Internet-Based Information Systems

Table of Content:

1 Internet (Overview)
2 Internet-Based Applications
   2.1 Server-Side Scripting
   2.2 Client-Side Scripting
3 PHP-Hypertext Preprocessor
   3.1 PHP Basics
   3.2 Interface to a DBMS
4 Document Object Model and Java Script
   4.1 Java Script Basics
   4.2 Document Object Model
5 XML-eXtensible Markup Language
   5.1 XML Basics
   5.2 Well-Formed XML Document
   5.3 Document Type Definition (DTD)
      5.3.1 Element Type Definition (EDT)
      5.3.2 Element Attributes (ATTLIST)
   5.4 XML Schema
      5.4.1 Name Spaces
      5.4.2 Simple Element Types
      5.4.3 Complex Element Types
      5.4.4 References
6 XSL-eXtensible Stylesheet Language
   6.1 Namespaces
   6.2 Transforming XML Documents
   6.3 Formatting HTML Documents (CSS)
   6.4 Formatting XML Documents
7 Linking XML Resources
   7.1 XML Linking Model
   7.2 XML Link Namespaces
   7.3 xLink Notation
   7.4 xPointer Notation

Sample Courseware

Internet-Based Information Systems

*Internet* is the largest world-wide computer network that exists today. It is in fact a network of networks that is estimated to connect several million computers and with over 100 million individual users around the world - and it is still growing rapidly
Internet (Overview)

A notable feature of the Internet is that it brings together multiple hardware and operating system platforms from dozens of different manufacturers. Clearly, communication between these different platforms would not be possible unless they agree to some way of exchanging data. The Internet Protocols define such data exchange schemes, comprising two kinds of standards:

First is TCP/IP, which is an acronym for Transmission Control Protocol/Internet Protocol.

TCP/IP specifies the data transport layer of communication, which treats a data transaction between two computers as a stream of bytes referred to as a transport data unit. Simply put, data exchange between any two computers on the net is supported by TCP/IP if the data is sent in one or more transport data units.

Internet Data Service protocols are used by internet applications. There are a number of such protocols, each designed for some particular purpose. There are special protocols, for example, to support distributed collaborative hypermedia systems (HTTP), Internet News System (News) and File Transfer Systems (FTP).

HyperText Transfer Protocol (HTTP) is an example of an Internet Data Service protocol. It is designed to support communication between clients and a hypermedia information server.

- Clients send requests for certain services to a server.
- The server responds by sending back relevant data to the clients.
Some requests can also cause side effects in the information maintained by the server, such as addition or deletion of certain documents. HTTP basically defines the internal structure of supported requests and responses.

**The World Wide Web (WWW)** is a globally distributed collection of so-called WWW documents. These are in fact documents written in a mark-up language called a **HyperText Mark-up Language (HTML)**.

The pages residing on some particular host machine are made accessible over the net through HTTP. In other words, the WWW architecture is essentially that of multiple HTTP servers on the Internet serving WWW pages to HTML clients.

**The Uniform Resource Locator (URL)** is one of the most important Internet concepts. It may be viewed as a means of uniquely identifying resources on the net. In HTTP, URLs identify the data to be transmitted.

HTML allows for URLs to be embedded in its pages. This is the basic linking mechanism in WWW: the embedded URLs typically point to other HTML pages.

Thus the **World Wide Web (WWW)** can be seen as a distributed collection of multi-media (HTML) documents interrelated by means of computer-navigable links. The fact that HTML is the WWW de facto standard for describing how information is structured and displayed underlines its importance to the web architecture. It allows different vendors to develop **WWW browsers** that, while running on different hardware and software platforms, still display web pages in approximately the same way.
A mark-up code is simply an ASCII character sequence distinct from the text. Typically, text is bracketed by a start code and an end code, and the text thus enclosed is subject to the properties that the code describes. HTML mark-up codes are called **HTML tags** and are distinguished from text by adopting the following notation:

- a start tag is written as "<tag-X>" where tag-X is some reserved code identifier
- the corresponding end tag is written as "</tag-X>"

```html
<TAG-X> Text bracketed by TAG-X</TAG-X>  
<TAG-Y> Text bracketed by TAG-Y</TAG-Y>
```

HTML tags may be used in combination to achieve multiple text emphasis effects: eg.

```html
<B> <I> bold and italics <U> and underlined; </U> </I> </B>  
<BR>  
<FONT size=+2> this line is not underlined and 2 sizes larger; </FONT>  
<BR>  
and this is back to normal, unemphasised text
```

will display something like the following:

```
bold and italics and underlined;  
but this line is not underlined and 2 sizes larger;  
and this is back to normal, unemphasised text
```

An HTML document would not be a multimedia document if it only handles text. Other media objects are introduced as so-called inline objects. These objects exist as files that are separate from an HTML document and are included at appropriate points using special tags.

An image is included using the tag

```html
<IMG SRC="lesson08/file name" ... >
```
As mentioned earlier, a multimedia document becomes a hypermedia document with the addition of hypertext-style links. Links specified in HTML allows the browser to navigate to either a new point in the same document or to a different document.

Links are introduced using the anchor tag:

```html
<A HREF="URL"> anchor </A>
```

Internet-Based Applications

Internet is based on the **client-server architecture**. There are two main methods for developing Internet-Based Information systems:

- Server-Side programming (scripting)
- Client-Side programming (scripting)

Server-Side Scripting

Most queries currently made to WWW servers fetch static data stored in a portion of the file system associated with the server.

The **CGI interface** provides a means for a client to request that an arbitrary program be executed by the server. The reason for running that program can be to produce side effects, such as updating a data base or sending e-mail to someone, but more often the program is run in order to
return data directly to the client/user in the form of an HTML document generated by the program.

The CGI interface provides a very powerful mechanism for building so-called Internet-Based Information systems.

It should be especially noted that CGI applications may communicate to a file system and other software packages installed on the server. For example, CGI scripts may provide an internet access (i.e. interface) to a big local database, expert system, etc.

Generally, a CGI script is invoked by an HTTP request looking as follows:

http://[Uniform Resource Locator of the script] ? [parameters]

Parameters are passed to a CGI application as a value of special environment variable "QUERY_STRING".
Values are assigned to environment variables by the server before the CGI program begins execution and, thus, are available to it when it begins.

For example:

```
http://coronet.iicm.edu/cgi-bin/getMail.cgi ? Name=Nick&City=Graz
QUERY_STRING="Name=Nick&City=Graz"
```

Parameters are typically sent as a result of processing a so-called HTML FORM.

It often represent a query string, such as a query to a database, depending on the function of the FORM. You can, of course, manually enter parameters directly in the URL. for example:

```
<A HREF="http://coronet.iicm.edu/cgi-bin/sentMail.cgi?Name=Nick&Topic=Important">Click here to run it</A>
```

A form is introduced by the tag `<FORM>` and terminated by the inverse tag `</FORM>`. The attributes of the `<FORM>` tag includes METHOD and ACTION. For example:

```
<FORM METHOD=GET ACTION="http://host/cgi-bin/script_name">
</FORM>
```

- METHOD specifies which technical protocol the web server will use to pass the form data to the program that processes it, and
- ACTION tells the server exactly which program that is.

A form field to request the user to enter text that is to be sent to the CGI script is introduced by the following tag:

```
<INPUT TYPE="text" NAME= "name of CGI script parameter" SIZE="width of the input area">
```

Note that the input data is sent to the CGI script in the form

```
"Name of the parameter" = "Entered Value"
```

The CGI script processes the entered data and responds with a new HTML document

If a particular form contains multiple elements, the following tag is used to pass the submission of the input data to the CGI script:

```
<INPUT TYPE= "submit" NAME="parameter" VALUE="Value if pressed">
```
The button when pressed will send, in addition to any information entered in the form, the message "parameter"= "Value if pressed".

Note that there may be several of these input tags within a form. The VALUE attribute identifies which button, i.e. <INPUT> has been selected. When the user clicks the "submit" button, the browser collects the values of each of the input fields and sends them to the web server identified in the ACTION keyword of the FORM open tag. The web server then passes that data to the program identified in the ACTION, using the METHOD specified.

Server-side Internet Programming Languages:

- PERL
- Java Script
- Java Servlets
- PHP

Client-Side Scripting

Actually, Internet Browsers are also much more complex software systems than just an HTML interpreter as we saw it before.

Applets are normally small software applications, but they do not run standalone. Instead, applets comply to a set of conventions that lets them run within a Java-compatible browser on the WWW client.

Applets are embedded directly to HTML code using tags looking as follows:

```html
<applet code = "x.jar"
    width = "number of pixels" height = "number of pixels">
    <param name="a" value="b">
</applet>
```
Thus a WWW client can fetch an applet from a server site and run it locally to provide any kind of visual effects and/or interaction that is needed.

Whenever a browser encounters the applet tag

```
<applet code = "x.jar"
width = "number of pixels" height = "number of pixels">
<param name="a" value="b">
</applet>
```

it is rendered as follows:

- 1. A rectangle space defined by the width and height parameters is reserved on the screen;
- 2. A new virtual machine is activated and the reserved space is allocated for such machine to be used as a virtual display window;
- 3. The code is rendered by the virtual machine using parameters predefined by the applet tag.

Scripts are just fragments of source code which are embedded directly into HTML documents. The code is interpreted directly by an internet browser. Scripts are embedded directly into HTML code using tags looking as follows:

```
<SCRIPT>
...
</SCRIPT>
```

Thus a WWW client does not need to additionally fetch scripts from a server.
On the first glance the scripting technique seems to be very similar to applets discussed early. In reality, these two methods are essentially different:

- applets run more or less independently of an HTML document. Browser just allocates a virtual screen for an applet and let the virtual machine to control it. There is no way of accessing the document elements, or to modify them.
- client-site scripts may easily access elements of a current document to modify them (say, alter links, images, textual fragments, etc.)

