eXtensible Markup Language (XML) / JSON

Chapter 19

Objectives

1. XML Overview
2. DTD: Document Type Definition
3. XSD: XML Schema Document
4. XPATH
5. JSON
6. Self-Reading: XML Processing
7. Self Reading: XML Style Transformations

Section 1 of 7

XML OVERVIEW

XML Overview

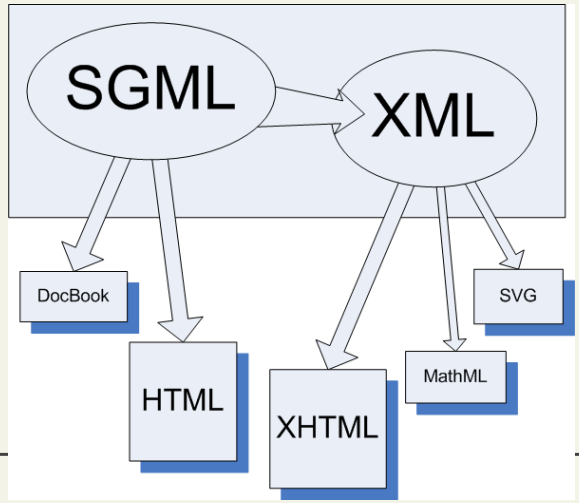
- Developed from SGML
- Became a W3C Recommendations in 1998
- A *meta-markup* language -> unlike HTML, XML can be used to mark up any type of data. Tags are not predefined (user generated)
- Deficiencies of HTML and SGML
  - Many complex features that are rarely used
- HTML is a markup language, XML is used to define markup languages
- Markup languages defined in XML are known as ***applications***
- XML can be written by hand or generated by computer
  - Useful for data exchange
- Foundation for several next-generation web technologies:
  - RSS, AJAX, Web Services, etc.

XML Overview

One of the key benefits of XML data is that it is plain text: it can be **read** and **transferred** between applications and different operating systems as well as being **human-readable and understandable** as well.

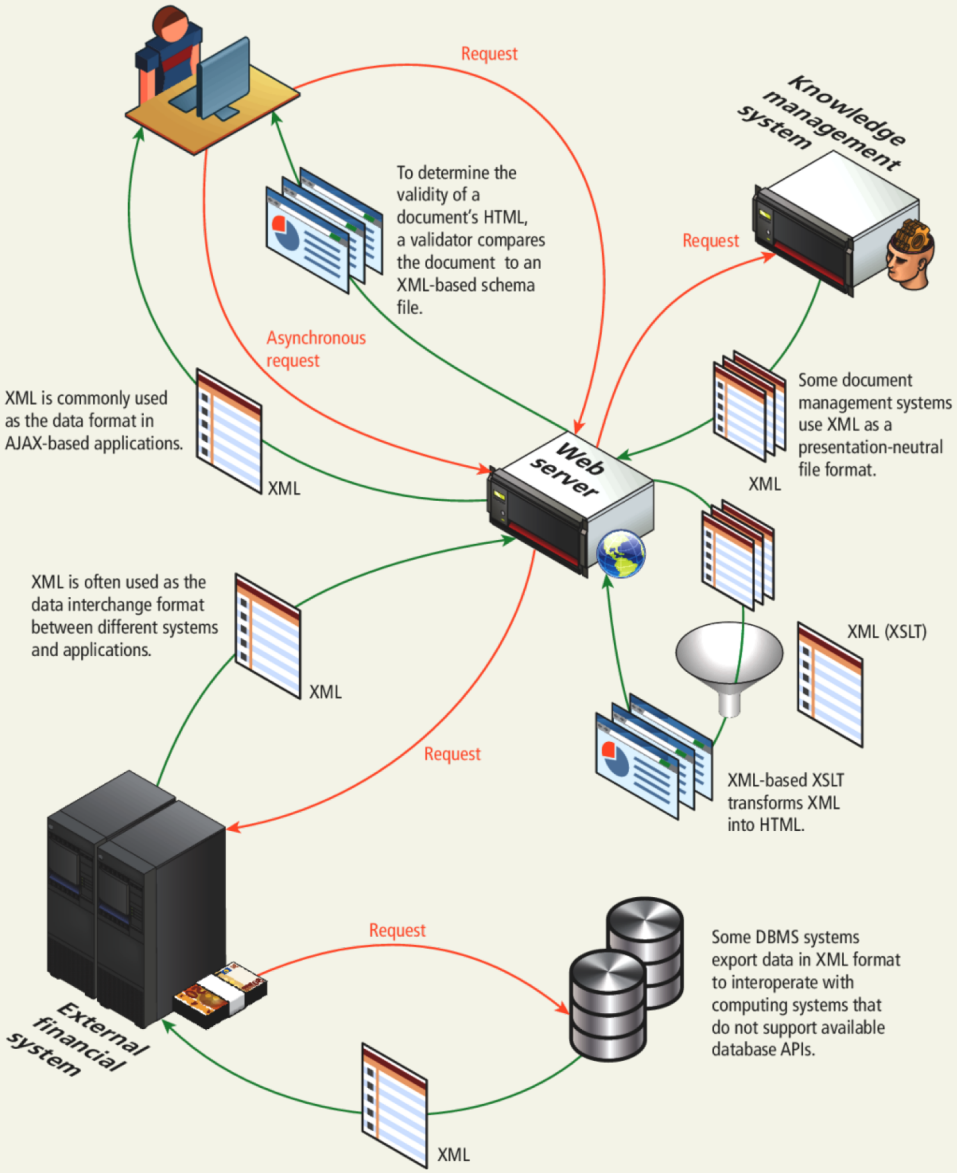
XML is used to **structure** and **describe** information.

