SoftServe
Ability. Agility. Advantage.

Introduction to XML

SoftServe University
November, 2010

Agenda

- History of XML
- What is XML?
- Why not HTML?
  - XML Structure
  - Defining DTD (Document Type Definition)
- Declaring DTD and Schemas
- Case studies

History of XML

- The concept of the hypertext has appeared in 1965 (Base principles are formulated in 1945).
- In 1986 was created **SGML** language (Standard Generalized Markup Language). The main defect — complexity.
- With its help language of a marking of hypertext documents — **HTML** has been created. Specification HTML has been confirmed in 1992.
- The main **defect of HTML** — **limitation of quantity tags. Indifference to document structure.**
- Was **created** language **XML**: Simplicity HTML; Logic of marking SGML; Internet requirements.
- It is possible to consider as year of birth XML 1996. Specification XML has been confirmed in 1998.
- There are **two current versions** of XML. The first (XML 1.0) was initially defined in 1998. It has undergone minor revisions since then, without being given a new version number, and is currently in its fifth edition, as published on November 26, 2008.
- The second (**XML 1.1**) was initially published on February 4, 2004, the same day as XML 1.0 Third Edition, and is currently in its second edition, as published on August 16, 2006. It contains features (some contentious) that are intended to make XML easier to use in certain cases.

What is XML?

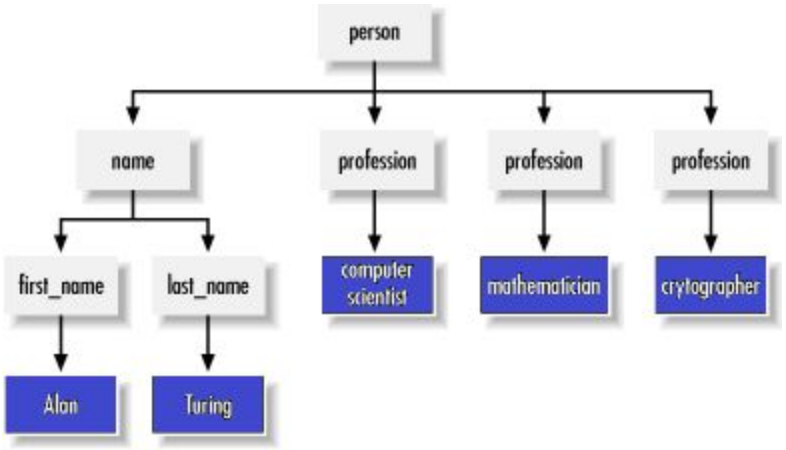
- XML – **Extensible Markup Language**
- Based upon HTML
- Describe your own tags
- Uses **DTD** (Document Type Definition) or **Schemas** to describe the data
- XML is not a replacement for HTML
- XML is a language for creating other languages

```
<?xml version="1.0" encoding="UTF-8"?>
<case>
   <product>
      <manufacturer>Sony</manufacturer>
      <speed>16</speed>
      <description>CD-RW disks</description>
   </product>
   <product>
      <manufacturer>Verbatim</manufacturer>
      <speed>52</speed>
      <description>DVD+R disks</description>
   </product>
</case>
```

What is XML?

- **XML Trees**. Parents and children. The root element

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```



What is XML?

- XML Attributes
- One element cannot have **multiple attributes** with the same name.
- Attribute values are in single quotes (') or double quotes (").

```
<?xml version="1.0" encoding="UTF-8"?>
<person  born="1912-06-23"   died="1954-06-07">
   Alan Turing
</person>

Element:
<book>
   <title>Trisman Shandy</title>
</book>

Attribute:
<book title="Trisman Shandy"></book>


<animal>Lion</animal>

<animal class="mammals">Lion</animal>
```

XML Elements

- Uses same building blocks as HTML – Element, Attribute, and Values
- An element consists of opening tag and closing tag **<animal>Lion</animal>**
- **<animal class="mammals">Lion</animal>** animal is the element, class is the attribute and mammals is the value
- First line: <?xml version="1.0"?>
- Begin tag-end tag.
- Not cross-overlapped.
- Root tag.
- Text with spaces.
- Case-sensitive.
- **XML Names Element** and other XML names may contain essentially any alphanumeric character. This includes the standard English letters A through Z and a through z as well as the digits 0 through 9. XML names may also include non-English letters, numbers, and ideograms such as ö. XML names may not contain other punctuation characters such as quotation marks(“), apostrophes(‘), dollar signs($), carets (^), percent symbols (%), and semicolons(;).

Rules for XML

- Documents need to be "**well-formed**" which means following these rules:
  - A root element is required that contains all other elements;
  - Every element must have a closing tag;
    - <picture file="test.jpg"/>
    - <name>Lion</name>
  - Actual and declared character encoding should match document;
  - Special symbols need to be declared in DTD except for & < > " ' which are predefined;
- Comments are written like this:
  - <!--This is the comment line-->
  - To denote the **element-free**, which is called an empty element, you must use a special form of record

    <foo></foo> or <foo />

Rules for XML

- When an XML document includes samples of XML or HTML source code, the < and & characters in those samples must be encoded as < and &.
- Entity References
  - **<** The less-than sign; a.k.a. the opening angle bracket (**<**)
  - **&** The ampersand (**&**)
  - **>** The greater-than sign; a.k.a. the closing angle bracket (**>**)
  - " The straight, double quotation marks (**"**)
  - **'** The apostrophe; a.k.a. the straight single quote (**'**)
- The more sections of literal code a document includes and the longer they are, the more tedious this encoding becomes. Instead you can enclose each sample of literal code in a CDATA section. A **CDATA section** is set off by a **<![CDATA[and]]>**. Everything between the <![CDATA[**and the**]]> is treated as raw character data. Less-than signs don't begin. Ampersands don't start entity references. Everything is simply character data, not markup.
  - <![CDATA[<name>Lion</name>]]> **The <name> element** will not be interpreted

Schemas and validation

- In addition to being well-formed, an XML document may be valid. This means that it contains a reference to a **Document Type Definition** (**DTD**), and that its elements and attributes **are declared in that DTD** and follow the grammatical rules for them that the DTD specifies.
- The **oldest schema** language for XML is the Document Type Definition (DTD), inherited from **SGML**. DTDs have the following benefits:
  - DTD support is ubiquitous due to its **inclusion** in the **XML 1.0** standard.
  - DTDs are terse compared to element-based schema languages.
  - DTDs allow the declaration of standard public entity sets for publishing characters.
  - DTDs define a document type rather than the **types** used by a namespace, thus grouping all constraints for a document in a single collection.
- DTDs have the following limitations:
  - They have no explicit support for newer features of XML, most importantly namespaces.
  - Lack of expressiveness. DTDs only support rudimentary datatypes.
  - Lack of readability. They use a syntax based on regular expression syntax, inherited from SGML, to describe the schema.
- The peculiar feature that distinguish DTDs from other schema types are the syntactic support for embedding a DTD within XML.

A Simple DTD Example

```
<!ELEMENT person      (name, profession*)>
<!ELEMENT name        (first_name, last_name)>
<!ELEMENT first_name  (#PCDATA)>
<!ELEMENT last_name   (#PCDATA)>
<!ELEMENT profession  (#PCDATA)>
```

- The first element declaration in states that each person element must contain exactly one name child element followed by zero or more profession elements.
  - ? - Zero or one of the element is allowed.
  - * - Zero or more of the element is allowed.
  - + - One or more of the element is required.
- Thus, every person must have a name and may or may not have a profession or multiple professions. However, the name must come before all professions. For example, this person element is valid:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
</person>
```

A Simple DTD Example

- However, this person element **is not valid** because it omits the name:

```
<person>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

- This person element **is not valid** because a profession element comes before the name:

```
<person>
  <profession>computer scientist</profession>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

A Simple DTD Example

- The person element may not contain any element except those listed in its declaration. The only extra character data it can contain is whitespace. For example, this **is an invalid** person element because it adds a **publication** element:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
  <publication>On Computable Numbers...</publication>
</person>
```

- This **is an invalid** person element because it adds **some text outside** the allowed children:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  was a <profession>computer scientist</profession>,
  a <profession>mathematician</profession>, and a
  <profession>cryptographer</profession>
</person>
```

A Simple DTD Example

- **Internal** DTD Subsets

```
<?xml version="1.0" encoding="UTF-8" ?>
<!DOCTYPE person [
  <!ELEMENT first_name (#PCDATA)>
  <!ELEMENT last_name  (#PCDATA)>
  <!ELEMENT profession (#PCDATA)>
  <!ELEMENT name       (first_name, last_name)>
  <!ELEMENT person     (name, profession*)>
]>
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
  <profession>cryptographer</profession>
</person>
```

A Simple DTD Example

- Declaring a Personal **External** DTD. Refer to the DTD file (**person.dtd**) that you have created by the URL
- When you use an external DTD subset, you should give the **standalone** attribute of the XML declaration the value **no**. For example:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE person SYSTEM "http://www.softservecom.com/dtds/person.dtd">
```