PHP-Hypertext Preprocessor

**PHP** (recursive acronym for "**PHP: Hypertext Preprocessor**") is a widely-used Open Source general-purpose server-side scripting language that is especially suited for Web development. There are three PHP features that make it, perhaps, a most popular tool for developing information systems based on the Internet:

- embedding PHP scripts into ordinary HTML pages what allows to combine expressive power of both languages.
- flexible interface to many modern Database Management Systems (MySQL, Oracle, Sybase, mSQL, Generic ODBC, and PostgreSQL)
- possibility to dynamically output images and other multi-media files

PHP Basics

**PHP** is what is known as a **server-side scripting language**. Thus the language interpreter must be installed and configured on the server before one can execute commands. Now, we assume that your Web server has the PHP support activated and that all files with the extension .php3 are handled by PHP interpreter. If that's the case just create .php3 files, put
them somewhere in your Web server directory and the server will parse them on a request, outputting whatever the result of the execution may be back to the client. There is no need to compile anything.

So, let us start, as so many times before, with a file called `hello.php3` that will produce a simple output: "Hello, World" enclosed by some HTML tags. The code of a PHP program may look as follows:

```html
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<B>I say <? PRINT "Hello, World"; ?></B>
</body>
</html>
```

The PHP interpreter returns the following HTML file:

```html
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<B>I say "Hello, World"
</B>
</body>
</html>
```

Alternatively, the PHP script may be embedded into HTML using tags looking as follows:

```html
<html>
<head>
<title>PHP Test</title>
</head>
<body>
<B>I say
<script language="php">
 PRINT "Hello, World";
</script>
</B>
</body>
</html>
```

Variables in PHP are represented by a dollar sign followed by the name of the variable. The variable name is case-sensitive.

```php
$var = "Nick";
```
$A = "Denis";
echo "$a, $A";  // outputs "Nick, Denis"
?>

In PHP, variable types are always assigned by types of values. PHP control statements are almost identical to control statements in C and Java programming languages. (See, for example, "while" control statement below)

<?
$i = 0;   // integer
$length = 3;
$A[0] = "First";   // array of strings
while ($i < $length)
{
    echo "$A[$i]"
    echo "&ltBR&gt;";
    $i++;
}
?>

The script above would return the following HTML fragment:

First&ltBR&gt;Second&ltBR&gt;Third&ltBR>

Consider the following HTML form:

<form action = "action1.php3" method = "POST">
Name: <input type = "text" name = "name" size = "20">
<br>
I prefer:  
<select name = "preference">
<option value = Movies>Movies
<option value = Music>Music
<option value = Theater>Theater
</select>
<br>
<input type = "submit" value = "Send it!" >
</form>

After entering the requested info and pressing "Send it!" button:
The client will send the following HTTP request to the server:

http://[host]/[path]/action1.php3?name=[value]&preference=[value]

For example:

Name: Nick  
I prefer: Theater 
Send it!

Would produce:

http://[host]/[path]/action1.php3?name=Nick&preference=Theater

The HTTP request:

http://[host]/[path]/action1.php3?name=[value]&preference=[value]

Is interpreted as follows:

- Server creates two environment variables: $name and $preference with values received as a part of the request.
• Server invokes the script \texttt{action1.php3} from the specified directory.
• Variables $\texttt{name}$ and $\texttt{preference}$ can be processed by PHP imperative statements as ordinary global variables.

The script will handle the variables passed from the form mentioned above:

```php
<?
echo "<center>";
echo "Hello, $name."
echo "<br>";
echo "You like $preference.<br>";
echo "Thank you for your cooperation.";
echo "</center>";
?>
```

Consider the HTTP request once again:

\texttt{http://[host]/[path]/action1.php3?name=[value]&preference=[value]}

More systematic way of processing input parameters is offered as two global arrays $\texttt{$_POST}$ and $\texttt{$_GET}$. These arrays contain all parameters sent by methods \texttt{POST} and \texttt{GET} respectively.

Thus, the script can handle the variables passed from the form mentioned above:

```php
<?
$name = $\texttt{$_GET["name"]};
$preference = $\texttt{$_GET["preference"]};
echo "<center>";
echo "Hello, $name."
echo "<br>";
echo "You like $preference.<br>";
echo "Thank you for your cooperation.";
echo "</center>";
?>
```

A function may be defined using syntax such as the following:

```php
<?
function fact ($arg)
{
    $retval = 1;
    var $i = 1;
    while ($i <= $arg)
    {
        $retval = $retval*$i;
        $i++;
    }
    return $retval;
}
$f3 = fact (3);
echo "$f3";
?>
```
PHP supports rather powerful library of predefined functions. There are functions that you may use to send emails, open network connections or calculate trigonometric functions. A big family of standard PHP functions allows to manipulate with data residing on different database servers, such as MySQL server, Oracle server, etc. As a very simple example, we can call a standard PHP function called "date". This function returns the current date in a specified format:

```php
<?
	$today = date("Y-m-d");
	echo "<center>";
	echo "Hello, $name.";
	echo "<br>";
	echo "You like $preference.<br>";
	echo "Today is: $today";
	echo "</center>";
?>
```

**Interface to a DBMS**

Standard PHP distribution comes with a number of standard functions which allow scripts to communicate to a wide range of currently popular database management systems (DBMS). There are, for instance, function libraries for manipulating MySQL databases, Oracle databases, Informix database and others.

Normally a database transaction is carried out as the following sequence of actions:

- connect to a DBMS (there may be a DBMS installed on the same server or on another Internet Server);
- select a database (there may be a number of databases accessible via a single DBMS);
- send a query as a string to the DBMS;
- get a result as an array of tuples;
- disconnect;

Consider the following database:

- **Customer(C#,Cname,Ccity,Phone)**
- **Product(P#,Pname,Price)**
- **Transaction(C#,P#,Date,Qnt)**
Suppose, the database is supported by MySQL DBMS.

A simplest query:

"Get product names for products bought by customer number 1"

is implemented by the following script:

```php
$hostname = "localhost";
$username = "student";
$password = "student";
$dbName = "MyFirm";
MYSQL_CONNECT($hostname,$username,$password);
MYSQL_SELECT_DB("$dbName");
$query = "SELECT Pname FROM Product,Transaction";
$query = "$query WHERE C# = 1 AND Product.P# = Transaction.P#";
$result = MYSQL_QUERY($query);
```

Obviously, the script can generalized to allow users to input arbitrary customer number (C#) and select products bought by the customer.

**User Interface = HTML Form**

```html
<form action = "query.php3" method = "POST">
Customer: <input type = "text" name = "cnumber" size = "3">
<input type = "submit" value = "Send it!" >
</form>
```

From a programmer's point of view, the query result is a two-dimensional table where
• rows are addressed by an index
• columns are addressed by a unique name

```php
<?php
$hostname = "localhost";
$username = "student";
$password = "student";
$dbName = "MyFirm";
MYSQL.Connect($hostname,$username,$password);
MYSQL.Select_DB("$dbName");
$query = "SELECT * FROM Product";
$result = MYSQL.Query($query);
?>
```

<table>
<thead>
<tr>
<th>$result</th>
<th>P#</th>
<th>Pname</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>CPU</td>
<td>1000</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>VDU</td>
<td>1200</td>
</tr>
</tbody>
</table>

The table can be processed by means of two functions:

- **MYSQL_NUMROWS** returns a total number of the table rows
- **MYSQL_RESULT** returns a value of particular table element

Thus,

- **MYSQL_NUMROWS($result)** returns "2"
- **MYSQL_RESULT($result, 0, "Pname")** returns "CPU"
- **MYSQL_RESULT($result, 1, "P#")** returns "2"
- **MYSQL_RESULT($result, 1, "Price")** returns "1200"

Generally speaking, the result should be returned to the client in a form of a correct HTML file.

```php
...
$query = "SELECT Pname FROM Product,Transaction";
$query = "$query WHERE C# = $cnumber AND";
$query = "$query Product.P# = Transaction.P#";
$result = MYSQL.Query($query);
$r = MYSQL_NUMROWS($result);
$i = 0;
IF ($r == 0)
echo "Customer $cname bought no products";
ELSE
{echo "Customer $cname bought the following products<UL>";
WHILE ($i < $r)
{
```
Data base relations can be updated using the same scripting paradigm

```
<?php

 $hostname = "localhost";
 $username = "student";
 $password = "student";
 $dbName = "MyFirm";
 MYSQL_CONNECT($hostname,$username,$password);
 MYSQL_SELECT_DB("$dbName");
 $query = "INSERT INTO Product";
 $query = "$query VALUES('$Pnumber','$Pname', '$Prise')";
 $status = MYSQL_QUERY($query);
 ?>

```

**Document Object Model and Java Script**

*Java Script* is, perhaps, a most popular client-side scripting language.

There are two main reasons for embedding Java scripts into HTML pages:

- dynamic generation of HTML fragments directly on a client-site
- dynamic manipulation with elements of the HTML document (so-called Document Objects)

Java Script Basics

Java Script is a **client-side scripting language**. Hence, Java Script fragments are embedded directly into an HTML document, and are interpreted by the browser at the same order as other components of the document. Java Script fragments should be enclosed into `<SCRIPT>...</SCRIPT>` tags. For example:

```html
<html>
<head>
<title>Java Script Test</title>
</head>
<body>
<B>I say</B>
<SCRIPT LANGUAGE="JavaScript">
document.write(""Hello, World!"");
</SCRIPT>
</B>
</body>
</html>
```

The document will be displayed as follows:

I say "Hello, World";

In Java Script, variables may be explicitly created as objects of a particular class, or via assigning values. For example:

```javascript
<SCRIPT LANGUAGE="JavaScript">
var doc = new Object();
var s = new String();
var A = new Array("First","Second","Third");
var al = 3;
i = 0;
</SCRIPT>
```

Java Script control statements are almost identical to control statements in C and Java programming languages. (See, for example, "while" control statement below)
The document will be displayed as follows:

First
Second
Third

A Java Script function may be defined using syntax such as the following:

```
<SCRIPT LANGUAGE="JavaScript">
function fact (arg)
{
    retval = 1;
    i = 1;
    while (i <= arg)
    {
        retval = retval*i;
        i++;
    }
    return retval;
}

document.write("Factorial(3)=" + fact(3) + ";
document.write("Factorial(5)=" + fact(5) + ");
document.write("Factorial(7)=" + fact(7) + ");
</SCRIPT>
```

The document will be displayed as follows:
Factorial(3)=6  
Factorial(5)=120  
Factorial(7)=5040

There three ways to invoke a Java Script function:

- to call it from another Java Script function;
- to activate a hypertext link pointing to the function;
- to associate the function with an event;

Consider, for example, the following function:

```<SCRIPT LANGUAGE="JavaScript">
 function fact (arg) {
     retval = 1;
     i = 1;
     while (i <= arg) {
         retval = retval*i;
         i++;  
     }
     alert(retval);
 }
</SCRIPT>```

The function can be invoked:

- From another Java Script function:
  ```<SCRIPT LANGUAGE="JavaScript">
  fact(3);
  </SCRIPT>```
- By means of a hypertext link:
  ```<A HREF="javascript:fact(3)">Click</A> (Click)```
- If a certain event occurs:
  ```<form>
  <input type="button" value="Calculate" onClick="fact(6)">
  </form>```

### Document Object Model

When a modern Internet Browser parses an HTML document, it builds a so-called **Document Object Model (DOM)**.

