Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

XML/JSON Fundamentals

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka
A distributed streaming platform

Apache Software Foundation

APACHE Spark

Java

hadoop

- XML basics
- Namespaces and Schema
- XSL
- JSON basics
- XML Vs JSON

XML basics

What is XML

- XML stands for eXtensible Markup Language.
- A markup language is used to provide information about a document.
- Tags are added to the document to provide the extra information.
- HTML tags tell a browser how to display the document.
- XML tags give a reader some idea what some of the data means.

What is XML Used For?

- XML documents are used to transfer data from one place to another often over the Internet.
- XML subsets are designed for particular applications.
- One is RSS (Rich Site Summary or Really Simple Syndication). It is used to send breaking news bulletins from one web site to another.
- A number of fields have their own subsets. These include chemistry, mathematics, and books publishing.
- Most of these subsets are registered with the W3Consortium and are available for anyone's use.

Advantages of XML

- XML is text (Unicode) based.
  - Takes up less space.
  - Can be transmitted efficiently.
- One XML document can be displayed differently in different media.
  - Html, video, CD, DVD,
  - You only have to change the XML document in order to change all the rest.
- XML documents can be modularized. Parts can be reused.

Example of an HTML Document

```
<html>
  <head><title>Example</title></head.
<body>
  <h1>This is an example of a page.</h1>
  <h2>Some information goes here.</h2>
</body>
</html>
```

Example of an XML Document

```
<?xml version="1.0"/>
<address>
  <name>Alice Lee</name>
  <email>alee@aol.com</email>
  <phone>212-346-1234</phone>
  <birthday>1985-03-22</birthday>
</address>
```

Difference Between HTML and XML

- HTML tags have a fixed meaning and browsers know what it is.
- XML tags are different for different applications, and users know what they mean.
- HTML tags are used for display.
- XML tags are used to describe documents and data.

Rules For Well-Formed XML

- There must be one, and only one, root element
- Sub-elements must be properly nested
  - A tag must end within the tag in which it was started
- Attributes are optional
  - Defined by an optional schema
- Attribute values must be enclosed in “” or ‘ ’
- Processing instructions are optional
- XML is case-sensitive
  - <tag> and <TAG> are not the same type of element

Well-Formed XML?

- No, CHILD2 and CHILD3 do not nest properly

```
<xml? version="1.0" ?>
<PARENT>
      <CHILD1>This is element 1</CHILD1>
      <CHILD2><CHILD3>Number 3</CHILD2></CHILD3>
</PARENT>
```

Well-Formed XML?

- No, there are two root elements

```
<xml? Version=“1.0” ?>
<PARENT>
      <CHILD1>This is element 1</CHILD1>
</PARENT>
<PARENT>
      <CHILD1>This is another element 1</CHILD1>
</PARENT>
```

Well-Formed XML?

- Yes

```
<xml? version="1.0" ?>
<PARENT>
      <CHILD1>This is element 1</CHILD1>
      <CHILD2/>
      <CHILD3></CHILD3>
</PARENT>
```

An XML Document

```
<?xml version='1.0'?>
<bookstore>
  <book genre='autobiography' publicationdate='1981'
        ISBN='1-861003-11-0'>
    <title>The Autobiography of Benjamin Franklin</title>
    <author>
      <first-name>Benjamin</first-name>
      <last-name>Franklin</last-name>
    </author>
    <price>8.99</price>
  </book>
  <book genre='novel' publicationdate='1967' ISBN='0-201-63361-2'>
    <title>The Confidence Man</title>
    <author>
      <first-name>Herman</first-name>
      <last-name>Melville</last-name>
    </author>
    <price>11.99</price>
  </book>
</bookstore>
```

Namespaces: Overview

- Part of XML's extensibility
- Allow authors to differentiate between tags of the same name (using a prefix)
  - Frees author to focus on the data and decide how to best describe it
  - Allows multiple XML documents from multiple authors to be merged
- Identified by a URI (Uniform Resource Identifier)
  - When a URL is used, it does NOT have to represent a live server

Namespaces: Declaration

- Namespace declaration examples:

```
xmlns: bk = “http://www.example.com/bookinfo/”
xmlns: bk = “urn:mybookstuff.org:bookinfo”
xmlns: bk = “http://www.example.com/bookinfo/”
```

Namespaces: Examples

```
<BOOK xmlns:bk="http://www.bookstuff.org/bookinfo">
    <bk:TITLE>All About XML</bk:TITLE>
    <bk:AUTHOR>Joe Developer</bk:AUTHOR>
    <bk:PRICE currency='US Dollar'>19.99</bk:PRICE>

<bk:BOOK xmlns:bk="http://www.bookstuff.org/bookinfo"
     xmlns:money="urn:finance:money">
    <bk:TITLE>All About XML</bk:TITLE>
    <bk:AUTHOR>Joe Developer</bk:AUTHOR>
    <bk:PRICE money:currency='US Dollar'>
        19.99</bk:PRICE>
```

Namespaces: Default Namespace

- An XML namespace declared without a prefix becomes the default namespace for all
  sub-elements
- All elements without a prefix will belong to the default namespace:

```
<BOOK xmlns="http://www.bookstuff.org/bookinfo">
    <TITLE>All About XML</TITLE>
    <AUTHOR>Joe Developer</AUTHOR>
```

Namespaces: Scope

- Unqualified elements belong to the inner-most default namespace.
  - BOOK, TITLE, and AUTHOR belong to the default book namespace
  - PUBLISHER and NAME belong to the default publisher namespace

```
<BOOK xmlns="www.bookstuff.org/bookinfo">
    <TITLE>All About XML</TITLE>
    <AUTHOR>Joe Developer</AUTHOR>
    <PUBLISHER xmlns="urn:publishers:publinfo">
        <NAME>Microsoft Press</NAME>
    </PUBLISHER>
</BOOK>
```

Namespaces: Attributes

- Unqualified attributes do NOT belong to any namespace
  - Even if there is a default namespace
- This differs from elements, which belong to the default namespace

Entities

- Entities provide a mechanism for textual substitution, e.g.

Entity	Substitution
&lt;	<
&amp;	&

- You can define your own entities
- Parsed entities can contain text and markup
- Unparsed entities can contain any data
  - JPEG photos, GIF files, movies, etc.

The XML 'Alphabet Soup'

- XML itself is fairly simple
- Most of the learning curve is knowing about all of the related technologies

The XML 'Alphabet Soup'

XML	Extensible Markup Language	Defines XML documents
Infoset	Information Set	Abstract model of XML data; definition of terms
DTD	Document Type Definition	Non-XML schema
XSD	XML Schema	XML-based schema language
XDR	XML Data Reduced	An earlier XML schema
CSS	Cascading Style Sheets	Allows you to specify styles
XSL	Extensible Stylesheet Language	Language for expressing stylesheets; consists of XSLT and XSL-FO
XSLT	XSL Transformations	Language for transforming XML documents
XSL-FO	XSL Formatting Objects	Language to describe precise layout of text on a page

XPath	XML Path Language	A language for addressing parts of an XML document, designed to be used by both XSLT and XPointer
XPointer	XML Pointer Language	Supports addressing into the internal structures of XML documents
XLink	XML Linking Language	Describes links between XML documents
XQuery	XML Query Language (draft)	Flexible mechanism for querying XML data as if it were a database
DOM	Document Object Model	API to read, create and edit XML documents; creates in-memory object model
SAX	Simple API for XML	API to parse XML documents; event-driven
Data Island	XML data embedded in a HTML page	
Data Binding	Automatic population of HTML elements from XML data	


SYNERGETICS
GET IT RIGHT
Microsoft Partner
Gold Cloud Platform
Silver Learning

XML Schema

- XML itself does not restrict what elements existing in a document.
- In a given application, you want to fix a vocabulary -- what elements make sense, what their types are, etc.
- Use a **Schema** to define an XML dialect
  - MusicXML, ChemXML, VoiceXML, ADXML, etc.
- Restrict documents to those tags.
- Schema can be used to validate a document -- ie to see if it obeys the rules of the dialect.