XML is not only used on the web server and to communicate asynchronously with the browser, but is also used as a **data interchange** format for moving information between systems.



XML Overview

Used in many systems



XML Advantages and Disadvantages

Pros

- Human-readable, self-documenting format
- Strict syntax allows standardized tools
- International, platform-independent
- Can represent almost any general kind of data (record, list, tree)

Cons

- Bulky syntax/structure makes files large; can decrease performance

An Example XML Application

```
<?xml version="1.0" encoding="UTF-8"?>
<note>
   <to>Ana</to>
   <from>Dana</from>
   <subject>Reminder</subject>
   <message>
      Don't forget me this weekend!
   </message>
</note>
```

XML File Structure

- A header, then a single document tag that can contain other tags

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

- Tag syntax:

```
<element attributes> text or tags    </element>
```

- Attribute syntax:

```
name="value"
```

- comments:

```
<!-- comment -->
```

Well Formed XML

For a document to be **well-formed XML**, it must follow the syntax rules for XML:

- Element names are composed of any of the valid characters (most punctuation symbols and spaces are not allowed) in XML.
- Element names can't start with a number.
- There must be a single-root element. A **root element** is one that contains all the other elements; for instance, in an HTML document, the root element is <html>.
- All elements must have a closing element (or be self-closing).
- Elements must be properly nested.
- Elements can contain attributes.
- Attribute values must always be within quotes.
- Element and attribute names are case sensitive.

Well Formed XML

Sample Document

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<art>
  <painting id="290">
    <title>Balcony</title>
    <artist>
      <name>Manet</name>
      <nationality>France</nationality>
    </artist>
    <year>1868</year>
    <medium>Oil on canvas</medium>
  </painting>
  <painting id="192">
    <title>The Kiss</title>
    <artist>
      <name>Klimt</name>
      <nationality>Austria</nationality>
    </artist>
    <year>1907</year>
    <medium>Oil and gold on canvas</medium>
  </painting>
  <painting id="139">
    <title>The Oath of the Horatii</title>
    <artist>
      <name>David</name>
      <nationality>France</nationality>
    </artist>
    <year>1784</year>
    <medium>Oil on canvas</medium>
  </painting>
</art>
```

LISTING 17.1 Sample XML document

Valid XML

A **valid XML** document is one that is well formed and whose element and content conform to the rules of a certain **schema**

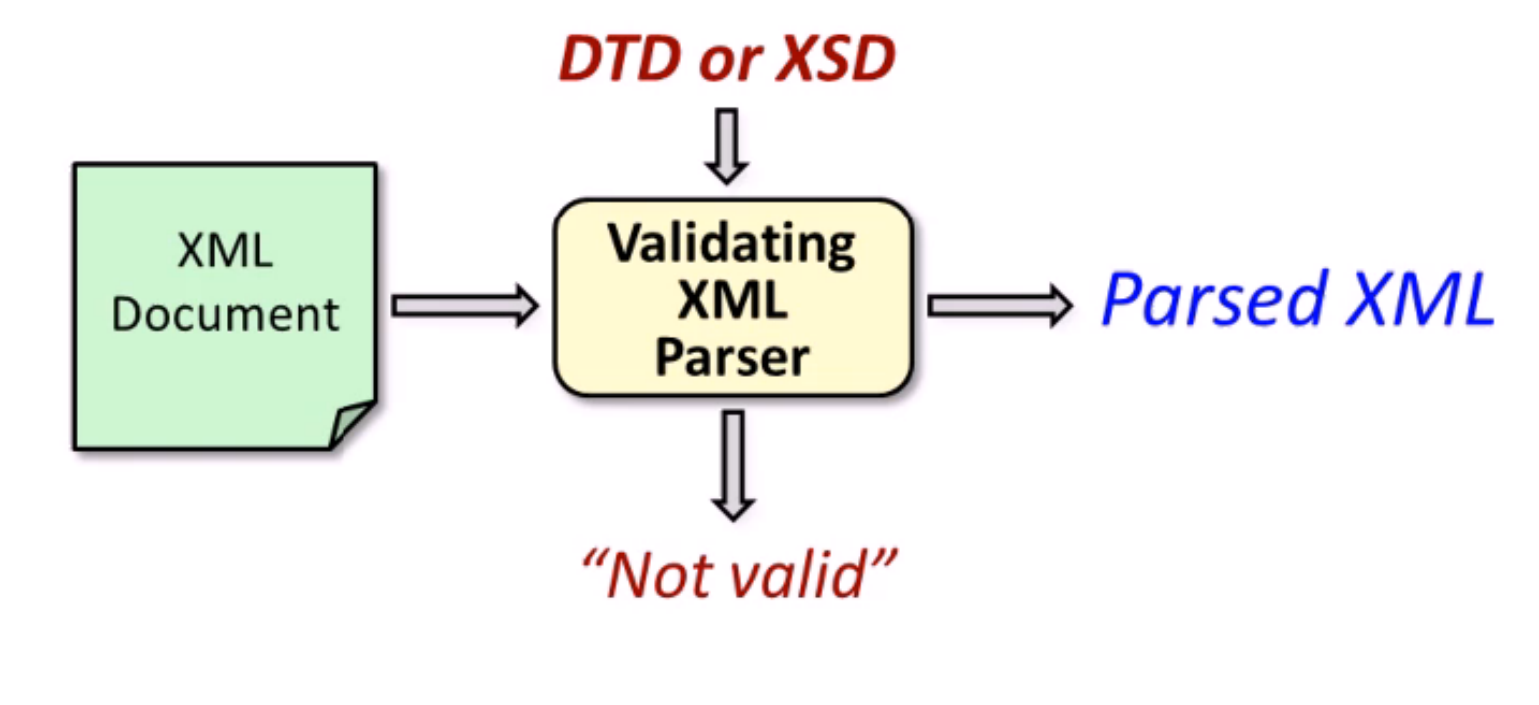
Schema: describes the structure of an XML document, by setting rules specifying which tags and attributes are valid, and how they can be used together

Used to check XML files to make sure they follow the rules set in the schema

Two ways to define a schema:

- Document Type Definition (DTD)
- W3C XML Schema

Valid XML


DTD or XSD
XML Document
Validating XML Parser
Parsed XML
"Not valid"

Section 2 of 7

DTD: DOCUMENT TYPE DEFINITION

DTD

Document Type Definition

A DTD tells the XML parser which elements and attributes to expect in the document as well as the order and nesting of those elements.

- A set of structural rules called *declarations*
- Define tags, attributes, entities
- Specify the order and nesting of tags
- Specify which attributes can be used with which tags.
- DTD can be:
  - Embedded inside XML (internal DTD).
  - Stored in a separate file (external DTD saved as .dtd).

DTD: Declarations

General syntax for declarations

<!*keyword* …. >

Note, not XML!

Four possible *keywords*:

ELEMENT, for defining tags

ATTLIST, for specifying attributes in your tags

ENTITY, for identifying sources of data

NOTATION, for defining data types for non-XML data

DTD: Elements

- **General syntax**
  - <!ELEMENT *element-name* (*content-description*)>
  - Content description specifies what tags may appear inside the named element and whether there may be any plain text in the content
- `EX: <!ELEMENT person (parent+, age, spouse?, sibling*)>`
- An element can be either an internal or a leaf node.
- **Multiplicity**
  - +
  - *
  - ?
- **Leaf elements can be:**
  - #PCDATA
  - EMPTY
  - ANY

DTD: Attributes

- Declaring attributes general syntax:

```
<!ATTLIST element-name
    (attribute-name attribute-type default-value?)+ >
```

- There are 10 attribute types, only **CDATA** will be used.
- Default values
  - A value
  - `#FIXED` value
  - `#REQUIRED` (no default value, each instance must specify value)
  - `#IMPLIED` (default, if not specified)

Data Type Definition

Example

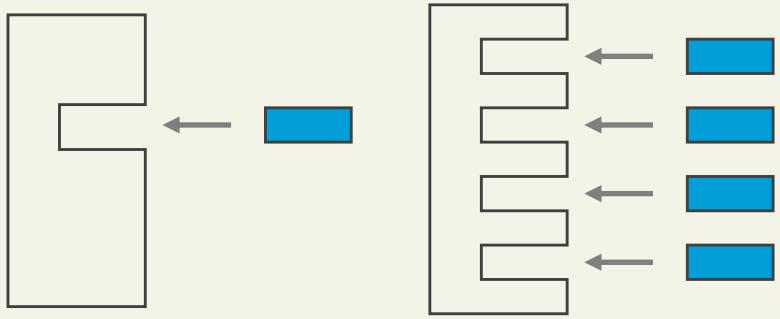
```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE art [
<!ELEMENT art (painting*)>
<!ELEMENT painting (title,artist,year,medium)>
<!ATTLIST painting id CDATA #REQUIRED>
<!ELEMENT title (#PCDATA)>
<!ELEMENT artist (name,nationality)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT nationality (#PCDATA)>
<!ELEMENT year (#PCDATA)>
<!ELEMENT medium (#PCDATA)>
]>
<art>
...
</art>
```

LISTING 17.2 Example DTD

XML Entities

XML document may be distributed among a number of files

- Each unit of information is called an **entity**
- Each entity has a name to identify it
- Defined using an entity declaration
- Used by calling an entity reference



Declared in DTD

```
<!DOCTYPE My_XML_Doc [
  <!ENTITY name "replacement">
]>

<!ENTITY xml "eXtensible Markup
Language">
```

The &xml; includes entities

The eXtensible Markup Language includes entities

Attributes or Elements

There are no rules about when to use attributes and when to use elements.

Avoid XML Attributes?

Some of the problems with using attributes are:

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

DTD: Internal or External

A document type declaration can either contain declarations directly or can refer to another file

Internal

<!DOCTYPE *root-element* [

declarations

]>

External file

<!DOCTYPE *root-name* SYSTEM *"file-name"*>

Data Type Definition

The main drawback with DTDs is that they can only validate the existence and ordering of elements. They provide no way to validate the **values of attributes** or the **textual content of elements**.

For this type of validation, one must instead use **XML schemas**, which have the added advantage of using XML syntax. Unfortunately, schemas have the corresponding disadvantage of being **long-winded** and **harder for humans to read** and comprehend.

Section 3 of 7

XSD: XML SCHEMA

XML Schema

Just one example