The document is considered to be an **hierarchy of objects**. Each object belongs to a particular class (HTML fragment, form, image, applet, etc.) and may consist of other objects (Children).
Each object **Document Object** has

- a unique **identifier** (name or index);
- a collection of **properties** (position, size, visibility, etc.);
- a number of **methods** which are used to access/modify properties.

Properties are inherited along the object hierarchy.
For example:

```html
<form name="X">
    <input name="a" type="text" value="My Text" size=20">
    <input name="b" type="button" value="Display" onClick="display()">
</form>
```

Properties of objects can be accessed by a Java Script using the following notation:

```
[Object].[Property]
```

For example:

```javascript
function display()
{
    alert("document.X.a.value=" + document.X.a.value);
    alert("document.X.b.value=" + document.X.b.value);
}
</script>
```

```html
<form name="X">
    <input name="a" type="text" value="My Text" size=20">
    <input name="b" type="button" value="Display" onClick="display()">
</form>
```
In a similar way, properties of objects can be dynamically modified using the following notation:

\[
\text{[Object].[Property]} = \text{[Value]}
\]

For example:

```html
<form name="X">
<input name="a" type="text" value="My Text" size=20">
<input name="b" type="button" value="Modify" onClick="modify()">
</form>

The HTML tag `<DIV>` allows to declare any HTML fragments as an object having certain properties.

```html
<DIV ID="Y" STYLE="position:absolute; background:#FFFF00; left:100; top:300;">
  <A HREF="javascript:modify()">
    <B>My Animated Object</B>
    <IMG SRC="lesson08/batter.gif" Border=0>
  </A>
</DIV>
```

As usual, Java Script can dynamically modify properties of such object.

```javascript
var x,y;
var o = new Object();
function modify(){
  o = document.getElementById("Y");
  x = 100; y = 400;
  o.style.left = x; o.style.top = y; move();
}function move()
  o.style.left = x; o.style.top = y;
if (x <= 200){x++;y--;setTimeout("move()");}
```

```html
<DIV ID="Y" STYLE="position:absolute; background:#FFFF00; left:100; top:300;">
  <A HREF="javascript:modify()">
    <B>My Animated Object</B>
    <IMG SRC="lesson08/batter.gif" Border=0>
  </A>
</DIV>
```

**XML-eXtensible Markup Language**

XML is one of the more famous computer markup systems. HTML defines a set of tags that associate formatting rules with bits of text.
We can say that syntax and semantics are fixed and can be more suitable for some applications (home page, reference manual, etc.), and less suitable for others (e-Learning, e-Commerce, Vector Graphics, mathematical documents, chemical documents, etc.)

Documents which have been marked up (which contain plain text as well as the tags that specify the rules for formatting that text) are read by an HTML processing application (a web browser for example) that knows how to display the text according to the rules.

XML Basics

Like HTML, XML (also known as Extensible Markup Language) is a markup language which relies on the concept of rule-specifying tags and the use of a tag-processing application that knows how to deal with the tags.

Specifically, rather than providing a set of pre-defined tags, as in the case of HTML, XML specifies the standards with which you can define your own markup languages with their own sets of tags. XML is a meta-markup language which allows you to define an infinite number of markup languages based upon the standards defined by XML.

Let us imagine a language suitable for encoding information about customers: Thus, the language will define tags to represent customers and information about customers.

The set of tags will be simple. However, they will be expressive. XML tags can be immediately understood just by reading the document.

```xml
<Customer>
  <ID>001</ID>
  <Name>Nick Scherbakov</Name>
  <Company>Interactive Internet (I2)</Company>
  <Email>nick@i2.ac.at</Email>
  <Phone>662-9999</Phone>
  <City>Graz</City>
</Customer>
```

For the data to be decoded by someone or something else, the encoding markup languages must follow standard rules including:
• The syntax for marking up
• The meaning behind the markup

In other words, a processing application must know what a valid markup is (perhaps a tag) and what to do with it if it is valid?

For example, how does an application know whether the following markup valid or not?

```xml
<EMAIL>nick@i2.ac.at</EMAIL>
<PHONE>662-9999</PHONE>
<CUSTOMER>
  <ID>001</ID>
  <NAME>Nick Scherbakov</NAME>
  <COMPANY>Interactive Internet (I2)</COMPANY>
  <CITY>Graz</CITY>
</CUSTOMER>
```

Our markup language must somehow communicate the syntax of the markup so that the processing application will know what to do with it.

In XML, the definition of a valid markup is handled by a **Document Type Definition (DTD)** which communicates the structure of the markup language.

The DTD specifies what it means to be a valid tag (the syntax for marking up).

For example, the following DTD:

```xml
<!ELEMENT CUSTOMER (ID, NAME, COMPANY, CITY, PHONE, EMAIL)>
<!ELEMENT ID (#PCDATA)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT COMPANY (#PCDATA)>
<!ELEMENT CITY (#PCDATA)>
<!ELEMENT PHONE (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
```

tells to a processing application that the markup:

```xml
<Customer>
  <ID>001</ID>
  <NAME>Nick Scherbakov</NAME>
  <COMPANY>Interactive Internet (I2)</COMPANY>
  <CITY>Graz</CITY>
  <PHONE>662-9999</PHONE>
  <EMAIL>nick@i2.ac.at</EMAIL>
</CUSTOMER>
```

is valid.

Yet we must also communicate the meaning of the markup as well as the syntax.

To specify what valid tags mean, XML documents are also associated with style sheets which provide GUI instructions for a processing application like a web browser.

In this example, the style sheet utilizes the functionality of HTML to define the formatting of "CUSTOMER" documents. But if the XML document was being processed by a program other than a web browser, the HTML translation step might be bypassed.

For example, the following style sheets:

```xml
<xsl:template pattern = "CUSTOMER">
  <UL><xsl:process-children></UL>
</xsl:template>
<xsl:template pattern = "ID">
  <LI><I><xsl:process-children></I></LI>
</xsl:template>
```

Processing applications combine the logic of

- the *style sheet*,
- the *DTD*, and
- the data of an *XML document*

and display it according to the rules and the data.

Once you have defined your DTD and XSL documents, you can define arbitrary number of XML documents, and run them through a processor to achieve a desired functionality.

That means we have three documents to pull together plus one processing program to write or buy. "A software module called an *XML processor* is used to read XML documents and provide access to their content and structure."

**Well-Formed XML Document**

As was discussed in part one, XML allows you to generate an infinite number of custom tags sets for your documents.

However, though you are free to be as innovative as you want with the XML tag sets that you create, you must follow the constraints of the XML tag set generation standards exactly. In other words, your *XML documents must be "well-formed"*. Specifically, a well-formed document must follow the XML standard.

Thus, rather than pre-defining a set of tags, XML defines a methodology for tag creation. If a document is not well-formed the XML processor will stop, complaining about a "fatal error".

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

- must begin with the XML declaration
- must have one unique root element
- all start tags must match end-tags
- XML tags are case sensitive
- all elements must be closed
- all elements must be properly nested
- all attribute values must be quoted
- XML entities must be used for special characters
XML declaration is a processing instruction that notifies the processing agent that the following document has been marked up as an XML document. It will look something like the following:

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

Once you have written your XML declaration, you are ready to begin coding your XML document. To do so, you should understand the concept of elements.

```xml
<?xml version = "1.0"?>
<CUSTOMER>
    <ID>001</ID>
    <NAME>Nick Scherbakov</NAME>
    <COMPANY>Interactive Internet (I2)</COMPANY>
    <CITY>Graz</CITY>
    <PHONE>662-9999</PHONE>
    <EMAIL>nick@i2.ac.at</EMAIL>
</CUSTOMER>
```

Elements are the basic unit of XML content. Syntactically, an element consists of a start tag, and an end tag, and everything in between. For example consider the following elements:

```xml
<NAME>Nick Scherbakov</NAME>
<COMPANY>Interactive Internet (I2)</COMPANY>
```

XML defines the text between the start and end tags to be "character data" and the text within the tags to be "markup".

A tag is pretty much anything between a `<` sign and a `>` sign. Care must be taken to assure that you maintain case within a tag set. In other words, the tags `<COMPANY>`, `<company>` would not be equivalent as they would in HTML.

Further, besides being spelled and capitalized the same way as their start tag counterparts, end tags should include an initial forward slash "/". Thus in most cases, a start tag of `<COMPANY>`, should be closed with a `</COMPANY>`.

if you need to use a tag that has no content, you may use a single start tag with a trailing forward slash such as:

```xml
<SALARY ... />`
```

The "<SALARY ... />" case is called an "Empty Element", empty because it has no content. Empty Elements often will have attributes that give them greater usefulness.

