XML Introduction

INFO-3138

What you should know…

- Know when and how to use XML documents
- How to define and use XML elements and attributes, including the pros and cons of using one or the other
- The components of an *XML Declaration*
- Comments, entities, CData sections, processing instructions
- Know how to define and interpret whether a document is *well-formed* or *valid*

XML

- E**X**tensible **M**arkup **L**anguage
- XML is a modern approach to a classic problem:

How to represent and share structured data between different platforms, protocols, and systems.

- Similar standards include:
  - Electronic Data Interchange (EDI) has been around since the 70's (see EDIFACT)
  - HL7 for health care (V2 is delimited,V3 is XML)
  - ASN.1, S-expressions, JSON, BSON, CSV,YAML

XML

- Prevalence of the web helped give rise to XML
  - A universal way to pass data between heterogeneous platforms (such as between a linux-based server and an ipad)
  - Includes a *very* powerful suite of standards for definition, validation, querying, and transformation
  - Is human readable
  - Works well with object-oriented principles (inheritance, composition, etc.)
- Other technologies have emerged, yet XML is still heavily used in web apps (https://www.w3.org/blog/2018/07/the-world-wide-success-that-is-xml/)
- Is the basis of other web-related languages such as XHTML, WSDL and RSS

XML Language Features

- A simple markup language
  - Uses familiar tag based representation
- Expresses data
  - Can be subjected to schemas that define data structure
  - Able to be “sent over the wire” (streamed/serialized)
    - NOTE: A common use of XML is to stream objects across a network
  - Used to store data

XML Language Features

- Doesn’t have any verbs
  - Declarative: describes/declares data, *not* task-oriented
- Is extensible
  - Has a very flexible grammar so its usefulness isn’t limited
- Is universal
  - Document structure is governed by strict rules
  - Not associated with any specific platform
  - Is a W3C specification

XML vs. HTML

- Both based on SGML
  - Standard Generalized Markup Language
- HTML
  - Specialized markup for describing the appearance/layout of a document
  - Focuses on how the data looks
  - Includes a finite set of predefined tags
- XML
  - For representing the data itself
  - Focuses on data content and structure
  - Very few predefined tags: you create your own!

The same data – different purposes

```
<h1>Users currently logged in</h1>
<ul>
       <li>Greta Johansen</li>
       <li>Bobbert Ballard</li>
       <li>Hazel Wassername</li>
</ul>
```

HTML

```
<loggedIn>
       <user id=“123” name=“Greta Johansen” />
       <user id=“124” name=“Bobbert Ballard” />
       <user id=“125” name=“Hazel Wassername” />
</loggedIn>
```

XML

Applications of XML

Application	Examples	Alternative technologies
Representing sets of data to be shared between dissimilar systems	A serialization of a database table (E.g., list of LTC bus stops)	- **CSV** - JSON
Remote Procedure Calls (Even between different languages!)	A single-page web application contacts a server to retrieve data to display	- JSON
Representation of Documents	Configuration files, conceptual representation of real documents (E.g., invoices, transcript, etc.), Web services	- JSON

Representing Sets of Data

- E.g., a list of bus stops, a list of customer contact details, a list of phone calls to an extension

- As you will see in this course, XML has the advantage of being very flexible:
  - Extensive querying abilities (XPath and XQuery)
  - Formatting standards (CSS, XSL, XSLT)
  - Established software tools

- However, XML is very bulky in this area compared to a standard like CSV

Remote Procedure Calls

- XML allows programs from differing platforms and languages to interoperate (see XML-RPC)

- For example, a Java server can expose `add(a, b)` and a PHP app can invoke that method with the following XML:

```
<add>
        <a>5</a>
        <b>2</b>
</add>
```

- However there are other protocols for RPCs in web applications that are more concise and arguably better suited such (e.g. JSON)

Representation of Documents

- This is an area where XML really shines

- Can be used to represent:
  - Digital versions of physical documents (e.g., an invoice)
  - A means to encode documents (e.g., Word .docx)
  - A conceptual model (E.g., BPMN models a business process and is represented as an XML document)

- Able to leverage schema languages (DTD and XSD), query languages (XPath and XQuery), transformational languages (XSLT)

XML Document Structure and Syntax

- Part II

XML Versions

- Version 1.0 – published in 1998
- Version 1.1 – 2nd edition published in 2006
- Primary difference is version 1.1 allows more flexibility in identifiers (e.g. for elements and attributes)
- We will focus on XML version 1.0 syntax
  - Version 1.0 is much more prevalent
  - A valid XML 1.0 document is also a valid XML 1.1 document as long as it doesn't contain control characters in the range [#x7F-#x9F] other than as escape characters

XML Representation

- Often is either contained in a file or transferred over a network

- XML files
  - UNICODE-based text file format
  - Usually uses the .XML file extension
    - Certain XML documents have specialized extensions: E.g., .WSDL, .BPMN, .JRXML
  - Can edit with text file editor like Notepad
  - Specialized editors exist that simplify document creation (E.g., XML Spy, XML notepad,Visual Studio, Eclipse)