```
<xs:schema attributeFormDefault="unqualified"
     elementFormDefault="qualified"
     xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="art">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="painting" maxOccurs="unbounded" minOccurs="0">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="title"/>
              <xs:element name="artist">
                <xs:complexType>
                  <xs:sequence>
                    <xs:element type="xs:string" name="name"/>
                    <xs:element type="xs:string" name="nationality"/>
                  </xs:sequence>
                </xs:complexType>
              </xs:element>
              <xs:element type="xs:short" name="year" />
              <xs:element type="xs:string" name="medium"/>
            </xs:sequence>
            <xs:attribute type="xs:short" name="id" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

LISTING 17.3 Example schema

Section 4 of 7

XPATH: TRAVERSING XML DOCUMENTS

XPath

Another XML Technology

XPath is a standardized syntax for searching an XML document and for navigating to elements within the XML document

XPath is typically used as part of the programmatic manipulation of an XML document in PHP and other languages

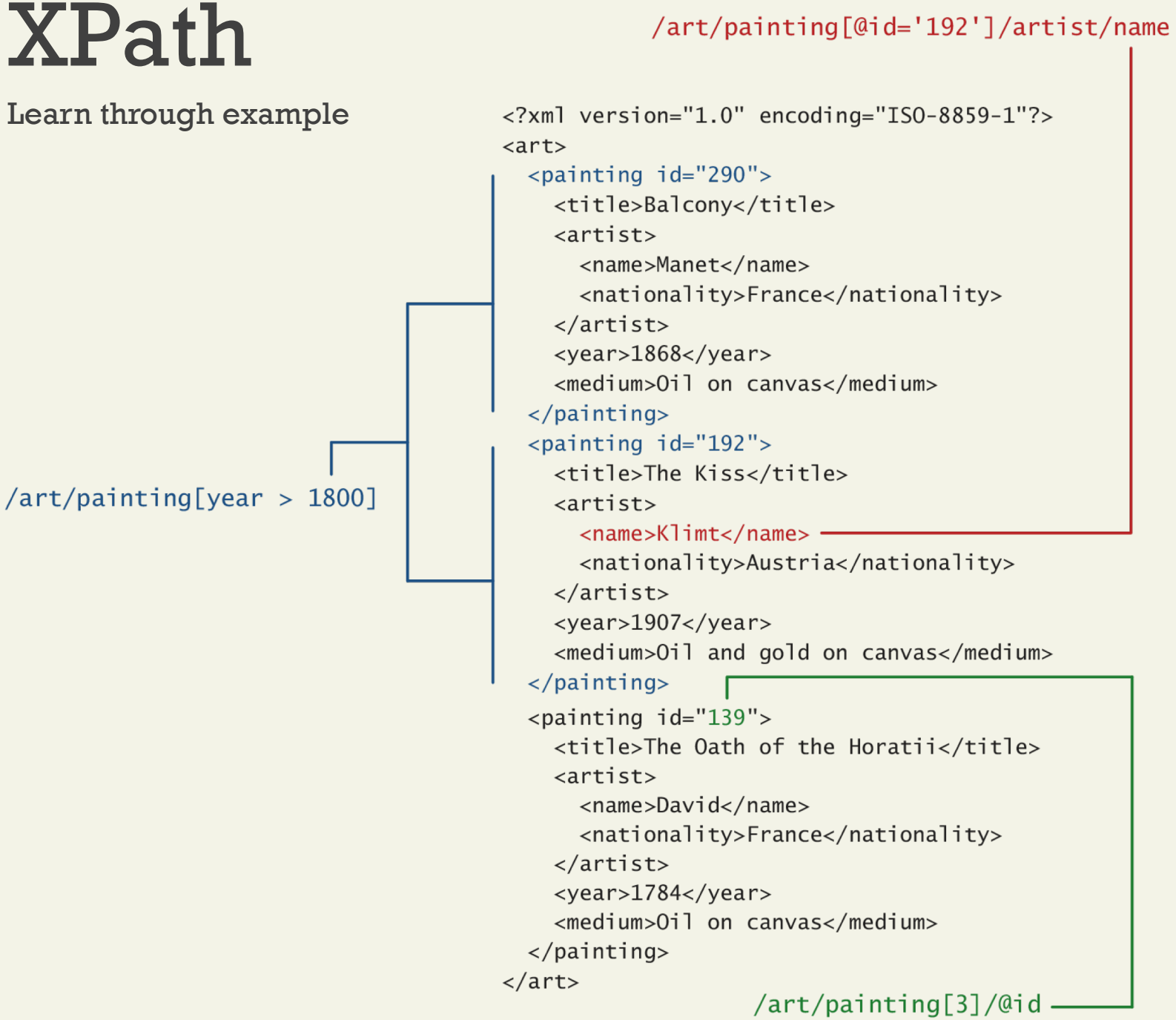
XPath uses a syntax that is similar to the one used in most operating systems to access directories.