```xml
<NAME>Nick Scherbakov</NAME>
<COMPANY>Interactive Internet (I2)</COMPANY>
<SALARY val="3000"/>
```

Also, note that XML elements may contain other elements but the nesting of elements must be correct.

Thus the following example is wrong:

```xml
<CUSTOMER>
    <NAME>Frank Lee
    <EMAIL>flee@flee.com
</CUSTOMER></NAME></EMAIL>
```

Instead, it should be:
All XML documents must have at least one root element to be well formed.

The root element, also often called the document tag, must follow the prolog (XML declaration plus DTD) and must be a nonempty tag that encompasses the entire document. You are supposed to match the root element name to the DTD declaration.

For example, this declaration:

```xml
<myFirm>
  <CUSTOMER>
    <ID>001</ID>
    <NAME>Nick Scherbakov</NAME>
    <COMPANY>Interactive Internet (I2)</COMPANY>
    <CITY>Graz</CITY>
    <PHONE>662-9999</PHONE>
    <EMAIL>nick@i2.ac.at</EMAIL>
  </CUSTOMER>
</myFirm>
```

implies that "myFirm" is my root element.

Finally, tags may specify any number of supporting attributes. Attributes, that must not duplicate in any one tag, specify a name/value pair, delimited by equal (=) sign in which the value is delimited by quotation marks such as:

```xml
<CUSTOMER style = "spectator" coloring = "black_and_white">
```

Unlike HTML, XML specifies that values MUST be delimited with quotation marks.

```xml
<CUSTOMER style = "spectator" coloring = "black_and_white">
  <NAME>Frank Lee</NAME>
  <EMAIL VALUE="flee@flee.com"/>
  <SALARY val="3000"/>
</CUSTOMER>
```

As we have already said, it is a pretty good rule of thumb to consider anything outside of tags to be character data, and anything inside of tags to be considered markup. But alas, in one case this is not true. In the special case of CDATA blocks, all tags and entity references are ignored by an XML processor that treats them just like any old character data.

```xml
<EXAMPLE>
  <![CDATA[
    <DOCUMENT>
      <NAME>Coleen Merriman</NAME>
      <EMAIL>cm@mydomain.com</EMAIL>
    </DOCUMENT>
  ]]>
</EXAMPLE>
```

As you might have guessed, the character string ]]> is not allowed within a CDATA block as it would signal the end of the CDATA block.

**Document Type Definition (DTD)**
In the last section, we reviewed the process of creating a "well-formed" XML document. Making your document well-formed is only half the battle. You must also make sure that the document is **valid**.

To pass the validity test, an XML document must conform to the specifications defined by a **Document Type Definition (DTD)**. You can think of the DTD as defining the overall structure and syntax of the document.

In short, the DTD specifies everything a parser needs to know in order for that parser to interpret a well-formed XML document.

![Diagram of XML Process]

**Element Type Definition (EDT)**

The simplest usage of a DTD involves actually adding the DTD into the prolog portion of your XML document, just after the XML processing instruction.

```xml
<?xml version = "1.0"?>
<!DOCTYPE MYFIRM [
... ELEMENT DEFINITIONS
]>  
<MYFIRM>  
<CUSTOMER>  
<ID>001</ID>  
<NAME>Nick Scherbakov</NAME>  
<COMPANY>Interactive Internet (I2)</COMPANY>  
<CITY>Graz</CITY>  
<PHONE>662-9999</PHONE>  
<EMAIL>nick@i2.ac.at</EMAIL>  
</CUSTOMER>  
</MYFIRM>
```

Document Type Definitions declare all of the valid document elements using **Element Type Declarations (ETDs)**.
ETDs specify the name of elements and whether or not those elements may have any children. Elements may have several types of children ranging from none, to plain parsed character data, to other elements, to other elements with their own children, to any of the above.

The keyword (#PCDATA) allows an element (NAME) to contain parsed character data.

```xml
<?xml version = "1.0"?>
<!DOCTYPE MYFIRM [
... ]
```
ETDs may specify any number of children elements by references to their names. For example, the NAME element may be declared as a child of CUSTOMER element.

Similarly, ETDs may specify an order of child elements. For example, the NAME, PHONE and EMAIL elements may be declared as children of CUSTOMER element which may appear in arbitrary order.
Notice that since we simply listed the children of CUSTOMER as a space delimited list, we do not specify any order in which the children should appear.

Had we used a **comma to separate the list**, we would be forcing an order. Thus if we redefined our DTD to use:

```xml
<?xml version = "1.0"?>
<!DOCTYPE MYFIRM [ 
... 
<!ELEMENT CUSTOMER (NAME,PHONE,EMAIL)> 
<!ELEMENT NAME (#PCDATA)> 
<!ELEMENT EMAIL (#PCDATA)> 
<!ELEMENT PHONE (#PCDATA)> 
]>
... 
<CUSTOMER> 
<NAME>Nick Scherbakov</NAME> 
<PHONE>582898</PHONE> 
<EMAIL>nsherbak@iicm.edu</EMAIL> 
</CUSTOMER> 

<CUSTOMER> 
<NAME>Denis Helic</NAME> 
<PHONE>10215027</PHONE> 
<EMAIL>dhelic@iicm.edu</EMAIL> 
</CUSTOMER> 
... 

The plus sign (+) after an element name means "one or more occurrence" of this element. Thus we can redefine our DTD to allow one or more EMAIL elements inside any CUSTOMER element, and one or more CUSTOMER elements inside our XML document (root tag MYFIRM).
The aterisk sign (*) after an element name means "zero or more occurrence" of this element. Thus we can redefine our DTD to make PHONE elements optional inside any CUSTOMER element.

<?xml version = "1.0"?><!DOCTYPE MYFIRM [
<!ELEMENT MYFIRM ( CUSTOMER+)>
<!ELEMENT CUSTOMER (NAME,PHONE*,EMAIL+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT PHONE (#PCDATA)> ]>
<MYFIRM>
  <CUSTOMER>
    <NAME>Nick Scherbakov</NAME>
    <EMAIL>nsherbak@iicm.edu</EMAIL>
    <EMAIL>n_scherbakov@hotmail.com</EMAIL>
  </CUSTOMER>
  <CUSTOMER>
    <NAME>Denis Helic</NAME>
    <PHONE>10215027</PHONE>
    <PHONE>8735617</PHONE>
    <EMAIL>dhelic@iicm.edu</EMAIL>
    <EMAIL>dhelic@iicm.tu-grau.ac.at</EMAIL>
    <EMAIL>denis_helic@hotmail.com</EMAIL>
  </CUSTOMER>
</MYFIRM>

Elements can be grouped together using brackets, parameters "one or more occurrence" and "zero or more occurrence" can be applied to groups. Thus we can redefine our DTD to group PHONE and EMAIL and allows them (PHONE and EMAIL) appear together one or more times.

<?xml version = "1.0"?><!DOCTYPE MYFIRM [
<!ELEMENT MYFIRM ( CUSTOMER+)>
<!ELEMENT CUSTOMER (NAME,(PHONE EMAIL)+)>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)> ]>
The pipe character (|) is used to specify an "OR" operation.

Thus, the following DTD specify an XML document in which all CUSTOMER elements would have a NAME child followed by either a PHONE or an EMAIL element (but not both).

<?xml version = "1.0"?>
<!DOCTYPE MYFIRM [ 
<!ELEMENT MYFIRM (CUSTOMER+)>
<!ELEMENT CUSTOMER (NAME,(PHONE | EMAIL))>
<!ELEMENT NAME (#PCDATA)>
<!ELEMENT EMAIL (#PCDATA)>
<!ELEMENT PHONE (#PCDATA)> ]>

<MYFIRM>
  <CUSTOMER>
    <NAME>Nick Scherbakov</NAME>
    <PHONE>582898</PHONE>
  </CUSTOMER>
  
  <CUSTOMER>
    <NAME>Denis Helic</NAME>
    <EMAIL>dhelic@iicm.edu</EMAIL>
    <PHONE>8735617</PHONE>
    <EMAIL>dhelic@iicm.edu</EMAIL>
    <PHONE>2731645</PHONE>
    <EMAIL>dhelic@hyperwave.com</EMAIL>
  </CUSTOMER>
</MYFIRM>

Using the "?" character specifies that the element named is optional.

Thus, in the following code, we specify that every CUSTOMER must have a NAME and either a PHONE or EMAIL and may have an optional CITY child.
Finally, we must mention the syntax for defining an empty tag. Of course, there is not much to it, you simply use the EMPTY keyword such as:

```xml
<!ELEMENT DELIMITER EMPTY>
```
Element Attributes (ATTLIST)

Well, as you might expect, just as you use the DTD to define valid elements, you also use the DTD to define valid element attributes.

We already went over attributes in the last section, but to refresh your memory, we used the following example:

```xml
<CUSTOMER style = "spectator" coloring = "black_and_white">
    <NAME>Nick Scherbakov</NAME>
    <PHONE>582898</PHONE>
    <EMAIL>n_scherbakov@hotmail.com</EMAIL>
</CUSTOMER>
```

To declare attributes in the DTD you use the general format of:

```xml
<!ATTLIST [element] [attribute] [type] [default value]>
```

- `[element]` is a reference to an element
- `[attribute]` is equal to the name of the attribute such as "STYLE" or "COLORING" in the example above.
- `[default value]` specifies the value that is used if none is specified by the document author.
- `[type]` specifies one of ten valid attribute types.