Schema determine …

- What sort of elements can appear in the document.
- What elements MUST appear
- Which elements can appear as part of another element
- What attributes can appear or must appear
- What kind of values can/must be in an attribute.

```
<?xml version="1.0" encoding="UTF-8"?>
<library>
   <book id="b0836217462" available="true">
      <isbn> 0836217462 </isbn>
      <title lang="en"> Being a Dog is a Full-Time Job </title>
      <author id="CMS">
         <name> Charles Schulz </name>
         <born> 1922-11-26 </born>
         <dead> 2000-02-12 </dead>
      </author>
      <character id="PP">
         <name> Peppermint Patty </name>
         <born> 1966-08-22 </born>
         <qualification> bold,brash, and tomboyish </qualification>
      </character>
      <character id="Snoopy">
         <name> Snoopy</name>
         <born>1950-10-04</born>
         <qualification>extroverted beagle</qualification>
      </character>
      <character id="Schroeder">
         <name>Schroeder</name>
         <born>1951-05-30</born>
         <qualification>brought classical music to the Peanuts Strip</qualification>
      </character>
      <character id="Lucy">
         <name>Lucy</name>
         <born>1952-03-03</born>
         <qualification>bossy, crabby, and selfish</qualification>
      </character>
   </book>
```

- We start with sample XML document and reverse engineer a schema as a simple example

First identify the elements:
author, book, born, character, dead, isbn, library, name, qualification, title

Next categorize by *content model*
Empty: contains nothing
Simple: only text nodes
Complex: only sub-elements
Mixed: text nodes + sub-elements

Note: content model independent of comments, attributes, or processing instructions!

Content models

- Simple content model: *name, born, title, dead, isbn, qualification*
- Complex content model: libarary, character, book, author

Content Types

- We further distinguish between complex and simple content **Types:**
  - Simple Type: An element with only text nodes and no child elements or attributes
  - Complex Type: All other cases

- We also say (and require) that all attributes themselves have simple type

Content Types

- Simple content type: *name, born, dead, isbn, qualification*
- Complex content type: library, character, book, author, title

Building the schema

- Schema are XML documents

- They must contain a schema root element as such

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.w3schools.com" xmlns="http://www.w3schools.com"
elementFormDefault="qualified">
... ...
</xs:schema>
```

Flat schema for library

Start by defining all of the simple types (including attributes):

```
<xs:schema xmlns:xs=http://www.w3.org/2001/XMLSchema>
  <xs:element name="name" type="xs:string"/>
  <xs:element name="qualification" type="xs:string"/>
  <xs:element name="born" type="xs:date"/>
  <xs:element name="dead" type="xs:date"/>
  <xs:element name="isbn" type="xs:string"/>
  <xs:attribute name="id" type="xs:ID"/>
  <xs:attribute name="available" type="xs:boolean"/>
  <xs:attribute name="lang" type="xs:language/>
   …/…
</xs:schema>
```

Complex types with simple content

Now to complex types:

```
<title lang="en">
  Being a Dog is  …
</title>

<xs:element name="title">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute ref="lang"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

"the element named *title* has a complex type which is a simple content obtained by extending the predefined datatype *xs:string* by adding the attribute defined in this schema and having the name *lang*."

Complex Types

All other types are complex types with complex content. For example:

```
<xs:element name="library">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="book" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>

<xs:element name="author">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="name"/>
      <xs:element ref="born"/>
      <xs:element ref="dead" minOccurs=0/>
    </xs:sequence>
    <xs:attribute ref="id"/>
  </xs:complexType>
</xs:element>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="name" type="xs:string"/>
   <xs:element name="qualification" type="xs:string"/>
   <xs:element name="born" type="xs:date"> </xs:element>
   <xs:element name="dead" type="xs:date"> </xs:element>
   <xs:element name="isbn" type="xs:string"> </xs:element>
   <xs:attribute name="id" type="xs:ID"> </xs:attribute>
   <xs:attribute name="available" type="xs:boolean"> </xs:attribute>
   <xs:attribute name="lang" type="xs:language"> </xs:attribute>
   <xs:element name="title">
      <xs:complexType>
         <xs:simpleContent>
            <xs:extension base="xs:string">
               <xs:attribute ref="lang"> </xs:attribute>
            </xs:extension>
         </xs:simpleContent>
      </xs:complexType>
   </xs:element>
   <xs:element name="library">
      <xs:complexType>
         <xs:sequence>
            <xs:element maxOccurs="unbounded" ref="book"> </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
   <xs:element name="author">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="name"> </xs:element>
            <xs:element ref="born"> </xs:element>
            <xs:element ref="dead" minOccurs="0"> </xs:element>
         </xs:sequence>
         <xs:attribute ref="id"> </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="book">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="isbn"> </xs:element>
            <xs:element ref="title"> </xs:element>
            <xs:element ref="author" minOccurs="0" maxOccurs="unbounded”/>
            <xs:element ref="character" minOccurs="0" maxOccurs="unbounded"/>
         </xs:sequence>
         <xs:attribute ref="available"> </xs:attribute>
         <xs:attribute ref="id"> </xs:attribute>
      </xs:complexType>
   </xs:element>
   <xs:element name="character">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref="name"/>
            <xs:element ref="born"/>
            <xs:element ref="qualification"/>
         </xs:sequence>
         <xs:attribute ref="id"> </xs:attribute>
      </xs:complexType>
   </xs:element>

</xs:schema>
```

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
   <xs:element name="library">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="book" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="isbn" type="xs:integer"> </xs:element>
                     <xs:element name="title">
                        <xs:complexType>
                           <xs:simpleContent>
                              <xs:extension base="xs:string">
                                 <xs:attribute name="lang" type="xs:language"
                                 > </xs:attribute>
                              </xs:extension>
                           </xs:simpleContent>
                        </xs:complexType>
                     </xs:element>
                     <xs:element name="author" minOccurs="0" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="name" type="xs:string"> </xs:element>
                              <xs:element name="born" type="xs:date"> </xs:element>
                              <xs:element name="dead" type="xs:date"> </xs:element>
                           </xs:sequence>
                           <xs:attribute name="id" type="xs:ID"> </xs:attribute>
                        </xs:complexType>
                     </xs:element>
                     <xs:element name="character" minOccurs="0" maxOccurs="unbounded">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="name" type="xs:string"> </xs:element>
                              <xs:element name="born" type="xs:date"> </xs:element>
                              <xs:element name="qualification" type="xs:string"
                              > </xs:element>
                           </xs:sequence>
                           <xs:attribute name="id" type="xs:ID"> </xs:attribute>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
                  <xs:attribute type="xs:ID" name="id"> </xs:attribute>
                  <xs:attribute name="available" type="xs:boolean"> </xs:attribute>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Same schema but with everything defined locally!

What's in a Schema?

- A Schema is an XML document (a DTD is not)
- Because it is an XML document, it must have a root element
  - The root element is `<schema>`
- Within the root element, there can be
  - Any number and combination of
    - Inclusions
    - Imports
    - Re-definitions
    - Annotations
  - Followed by any number and combinations of
    - Simple and complex data type definitions
    - Element and attribute definitions
    - Model group definitions
    - Annotations

Structure of a Schema

```
<schema>
   <!- any number of the following -->

   <include .../>
   <import> ... </import>
   <redefine> ... </redefine>
   <annotation> ... </annotation>

   <!- any number of following definitions -->

   <simpleType> ... </simpleType>
   <complexType> ... </complexType>
   <element> ... </element>
   <attribute/>
   <attributeGroup> ... </attributeGroup>
   <group> ... </group>
   <annotation> ... </annotation>
</schema>
```

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Simple Types

Elements

- What is an element with simple type?
  - A simple element is an XML element that can contain only text. It cannot contain any other elements or attributes.