XPath

Expression	Meaning
/	Find the root tag in the xml document
/root_tag	Find the root tag, but only if it is named “root_tag”
//element _x	Find all element_x tags, wherever they appear in the xml document
text()	Selects the text content of the current node
@name	Selects the name attribute of the current node
/doc/chapter[4]/section[2]	Selects the second section of the fourth chapter of the doc
body/p[last()]	Selects the last “p” tag in the “body” tag
..	Selects the parent of the current node
/html/body/p[@class=“a”]	Selects all “p” tags in the body under the html tag that have a class attribute whose value is “a”
//p[@class and @style]	Selects every “p” tag in the document that has both a class attribute and a style attribute

XPath

Learn through example



Section 5 of 7

JSON

JSON

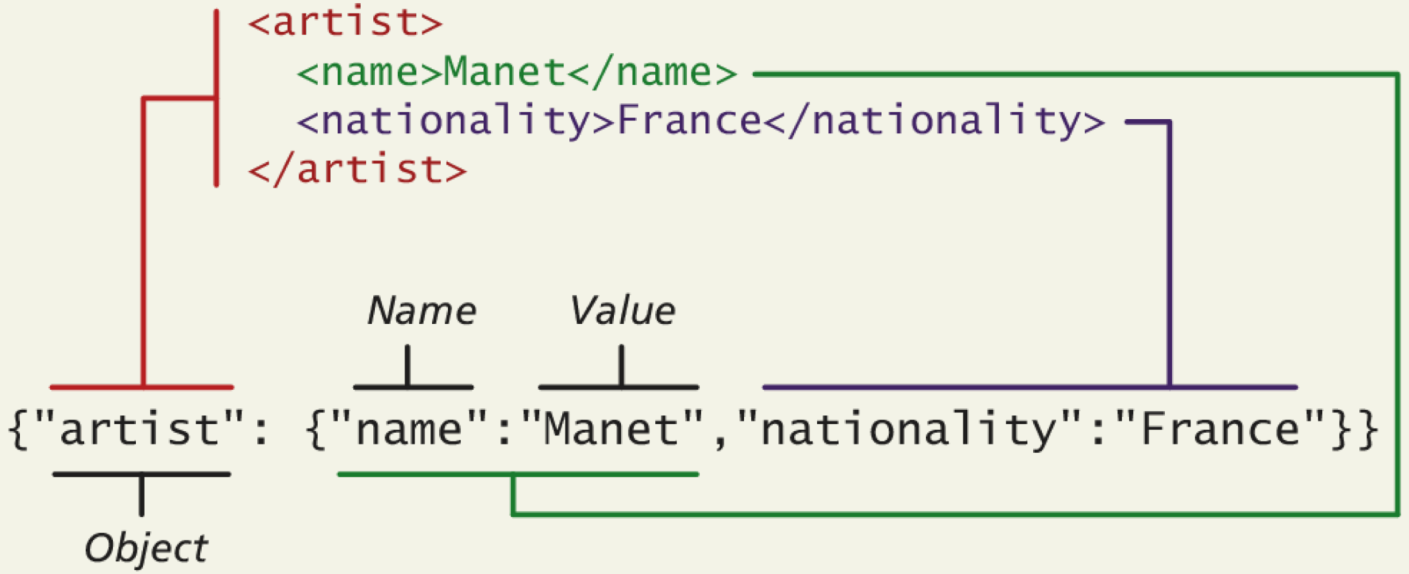
JSON stands for JavaScript Object Notation (though its use is not limited to JavaScript)

Like XML, JSON is a data serialization format. It provides a more concise format than XML.

Many REST web services encode their returned data in the JSON data format instead of XML.

JSON

An example XML object in JSON



JSON

An example XML object in JSON

```
{
   "paintings": [
      {
         "id":290,
         "title":"Balcony",
         "artist":{
            "name":"Manet",
            "nationality":"France"
         },
         "year":1868,
         "medium":"Oil on canvas"
      },
      {
         "id":192,
         "title":"The Kiss",
         "artist":{
            "name":"Klimt",
            "nationality":"Austria"
         },
         "year":1907,
         "medium":"Oil and gold on canvas"
      },
      {
         "id":139,
         "title":"The Oath of the Horatii",
         "artist":{
            "name":"David",
            "nationality":"France"
         },
         "year":1784,
         "medium":"Oil on canvas"
      }
   ]
}
```

LISTING 17.12 JSON representation of XML data from Listing 17.1

Using JSON in JavaScript

Creating JSON JavaScript objects

it is easy to make use of the JSON format in JavaScript:

```
var a = {"artist": {"name":"Manet","nationality":"France"}};

alert(a.artist.name + " " + a.artist.nationality);
```

When the JSON information will be contained within a string (say when downloading) the JSON.parse() function can be used to transform the string containing into a JavaScript object

Using JSON in JavaScript

Convert string to JSON object and vice versa

```
var text = '{"artist": {"name":"Manet","nationality":"France"}}';

var a = JSON.parse(text);

alert(a.artist.nationality);
```

JavaScript also provides a mechanism to translate a JavaScript object into a JSON string:

```
var text = JSON.stringify(artist);
```

Using JSON in PHP

JSON on the server

Converting a JSON string into a PHP object is quite straightforward:

```php
<?php
  // convert JSON string into PHP object
  $text = '{"artist": {"name":"Manet","nationality":"France"}}';
  $anObject = json_decode($text);
  echo $anObject->artist->nationality;

  // convert JSON string into PHP associative array
  $anArray = json_decode($text, true);
  echo $anArray['artist']['nationality'];
?>
```

Notice that the json_decode() function can return either a PHP object or an associative array.

Using JSON in PHP

Go the other way

To go the other direction (i.e., to convert a PHP object into a JSON string), you can use the json_encode() function.

```php
// convert PHP object into a JSON string
$text = json_encode($anObject);
```

Using JSON in PHP

JSON on the server

Since JSON data is often coming from an external source, always check for parse errors before using it, via the json_last_error() function:

```php
<?php
  // convert JSON string into PHP object
  $text = '{"artist": {"name":"Manet","nationality":"France"}}';
  $anObject = json_decode($text);
  // check for parse errors
  if (json_last_error() == JSON_ERROR_NONE) {
    echo $anObject->artist->nationality;
  }
?>
```

Self-Reading

XML PROCESSING

XML Processing

Two types

XML processing in PHP, JavaScript, and other modern development environments is divided into two basic styles:

- The **in-memory approach**, which involves reading the entire XML file into memory into some type of data structure with functions for accessing and manipulating the data.
- The **event or pull approach**, which lets you pull in just a few elements or lines at a time, thereby avoiding the memory load of large XML files.

XML Processing

In JavaScript

All modern browsers have a built-in XML parser and their JavaScript implementations support an in-memory XML DOM API.

You can use the already familiar DOM functions such as getElementById(), getElementsByTagName(), and createElement() to access and manipulate the data.

XML Processing

```
<script>
if (window.XMLHttpRequest)  {
  // code for IE7+, Firefox, Chrome, Opera, Safari
  xmlhttp=new XMLHttpRequest();
}
else  {
  // code for old versions of IE (optional you might just decide to
  // ignore these)
  xmlhttp=new ActiveXObject("Microsoft.XMLHTTP");
}
// load the external XML file
xmlhttp.open("GET","art.xml",false);
xmlhttp.send();
xmlDoc=xmlhttp.responseXML;

// now extract a node list of all <painting> elements
paintings = xmlDoc.getElementsByTagName("painting");
if (paintings) {
   // loop through each painting element
   for (var i = 0; i < paintings.length; i++)
   {
      // display its id attribute
      alert("id="+paintings[i].getAttribute("id"));

      // find its <title> element
      title = paintings[i].getElementsByTagName("title");
      if (title) {
         // display the text content of the <title> element
         alert("title="+title[0].textContent);
      }
   }
}
</script>
```

LISTING 17.5 Loading and processing an XML document via JavaScript

XML Processing

With JQuery

```
art = '<?xml version="1.0" encoding="ISO-8859-1"?>';
art += '<art><painting id="290"><title>Balcony … </art>';

// use jQuery parseXML() function to create the DOM object
xmlDoc = $.parseXML( art );
// convert DOM object to jQuery object
$xml = $( xmlDoc );

// find all the painting elements
$paintings = $xml.find( "painting" );
// loop through each painting element
$paintings.each(function() {
   // display its id
   alert($(this).attr("id"));
   // find the title element within the current painting element
   $title = $(this).find( "title" );
   // and display its content
   alert( $title.text() );
});
```

LISTING 17.6 XML processing using jQuery

XML Processing

With PHP

PHP provides several extensions or APIs for working with XML including:

- The **SimpleXML** extension which loads the data into an object that allows the developer to access the data via array properties and modifying the data via methods.
- The **XMLReader** is a read-only pull-type extension that uses a cursor-like approach similar to that used with database processing

XML Processing

With PHP using Simple XML

```php
<?php

$filename = 'art.xml';
if (file_exists($filename)) {
   $art = simplexml_load_file($filename);

   // access a single element
   $painting = $art->painting[0];
   echo '<h2>' . $painting->title . '</h2>';
   echo '<p>By ' . $painting->artist->name . '</p>';
   // display id attribute
   echo '<p>id=' . $painting["id"] . '</p>';

   // loop through all the paintings
   echo "<ul>";
   foreach ($art->painting as $p)
   {
      echo '<li>' . $p->title . '</li>';
   }
   echo '</ul>';
} else {
   exit('Failed to open ' . $filename);
}

?>
```

Variable and attribute names taken from xml

LISTING 17.8 Using simple XML

XML Processing

With PHP using Simple XML and XPath

```
$art = simplexml_load_file($filename);

$titles = $art->xpath('/art/painting/title');
foreach ($titles as $t) {
   echo $t . '<br/>';
}

$names = $art->xpath('/art/painting[year>1800]/artist/name');
foreach ($names as $n) {
   echo $n . '<br/>';
}
```

LISTING 17.9 Using XPath with SimpleXML

XML Processing

```
$filename = 'art.xml';
if (file_exists($filename)) {

   // create and open the reader
   $reader = new XMLReader();
   $reader->open($filename);

   // loop through the XML file
   while ( $reader->read() ) {
     $nodeName = $reader->name;

     // since all sorts of different XML nodes we must check
     // node type
     if ($reader->nodeType == XMLREADER::ELEMENT
         && $nodeName == 'painting') {
         $id = $reader->getAttribute('id');
         echo '<p>id=' . $id . '</p>';
     }

     if ($reader->nodeType == XMLREADER::ELEMENT
         && $nodeName =='title') {
         // read the next node to get at the text node
         $reader->read();
         echo '<p>' . $reader->value . '</p>';
     }
   }
} else {
   exit('Failed to open ' . $filename);
}
```

Less “automatic”

More Verbose

LISTING 17.10 Using XMLReader

XML Processing

Why choose when you can use both

```
// create and open the reader
$reader = new XMLReader();
$reader->open($filename);

// loop through the XML file
while($reader->read()) {
  $nodeName = $reader->name;
  if ($reader->nodeType == XMLREADER::ELEMENT
      && $nodeName =='painting') {
      // create a SimpleXML object from the current painting node
      $doc = new DOMDocument('1.0', 'UTF-8');
      $painting = simplexml_import_dom($doc->importNode
                  ($reader- >expand(),true));
      // now have a single painting as an object so can output it
      echo '<h2>' . $painting->title . '</h2>';
      echo '<p>By ' . $painting->artist->name . '</p>';
  }
}
```