NOTE: Since ATTLIST is a list it can have repeated attribute parts (and often does).

```xml
<?xml version = "1.0"?>
<!DOCTYPE CUSTOMER [
  <!ELEMENT CUSTOMER (NAME,((PHONE EMAIL),DELIMITER)*)>    
  <!ELEMENT NAME (#PCDATA)>  
  <!ELEMENT EMAIL (#PCDATA)>  
  <!ELEMENT PHONE (#PCDATA)>  
  <!ELEMENT DELIMITER EMPTY> 
  <!ATTLIST CUSTOMER
    STYLE CDATA #REQUIRED
    COLORING CDATA #REQUIRED
  >]

<CUSTOMER STYLE = "ordinary" COLORING = "windows">
    <NAME>Nick Scherbakov</NAME>
    <PHONE>582898</PHONE>
    <EMAIL>n_scherbakov@hotmail.com</EMAIL>
    <DELIMITER/>
    <EMAIL>nsherbak@iicm.edu</EMAIL>
    <PHONE>8735618</PHONE>
    <DELIMITER/>
</CUSTOMER>
```

Default value is defined by one of the following key-words:

- **REQUIRED** - there is no default value provided by the DTD, the attribute when actually implemented in an XML document must define a value.
- **IMPLIED** - a default value is specified by the DTD. If the document author does not override this default, the default will be used.
- **FIXED** - a default value is specified by the DTD. The document author may not modify this value.

```xml
<?xml version = "1.0"?>
<!DOCTYPE CUSTOMER [ 
<!ELEMENT CUSTOMER (NAME,(PHONE EMAIL)*,SALARY)>  
<!ELEMENT NAME (#PCDATA)>  
<!ELEMENT EMAIL (#PCDATA)>  
<!ELEMENT PHONE (#PCDATA)>  
<!ELEMENT SALARY EMPTY>
<!ATTLIST CUSTOMER
  STYLE CDATA #IMPLIED
  COLORING CDATA #REQUIRED
]>  
<!ATTLIST SALARY
  EURO CDATA #REQUIRED
]>  
<CUSTOMER COLORING = "windows">
<NAME>Nick Scherbakov</NAME>
<PHONE>582898</PHONE>
<EMAIL>n_scherbakov@hotmail.com</EMAIL>
<EMAIL>nsherbak@iicm.edu</EMAIL>
<PHONE>8735618</PHONE>
<SALARY EURO="3000"/>
</CUSTOMER>
```

there are 10 TYPES of content for attributes including:

- CDATA
- Enumerated
- ID
- IDREF
- IDREFS
- ENTITY
- ENTITIES
- NMTOKEN
- NMTOKENS
- NOTATION

**CDATA** refers to plain old character data that may be any string of characters that does not include ampersands (&), less than signs, (<), or quotation marks (").
**COLORING CDATA #REQUIRED**

```xml
<!ATTLIST SALARY EURO CDATA #REQUIRED >
```

```xml
MYFIRM

CUSTOMER COLORING = "windows">

NAME>Nick Scherbakov</NAME>

PHONE>582898</PHONE>

EMAIL>n_scherbakov@hotmail.com</EMAIL>

EMAIL>nsherbak@iicm.edu</EMAIL>

PHONE>8735618</PHONE>

SALARY EURO="3000"/

CUSTOMER> </MYFIRM>

ID represents a **unique ID name** for the attribute that identifies the element within the context of the document.

For the most part, ID is used primarily by programs or scripting languages that process the document. The value for ID must be a valid XML name beginning with a letter and containing alphanumeric characters or the underscore character without any whitespace.

NOTE: ID is incompatible with the #FIXED default but usually appears in conjunction with the #REQUIRED default.

```xml
<?xml version = "1.0"?>
<bracket xmlns="1.0">MYFIRM [

CUSTOMER CUSTOMER_ID= "1">

NAME>Nick Scherbakov</NAME>

PHONE>582898</PHONE>

EMAIL>n_scherbakov@hotmail.com</EMAIL>

CUSTOMER> <CUSTOMER CUSTOMER_ID= "33">

NAME>Denis Helic</NAME>

PHONE>8735612</PHONE>

EMAIL>dhelic@iicm.edu</EMAIL>

CUSTOMER>

MYFIRM>
```

The **IDREF** type allows the value of one attribute to be an element elsewhere in the document provided that the value of the IDREF is the ID value of the referenced element. **IDREFS** type allows such attribute to be coded as a list of references.
The **NM TOKEN** and **NM TOKENS** types are another example of those types that are useful primarily to processing applications. The types are used to specify a valid name(s). You might use them when you are associating some other component with the element, such as a JavaScript function.
XML Schema

The purpose of an XML Schema is to define the legal building blocks of an XML document, just like a DTD.

An XML Schema:

- defines elements that can appear in a document
- defines attributes that can appear in a document
- defines nesting rules
- defines data types for elements and attributes
- defines default and fixed values for elements and attributes

XML Schemas are the Successors of DTDs

Advantages:

- XML Schemas are extensible to future additions
- XML Schemas are richer and more useful than DTDs
- XML Schemas are written in XML
- XML Schemas support data types
- XML Schemas support namespaces, and can be nested

Name Spaces

A namespace is a vocabulary of a particular application: it identifies a set of predefined XML tags.

XML Schema uses a standard "XMLschema" name space, and defines a new name space.

http://www.w3.org/2001/XMLSchema

Source Namespace

Target Namespace
One difference between XML Schemas and DTDs is that the XML Schema vocabulary is associated with a name (namespace). Likewise, the new vocabulary that you define must be associated with a name (namespace).

With DTDs neither set of vocabulary is associated with a name (namespace) [because DTDs pre-dated namespaces].

For example, the following notation:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://coronet.iicm.edu">
    <xsd:element name="PName" type="xsd:string"/>
</xsd:schema>
```

defines

- "http://www.w3.org/2001/XMLSchema" as a source namespace
- "http://coronet.iicm.edu" as a target namespace
- prefix "xsd:" as a precedence for any valid element defined in the source namespace
- new element "PName" to be part of the target namespace

## Simple Element Types

A simple element is an XML element that can contain only string of characters. It cannot contain any other elements or attributes.

The content can be of many different types. It can be one of the types that are included in the XML Schema definition (boolean, string, date, etc.), or it can be a custom type that you can define yourself.

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://coronet.iicm.edu">
    <xsd:element name="PName" type="xsd:string"/>
    <xsd:element name="PPrice" type="xsd:integer"/>
</xsd:schema>
```

A valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="myFirm.xsd">
    . . .
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
    . . .
</myFirm>
```

Note, the notation

```xml
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="myFirm.xsd">
```

basically means that
• the XML file is based on the syntax defined by an instance of XML schema "http://www.w3.org/2001/XMLSchema"
• this instance can be located as a file "myFirm.xsd"
• "noNamespace" tells to the system that the tags defined in "myFirm.xsd" (for example <PName>), will not be prefixed with a name space prefix "xsi:".

Simple elements can have a default value OR a fixed value set.
A default value is automatically assigned to the element when no other value is specified.

```xml
<?xml version="1.0"?>
    <xsd:element name="PName" type="xsd:string" default="No name yet"/>
    <xsd:element name="PPrice" type="xsd:integer" default="100"/>
</xsd:schema>
```

the two XML fragments below are identical:
```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="myFirm.xsd">
    ... 
    <PName>No name yet</PName>
    <PPrice>100</PPrice>
    ... 
</myFirm>
```

Alternatively an element may be declared by means of nested XML Tags.

```xml
<?xml version="1.0"?>
    <xsd:element name="PName">
        <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                ... 
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
    <xsd:element name="PPrice">
        <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
                ... 
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

A valid XML fragment may look as follows:
```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="myFirm.xsd">
    ... 
    <PName>Graphic Card</PName>
    <PPrice>99</PPrice>
</myFirm>
```
Restrictions are used to control acceptable values for XML elements or attributes. Restrictions on XML elements are called facets.

```xml
<?xml version="1.0"?>
xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.licm.edu">
  <xsd:element name="PName">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\[A-Z]\([a-z][A-Z][0-9]\)+"/>
        <xsd:minLength value="5"/>
        <xsd:maxLength value="48"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="PPrice">
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="100"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

a valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <PName>Graphic Card</PName>
  <PPrice>99</PPrice>
  . . .
</myFirm>
```

Facets can be of many different types, They can define a syntax of the element content.

```xml
<?xml version="1.0"?>
xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.licm.edu">
  <xsd:element name="PName">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\[A-Z]\([a-z][A-Z][0-9]\)+"/>
        <xsd:minLength value="5"/>
        <xsd:maxLength value="48"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <xsd:element name="PPrice">
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="0"/>
        <xsd:maxInclusive value="100"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```
a valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <PName>Syrius1829</PName>
  <PPrice>99</PPrice>
  . . .
</myFirm>
```

**Complex Element Types**

A **complex element** is an XML element that can contain other elements and/or may have attributes. A complex element is declared by nested definition of all its components.

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="Product">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="PName" type="xsd:string"/>
        <xsd:element name="PPrice" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Notice that the child elements, "PName" and "PPrice", are surrounded by the "sequence" indicator. This means that the child elements must appear in the same order as they are declared; "PName" first and "PPrice" second.

Thus, a valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
  </Product>
  <Product>
    <PName>LAN Card</PName>
    <PPrice>178</PPrice>
  </Product>
  . . .
</myFirm>
```

The "all" indicator specifies by default that the child elements can appear in any order and that each child element must occur once and only once:

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="Product">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="PName" type="xsd:string"/>
        <xsd:element name="PPrice" type="xsd:integer"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```
<xsd:element name="Product">
  <xsd:complexType>
    <xsd:all>
      <xsd:element name="PName" type="xsd:string"/>
      <xsd:element name="PPrice" type="xsd:integer"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>

Thus, a valid XML fragment may look as follows:
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
  </Product>
  <Product>
    <PPrice>12</PPrice>
    <PName>LAN Card</PName>
  </Product>
  . . .
</myFirm>

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

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="Product">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="PName" type="xsd:string"/>
        <xsd:element name="PPrice" type="xsd:integer"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

Thus, a valid XML fragment may look as follows:
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
  </Product>
  <Product>
    <PPrice>12</PPrice>
  </Product>
  . . .
</myFirm>

The "sequence", "all" and "choice" indicators can be nested to define more complex orders of child elements.
Thus, a valid XML fragment may look as follows:

<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
    <Price_Euro>87</Price_Euro>
  </Product>
  <Product>
    <PName>LAN Card</PName>
    <Price_Dollar>18</Price_Dollar>
  </Product>
  . . .
</myFirm>

The **maxOccurs/minOccurs** indicators specify the maximum/minimum number of times an element can occur:

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="myFirm">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Product" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="PName" type="xsd:string"/>
              <xsd:element name="PPrice" type="xsd:integer" minOccurs="0" maxOccurs="5"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

Thus, a valid XML document (!) may look as follows:
Elements may have attributes. If an element has attributes, it is considered to be of complex
type. The attribute itself is declared with a nested tag "<xsd:attribute name="xxx" type="yyy"/>".