- Can also add restrictions (facets) to a data type in order to limit its content, and you can require the data to match a defined pattern.

Example Simple Element

- The syntax for defining a simple element is:
  - **<xs:element name="xxx" type="yyy"/>**
    where xxx is the name of the element and yyy is the data type of the element. Here are some XML elements:
    - <lastname>Refsnes</lastname>
    - <age>37</age>
    - <dateborn>1968-03-27</dateborn>

And here are the corresponding simple element definitions:

- <xs:element name="lastname" type="xs:string"/>
- <xs:element name="age" type="xs:integer"/>
- <xs:element name="dateborn" type="xs:date"/>

Common XML Schema Data Types

- XML Schema has a lot of built-in data types. Here is a list of the most common types:
  - xs:string
  - xs:decimal
  - xs:integer
  - xs:boolean
  - xs:date
  - xs:time

Declare Default and Fixed Values for Simple Elements

- 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. In the following example the default value is "red":
  - <xs:element name="color" type="xs:string" default="red"/>

  A fixed value is also automatically assigned to the element. You cannot specify another value. In the following example the fixed value is "red":
  - <xs:element name="color" type="xs:string" fixed="red"/>

Attributes (Another simple type)

- All attributes are declared as simple types.
- Only complex elements can have attributes!

What is an Attribute?

- Simple elements cannot have attributes.
  - If an element has attributes, it is considered to be of complex type.
  - But the attribute itself is always declared as a simple type.
  - This means that an element with attributes always has a complex type definition.

How to Define an Attribute

- The syntax for defining an attribute is:
  - <xs:attribute name="xxx" type="yyy"/>
    where xxx is the name of the attribute and yyy is the data type of the attribute. Here are an XML element with an attribute:
  - <lastname lang="EN">Smith</lastname>

    And here are a corresponding simple attribute definition:
  - <xs:attribute name="lang" type="xs:string"/>

Declare Default and Fixed Values for Attributes

- Attributes can have a default value OR a fixed value specified.
- A default value is automatically assigned to the attribute when no other value is specified. In the following example the default value is "EN":
- <xs:attribute name="lang" type="xs:string" default="EN"/>

  A fixed value is also automatically assigned to the attribute. You cannot specify another value. In the following example the fixed value is "EN":

- <xs:attribute name="lang" type="xs:string" fixed="EN"/>

Creating Optional and Required Attributes

- All attributes are optional by default. To explicitly specify that the attribute is optional, use the "use" attribute:
  - <xs:attribute name="lang" type="xs:string" use="optional"/>

  To make an attribute required:

  - <xs:attribute name="lang" type="xs:string" use="required"/>

Restrictions

- As we will see later, simple types can have ranges put on their values
- These are known as *restrictions*

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Complex Types

Complex Elements

- A complex element is an XML element that contains other elements and/or attributes.

- There are four kinds of complex elements:
  - empty elements
  - elements that contain only other elements
  - elements that contain only text
  - elements that contain both other elements and text

- **Note:** Each of these elements may (or must) contain attributes as well!

Examples of Complex XML Elements

- A complex XML element, "product", which is empty:
  - <product pid="1345"/>
- A complex XML element, "employee", which contains only other elements:
  - <employee>
    <firstname>John</firstname>
    <lastname>Smith</lastname>
    </employee>
- A complex XML element, "food", which contains only text:
  - <food type="dessert">Ice cream</food>
- A complex XML element, "description", which contains both elements and text:
  - <description> It happened on <date lang="norwegian">03.03.99</date>..</description>

An Example XML Schema


```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" elementFormDefault="qualified"
   attributeFormDefault="unqualified">
   <xs:element name="rooms">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="room" minOccurs="0" maxOccurs="unbounded">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element name="capacity" type="xs:decimal"/>
                     <xs:element name="equiptmentList"/>
                     <xs:element name="features" minOccurs="0">
                        <xs:complexType>
                           <xs:sequence>
                              <xs:element name="feature" type="xs:string"
                                 maxOccurs="unbounded"/>
                           </xs:sequence>
                        </xs:complexType>
                     </xs:element>
                  </xs:sequence>
                  <xs:attribute name="name" type="xs:string" use="required"/>
               </xs:complexType>
            </xs:element>
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Referencing XML Schema in XML documents

Sample Schema header

- The <schema> element may contain some attributes. A schema declaration often looks something like this:
  - <?xml version="1.0"?>

    <xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://www.w3schools.com" xmlns="http://www.w3schools.com"
    elementFormDefault="qualified">

    … …

    </xs:schema>

Schema headers, cont.

- The following fragment:

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

indicates that the elements and data types used in the schema (schema, element, complexType, sequence, string, boolean, etc.) come from the "http://www.w3.org/2001/XMLSchema" namespace.

It also specifies that the elements and data types that come from the "http://www.w3.org/2001/XMLSchema" namespace should be prefixed with xs: !!

Schema header, cont.

- This fragment:
  - targetNamespace=http://www.w3schools.com
    indicates that the elements defined by this schema (note, to, from, heading, body.) come from the "http://www.w3schools.com" namespace.

- This fragment:
  - xmlns=http://www.w3schools.com
    indicates that the default namespace is "http://www.w3schools.com".

- This fragment:
  - elementFormDefault="qualified“
    indicates that any elements used by the XML instance document which were declared in this schema must be namespace qualified.

Referencing schema in XML

- This XML document has a reference to an XML Schema:

```
<?xml version="1.0"?>
<note xmlns="http://www.w3schools.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.w3schools.com note.xsd">
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

Referencing schema in xml, cont.

- The following fragment:
  - xmlns=http://www.w3schools.com
    specifies the default namespace declaration.