Example XML Document

```
<?xml version=“1.0” encoding=“utf-8” standalone=“yes”?>
<books>
     <book id=“123”>
          <title>Design Patterns</title>
          <authors>
               <author>
                    <name>Erich Gamma</name>
               </author>
          </authors>
     </book>
     <book id=“124”>
          <title>Extensible Markup Language (XML) 1.0</title>
          <authors/>
     </book>
     <!-- Continued on the next slide-->
```

Declaration

Element

Attribute

Closed Element

Comment

Example XML Document

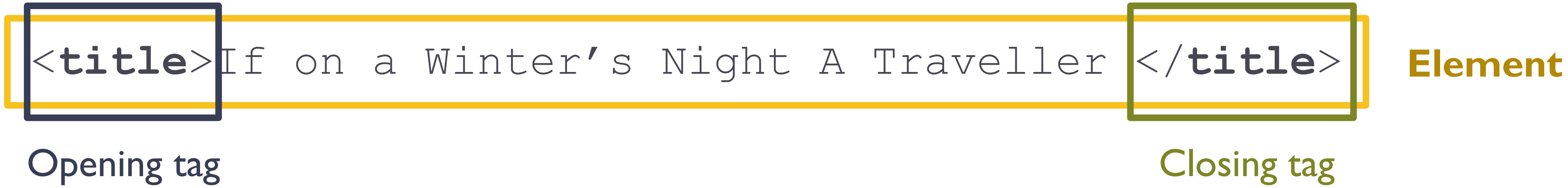
Unparsed Character Data/CDATA

```
    <book id=“125”>
        <title><![CDATA[
            A Book Title with < and > symbols.
        ]]></title>
        <authors>
            <author>
                <name>Bob &amp; Nancy</name>
            </author>
        </authors>
    </book>
</books>
```

Entity

Elements

Definition An XML element is everything from (including) the element's start tag to (including) the element's end tag.



`<title>If on a Winter’s Night A Traveller</title>`

- In this example:
  - The tag or element’s name is **`title`**
  - Its *value* is “`If on a Winter’s Night a Traveller`”

http://www.w3schools.com/xml/xml_elements.asp

Elements

- Element names:
  - Are CaSe SeNsItIvE (unlike HTML, like XHTML)
  - May not contain whitespace
  - Must begin with a letter or underscore (but not "xml")
  - Can contain other characters, but…
    - Colons ":" are used to indicate a namespace. More on that later…

- An XML document ***must*** have ***exactly one*** parent element which contains all other elements.

Elements

- An element can contain:
  - Other elements (I.e., ***nestec***
  - Text (also called **simple** co
  - Attributes
  - Or a mix of all of the above

Prof's Opinion

`book` has both text and element children. This is technically OK, but should really be avoided. A common exception is when the child elements add to the interpretation of the text. For example:

```
<p>Although 'p' has both text and
element <i>content</i>, it should be
<a>clear</a> that all content within
'p' are the same string, even if
<blink>there</blink> are other
elements.</p>
```

```
<book>
     Oliver Twist
     <author>Charles Dickens</author>
</book>
```

- *Indenting is purely for human readability!*
  - However, white space is preserved

Closed Elements

- Closed elements are elements that contain no explicit content
- Are sometimes used to conform with the structure of the data when the actual values are not known or null
- Two syntaxes – both are equivalent:
  - `<elementName></elementName>`
  - `<elementName/>`
- For example:

```
<book id=“124”>
        <title>Extensible Markup Language (XML) 1.0</title>
        <authors/>
</book>
```

Attributes

Definition Attributes appear within the opening tag of an element to provide additional information. They often provide information that is not a part of the data, but may be used in its interpretation (i.e., meta-data).

```
<user id="123"></user>
```

- In the example:
  - `user` is the element's name
  - `id` is an attribute name
  - `123` is the value assigned to the attribute

http://www.w3schools.com/xml/xml_attributes.asp

Attributes

- An element may have any number of attributes
- Attribute names follow the same naming rules as elements
- Attributes must always appear inside “” or ‘’ regardless of data type
  - If the attribute value contains quotes of one type:
    1. Contain the value using the other type
    2. Use an *entity* (coming by the end of these notes!)
- Attributes only support text/simple content (no nesting)

Using Attributes

- Elements often thought of as nouns and attributes as adjectives
- Or, directives for interpreting an element. For example:

```
<hello>
        <value lang=“en”>Hello</value>
        <value lang=“fr”>Bonjour</value>
        <value lang=“it”>Buongiorno</value>
</hello>
```

- Their use is a personal preference
  - Some people object to the use of attributes at all
  - Microsoft likes them

Using Attributes

- Consider the example on the previous page without attributes

```
<hellos>
        <hello>
                <lang>en</lang>
                <value>Hello</value>
        </hello>
        <hello>
                <lang>fr</lang>
                <value>Bonjour</value>
        </hello>
</hellos>
```

- Overcomplicated documents like this fuel a lot of today’s XML haters

In Favour of Attributes

- Attributes can provide metadata that may not be relevant to most applications dealing with our XML
  - Such apps can simply ignore the attributes
- Attributes are easier to use (no nesting, no closing tags)
- Attributes look better

Against Attributes

- Elements can do everything attributes can
- Elements can be extended to elaborate further when necessary (can't extend an attribute)
- Elements look better

XML Declaration

- Identifies your document as XML
- Basic syntax:
  - `<?xml version="1.0" ?>`
- Not required, but should be included
  - Many tools and parsers "freak out" if this isn't there
- Must appear at the *top* of an XML document
  - That is, the **very first line** (not even preceded by a comment!)