Thus, a valid XML document may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="myFirm.xsd">
  <Product>
    <PName>Graphic Card</PName>
    <PPrice Euro="67"/>
  </Product>
  <Product>
    <PName>LAN Card</PName>
    <PPrice Euro="18"/>
    <PPrice Euro="19"/>
    <PPrice Euro="20"/>
  </Product>
</myFirm>
```

All attributes are optional by default. To explicitly specify that the attribute is required, use the
"use" attribute:
Thus, a valid XML document may look as follows:

```
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  <Product>
    <PName>Graphic Card</PName>
    <PPrice Euro="67"/>
  </Product>
  <Product>
    <PName>LAN Card</PName>
    <PPrice Euro="18"/>
    <PPrice Euro="19"/>
    <PPrice Euro="20"/>
  </Product>
</myFirm>
```

References

Types defined in XML schema may be named and elements can simply reference a named type:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="BestSeller" type="productinfo"/>
  <xsd:element name="Product" type="productinfo"/>
  <xsd:complexType name="productinfo">
    <xsd:sequence>
      <xsd:element name="PName" type="xsd:string"/>
      <xsd:element name="PPrice" minOccurs="1" maxOccurs="5"/>
      <xsd:complexType>
        <xsd:attribute name="Euro" type="xsd:integer" use="required"/>
      </xsd:complexType>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```
Notice that the elements, "Product" and "BestSeller" refer to one and the same element type "productinfo".

Thus, a valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
  </Product>
  <BestSeller>
    <PName>LAN Card</PName>
    <PPrice>178</PPrice>
  </BestSeller>
  . . .
</myFirm>
```

Complex type element can be based on an existing complex type and extended with new elements.

```xml
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://coronet.iicm.edu">
  <xsd:element name="BestSeller" type="bestinfo"/>
  <xsd:element name="Product" type="productinfo"/>
  <xsd:complexType name="productinfo">
    <xsd:sequence>
      <xsd:element name="PName" type="xsd:string"/>
      <xsd:element name="PPrice" type="xsd:integer"/>
    </xsd:sequence>
  </xsd:complexType>
  <xsd:complexType name="bestinfo">
    <xsd:complexContent>
      <xsd:extension base="productinfo">
        <xsd:attribute name="DeliveryTime" type="xsd:integer"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:schema>
```

Thus, a valid XML fragment may look as follows:

```xml
<?xml version="1.0"?>
<myFirm xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="myFirm.xsd">
  . . .
  <Product>
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
  </Product>
  <BestSeller DeliveryTime="3">
    <PName>LAN Card</PName>
    <PPrice>178</PPrice>
  </BestSeller>
```

XSL-eXtensible Stylesheet Language

Rocollet that with XML we can use any tags we want, and the meaning of these tags are not automatically understood by the browser: `<table>` could mean a HTML table or maybe a piece of furniture. Because of the nature of XML, there is no standard way to display an XML document.

In order to display XML documents, it is necessary to have a mechanism to describe how the document should be displayed. One of these mechanisms is Cascading Style Sheets (CSS), but **XSL (eXtensible Stylesheet Language)** is the preferred style sheet language of XML, and XSL is far more sophisticated than the CSS used by HTML.

**XSL (eXtensible Stylesheet Language)** is entirely based on XML and on concept of **namespaces**

Namespaces

**XSL** consists of two parts that are defined by different namespaces:

- a method for **transforming XML documents** (TAGs that can filter and sort XML data defined by namespace "http://www.w3.org/1999/XSL/Transform")
- a method for **formatting XML documents** (TAGs that can format XML data, based on the data value defined by namespace "http://www.w3.org/1999/XSL/Format")

Recollect that the notation:
 means that

- the transformation TAGs defined by "http://www.w3.org/1999/XSL/Transform" are prefixed with "xsl:"
- the formatting TAGs defined by "http://www.w3.org/1999/XSL/Format" are prefixed with "fo:"

**Transforming XML Documents**

XSL templates *style* elements of an XML document. Simply stated, the term "styling" may be seen as replacing XML TAGs with valid HTML fragments.

One critical capability of a stylesheet language is to locate source elements to be styled. XSL, for example, does this with "selectors."

Most templates have the following form:

```xml
<xsl:template match=[selector]>
  <HTML TAGs><xsl:apply-templates/></HTML TAGs>
</xsl:template>
```

```xml
<xsl:template match="CUSTOMER">
  <UL><xsl:apply-templates/></UL>
</xsl:template>
```

```xml
<xsl:template match="NAME">
  <LI>Name: <xsl:apply-templates/></LI>
</xsl:template>
```

```xml
<xsl:template match="PHONE">
  <LI>Phone: <xsl:apply-templates/></LI>
</xsl:template>
```

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

The XML document:

```xml
<CUSTOMER>
  <NAME>Nick Scherbakov</NAME>
  <PHONE>582898</PHONE>
  <EMAIL>nsherbak@iicm.edu</EMAIL>
</CUSTOMER>
```

```xml
<CUSTOMER>
  <NAME>Denis Helic</NAME>
  <PHONE>10215027</PHONE>
  <EMAIL>dhelic@iicm.edu</EMAIL>
</CUSTOMER>
```

will be converted into the following HTML:

```html
<UL>
  <LI>Name: Nick Scherbakov</LI>
  <LI>Phone: 582898</LI>
</UL>
```
Similarly to parsing HTML, when a XSL transformer parses an XML document, it builds a so-called XML Document Object Model (XML DOM).

The document is considered to be an hierarchy of elements. Each element belongs to a particular type and may consist of other elements (Children).

In context of an XML DOM, Selectors may have much more complex format. For example,

- "/" Matches the root of the document
- "CUSTOMER|PRODUCT" Matches <CUSTOMER> or <PRODUCT> elements

The value of the required "match" attribute may contain an XPath expression. It works like navigating a file system where a forward slash (/) selects subdirectories.

- "CUSTOMER/NAME" Matches all <NAMES> elements that have a parent of <CUSTOMER>
- "PRODUCT/NAME" Matches all <NAMES> elements that have a parent of <PRODUCT>
The XML document:

```xml
<Product>
  <Name>WBT-Master</Name>
</Product>

<Customer>
  <Name>Nick Scherbakov</Name>
  <Phone>582898</Phone>
  <Email>nsherbak@iicm.edu</Email>
</Customer>

<Customer>
  <Name>Denis Helic</Name>
  <Phone>10215027</Phone>
  <Email>dhelic@iicm.edu</Email>
</Customer>
```

will be converted into the following HTML:

```html
<UL>
  <LI>Product name: WBT-Master</LI>
</UL>

<UL>
  <LI>Customer name: Nick Scherbakov</LI>
</UL>

<UL>
  <LI>Customer name: Denis Helic</LI>
</UL>
```

Attribute values may be used to locate XML element.

For example,

- "CUSTOMER[@TYPE='Individual']" Match <CUSTOMER> elements that have "TYPE" attribute with the value "Individual".
The other model for applying style is to select each action procedurally. A series of templates is created, such that each template explicitly select elements of an XML document and retrieve their content and values of attributes.

For example, `<xsl:value-of select="NAME"/>` selects a first element `<NAME/>` ... `<NAME/>` and returns its content.

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html>
      <body>
        <UL>
          <LI><xsl:value-of select="NAME"/></LI>
        </UL>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The XML document:

```xml
<NAME>Nick Scherbakov</NAME>
<NAME>Denis Helic</NAME>
<PHONE>10215027</PHONE>
<EMAIL>dhelic@iicm.edu</EMAIL>
```

will be converted into the following HTML:

```html
<html>
<body>
<UL>
  <LI>Nick Scherbakov</LI>
</UL>
</body>
</html>
```
The **xsl:for-each** element locates elements in the XML document and repeats a template for each one.

All other select statements nested into the loop locates a child in the hierarchy.

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="CUSTOMER">
          <UL>
            <LI><xsl:value-of select="NAME" /></LI>
          </UL>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The XML document:

```xml
<CUSTOMER>
  <NAME>Nick Scherbakov</NAME>
  <PHONE>582898</PHONE>
  <EMAIL>nsherbak@iicm.edu</EMAIL>
</CUSTOMER>
<CUSTOMER>
  <NAME>Denis Helic</NAME>
  <PHONE>10215027</PHONE>
  <EMAIL>dhelic@iicm.edu</EMAIL>
</CUSTOMER>
```

will be converted into the following HTML:

```html
<html>
  <body>
    <UL>
      <LI>Nick Scherbakov</LI>
    </UL>
    <UL>
      <LI>Denis Helic</LI>
    </UL>
  </body>
</html>
```

One of important XSL constructions is known as a **filter**

Most often, a particular value of an attribute is used as a filter.

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <body>
      <xsl:for-each select="CUSTOMER[@TYPE='Individual']">
      </xsl:for-each>
    </body>
  </xsl:template>
</xsl:stylesheet>
```
The XML document:

```xml
<CUSTOMER TYPE="Corporative">
  <FIRM>MM International</FIRM>
  <PHONE>582898</PHONE>
  <EMAIL>nshebak@iicm.edu</EMAIL>
</CUSTOMER>

<CUSTOMER TYPE="Individual">
  <NAME>Denis Helic</NAME>
  <PHONE>10215027</PHONE>
  <EMAIL>deline@iicm.edu</EMAIL>
</CUSTOMER>
```

will be converted into the following HTML:

```html
<html>
<body>
<UL>
  <LI>Denis Helic</LI>
</UL>
</body>
</html>
```

A similar notation is used by so-called **conditional XSL patterns**:
The XML document:

```xml
<xsl:template>
</xsl:stylesheet>

The XML document:

```xml
cUSTOMER TYPE="Corporative">
<FIRM>MM Internationsl</FIRM>
<PHONE>582898</PHONE>
<EMAIL>nsherbak@iicm.edu</EMAIL>
</CUSTOMER>
cUSTOMER TYPE="Individual">
<NAME>Denis Helic</NAME>
<PHONE>10215027</PHONE>
<EMAIL>dhelic@iicm.edu</EMAIL>
</CUSTOMER>
```

will be converted into the following HTML:

```html
<html>
<body>
<H2>MM International</H2>
<UL>
  <LI>Denis Helic</LI>
</UL>
</body>
</html>
```

To insert an conditional choose test against the XML content, simply add an `xsl:choose`, `xsl:when` (multiple) and `xsl:otherwise` elements to your XSL document

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
<xsl:template match="/">
<html>
<body>
<xsl:for-each select="CUSTOMER">
<xsl:choose>
<xsl:when match=". [@TYPE='Individual']">
<UL>
  <LI><xsl:value-of select="NAME"/></LI>
</UL>
</xsl:when>
<xsl:when match=". [@TYPE='Corporative']">
<H2><xsl:value-of select="FIRM"/></H2>
</UL>
</xsl:when>
<xsl:otherwise>
<B>Invalid Customer Type!</B>
</xsl:otherwise>
</xsl:for-each>
</body>
</html>
</xsl:template>
</xsl:stylesheet>
```

The XML document:
To sort elements of an XML file we can use an **order-by** attribute of **for-each** template.
The order-by attributes takes a plus (+) or minus (-) sign, to define an ascending or descending sort order, and an element name to define the sort element.