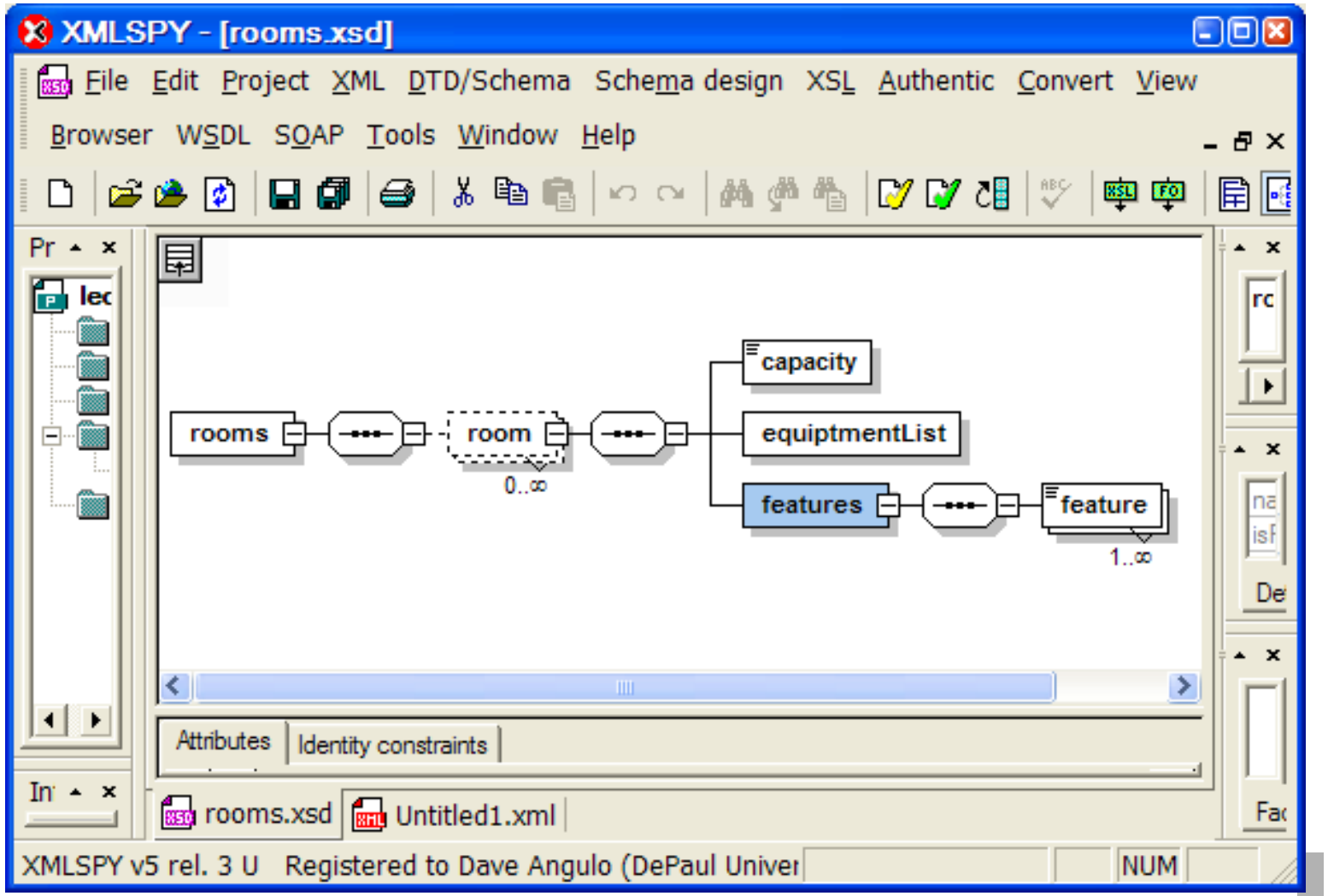
  - This declaration tells the schema-validator that all the elements used in this XML document are declared in the "http://www.w3schools.com" namespace.

...

- Once you have the XML Schema Instance namespace available:
  - xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance

    you can use the schemaLocation attribute. This attribute has two values. The first value is the namespace to use. The second value is the location of the XML schema to use for that namespace:

  - xsi:schemaLocation="http://www.w3schools.com note.xsd"


XMLSPY - [rooms.xsd]
File Edit Project XML DTD/Schema Schema design XSL Authentic Convert View
Browser WSDL SOAP Tools Window Help
rooms
room
0..∞
capacity
equiptmentList
features
feature
1..∞
Attributes
Identity constraints
rooms.xsd
Untitled1.xml
XMLSPY v5 rel. 3 U Registered to Dave Angulo (DePaul Univer
NUM

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Using References

Using References

- You don't have to have the content of an element defined in the nested fashion as just shown

```
<xs:element name="rooms">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="room">
     <xs:complexType>
          <xs:sequence>
             <xs:element name="capacity" type="xs:decimal"/>
```

- You can define the element globally and use a reference to it instead

```
<xs:element name="rooms">
 <xs:complexType>
  <xs:sequence>
    <xs:element ref="room"/>
</xs:sequence></xs:complexType></xs:element>

<xs:element name="room">
 ...
</xs:element>
```

Rooms Schema using References

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified" attributeFormDefault="unqualified">

<xs:element name="rooms">
 <xs:complexType>
  <xs:sequence>
   <xs:element ref="room" maxOccurs="unbounded"/>
  </xs:sequence>
 </xs:complexType>
</xs:element>

<xs:element name="room">
  <xs:complexType>
    <xs:sequence>
            <xs:element name="capacity" type="xs:decimal"/>
            <xs:element name="equiptmentList"/>
            <xs:element ref="features" minOccurs="0" maxOccurs="1"/>
    </xs:sequence>
    <xs:attribute name="name" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

<xs:element name="features">
   <xs:complexType>
      <xs:sequence>
            <xs:element name="feature" type="xs:string"
                  maxOccurs="unbounded"/>
      </xs:sequence>
   </xs:complexType>
```

Types

Both elements and attributes have types, which are defined in the Schema.
One can reuse types by giving them names.

```
<xsd:element name="Robot">
   <xsd:complexType>
            <xsd:sequence>
                     <xsd:element ref="Sensor_List" minOccurs="0"/>
                     <xsd:element ref="Specification_List" minOccurs="0"/>
                     <xsd:element ref="Note" minOccurs="0"/>
            </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

OR

```
<xsd:element name="Robot" type="RoboType">
<xsd:complexType  name="RoboType" >
            <xsd:sequence>
                     <xsd:element ref="Sensor_List" minOccurs="0"/>
                     <xsd:element ref="Specification_List" minOccurs="0"/>
                     <xsd:element ref="Note" minOccurs="0"/>
            </xsd:sequence>
</xsd:complexType>
</xsd:element>
```

Other XML Schema Features

- Foreign key facility (uses Xpath)

- Rich datatype facility
  - Build up datatypes by inheritance
  - Don’t need to list all of the attributes (can say "these attributes plus others").
  - Restrict strings using regular expressions

- Namespace aware.
  - Can restrict location of an element based on a namespaces

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Restrictions

Datatype Restrictions

- A DTD can only say that price can be any non-markup text. Like this translated to Schemas

```
<xsd:element name="zip" type="xsd:string"/>
```

- But in Schema you can do better:

```
<xsd:element name="zip" type="xsd:decimal"/>
```

- Or even, make your own restrictions

```
<xsd:simpleType name="ZipPlus4">
    <xsd:restriction base="xsd:string">
        <xsd:length value="10"/>
        <xsd:pattern value="\d{5}-\d{4}"/>
    </xsd:restriction>
</xsd:simpleType>
<xsd:element name="zip" type="ZipPlus4">
```

Restriction Ranges

- The restrictions must be "derived" from a base type, so it's object based

```
<xs:element name="LifeUniverseAndEverything">
   <xs:simpleType>
            <xs:restriction base="xs:integer">
                     <xs:minInclusive value="42"/>
                     <xs:maxInclusive value="42"/>
            </xs:restriction>
   </xs:simpleType>
</xs:element>
```

- Preceding "derived" from "integer"
- Has 2 restrictions (called "facets")
  - The first says that it must be greater than 41
  - The second says that it must be less than 43
- XML file is "42"

```
<LifeUniverseAndEverything xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="LifeUniverseEverything.xsd">42</LifeUniverseAndEverything>
```

Facet	Description
enumeration	Defines a list of acceptable values
fractionDigits	The maximum number of decimal places allowed. >=0
length	The exact number of characters or list items allowed. >=0
maxExclusive	The upper bounds for numeric values (the value must be less than the value specified)
maxInclusive	The upper bounds for numeric values (the value must be less than or equal to the value specified)
maxLength	The maximum number of characters or list items allowed. >=0
minExclusive	The lower bounds for numeric values (the value must be greater than the value specified)
minInclusive	The lower bounds for numeric values (the value must be greater than or equal to the value specified)
minLength	The minimum number of characters or list items allowed >=0
pattern	The sequence of acceptable characters based on a regular expression
totalDigits	The exact number of digits allowed. >0
whiteSpace	Specifies how white space (line feeds, tabs, spaces, and carriage returns) is handled

Enumeration Facet

```
<xs:element name="FavoriteColor">
  <xs:simpleType>
      <xs:restriction base="xs:string">
          <xs:enumeration value="red"/>
          <xs:enumeration value="no blue"/>
          <xs:enumeration value="aarrrrggghh!!"/>
      </xs:restriction>
  </xs:simpleType>
</xs:element>
```

Patterns (Regular Expressions)

- One interesting facet is the pattern, which allows restrictions based on a regular expression

- This regular expression specifies a normal word of one or more characters:

```
<xs:element name="Word">
        <xs:simpleType name="WordType">
                <xs:restriction base="xs:string">
                        <xs:pattern value="[a-zA-Z]+"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Patterns (Regular Expressions)

- Individual characters may be repeated a specific number of times in the regular expression.

- The following regular expression restricts the string to exactly 8 alpha-numeric characters:

```
<xs:element name="password">
        <xs:simpleType>
                <xs:restriction base="xs:string">
                        <xs:pattern value="[a-zA-Z0-9]{8}"/>
                </xs:restriction>
        </xs:simpleType>
</xs:element>
```

Whitespace facet

- The "whitespace" facet controls how white space in the element will be processed

- There are three possible values to the whitespace facet
  - "preserve" causes the processor to keep all whitespace as-is
  - "replace" causes the processor to replace all whitespace characters (tabs, carriage returns, line feeds, spaces) with space characters
  - "collapse" causes the processor to replace all strings of whitespace characters (tabs, carriage returns, line feeds, spaces) with a single space character

```
<xs:simpleType>
  <xs:restriction base="xs:string">
        <xs:whitespace value="replace"/>
  </xs:restriction>
```

Types

Both elements and attributes have types, which are defined in the Schema.
One can reuse types by giving them names.

Addr.xsd:

```
<xsd:element name="Address">
   <xsd:complexType>
          <xsd:sequence>
                 <xsd:element name="Street" type="xsd:string"/>
                 <xsd:element name="Apartment" type="xsd:string"/>
                 <xsd:element name="Zip" type="xsd:string"/>
          </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

OR

```
<xsd:complexType name="AddrType">
   <xsd:sequence>
          <xsd:element name="Street" type="xsd:string"/>
          <xsd:element name="Apartment" type="xsd:string"/>
          <xsd:element name="Zip" type="xsd:string"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:element name="ShipAddress" type="AddrType"/>
<xsd:element name="BillAddress" type="AddrType"/>
```

Types

- The usage in the XML file is identical:

```
<?xml version="1.0" encoding="UTF-8"?>
<PurchaseOrder xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Address-WithTypeName.xsd">

    <BillAddress>
        <Street>1108 E. 58th St.</Street>
        <Apartment>Ryerson 155</Apartment>
        <Zip>60637</Zip>
    </BillAddress>
    <ShipAddress>
        <Street>1108 E. 58th St.</Street>
        <Apartment>Ryerson 155</Apartment>
        <Zip>60637</Zip>
    </ShipAddress>
```

Type Extensions

- A third way of creating a complex type is to extend another complex type (like OO inheritance)

```
<xs:element name="Employee" type="PersonInfoType"/>

<xs:complexType name="PersonNameType">
  <xs:sequence>
        <xs:element name="FirstName" type="xs:string"/>
        <xs:element name="LastName" type="xs:string"/>
  </xs:sequence>
</xs:complexType>

<xs:complexType name="PersonInfoType">
  <xs:complexContent>
        <xs:extension base="PersonNameType">
              <xs:sequence>
                    <xs:element name="Address" type="xs:string"/>
                    <xs:element name="City" type="xs:string"/>
                    <xs:element name="Country" type="xs:string"/>
              </xs:sequence>
        </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Type Extensions (use)

- To use a type that is an extension of another, it is as though it were all defined in a single type

```
<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="TypeExtension.xsd">
        <FirstName>King</FirstName>
        <LastName>Arthur</LastName>
        <Address>Round Table</Address>
        <City>Camelot</City>
        <Country>England</Country>
</Employee>
```

Simple Content in Complex Type

- If a type contains only simple content (text and attributes), a <simpleContent> element can be put inside the <complexType>

- <simpleContent> must have either a <extension> or a <restriction>

- This example is from the (Bridge of Death) Episode Dialog:

```
<xs:element name="dialog">
  <xs:complexType>
      <xs:simpleContent>
          <xs:extension base="xs:string">
              <xs:attribute name="speaker"
                  type="xs:string" use="required"/>
          </xs:extension>
      </xs:simpleContent>
  </xs:complexType>
```

Model Groups

- Model Groups are used to define an element that has
  - mixed content (elements and text mixed)
  - element content

- Model Groups can be
  - all
    - the elements specified must all be there, but in any order
  - choice
    - any of the elements specified may or may not be there
  - sequence
    - all of the elements specified must appear in the specified order

"All" Model Group

- The following schema specifies 3 elements and mixed content

```
<xs:element name="BookCover">
   <xs:complexType mixed="true">
          <xs:all minOccurs="0" maxOccurs="1">
                 <xs:element name="BookTitle" type="xs:string"/>
                 <xs:element name="Author" type="xs:string"/>
                 <xs:element name="Publisher" type="xs:string"/>
          </xs:all>
   </xs:complexType>
</xs:element>
```

- The following XML file is valid in the above schema

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

   Title:  <BookTitle>The Holy Grail</BookTitle>
   Published:      <Publisher>Moose</Publisher>
   Author:         <Author>Monty Python</Author>
</BookCover>
```

✓Attributes

```
<xs:element name="dialog">
  <xs:complexType>
        <xs:simpleContent>
            <xs:extension base="xs:string">
                  <xs:attribute name="speaker"
                                type="xs:string"
  use="required"/>
            </xs:extension>
        </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

The attribute declaration is part of the type of the element.

...

Attributes

```
<xsd:element name="cartoon">
  <xsd:complexType>
        <xsd:sequence>
              <xsd:element ref="character" minOccurs="0"
              maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="name" type="xsd:string" use="required"/>
        <xsd:attribute name="genre" type="xsd:string" use="required"/>
        <xsd:attribute name="syndicated" use="required">
              <xsd:simpleType>
                    <xsd:restriction base="xsd:NMTOKEN">
                          <xsd:enumeration value="yes"/>
                          <xsd:enumeration value="no"/>
                    </xsd:restriction>
              </xsd:simpleType>
        </xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

If an attribute type is more complicated than a basic type, then we spell out the type in a

Optional and Required Attributes

- All attributes are optional by default. To explicitly specify that the attribute is optional, use the "use" attribute:

```
<xs:attribute name="speaker" type="xs:string" use="optional"/>
```

- To make an attribute required:

```
<xs:attribute name="speaker" type="xs:string" use="required"/>
```

Other XML Schema Features

- Foreign key facility (uses Xpath)
- Rich datatype facility
  - Build up datatypes by inheritance
  - Don’t need to list all of the attributes (can say “these attributes plus others).
  - Restrict strings using regular expressions
- Namespace aware.
  - Can restrict location of an element based on a namespaces

XML Schema Status

- Became a W3C recommendation Spring 2001
  - World domination expected imminently.
  - Supported in Xalan.
  - Supported in XML spy and other editor/validators.

- On the other hand:
  - More complex than DTDs.
  - Ultra verbose.

Validating a Schema

- By using Xeena or XMLspy or XML Notepad.
  - When publishing hand-written XML docs, this is the way to go.

- By using a Java program that performs validation.
  - When validating on-the-fly, must do it this way

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Some guidelines for Schema design

Designing a Schema

- Analogous to database schema design --- look for intuitive names

- Can start with an E-R diagram, and then convert
  - Attributes to Attributes
  - Subobjects to Subelements
  - Relationships to IDREFS

- Normalization? Still makes sense to avoid repetition whenever possible–
  - If you have an Enrolment document, only list Ids of students, not their names.
  - Store names in a separate document
  - Leave it to tools to connect them

Designing a Schema (cont.)