XML Declaration Attributes

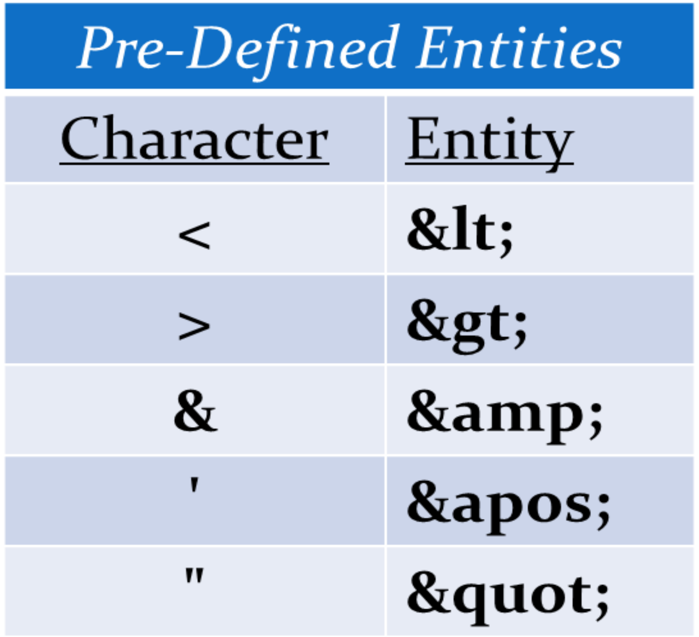
- The XML declaration is not considered an element, but uses attributes
- **`<?xml version="1.0" encoding=“utf-8” standalone=“yes”?>`**
  - Without a **`version`** attribute an XML parser will assume the document uses latest version of XML
  - The **`encoding`** attribute identifies the format used to encode the UNICODE characters for transmission
  - The **`standalone`** attribute identifies whether the document depends on any externally-defined constraints

XML Comments

- Same syntax as in HTML
  - `<!-- This is a comment -->`

Entities

- A form of markup
- Usually for characters that can’t be rendered literally as data
- Always begin with **&**
- Always end with **;**
- Some are *pre-defined* in XML
- Can also create user-defined entities



Pre-Defined Entities

Character	Entity
<	**&lt;**
>	**&gt;**
&	**&amp;**
'	**&apos;**
"	**&quot;**

CData Sections

- **CData** = Character data
- **CData** sections are bypassed by the parser
- This allows us to pass "non-valid" XML in an element

```
<code>
        <![CDATA[  if(6 > 7) { return 6 < 5; }  ]]>
</code>
```

XML Document Validation

There are two levels of validation:

1. **Well-formed document**
   - Uses only correct XML syntax/structure

2. **Valid document**
   - Is well-formed and…
   - Conforms to a set of constraints that are associated with the document via either
     - □ A DTD (Document Type Definition), or…
     - □ An XML Schema

A Well-Formed XML Document

Definition

- □ A document is said to be well-formed if it:
  - □ Has exactly one root element
  - □ Has element and attribute names that follow the naming rules
  - □ Includes a start tag and an end tag for each element
  - □ Includes only properly nested elements/tags
    - i.e., Consistent order for opening/closing order for tags. The following would be wrong:

      `<a><b> Hello world </a></b>`
  - □ Includes only attributes with values delimited with quotes (both single or both double)

Validating with an XML Parser

- The data in an XML document exists to be used or *consumed* in some manner
- The XML code is parsed to tokenize the data which facilitates processing
- The W3C XML specification says that an XML parser must stop parsing as soon as it encounters an error
- For example, IE has a built-in parser which will fail to display the document if it isn’t well-formed

Processing Instructions

```
<?target … instruction … ?>
```

- Built-in processing instructions are used in some contexts such as XML Stylesheets