- If your DTD will be used by others, you should name your DTD in a standard way by a formal public identifier (FPI):

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE person PUBLIC "\\John Smith\\DTD Person\\EN\\"
    "http://www.animals.com/xml/person.dtd">
```

- The XML parser will try to locate the DTD in a public repository.
- If the DTD is not there, then the parser looks for the URL given.

A Simple DTD Example

- **Attribute Declarations**

```
<!ATTLIST image source  CDATA #REQUIRED
                width           CDATA #REQUIRED
                height          CDATA #REQUIRED
                alt             CDATA #IMPLIED
>
```

- This declaration says the source, width, and height attributes **are required**. However, the alt attribute is optional and may be omitted from particular image elements. All four attributes are declared to contain character data, the most generic attribute type.
- **CDATA**: A CDATA attribute value can contain any string of text acceptable in a well-formed XML attribute value.
- **NMTOKEN**: An XML name token is very close to an XML name. It must consist of the same characters as an XML name, that is, alphanumeric and/or ideographic characters and the punctuation marks _, -, ., and :.

A Simple DTD Example

- **NMTOKENS** A NMTOKENS type attribute contains one or more XML name tokens separated by whitespace. For example, you might use this to describe the dates attribute of a performances element, if the dates were given in the form 08-26-2000, like this:

```
<performances dates="08-21-2001 08-23-2001 08-27-2001">
Kat and the Kings
</performances>
```

- **Enumeration**:

```
<!ATTLIST date month (January | February | March | April | May | June  | July |
August | September | October | November | December) #REQUIRED>
```

- **ENTITY** An ENTITY type attribute contains the name of an unparsed entity declared elsewhere in the DTD. For instance, a **movie** element might have an entity attribute identifying the MPEG or QuickTime file to play when the movie was activated:

```
<!ATTLIST movie source ENTITY #REQUIRED>
```

- If the DTD declared an unparsed entity named X-Men-trailer, then this movie element might be used to embed that video file in the XML document:

```
<movie source="X-Men-trailer"/>
```

A Simple DTD Example

- an.ent is an **external file** with XML code

  <!ENTITY animal_names SYSTEM "an.ent">

- Reference shortcut by www;

  <!ENTITY www "World Wide Web">

- **ID** An ID type attribute must contain an **XML name** (not a name token but a name) that is unique within the XML document. More precisely, no other ID type attribute in the document can have the same value.
- **IDREF** An IDREF type attribute **refers** to the ID type attribute of some element in the document. Thus, it must be an XML name. IDREF attributes are commonly used to establish relationships between elements when simple containment won't suffice.

```
<!ATTLIST employee social_security_number ID      #REQUIRED>
<!ATTLIST project  project_id             ID      #REQUIRED>
<!ATTLIST team_member person              IDREF   #REQUIRED>
<!ATTLIST assignment  project_id          IDREF   #REQUIRED>
```

A Simple DTD Example

```
<project project_id="p1">
  <goal>Develop Strategic Plan</goal>
  <team_member
person="ss078-05-1120"/>
  <team_member
person="ss987-65-4320"/>
</project>
<project project_id="p2">
  <goal>Deploy Linux</goal>
  <team_member
person="ss078-05-1120"/>
  <team_member
person="ss9876-12-3456"/>
</project>
```

```
<employee
social_security_label="ss078-05-1120">
  <name>Fred Smith</name>
  <assignment project_id="p1"/>
  <assignment project_id="p2"/>
</employee>
<employee
social_security_label="ss987-65-4320">
  <name>Jill Jones</name>
  <assignment project_id="p1"/>
</employee>
<employee
social_security_label="ss9876-12-3456">
  <name>Sydney Lee</name>
  <assignment project_id="p2"/>
</employee>
```

In this example, the project_id attribute of the project element and the social_security_number attribute of the employee element would be declared to have type ID. The person attribute of the team_member element and the project_id attribute of the assignment element would have type IDREF.

XML Schema

- **XDR** (XML Data Reduced) - **old** Microsoft schema.
- **XSD** (XML Schema Definition) opened standart from W3C.

```
<?xml version="1.0"?>
<books schemalocation="file:///c:/data/books.xsd">
    <book>
        <title>Beginning Visual C#</title>
        <author>Karli Watson</author>
        <code>7582</code>
    </book>
    <book>
        <title>Professional C# 3th Edition</title>
        <author>Simon Robinson</author>
        <code>7043</code>
    </book>
</books>
```

XML Schema

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
  <element name="books">
    <complextype>
      <choise maxOccurs="unbounded">    <!-- NO ORDER, COUNT >1 -->
        <element name="book">
          <complextype>
            <sequence>                  <!-- ORDER -->
              <element name="title" />
              <element name="author" />
              <element name="code" />
            </sequence>
          </complextype>
        </element>
      <choise>
      <attribute name="schemalocation" />
    </complextype>
  </element>
<schema>
```

XML Schema

- XML Schema was designed to determine the **rules** that must comply with the document.
- XML Schema has been designed so that it can be used to create software for document processing XML.
- XML Schema includes:
  - Element and attribute names;
  - Relationships between elements and attributes and their structure;
  - Data types.
- **Example**: XML Schema describes the data about the country (file "country.xsd")

```
<country
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xsi:noNamespaceSchemaLocation="country.xsd">
  <countryname>France</countryname>
  <population>59.7</ population >
</ country >
```

A Simple XSD Example

- **XSD Schema** (file "country.xsd")

```
<xs:schema
      xmlns:xs="http://www.w3.org/2001/XMLSchema">
<xs:element name="country" type="country"/>
<xs:complexType name="country">
     <xs:sequence>
          <xs:element name="countryname" type="xs:string"/>
           <xs:element name="population" type="xs:decimal"/>
     </xs:sequence>
</xs:complexType>
</xs:schema>
```

Processing of XML-documents. XML-parser

- For working with XML are used XML-parser (**software** processing and visualization).
- There are two basic types of parsers:
  - Simple API for XML (**SAX**) and
  - Document Object Model (**DOM**).
- **SAX** is based on the cursor and the events that occur when passing over the nodes of XML documents. SAX-parser doesn't require much memory. You can navigate through the document in one direction only.
- **DOM** loads the document completely in memory and presents it in a tree. You can freely move through the XML-document.
- **XSL** Transformations (XSLT) - a transformation language, intended to change the structure of an XML document or convert it to a document on a different dialect of XML. XSLT processor can use XSLT style transformations for tree data. Can then be **serialized** in XML, HTML, or other format.
- XSL-transformation is performed, usually on the server, then its result is sent to the client browser. XSL based on XML, which means that XSL is more flexible, versatile.

Cascading Style Sheet (CSS)

- **CSS**-formatting is applied to the HTML(XML)-document, the browser on the client side.
- XML has no default formatting.
- XML needs information on how a tag will be displayed before being shown on a browser.
- Every element to be styled with CSS can be thought of as an invisible box.
- The user controls the size, color, and spacing of this box.

```
<animal>
  <name>lion</name>
  <weight>350 pounds</weight>
  <height>350 pounds</height>
</animal>
```

Format of CSS

- animal {display:block;position:relative}
- **animal** is the element and the declarations inside the curly brackets determine how the chosen element will be displayed
- Declarations should be saved in a file with .css extension

- **Inserting a Style Sheet to XML Document**
- <?xml version="1.0" ?>
- <?xml-stylesheet type="text/css" href="animal.css" ?>

- **animal.css** is the style sheet that contains the declarations. For example,

```
animal {display:block}
weight {display:block}
height {display:inline}
```

Format of CSS

- **Block-level** elements start at the beginning of a line and so does the element that follows
  - animal {display:block}
- **Inline** elements appear within a line
  - weight {display:inline}
- Elements can be not displayed at all
  - height {display:none}

- **Positioning the Element**
  - description {display:block;position:relative;left:100px}
- **Relative** - Moves the element with respect to the original position.
- In this case the description block is moved 100 pixels to the **left**
- Element can be moved **left**, **right**, **top**, **bottom**
  - description {display:block;position:absolute;left:100px}
- **absolute** -Moves the element with respect to the parent position.

Format of CSS

- Setting the **Height or Width**.
  - description {display:block;position:relative;left:100px; width:340px}
- Sets the **width** of the element box to be 340px wide
- **height** can also be set to an element.
- Changing the Foreground **Color.**
  - name{display:block;color:magenta}
- There are 16 predefined colors
- **magenta** can also be written in the #rrggbb hexadecimal representation or rgb(r,g,b) where r,g, and b are integers from 0-255 that contains the amount of red, green, or blue in the color.
- Changing the **Background**.
  - name{display:block;color:magenta; background:url(background.jpg)}
  - name{display:block;color:magenta; background:color:magenta}
- This sets the background for the element, not the entire page

Format of CSS

- Choosing a **Font**.
  - name{font-family:Georgia, Times}
- The user can specify more than one font, just in case the font is not available
- The font appears in the element specified
- **Italics and Bold** Elements.
  - name{font-style:italic}
  - name{font-weight:bold}
  - name{font-weight:bolder}
  - name{font-weight:lighter}
- Bold elements can be **bold**, **bolder**, or **lighter**
- **Underline** and Font Size.
  - name{text-decoration:underline}
  - name{font-size:10pt}
- font size can also be absolute like **small**, **medium**, **large**

JSON

- **JSON** (JavaScript Object Notation) - text data interchange format based on JavaScript and is usually used with this language. Like many other text formats, JSON is easy to read.
- Derives from JavaScript (on a subset of the standard 1999), the format is language independent and can be used with almost any programming language. For many languages there is a ready code for creating and processing data in JSON.
- **JSON** is built on **two structures**:
  - A set of name/value pairs. In different languages it is implemented as an object, record, structure, dictionary, hash table, a list of key or associative array.
  - Ordered a set of values. In many languages it is implemented as an array, vector, list, or sequence.
- The string is very similar to a string in C and Java. Number is also very similar to C or Java-number, except that only used the decimal format. Spaces can be inserted between any two characters.
- Although JSON is intended as a data serialization format, its design as a subset of the JavaScript programming language poses several security concerns.

JSON, Example

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address":
    {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021"
    },
    "phoneNumber":
    [
        {
          "type": "home",
          "number": "212 555-1234"
        },
        {
          "type": "fax",
          "number": "646 555-4567"
        }
    ]
}
```

The following example shows the JSON representation of an object that describes a **person**.

The object has string fields for first name and last name, a number field for age, contains an object representing the person's address, and contains a list (an array) of phone number objects.

An equivalent form for the above in XML could be:

JSON, XML

```
<Objects>
  <Property><Key>firstName</Key> <String>John</String></Property>
  <Property><Key>lastName</Key> <String>Smith</String></Property>
  <Property><Key>age</Key> <Number>25</Number></Property>
  <Property><Key>address</Key>
    <Object>
      <Property><Key>streetAddress</Key> <String>21 2nd Street</String></Property>
      <Property><Key>city</Key> <String>New York</String></Property>
      <Property><Key>state</Key> <String>NY</String></Property>
      <Property><Key>postalCode</Key> <String>10021</String></Property>
    </Object>
  </Property>
  <Property><Key>phoneNumber</Key>
    <Array>
      <Object>
        <Property><Key>type</Key> <String>home</String></Property>
        <Property><Key>number</Key> <String>212 555-1234</String></Property>
      </Object>
      <Object>
        <Property><Key>type</Key> <String>fax</String></Property>
        <Property><Key>number</Key> <String>646 555-4567</String></Property>
      </Object>
    </Array>
  </Property>
</Objects>
```

JSON, XML

- However, the following simplified XML using attributes (name-value pairs) is more likely be used by an XML practitioner:

```
<Person firstName="John" lastName="Smith" age="25">
  <Address streetAddress="21 2nd Street" city="New York"
                   state="NY" postalCode="10021" />
  <PhoneNumbers>
    <PhoneNumber type="home" number="212 555-1234"/>
    <PhoneNumber type="fax"  number="646 555-4567"/>
  </PhoneNumbers>
</Person>
```

- Note that while both the JSON and XML forms can carry the same data, the (second) XML example also conveys semantic content/meaning, meaning it will also need an **XSD** describing the data contained in the physical XML.
- **XML** is often used to describe structured data and to serialize objects. Various XML-based protocols exist to represent the same kind of data structures as JSON for the same kind of data interchange purposes.

Questions?

SoftServe
Ability. Agility. Advantage.

Contacts

Europe Headquarters

52 V. Velykoho Str.
Lviv 79053, Ukraine

Tel: +380-32-240-9090
Fax: +380-32-240-9080

E-mail: info@softservecom.com

US Headquarters

13350 Metro Parkway, Suite 302
Fort Myers, FL 33966, USA

Tel: 239-690-3111
Fax: 239-690-3116

E-mail: info@softservecom.com

www.softservecom.com

Thank You!