- Difficulties:
  - Many more degrees of freedom than with database schemas:
  - e.g. one can associate information with something by including it as an attribute or a subelement.

```
<ADDRESS NAME="Martin Sheen", Street="1222 Alameda
  Drive" ,City="Carmel", State="CA", ZIP="40145">

<ADDRESS>
    <NAME> Martin Sheen </NAME>
    …
    <ZIP> 4145 </ZIP>
</ADDRESS>
```

  - ELEMENTS are more extensible – use when there is a possibility that more substructure will be added.
  - ATTRIBUTES are easier to search on.

“Rules” for Designing a Schema

- Never leave structure out. The following is definitely a bad idea:
  - `<ADDRESS> Martin Sheen 1222 Alameda Drive, Carmel, CA 40145 </ADDRESS>`
- Better would be:
  - `<ADDRESS firstName=“Martin” lastname=“Sheen” streenNum=“1222” streenName=“Alameda Drive” city=“Carmel” state=“CA” zip=“40145” />`
- Or:

```
<ADDRESS>
   <name>
      <first>Martin</first><last>Sheen</last>
   </name>
   <street>
      <num>1222</num><name>Alameda Drive</name>
   </street>
   <city>Carmel</city>
   <state>CA</state><zip>40145</zip>
</ADDRESS>
```

More"Rules" for Designing a Schema

- When to use Elements (instead of attributes)
  - Do not put large text blocks inside an attribute
    - (Bad Idea) <book type="memoir" content="Bravely bold Sir Robin rode forth from Camelot.

He was not afraid to die, O brave Sir Robin.
He was not at all afraid to be killed in nasty ways,
Brave, brave, brave, brave Sir Robin!

He was not in the least bit scared to be mashed into a pulp,
Or to have his eyes gouged out and his elbows broken,
To have his kneecaps split and his body burned away
And his limbs all hacked and mangled, brave Sir Robin!

His head smashed in and his heart cut out
And his liver removed and his bowels unplugged…">

  - Elements are more flexible, so use an Element if you think you might have to add more substructure later on.

More “Rules” for Designing Schemas

- More on when to use Elements (instead of Attributes)
  - Use an embedded element when the information you are recording is a constituent part of the parent element
    - one's head and one's height are both inherent to a human being,
    - you can't be a conventionally structured human being without having a head and having a height
    - One's head is a constituent part and one's height isn't -- you can cut off my head, but not my height
  - use embedded elements for complex structure validation (obvious)
  - use embedded elements when you need to show order (attributes are not ordered)

More “Rules” for Designing Schemas

- When to use Attributes instead of Elements
  - use an attribute when the information is inherent to the parent but not a constituent part (height instead of head)
  - use attributes to stress the one-to-one relationship among pieces of information
    - to stress that the element represents a tuple of information
    - dangerous rule, though
      - Leads to the extreme formulation that a `<chapter>` element can have a `TITLE=` attribute
      - And then to the conclusion that it really ought to have a `CONTENT=` attribute too
      - Then you find yourself writing the entire document as an empty element with an attribute value as long as the Quest for the Holy Grail
  - use attributes for simple datatype validation (obviously)

Bookings XML Schema

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified" attributeFormDefault="unqualified">
  <xs:element name="bookings">
    <xs:complexType>
      <xs:sequence>
          <xs:element ref="lastUpdated" maxOccurs="1" minOccurs="0"/>
          <xs:element ref="meetingDate" maxOccurs="unbounded"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="year" type="xs:integer"/>
  <xs:element name="month" type="xs:string"/>
  <xs:element name="day" type="xs:integer"/>
```

Note that there are four *global types* in this document!

Bookings, cont.

```
<xs:element name="meetingDate">
    <xs:complexType>
      <xs:sequence>
            <xs:element ref="year"/>
            <xs:element ref="month"/>
            <xs:element ref="day"/>
            <xs:element ref="meeting" maxOccurs="unbounded" minOccurs="0"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

<xs:element name="lastUpdated">
    <xs:complexType>
      <xs:attribute name="date" type="xs:string"/>
      <xs:attribute name="time" type="xs:string"/>
    </xs:complexType>
</xs:element>
```

Bookings, cont.

```
<xs:element name="meeting">
    <xs:complexType>
      <xs:sequence>
            <xs:element name="meetingName" maxOccurs="1" minOccurs="1" type="xs:string"/>
            <xs:element name="roomName" maxOccurs="1" minOccurs="1" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

An Example Bookings Document

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

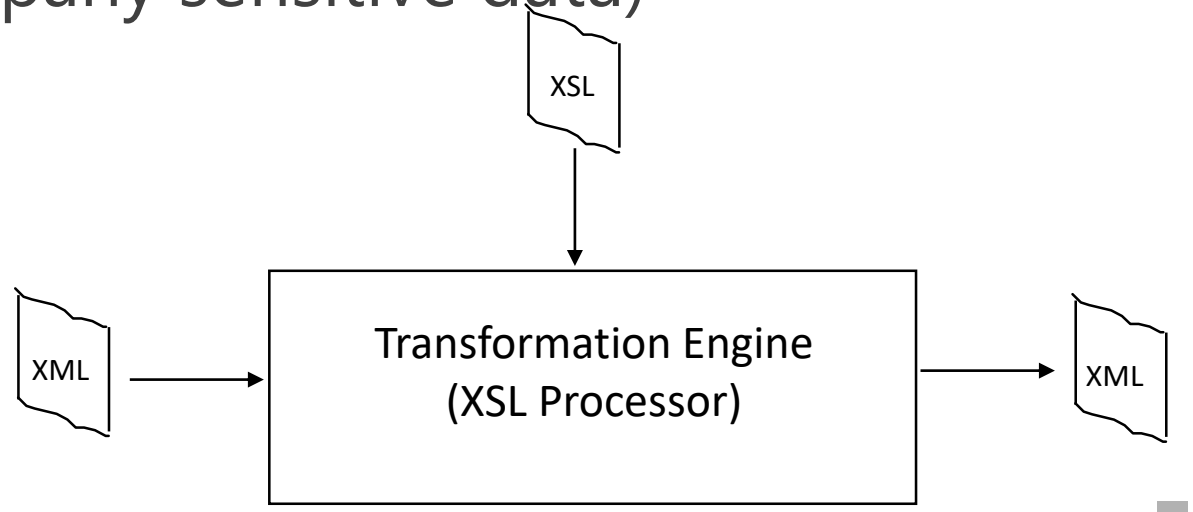
• **Reverse engineer a reasonable schema for the following sample xml file**

```
<?xml version="1.0" encoding="UTF-8"?>
<bookings xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="../schemas/Bookings.xsd">
        <meetingDate>
                <year>2003</year>
                <month>April</month>
                <day>1</day>
                <meeting>
                        <meetingName>Democratic Party</meetingName>
                        <roomName>Green Room</roomName>
                </meeting>
                <meeting>
                        <meetingName>Republican Party</meetingName>
                        <roomName>Red Room</roomName>
                </meeting>
        </meetingDate>
</bookings>
```

XSL

Transformation Language

- XSL may be used as a transformation language --> it may be used to transform an XML document into another XML document (perhaps the new one is the same, minus company sensitive data)



Example: Filter Gold Members

```
<?xml version="1.0"?>
<FitnessCenter>
    <Member id="1" level="platinum">
        <Name>Jeff</Name>
        <Phone type="home">555-1234</Phone>
        <Phone type="work">555-4321</Phone>
        <FavoriteColor>lightgrey</FavoriteColor>
    </Member>
    <Member id="2" level="gold">
        <Name>David</Name>
        <Phone type="home">383-1234</Phone>
        <Phone type="work">383-4321</Phone>
        <FavoriteColor>lightblue</FavoriteColor>
    </Member>
    <Member id="3" level="platinum">
        <Name>Roger</Name>
        <Phone type="home">888-1234</Phone>
        <Phone type="work">888-4321</Phone>
        <FavoriteColor>lightyellow</FavoriteColor>
    </Member>
</FitnessCenter>
```

```
<?xml version="1.0"?>
<FitnessCenter>
    <Member id="1" level="platinum">
        <Name>Jeff</Name>
        <Phone type="home">555-1234</Phone>
        <Phone type="work">555-4321</Phone>
        <FavoriteColor>lightgrey</FavoriteColor>
    </Member>
    <Member id="3" level="platinum">
        <Name>Roger</Name>
        <Phone type="home">888-1234</Phone>
        <Phone type="work">888-4321</Phone>
        <FavoriteColor>lightyellow</FavoriteColor>
    </Member>
</FitnessCenter>
```