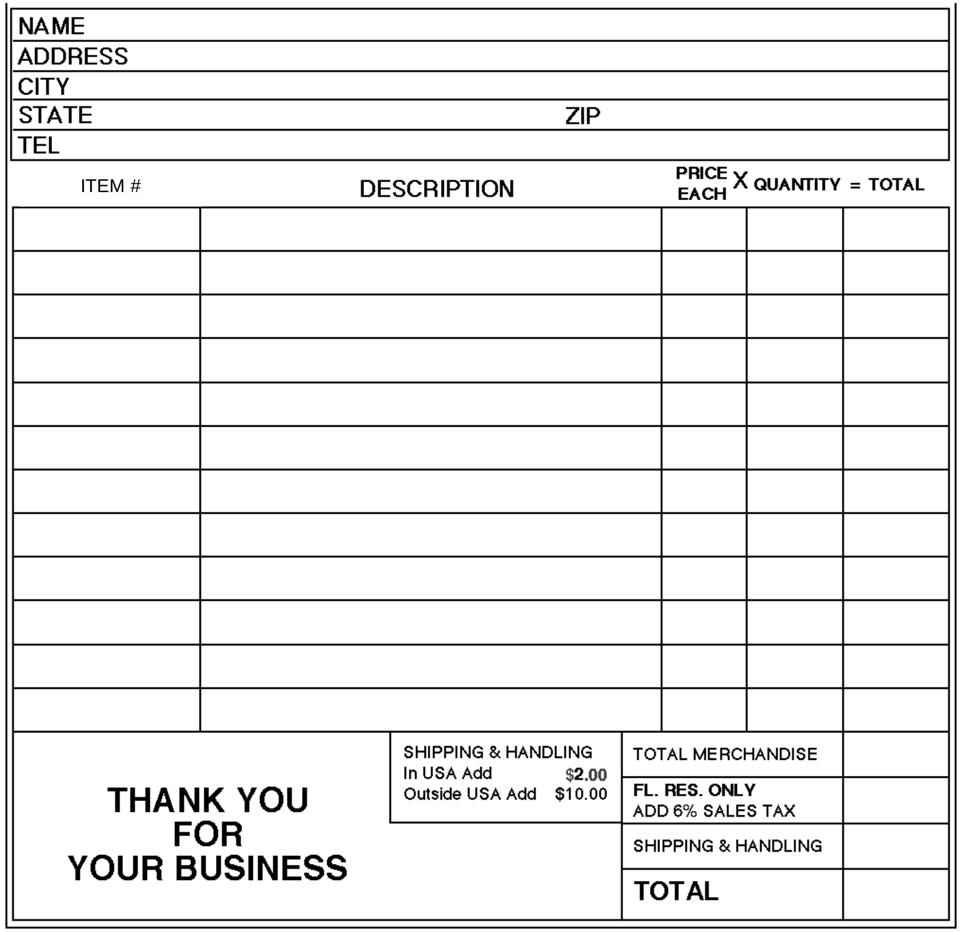
```
<?xml-stylesheet href=“mystyle.css” type=“text/css” ?>
```

- Application-specific processing instructions can be added to any XML document
  - “Hints” to application
  - Rarely used
  - *NOT RECOMMMENDED*

```
<?CDApplication MessageBox(“Missing data.”)?>
```

Let’s work through a problem…

- Convert this document to XML inserting some “dummy” data



NAME
ADDRESS
CITY
STATE ZIP
TEL

ITEM #	DESCRIPTION	PRICE EACH	X QUANTITY	= TOTAL

THANK YOU
FOR
YOUR BUSINESS

SHIPPING & HANDLING
In USA Add $2.00
Outside USA Add $10.00

TOTAL MERCHANDISE	
FL. RES. ONLY ADD 6% SALES TAX	
SHIPPING & HANDLING	
TOTAL	

XPath

INFO-3138

What is XPath?

- XPath, a W3C recommendation, is a language for addressing parts of an XML document, designed to be used by both XSLT and XPointer
  - XPath 1.0 created (with XSL/XSLT) in 1999
  - XPath 2.0 created in 2006
  - XPath 3.0 created in 2015
  - XPath 3.1 created in 2017

- **IMPORTANT:**
  - Microsoft .NET’s *System.Xml.XPath* API supports XPath 2.0

What’s XPath for?

- A tool that can be used when processing XML data to specify items or subsets of the document
- Leveraged by other XML-related technologies such as *XSLT* and *XQuery*

The General Idea

- Retrieves raw or aggregate data from an XML source
  - Raw data includes sets of elements, attributes, text nodes, etc. that match specific search criteria
    - Result is generally described as a “node-set”
    - If nothing matches then nothing (an empty node-set) gets returned
  - Aggregate data can be totals, averages, frequencies, etc.

- Treats XML documents hierarchically
  - i.e. parents, children, grandchildren, etc.
  - Must navigate the hierarchy to address nodes
  - Uses notation that is like a file’s pathname:
    - `/library/book/title[2]/text()`
    - Refers to the text of the second title node that is within the book node that is within the library node

Predicates

- Predicates are optional conditions
  - like a WHERE clause in SQL
- In the example
  - **`/library/book/title[2]/text()`**
- there is a predicate (a short form, actually)
- **`title[2]`** indicates the second title node (if there is one)
- The full syntax for this would be:
  - **`title[position() = 2]`**
- For each node, the full syntax could then be:
  - **`axis :: node-test [predicate]`**
  - **`E.g. descendant::food[fiber>3]`**

Compound paths

- Up until now, you have been able to use only a single path
- You can also combine paths using the | operator (the “pipe” or “or”)
- For example:
  - “**`name/firstname | day/month`**”
  - refers to *either one* of these paths

Quick Exercise 1

- Using *Nutrition.xml*, create a number of XPath expressions that show:
  - The *name* of each *food* item
  - The *name* of the second *food* item
  - The number of grams of *fiber* required daily

XPath “axis”

- You can specify a particular part or “type” of a document by using its “axis”:
  - Like element, attribute, etc.
- This limits the scope of your search to a particular type of thing or part of the document
- Syntax is
  - **`axis::`**

Some axis examples

- **`ancestor::`**
  - Holds the ancestor of a particular node
- **`ancestor-or-self::`**
  - The current node and the ancestor nodes
- **`attribute::`**
  - An attribute (can also use “`@`” symbol)
- **`child::`**
  - The child of a node
  - This is the default axes

More axis examples

- **`descendant::`**
  - a child, or further (a child of a child, for instance)
  - can use “`//`” as a short form
- **`descendant-or-self::`**
  - The current node and the descendant nodes
- **`namespace::`**
  - The namespace of the current node

Some node "tests"

- In the example: `/library/book/title[2]/text()`
  there is a "test" at the end that looks for the text of a particular node
- Here are some other node tests:

Test	Matches
nodename	A specific element
*	Any element
@attributename	A specific attribute
@*	Any attribute
node()	Any node of any kind
comment()	Any comment node
processing-instruction()	Any nodes with <? like <?DOCTYPE…
text()	Only the text inside a node

Some examples

- Let’s look at some examples of axis

Operators and functions

- XPath also has a number of functions and operators build in:
  - >, <, <=, >=, =, != (note: “=” is “equals”)
  - and, or
  - +, -, *, div, mod
  - boolean(), false(), not(), true()
  - ceiling(), floor(), number()
  - round(), sum(), count()
  - id(), last(), local-name(), name(), position()
  - contains(), starts-with()

Operators and functions

- Operations
  - +, -, *
  - **`div`** is divide
  - **`mod`** is modulus

Operators and functions

- **`true()`** and **`false()`**
  - Simply return a true or a false. That's it
- **`not()`** negates a boolean operator
- **`boolean()`**
  - converts something to a true or false

Operators and functions

- **`ceiling():`**
  - Returns the smallest int that is larger than the number you passed (like rounding up)
- **`floor():`**
  - Returns the largest int that is smaller than the number you passed (like rounding down)
- **`number():`**
  - Converts text (like in an attribute) into a number

Operators and functions

- `round()`
  - Rounds numbers (the “usual” way)
- `sum()`
  - Returns the sum (all added together)
- `count()`
  - Returns the number (the “count”)

Operators and functions

- `id()`
  - Matches based on the node (or attribute) id
- `last()`
  - The ‘last’ node in a collection
- `local-name()`
  - Finds things based on the name of the node
- `name()`
  - like name, but returns the fully qualified name
- `position()`
  - Finds things based on the position in the node list

Operators and functions

- **`starts-with()`**
  - Returns true if the substring starts with something
  - //name[starts-with(@type,“C")]
- **`last(), first()`**
  - Used with position to get last and first elements
- **`contains()`**
  - Used with strings, looks for strings within strings
  - //name[contains(@type,“Chocolate")]

Practice Exercises

- Using the Nutrition.xml, create XPath expressions to return the following:
  1. The name of the food in each food element
  2. The grams of protein for each food element
  3. The serving size for each food for which the serving size is given in grams (serving units="g")
  4. The grams of fiber for the first food element
  5. The name of each food that is manufactured (mfr) by "Lees“

Practice Exercises - continued

6. The NUMBER of food items that contain more than 1 g of fiber
7. The NUMBER of food items that contain 5% or less of the daily amount of total-fat
8. The TOTAL number of calories for all the food items
9. The AVERAGE number of calories per serving of ALL the food items

XSD

INFO-3138

Definition of “Schema”

- Generally defined as:

  The organization or structure of a database, usually derived from data modeling.

- Typically described using a controlled vocabulary that names:
  - Items of data (and their types)
  - Constraints
  - Relationship between data items

Schemas in XML

- Similar to idea of database schemas
- Can be used to impose rules about the structure and content of a set of XML “instance” documents
- An XML document can be validated against a schema to determine if it conforms with the schema’s rules

Types of Schemas for XML

- There are actually several types of schema “dialects”. Here are a few:
  - XML Data Reduced (a.k.a. XDR) – created by Microsoft; first to be introduced and now pretty obscure
  - RELAX NG – Japanese national schema standard
  - **XML Schema** (a.k.a. **XSD**) – developed by the W3C and has more or less become a standard
- The rest of these notes will refer to **XSD**

Schema vs. Instance

- A schema is really a blueprint for a set of XML documents
- This is similar to how a class is a blueprint for a set of objects in OOP
- The XML documents that a schema applies to are called “instance” documents

Basic Schema Principals

- Two distinct parts of markup in XML schema that loosely correspond to these
  1. Definitions: create new types (both simple and complex)
  2. Declarations: describe the content models of elements and attributes in document instances

- In other words, we create a schema by:
  1. creating types as “building blocks”
  2. use the types to describe the XML “instances” of the schema

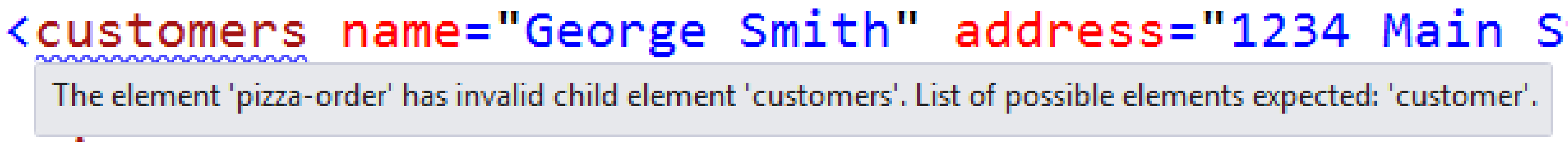
A Basic Schema Document

- An XML Schema consists of a schema element which declares the *XMLSchema* namespace and which contains zero or more declarations.
- Note you don’t have to use “xsd” as the namespace prefix (shown). In fact, by default Visual Studio will use “xs” as the prefix.
- Note that the version attribute shown is optional and specifies the version of the schema.

```
<xsd:schema
  xmlns:xsd=“http://www.w3.org/2001/XMLSchema” version=“1.42.57”>
    <!– Declarations go here →
</xsd:schema>
```

XSDs in Visual Studio (VS)

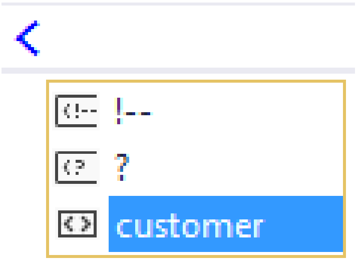
- VS is a nice tool for creating and testing XSDs
- VS has a document template for creating an XSD
- VS will validate your XML document against any associated schemas



```
<customers name="George Smith" address="1234 Main St"
```

The element 'pizza-order' has invalid child element 'customers'. List of possible elements expected: 'customer'.

- VS's *intellisense* will also guide you in selecting:
  - valid XSD syntaxes in an XSD document
  - schema-valid XML in an XML instance document



Associating with a Document

- Two ways:
  - via the parser (software)
  - via either the *schemaLocation* or *noNamespaceSchemaLocation* attributes within the XML instance document

- Because of namespaces a single XML document can be associated with many schemas

Associating an XSD with a Document

- If the schema (XSD) does NOT use a namespace:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<rootElement
   xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="schemaDoc.xsd">
```

- If the schema (XSD) DOES use a namespace:

```
<?xml version="1.0" encoding="utf-8" standalone="no"?>
<rootElement
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:schemaLocation="xsd_namespace schemaDoc.xsd">
```

Primary Components

- There are four primary Schema components:
  - Element *declarations*
  - Simple Type *definitions*
  - Complex Type *definitions*
  - Attribute *declarations*

Basic Element Declaration

<xsd:element name="*name*" type="*type*" minOccurs="*int*" maxOccurs="*int*"/>

- Where:
  - *name* is a user defined name
  - *type* can be a simple type (e.g., xsd:string) or the name of a complex type
  - *minOccurs* can be a non-negative integer
  - *maxOccurs* can be a non-negative integer or unbounded

Element Attributes

- <element>
  - Attributes:
    - name
    - ref
    - type
    - minOccurs
    - maxOccurs
    - default
    - fixed
    - id

Basic Element Declaration Examples

```
<xsd:element name=“MyString” />

<xsd:element name=“MyDate” type=“xsd:date” />

<xsd:element name=“MyTime” type=“xsd:time”  minOccurs=“0”
        maxOccurs=“1”/>

<xsd:element name=“MyYear” type=“xsd:gYear”  minOccurs=“1
        maxOccurs=“unbounded”/>
```

Data Types

- Schemas allow us to strongly type the content of our XML data
- Two varieties of Built-in types:
  - Primitive Types
  - Derived Types
    - Built from other data types
    - Created by restricting existing data-types
    - derived from a *base* type
- We can also build our own (user-derived) types

Built-in Primitive Data Types

- Some Built-in examples:

string	“Barrack Obama”
boolean	true, false, 1, 0
float	single precision (32 bit) floating point
double	double precision (64 bit) floating point
decimal	arbitrary precision decimal numbers
time	a time in format HH:MM:SS
date	calendar date in format CCYY-MM-DD
Notation	represents NOTATION from XML 1.0

Derived Data Types

- Can contain any well-formed XML that is valid according to their definition
- May be built-in or user derived

Built-in Derived Data Types

- Some Built-in examples:

integer	base type is number
int	base type is integer (wrv*)
unsignedLong	base type is nonNegativeInteger (wrv*)
byte	base type is short
positiveInteger	base type is nonNegativeInteger (wrv*)
NMTOKEN	base type is token

*wrv = with restricted values

- For the complete list of built-in primitive and derived types, visit:
  - https://www.w3.org/TR/2004/REC-xmlschema-2-20041028/#built-in-datatypes

User Derived Data Types

- XML Schema offers us two options:
  - Simple types
    - <xsd:simpleType …>
    - Use the simpleType element when you want to create a new type that is a refinement of a built-in type (string, date, gYear, etc)
  - Complex types
    - <xsd:complexType …>
    - Use the complexType element when you want to define child elements and/or attributes of an element

User Derived Data Types - Example

-
  ```
  <xsd:simpleType name =“myNegativeInteger”>
     <xsd:restriction base=“xsd:integer”>
      <xsd:minInclusive value=“-1290” />
       <xsd:maxInclusive value=“-1” />
     </xsd:restriction>
   </xsd:simpleType>
  ```

- Don’t worry too much about the specifics of the code, we will revisit <simpleType> elements

Atomic and List Data Types

- Atomic: types that have *values* that are defined to be indivisible
- List: a *value* that’s comprised of a finite-length sequence of atomic values
  - Always derived type
  - Must be delimited by white space character(s)
  - Thus individual values in the list cannot contain spaces

Aspects of Data Types

- All XML Schema datatypes are comprised of 3 parts
  - A value space
    - The set of distinct values where each of the values is represented by one or more literals in the lexical space
  - A lexical space
    - A set of valid string literals *representing* values
  - A set of facets
    - Properties of the value space, its individual values

Facets

- The defining properties of a datatype which distinguishes that datatype from others
- Fundamental Facets:
  - Equality
  - Order
  - Bounds
  - Cardinality
  - Numeric/Non-numeric

Constraining Facets

- Used to constrain the permitted value of a datatype:
  - length, minLength, maxLength
  - pattern
  - enumeration
  - minExclusive, maxExclusive, minInclusive, maxInclusive
  - precision, scale
  - encoding
  - duration, period (applies only to recurringDuration datatype)

Length, MinLength, MaxLength Facets

- For strings, the number of chars
- For binary, the number of bytes
- For list types, the number of elements in the list
- Cannot be used on numeric datatypes

Pattern Facet

- A constraint on the lexical representation of the datatype
- The “mask” is specified using a regular expression “regexp” (similar to that defined in Perl)

Enumeration and Whitespace Facets

- enumeration
  - Like the enumerations in any programming language
    - Limits a datatype to a list of specific values
  - Useful for all simple datatypes except boolean
- whitespace
  - Indicates whether whitespace is allowed in the value space
  - preserve, replace or collapse

Minimum and Maximum Value Facets

- minExclusive, maxExclusive, minInclusive, maxInclusive
  - The mins define the lower bound of the value space
  - The maxs define the upper bound
  - Exclusive means < or >
  - Inclusive means <= or =>

Precision, Scale and Encoding Facets

- precision, scale
  - Applies to all decimal datatypes
  - Precision is the maximum number of digits
  - Scale is the maximum number of digits in the fractional portion
- Encoding
  - Applies to the lexical space of binary datatypes
  - Either hex or base64

Multiple Facets: *and* them or *or* them?

```
<xsd:simpleType name="TelephoneNumber">
   <xsd:restriction base="xsd:string">
      <xsd:length value="8"/>
      <xsd:pattern value="\d{3}-\d{4}"/>
   </xsd:restriction>
</xsd:simpleType>
```

An element declared to be of type *TelephoneNumber* must be a string of *length=8 and* the string must follow the pattern: 3 digits, dash, 4 digits.

```
<xsd:simpleType name="shape">
   <xsd:restriction base="xsd:string">
     <xsd:enumeration value="circle"/>
     <xsd:enumeration value="triangle"/>
     <xsd:enumeration value="square"/>
   </xsd:restriction>
</xsd:simpleType>
```

An element declared to be of type shape must be a string with a value of *either* circle, *or* triangle, *or* square.

- For patterns and enumerations you "or" them together
- For all other facets you "and" them together

Data Type Definitions

- Two techniques:
  - Simple:
    - To create “simple” user-derived data types
    - These are types that hold data, rather than nested content
    - Use *simpleType* to define
  - Complex:
    - Primarily used to describe content models
    - For elements that have child elements and/or attributes
    - Can use to disallow content as well (for closed elements)
    - Use *complexType* to define

Simple Datatype Definitions

- XML Representation Summary:

```
<simpleType
   final = (#all | (list | restriction | union)
   id = ID
   name = NCName >

   Content: (annotation?, (restriction | list | union))

</simpleType>
```

Simple Datatype Definitions (2)

- restriction element
  - *base* attribute
  - Any facet constraint elements

- list element
  - *itemType* attribute

- List length can be set through a derived type

Simple Datatype Definitions (3)

- Derivation by Restriction
  - Examples: negativeInteger, AreaCode

- Derivation by List
  - Example: ListOfFloats

Simple Types: Named

- A “Named” Simple Type

```
<xsd:element name=“aName" type=“aType" minOccurs=“0" maxOccurs=“1"/>

<xsd:simpleType name=“aType”>
        <xsd:restriction base="type">
        …
        </xsd:restriction>
</xsd:simpleType>
```

Simple Types: Anonymous

- An “Anonymous” Simple Type

```
<xsd:element name=“aName" minOccurs="int" maxOccurs="int">
    <xsd:simpleType>
        <xsd:restriction base="type">
        …
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Defining Complex Types