LISTING 17.11 Combining XMLReader and SimpleXML

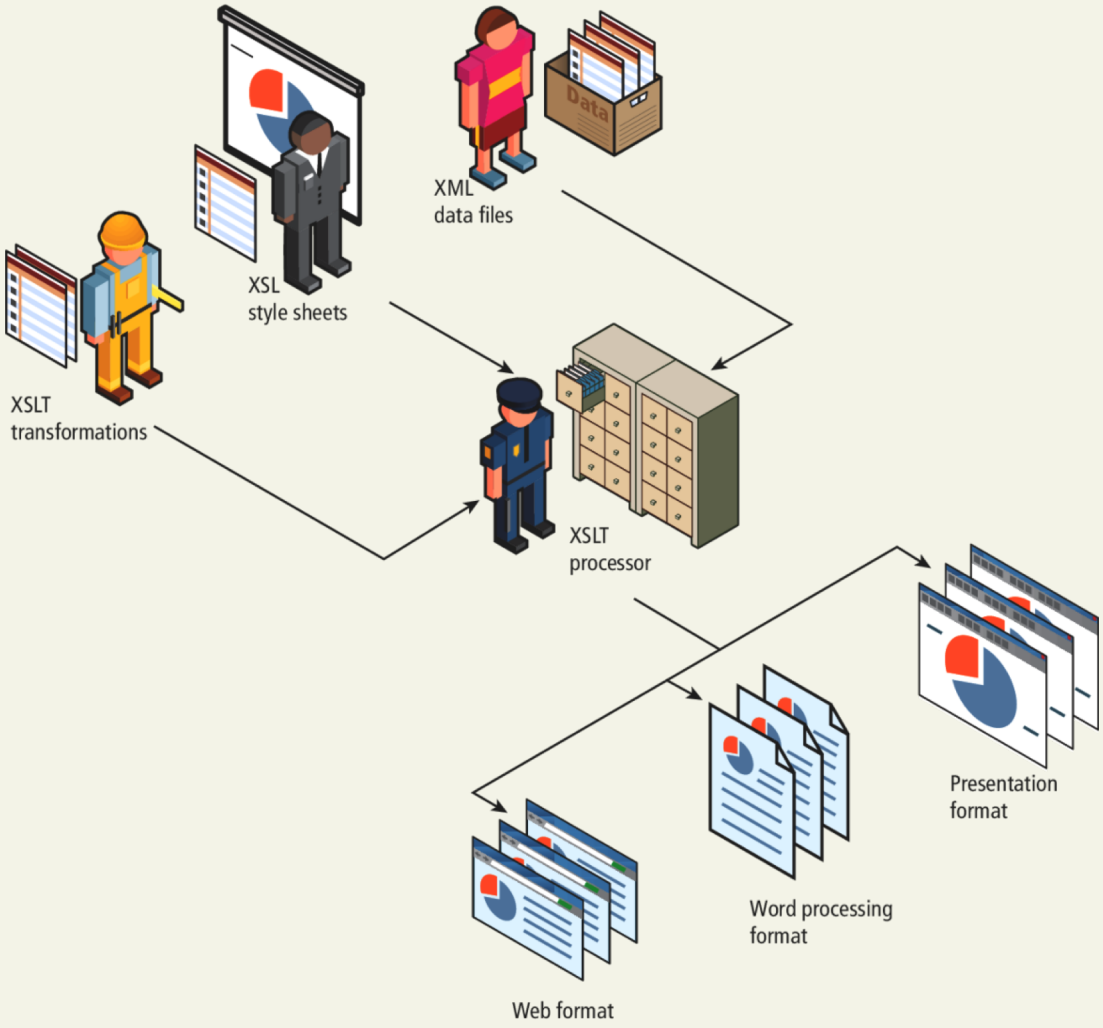
Self Reading

XSLT: XML STYLE TRANSFORMATION

XSLT

XML Stylesheet Transformations

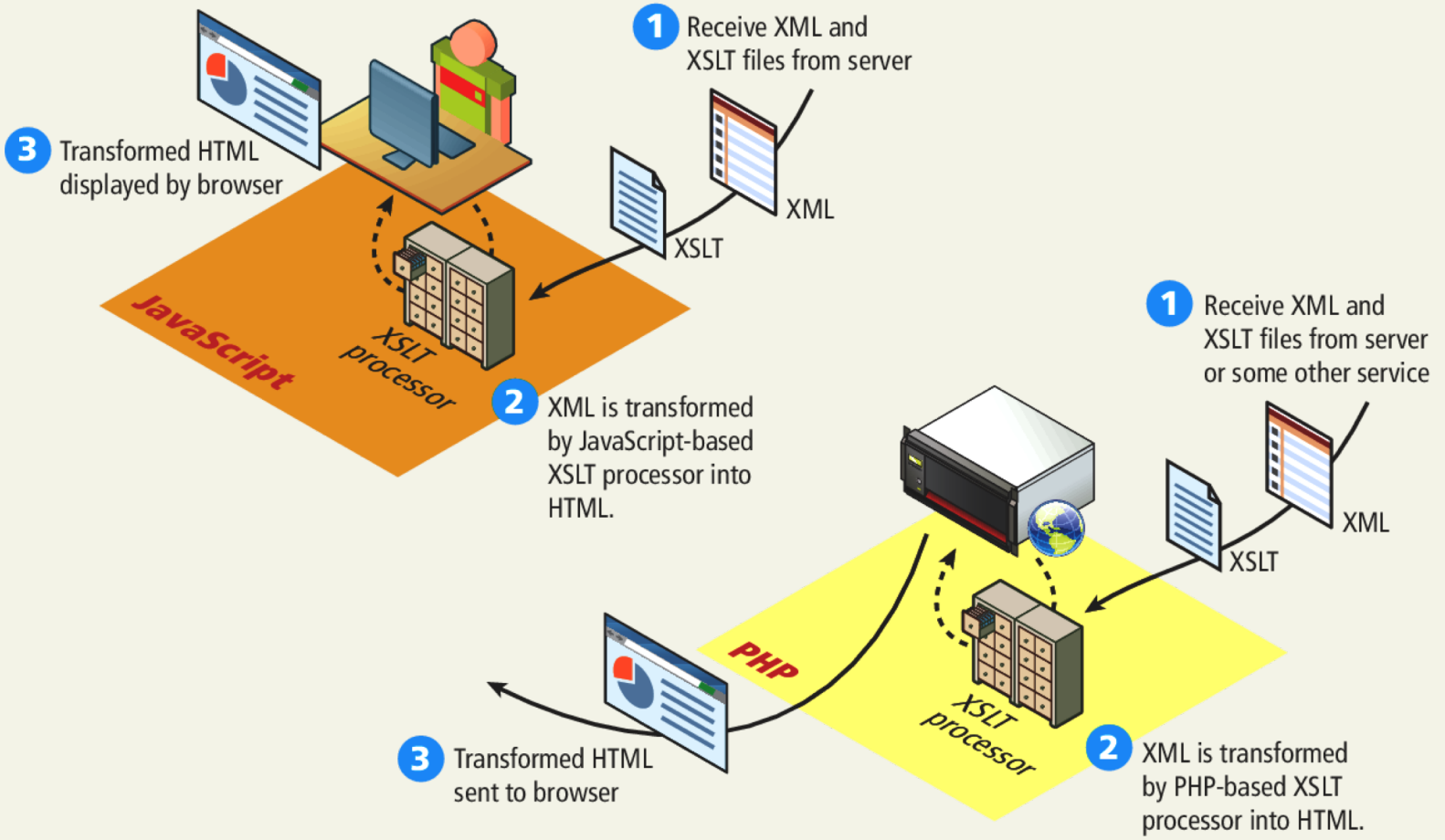
XSLT is an XML-based programming language that is used for transforming XML into other document formats



XSLT

Another usage

XSLT is also used on the server side and within JavaScript



XSLT

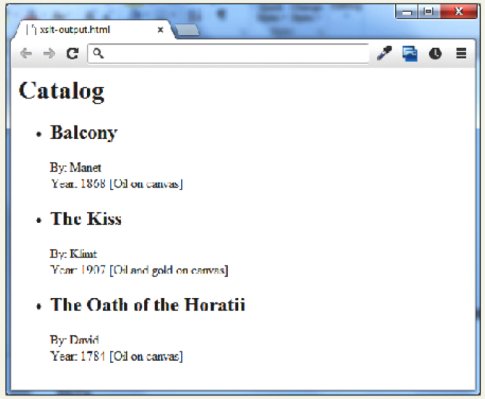
Example XSLT document that converts the XML from Listing 17.1 into an HTML list

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<html xsl:version="1.0"
      xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
      xmlns="http://www.w3.org/1999/xhtml">
<body>
   <h1>Catalog</h1>
   <ul>
    <xsl:for-each select="/art/painting">
      <li>
        <h2><xsl:value-of select="title"/></h2>
        <p>By: <xsl:value-of select="artist/name"/><br/>
           Year: <xsl:value-of select="year"/>
           [<xsl:value-of select="medium"/>]</p>
      </li>
    </xsl:for-each>
   </ul>
</body>
</html>
```

LISTING 17.4 An example XSLT document

XSLT

An XML parser is still needed to perform the actual transformation




```