```xml
<xsl:stylesheet xmlns:xsl="http://www.w3.org/TR/WD-xsl">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="CUSTOMER" order-by="+ @CUSTOMER_ID">
          <ul>
            <li><xsl:value-of select="NAME"/></li>
          </ul>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```

The XML document:

```xml
<CUSTOMER CUSTOMER_ID="33">
  <NAME>Nick Scherbakov</NAME>
  <PHONE>582898</PHONE>
  <EMAIL>nsherbak@iicm.edu</EMAIL>
</CUSTOMER>
<CUSTOMER CUSTOMER_ID="22">
  <NAME>Denis Helic</NAME>
  <PHONE>10215027</PHONE>
  <EMAIL>dhelic@iicm.edu</EMAIL>
</CUSTOMER>
```
Formatting HTML Documents (CSS)

Before we continue with XSL formatting facilities we should have some basic understanding of so-called CSS (Cascading Style Sheets).

- Styles refer to the HTML Document Object Model
- Each object is displayed within a rectangular area on the screen having certain properties
- Styles define how to display objects of a document (i.e. HTML elements)
• Styles are normally stored in the document or in a separated Style Sheets file.
• Multiple style definitions will cascade into one

HTML tags were originally designed to define the content of a document. They were supposed to say "This is a header", "This is a paragraph", "This is a table", by using tags like <h1>, <p>, <table>, and so on.
Presentation layout (graphical view) is defined by so-called default styles associated with the HTML tags.

For example, we can see that for an <H1> tags styles are set as:

```
Header
• Background:transparent;
• Border:0;
• Font-size:20px;
• Font-weight:bold;
• Margin:20px;
```

It became more and more difficult to create Web sites where the content of HTML documents was clearly separated from the document's presentation layout.
To solve this problem, the World Wide Web Consortium (W3C) - the non profit, standard setting consortium responsible for standardizing HTML - created STYLES in addition to HTML 4.0.

**Style Sheets Can Save a Lot of Work**
Styles in HTML 4.0 define how HTML elements are displayed, just like the font tag and the color attribute in HTML 3.2. Styles are normally saved in files external to your HTML documents. External style sheets enable you to change the appearance and layout of all the pages in your Web, just by editing a single CSS document. If you have ever tried to change the font or color of all the headings in all your Web pages, you will understand how CSS can save you a lot of work.

CSS is a breakthrough in Web design because it allows developers to control the style and layout of multiple Web pages all at once. As a Web developer you can define a style for each HTML element and apply it to as many Web pages as you want. To make a global change, simply change the style, and all elements in the Web are updated automatically.

**Styles are Inherited**
Properties set by an element are inherited by all its child-elements, and can be overriden.

```
Border
The same -
Border
  Customer 1
    Name: Nick
    Phone: 8735618
Font
The same Font
```

**Multiple Styles Will Cascade Into One**
Style Sheets allow style information to be specified in many ways. Styles can be specified inside a single HTML element, inside the <head> element of an HTML page, or in an external CSS file. Even multiple external Style Sheets can be referenced inside a single HTML document.

What style will be used when there is more than one style specified for an HTML element?

Generally speaking we can say that all the styles will "cascade" into a new "virtual" Style Sheet by the following rules, where number four has the highest priority:
• Browser default
• External Style Sheet
• Internal Style Sheet (inside the tag)
• Inline Style (inside HTML element)

So, an inline style (inside an HTML element) has the highest priority, which means that it will
override every style declared inside the <head> tag, in an external style sheet, and in a browser
(a default value).
The CSS syntax is made up of three parts: a selector, a property and a value:
   selector {property: value}
The selector is normally the element/tag you wish to define, the property is the attribute you wish
to change, and each property can take a value. The property and value are separated by a colon
and surrounded by curly braces:
   body {color: black}
If the value is multiple words, put quotes around the value:
   p {font-family: "sans serif"}
Note: If you wish to specify more than one property, you should separate each property with a
semi-colon. The example below shows how to define a center aligned paragraph, with a red text
color:
   p {text-align: center; color: red}
To make the style definitions more readable, you can describe one property on each line, like
this:

   p
   {
      text-align: center;
      color: black;
      font-family: arial
   }
You can group selectors. Separate each selector with a comma. In the example below we have
grouped all the header elements. Each header element will be green:
   h1, h2, h3, h4, h5, h6
   {
      color: green
   }
With the class attribute you can define different styles for the same element. Say that
you would like to have two types of paragraphs in your document: one right-aligned
paragraph, and one center-aligned paragraph. Here is how you can do it with styles:
   p.right {text-align: right}
   p.center {text-align: center}
You have to use the class attribute in your HTML document:

```html
<p class="right">
This paragraph <BR>will be <BR>right-aligned.
</p>
```
This paragraph will be right-aligned.

```html
<p class="center">
This paragraph <BR>will be <BR>center-aligned.
</p>
```
This paragraph will be center-aligned.

The id attribute can be defined in two ways. It can be defined to match all elements with
a particular id, or to match only one element with a particular id.
In this example the id attribute will match all elements with id="intro":

```html
#intro
{
   font-size:110%;
   font-weight:bold;
```
In this example the id attribute will match only p elements with id="intro":

```html
p#intro
{
  font-size:110%;
  font-weight:bold;
  color:#0000ff;
  background-color:transparent
}
```

You can also omit the tag name in the selector to define a style that can be used by many elements:

```css
.center {text-align: center}
```

In the code below both the h1 element and the p element are classed with "center". This means that both of the elements will follow the rules in the ".center" selector:

```html
&lth1 class="center">
Centered <BR>Header
</h1>

<p class="center">
Centered<BR>text
</p>
```

CSS Properties:

- font properties
- margin and spacing properties
- border and padding properties
- keeps/breaks
- horizontal alignment/justification
- indentation
- more formatting object specific properties

For example:

```css
.rText {background-color:white;
margin-left:1pt; margin-right:1pt;
border-top-width: 1pt; border-bottom-width:1pt; border-right-width:1pt;
border-left-width:1pt; border-style:outset; border-color:blue;
font-family:arial,san-serif;font-size:15px;
padding-top:5pt; padding-bottom:5pt; padding-right:5pt; padding-left:12pt;
}
```

### Formatting XML Documents

**XSL Formatting Objects (XSL-FO)** are the second half of the Extensible Stylesheet Language (XSL). XSL-FO is an XML application that describes how pages will look when presented to a reader. A style sheet uses the XSL transformation language to transform an XML document in a semantic vocabulary into a new XML document that uses the XSL-FO presentational vocabulary.

While one can hope that Web browsers will one day know how to directly display data marked up with XSL formatting objects, for now an additional step is necessary in which the output document is further transformed into some other format, such as Adobe’s PDF.
There are exactly 56 XSL formatting object elements. You can consult http://www.ibiblio.org/xml for a precise definition of all of them.

In this chapter, we discuss just one formatting object: "fo:block" just to illustrate the XSL Formatting paradigm.

The XSL formatting model is based on rectangular boxes called areas that can contain text, empty space, images, or other formatting objects. The "fo:block" identifies a rectangular area and impose a certain formatting properties onto all objects inside the block.

```xml
<xsl:template match="CUSTOMER">
<fo:block font-size="12pt" space-before="20px"
    margin-left:1pt; margin-right:1pt;
    border-top-width: 1pt; border-bottom-width:1pt; border-right-width:1pt;
    border-left-width:1pt; border-style:outset; border-color:blue;
    text-align="justified">
Customer:
<UL><xsl:apply-templates/></UL>
</fo:block>
</xsl:template>

<xsl:template match="NAME">
<fo:block font-size="12pt">
<LI><xsl:value-of select="NAME"/></LI>
</fo:block>
</xsl:template>
```

The XML document:

```
<CUSTOMER>
<NAME>Nick Scherbakov</NAME>
<PHONE>582898</PHONE>
<EMAIL>nsherbak@iicm.edu</EMAIL>
</CUSTOMER>

<CUSTOMER>
<NAME>Denis Helic</NAME>
<PHONE>10215027</PHONE>
<EMAIL>dhelic@iicm.edu</EMAIL>
</CUSTOMER>
```

will be converted into the following HTML:

Customer:

- Nick Scherbakov

Customer:

- Denis Helic
7 Linking XML Resources

Why do we need a new linking model?

HTML links are just too simple:

- An anchor must be placed at every link destination (problem with read-only documents)
  - we want to express relative locations (XPointer!).
- The link definition must be at the same location as the link source
  - we want out-of-line links ("link databases").
- Only individual nodes can be linked to
  - we want links to whole tree fragments.
- A link always has one source and one destination
  - we want links with multiple sources and destinations.
- Links are anonymous
  - we want human-readable labels.

7.1 XML Linking Model

XML Linking Model:

- Link: a set of traversable arcs between two or more resources.
- Locator: An identification of a remote resource that is participating in the link.