- XML Representation Summary:

```
<xsd:complexType
    abstract = boolean : false
    block = (#all | (list of (extension | restriction ))
    final = (#all | (list of (extension | restriction ))
    id = ID
    mixed = boolean : false
    name = NCName >
    Content: ( annotation?, (simpleContent | complexContent |
                ((group | all | choice | sequence)?,
                ((attribute | attributeGroup)*, anyAttribute?))) )>
</xsd:complexType>
```

Complex Type Definitions

- A set of attribute declarations and a content type that respectively pertain to the attributes and children of the element type that is being specified
- Provide a mechanism to validate a document instance containing the type
- Describe element attribute existence and content
- Describe the content of an element type
- Derive its definition from another simple or complex type
- Control the ability to derive additional types

Content Models

- The formal description of the structure and permissible content of an element

Content Models: An Element’s Attributes

- <element>
  - Attributes:
    - name
    - ref
    - type
    - minOccurs
    - maxOccurs
    - default
    - fixed
    - id

Content Models: An Element’s Child Elements

- Element Content Models
  - **<choice>**
    - One or more child elements selected from a list
    - Has no imposed sequence
  - **<sequence>**
    - A specified sequence of child elements
  - **<all>**
    - All elements in a list must appear as child elements
    - Has no imposed sequence
    - No node can appear more than once or be a group element
    - Must appear as sole child at the top of the content model

Time for some examples

- Let’s build some complexTypes…

Complex Types: Named

- A “Named” Complex Type

```
<xsd:element name=“aName" type=“aType" minOccurs=“0" maxOccurs=“1"/>


<xsd:complexType name=“aType”>
   …
</xsd:conplexType>
```

Complex Types: Anonymous

- An “Anonymous” Complex Type

```
<xsd:element name=“aName" minOccurs="int" maxOccurs="int">
    <xsd:complexeType>
     …
    </xsd:complexType>
</xsd:element>
```

Summary: Ways to Declare Elements

1.
   ```
   <xsd:element name="name" type="type" minOccurs="int" maxOccurs="int"/>
   ```

2.
   ```
   <xsd:element name="name" minOccurs="int" maxOccurs="int">
     <xsd:complexType>
         …
      </xsd:complexType>
   </xsd:element>
   ```

3.
   ```
   <xsd:element name="name" minOccurs="int" maxOccurs="int">
      <xsd:simpleType>
         <xsd:restriction base="type">
             …
         </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
   ```

Other Content Specifications

- Any
  - <xsd:element name="MyElement" type="xsd:anyType">
  - This is the default and imposes no restrictions
- Empty
  - No text or child elements, can have attributes
  - Eg. Restricts a complex type to only have attributes
- Element-only
  - Created by creating a complex type with child elements
- Mixed
  - Text and<xsd:complexType mixed="true"> …
  - elements, order of elements can be constrained!

Attribute Declaration

- Provide a description that can be used for validation
- Constrain attribute values to a specific simple datatype
- Require/prevent the appearance of an attribute
- Provide default (or fixed) attribute values

Attribute Declaration Syntax

- XML Representation Summary:

```
<attribute
  default = string
  fixed = string
  form = (qualified | unqualified)
  id = ID
  name = NCName
  ref = QName
  type = QName
  use = (optional | prohibited |        required) : optional >
  Content: (annotation?, (simpleType?))
</attribute>
```

Attribute Declaration Details

- May appear in two places in the schema:
  - As a globally scoped declaration within the <schema> element
  - Within a <complexType> declaration
    - Attribute declarations always come LAST, after element declarations
    - The attributes are always with respect to the element they are defined (nested) within
- Attribute types may be explicitly declared
- When declaring a global attribute do not specify a “use”

Summary: Declaring Attributes (2 ways)

1.
```
<xsd:attribute name="name" type="simple-type"
      use="how-its-used" default/fixed="value"/>
```

2.
```
<xsd:attribute name="name" use="how-its-used"
      default/fixed="value">
      <xsd:simpleType>
            <xsd:restriction base="simple-type">
                  <xsd:facet value="value"/>
                  …
            </xsd:restriction>
      </xsd:simpleType>
</xsd:attribute>
```

Referencing “Globals”

- “Global” scope refers to Elements and Attributes that are *declared* directly under schema
- You can re-use the definitions for Global Elements and Attributes by referencing them via the “ref” attribute
- Example:

```
<xsd:element name=“length” type=“xsd:integer”/>
…
<xsd:complexType name=“dimensions”>
  <xsd:sequence>
    <xsd:element ref=“length” minOccurs=“1”>
…
```

Annotations

- Allows our schema to be self-describing
- <annotation>

  <appInfo>: information applicable to applications

  <documentation>: the place for comments about the schema
- Rules for annotations:
  - they may occur before and after any global component
  - they may occur only at the beginning of non-global components

Namespaces in Validation

- When we describe our schemas only the direct child nodes of the “schema” are impacted by the “targetNamespace”
  - This applies to both elements & attributes (but we’ll use globally scoped attributes)
  - So we’ll add the following attribute to our schema preamble

    elementFormDefault="qualified“