<?xml version="1.0" encoding="UTF-8"?>
<html xmlns="http://www.w3.org/1999/xhtml">
<body>
<h1>Catalog</h1>
<ul>
  <li>
    <h2>Balcony</h2>
    <p>By: Manet<br/>
    Year: 1868 [Oil on canvas]</p>
  </li>
  <li>
    <h2>The Kiss</h2>
    <p>By: Klimt<br/>
    Year: 1907 [Oil and gold on canvas]</p>
  </li>
  <li>
    <h2>The Oath of the Horatii</h2>
    <p>By: David<br/>Year: 1784 [Oil on canvas]</p>
  </li>
</ul>
</body>
</html>
```

XSLT

- XSLT Stylesheets are defined using the <xsl:stylesheet> root tag
- Stylesheets typically contain one or more <xsl:template> tags that define each template
  - Templates have name or/and match attributes
- Templates contain other XSLT tags that control how the XML data is transformed

Common XSLT Elements

<xsl:stylesheet>

<xsl:template name="name" match ="xpath">

<xsl:value-of select="xpath">

<xsl:attribute>

<xsl:text>

<xsl:for-each select="xpath">

<xsl:if test="condition">

<xsl:choose>, <xsl:when>, <xsl:otherwise>

<xsl:sort select="xpath">

What You've Learned

1. XML Overview
2. DTD: Document Type Definition
3. XSD: XML Schema Document
4. XPATH
5. JSON
6. Self-Reading: XML Processing
7. Self Reading: XML Style Transformations