XML Linking Model:

- Linking element: An XML element that asserts the existence and describes the characteristics of a link.
- one linking element defines a set of traversable arcs between some resources.

```
<myLink>
  <locator href="#Fred" role="student"/>
  <locator href="#Hermann" role="student"/>
  <locator href="#Nick" role="teacher"/>
  <arc from="student" to="teacher" show="embed"/>
</myLink>
```
7.2 XML Link Namespaces

XML linking is carried out by means of three different languages (namespaces):

- **XLink** (`xmlns:xlink="http://www.w3.org/1999/xlink"`)
  - A generalization of the HTML link concept
  - Higher abstraction level (intended for general XML - not just hypertext)
  - More expressive power (multiple destinations, special behaviours, out-of-line links, ...)
  - Uses XPointer to locate resources

- **XPointer**
  - An extension of XPath suited for linking
  - Specifies connection between XPath expressions and URIs

- **XPath**
  - A declarative language for locating nodes and fragments in XML trees
  - Used in both XPointer (for addressing) and XSL (for pattern matching)

For example, the following notation:

```xml
<myLink xmlns:xlink="http://www.w3.org/1999/xlink">
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:locator href="#Nick" role="teacher"/>
  <xlink:arc from="student" to="teacher" show="embed"/>
</myLink>
```

defines

- "http://www.w3.org/1999/xlink" as a source namespace
- Prefix "xlink:" as a precedence for any valid element defined in the source namespace
- New element "myLink" to be part of the target namespace

7.3 xLink Notation

- A general linking element is defined using an `xlink:type=extended/simple` element (which can contain the following)
- A remote resource is defined using a `locator` element with a `href` attribute (an XPointer expression locating the resource).
- Arcs (traversal rules) are defined using `arc` elements:
  - A `locator` has a `role` attribute.
  - An `arc` element has a `from` and a `to` attribute.
  - The `arc` element defines a set of arcs: from each resource having the `from` role to each resource having the `to` role.
  - If `from` or `to` is omitted, any role matches.

```xml
<myLink xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:locator href="#Nick" role="teacher"/>
  <xlink:arc from="student" to="teacher" show="embed"/>
</myLink>
```

Arcs can be annotated with abstract behaviour information using the `show` and `actuate` arc attributes:

- **actuate** controls activation of links
  - Possible values:
    - `onLoad` - traverse the link immediately when recognized similar to image links.
    - `onRequest` - traverse when explicitly requested similar to normal HTML links.
    - `undefined` - behaviour specified elsewhere.
- **show** controls behaviour when the link is activated
  - Possible values:
    - `embed` - insert the target resource (the one at the end of the arc) immediately after the display of the source resource (the one at the beginning of the arc, where traversal was initiated). *Similar to images in HTML.*
    - `new` - display the target resource some other place without affecting presentation of the resource from which traversal was initiated. *Similar to a pop-up window in an HTML document.*
    - `replace` - replace the presentation of the resource containing the linking element with a presentation of the one being linked to. *Similar to normal HTML links.*
For example, the following link element:

```xml
<myLink xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:type="extended">  
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:locator href="#Nick" role="teacher"/>
  <xlink:arc from="student" to="teacher" actuate="onLoad"
            show="embed"/>
</myLink>
```

defines a system behaviour as follows:

- whenever a resource "student" ("Fred" or "Hermann") is accessed, link to the resource teacher ("Nick") is traversed automatically
- the resource "teacher" is displayed immediately after the resource "student" as an embedded window.

Processing XML links as it was presented above, may be seen as a special mechanism for imposing additional properties onto other XML resources (fragments).

For example, the following link element:

```xml
<myLink xmlns:xlink="http://www.w3.org/1999/xlink"
xlink:type="extended">  
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:locator href="#Nick" role="teacher"/>
  <xlink:arc from="student" to="teacher" actuate="onLoad"
            show="embed"/>
</myLink>
```

may be perceived as follows:

- find XML resources #Hermann and #Fred
- add references to the resource #Nick to the resources #Hermann and #Fred
- set behaviour attributes as "actuate='onLoad' show='embed'" for the references.
Note that a link element can define a resource itself.

```xml
<teacher xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:arc from="local" to="student" actuate="onLoad" show="embed"/>
  Nick Scherbakov
</teacher>
```

In this case the resource is called a **local recourse**, and has a fixed role "local".

Note that a link element can define a resource itself.

```xml
<teacher xmlns:xlink="http://www.w3.org/1999/xlink" xlink:type="extended">
  <xlink:locator href="#Fred" role="student"/>
  <xlink:locator href="#Hermann" role="student"/>
  <xlink:arc from="local" to="student" actuate="onLoad" show="embed"/>
  Nick Scherbakov
</teacher>
```
In this case the resource is called a **local recourse**, and has a fixed role "local".

For compatibility and simplicity, XML defines two kinds of links:

- **extended** - the general ones we have seen so far
- **simple** - a restricted version of extended links: only for two-ended inline links (enough for HTML-style links)

Convenient shorthand notation for simple links:

```
<xlink:simple href="..." show="..." actuate="..."> 
Local Resource
</xlink:simple>
```

which is equivalent to:

```
<xlink:extended>
<xlink:locator href="..." role="remote"/>
<xlink:arc from="local" to="remote" show="..." actuate="..."/>
Local Resource
</xlink:extended>
```

(show and actuate are optional.)

### 7.4 xPointer Notation

**XPointer**, the XML Pointer Language, defines an addressing scheme for individual parts of an XML document. These addresses can be used by any application that needs to identify parts of or locations in an XML document. URI in an XLINK can include an XPointer fragment identifier that locates one particular element in the targeted document. XPointers use the same XPath syntax that you're familiar with from XSL transformations to identify the parts of the document they point to, along with a few additional pieces. Recollect that **XPath expression** works like navigating a file system where a forward slash (/) selects subdirectories.

- "CUSTOMER/NAME" Matches all <NAMES> elements that have a parent of <CUSTOMER>
- "PRODUCT/NAME" Matches all <NAMES> elements that have a parent of <PRODUCT>

```
<xsl:template match="CUSTOMER|PRODUCT">
  <UL><xsl:apply-templates/></UL>
</xsl:template>
<xsl:template match="CUSTOMER/NAME">
  <LI>Customer name: <xsl:apply-templates/></LI>
</xsl:template>
<xsl:template match="PRODUCT/NAME">
  <LI>Product name: <xsl:apply-templates/></LI>
</xsl:template>
</xsl:stylesheet>
```

Traditional URLs are simple and easy to use, but they're also quite limited. For one thing, a URL only points at a single, complete document. More granularity than that, such as linking to the third sentence of the seventeenth paragraph in a document, requires the author of the targeted document to manually insert named anchors at the targeted location. The author of the document doing the linking can't do this unless he or she also has write access to the document being linked to.

It would be more useful to be able to link to a particular element or group of elements on a page without having to change the document you're linking to.

Thus, XPointers solve the problem of addressing the individual XML elements. XPointers enable you to target a given element by number, name, type, or relation, to other elements in the document.

XPointer is a special expression attached to a basic document URL:

```
http://coronet.iicm.edu/test.xml#xpointer(...)```

Many (though not all) XPointers are location paths. Location paths are built from location steps. Each location step specifies a point in the targeted document, always relative to some other well-known point such as the start of the document or the previous location step. This well-known point is called the context node.
Consider also the following xPointer:

```
xpointer(/child::MYFIRM/child::CUSTOMER[position()=1])
```

This XPointer is built from two location steps:

- `/child::MYFIRM`
- `/child::CUSTOMER[position()=1]`

The first location step is an absolute step that selects all child elements of the root node whose name is MYFIRM. There's exactly one such element. The second location step is then applied relative to the MYFIRM element returned by the first location step. All of its child nodes are considered. Those that satisfy the node test — that is, elements whose name is CUSTOMER — are returned. There are 2 of these nodes. Each of these 2 nodes is then compared against the predicate to see if its position is equal to 1. This turns out to be true for only one node, Nick Scherbakov's CUSTOMER element, so that is the single node this XPointer points to.

In general, a location step has three parts: the axis, the node test, and an optional predicate. These are combined in this form:

```
axis::node-test[predicate]
```

For example, in the location step

```
child::CUSTOMER[position()=2],
```
the axis is child,
the node-test is CUSTOMER, and
the predicate is [position()=2].
This location step selects the second CUSTOMER element along the child axis, starting from the context node or, less formally, the second CUSTOMER child element of the context node. Of course, which element this actually is depends on what the context node is.

Generally speaking:

- The **axis** tells you in what direction to search from the context node. For instance, an axis can say to look at things that follow the context node, things that precede the context node, things that are children of the context node, things that are attributes of the context node, and so forth.
- The **node test** tells you which nodes to consider along the axis. The most common node test is simply an **element name**. However the node test may also be the asterisk (*) wild card to indicate that any element is to be matched, or one of several functions for selecting comments, text, attributes, processing instructions, points, and ranges.
- The group of nodes along the given axis that satisfy the node test form a **location set**.
- The predicate is a boolean expression that tests each node in that set. If that expression returns false, then the node is removed from the set.

```
<?xml version = "1.0"?>
<MYFIRM>
  <CUSTOMER>
    <NAME>Nick Scherbakov</NAME>
    <PHONE>582898</PHONE>
    <EMAIL>n_scherbakov@hotmail.com</EMAIL>
    <EMAIL>nsherbak@iicm.edu</EMAIL>
    <PHONE>8735618</PHONE>
  </CUSTOMER>
  <PRODUCT>
    <PName>Graphic Card</PName>
    <PPrice>98</PPrice>
  </PRODUCT>
  <PRODUCT>
    <PPrice>12</PPrice>
    <PName>LAN Card</PName>
  </PRODUCT>
</MYFIRM>
```

Often, after the entire location step — axis, node test, and predicate — has been evaluated, what's left is a **single, unique node**. A location set like this with only one node is called a **singleton**. However, not all location steps produce singletons. In some cases, you may finish with multiple nodes in the final location set.
XPointers are not limited to location paths. In fact they can use any expression that returns a node set. In particular, they can use functions that return node sets.

The id() function is one of the simplest and most robust means of identifying an element node. It selects the element in the document that has an ID type attribute with a specified value. For example: