the following in the browser:



Drawing Lines

We have the `<line>` element specified in SVG to draw lines. This element will have the `x1` and `y1` attributes to specify the starting point of the line and also the `x2` and `y2` attributes to specify the ending point of the line. The `stroke` attribute will specify the color of the line (lines do not have fill attribute) and to define the thickness of the line, we use the `stroke-width` attribute. You can also create dashed lines by adding a `stroke-dash-array` attribute to specify the dash length and interval.

We will now create another SVG file with lines!

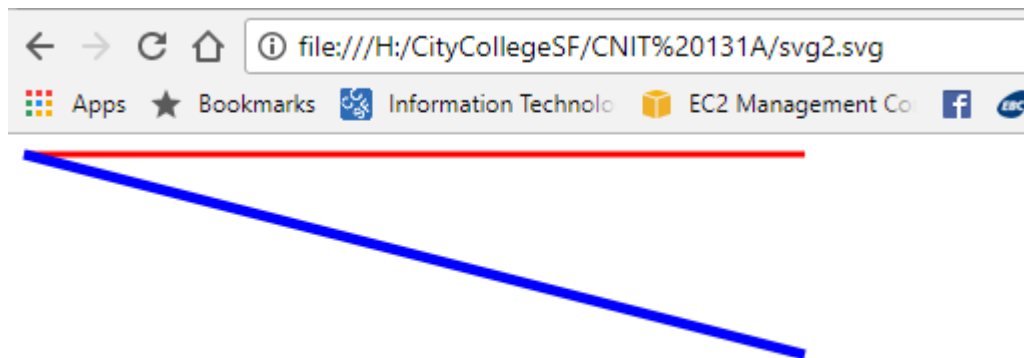
- 1) Open the **svg1.svg** you created in the exercise before in the web editor
- 2) Delete everything between `<svg>` and `</svg>` (the `rect`, `circle`, `ellipse`, etc. elements)
- 3) Save the file as **svg2.svg**
- 4) Right after the opening `<svg>`, we will draw a red horizontal line – see the code below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <line x1="10" y1="10" x2="400" y2="10" stroke="red" stroke-width="3px" />
5 </svg>
```

- 5) We will now draw a blue angled line – see the code below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <line x1="10" y1="10" x2="400" y2="10" stroke="red" stroke-width="3px" />
5 <line x1="10" y1="10" x2="400" y2="110" stroke="blue" stroke-width="5px" />
6 </svg>
```

Right now, if you show your svg file in the browser, you should have something as shown below:



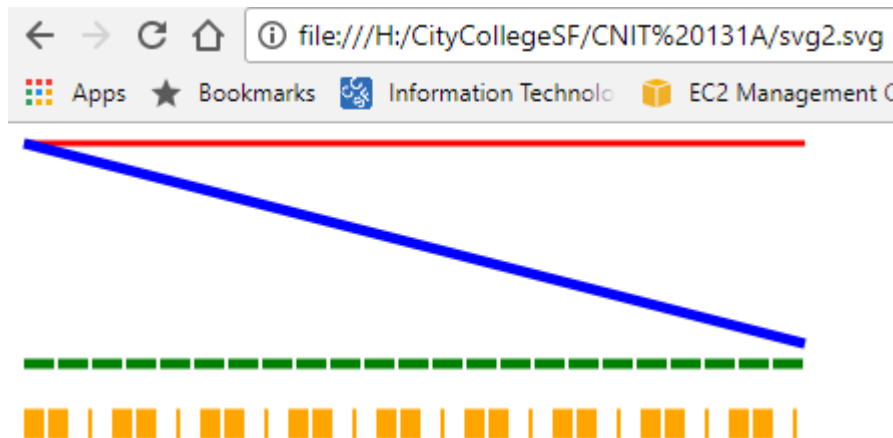
6) We will now add a green dashed line with a dash length of 15 and dash interval of 2 – see the code below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <line x1="10" y1="10" x2="400" y2="10" stroke="red" stroke-width="3px" />
5 <line x1="10" y1="10" x2="400" y2="110" stroke="blue" stroke-width="5px" />
6 <line x1="10" y1="120" x2="400" y2="120" stroke="green" stroke-width="5px" stroke-dasharray="15,2" />
7 </svg>
```

7) We will now draw a fancier dashed line with the orange color and dash lengths and intervals of 10 and 2 – see the code below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <line x1="10" y1="10" x2="400" y2="10" stroke="red" stroke-width="3px" />
5 <line x1="10" y1="10" x2="400" y2="110" stroke="blue" stroke-width="5px" />
6 <line x1="10" y1="120" x2="400" y2="120" stroke="green" stroke-width="5px" stroke-dasharray="15,2" />
7 <line x1="10" y1="150" x2="400" y2="150" stroke="orange" stroke-width="15px" stroke-dasharray="10,2,10" />
8 </svg>
```

The browser will now have the following (after you save the **svg2.svg** file and refresh the browser):



Note: The x and y coordinates specify the center of a line and the stroke is applied equally to each side – for example, a line with a stroke-width of 6px will apply 3p to each side of the specified center.

Using the <defs> element

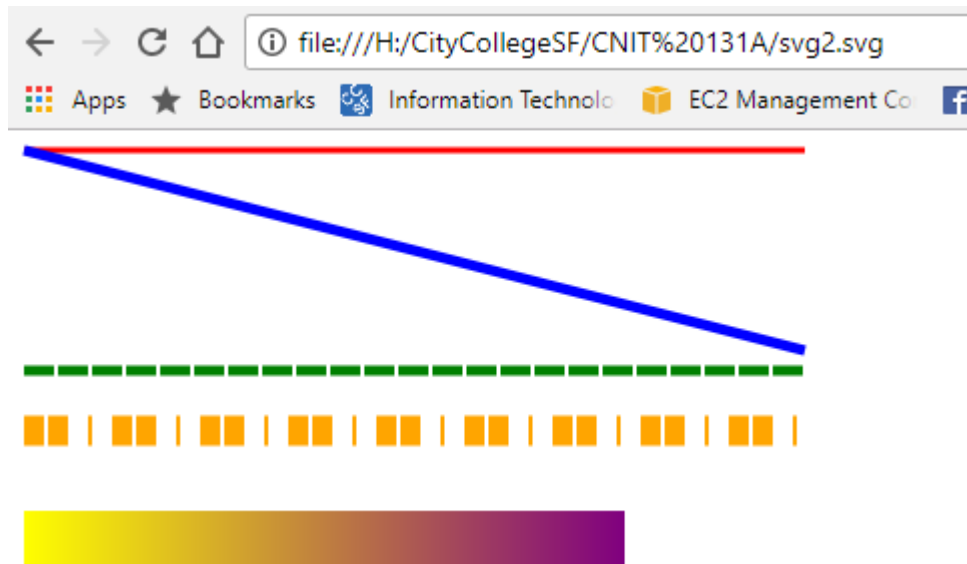
The <defs> element can be used so, inside of it (between <defs> and </defs>) you can define constants. Then you can use those constants for any shape/element you draw in your code.

For example, let's modify the **svg2.svg** file and let us create the <defs>...</defs> element and inside that block we will define a linear gradient fill called LGx that runs from yellow to purple, then we will use that constant defined (LGx) to fill the <rect> element (to be the color of the rectangle element) – look at the code below where you see the <defs> element coded and the linearGradient constant LGx being defined and then the <rect> element using the LGx constant for the fill attribute (to color the rectangle):

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <line x1="10" y1="10" x2="400" y2="10" stroke="red" stroke-width="3px" />
5 <line x1="10" y1="10" x2="400" y2="110" stroke="blue" stroke-width="5px" />
6 <line x1="10" y1="120" x2="400" y2="120" stroke="green" stroke-width="5px" stroke-dasharray="15,2" />
7 <line x1="10" y1="150" x2="400" y2="150" stroke="orange" stroke-width="15px" stroke-dasharray="10,2,10" />
8 <defs>
9   <linearGradient id="LGx">
10     <stop offset="0%" stop-color="yellow" />
11     <stop offset="100%" stop-color="purple" />
12   </linearGradient>
13 </defs>
14 <rect x="10" y="190" width="300px" height="30px" fill="url(#LGx)" />
15 </svg>
```

Note: The same way that you used the <linearGradient> element, you could have used the <radialGradient> if you want to give a radial gradient effect to the coloring.

Save the **svg2.svg** and refresh the browser and you should see the following:



Note: The id value needs to be unique but we know that this is a specification that comes from W3C – in the HTML code, you need to have a unique id value per page!

Producing Graphic Paths

You can specify a particular “path” that lines can follow by specifying a list of x,y coordinates to the points attribute in a <polyline> element. This will work very much like the <polygon> element we have studied before.

The <path> element is the most important SVG element because it allows you to draw both straight and curved lines along a specified path. It also supports fill and stroke values so you can draw complex shapes. The d (data) attribute specifies the path as a series of x,y coordinates in a list and those pairs are separated by a white space (not commas).

You can also include commands in the path list like the ones shown below:

Command	Data	Description
Z	-	closePath – it will draw a line from the final point to the first point
M	x y	moveTo – move the “pen” to x y point but draw no line. All path data must begin with this command
L	x y	lineTo – draw a line from the current point to the x y point
H	x	Horizontal lineTo – draw a horizontal line from the current point to the x point
V	y	Vertical lineTo – draw a vertical line from the current point to the y point
Q	x1 y1 x y	quadratic Bezier curveTo – draw a curved line from the current point to x y point using control point x1 y1 to calculate the curve

Note: most graphics SVG drawing software tend to use the <path> element to describe all types of shapes. For example, the free [Inkscape SVG drawing program](#) that is available for Windows, Mac and Linux.

Let us try the <path> element.

- 1) Open the **svg2.svg** file in your web editor
- 2) Delete all the elements between <svg> and </svg>
- 3) Save the file as **svg3.svg**
- 4) We will now define an element to draw a green line along a set of coordinates – look at the code with the <polyline> element inserted:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <polyline stroke="green" stroke-width="5px" fill="none" points="10,10 280,10 280,75 360,100" />
5 </svg>
```

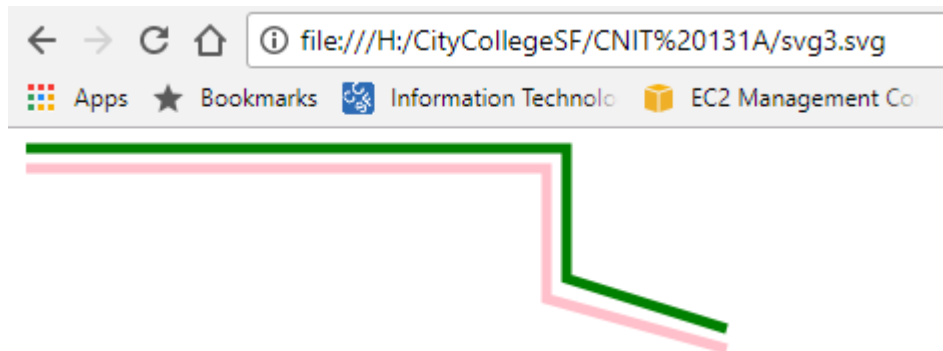
Note: The fill value needs to be explicitly specified as “none” if no fill is required along paths!

5) Let us now draw a pink line along a similar path to the last line – see the code below with the <path> element being defined:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3 width="500px" height="250px">
4 <polyline stroke="green" stroke-width="5px" fill="none" points="10,10 280,10 280,75 360,100" />
5 <path stroke="pink" stroke-width="5px" fill="none" d="M 10 20 L 270 20 270 85 360 110" />
6 </svg>
```

Notice how the data (d) attribute has the points defined separated by a white space instead of comma (,) and also notice the use of the commands that we showed to you in the table found in the previous page – the M command which is always the start of any path and then the L command that will keep drawing lines from the current point to the next defined, for example, the first x y pair after the L command is 270 20 which means that a line will be drawn from the current point (the starting point which was at 10 20 – right after the M command) until the 270 20 point (x=270 and y=20) – the same logic will apply to the other 2 pairs of points after the L command.

Now, if you save the **svg3.svg** and show it in the browser, it should show the green and pink lines drawn as it is showing in the image below:



6) Now we will continue working on the **svg3.svg** file and we will add a new line, color blue, along a similar path but this time we will use the H and V commands – look at the code below with a new <path> element defined:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
width="500px" height="250px">
<polyline stroke="green" stroke-width="5px" fill="none" points="10,10 280,10 280,75 360,100" />
<path stroke="pink" stroke-width="5px" fill="none" d="M 10 20 L 270 20 270 85 360 110" />
<path stroke="blue" stroke-width="5px" fill="none" d="M 10 30 H 260 V 95 L 360 120" />
</svg>
```

Note: remember that you are coding XML which is very strict and if you forget, for example, the double quotes, or forget to close an element, you might get in the browser errors like this one here:

This page contains the following errors:

error on line 7 at column 1: Unescaped '<' not allowed in attributes values

Below is a rendering of the page up to the first error.



This happened when I did not close the double quotes of the d attribute of the last <path> element (for the blue line) – what is interesting is that it showed the error in the next line which had </svg> (the closing svg tag) – so, you need to be very detail-oriented to read the error and not focus only on the line/column that the browser is pointing at because the error might be in the previous line!

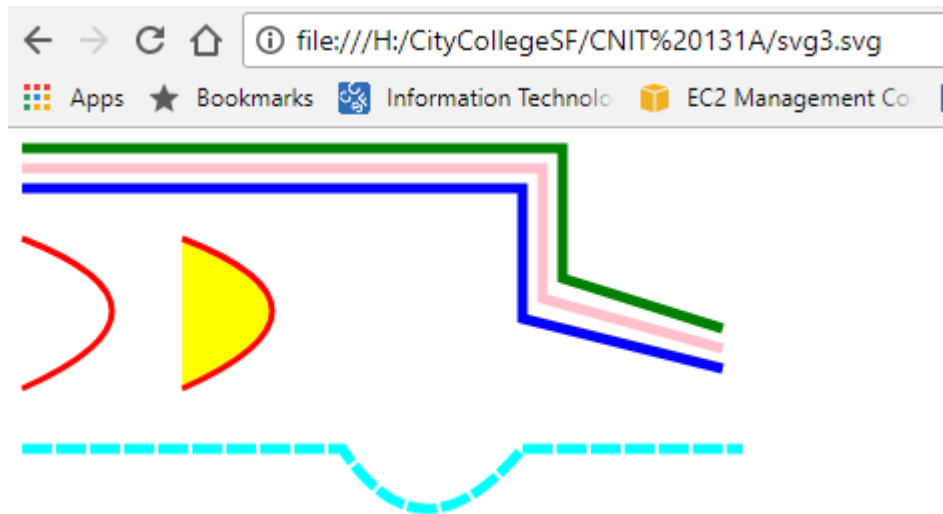
7) We will now draw two curves – one opened and one with the path filled – they will have the red color and the fill will be yellow – see the code showing the 2 curves added (2 new <path> elements):

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
width="500px" height="250px">
<polyline stroke="green" stroke-width="5px" fill="none" points="10,10 280,10 280,75 360,100" />
<path stroke="pink" stroke-width="5px" fill="none" d="M 10 20 L 270 20 270 85 360 110" />
<path stroke="blue" stroke-width="5px" fill="none" d="M 10 30 H 260 V 95 L 360 120" />
<path stroke="red" stroke-width="3px" fill="none" d="M 10 55 Q 100 90 10 130" />
<path stroke="red" stroke-width="3px" fill="yellow" d="M 90 55 Q 180 90 90 130" />
</svg>
```

8) The last line we will draw will be of cyan color, dashed combining horizontal lines with a curve – see the code added below here:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
width="500px" height="250px">
<polyline stroke="green" stroke-width="5px" fill="none" points="10,10 280,10 280,75 360,100" />
<path stroke="pink" stroke-width="5px" fill="none" d="M 10 20 L 270 20 270 85 360 110" />
<path stroke="blue" stroke-width="5px" fill="none" d="M 10 30 H 260 V 95 L 360 120" />
<path stroke="red" stroke-width="3px" fill="none" d="M 10 55 Q 100 90 10 130" />
<path stroke="red" stroke-width="3px" fill="yellow" d="M 90 55 Q 180 90 90 130" />
<path stroke="cyan" fill="none" stroke-width="5px" stroke-dasharray="15,2" d="M 10 160 H 170 Q 210 220 260 160 H 370" />
</svg>
```

If you save the file and refresh it in the browser, you should get something as you see below:



REMEMBER!!!

Coordinates in path data attribute (d) must be separated by white space only! Not commas (,)!!!

Transforming Graphic Groups

You can structure SVG documents to group several shapes within the `<g>` `</g>` element. The presentational attributes you can use for the `<g>` element will be inherited by its child elements. For example, if you have `<g fill="red">` each child element would inherit the fill value to be of color red. But a value inherited from the parent can be overridden if you assign a different value to the same attribute in the child tag, so, if you had an element such as `<rect fill="green">`, inside the `<g fill="red">`, the green color would override the red color and the rectangle would be green.

The appearance of an SVG element can be adjusted if you use the transform attribute specifying a transformation function such as: scale, rotate, skew, or translate and if this is assigned to a `<g>` element, the transformation is applied to the entire group.

- **Translate function** – adjusts position by adding the values supplied as arguments to the current x,y coordinates
- **Rotate function** – adjusts the current orientation using the angle supplied as an argument
- **Scale function** – adjusts the size by multiplying the current width and height by the values supplied as arguments
- **Skew function** – skews the appearance of the element or group by providing separate skewX and skewY functions to adjust each axis to the angle that is supplied as argument

Let us now create a new .svg file and practice with transforming a group of elements.

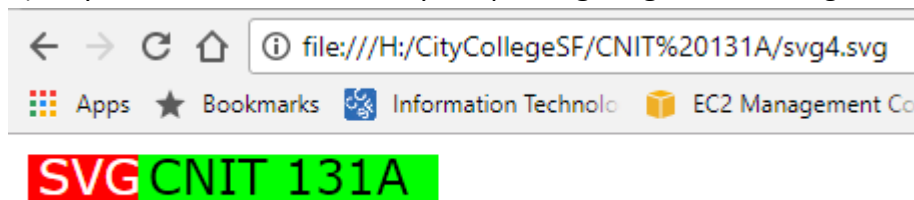
- 1) Open **svg3.svg** in a web editor and delete all the elements between `<svg>` and `</svg>`
- 2) Save the file as **svg4.svg**

3) Let us now create a group like you see in the image below showing the code already with the <g> element created:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
width="500px" height="250px">
  <g font-size="1.5em" font-family="Verdana">
    <rect x="10" y="10" width="55px" height="25px" fill="red" />
    <rect x="65" y="10" width="150px" height="25px" fill="lime" />
    <text x="15" y="30" fill="white">SVG</text>
    <text x="70" y="30" fill="black">CNIT 131A</text>
  </g>
</svg>
```

Do not forget to close the tags, and to open and close the double quotes, close the self-closing elements, etc. otherwise you will get error messages!!!

4) All you see in the browser, if you open **svg4.svg**, is something like this below:



5) Copy the <g> element 4 times because now we will apply some transformations (so, you will have from <g font-size="...."> until </g> 4 times more still before the </svg>)

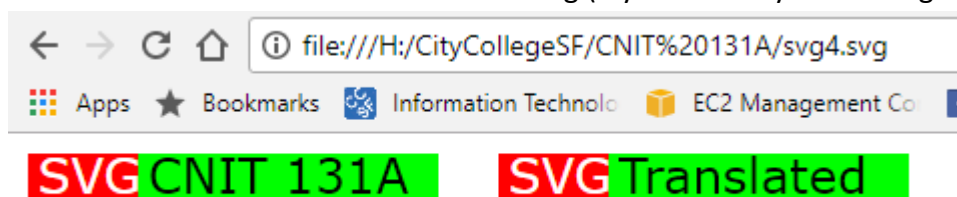
6) In the first <g> element, you will change the second <text> element to be **Translated** instead of **CNIT 131A** and insert the following attribute in the <g> element to translate it (to adjust the group's x,y position):

transform = "translate(235, 0)". The code for this group would look like what you see below:

```
10 <g font-size="1.5em" font-family="Verdana" transform="translate(235, 0)">
11   <rect x="10" y="10" width="55px" height="25px" fill="red" />
12   <rect x="65" y="10" width="150px" height="25px" fill="lime" />
13   <text x="15" y="30" fill="white">SVG</text>
14   <text x="70" y="30" fill="black">Translated</text>
15 </g>
```

Notice the change of the second <text> element to be **Translated** and the insertion of the transform attribute in the <g> element!

You should see in the browser the following (if you had only the two <g> elements):



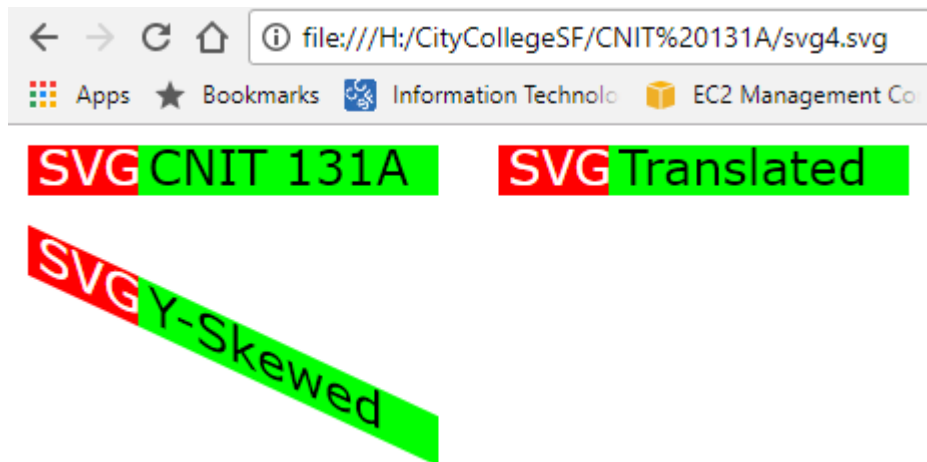
7) In the other group element you inserted (remember that you copied 4 times the first group element we coded), change the content of the last <text> element to be **Y-Skewed** instead of **CNIT 131A** and insert the following attribute in the <g> element:

```
transform="translate(0,35),skewY(25)"
```

So, you are moving the whole group down (by 35) and then y-skewing it by 25 degrees – the code would look like this:

```
16 <g font-size="1.5em" font-family="Verdana" transform="translate(0,35),skewY(25)">
17   <rect x="10" y="10" width="55px" height="25px" fill="red" />
18   <rect x="65" y="10" width="150px" height="25px" fill="lime" />
19   <text x="15" y="30" fill="white">SVG</text>
20   <text x="70" y="30" fill="black">Y-Skewed</text>
21 </g>
```

After you save and refresh, that's what you would have in the browser (if you had only the 3 <g> elements):



8) Let us now work with the third <g> element! For this one, you will change the second <text> element to be **Rotated** instead of **CNIT 131A** and you will insert the following attribute in the <g> element:

```
transform="translate(235,50),rotate(35)"
```

Here is the code for this group:

```
2 <g font-size="1.5em" font-family="Verdana" transform="translate(235,50),rotate(35)">
3   <rect x="10" y="10" width="55px" height="25px" fill="red" />
4   <rect x="65" y="10" width="150px" height="25px" fill="lime" />
5   <text x="15" y="30" fill="white">SVG</text>
6   <text x="70" y="30" fill="black">Rotated</text>
7 </g>
```

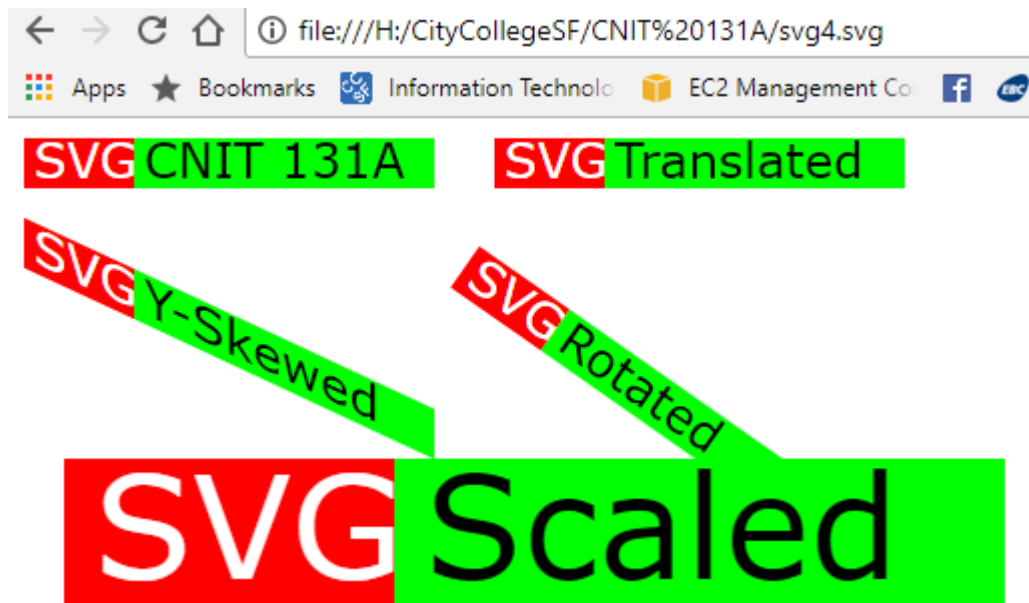
9) Let us now work with the fourth and last <g> element you pasted in your code! For this one, you will change the second <text> element to be **Scaled**, instead of **CNIT 131A** and then insert the following attribute in the <g> element:

```
transform="translate(0,140), scale(3)"
```

Here is the code for this group:

```
28 <g font-size="1.5em" font-family="Verdana" transform="translate(0,140),scale(3)">
29   <rect x="10" y="10" width="55px" height="25px" fill="red" />
30   <rect x="65" y="10" width="150px" height="25px" fill="lime" />
31   <text x="15" y="30" fill="white">SVG</text>
32   <text x="70" y="30" fill="black">Scaled</text>
33 </g>
```

Now, if you save **svg4.svg** and show in the browser, you would get something as:

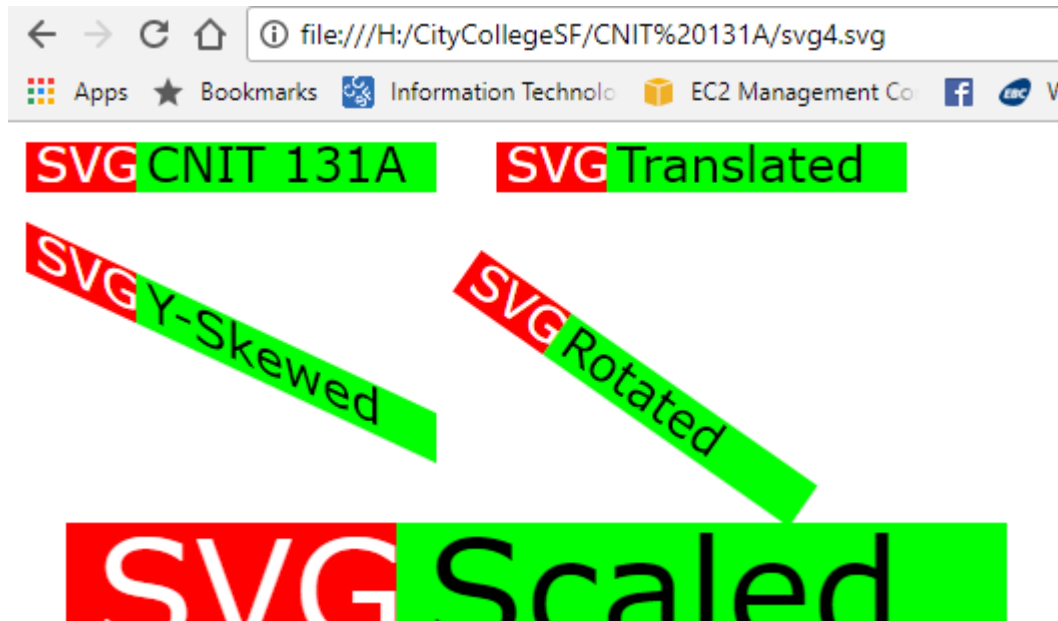


You might think that the SVG Scaled shape was on top of part of the SVG Rotated and you would rather bring the SVG Scaled a little bit down. What would be the value you would change?

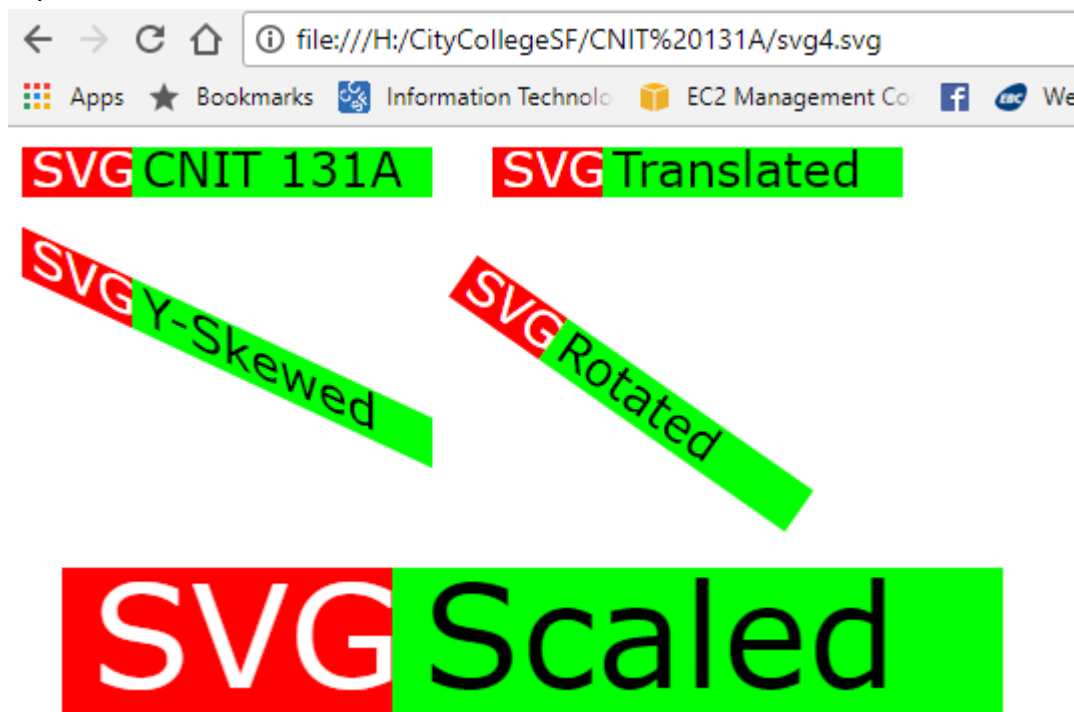
If you look at the code shown in step 9, you might realize that you should change the **translate** value of the **transform** attribute of the **<g>** element as this is the part that is shifting the shape down and/or to the right. In the case of the SVG Scaled group, the **translate(0,140)** is shifting 0 (zero) to the right and it's shifting 140 down. So, we now would simply change these values to be, for example **translate(0,170)**. See the change below (the changed code is underlined with red):

```
<g font-size="1.5em" font-family="Verdana" transform="translate(0,170),scale(3)">
  <rect x="10" y="10" width="55px" height="25px" fill="red" />
  <rect x="65" y="10" width="150px" height="25px" fill="lime" />
  <text x="15" y="30" fill="white">SVG</text>
  <text x="70" y="30" fill="black">Scaled</text>
</g>
```


If you make that change, when you save the file and refresh the browser, that's how it would look like:



Oops! It seems that the SVG Scaled is being cropped?!? If you analyze the code again, from the top, you will notice that the <svg> element has a certain pre-determined width and height and that's the space available for us to draw our shapes. So, if you want to add more graphics/shapes, you need to create more space, then you need to change the values of the width and/or height of the <svg> element. After changing the **height** of the <svg> element to be **350**, and even changing the **translate** of the SVG Scaled group to be **(0,190)**, here is what I got in my browser:



Now all the elements (groups) fit within my <svg> element!

Adding hyperlinks in SVG

The SVG specification does not provide native support for hyperlinks but this can be done by incorporating the **XLink** language into an SVG document. You do that by simply including its namespace in the root <svg> element.

The XLink language supports the href attribute (the same of the <a> tag) used to specify a target link but, as we are using XLink via namespace, we need to include the namespace prefix – for example xlink:href

When you add the XLink to an SVG document, you also enables other types of links such as:

- To specify the location of a remote JavaScript document – when used in a <script> element
- To specify the identity of a defined group to copy – when used in a <use> element
- To specify the identity of a path to align text on – when used in a <textPath> element

Copying multiple instances of a group with a <use> element is easier and more efficient than repeating all the elements for each instance like we did in the previous example.

We will create some different links.

1) Open **svg4.svg** in your web editor

2) Delete all the elements between <svg> and </svg> and save the file as **svg5.svg**

3) Modify the <svg> root element to include the XLink namespace that would be something as:

xlinkns:xlink = “http://www.w3.org/1999/xlink” – see this namespace included in the <svg> element below:

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
3   xmlns:xlink="http://www.w3.org/1999/xlink" width="500px" height="350px">
4
5 </svg>
```

4) Let us now add some elements to create different hyperlinks – see the code below with all the code added:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="500px" height="350px">
  <a xlink:href="http://www.ccsf.edu">
    <text x="10" y="150">City College of San Francisco</text>
  </a>
  <script type="text/javascript" xlink:href="script.js" />
  <path id="mypath" fill="none" stroke="none" d="M 40 120 Q 50 -20 390 80" />
  <text fill="blue" font-size="2em">
    <textPath xlink:href="#mypath">
      Creating a Curved Text with SVG
    </textPath>
  </text>
  <defs>
    <g id="redalert" font-size="2.5em">
      <circle r="30px" fill="red" />
      <text x="-10" y="20" fill="white">!</text>
    </g>
  </defs>
  <use x="100" y="100" xlink:href="#redalert" />
  <use x="200" y="100" xlink:href="#redalert" />
  <use x="300" y="100" xlink:href="#redalert" />
</svg>
```

Notice that the namespace of the **XLink** language is setting up the **xlink** as a prefix (**xmlns:xlink**) and that prefix will then be used every time you set a hyperlink or any type of link in this XML document.

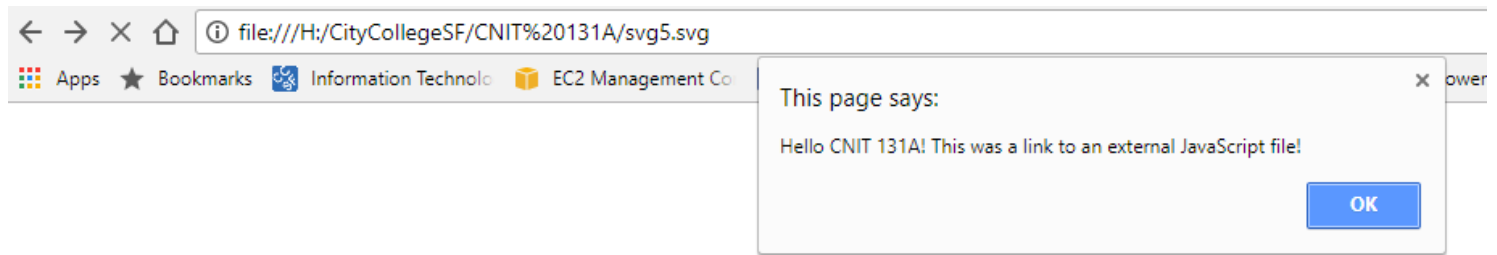
The first element <a> element, right after the <svg> element is creating a hyperlink to the City College website (http://www.ccsf.edu) using the text City College of San Francisco.

The second element is a <script> tag and it's referring to the script.js file – this external JavaScript has only an alert command that generates an alert box with the message that you see inside the parentheses of the alert command in the script.js file.

The third element <path> is creating a path with id mypath that is then used by the <text> element that follows it – notice that the <textPath> element, inside the <text> element refers to (using xlink:href) to the mypath id value of the <path> element.

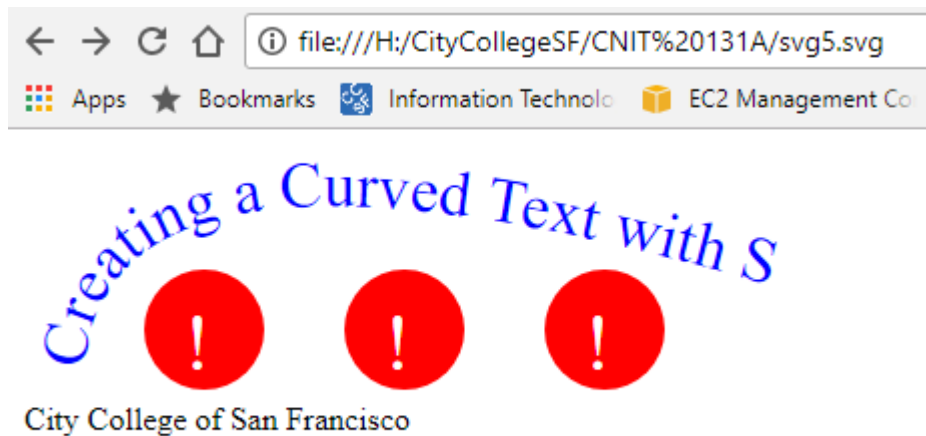
The <defs> element is defining a constant that will then be reused/repeated 3 times with the three <use> elements you see after the <defs> element. The <defs> element is defining a red circle that contains the text of just an exclamation mark (!). The circle is color red and the text is color white. The font-size of the text is determined by the <g> element (that is defining a group) that is inherited by any child element of the group.

If you save the file and open in the browser, you should have first something as:



City College of San Francisco

You see the City College of San Francisco text that belongs to the first <a> element that was defining and you also see the alert box that is coming due to the command from script.js file that was processed. Remember that XML is also being parsed by the browser from top to bottom and that is why you still do not see the other elements defined after the <script> element. If you click the OK button of the alert box, you should get something like:



Embedding SVG in an HTML document

We already know that SVG images can be easily scaled to any dimension without loss of quality. The width and height attributes used in the <svg> element determine the size of the SVG. We can add a **viewBox** attribute to define the viewable area of the SVG and then we can make the image to appear scaled down. This viewBox attribute can then be used to reduce the width and height to the desired scale. This attribute describes the x,y coordinates of the top-left and bottom-right corners of the viewable area. For example the viewBox="0,0 300,200" describes a viewable area that is 300 pixels wide and 200 pixels high and this area could be scaled down to 50% by specifying width="150px" and height="100px".

Once you scale the SVG image to the desired size, you can embedded it into the HTML document using the **<object>** tag that will specify the same dimensions as the width and height attributes of the <svg> element. The type attribute of the <object> tag will be type="image/svg+xml" and the data attribute will contain the URL (path) to the .svg file.

Let us then embed the SVG file we created in `svg4.svg`.

1) Open the **svg4.svg** file in your web editor and save that file as **svg6.svg**.

2) Take a look at the value given to the width and height attributes of the `<svg>` root element – respectively those values are 500 and 350. We will then set the `viewBox` attribute, inside the `<svg>` element, to the overall image – so, your `viewBox` attribute will be:

`viewBox="0,0 500,350"`

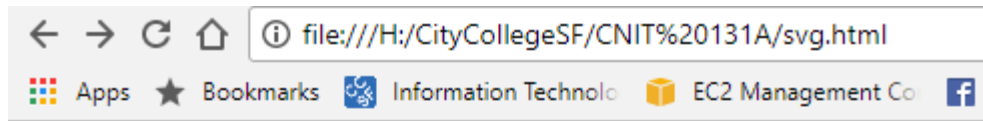
3) Modify the width and height attribute of the `<svg>` root element to the desired size you want to present – see the code below showing that the width was set to 250px and height was set to 175px (50% scale down) and also the `viewBox` attribute set:

```
<?xml version="1.0" encoding="UTF-8"?>
<svg version="1.1" baseProfile="full" xmlns="http://www.w3.org/2000/svg"
xmlns:xlink="http://www.w3.org/1999/xlink" width="250px" height="175px" viewBox="0,0 500,300">
```

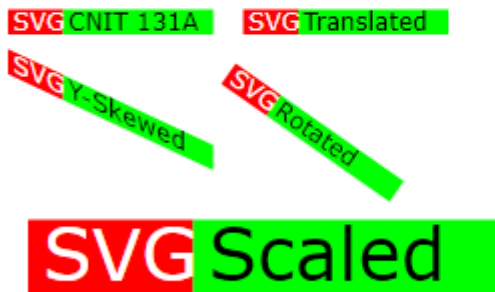
4) Open the file **svg.html** and in the body, right between the `h1` element and the `p` element you see in that page, add the **<object>** element to present the **svg6.svg** – see the code below of the **svg.html** file with the **<object>** element inserted:

```
1  <!DOCTYPE html>
2  <html lang="en">
3  <head>
4    <title>Inserting SVG</title>
5    <meta charset="utf-8">
6    <meta name="description" content="working with SVG in the HTML">
7  </head>
8  <body>
9    <h1>Here is my SVG</h1>
10   <object width="250px" height="175px" type="image/svg+xml" data="svg6.svg"></object>
11   <p>Just a paragraph after the SVG image is presented</p>
12 </body>
13 </html>
```

5) Save the **svg.html** file, make sure that the **svg6.svg** file has also been saved and open the **svg.html** in the browser. You should get something as:



Here is my SVG



Just a paragraph after the SVG image is presented

Note: Commas (,) are optional in the viewBox statement – you can separate the coordinates by white space – but I believe it's more clear and more readable writing each x,y pair separated by comma (,).

Of course, you can simply add an SVG image to your HTML page by using the img tag such as:

```

```

Look at **svg1.html** and see the **svg6.svg** inserted as an image. You can even use the width and height attributes in the img element to scale down or up the image.

Note: you use the <object> when you want to integrate an external SVG file into an HTML document without the limitations of treating the SVG as an image. Remember that the <object> tag can also be used to embed files of any arbitrary type (a browser plug-in or extension might be necessary to interpret that file type). When you use the <object> element you might make the SVG available for older browsers that cannot display SVG directly (as long as they have an SVG plug-in).

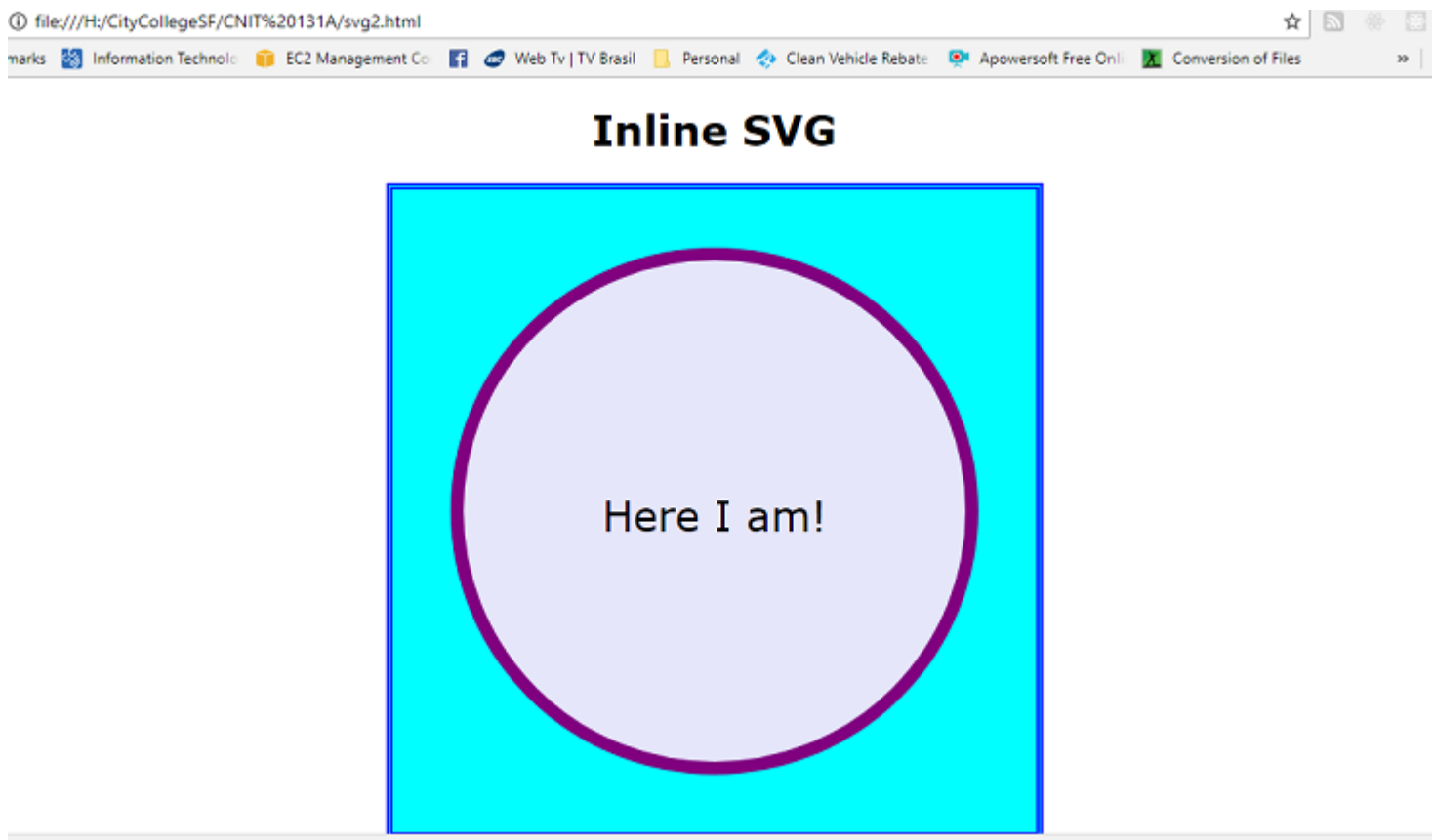
Another note: Prior to the <object> element, some browsers used the non-standard <embed> for the same purpose. As it has been now adopted into the W3C standards, you can use <embed> instead of <object> if you are concerned about supporting older browsers – you can use <embed> as a fallback content inside the <object> element. In the <embed> the source data file is specified using the src attribute, not the data attribute and the <embed> element cannot have any child content (there is no fallback in case the <embed> fails)

Other ways to use SVG file in HTML

The background-image CSS property accepts a URL to an image file and that image file can certainly be a .svg file and you can set the size of the SVG to be used using background-size CSS property. Remember that you can also use background-repeat, background-position to set respectively the repetition pattern and the position of the background image.

You can also use SVG files in CSS as a list-image or border-image (CSS property used to create fanciful borders).

Inline SVG in HTML5 – If you are using HTML5, you are able to include the svg element directly in the HTML markup. This technique is supported in all major desktop web browsers for versions released in 2012 and later and most of the latest mobile browsers. In the case of HTML5 you do not need to declare the namespace as the HTML parser will automatically recognize the <svg> element and all its children. By default, the SVG will be positioned with the inline display mode (it's inserted within the same line as the text before and after it) and it will be sized based on the height and width attributes of the <svg> element. By using CSS, you can change the size by setting the width and height properties for example and use display, margin, padding CSS properties to change the position. See an example in **svg2.html** file. Look at the code of this file and when you open in the browser, it should look something as:



Notice how we used CSS properties targeting the <svg> element directly (svg as selector) and styles defined for the parent (body element) are inherited as well. Notice that we also used circle as selector to target the <circle> SVG element.

Extra Resources

There are some software tools that make it easier to create graphics and easier to process your files so they are ready to be deployed on your web server.

A lot of developers, especially the ones that do not have a strong background in graphic design, start with someone else's art (ready-to-use SVG), especially for icons. As a developer, you just need to make sure of the license, before using any graphical art – by default, creative works are “all rights reserved” which means that the absence of copyright information does not mean it is public domain.

Here are some list of websites that should help you start your search for SVGs:

Open Clip Art Project – The [OCAL Project](#) is the oldest and probably the largest repository of SVG content and some are available either through Creative Commons or public domain licenses. This project is integrated with the [Flaming Text ImageBot](#) which is a graphic editor that allows you to tweak some SVG style properties online.

Pixabay – The [Pixabay](#) includes photos, illustrations, and vector graphics. Some of the vector graphics are stored in Adobe Illustrator format and you would need software to convert it to SVG.

Wikimedia Commons – The [media repository for Wikipedia](#) images, audio, and video in a wide variety of formats. All files available under some sort of “copyleft” license – some require attribution or are restricted to noncommercial use and detailed license information is available on each file's catalog page.

Iconic – It's a commercial SVG icon library but they offer a set of about 200 icons free to use (MIT license – just make sure that the license information is available with the code) – the [Use Iconic Set](#) includes most common user interface buttons in single-element icons that you can style to any color you choose

The Noun Project – Icon-focused library – the objective is to create a visual language for clear international communication. Access to their library is via monthly subscription.

There are other ones that will not be listed here!

You can also use certain software to produce the SVG graphic yourself such as:

Adobe Illustrator – since 1991 – has been consistently at the cutting edge of vector graphics support for many decades. Just remember that SVG is not the native format of this software.

Inkscape and Sodipodi – Sodipodi was one of the earliest SVG editors (inspired by Adobe Illustrator but having SVG as native format). [Inkscape](#) started as a branch of Sodipodi and now is the more actively developed program

Microsoft Visio – Used much more for designing charts and diagrams.

Google Docs – The [drawings](#) uses SVG natively but only supports a basic set of SVG features

SVG-edit – The [SVG edit](#) is another online SVG application originally sponsored by Google, runs in your web browser either from a downloaded file or directly from the web server. The documentation is not so good

Draw SVG – The [Draw SVG](#) is another online application that is more complete and well documented

Building Accessible SVG – Article from [Deque about Creating Accessible SVGs](#)

Get Wave (from Creative Labs) – You can use [Get Wave](#) to create nice shapes (waves) to use as background-image or even as an image – Look at an example [developed by the instructor](#), where you will see the document-level CSS using the bg.svg file that was created from the Get Wave website. I had to include the lines before related to the XML part to save the file as bg.svg and then use as a background-image.

Formito – This is a [free favicon maker](#) that can produce favicon in SVG or PNG format.

Final Note:

Some of you might think that you should not know how to manually build SVG and, in this case, you should read this article (mini-tutorial) from [TutsPlus on How to hand-code svg](#).

Something about this?!? <https://css-tricks.com/use-and-reuse-everything-in-svg-even-animations/> for the <use> element???

Editors for SVG - <https://css-tricks.com/browser-based-svg-editors/>

Writing XML

Every time you write a document using XML, you will need to follow the “XML grammar”.

You can write XML in any web editor or plain text editor. There are also some XML-specific editors that have been created and some of those were referred to you in the course. It’s assumed that you know how to use a web editor and that you know how to create a new document, save it, rename it, delete it, etc.

When you create an XML document, it needs to be saved with the .xml extension (as you saw in the examples in the first lecture you read).

Open in your web editor the file **books1.xml**. The first line of the XML document is `<?xml version="1.0"?>` - this is called XML declaration that is used to declare which version of XML you are using. The next line you see the first tag - `<books>` - this tag is called the root element and in any XML document you can only have one root element. Then you see the tags `<book>` and as children of this tag you see the `<name>`, `<author>`, and `<year>` tags. The `<book>` element is an extension of the XML because it is used to group data that will describe one book – that’s why you see 3 different `<book>` elements. The `<name>` element contains an attribute called `language` which is just being used to store the language of that book. Attributes are used to include additional information to the element without exactly adding text to the element itself.

Here are some basic rules to write a valid XML document:

- **A root element is required** – only one root can be found in any XML document and the only pieces of XML that are allowed preceding the root are comments and processing instructions (we will talk about those ones later).
- **You need to close tags** – every element you create in XML must have a closing tag. Later we will talk about empty elements and those can have a forward slash (/) before the final >
- **Elements need to be properly nested** – if you look again at the **books1.xml** code, you will see that the `<book>` element was started but then I needed to start the element `<name>`. So, I had then to close `</name>` before I would be able to close `</book>`
- **XML is case sensitive** – elements named Book, BOOK, book are considered different, separate and unrelated to each other
- **Values must come between quotation marks** – when coding an attribute, the value of the attribute must always be enclosed in either matching single or double quotes

Elements, Attributes, Values

An XML element can contain text, attributes, and other elements – again, in **books1.xml**, you can see that the element `<book>` contains other elements `<name>`, `<author>`, `<year>` and the element `<name>` contains text (the title of the book) and also an attribute `language` with the value between double quotes.

The information contained in an attribute is generally considered metadata which means information ABOUT the data in the element and NOT data itself. Any element can have as many attributes as necessary, but those attributes need to have unique names.

Extra white space is ignored by the XML processor the same way the browser ignores extra spaces in HTML and you can certainly use extra spaces to organize your XML document (like what you see in **books.xml** and **books1.xml**).

Notice that the declaration tag (the first one at the top of the XML document) starts with `<?` And ends with `?>`. Any tag that begins with `<?` And ends with `?>` is called processing instruction – in that case, it's giving the "instruction" of which version of XML is being used. You will see that processing instructions tags are also used to specify the CSS (stylesheet) that you want to use and for other things as well. In that first processing instruction tag, you could also have the encoding attribute to specify the character encoding that is being used such as:

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

Some tips:

- Element (and attribute) names should be short but descriptive.
- Element and attribute names must begin with a letter, a colon (:), or an underscore (_) and you should never begin a name with the letters *xml* (no matter which combination of lower or uppercase) as these letters are reserved and cannot be used
- Although you can have colons, hyphens and periods within the name of elements and attributes, it's recommended that you avoid those as they are used for specific circumstances such as: identification of namespaces, subtraction, object properties respectively)
- No element is allowed outside the opening and closing root tags except processing instruction tags
- Closing tags are NEVER optional
- Names do not need to be in English or even having characters of the Latin alphabet but if your software does not support those other types of characters, they may not display or not be processed properly
- You can nest as many elements as you need to describe your information just remember to close their tags in the appropriate order as mentioned before. It's good practice to indent the children elements to make the XML reading be easier for humans and most web editors will do that for you (look how the elements `<name>`, `<author>`, and `<year>` are indented in relation to the element `<book>` in the **books.xml** or **books1.xml**)
- You cannot have two attributes, in the same element, with the same name and the values of attributes must always be enclosed in quotes (single or double quotes) – if, by any chance, you have an attribute that the value contains quotes, use the other type of quotes to identify the whole value – for example: suppose that I have an attribute called `comments` and the value is *she said, "the war is over"* – the way to code that attribute would be `comments = 'she said, "the war is over"'`

Empty Elements

Those are elements that do not have content of their own. They will have attributes to store data about the element. For example, you might have the image of a book. You can code this using one of the methods shown below:

```
<book_image file="book1.jpg" w="250" h="180" />
```

OR

```
<book_image file="book1.jpg" w="250" h="180"></book_image>
```

Both methods are equivalent!

Comments

You can write comments in an XML document and the comments will not be parsed by the processor. The way to write comments in XML is exactly the same as in HTML. You start with `<!--` and then close the comment with `-->`. You can include spaces, text, elements, line breaks, etc but you should not use double hyphen (`--`) within the comment itself. You cannot nest comments. When you are testing your XML code, you can comment some areas (some elements)

Entities

There are many letters and symbols that can be inserted into an HTML document by using entities but in XML there are only five predefined entities:

- `&` - to create the ampersand character (&)
- `<` - to create the less than sign character (<)
- `>` - to create the greater than sign character (>)
- `"` - to create the double quotation mark character ("")
- `'` - to create a single quotation mark character ('')

Those entities should be used whenever you want to write those specific characters in your XML document and the reason they were created was exactly so the processor does not make confusion with the text `>` and the character `>` to close the name of an element. As we have seen before when talking about attributes, it's possible to write double and single quotes directly without using the entities but you need to be careful to match exactly the right quotation marks you are using.

You can create additional entities to be used in your XML and we will see how this is done later in the course.

Displaying Elements as Plain Text

Imagine that you want to write something about XML, for example, a tutorial. In that case, you would need to write your XML in a way that the XML processor would not interpret those ones that you are writing and that you want to show as text. To do this, you will enclose the information you want to present as regular text in a CDATA section. For example:

```
<?xml version="1.0"?>
<xml-tutorial>
  <tags>
    <![CDATA[
      <books>
        <book>
          <name language="English">Intro to XML</name>
          <author>Joseph Smith</author>
```

```
        <year>2004</year>
    </book>
</book>
    <name language="Russian">XML and CSS</name>
    <author>Lidia Juniper</author>
    <year>2003</year>
</book>
</book>
    <name language="English">XML vs JSON</name>
    <author>Albert Jones</author>
    <year>2010</year>
</book>
</books>
]]>
</tags>
</xml-tutorial>
```

Between `<![CDATA[` and `]]>` you will type the XML code that you want to be shown as regular text and not be processed by the XML processor.

The example above can be found in the file `xmldata.xml` and if you open this file in a good web editor (a color-coded one), you will notice that when you start the `<![CDATA[` the color will change until you finish this area with `]]>` like the code above is showing (the code was extracted from Notepad++). This change in color is exactly to show that the XML code inside that CDATA area will not be processed.

You can also use CDATA to enclose HTML or JavaScript so that those are not parsed by the XML processor. You cannot nest CDATA sections.

CDATA means Character Data, meaning that it will not be interpreted by the XML processor. It's different than PCDATA that means Parsed Character Data.

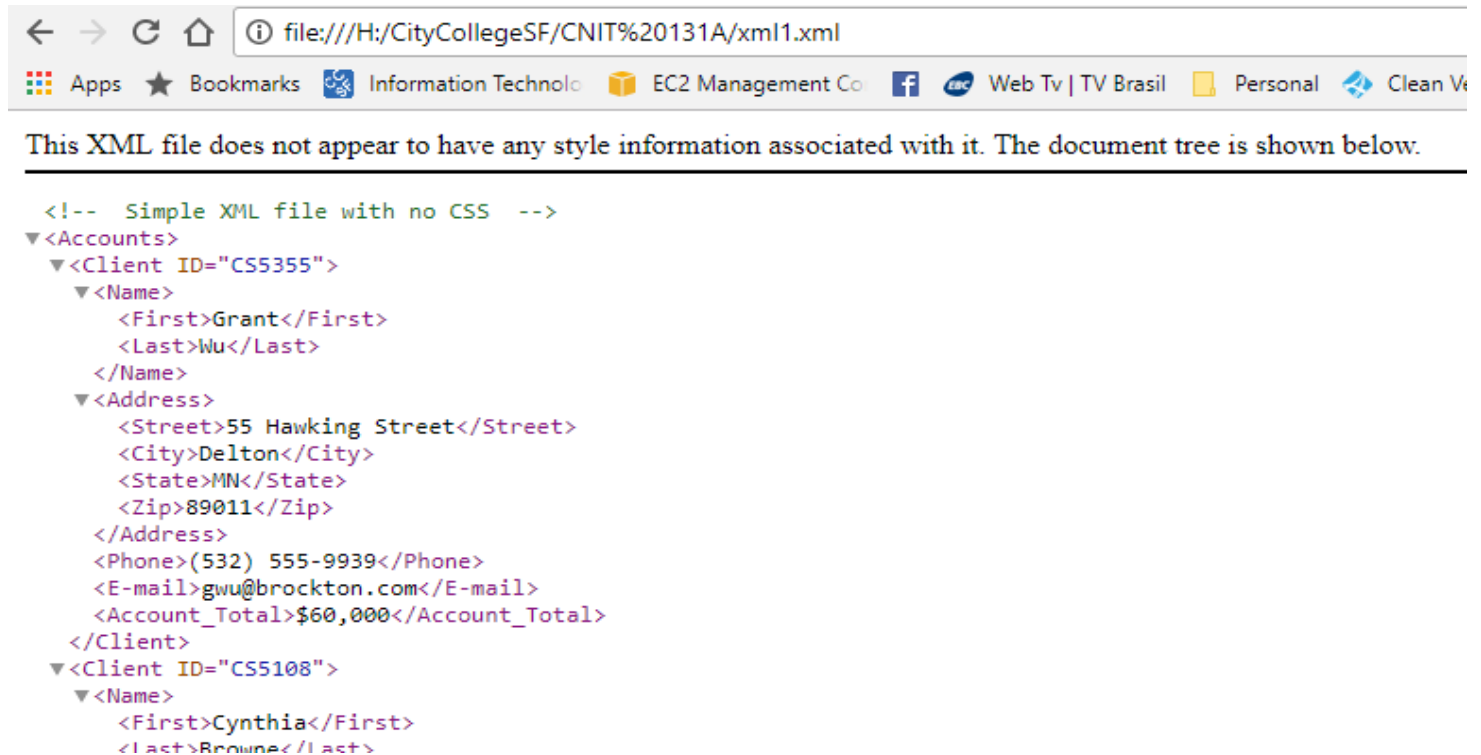
The special entities, such as `<`, will be ignored if found in the CDATA section and will be presented as `<`; and not converted/processed to `<`. It's important to notice that if you need to write `]]>` inside the CDATA and it's not to close the CDATA, then you must write the `>` as `>`;

Applying CSS to XML

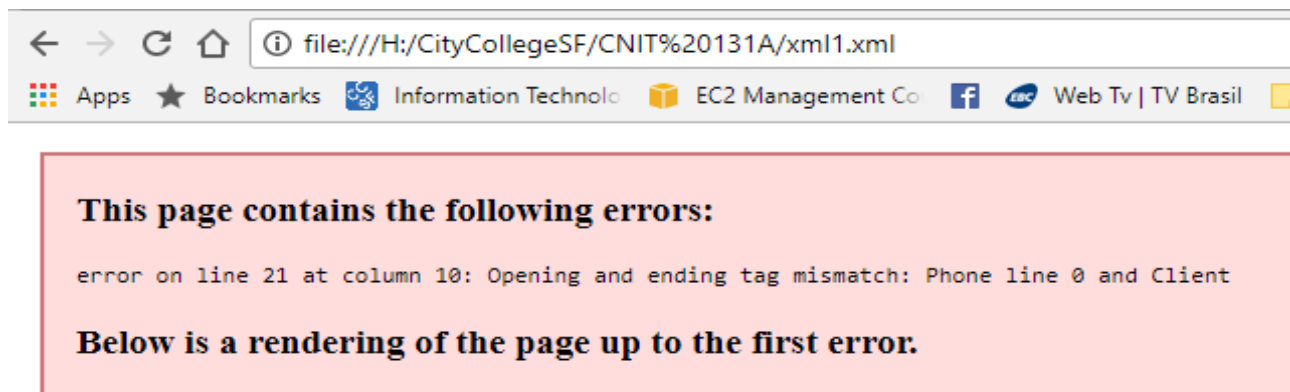
How to insert CSS file in XML

XML documents are not presented in the browser as HTML documents are.

Open the XML file **xml1.xml** in the browser – you should see something as:



As you can see, the XML document is presented with their elements (tags). Now, if you have an error in your XML – for example, you do not close the **<Phone>** element (if you delete from **xml1.xml** the **</Phone>** closing tag), depending on the XML Editor you are using, you might receive an error message right when you try to save the new file. But even if you ignore the error message from the web editor, you should see something as below in your browser:



Grant Wu 55 Hawking Street Delton MN 89011 (532) 555-9939 gwu@brockton.com \$60,000

Notice that the message, in Google Chrome, is very specific – **“Opening and ending tag mismatch”**. As soon as you correct the file and save it, you should be able to open it like the first image shown in the previous page.

Even when the XML code is valid, the way that the XML document is presented in the browser, is not very friendly, not very easy to be read by the user.

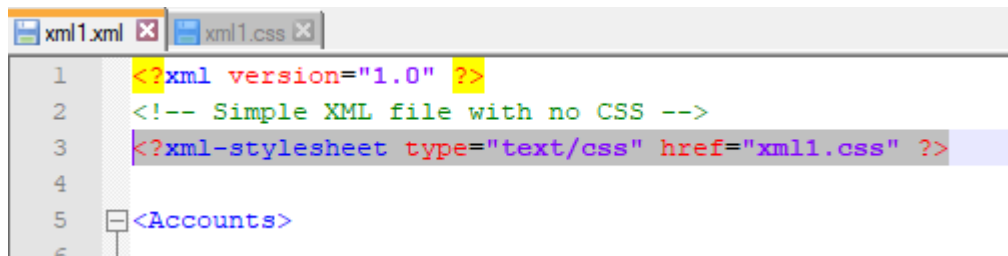
One of the simplest ways to make an XML document more readable is by using CSS. The same CSS properties, that you would use for an HTML document, can be used for an XML document.

Look at the **xml1.css**. As you can see, the CSS selectors are referring directly to the elements used in the XML document and even using some well-known pseudo classes. We know that CSS is case-sensitive, so observe that the name of the elements, in the CSS selector, match the way they were defined in the XML document.

But how do we link an external CSS file to an XML document? We need to insert the following line, in the XML document – it can be inserted right after the comment:

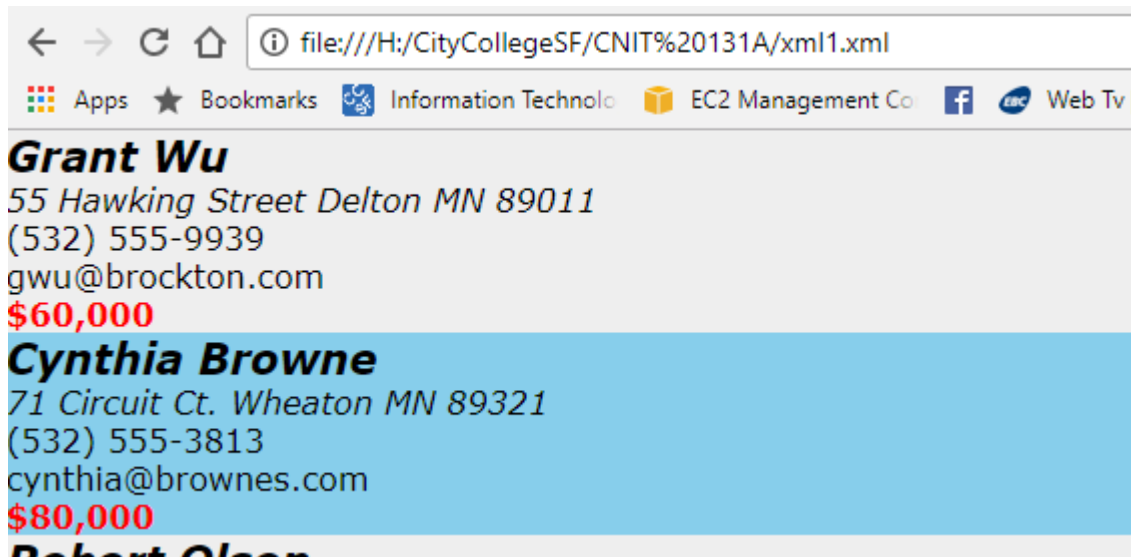
```
<?xml-stylesheet type="text/css" href="xml1.css" ?>
```

Look the top of **xml1.xml** after the link to the external CSS **xml1.css** was inserted:



```
1 <?xml version="1.0" ?>
2 <!-- Simple XML file with no CSS -->
3 <?xml-stylesheet type="text/css" href="xml1.css" ?>
4
5 <Accounts>
6
```

If you now save **xml1.xml** and open it in the browser, you should see something as:



Note:

If you do not apply **display:block;** to the XML elements, the elements, when presented using CSS, will be all in one single line wrapped by the width of the browser – the elements are, by default, considered inline elements.

Tools:

There are some XML Editors that you can check out:

- [EditiX](#) – free for 30 days as evaluation – for Windows/Linux/Unix/Mac
- [XMLmind](#) – free download – for Windows/Linux/Unix/Mac
- [XMLwriter](#) – free for 30 days as evaluation – for Windows
- [XML Copy Editor](#) – free download – for Windows/Unix/Linux/Mac

As we are talking about editors, let me share some other tools:

- [XML Validator online](#) – you can upload a file or you can write the XML in the space provided
- [XML Validator from W3Schools](#)
- [XML Validator online](#) – validates XPath, Schema, DTD (it mentions it is in Beta mode)
- [XML Schema Generator](#) – you type the XML in the space provided and it will give the Schema to you
- [XSD/XML Schema Generator](#)
- [Conversion Tool from DTD to XML Schema](#)

XSLT

You have now the basic rules to write a good and valid XML document and you also know how to use CSS to format the data of an XML document to present it to the user.

There is another way to format XML data and originally the details for formatting XML documents came in a specification called XSL (eXtensible Style Language) but later W3C divided XSL into two pieces: XSLT (T = Transformation) and XSL-FO (FO = Formatting Objects).

When you apply XSLT to an XML document, you might end up with another XML document or an HTML document, or even any document type you would need. XSLT will be used to basically analyze the content of the XML document and then take specific actions – for example, you can use XSLT to reorder the output of the XML data following certain criteria, or you can choose which pieces of data will be displayed, etc.

XSL-FO is used generally to format XML to print output – for example, going directly to a PDF format. We will talk about XSL-FO later.

Using XSLT to transform XML

Let us open the **books2.xml** and **books.xsl** files in the web editor – you will find those files among the files you downloaded for this course.

Here is a basic explanation of how the transformation process happens:

- 1) The process starts with an XML document that contains the data and an XSLT style sheet document that contains the rules to transform the data. The XML document will have a link to the XSLT style sheet document – in the books2.xml, the link statement is found in line 2.
- 2) It is necessary to have an XSLT processor so the transformation can happen or a browser that supports XSLT (most current browsers support XSLT)
- 3) The transformation, as mentioned before, can produce any type of file but most of our examples will produce an HTML file. When you open this XML file in an XSLT processor or a browser, the process basically is notified to perform the XSLT transformation before displaying the data.
- 4) The data of the XML document will be analyzed and converted to a node tree (a hierarchical representation of the XML document) – see, here below, an example of a node tree just for up to the first book element:

```
books      ..... root node
  book      ..... element node
    name     ..... element node
      language ..... attribute node
        English ..... text node
      Intro to XML ..... text node
    author   ..... element node
      Joseph Smith ..... text node
    year     ..... element node
      2004     ..... text node
```


Notice that in the tree, a node is one individual piece of the XML document: an element, or an attribute, or text content

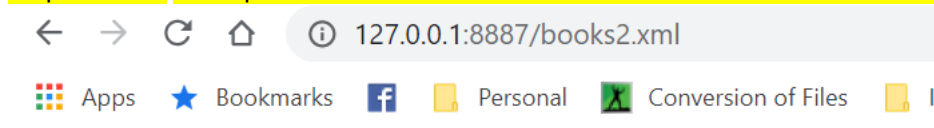
- 5) Once the processor identifies the nodes in the XML file, it will grab the XSLT style sheet document that was linked (“mentioned”) in the XML document for instructions on what to do with the XML data. The instructions are contained in templates that are like functions of a programming language. The template has two parts: label – to identify the nodes in the XML document to which the template will apply; instruction – the actual transformation that will happen (the instructions can output or further process the nodes). You can also find literal elements that should be output as is.

The transformation begins by processing the root template that needs to exist in every XSLT. The root template is defined with `<xsl:template match="/">` and it will be closed with `</xsl:template>`. You will see those two lines in the books.xml code. You can certainly have other templates (sub-templates) inside that main template that can be applied to other nodes in the XML document. The transformation will continue up to the last instruction of the root template.

The transformed document will be either saved to another file or displayed in a browser or both. So, if you open **books2.xml** in the browser, you will probably get this output:

NOTE:

You will need to open **books2.xml** from a secure server otherwise the XSL file will not be called, and no transformation will be processed – so upload both files to a secure server or use web server for chrome like I did showing in the image below



Books

The Intro to XML was written by Joseph Smith

Some interesting notes about XSLT

XSLT are text files and are always saved with .xml extension

XSLT uses the XPath language to identify nodes. We will learn about XPath later in the course.

Writing an XSLT Style Sheet from scratch

You will use the same web editor you have used to open and write your XML documents. Open the web editor with an empty file. The first line of any XSLT file is exactly the same first line an XML document has

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

The next line will be the one shown below that specifies the namespace for the style sheet and also declares its prefix (*xsl* right after *xmlns:*)

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

Leave a couple of lines that will be used for you to create the XSL instructions and at then type the line shown below that will close the style sheet:

```
</xsl:stylesheet>
```

Although we use two words to refer to the XSL file – style and sheet – those two are not separated in the two declarations shown above. Do not worry about namespaces right now, we will learn more about that later in the course.

Now, we need to create the root template which will define the set of rules that will be applied to the root node of the XML document. You will now put your cursor between the opening <xsl:stylesheet...> and the closing </xsl:stylesheet>, it can be one or a couple of lines below the opening <xsl:stylesheet....>. You will then type:

```
<xsl:template match="/">
```

The forward slash (/) matches the root node of the XML document. After this line, you can leave a couple of lines empty and then you will close the root template with:

```
</xsl:template>
```

Every XSLT transformation needs the root template to start. If you do not include the root template, a root template built into the XSLT processor will be automatically used and this automatic one generally lists all the data in the XML document in plain text.

This should be what you have up to now in your XSLT file:

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

```
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
```

```
<xsl:template match="/">
```

```
</xsl:template>
```

```
</xsl:stylesheet>
```

Inside the <xsl:template match="/"> and </xsl:template> you will right the rules to output your XML data and if you want to have the output in HTML format, you will need to add the head, title, body, and other necessary HTML tags. By the way, you can even add CSS and JavaScript!

Before you start typing the rules, it's better to specify that you want your output in HTML format. So, right before the line <xsl:template match="/">, you will include the following:

```
<xsl:output method="html"/>
```

The method could be html, or xml, or text and if you omit this instruction, the default value will be XML.

After the <xsl:template match="/">, you will put the following HTML:

```
<html><head><title>Books Styled</title></head>
```

```
<body><h1 style="font-style:italic; color:red;">Books</h1>

<p>  The <xsl:value-of select="books/book/name"/> was
written by <strong><xsl:value-of select="books/book/author"/></strong></p>

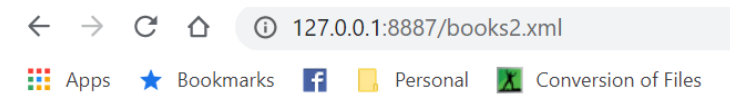
</body></html>
```

NOTES:

- We are including inline CSS using the style attribute in the <h1> element and in the element.
- We are also including other known HTML tags such as the to have the data written in bold.
- We **NEED** to use the forward slash (/) to close the self-closed tag because we are following the XML rules that every tag needs a closing (even when we are writing XSLT files).
- Of course, we are including all the minimum necessary HTML tags to compose the HTML output as the HTML needs also to be valid – it's assumed you know the basics of HTML markup code (at least the <html>, <head>, <title>, and <body> tags)

Save this new XSLT file as **bookstyle.xsl**.

If you change **books2.xml** to use the **bookstyle.xsl** file and then refresh the page that was presenting **books2.xml** in the browser, you should see something as:



Books



The Intro to XML was written by **Joseph Smith**

REMEMBER THAT YOU NEED TO CALL **books2.xml USING A SECURE SERVER AND ALL REFERRED FILES NEED TO BE IN THE SAME FOLDER!!!**

The <xsl:value-of select=

We are using this instruction in the XSLT file to select the **name** and the **author** of the book but, as you can see in the **books2.xml** file, we have more than one **book** element, right? The select expression for the **name** is `select="books/book/name"` which gives the direct path to the **name** element within the **books2.xml** node tree structure. The same happens with the selection of **author** when using `select="books/book/author"`. If the select expression matches more than one node of the XML document, only the first node value will be output. We will see later how to use a looping structure in your XSLT file to display or act on multiple nodes of the XML document. We will also study how to select certain elements only – for example: if you wanted to select the **book** node(s) that has(have) the attribute **language** equal to **"Russian"**.

When the select expression matches a node, the string (text contained by the node) will be the output and if the node has child elements, the output will include the text contained in those child elements as well.

When the select expression finds a number as content of the node, the number is converted to a string to be outputted and if that content is a Boolean value (true or false), the output will be the text “true” or the text “false”.

Looping over nodes

We will modify the **bookselect.xml** file so all the book nodes will be displayed in the HTML output.

If you open **bookloop.xml** in your web editor and you compare with the **bookselect.xml** you have built (also found among our files), you will see that the difference in the code between those files is (show below the difference with a highlight the code that has been changed/included):

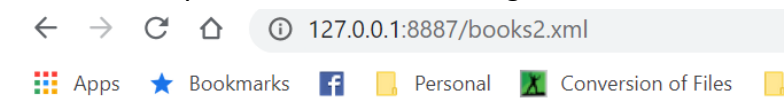
```

```

```
<xsl:for-each select="books/book">
  <p>The <xsl:value-of select="name"/> was written by <strong><xsl:value-of
select="author"/></strong></p>
</xsl:for-each>
```

The first change was to remove the `` tag from inside the paragraph (`<p>`) as we want to present the image only once, not for every book node! You will also include the `xsl:for-each select="x"` instruction where **x** is the expression that will give the node that will be selected. In the case of the expression **books/book**, all the **book** node(s) is(are) being selected inside the **books** root element. As you are already selecting the **book** node(s), then you had to change the expression in the **select** of `xsl:value-of` to point directly to the child node we want to select inside the **book** node (the **name** and the **author** nodes). The other inclusion was the closing `/xsl:for-each` to close the loop.

Now, you should change the **books2.xml** to point to the **bookloop.xml** file and then when you open **books2.xml** in the browser, you should see something as:



Books



The Intro to XML was written by **Joseph Smith**

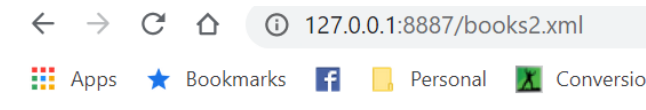
The XML and CSS was written by **Lidia Juniper**

The XML vs JSON was written by **Albert Jones**

As you can see, all the **book** nodes are being presented with their respective text from **name** and **author** nodes.

You should place the `xsl:for-each` loop instruction right before the rules that you want to repeat for each node that is found. If you would format the name and author nodes for example in a table, then you should open the table tag (`<table>`) and also the first row with the table headings (`<tr><th>Title of the`

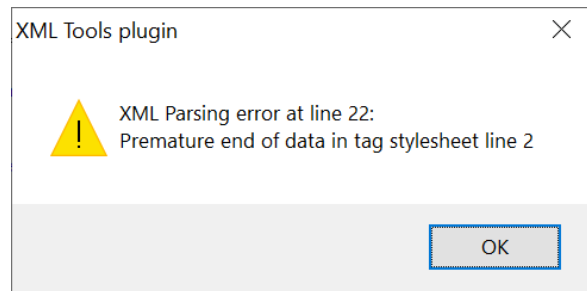
book</th><th>Author</th></tr>) before the `xsl:for-each` loop and close the table (`</table>`) after the closing of `/xsl:for-each` too. Then, the table rows and table cells (`<tr><td>`) as part of the process that will loop over the node (in our example, the `<tr>` would replace the `<p>` and then before the select of name and author nodes, you would put `<td>`, at the end of the select you would close that table cell with `</td>` and then, instead of the closing `</p>`, you would put `</tr>` to close each table row of the loop). You can try to have that table built and see if you would get something as:



Books



Title of the Book	Author
Intro to XML	Joseph Smith
XML and CSS	Lidia Juniper
XML vs JSON	Albert Jones



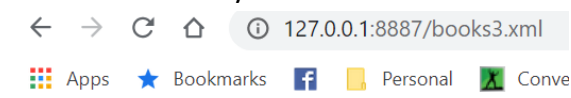
NOTE:

If you forget to close any tag in the XSL file, even the HTML tags, when you save, you will get an error message something like what you see here on the left side – notice that the line number of your code will be mentioned (in the image here it refers to line 22 of your code. You really need to close every tag when you are writing XML-type files and XSL is an XML-type file! So, do not open a `<td>` if you have not closed the previous one or do not try to save an

XSL file if you have not closed your paragraph with `</p>`, etc.! This message was shown in my simple Notepad++ editor that I was using in my Windows 10 laptop!

If you tried to format the name and author of the books in a table and was not able to, look at the **booklooptable.xsl** file found among the files you downloaded for this course and use it in your **books2.xml**!

Open now **books3.xml** and **bookloopfilter.xsl** files in your web editor. Using a secure server, open the XML file in the browser and you should see:



Books



Title of the Book	Author
Intro to XML	Joseph Smith
XML vs JSON	Albert Jones

Now you only see two books being presented, right? Well, if you pay attention to **books3.xml** you will notice that the **language** attribute is being coded directly inside the **book** element, not in the **name** element as it was in

books2.xml. Then, if you pay attention the selection expression in **bookloopfilter.xsl** is done in the `xsl:for-each` loop rule as:

```
<xsl:for-each select="books/book[@language='English']">
```

Once I'm pointing to the **book** node, the filter is being done using the **language** attribute that has the value of **"English"** (note that the double quote is not used around the English text in the expression of the filter to not make confusion with the double quotes around the whole value of **select**). That's why now you only see the books that have the attribute **language** as `language="English"`!

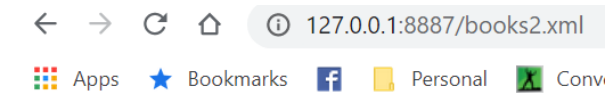
Using `<xsl:if test="expression">...</xsl:if>`

This rule can be used when you want to process a node only if a certain condition is met. The condition will be tested against the **expression** you will write for the **test** attribute.

We can use the `xsl:if` rule with the **books2.xml** where the **language** attribute is placed in the **name** node. Check the code in the file **bookloopif.xsl**. Notice that now, before building the table row of each **book**, there is the opening `xsl:if` rule and after building the whole table row (after `</tr>`) you see the closing `</xsl:if>` - this part is shown below (an extraction of the **bookloopif.xsl** file highlighting the if rule):

```
<xsl:for-each select="books/book">
  <xsl:if test="name[@language='English']">
    <tr><td><xsl:value-of select="name"/></td><td><xsl:value-of
select="author"/></td></tr>
  </xsl:if>
</xsl:for-each>
```

If you modify **books2.xml** to link the **bookloopif.xsl** to it, save the file and then open in the browser, you should see something as (same result showed before when using **books3.xml** and **bookloopfilter.xsl**):



Books



Title of the Book	Author
Intro to XML	Joseph Smith
XML vs JSON	Albert Jones

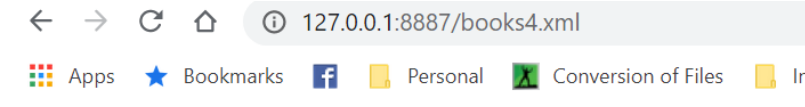
Using `xsl:choose`

The `xsl:if` instruction you saw in **bookloopif.xsl** only allows one condition and one resulting action. But you can use `xsl:choose` if you want to test different conditions and have the action according to each condition.

Look at files **books4.xml** and **bookloopchoose.xsl**. The XML file has one more book that has a **"-"** for the **year** node. The XSL file has a little bit more CSS for the HTML output just to put a border around the table and around the cells of the table (`<td>` and `<th>` tags). A third column was added to the row as you can see in `<th>Year</th>` to bring the year that the book was published.

The main change is found in the rules while building the table where you see the `xsl:choose`, `xsl:when`, `xsl:otherwise` rules being added as you will output the text **unknown** if a dash ("-") is found in the **year** node

When you present the **books4.xml** in the browser you will see:



Books



Title of the Book	Author	Year
Intro to XML	Joseph Smith	2004
XML vs JSON	Albert Jones	2010
Intro to XML & JSON	Felicia Logan	unknown

NOTE:

The closing of the table row `</tr>` had to be put AFTER the `xsl:choose` rule finished otherwise you would create an error in the parsing of the code as you would have two closing `</tr>` tags having only one opening `<tr>` tag opened! See the code below with the highlights on the main changes (not showing here the change in the beginning of the table where the first table row is formed with the table heading cells – adding the `<th>Year</th>`):

```
<xsl:for-each select="books/book">
  <xsl:if test="name[@language='English']">
    <tr><td><xsl:value-of select="name"/></td>
      <td><xsl:value-of select="author"/></td>
      <xsl:choose>
        <xsl:when test="year != '-'>
          <td><xsl:value-of select="year"/></td>
        </xsl:when>
        <xsl:otherwise>
          <td>unknown</td>
        </xsl:otherwise>
      </xsl:choose>
    </tr>
  </xsl:if>
</xsl:for-each>
```

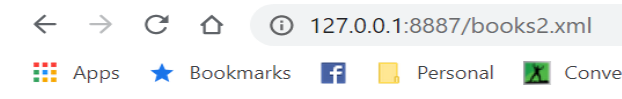
When you use `xsl:choose`, you can add more than one `xsl:when` condition but once a condition is found to be true, all the remaining conditions will be ignored – only the action of the first true condition will be processed!

Using `<xsl:sort>` to sort nodes

The nodes are presented in the order they were coded in the XML document. If you want to present them in a different order, then you will need to use the `<xsl:sort>` instruction.

Look at file **bookloopsort.xsl**. It's highlighted below the `xsl:sort` instruction in the code. Notice that this instruction is a self-closed tag. If you change **books2.xml** to use **bookloopsort.xsl**, you should get this result in the browser when presenting **books2.xml**.

REMEMEBER THAT YOU NEED TO USE A SECURE SERVER AND BOTH FILES NEED TO BE INSIDE THE SAME FOLDER OF THAT SECURE SERVER!!!



Books

Title of the Book	Author
XML vs JSON	Albert Jones
XML and CSS	Lidia Juniper
Intro to XML	Joseph Smith

The Year was removed just to have less data in the output. Notice the main part of the XSLT code (highlighting the `xsl:sort` instruction:

```
<xsl:for-each select="books/book">
  <xsl:sort select="name" order="descending" data-type="text" />
  <tr><td><xsl:value-of select="name"/></td>
    <td><xsl:value-of select="author"/></td></tr>
</xsl:for-each>
```

Here are the parameters you see in the `xsl:sort`: **select** – as always to select the element/node that the sort will be applied to (in the case here, the sort will be applied to the **name** node); **order** – the default is the ascending order and in the case shown the content of the **name** node will be sorted in a descending order; **data-type** – the default is text (there would be no need to type this one in our example, but just to show to you that this is another parameter you can use and if you were dealing with numbers you would put **data-type="number"**).

That is why when **books2.xml**, data is displayed in the browser, using **bookloopsort.xsl**, you will see the name of the book in descending order of the text in the **name** node!

NOTE:

Make sure you use the correct **data-type** when sorting because if you sort numbers with **data-type="text"** (default), you might get some error in the order the numbers will be presented such as the numbers 97, 7, 100 if considered "text", would be presented as 100, 7, 97 when in ascending order. Another important point is that you can nest `xsl:sort` within other `xsl:sort` as this will allow you to sort on multiple nodes.

Creating Output Attributes

Depending on the HTML you will be generating (or even if you will be generating another XML document), you might need to create attributes in the output – for example, if you are generating a `` tag, or an `<a>` tag, you would need to create the ***src*** and ***alt*** attributes for the `` tag and the ***href*** attribute for the `<a>` tag.

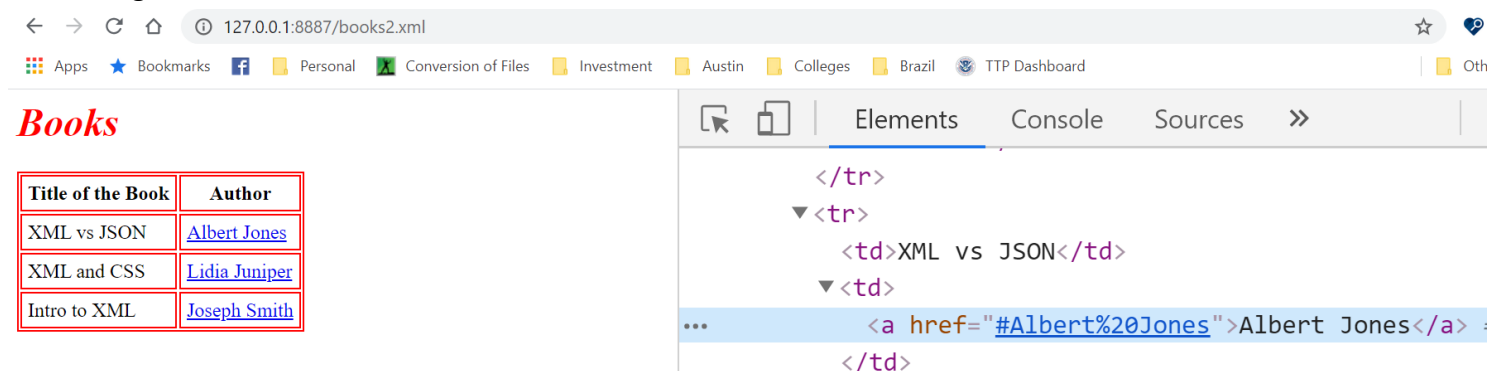
Look at the code in the file **bookauthorlink.xsl**. Here is the part of the code that contains the `xsl:attribute` instruction:

```
<xsl:for-each select="books/book">
  <xsl:sort select="name" order="descending" data-type="text" />
  <tr><td><xsl:value-of select="name"/></td>
    <td>
      <a>
        <xsl:attribute name="href">
          #<xsl:value-of select="author" />
        </xsl:attribute>
        <xsl:value-of select="author" />
      </a>
    </td></tr>
</xsl:for-each>
```

The opening `<a>` and closing `` were included around the content of the `author` node and the `xsl:attribute` instruction was used to define that the `href` attribute of the `<a>` would be formed by the text contained in the ***author*** node (the author's name). This means that the HTML of the hyperlink for the first book node will be:

```
<a href="#Albert%20Jones">Albert Jones</a>
```

If you modify **books2.xml** to use the **bookauthorlink.xsl** and then open **book2.xml** in the browser, you will see something as:



Title of the Book	Author
XML vs JSON	Albert Jones
XML and CSS	Lidia Juniper
Intro to XML	Joseph Smith

This image is showing the XML file displayed in the Chrome browser and on the right side shows the Chrome Developer's Tool (CDT) opened (to open the CDT you click on the top right icon on the Chrome browser, then select **More Tools**, and then select **Developer Tools**) and using the **Elements** tab to see each HTML element that was generated by the transformation imposed by **bookauthorlink.xsl** on to **books2.xml** file.

Notice the code of the hyperlink on the CDT: the **%20** is to substitute the blank space between the text **Albert** and **Jones** and the pound sign (**#**) was added from the XSLT code. The **author** content is displayed because of the second `xsl:value-of` while the first `xsl:value-of` instruction was used to place the **author** content beside the pound sign (**#**). This means that you are using the `xsl:value-of`

instruction to generate the value of the attribute you are outputting.

Using Templates

We know that the root template – `xsl:template` – is the first thing processed in any XSLT style sheet document. The template is the set of rules that will be applied to the root node of the XML document.

What you will learn now is that XSLT allows you to create more templates – you do not need to have only the root template. This feature allows you to create different sets of processing rules that can be applied to different parts of the XML document.

One of the main benefits of using templates is that you can reuse a template for other nodes meaning that you can create a template and simply apply it whenever necessary.

Open in your web editor the files **students.xml** and **studentstemplates.xsl**. Also, using a secure server, open **students.xml** in the browser. Your browser should show something as:



Students

W0023139

First Name:Deborah

Last Name:Kad

Nick Name:Deb

W0134265

First Name:Adash

Last Name:Gupta

Nick Name:Adash

W1923411

First Name:Stephanie

Last Name:Singh

Nick Name:Steph

Let us analyze what we have in the **studentstemplates.xsl**:

- From line 4 until line 12 you have the `xsl:template` that we have been using so far (defining the start of an HTML document, etc.) but, when you look at line 9, you see `<xsl:apply-templates select = "class/student" />` and that's a difference from what we have been using – instead of using `<xsl:value-of select =.../>` to get the value of a certain node, we are using **apply-templates**.
- The `<xsl:apply-templates select = "class/student" />` instruction is selecting the **student** node inside the **class** root node and if you pay attention, you will notice that from line 14 to line 21 of the code you have a new `xsl:template` created with the `select = "class/student"`. So, when the processor hits line 9 - `<xsl:apply-templates select = "class/student" />` - it will then use the template defined by the `<xsl:template match = "class/student">` that starts on line 14 and ends on line 21

- Notice that the `<xsl:template match = "class/student">` is formed by other templates – it has 4 `<xsl:apply-templates>` instructions. The first one with `select = "@nbr"` (**interesting to notice that you can select an attribute – just need to use the @ symbol in front of the name of the attribute!!!**); the second one will be `select = "firstname"`, then `select = "lastname"`, and finally `select = "nickname"`. Each one of those refer to their respective templates that you see below in the code: `<xsl:template match = "@nbr">` (line 23); the `<xsl:template match = "firstname">` (line 29); the `<xsl:template match = "lastname">` (line 35); and `<xsl:template match = "nickname">` (line 41).
- Each template is defining how the content of the specific node will be displayed and whenever necessary, within that code, you can reuse the template(s) created.

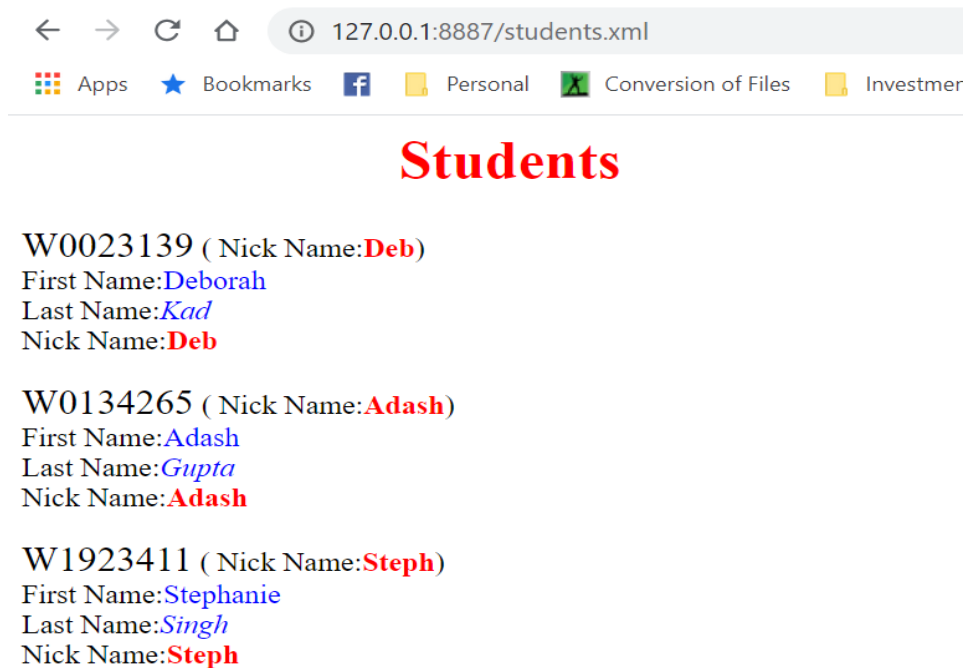
To see how those templates can be reused, we will modify a little bit the **studentstemplates.xml** file. You need to open this file in the editor and then you will substitute line 16 that currently has:

```
<xsl:apply-templates select = "@nbr" /> <br />
```

By the following code:

```
<xsl:apply-templates select = "@nbr" /> (<xsl:apply-templates select = "nickname" />) <br />
```

You are applying the **nickname** template again to the first line of the display, right after the **student nbr** and between parentheses. If you save the file and then reopen **students.xml** in the browser (or refresh it), you should see something as:



← → ↻ 🏠 ⓘ 127.0.0.1:8887/students.xml

📱 Apps ★ Bookmarks 📘 Personal 📄 Conversion of Files 📁 Investor

Students

W0023139 (Nick Name:**Deb**)
 First Name:**Deborah**
 Last Name:**Kad**
 Nick Name:**Deb**

W0134265 (Nick Name:**Adash**)
 First Name:**Adash**
 Last Name:**Gupta**
 Nick Name:**Adash**

W1923411 (Nick Name:**Steph**)
 First Name:**Stephanie**
 Last Name:**Singh**
 Nick Name:**Steph**

This simple example shows that if you define a template, by reusing it, you are avoiding retyping on different places, the same code all over again!

Notes:

- The root template is simply a template with a pattern that matches the root node and only the root template is called automatically while the other templates need to be manually applied or they will be ignored.
- When you have multiple templates in your XSL file (like in the **studentstemplates.xsl** file), the order of the `xsl:apply-templates` will determine the order the templates will be processed.
- If you do not specify the `select` attribute in the `xsl:apply-templates` instruction, the processor will look for and apply a template to each of the current node's children
- You can include an `xsl:sort` element in the `xsl:apply-templates`

We can apply a sort to the `xsl:apply-templates select = "@nbr"` that is found in line 9. Just modify line 9 from:

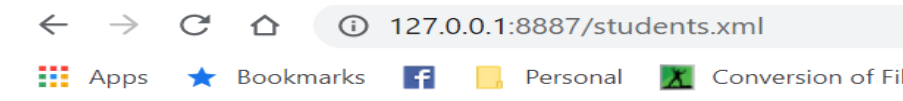
```
<xsl:apply-templates select = "class/student" />
```

To be:

```
<xsl:apply-templates select = "class/student">
  <xsl:sort select="lastname" />
</xsl:apply-templates>
```

The modification was to sort the data by the content of **lastname** node and, as we are including the `xsl:sort` instruction, the `xsl:apply-templates` will not be a self-closed tag anymore and that's the reason we removed the forward slash (/) from the initial self-closed tag, and added the closing `</xsl:apply-templates>` tag and the `xsl:sort` between.

After saving this modification, you can reopen **students.xml** in the browser – REMEMBER TO USE A SECURE SERVER – and you should see the same list as the previous image, but the students sorted by their last name:



Students

W0134265 (Nick Name:**Adash**)

First Name:**Adash**

Last Name:**Gupta**

Nick Name:**Adash**

W0023139 (Nick Name:**Deb**)

First Name:**Deborah**

Last Name:**Kad**

Nick Name:**Deb**

W1923411 (Nick Name:**Steph**)

First Name:**Stephanie**

Last Name:**Singh**

Nick Name:**Steph**

XPath

When you learned about XSLT, you created and applied templates to transform the XML document. When you created the template, you specified a pattern, using the match attribute, to specify the node(s) that the template would be applied to and when writing this pattern/expression, you are in reality using the syntax of XPath (XML Path Language).

XPath has built-in functions to do math, process strings, and test conditions in any XML document and you will see these functions in more details later in the course.

The foundation of XPath is the ability to use location paths to reach a node or node set. The location path uses relationships that describe the location of a node or set of nodes relatively to a given node. For that purpose, it is necessary to remember that any XML document can be represented in a tree structure and in an XML node tree, everything is a node and every node is in some way related to another.

The root node (document node) can have any number of child nodes and, of course, from the hierarchical perspective, for these child nodes, the root node is a parent node. Again, going down the tree structure, the child node can also have any number of child nodes too, etc.

When two child nodes have the same parent, they are called sibling nodes. The descendant nodes are a node's child nodes, also the child nodes' children, etc. and the opposite would be the ancestor nodes.

Once you know the relationship of the nodes, you can use XPath to access any node from any other node.

IMPORTANT NOTE:

New versions of XSLT and differences

Version 1.0 was released in 1999 and version 2.0 in 2007. Here are [the differences between those two versions](#).

At the time the version 2.0 was released there were some developers that wanted to convert to the newer version right away mainly for the following features that came with version 2.0:

- Ability to define your own (schema) types
- The XPath 2.0 sequence type that did not exist in version 1.0
- The xsl:function instruction that brought the ability to define and write functions in pure XSLT
- The for clause in XPath 2.0
- Better and powerful string processing with XPath 2.0 that supports regular expressions in tokenize(), matches(), and replace() functions.
- Move convenient, powerful and expressive grouping: the xsl:for-each-group instruction
- More powerful XPath 2.0 functions for date, time, duration, etc.
- New value comparison operators such as lt, le, eq, gt, ge, ne

These are some of the improvements made from version 1.0 to version 2.0. Some steps were defined in case developers wanted to convert from XSLT 1.0 to 2.0:

- Change the version attribute of xsl:stylesheet or xsl:transform element from 1.0 to 2.0

- Remove any xxx:node-set() functions
- Be ready to test the surprises you may get from xsl:value-of that for version 2.0 outputs not just the first, but all items of a sequence.
- Try to use the new xsl:sequence instruction as much as possible to replace any xsl:copy-of instructions or use it instead of xs:value-of any time when the type of the output is not a string or text node.
- Test after the conversion
- You can also decide which named templates you could rewrite as xsl:function

XSLT version 3.0 was released in 2017.

All these versions are backwards compatible, and this means that XSLT 1.0 stylesheets should still work in more modern XSLT engines with little or no modification. When version 3.0 came up some open source and commercial versions of conversion came up making the upgrade easy.

One of the main implementations in that version is that in XSLT 3.0, an inbound document can be in JSON and not only in XML. The processor can then take that file, use the json-to-xml() function to convert it into a known XML format, process the data through the templates, then convert the output back into JSON (or even into HTML5 among other formats).

One difference is that any time an expression can be evaluated as a string or similar atomic value, the <xsl:value-of> statement can be replaced with {} (braces). These expressions are called text value templates or TVTs.

Imagine this expression below used to generate the full name of an employee using the first and last name and done in XSLT 1.0:

```
<j:string key="fullname"><xsl:value-of
  select="fn:concat(j:string[@key='firstname'], ' ',
j:string[@key='lastname'])"/></j:string>
```

This could be rewritten in XSLT 3.0 as:

```
<j:string key="fullname">{
  j:string[@key='firstname'] || ' ' || j:string[@key='lastname']
}</j:string>
```

It requires fewer keystrokes (reduces the verbosity of the language) but it is also easier to follow especially due to the use of the concatenation operator || replacing the concat() function.

There were also other modifications/enhancements in XSLT 3.0 that you can find in the [article from Kurt Cagle with nice examples](#)

The main idea is to remember that the processors are backward compatible and there might be reasons that companies might decide not to upgrade: cost, lack of staff, time to dedicate for testing/debugging, etc.

Relative vs Absolute Location Paths

Those are the two kinds of location paths that you can use.

The **relative location path** is made of a sequence of location steps that are separated by a forward slash (/) and once selected each node in that set is used as the current node for the following step and so on.

The **absolute location path** is made of a forward slash (/) and optionally followed by a relative location path. The forward slash (/) by itself will select the root node. And if there is a relative location path following that initial forward slash (/) that means that the location path is a relative location path that starts in the root node.

Node Types

In XPath there are 7 different node types:

- Root node – there is always only one root node
- Element node
- Text node
- Attribute node
- Comment node
- Processing instruction node
- Namespace node

For each of those node types there is a way to get its value and for some of those node types the value is the node itself and there are cases that for certain node types the value is based on the value of the descendant nodes.

Current Node

The XSLT processor goes through your XSL file and it will work on one node at a time and by using the `xsl:template`, `xsl:apply-templates`, `xsl:for-each` elements the processor will know which parts of the XML document to process and when.

The node that is currently being processed is called the **current node**.

By default, the current node is the one specified by the current template (the value you see specified for the **match** attribute). If you see an `xsl:apply-templates` the current node will be the one in the **match** attribute of the template being specified and then, when the XSLT processor “returns” from that template, the current node will be again to be the one from the original **match** attribute – for example, in the case of the **studenttemplates.xsl** file – when the XSLT found the first `xsl:template` (line 4), the **match** attribute value was `“/”` which means that at that moment, the current node was the root node **class**. When it found the `<xsl:apply-templates select = “class/student” />` instruction, and it had to refer to the `<xsl:template match = “class/student”>`, the current node became the **student** node.

When there is an `<xsl:for-each>` instruction, the current node will change to the one specified by the ***select*** attribute and when that instruction finishes, the current node reverts back to whatever it was before that instruction was processed.

When you are processing the node and you want to use it in a select attribute, you can refer to it by using a single period (.) as `select = "."`.

Sometimes, you might not want to use the entire node set and in those cases you can add a test (predicate) which we will cover in later in this chapter.

Selecting the child of a node

Suppose you are processing a current node that has some elements that you want to use. Instead of writing the location path starting from the root node, you can refer to the child node(s) by simply using their name. Look at files **monument.xml** and **monumenthistory.xsl**. Here is just a part of the monument.xml being shown:

```
...
<monument>
  <name language="English">A Man</name>
  <name language="Spanish">Un Hombre</name>
  ...
  <history>
    <yearbuilt era="BC">300</yearbuilt>
    <yeardestroy era="BC">240</yeardestroy>
    <destroyedby>tornado</destroyedby>
  </history>
  ...
</monument>
```

The ... represent there is data before and data after and here the XML document is not being presented entirely.

You can then refer to the nodes in your XSL file such as (below you see a piece of the XSL file):

```
...
<xsl:template match="history">
  The <xsl:value-of select=".. /name[@language='English']" />
  <xsl:apply-templates select=".. /name[@language!='English']" />
  was built in <xsl:value-of select="yearbuilt" /><xsl:value-of select="year_built/@era" /><br
/>
  <xsl:choose>
    <xsl:when test="destroyedby != 0">
      It was destroyed by in <xsl:value-of select="yeardestroy" /><xsl:value-of
select="yeardestroy/@era" /> by <xsl:value-of select="destroyedby" /> .
    </xsl:when>
    <xsl:otherwise>
      You can still visit it today.
    </xsl:otherwise>
  </xsl:choose>
<br /><br />
</xsl:template>
...
```

Notice where the template starts – at the ***history*** node – meaning that when this template is processed, ***history*** is the current node. Then, as other elements exist for that current node, you can access the data of those elements exactly as showing in the ***select*** attribute (also in the ***test*** attribute) you see in the instructions above – the

yearbuilt can be accessed directly as a child node of the **history** node but then, you need the forward slash (/) – using relative location – for the attribute **era** of the **yearbuilt** and **yeardestroy** nodes as, in the tree structure, this attribute belongs (is “below” in the hierarchy) to those nodes respectively.

NOTE:

Between the `<xsl:value-of select="yearbuilt" />` and `<xsl:value-of select="yearbuilt/@era" />` you could see the element `<xsl:text> </xsl:text>`. This element is used to add literal text to the output and it cannot contain any other element – it can handle special characters (for example: &, >) or it can control white space – the way the `xsl:text` is written here, it will be adding a white space between the content of **yearbuilt** and **era**.

Selecting a Sibling or Parent Node

When you have a clear relationship between the current node and the one that you desire, it’s much easier to write the shortcut format than to write the absolute relationship path from the current node to the one you desire.

If you refer again to **monumenthistory.xml** file and the **monument.xml** file (so you can have a reference to the tree structure of the nodes in that XML file). Then you can try to understand the code below (showing just a part again of the XSL file):

```
...
<xsl:template match="history">
  The <xsl:value-of select=" ../name[@language='English'] "/>
  <xsl:apply-templates select=" ../name[@language!='English'] "/>
...
```

The current node is **history** (that is the value in the **match** attribute). But I need to refer to the **name** node to write the value of that node in the sentence being created and the **name** node is a sibling of the **history** node. So, how then was the location path built? In the select attribute you start with two periods (..) and this means that you are going up one level from the current node (**history**), meaning you are going to the parent node of **history** which is the **monument** node and then you write the forward slash (/) and the name of the node you want to select (in our case we need to select the **name** node) and, in the case here, we are selecting the **name** node with the **language** attribute equal to **English**. When it gets to the `xsl:apply-templates` instruction, you are selecting the same node – **name** – using the same location path used for `xsl:value-of` instruction but now the attribute you are selecting is NOT equal (!=) English.

Another observation in the piece of code presented here is that you are conditionally selecting a node by using what is called **predicate** – what you see between [and] in `xsl:apply-templates` and `xsl:value-of` instructions. The **predicates** you see in the code is a test expression and in the case shown here, it’s testing the value of the **language** attribute. One detail to be observed is that **predicates** are not only comparisons! For example, if you would write `[@language]` you would be selecting all the current node’s elements that have an attribute language no matter the value of that attribute.

NOTE:

Predicates need to be coded between [and]!

You can use multiple predicates to narrow even more your search. For example, if you have:

```
name[@language='English'][position() = last()]
```

This would mean that you would be selecting the **name** elements that have a **language** attribute equal **English** and that are the last node in the whole set.

If you had something as:

```
[last()]/@*
```

You would be getting all the attributes of the last element of the current node set

This example shows then how you can select the parent node and then, from that node, you can get to the sibling node and if you would need to select the node that is a child of that sibling node, you would continue the path with another forward slash (/) to get to the desired node.

If you see something as `select = "../**"` it means that you are going up, from the current node to the parent node and then using the forward slash (/) to select a child node but, in this case, the asterisk (*) means that you are selecting all the child elements of that parent node.

In the XSL file you also have this part below:

```
was built in <xsl:value-of select="yearbuilt" /><xsl:value-of select="yearbuilt/@era" /><br />
```

This means that you are selecting the **yearbuilt** node but you are also getting the value of the attribute of that node that is called **era**. So, whenever you want to reach the attribute of a node, you simply use the forward slash (/) followed by the @ symbol with the name of the attribute. And if after the @ symbol you see an asterisk (*), it means you are selecting all the attributes of that node.

NOTE:

The @ is sometimes referred to as the **attribute axis** and in XPath the axis is a set of nodes relative to the current node. There are other axes in the XPath language but they are rarely used exactly because of the examples shown here on how we can reach the parent and sibling nodes from the current node using the relative location path – but you can check those other axes at the W3School website (https://www.w3schools.com/xml/xpath_axes.asp)

Up to this point we have created only relative location paths, but you can certainly create absolute location paths (that do not rely on the current node). For example, in the **monumenthistory.xml** file, instead of having the code as:

```
The <xsl:value-of select="../name[@language='English']"/>
```

that is using the relative location path, we could have:

```
The <xsl:value-of select="/monuments/monument/name[@language='English']"/>
```

that is using the absolute location path starting with the forward slash (/) – that indicates that you are starting from the root node – and then going down the tree structure of the **monument.xml** until reaching the name node.

NOTE:

At any part of the absolute location path, you can use the * (asterisk) to specify that you want all the elements at that specific level. And, of course, pay attention to what you are coding when using the absolute location path as you will be ignoring the current node and then you need to know pretty well the tree structure of the XML document.

What if you want to select all the descendants of a node? That is when the double forward slash (//) comes handy! Look at the code of **monumentimagefiles.xml** and you will see the following code in both the `xsl:apply-templates` and the `xsl:template` instructions:

```
<ul>
  <xsl:apply-templates select="//*/@file">
  </xsl:apply-templates>
</ul>

and

<xsl:template match="//*/@file">
  <li><xsl:value-of select="." /></li>
</xsl:template>
```

The `//*/@file` has the two forward slashes (//) and asterisk (*) is selecting all the descendants of the root node and then you have a forward slash (/) followed by the @ symbol and the name of an attribute. Which means that all nodes with the **file** attribute that are descendants of the current node, will be selected.

If you apply the **monumentimagefiles.xml** to the **monument.xml** document and open it in the browser (FROM A SECURE SERVER), you should see something as:



List of Image Files of Monuments

- man.jpg
- womanface.jpg
- moongod.jpg

NOTE:

If you wanted to select all the descendants of the current node, you would need the period (.) before the two forward slashes (//) – as `./`. If you wanted to select all the descendants of any node, you would start with the two forward slashes (//) and then you would build the absolute location path to get to the node whose descendants you would be targeting. To get to a node that you do not know where it is in the tree structure of the XML document, you can use `//xyz` where **xyz** is the name of the element you are targeting and this technique would output all the **xyz** elements no matter where they are within the tree structure.

XPath Axes – more details

We already know that axes are used to identify elements by their relationship (parent, sibling, child, etc.). Here is a list of various axes values that you can also use when trying to reach one node of your XML document:

- ancestor – represents the ancestors of the current node (including parents up to the root node)
- ancestor-or-self – represents the current node and its ancestors
- attribute – represents the attributes of the current node
- child – represents the children of the current node
- descendant – represents the descendants of the current node (include the children up to the leaf node = the node with no more children)
- descendant-or-self – represents the current node and its descendants
- following – represents all nodes that come after the current node
- following-sibling – represents the following siblings of the context node (siblings are at the same level as the current node and share the parents of the current node)
- namespace – represents the namespace of the current node
- parent – represents the parent of the current node
- preceding – represents all nodes that come before the current node (before its opening tag)
- self – represents the current node

Let us see some example: open in the web editor **studentsxpath.xml** and **studentsxpath.xsl** – you will notice these two lines in the XSL code:

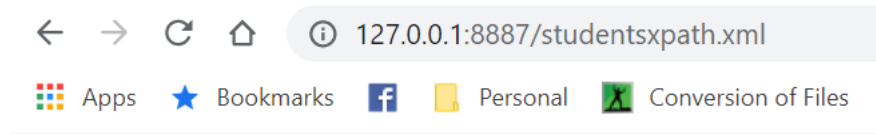
```
<body>
  <xsl:value-of select = "/students/student/preceding-sibling::comment()" />
  <br/><br />
  <xsl:text>First Student: </xsl:text>
  <xsl:value-of select = "/students/student/child::firstname" />
</body>
```

Notice the use of the axes shown in the list above written in red bold color such as: ***preceding-sibling::comment()*** and ***child::firstname*** – the first is setting ***student*** as the current node and, from there selecting the ***preceding-sibling*** which is the ***comment()*** (comment in XML code format, as there is no ***comment*** tag – that is why you need the parentheses); and the ***child::firstname*** that is selecting the child of the ***student*** node that is called ***firstname***.

Of course, this code could be simply:

```
<body>
  <xsl:value-of select = "/students/comment()" />
  <br/><br />
  <xsl:text>First Student: </xsl:text>
  <xsl:value-of select = "/students/student/firstname" />
</body>
```

And it would end up with the same result in the browser as shown in the image below (**remember that both XML and XSL files need to be in a secure server**):



Comment: This is the list of our students

First Student: Lucas

XPath Functions

The XPath functions are used to apply additional logic to the node sets selected to return only the data that you need.

Remember that when you use the `<xsl:value-of` instruction, the output will be the string value of the first node in a node set and when you use XPath functions you will be able to perform one or more operations on that string to modify the final result that will be outputted.

Comparison Test

One of the most common tests performed is the comparison of two values. Look at **newemployees.xsl** file that you can use as the stylesheet for **employees.xml**. You can see that the comparison is being applied at the `<xsl:apply-templates` instruction which means that the **employee** template (`<xsl:template match="employee">`) will be applied to the **employee** node, to bring data from that node (**firstname** and **hired**) to be part of the output but only if the value in the **hired** node is **>** 2006 (the **>** is a special character that represents greater than sign >). You can display **employees.xml** in the browser, using **newemployees.xsl** as the stylesheet and you will notice that only the node for Laura will be presented in the output.

NOTE:

The comparison can be done using: `=` (equal to), or `!=` (not equal to), or `<` (less than), or `<=` (less than or equal to), or `>` (greater than – as used in the example presented), or `>=` (greater than or equal to). Remember that if you do not start the special character with **&** (ampersand) and you do not finish with **;** (semicolon), you will get an error!

You can apply the comparison test in `xsl:template` instruction and also as condition test when using `xsl:if` and `xsl:when` instructions.

If the comparison is to a string, you should type the string enclosed in single quotes.

Position Test

You can select a node by its position (first, second, third, ... last). To test a node's position you will use the **position() = x** where **x** is the number that identifies the position of the node within the current node set. You can find the last node in a node set by using **last()** as the value for **position()** and, of course, the **position() = last() - 1** would translate to one node before the last node.

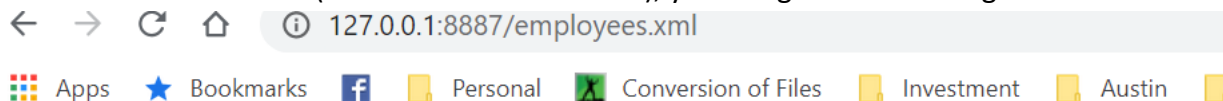
Notice how interesting the example in the **ouremployees.xsl** XSL file! You have the following code below (showing only the part that will be analyzed):

```
<xsl:for-each select="employees/employee">
  <xsl:value-of select="firstname" /><xsl:text> </xsl:text><xsl:value-of
select="lastname" />
  <xsl:choose>
    <xsl:when test="position()=last()"></xsl:when>
    <xsl:when test="position()=last() - 1">, and </xsl:when>
    <xsl:otherwise>, </xsl:otherwise>
```

```
</xsl:choose>
</xsl:for-each>
```

The `xsl:for-each` instruction is being used to loop over each one of the existing **employee** nodes (that's what you see in the value of **select** which is **employees/employee** making **employee** the current node). From that current node the value found in **firstname** is being written, then a blank space (using `<xsl:text>`
`</xsl:text>` instruction) and then the value found in the **lastname** node. Then, after writing the **lastname** value, it will be necessary to test the position of the node that is currently being processed to decide if the **lastname** will be followed by a . (period) for being the **last()** node of the employee node set, or if it will be , **and** for being the second last node (using the test **position() = last() - 1**) and if it's any other node (not the last, neither the second last) the , (comma) will be written.

If you apply this stylesheet – **ouremployees.xsl** – to the **employees.xml** document, and open the **employees.xml** document in the browser (FROM A SECURE SERVER), you will get the following:



Current Employees

Our employees are: Douglas Stuart, Laura Fonseca, Wallace McLean, and Zumara Arniston.

Mathematical Operations

It is possible to include simple mathematical operations in your expressions. For example, open **employeesmonthsalary.xsl** in your web editor. Notice how the node **salary** is being selected and will output the monthly salary of the employee based on a simple mathematical operation of "**salary div 12**" as the **salary** in the **employees.xml** document is a year-based salary.

You might need to count nodes to know how many nodes in a set you have – see the **count()** function being used in the beginning of the paragraph in **employeesmonthsalary.xsl** as `<xsl:value-of select="count(employees/employee)" />`. It will count the number of **employee** nodes and will then return that total as output.

NOTES:

- Notice that **div** was used instead of forward slash (/) to represent division. The other math operations are: addition – represented by the plus sign (+), subtraction – represented by hyphen (-), multiplication – represented by asterisk (*). The **mod** operation is to obtain the remainder of a division – for example: **15 mod 2** is equal to 1 because 15 divided by 2 is 7 with a remainder of 1.
- Multiplication and division have priority over addition and subtraction and then will be resolved before but you can use parentheses to override this – for example: **4 + 2 * 5** is equal to 14; it's not equal to 30 but if written as **(4 + 2) * 5** it will then be equal to 30.

- If you apply the **employeesmonthsalary.xsl** as the stylesheet of **employees.xml** and present in the browser, you will notice that some numbers have a lot of decimals. You can control the output by using some number formatting functions that we will learn later.
- The location path can optionally contain predicates – for example, we could have **count(employees/employee/hired[. > 2006])** and, in the case of **employees.xml**, it would count only the **employee** nodes with **hired** value greater than 2006.

Formatting Numbers

There is a function called **format-number** that can be used to format the numeric output of an operation (the example with the **employees.xml** when you apply the **employeesmonthsalary.xsl** stylesheet). Here are the basic rules to use the format-number function:

1. Write **format-number**
2. Type the expression that will produce a number to be formatted between parentheses
3. Still inside the parentheses, right after the mathematical operation that will produce the number, you will type a comma (,) then you will type a single space, then you will type, between single quotes (‘ and ’) the way you want your number formatted – for each digit that should always appear, you will type a **0** (zero), for each digit that should only appear if it’s not equal to 0 (zero), you should type a pound sign (#). If you want, type a period (.) to set where the decimals will be and after that period (.) type **0** (zero) as many times as you want the number of decimals to be. The same way, you can use a comma (,) to separate the thousands of your number.

In our example, you will substitute the expression that is calculating the monthly salary of the employee in **employeesmonthsalary.xsl** and instead of having:

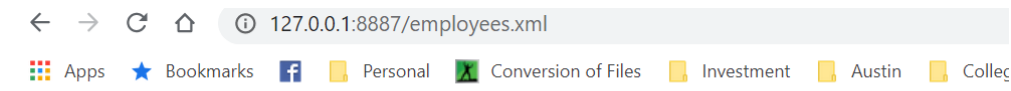
```
select="salary div 12"
```

you will have:

```
select="format-number(salary div 12, '##,##0.00')"
```

The expression **##,##0.00** means that we are expecting two digits that **ARE NOT 0** (zero) then the comma (,) to separate the thousand, then we are expecting 3 digits (two that **ARE NOT 0** (zero) and one that **can be or not 0** (zero)). Then, you have the period (.) where the decimal part will start and for that decimal part, we want to have only two digits that **can be or not 0** (zero).

If you apply the modified **employeesmonthsalary.xsl** to **employees.xml** and open the XML in the browser (FROM A SECURE SERVER), you should see something as:

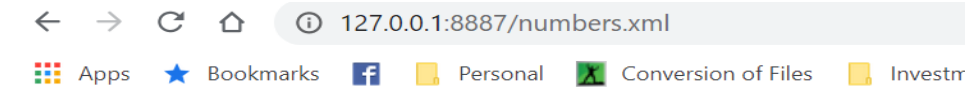


Monthly Salary of Employees

We currently have 4 employees in our start up and here are their names and current monthly salaries:

Douglas Stuart gets 10,000.00 per month.
 Laura Fonseca gets 4,583.33 per month.
 Wallace McLean gets 10,416.67 per month.
 Zumara Arniston gets 12,500.00 per month.

When using `format-number`, negative numbers will be shown with a minus (-) sign in front of it but, if you prefer that parentheses is shown, you should use the `;(xxx)` right after the first format expression still inside the single quotes. Look at some examples of `format-number` in the **numbers.xsl** file that is applied to **numbers.xml** (in reality, this XML file is just to show the results of the XSL file in the browser, it does not have any data). When opening **numbers.xml** (FROM A SECURE SERVER) in the browser, you should see:



Using the `format-number` function

500100
500100
500100.00
500100.0
500,100.00
23%
(2.34)

When you use the ***format-number*** function, if there are decimal places lost, the XSLT processor will round the resulting number. But if you want to round numbers during some math calculation, you can use:

- The ***round()*** – it will round to the nearest integer
- The ***ceiling()*** – it will always round up
- The ***floor()*** – it will always round down

Look at some examples using the ***pi*** number which has the value of **3.1415926535897932**:

`<xsl:value-of select="round(3.1415926535897932 * 10000) div 10000" />` - it will give 3.1416

`<xsl:value-of select="ceiling(3.1415926535897932 * 10000) div 10000" />` - it will give 3.1416

`<xsl:value-of select="floor(3.1415926535897932 * 10000) div 10000" />` - it will give 3.1415

`<xsl:value-of select="format-number(3.1415926535897932, '#.####') " />` - it will give 3.1416

`<xsl:value-of select="format-number(3.1415926535897932 * 100, '#.####') " />` - it will give 314.1593

Totaling Numbers

You can use the `sum()` function to add up all the values of the nodes. You could use, for example this function if you wanted to add up all the salaries of the `employees.xml` file – for example, you could add these lines of code in the `employeesmonthsalary.xsl` right after the closing `</p>` tag but still before the closing `</body>` tag:

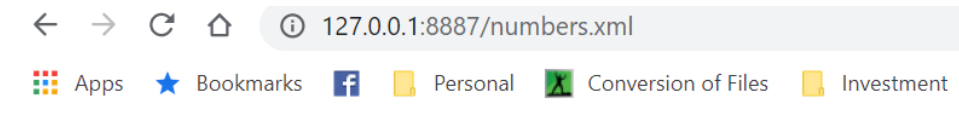
```
<p>The total of salaries of our company is <xsl:value-of select="format-  
number(sum(employees/employee/salary), '##,###.00') " /></p>
```

You would then get the total value of all the **salary** nodes from **employee.xml** being outputted in the format set in the instruction – in the case of the existing `employees.xml` data, you would see **450,000.00**

Substrings and Changing the Case

Sometimes you might need just a part of the string that would be shown in the output and that's when the **substring-after** and the **substring-before** coming handy.

Look at the **substring.xsl** file and especially the last `xsl:value-of` instruction where you see the **substring-after** being applied on top of the result from the **substring-before** of a telephone number to get ONLY the 3 digits of the area code of a telephone number. If you apply this XSL file to the `numbers.xml` (that has no data), you will see the following on the browser (FROM A SECURE SERVER):



Using the substring before and after

City: San Francisco
State: CA
Area Code: 415

Sometimes you might need to display text all in upper or lowercase. If that is what you need you can use the **translate()** function. If you open `employeemonthsalary.xsl` and modify the code from:

```
<xsl:value-of select="firstname" />
```

to:

```
<xsl:value-of select="translate(firstname, 'abcdefghijklmnopqrstuvwxy',  
'ABCDEFGHJKLMNOPQRSTUVWXYZ') " />
```

You will see that the value in the **firstname** node will come all in uppercase. If you wanted the opposite (from uppercase to all lowercase), you would switch the position of **'abcde....'** and **'ABCDE...'** in the code above.

In reality, you can use the **translate()** function to translate any character into any other character. For example, if you had (notice the ***** replacing the uppercase **O** in the second set of the translate function below):

```
translate(firstname, 'abcdefghijklmnopqrstuvwxy', 'ABCDEFGHJKLMN*PQRSTUVWXYZ')
```

You would see the **firstname** with value **Douglas** being shown as **D*UGLAS**.

NOTE – Differences between XPath versions:

XPath version 2.0 brings better and more straight-forward functions to convert to upper or lowercase such as *upper-case()* and *lower-case()*.

Version 2.0 was released in 2007. The version 2.0 supports all the primitive simple types built into XML schema as well as the types you can derive by restriction from those primitive simple types. There is also the sequence construction operator – for example, you can extract items from a sequence using the `[]` operator. For example:

```
(4, 7, 10)[2]
```

This expression will return 7. You can also have something as `(1 to 100)` to create a sequence but you cannot nest sequences.

But W3C wanted to make the transition from 1.0 to 2.0 easier so the way sequences were constructed were designed to be somewhat node-set friendly to be compatible with 1.0.

XPath 2.0 is centered around sequences and new expressions came up to work with them – for example, the `for` expression that you can use to loop or iterating over all items in a sequence. Suppose you want to find the average **height** among different monuments in your collection. The `for` expression would look something as:

```
avg(for $variable in /monuments/monument return $variable/height)
```

The `for` expression is used to loop over all `<height>` values and this is being done with a variable. Variables in XPath 2.0 start with a `$` and then the variable name is based on an XML-legal name. The path expression `/monument/monument` to return a sequence holding all `<monument>` elements in the document. We used the `return` keyword because `$variable` holds a new `<monument>` element each time through the loop and, by doing that, we get a sequence of all `<height>` elements. Then, to get the average height of the monuments, we used the `avg` expression.

Another interesting difference is that in XPath 1.0 you could use the `|` operator to create the union (combination) of two sets. In XPath 2.0 you can create not only unions but also intersections that will contain all the items two sequences have in common and you can also use differences which contain all the items that two sequences have that are not in common.

So, the main differences between XPath 1.0 and 2.0 are in the data model and type system – the move to a model where everything is a sequence of items and the items can be either atomic values or nodes.

XPath 3.0 was released in 2014. And one year later came version 3.1 and the differences between 2.0 and 3.1 are modest and you can check the differences in the [Revision log from W3C](#)

The main idea in this class is to know what XPath can be used for which is basically concentrated in what came since version 1.0.

Other XPath Functions

There other XPath functions that can help you and here are some websites that provide to you more XPath functions and some examples:

From the [Mozilla-MDN website - Functions](#)

From the [Way-2 Tutorial – Xpath good examples](#)

XSL-FO

The XSL was originally the single specification for formatting XML but before finishing it, W3C divided XSL into: XSLT (for Transformations) and XSL-FO (for Formatting Objects).

XSL-FO enables you to specify page layouts, including setting the margins and line spacing. Enables you to create headers, footers, generate endnotes, footnotes, cover sheets, etc. It was designed to format XML data.

You have already studied how to format/style data using CSS and using XSLT. You saw that XSLT can format XML data by transforming it into HTML. XSL-FO was designed to format XML data for output to print. You will see the examples in this part of the course when we will be using XSL-FO to generate printable output in PDF format.

To see XSL-FO working and/or edit XSL-FO, it is better to have an XSL-FO processor – there are some editors out there that I listed on the last page of the Chapter 1 lecture.

XSL-FO 1.0 became official in 2001 and was updated in 2006 to version 1.1. It is considered feature complete by W3C and the last update for the Working Draft was in January 2012 but the working group closed at the end of 2013 mainly because since 2013 CSS3-paged is a W3C proposal to replace XSL-FO and that is the reason this chapter will be really short so you can get to know XSL-FO and recognize when you see XSL-FO being used.

Every XSL-FO document can be broken into two parts that are enclosed in a top-level element ***fo:root***. The first part describes the overall structure of how the final output will be by using XSL-FO elements and attributes to set margins, page width, page height, etc. It also identifies different page templates.

This overall structure contains an ***fo:layout-master-set*** element that can contain one or more ***fo:simple-page-master*** element (child element). This child elements will describe page templates that are broken into five region elements, for example the ***fo:region-body***.

The second part of an XSL-FO document contains and formats the page content for the final output. You will find in this part elements such as ***fo:page-sequence*** and each will correspond to an ***fo:simple-page-master*** element defined in the first part.

Each page sequence contains one ***fo:flow*** element for each of the regions defined within the ***fo:simple-page-master*** element and each flow element will have one or more ***fo:block*** elements which will contain the actual page content.

Working with XSL-FO

An XSL-FO document is written using XML syntax. The XSL-FO document is a text-only file and is saved with the ***.fo*** extension. The first line of an XSL-FO document is the standard XML declaration that we already know.

There is a simple example called **example1.fo** among the files you downloaded for this course but to process that file, you would need an XSL-FO processor installed in your machine – you can read more about processing XSL-FO in this [Introduction to XSL-FO](#) web page.

But notice that the first line of the code is the XML declaration. The second line is the root element (required by XML syntax) – the **fo:root** element where the **xmlns** attribute points to the XML namespace for XSL-FO. Then you can see the overall structure part with one **fo:layout-master-set** which contains the **fo:simple-page-master** element with the attribute **master-name="monuments"**. The value of this attribute will be referenced in the second part of the XSL-FO document. Still inside the overall structure, you see that this document has a single region element called **fo:region-body**.

The page content (second part) is where you find the element **fo:page-sequence** that has the value of the attribute **master-reference** exactly to refer to the **master-name** attribute of **fo:simple-page-master** element of the overall structure. Inside the **fo:page-sequence** you can see the **fo:flow** element with the attribute **flow-name="xsl-region-body"** and this element has only one **fo:block** child element that contains the actual content that will be outputted.

Styling Blocks

As you could see in the simple example presented, a content can be a paragraph, a headline, etc. and it should be contained in blocks (**fo:block** element). But those blocks can be styled.

To style the **fo:block**, you would insert, as an attribute of that element, the name of the style property you want to use and set a value to it. Look at **example2.fo** and you will see some style applied to the **fo:block** elements created in this example. In this example you see **fo:block** elements that are children of another **fo:block** element and this is possible as those inner child elements will inherit the **font-size="14px"** from the parent.

Interesting to notice that many styling properties of the **fo:block** are the same as CSS properties and this was one of the reasons to start substituting XSL-FO by CSS in the W3C standards.

Adding Images

An image would also be contained by the **fo:block** element but to add the image you would need to use **fo:external-graphic** element such as:

```
<fo:external-graphic src="url('image.jpg')" content-height="200px" />
```

Page Template

This is defined in the overall structure part and it is contained in a single **fo:layout-master-set** element. This element might have one or more **fo:simple-page-master** element and this is what defines the page templates. When coding the **fo:simple-page-master** element you might find attributes such as **page-width**, **page-height**, **margin** that will define exactly the width, height, and margin (top, right, bottom, and left) of the template.

You can also set up a page template header, although the **example1.fo** only showed the body region in a page template (**fo:region-body** element). There are four other page template regions: header, left and right sidebars, and footer. To create a header, you would use the element **fo:region-before** and inside that element you can use the attribute **extent** with a numeric value such as 1in, or 20px which will correspond to the height of the header

region. Of course, you can set styles for the header too. But when you set the ***fo:region-before***, you are just setting the structure of it, you are not setting the content – remember that the content will be set in the content part of the XSL-FO structure. After setting up in the template, you will need to find the ***fo:page-sequence*** element whose ***master-reference*** attribute is equal to the ***master-name*** attribute from where you set the ***fo:region-before***. Then, as a child of this ***fo:page-sequence***, you will create the ***fo:static-content*** element with the attribute of ***flow-name="xsl-region-before"*** and between the opening ***fo:static-content*** and closing ***/fo:static-content*** you will create the ***fo:block*** element that will contain the content of the header – see an example in **example3.fo** file.

NOTE:

Although the header, region, footer, left and right regions are defined separately, in reality they are all part of the body region and, because of that, the margin of the body region must be the same (or greater than) the extent of the header region. If it is not, the body will overlap the header.

Extra

There are other things you can do when using XSL-FO such as: insert page breaks, insert the number of each page in the footer or the header, output page content in columns, etc. but as the group for XSL-FO folded in W3C, I will not extend this topic but at least you have the basics of what XSL-FO was created to do with an XML document. But if you ever need to learn more details about XSL-FO, you can refer to [FileFormat Document website](#) or using the [XSL-FO Reference from W3Schools](#)

DTD

Schemas, although not required, are very important for keeping XML documents consistent. For example, a group of historian people could create a schema called MonML (Monuments Markup Language) as a system for cataloging data about the monuments and this MonML might have elements like name, pic, history, etc.

You can compare any XML document to its corresponding schema to validate whether the XML document conforms to the rules specified in the schema.

DTD (Document Type Definition) is one of the existing systems for writing schemas. It is an older but widely used system with a limited syntax. Later on, in this course, we will study another system called XML Schema and then you will be able to compare and choose which one to be used according to your situation.

Working with DTD

DTD is nothing else than a set of rules that defines a custom markup language in XML. It identifies elements and their attributes and if the XML document that is created does not adhere to the rules defined by the corresponding DTD, it is not considered valid.

We already know that the XML building blocks are elements, attributes, and values. Elements are the foundational units and they can contain values, have attributes, and even other elements. The DTD will define a list of elements and any child elements that each element can have. It will also define each attribute the element can have and even if the attribute is required or optional. As you can see, the DTD defines the legal structure of the custom markup language and any valid XML document that is part of this language.

The DTD is a text-only document and generally saved with **.dtd** extension. It is not an XML document and that is why you will not see it beginning with the XML declaration.

Definitions in a DTD

Element that contains text

Many elements in your XML document will contain only text and although an element such as course might have child elements as name, description, room, teacher these elements most likely will contain just text.

To define an element that only contains text you will use the `<!ELEMENT xxx` where **xxx** will be the name of the element you want to define. If you look at **monument.xml** file (found among the files you downloaded for this course), the DTD for the **name** element would be something as `<!ELEMENT name (#PCDATA)>`. The **#PCDATA** defines the element name as one that will only allow text content.

NOTES:

PCDATA means parsed character data and text (called string) can be any series of letters, numbers, and symbols, for example: "Hi," or "55 Jungle Rd." or "94112"

XML is case-sensitive and if you type `<!Element` it will not work. The exclamation mark is part of the syntax and should not be forgotten. The name of the element can be written in a mixed case format but then you should always use that same format whenever referring to the name of that element.

Empty element

Remember that an empty element in XML does not have content value of its own and uses its attributes to store data. An empty element would be defined in DTD as `<!ELEMENT xxx EMPTY>` where **xxx** is the name of the element.

Notice that you do not use parentheses around **EMPTY** as used for **#PCDATA** and the way to define the attribute in DTD will be discussed later

Element with a child

To define in DTD for an element that has a child, you would code like `<!ELEMENT xxx (child)>` where **xxx** is the name of the element and **child** is the name of the child element. In this case, the xxx element will have only

If an element is defined to contain one other element, it means that it will not contain anything except that element, not even text. Later we will see how to make a child element be optional or even appear multiple times. We will also see later how to control the order in which the elements must appear in an XML document.

Remember that once you defined in the DTD that an element will contain a child, every time you use that element in an XML document, it must have the child defined otherwise the XML document will not be considered valid.

What if the element has a sequence of child elements (more than one child element)? You can define that in DTD and once you set up the sequence of the child elements, that will define the order in which the child elements must appear. In case you need to define an element with a sequence of child elements, you would write `<!ELEMENT xxx (child1, child2, ...)>` where **xxx** is the name of the parent element and **child1, child2**, etc. are the names of the child elements that will show up in that exact sequence order.

You must remember to include the comma (,) between each of the child's names and you cannot use (**#PCDATA**) in any part of that sequence as that sequence can only contain elements' names.

But what if one of those child elements can appear more than once? There are special symbols you can use in DTD to define how many times a child element can appear within a parent element. If you have something as `<!ELEMENT monument (name*, pic, history)>` it means that the **monument** element has 3 child elements: **name**, **pic**, and **history**. But the asterisk (*) right after **name** would mean that this child element can appear as many times as necessary or not at all (zero or more times); if you see the plus sign (+), that would mean that the child element must appear at least once and as many times as desired (one or more times); if you see the interrogation mark (?), that would mean that the child element can appear one time or not appear at all (zero or one time). If you do not see any of those signs it simply means that the child element must appear exactly one time. We will see later on how to define a pre-defined number of occurrences for a sequence but there is no way to specify the exact number of times a child element should be shown unless you would code something as

`<!ELEMENT monument (name, name, pic, history)>` and here you are specifying that the child element **name** would show up exactly twice one after the other, then you would see **pic**, and then **history** all being child elements of **monument** element.

Choices

Sometimes you might need a certain XML element to contain one thing or another and to define choices for the content of an element, you would use something as: `<!ELEMENT xxx (child1 | child2 | child3)>` where **xxx** is the name of the parent element and then each of the possible child elements – what you see between parentheses would mean **child1 OR child2 OR child3**.

After the closing parentheses you can add those quantifiers we learned before (*****, **+**, **?**) but if an asterisk (*****) is applied to a list of choices, it means that the element can contain any number of the individual choices in any order. One of the child elements can be a simple **#PCDATA** meaning that you would have text only, not a child element per say.

Up to this point, let us look at one example of a DTD that was defined based on an XML document – look at **colleges.xml** and **colleges.dtd** files (both files can be found among the files/folders you downloaded for this course). Open both in a web editor and then modify the top of the code of the XML document to embed the DTD inside the XML like what you see below (showing here just the first line of the XML document and what was inserted, in red color, to embed the colleges.dtd code there):

```
<?xml version = "1.0"?>
<!DOCTYPE colleges [
<!ELEMENT colleges (college+)>
<!ELEMENT college (#PCDATA | name | location | city | country)*>
<!ELEMENT name (#PCDATA)>
<!ELEMENT location (#PCDATA)>
<!ELEMENT city (#PCDATA)>
<!ELEMENT country (#PCDATA)>
]>
```

Now, if you use the [XML Validator Online](#) you will be able to validate the XML document against the DTD that was created and it should be all valid.

Let us understand what was coded in **colleges.dtd**! The first line is declaring the elements **colleges** that will contain one or more **college** child elements (see the plus sign (**+**) following the name of the child element). Then, the second line is declaring the **college** element that will have child elements, not necessarily in the order presented and they might be presented or not (because of the pipe (**|**) you see between the list of names of the child elements) – you see **#PCDATA** and if you look at **colleges.xml** you will see that one of the **college** child element only has text content with no other element. Then the last 4 lines, of **colleges.dtd**, are defining the child elements for the **college** element (**name**, **location**, **city**, **country**) which are all text type and that is why you see only **#PCDATA** defined for each of those child elements.

Element that contains anything

To define an element that can contain anything (any combination of elements and text) you can use the following `<!ELEMENT xxx ANY>` where **xxx** is the name of the element. But you should be very careful in deciding to use **ANY** as the objective of a DTD is to set up rules for what an element can and cannot contain and if you allow every element to contain anything, then why would you create a DTD to start with? DTDs are not required but they help keep data consistent. Although you may define an element with **ANY**, if the element contains child elements, those child elements need to be defined in the DTD too as all elements that appear in a valid XML document must be defined in the DTD.

NOTE:

Every element must be defined only once in the DTD and NO MORE! Even if the element shows up in different places, it must be only defined once.

Defining Attributes

We know that attributes provide additional data about an element, but they are not part of the content, they just add some information about the content. Elements are generally set in the code when you want their content to be displayed while attributes are created to add some information about the content. The values of attributes are not easily validated by a DTD and they cannot contain multiple values whereas an element can. One usual example of attributes is the store of IDs. In general, the way you will structure the XML, should be based on its usage which means that if there is a piece of information that you will not be using too much neither do anything with it, the attribute would be a good choice to store that information.

You cannot code an attribute in your XML document unless it has been declared in the DTD. To define an attribute in the DTD you would use something as: `<!ATTLIST xxx atrname CDATA req>` where **xxx** is the name of the element that will have the attribute called ***atrname***; **CDATA** is exactly to indicate that the attribute type is text but it will not be parsed by the processor (differently than **#PCDATA**); the ***req*** will be **#IMPLIED** to indicate that the attribute is optional (may be omitted) OR **#REQUIRED** to indicate that the attribute must contain a value (is required).

All the parts of the declaration of an attribute are case sensitive meaning that whatever I showed here in uppercase it is because it needs to be that way. You can define more than one attribute per element, and you would do that by repeating the sequence of ***atrname* CDATA *req*** before closing `>` - it would be something as:

```
<!ATTLIST note
    to    CDATA #REQUIRED
    from  CDATA #REQUIRED>
```

Here you have the element called ***note*** with two attributes – ***to***, ***from*** – both being required!

Instead of **#IMPLIED** or **#REQUIRED**, you can set an attribute to have default values and to do this you would code `<!ATTLIST xxx atrname CDATA "def">` where **xxx** is the name of the element, ***atrname*** is the name of the attribute and ***def*** (between double quotes) is the default value you want to establish. For example:

```
<!ATTLIST weight units CDATA "pounds">
```

The **weight** element will have an attribute named **units** that the default value will be **pounds** if none is set in the XML document. If you type `#FIXED "def"`, then **def** will be the value set to the attribute if none is set in the XML document and if there is a value for that attribute it HAS TO BE def for the XML to be valid.

NOTE:

You cannot combine a default value with either **#REQUIRED** or **#IMPLIED** as neither option would make sense when you are setting a default value.

Attribute with choices

You can define attribute types that will allow more than just character data (**CDATA**). To define this type of attribute you would code `<!ATTLIST xxx attrname (choice1 | choice2) req>` where **xxx** is the name of the element; **attrname** is the name of the attribute; **choice1**, **choice2**, etc. represent possible values for the attribute and **req** represents any of the attribute statuses (**#IMPLIED**, **#REQUIRED**). For example: `<!ATTLIST weight units (lbs | kg) #REQUIRED>` means that the element **weight** has an attribute called **units** that can have the value of **lbs** or **kg** and is required.

Attribute with unique value

Some special attributes (for example, ID of a product, or a student ID) should be defined to have a unique value (cannot be repeated throughout the XML document). To define an attribute like this you would code `<!ATTLIST xxx attrname ID req>` where **xxx** is the name of the element, **attrname** is the name of the attribute; **ID** (typed like shown here) is defining that the value of that attribute should be unique and non-repeatable throughout the XML document; **req** represents the attribute status but when you define the attribute as ID, that **req** can only be **#REQUIRED** or **#IMPLIED** meaning you cannot define a default value for it. Here is an example of an **ID** attribute being declared:

```
<!ATTLIST student studentNbr ID #REQUIRED>
```

Here you have the **student** element that has an attribute called **studentNbr** that is being defined as an **ID** attribute (needs to be unique throughout the XML document) and is required.

NOTE:

The XML document will not be considered valid if two elements with **ID** attributes have the same value, no matter if the element or attribute names are the same or different, for example, in the same XML document you could not have `<person identifier="ID120"/>` and `<company identifier="ID120"/>`. The only exception to this rule is that you can have unlimited number of omitted **ID** attributes which will imply in a null value. The value of an ID attribute must follow the XML rules for names, and this means that the ID attribute cannot contain only numeric values unless you prefix them with a letter or underscore.

An attribute whose value is the same as any existing **ID** attribute is called an **IDREF** attribute. An attribute whose value is a list (list items separated by white space) of existing **ID** attribute values is called an **IDREFS** attribute. And to reference attributes with unique values, you should use the code as `<!ATTLIST xxx attrname IDREF req>` where **xxx** is the name of the element; **attrname** is the name of the attribute; **IDREF** (or **IDREFS**) to define an attribute that can contain a value matching an existing **ID** attribute's value (or an attribute that can contain several values, separated by white space, that match any existing **ID** attribute's value); and **req** to identify the attribute status (**#IMPLIED**, **#REQUIRED**).

NOTE:

There may be many **IDREF** attributes that refer to the same **ID** and that is fine as it is just the **ID** that needs to be unique to one element.

Here is a simple example of an XML document and the way the DTD had to be coded in order to not have conflict with the same value of two attributes you see in the XML document:

XML document:

```
<artist name="Elton John" location="United Kingdom" artistID="EJ"/>
```

```
<album name="Yellow Brick Road" albumID="EJ"/>
```

DTD:

```
<!ELEMENT artist EMPTY>
<!ATTLIST artist name CDATA #REQUIRED>
<!ATTLIST artist location CDATA #REQUIRED>
<!ATTLIST artist artistID ID #REQUIRED>
<!ELEMENT album EMPTY>
<!ATTLIST album name CDATA #REQUIRED>
<!ATTLIST album albumID IDREF #IMPLIED>
```

Notice the use of **IDREF** when declaring the attribute **albumID**, otherwise the XML document would be invalid as **albumID** has the same value of **artistID** which was an attribute declared as being an **ID** attribute (unique value throughout the XML document).

Attributes being restricted to valid XML names

There is one restriction you can apply to attributes. If you define the attribute as being **NMTOKEN**, it means that it must be a valid XML name (it begins with a letter or an underscore and contains only letters, numbers, underscores, hyphens, and periods). To declare an attribute as **NMTOKEN**, you would code `<!ATTLIST xxx attrname NMTOKEN req>` where **xxx** is the name of the element; **attrname** is the name of the attribute; **NMTOKEN** (all uppercase) meaning that the attribute value must be a valid XML name – cannot have white spaces in the value – (or **NMTOKENS** if you want the attribute value to be a list of valid XML names); **req** to identify the attribute status.

NOTE:

If you want the value of an attribute to be a valid XML name but also be unique throughout the XML document, you should use **ID** instead of **NMTOKEN**.

In the example we previously saw, if I had declared the **location** attribute as `<!ATTLIST artist location NMTOKEN #REQUIRED>`, the value of the attribute, as shown in the XML document would be invalid as it contains spaces.

Look at the code in **recipes.xml** and then **recipes.dtd** (you can find both files among the files/folders you downloaded for this course) and see if you can understand the declarations of elements and attributes in **recipes.dtd**.

Entities and Notations in DTDs

You have probably used special characters while coding HTML – for example: **<** to represent `<` or **>** to represent `>`, etc. Well, entities act like automatic text entries (shortcuts) – you define the name of the entity and the text it should expand into when it is referenced in your XML document; then, when you type the entity reference in an XML or DTD, it will be replaced with the text you defined.

There are two main types of entities:

- **General entities** – can only be expanded in XML documents; you can have internal or external, and parsed or unparsed types of general entities
- **Parameter entities** – can only be expanded in DTDs; these entities are always parsed but they can be internal or external

The simplest type of entity is defined in a DTD and officially these entities are called internal general entities. You use the following code to define this type of entity `<!ENTITY entname "content">` where **entname** will be the name of the entity and **content** (between double quotes) will be the content that will be shown when the processor parses the XML document that is using the **entname** you are defining.

NOTES:

The **entname** must follow the XML rules of names.

In XML there are five built-in general entities: **<**, **>**, **&**, **"**, **'**; – any other entity needs to be declared in the DTD before it is used.

MOST browsers do not accept external entities declarations – for some browsers, you can test the declaration of an entity by including it at the top of your XML code – take a look at **email.xml** and **email.dtd** (both files can be found among the files/folders you downloaded for this course) – you can see that the declaration of the entity **course** is being done in the XML document within the part that links the XML document to the DTD.

Using general entities

Once a general entity is defined in the DTD, you can use it in the XML document that references that DTD. To use the general entity in the XML document, you just need to type **&entname;** where you want to add the content related to the entity you are using.

If you use an entity that has not been defined in the DTD, the parser will return an error.

If you want to add special symbols (for example, the copyright symbol ©), you can use **Unicode characters** – they look very similar to entities but they are not entities and **do not need to be declared in the DTD**. They can be used as **&#n** where **n** is the decimal number that represents the character or, if you want to use the hexadecimal representation, you should use **&#xn** (notice the **x** before the number) – there are many tables of Unicode Characters you can find on the web and here is one from [Rapid Tables](#) for your reference.

You cannot use entities in XSLT documents.

There might be cases that you would need to define an entity that could be used in different DTDs and, in this case, it is more convenient to save it in a separate external document. That external document would be saved with the extension **.ent**. **But it is important to alert that there are some vulnerabilities when you declare your entities as external** – here is an article [Port Swigger – XML external entity \(XXE\) injection](#).

Entities for unparsed content

Entities that contain text are called **parsed entities** as the XML parser will look at them and analyze them when going through the XML document. **Unparsed entities** do not usually contain text (they can) but most importantly, they can be bypassed by the XML parser. This type of entities can be used to embed non-text or non-XML content into an XML document.

First you need to create the unparsed content that can be literally anything – plain text, video, PDF, image, etc. Then, you need to identify how to process the unparsed content using a **notation** which means that in the DTD, wherever you want to embed that content, you would type `<!NOTATION notname SYSTEM not.instrucion>` where **notname** is the name you are using to identify the unparsed content and **not.instruction** is the information that defines how to process that content. For example: `<!NOTATION jpg SYSTEM "image/jpeg">` declares a notation called **jpg** that will be used later when you create the entity for the unparsed content. Then, the entity for that unparsed content would be created as `<!ENTITY schoolpic SYSTEM "school.jpg" NDATA jpg>` - the entity name is **schoolpic** that refers to an external (**SYSTEM**) file called **school.jpg** and more information about that file can be obtained by looking at the notation identifier (**NDATA**) called **jpg**.

The **not.instruction** can be a MIME type, a URI to indicate a local or external application

Now that an entity has been defined for the unparsed content, you can embed it in your XML document, and they are referred through a special ENTITY attribute type. To declare the attribute that will contain the reference to the unparsed entity, you would code in the DTD the element that will contain the attribute referencing to the unparsed entity and then you would use `<!ATTLIST xxx attrname ENTITY status>` where **xxx** is the name of the element that will have the attribute called **attrname**; the word **ENTITY** indicates that the attribute

can contain references to an unparsed entity (could be **ENTITIES** if you would like the attribute to be able to contain multiple references to unparsed entities separated by white space); and **status** (for example, **#REQUIRED**) or default value for the attribute.

For example:

This would be in the DTD:

```
<!ELEMENT name (#PCDATA)>
<!ATTLIST name language CDATA #REQUIRED>
<!NOTATION jpg SYSTEM "image/jpeg">
<!ENTITY monument_image SYSTEM "theman.jpg" NDATA jpg>
<!ELEMENT photo EMPTY>
<!ATTLIST photo source ENTITY #REQUIRED>
```

This would be in the XML:

```
<?xml version="1.0" standalone="no"?>
...
<monuments>
  <monument>
    <name language="English">The Man</name>
    <name language="Spanish">Le Hombre</name>
    <photo source="monument_image" />
  </monument>
</monuments>
```

Notice that the value of the **source** attribute, in the **photo** element is a reference to the entity declared in the DTD and that entity refers to the **jpg** notation!

Note that although XML parsers are supposed to be able to use the notation information to help them to view/display the unparsed entity, most browsers cannot do it and will not show the embedded data. Among developers, there is a certain consensus that using **unparsed entities** is complicated and confusing and that instead of using it, you should set an element's value to a URL that would point to any file of your choice and use attributes to clarify additional information if you so desire.

There is another example, among the files/folders you downloaded for this course, that refers to a declared entity for the value of an attribute – in reality, this examples refers to two different entities for the value of the attribute – look at **stars.xml**

Parameter Entities

Up to this point, we have seen entities that reference text or file that can be used in an XML document. But, in a DTD, you can also create entities for the DTD itself which are known as **parameter entities**. You would know you have a parameter entity if you would find a code like this `<!ENTITY % entname "content">` where **%** show that this is a parameter entity; the **entname** is the name of the entity which you will refer to when using the entity in the code of the DTD; the **"content"** is surrounded by double quotes and it represents the text that will be shown when the entity is used in the DTD.

Once the parameter entity is defined, then you can use it in the DTD by just typing **%entname**; so, for example, instead of typing (**#PCDATA**) for every element that you define in the DTD, you could first define a parameter

entity as `<!ENTITY % pc "(#PCDATA)">` and then, when defining an element in your DTD you could simply code `<!ELEMENT firstname %pc;>` and this would be translated to `<!ELEMENT firstname (#PCDATA)>`

You need to create the parameter entity first before you can use it and parameter entities can only be used within the DTD. There are ways to create an external parameter entity, but we will not go over it for the same reason that we will not be using external entities.

Using a DTD

Up to this point you learned the syntax rules on how to create a DTD. We know that the DTD defines rules for every element and attribute of an XML document. In order to validate the XML against the DTD, you need to declare the DTD in the XML document.

We have seen some examples of DTDs that were written and saved in a **.dtd** type of file but the DTD can also be written within the XML document. The main benefit of writing the DTD in the XML is that you will have only one file to manage and a lot of the free online validators out there will only validate your XML against the DTD if the DTD is written in the XML. On the other hand, having an external DTD is good in case you need this same DTD to validate a different XML document.

Declaring external DTD in the XML document

Once you create an external DTD (file with the **.dtd** extension), you need to refer to it from your XML document and this is done with the following code `<!DOCTYPE root SYSTEM "xxx.dtd">` where **root** is the name of the root element of your XML; the expression **SYSTEM** indicates that the DTD is defined in an external document; **xxx.dtd** is the name/path of the file with the DTD content.

NOTE:

You need to add in the XML declaration `<?xml version="1.0"?>` the expression `standalone="no"` so it will be `<?xml version="1.0" standalone="no"?>` and this expression tells the XML parser that the document will rely on an external file and, in our case, the one that contains the DTD.

Declaring an internal DTD



The syntax rules to create an internal DTD are the same as we studied for the external DTD the difference between the external to internal DTD is the way you will code the document type declaration (to declare the DTD in the XML document). To declare an internal DTD you will type `<!DOCTYPE root [dtd]>` where **root** is the root of your XML document; **dtd** is the whole code for the DTD that you would code in an external file – you just need to remember to enclose the DTD within **[** and **]**.

By the way, you can use both internal and external DTDs to validate one single XML document just remember that the `<!DOCTYPE` for the external would come first and the internal DTD would come after that and if there is

any conflicting rule, the internal DTD will win and, just to let you know, it is not a good practice to do that as most browser treat conflicting DTDs as errors.

There are some editors that validate the XML against the DTD for you and there are some online validators as well such as [XML Validator Online \(Studio\)](#) or [XML Validation](#). Open the latest one upload the file **emailvalidation.xml** that you will find among the files/folders you downloaded for this course. Notice that this file is using internal DTD declaration. After you select the file to be uploaded, click on the **Validate** button to validate the code. You will get an error that looks like the image below (showing only the error message):

An error has been found!

Click on  to jump to the error. In the document, you can point at  with your mouse to see the error message.

Errors in the XML document:

 24: 9 The content of element type "email" must match "(note)".

At first, reading the message, might be confusing because it points to line 24 of the XML code you pasted in the box and if you check the code, line 24 has the `</note>` tag which is the closing of the `<note>` tag. It is an XML where the root is **email** and inside that email you have 2 child elements called **note** and each of those elements is following the rule from the DTD where it says `<!ELEMENT note (to, from, subject, msg)>`. So, what is the problem? Well, take a look at the declaration of the element **email** (the root element that, of course, also needs to be declared) – it has: `<!ELEMENT email (note)>` - and again, you might say: “Yes, I understand, the element **email** should have a child element called **note**!” and I would say: “Right!” but I would also ask you: “how many child **note** elements are you allowed to have according to that rule in the DTD? And if you answered 1, then, that would be the problem with **emailvalidation.xml** file as this file brings 2 children to the **email** root element.

What should be done in this case? Well, it depends on the application you would be working on. For sure, if the application would accept more than one **note** as child element of email, then you need to do a simple change in the DTD part! Add the sign, right after (note) that would show that the email element can have 1 or more child note elements. Do you remember those signs? Here are those signs again:

- **No sign** → it means that the child element can show up only ONCE
- The ***** → it means that the child element can show up 0 (zero) or n times
- The **+** → it means that the child element needs to show up at least once but can also show up n times
- The **?** → it means that the child element can be shown 0 (zero) or 1 time only

So, if the application requires the **note** element to be shown and it can appear more than 1 time, then the right way to write the rule for the **email** element would be `<!ELEMENT email (note)+>`. Notice the **+** sign added! If you add that plus sign (**+**) like it is being shown here and save **emailvalidation.xml**, you can certainly do the validation again (uploading the file again) and you will get no errors!

You can try other things in the **emailvalidation.xml** file too such as adding a **note** element not having one of the child elements specified in `<!ELEMENT note (to, from, subject, msg)>` and see the error message. Or add a **note** that does not have a **subject** child element.

NOTE:

If you find a validator online that allows you to use external DTD, you need to make sure that the DTD file will also be found online as generally those online validators will not look somewhere else for that file. And, of course, in this case, you need to include the complete path to the DTD when you declare it in your XML document.

Pros and Cons of DTDs

DTDs are schemas. DTDs specify the elements, attributes, and relationships that a valid XML document should contain. Although DTD is very powerful, there are other schema languages for XML and the XML Schema is one of the most recognized ones and we will learn XML Schema in the next part of our course.

As any other technology, DTD has its pros and cons.

Pros:

- DTDs are very compact and not so difficult to understand with a little direction.
- They can be defined inline (internal DTDs) and you can define entities
- They are supported by most XML parsers

Cons:

- DTDs are not written following XML syntax
- DTDs do not support namespaces (we will learn about this topic later)
- DTDs do not have data typing, meaning they do not require data to be an integer, or a date, or a string, etc. which decreases the strength of the validation
- DTDs have limited capacity to define how many child elements can be nested within a given parent element

Extra Notes

There is an interesting [DTD generator](#) that can generate a DTD for you based on your XML document but I would recommend only using this tool if you are sure that your XML is valid, although you could certainly adjust your DTD if there is something missing in the DTD generated.

Extra Materials

[DTD Tutorial at W3Schools](#) – nice tutorial with basic examples

[Creating a DTD for a certain XML Document](#) – video about 5:20 minutes showing a simple XML example and how to create a DTD based on the XML document

[Quackit.com DTD Tutorial](#) – sequence of pages with a nice tutorial for DTD

[XML Files – DTD Examples](#) – some examples of DTDs – this website also has a nice tutorial for DTD

Namespaces

We have learned how to create XML documents, how to display them, how to use CSS to style them and how to define the set of elements and attributes they can contain.

Imagine now that you want to combine an XML document with another developer's XML document! You might discover that both documents might be using the same name for some elements and if you end up combining those two XML documents, the source element data will become unclear and probably meaningless.

The solution is to group the element names of each document into their own space and when referring to a certain element, you would need to identify it with the space in which it resides. For example, I could call my space "**Claudia**" and identify the elements in my XML documents with the name of the space such as **Claudia:name** and the other developer's space could be called for example "**Marvin**" and the element **name**, in Marvin's space would then be called **Marvin:name**. Now, we can combine Claudia's and Marvin's XML documents and the **Claudia:name** element will not be confused with **Marvin:name** element.

That is the foundation of namespace and what I used, in the simple example described in the previous paragraph, was by the meaning of an identifier that is called namespace name. We will also see that the required format of a namespace name is more structured than what was shown in the simple example here.

Namespace Name

A namespace name must be unique and permanent and in XML they are written in the form of a URI (Uniform Resource Identifier).

To design a namespace name you would start with your domain name and then add a descriptive information (like a path in the URI) to create a unique name for your namespace – see the image below showing an example of a namespace name:

<http://www.domain.com/ns/myspace/1.0>

The **http://** is the protocol. The **www.domain.com** would be the domain name. Then you see **ns** that is just an optional namespace directory information. The **myspace** is the short description of the namespace followed by another forward slash and the optional version number. All of this to create a unique name.

When you use your own domain name, you can ensure that the name is unique as no one else can use your domain name. It is also a good idea to create a standard for yourself so then your namespace names are consistent and persistent (basically permanent).

There is some confusion over what is being pointed by the namespace name and although it might point to a DTD or XML Schema, it is something that is not required in the [W3C's XML Namespaces Recommendation](#), and, besides, even if the URI points to a file, the XML software does not even look at it!

Once you designed a namespace name, it can be declared as the default namespace for your XML document – for example:

```
<students xmlns="http://www.cdasilva.info/ns/student">
```

The **students** is the root element of a certain XML document. The **xmlns** means XML Namespace.

When you declare a default namespace for the root element, it means that all elements in the document are considered to be from that same namespace.

You can declare a default namespace for any element in your document and if you declare a default namespace to another element, it will override the namespace declared by any ancestor element – for example, look at **namespace1.xml** file that you can find among the files/folders you downloaded for this course. The element address is declared as part of the **http://www.cdasilva.info/ns/address** namespace and, because of that, **address**, **number**, **name**, **city**, **state**, and **zipcode** are all part of that namespace and this avoids the element conflict between the **name** element from the **student** namespace and the **name** element from the **address** namespace. This also means that by labeling an element with a default namespace, you are also affecting the child elements.

In case where a default namespace is not declared for a particular element, or it has not inherited a namespace from one of its ancestors, it is considered “in no namespace” and they can be assigned default namespaces which would override the no namespace state.

Using Prefix for Namespace

We already know that declaring a default namespace for an element will apply to this element’s children, but you might need to label specific elements with a namespace without affecting their children. That is when you would use a **prefix** that first you would declare that prefix for the namespace and then use it to label the individual elements you so desire.

To declare a prefix for a namespace, you would write **xmlns:prefix="namespace name"** instead of **xmlns="namespace name"** where **prefix** will be the desired prefix you want to use.

Look at **namespace2.xml** file – you will find this file among the files/folders you downloaded for this course – both namespaces are being declared at the root element **students**, one is being declared as a default with **xmlns=** and the other is being declared by using **add** as a prefix (identifier of that namespace) – the one declared with **xmlns:add=**. Then, the **add** prefix is being used in the XML document to identify the elements that belong to that particular namespace.

NOTE:

Prefixes cannot begin with the letters xml in any combination of upper or lowercase. Once you declare a prefix, you can use it in any element contained wherein you declared the prefix including the element itself where you declared the prefix. You can declare as many namespaces prefixes as necessary in any element. In our example, we labeled all child elements of **address** with the prefix to signal that those elements belong to that namespace

but if we had just labeled, for example, the **address** element with that prefix, the child elements would not be affected by that **address** namespace.

When you use prefixes, only those elements whose names are preceded by the prefix, are identified with the namespace declared with that prefix which is different from what we learned regarding default namespaces. The XML processor will consider unprefix elements to belong to the default namespace if one was defined or to no namespace if none was defined.

Notice that the closing tag of the prefixed element also contains the prefix!

Declare as default namespace the one that you notice is used the most throughout the XML document.

Look at a simple example using two different namespaces – **examplnamespace.xml** – you will find this file among the files/folders you downloaded for this course. Notice the default namespace declared for the elements that belong to **html4** namespace and the other namespace declared with prefix **xdc**. When you look at the XML, you will notice that whenever you have an unprefix element, it means it belongs to the **html4** namespace (default namespace) and the prefixed ones belong to **book** namespace.

Namespaces and Attributes

You do not need to prefix attributes because the attribute is physically contained within the element so, even if you would have for example a **student** element with **nbr** attribute and a **professor** element with an attribute named also **nbr**, there would be no confusion as the attributes belong respectively to each of their elements (**student** or **professor**). That is why you do not see attributes with prefixes and, therefore, they are considered to be “**in no namespace**” as default namespaces do not apply to attributes. In some books, or tutorials, you might hear that attributes are **locally scoped** which simply means that they are identified by the namespace of the element that they are contained.

So, although it is rare to find attributes with prefix, it does not mean it would be wrong – suppose you would combine two elements from different namespaces into a single element and both would have an attribute with the same name – this would be a case for you to differentiate each attribute with a prefix because an element cannot have multiple attributes with the same name.

Populating XML Namespaces

We have learned that XML Schemas specify the elements and attributes for an XML document to be considered valid, but they can also specify the elements and attributes contained in an XML namespace. The way to do that is by specifying the **target namespace** to which the elements and attributes will belong – this process is known as **populating the namespace**.

In the root element of the XML Schema you should type `targetNamespace="URI"` where URI is the namespace that is being populated.

It is important to know that when you populate a namespace only the **globally** defined (top-level) elements and attributes will be associated with the namespace (only the ones that are child of the **xs:schema** element –

could be an element, an attribute, a complex or simple type definition, a named group, or named attribute group).

For example – suppose the following XML Schema:

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.cdasilva.info/ns/order">

  <xs:element name="order">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="orderperson" type="xs:string"/>
        <xs:element name="shipto">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" 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>
        <xs:element name="item" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="note" type="xs:string" minOccurs="0"/>
              <xs:element name="quantity" type="xs:positiveInteger"/>
              <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="orderid" type="xs:string" use="required"/>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

See the **targetNamespace** being defined in red bold color – in that case the **order** element (the only one that is globally defined) is the one identified with the **http://www.cdasilva.info/ns/order** namespace and the other elements are not.

But as you populated a namespace, you need to adjust the relationship between the XML Schema and the XML document – the order.xml has already the code indicating the XML Schema's location it is using and specifying the location of an XML Schema that has populated namespace is very similar. So, the order element, in the order.xml element, would change to be:

```
<order orderid="72345"
  xmlns="http://www.cdasilva.info/ns/order"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.cdasilva.info/ns/order
  ordersns.xsd">
```

The ***xmlns*** declares the namespace for the XML document, the ***xmlns:xsi*** declares the namespace for the xsi-prefixed items and the ***xsi:schemaLocation*** declares the namespace and the actual file with which that namespace was populated. The ***orders.xsd*** could be in the same line, separated by a blank space, of the URI in the ***xsi:schemaLocation*** declaration but it is important to notice that there is NO closing double quotes after the URI as well as there is no opening double quotes before the name of the XML Schema ***.xsd*** file.

XML Schema Components in Namespaces

Components, in XML Schema, are named simple or complex types, elements, attributes, named groups (including attribute groups). Once you associate XML Schema components with a namespace (by using the process we mentioned before of populating namespace), you can refer to those components within that (or other) XML Schema. But since they are associated with a namespace, you MUST specify the namespace when you refer to them.

So, inside the XML Schema file (***.xsd*** file), inside the ***xs:schema*** root tag, you will add `xmlns="URI"` where ***URI*** is the namespace with which the components are associated. You still need to leave there the `xmlns:xs="http://www.w3.org/2001/XMLSchema"` as this is the namespace that is being referenced when you see the ***xs:string***, ***xs:complexType***, etc. in the XML Schema code. Then, any unprefixed global types in your XML Schema will be found in the XML Schema that corresponds to the default namespace (we know that the default namespace is declared with `xmlns=`).

You could also declare the namespace with a prefix such as `xmlns:prefix="URI"` where prefix is how you will be identifying the components in the XML Schema that belong to that namespace indicated by the URI. Then, you would use the ***prefix***: before the component name for the opening and closing tags of that component. Remember the difference of default namespace and prefixed namespace!

NOTE:

The target namespace allows you to populate that namespace with components while the namespace declaration (discussed in this part) allows you to identify components, in your document, that are part of that namespace. So, you would still see the `targetNamespace="URI"` attribute in the ***xs:schema*** root element!

Validating XML against the XML Schema with Namespace

We have populated an XML namespace and associated it with an XML document. We have also seen how to declare the namespace (the populated one), as default, in the XML Schema. Now, we are ready to validate an XML document against the XML Schema as both are now referencing the same namespace.

When we validated before, we did not worry about namespaces because none of the elements were ***qualified*** (associated with a target namespace). Once the elements are ***qualified***, then you will need to specify that namespace for those elements.

To write XML documents with qualified elements, you need to declare a namespace with a prefix – in the root element, of your XML document, you will use `xmlns:prefix="URI"` where **prefix** will identify the elements in this XML document that belong to the namespace indicated by **URI**. Then, you will prefix those elements with the namespace prefix.

You can also indicate the namespace of the elements using a default namespace declaration `xmlns="URI"` and then no prefix would be used in the elements, but this is not very common. You should only qualify those elements that are actually associated with their corresponding namespace.

Adding Locally Defined Elements

When populating a namespace with an XML Schema (remember the **targetNamespace** attribute we have already learned), only the globally defined components are associated with that target namespace by default. So, only the global components must be qualified or identified with a namespace name in a valid XML document.

You can also require that locally defined components (the ones that are one or more levels down from the root element) be added to the target namespace – they must be qualified then for the XML document to be considered valid.

You can add all the elements and/or attribute declarations at once – by adding, in the **xs:schema** element, the attribute `elementFormDefault="qualified"`, to add the attributes you would type, in that same **xs:schema** element `attributeFormDefault="qualified"`. The default value for these attributes is **unqualified** meaning that only globally defined components (the ones at the top-level) are associated with the target namespace by default.

It is considered best practices to add all locally defined elements when populating a namespace although this will override the default. For example, suppose you have the following XML Schema (.xsd file – showing just a part of it):

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
            xmlns="http://www.cdasilva.info/ns/monument"
            targetNamespace="http://www.cdasilva.info/ns/monument"
            elementFormDefault="qualified">
  <xs:element name="monuments">
    ...
```

Because we are using the `elementFormDefault="qualified"`, as shown in red bold color, the locally declared elements will be associated with the target namespace and as such, they must be written using a qualified name so the XML document will be considered valid. The XML document (showing just a part of it below) would be valid as all locally defined elements in the XML Schema are now qualified with the **mon:** prefix.

```
<mon:monuments xmlns:mon="http://www.cdasilva.info/ns/monument"
                xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
                xsi:schemaLocation="http://www.cdasilva.info/monument monument.xsd">
  <mon:monument>
    <mon:name language="English">The Man</mon:name>
    <mon:name language="Spanish">El Hombre</mon:name>
    <mon:pic file="man.jpg" w="120" h="120" />
    <mon:history>
```

```
<mon:yearbuilt era="BC">300</mon:yearbuilt>
```

...

We showed the example of adding all locally defined elements at once, but we can also add specific locally defined elements instead of adding all. To add a particular locally defined element to the target namespace, you would include, in the element's definition, the attribute `form="qualified"` regardless of where that element is defined and then you will associate it with the target namespace. The same way that you can add this attribute with the value *"qualified"*, you could add it with the value *"unqualified"* and then, even if you had the `elementFormDefault="qualified"` in the root ***xs:schema***, the element with `form="unqualified"` would not be associated with the target namespace.

Like elements, XML Schema attributes can use the same *form* attribute to identify if they should be associated or not with the target namespace regardless of the default you might have set (if you used `attributeFormDefault="qualified"` in the root ***xs:schema*** element).

XML Schemas in Multiple Files

The XML Schema's components can be divided into different files – you would do that if you would like to reuse the different files in different XML Schemas or simply to make it easier to handle big XML Schemas.

First, you would divide the components among different files – each file should be text-only and saved with the ***.xsd*** extension.

Then, you would reference the XML Schema document in which you wanted to include these components from the new file by adding `<xs:include schemaLocation="fileURI" />` right after the ***xs:schema*** element. The *fileURI* is the URI of the XML Schema document that contains the components you wanted to include.

It is important to notice that the value of ***targetNamespace*** attribute of the included XML Schema document must be the same as the value of that attribute of the XML Schema document that is receiving the components – later, we will see the way to include XML Schemas with different namespaces.

For example, you could have an XML Schema file called **demo.xsd** as:

```
<xs:schema xmlns="http://www.cdasilva.info/ns/includedemo"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.cdasilva.info/ns/includedemo"
  elementFormDefault="qualified" id="demo">
  <xs:include schemaLocation="demo1.xsd"/>
  <xs:element name="root" type="roottype"/>

  <xs:complexType name="roottype">
    <xs:sequence maxOccurs="unbounded">
      <xs:element name="record" type="recordtype"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="recordtype">
    <xs:annotation>
      <xs:documentation>this is the main element.
```

```

        </xs:documentation>
    </xs:annotation>
    <xs:sequence>
        <xs:element name="id" type="xs:integer" />
        <xs:element name="name" type="xs:string" />
        <xs:element name="otherroot" type="otherroottyp" />
    </xs:sequence>
</xs:complexType>
</xs:schema>

```

Notice the **xs:include** element referring to another **.xsd** file (another schema) called **demo1.xsd** and the XML Schema code in the **demo1.xsd** could be something as:

```

<xs:schema xmlns="http://www.cdasilva.info/ns/includedemo"

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

    targetNamespace="http://www.cdasilva.info/ns/includedemo"

    elementFormDefault="qualified" id="demo1">
    <xs:element name="otherroot" type="otherroottyp"/>

    <xs:complexType name="otherroottyp">
        <xs:sequence maxOccurs="unbounded">
            <xs:element name="otherrecord" type="otherrecordtyp"/>
        </xs:sequence>
    </xs:complexType>

    <xs:complexType name="otherrecordtyp">
        <xs:annotation>
            <xs:documentation>
                this is the main element.
            </xs:documentation>
        </xs:annotation>
        <xs:sequence>
            <xs:element name="id" type="xs:integer" />
            <xs:element name="name" type="xs:string" />
        </xs:sequence>
    </xs:complexType>
</xs:schema>

```

Notice that the value of the **targetNamespace** attribute in **demo1.xsd** is equal to the value of that attribute in **demo.xsd** that is the XML Schema that is including **demo1.xsd**.

A valid XML document would look something as:

```

<?xml version="1.0"?>
<root xmlns="http://www.cdasilva.info/ns/includedemo"

    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

    xsi:schemaLocation=
        "http://www.cdasilva.info/ns/includedemo demodata.xsd">
    <record>
        <id>1</id>
        <name>John Doe</name>
        <otherroot>

```

```

        <otherrecord>
            <otherid>0</otherid>
            <othername>Jane Doe</othername>
        </otherrecord>
    </otherroot>
</record>
</root>

```

The schemas are in the same namespace – <http://www.cdasilva.info/ns/includedemo> - and this namespace is also set as the default namespace for both schemas (notice the value of the *xmlns* attribute in the *xs:schema* root of both XML Schemas). Since the XML Schemas share a namespace, each root element must then have a unique name, but other elements or types within a schema could share names with elements or types found in another schema in the same namespace. All elements are treated as “*qualified*” – see the `elementFormDefault="qualified"` you find in both XML Schemas in the root *xs:schema* element.

The **demo1.xsd** schema does not depend on the existence of the **demo.xsd** schema which means that it could be used separately to validate another XML structure (where `<otherroot>` would be the root element and that element, could contain an unlimited number of *otherrecord* elements (notice the *maxOccurs* with the value of “*unbounded*” in the *xs:sequence* that defines the *otherrecord* element in **demo1.xsd**).

The **demo1** schema is included in the **demo** schema by using the *xs:include* statement and then it refers the *otherroottype* from **demo1.xsd** as the type for the *otherroot* element (marked in green bold color in **demo.xsd** code).

Using XML Schemas with Different Namespaces

If you wanted to use XML Schema components from another XML Schema that would not have the same target namespaces, you would use the *xs:import* instead of the *xs:include*. The way to write the *xs:import* statement is:

```
<xs:import namespace="URI" schemaLocation="schemaURI" />
```

where **URI** is the namespace name of the XML Schema being imported and *schemaURI* is the location of the file that contains the XML Schema that defines the namespace.

Example:

```

<?xml version="1.0"?>
<xs:schema xmlns="http://www.cdasilva.info/ns/monument"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.cdasilva.info/ns/includedemo"
    xmlns:old="http://www.cdasilva.info/ns/oldmonument">

    <xs:import namespace="http://www.cdasilva.info/ns/oldmonument"
        schemaLocation="oldmonument.xsd"/>
    ...

    <xs:complexType name="monument">
        <xs:sequence>
            <xs:element name="name" type="nameType" maxOccurs="unbounded" />
            <xs:element name="location" type="old:locType" />
        </xs:sequence>
    ...

```

The definition of the type of the *location* element is coming from another schema – *oldmonument.xsd* – that belongs to a different namespace (*http://www.cdasilva.info/ns/oldmonument*) which is not the same namespace that this XML Schema presented belongs to (*http://www.cdasilva.info/ns/monument*). That is the reason to be using *xs:import* instead of *xs:include* and also using the prefix set for the namespace (which was set to be *old*) whenever you need to refer to something related to the schema that is being imported.

The Schema of Schemas

We learned that an XML Schema is, in reality, an XML file and that is why the first line of the XML Schema is the XML declaration `<?xml version="1.0"?>`, then, the second line of the XML Schema is the *xs:schema* element which is the root element and we have been using the *xs:* prefix to declare a namespace that is the W3C's XML Schema namespace. This namespace contains definitions for components that you have seen in most of our examples such as: *xs:element*, *xs:attribute*, *xs:complexType*, etc. But the specification of the XML Schema does not require the use of *xs:* prefix (we could even use another prefix such as *foo:*, *w3c:*, etc. and *xs:* is used mainly because it is a practice in the market to use that prefix for the W3C's XML Schema namespace). In reality, if your XML Schema is mostly made of built-in types, it may be easier and quicker to declare the W3C's XML Schema namespace as the default namespace instead of declaring it with the *xs:* prefix and then you would not need to use the *xs:* prefix for the components of the schema that belong to that namespace.

If you would do that, your XML Schema code could become something as:

```
<?xml version="1.0"?>
<schema xmlns:mon="http://www.cdasilva.info/ns/monument"
  xmlns="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.cdasilva.info/ns/includedemo">

  <element name="monuments">
    <complexType>
      <sequence>
        <element name="monument" type="mon:monumentType" maxOccurs="unbounded" />
      </sequence>
    </complexType>
  </element>
  ...
```

Notice in green bold color that now the prefixed namespace is not the W3C's XML Schema namespace which is now being set as the default namespace by using *xmlns=*. That is why you do not see *<xs:element>* and, instead, you see *<element>* as this component belongs to the W3C's XML Schema namespace that, for this example, was declared as default. Also notice that the value of *type* that defines the element *monument* is now prefixed as it belongs to the other namespace.

Final Observation

One of the biggest complaints about DTDs is that they do not support XML namespace declarations. For sure, you can even prefix an element in your DTD but the parser will not know which XML namespace that prefix refers to. In reality, the parser does not even see the characters before the colon (:) as a prefix, they just see as part of the

name of the element and this alone has been one of the main reasons that XML Schema are more used than DTDs.

Regarding XSLT we have also learned that the first line of the XSLT Style Sheet is an XML declaration which means that XSLT Style Sheets are XML documents as well. And with this in mind, if you are working with XML elements that belong to a certain namespace, you will need to declare the namespace and prefix each of those elements in your XSLT file as well. To use an XML namespace in an XSLT file, you need to include `xmlns:prefix="URI"` in the root element of the XSLT which is `<xsl:stylesheet>`, where **URI** is the name of the XML namespace to which the **prefix** will refer to. Then, you will label individual elements as necessary by typing `<prefix:element>`. For example, the XSLT Style Sheet for the XML with a declared namespace would look something as:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
    xmlns:mon="http://www.cdasilva.info/ns/monument">
<xsl:template match="/">
    <html><head><title>Monuments</title></head>
    <body>
    <h1>Great Monuments</h1>
    ...
    <p>These are the great monuments
    <xsl:for-each select="mon:monuments/monument/name[@language='English']">
        <xsl:value-of select="." />
    ...
```

This partial XSLT shows the declaration of the namespace **`http://www.cdasilva.info/ns/monument`** with the prefix **`mon`** and because the **`monuments`** element belongs to that namespace, it has been prefixed with **`mon:`**

NOTE:

In the case of XSLT you cannot remove the prefix **`xsl`** from the stylesheet's elements like the way you can do with the XML Schema (by declaring the W3C's XML Schema namespace as the default namespace) because the default namespace in an XSLT Style Sheet is not used for unprefixed names.

Extra Resources

[Validate XML using two different XML Schemas and namespaces](#) – it is a simple example but shows the code and explains the code

[Understanding Namespaces \(oracle.com\)](#) – nice basic explanation using some examples of valid and invalid declarations

[Freeformatter.com](#) – to validate XML against XSD (Schema)

[W3Schools Intro to Schemas](#) – for a review of schemas

[W3Schools Intro to Namespaces](#) – light introduction to namespaces

[Examples of XML Schema with namespaces](#) – interesting examples

[Namespace and XSD Tutorial from Codeproject.com](#) – good example using an XML file and different schemas

[Video \(about 3:40min\) with intro to namespaces](#)

[Creating and using namespace \(video about 4:40min\)](#)