XML Transformations - all about (Template) “Rules”

- “Hey xsl processor, when you encounter the root element (e.g., FitnessCenter) do [action1]”
- “Hey xsl processor, when you encounter the Member element do [action2]”
- “Hey xsl processor, when you encounter the Name element do [action3]”
- And so forth

XML Transformations - all about (Template) “Rules”

- Each template rule has two parts:
  - A pattern or matching part, that identifies the XML node in the source document to which the action part is to be applied. Matching information is contained in an attribute.
  - An action part that details the transformation of the node

XSL Document Structure

```
<?xml version="1.0"?>
<xsl:stylesheet>
    <xsl:template match="/">
        [action]
    </xsl:template>
    <xsl:template match="FitnessCenter">
        [action]
    </xsl:template>
    <xsl:template match="Member">
        [action]
    </xsl:template>
    ...
</xsl:stylesheet>
```

Template Rules

Template rules take the following general form:

```
<xsl:template match="pattern">
  [ action ]
</xsl:template>
```

Template Rules (Example)

```
<xsl:template match="Member">
        <xsl:apply-templates/>
</xsl:template>
```

<xsl:template match="Member">

"Hey XSL processor, as you parse through the XML document and you get to a <Member> element use this template rule."

<xsl:apply-templates/>

"Go to each of my children (the Member children) and apply the template rules to them."

Terminology

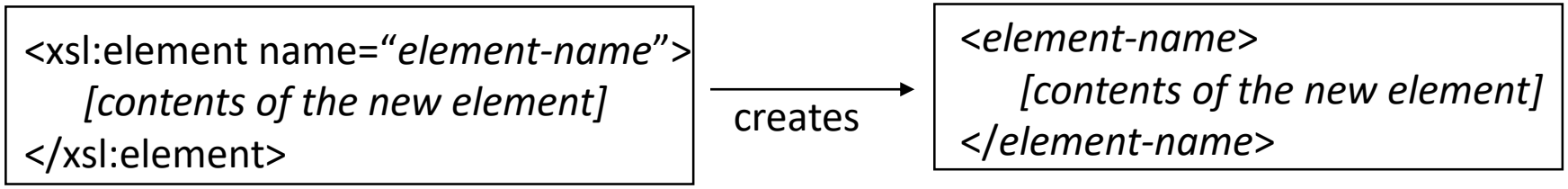
In FitnessCenter.xml we have (snippet):

```
<FitnessCenter>
    <Member>
        <Name>Jeff</Name>
        <Phone type="home">555-1234</Phone>
        <Phone type="work">555-4321</Phone>
        <FavoriteColor>lightgrey</FavoriteColor>
    </Member>
    ...
</FitnessCenter>
```

"Member is a *child* element of the FitnessCenter element. Name, Phone, Phone, and FavoriteColor are *children* elements of the Member element. Member is a *parent* of Name. FitnessCenter and Member are *ancestors* of Name."

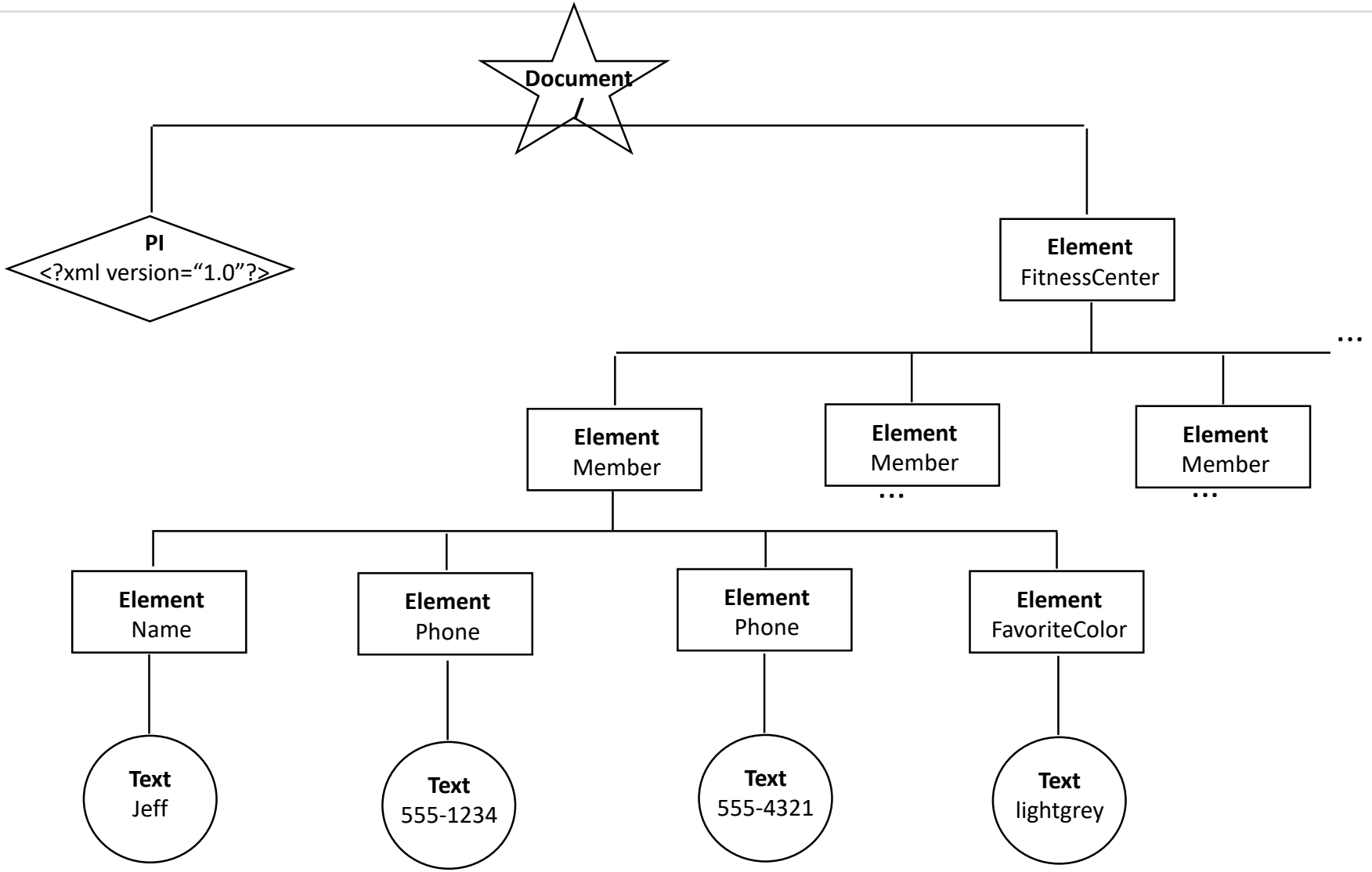
xsl:element

- Suppose that you are writing a stylesheet to generate an XML document. Obviously, you will need your stylesheet to output elements.
  - xsl:element is used to create elements



Identity Transformation

- For our first example, lets create a stylesheet which simply creates an XML document that is a copy of the input XML document


Document
PI
<?xml version="1.0"?>
Element
FitnessCenter
...
Element
Member
Element
Member
...
Element
Member
...
Element
Name
Element
Phone
Element
Phone
Element
FavoriteColor
Text
Jeff
Text
555-1234
Text
555-4321
Text
lightgrey

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

   <xsl:output method="xml"/>

   <xsl:template match="/">
      <xsl:apply-templates/>
   </xsl:template>

   <xsl:template match="FitnessCenter">
      <xsl:element name="FitnessCenter">
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>

   <xsl:template match="Member">
      <xsl:element name="Member">
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>
```

Cont. -->

```
    <xsl:template match="Name">
      <xsl:element name="Name">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:template>

    <xsl:template match="Phone">
      <xsl:element name="Phone">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:template>

    <xsl:template match="FavoriteColor">
      <xsl:element name="FavoriteColor">
        <xsl:apply-templates/>
      </xsl:element>
    </xsl:template>

    <xsl:template match="text()">
            <xsl:value-of select="."/>
    </xsl:template>

</xsl:stylesheet>
```

(see xml-example01)

```
<?xml version="1.0" encoding="UTF-8"?>
<FitnessCenter>
	<Member>
		<Name>Jeff</Name>
		<Phone>555-1234</Phone>
		<Phone>555-4321</Phone>
		<FavoriteColor>lightgrey</FavoriteColor>
	</Member>
	<Member>
		<Name>David</Name>
		<Phone>383-1234</Phone>
		<Phone>383-4321</Phone>
		<FavoriteColor>lightblue</FavoriteColor>
	</Member>
	<Member>
		<Name>Roger</Name>
		<Phone>888-1234</Phone>
		<Phone>888-4321</Phone>
		<FavoriteColor>lightyellow</FavoriteColor>
	</Member>
</FitnessCenter>
```

Note that we've lost the attribute on the Member element

Getting Member's Attribute:

```
<xsl:template match="Member">
    <xsl:element name="Member">
      <xsl:for-each select="@*">
        <xsl:attribute name="{name(.)}">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>
```

For each attribute

Add an attribute to the element being output. The name of the attribute is the name of the current attribute being processed. The value of the attribute is the value of the current attribute being processed.

```
<?xml version="1.0" encoding="UTF-8"?>
<FitnessCenter>
      <Member level=“platinum”>
            <Name>Jeff</Name>
            <Phone type="home">555-1234</Phone>
            <Phone type="work">555-4321</Phone>
            <FavoriteColor>lightgrey</FavoriteColor>
      </Member>
      <Member level=“gold”>
            <Name>David</Name>
            <Phone type="home">383-1234</Phone>
            <Phone type="work">383-4321</Phone>
            <FavoriteColor>lightblue</FavoriteColor>
      </Member>
      <Member level=“platinum”>
            <Name>Roger</Name>
            <Phone type="home">888-1234</Phone>
            <Phone type="work">888-4321</Phone>
            <FavoriteColor>lightyellow</FavoriteColor>
      </Member>
</FitnessCenter>
```

Generalize

- Our identity stylesheet will only work for FitnessCenter XML documents. We can make a stylesheet which does an identity transformation on any XML document.

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

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="*">
    <xsl:element name="{name(.)}">
      <xsl:for-each select="@*">
        <xsl:attribute name="{name(.)}">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </xsl:for-each>
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="text()">
                <xsl:value-of select="."/>
  </xsl:template>

</xsl:stylesheet>
```

Default Template Rules

- Every xsl document has two default template rules
- These rules are applied when the XSL Processor cannot find a template rule to use in your stylesheet
- Here are the two default template rules:

```
<xsl:template match="/ | *">
        <xsl:apply-templates/>
</xsl:template>
```

"Match on the document or any element. The action is to go to the children and execute their template rules."

```
<xsl:template match="text()">
        <xsl:value-of select="."/>
</xsl:template>
```

"Match on a text node. The action is to output the value of the text node."

Multiple Applicable Rules

Suppose that the XSL Processor is processing FitnessCenter and it gets to the <Member> element. Why does it use:

<xsl:template match="Member">...

and not the default template rule:

<xsl:template match="/ | *">...

??? After all, both apply.

Answer: given two rules that apply, *the more specific rule wins*.

--> Clearly, "*" is much more general than "Member".

"*" matches on any element. "Member" just matches on the Member element.

Smallest Identity Transformation Stylesheet

- Now that we know about the default template rules, we can further reduce the size of the stylesheet.

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

   <xsl:output method="xml"/>

   <xsl:template match="*">
      <xsl:element name="{name(.)}">
         <xsl:for-each select="@*">
            <xsl:attribute name="{name(.)}">
               <xsl:value-of select="."/>
            </xsl:attribute>
         </xsl:for-each>
         <xsl:apply-templates/>
      </xsl:element>
   </xsl:template>

</xsl:stylesheet>
```

<xsl:apply-templates select="*pattern*">

- The xsl:apply-templates element (without the select attribute) tells the XSL Processor to apply the template rules to all children (in document order)
- The xsl: apply-templates element can have a select attribute that tells the XSL Processor to process only the child element that matches "*pattern*".
  - Thus, the select attribute rule enables us to specify the order in which the children are processed

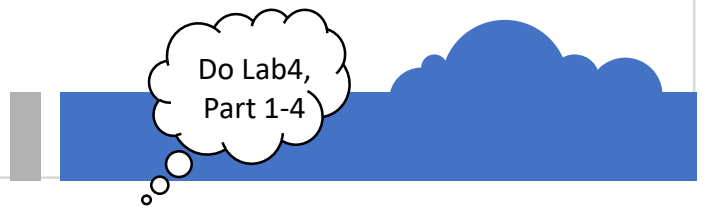
<xsl:apply-templates select="*pattern*">

```
<xsl:template match="Member">
    <xsl:apply-templates select="Name"/>
    <xsl:apply-templates select="Phone[@type='work']"/>
</xsl:template>
```

"Go to the template rule for my Name child element. Then go to the template rule for the work Phone child element."

```
<xsl:template match="Member">
    <xsl:apply-templates select="*"/>
</xsl:template>
```

"Go to all the child **element** nodes (not to any child text nodes)."



mode Attribute

- Allows you to create multiple template rules for the same element. Each template rule can process the element differently.
- So, you can have multiple template rules for the same element. Just give each template rule a different mode

```
<xsl:template match="Name" mode="Normal">

<xsl:template match="Name" mode="footnote">
```

Problem

- Identity transform the FitnessCenter.xml document. However, after you have copied all the Members, follow up with a (new) GoldMembers section, containing the name of each gold member (within stars)
- The next slide shows what the output XML file should look like

```
<?xml version="1.0" encoding="UTF-8"?>
<FitnessCenter>
      <Member level="platinum">
            <Name>Jeff</Name>
            <Phone>555-1234</Phone>
            <Phone>555-4321</Phone>
            <FavoriteColor>lightgrey</FavoriteColor>
      </Member>
      <Member level="gold">
            <Name>David</Name>
            <Phone>383-1234</Phone>
            <Phone>383-4321</Phone>
            <FavoriteColor>lightblue</FavoriteColor>
      </Member>
      <Member level="platinum">
            <Name>Roger</Name>
            <Phone>888-1234</Phone>
            <Phone>888-4321</Phone>
            <FavoriteColor>lightyellow</FavoriteColor>
      </Member>
      <GoldMembers>
        <Name>***David***</Name>
      </GoldMembers>
</FitnessCenter>
```

Note that the names here are processed differently than the name in the GoldMembers section

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

  <xsl:output method="xml"/>

  <xsl:template match="/">
    <xsl:apply-templates/>
  </xsl:template>

  <xsl:template match="FitnessCenter">
    <xsl:element name="FitnessCenter">
      <xsl:apply-templates/>
      <xsl:element name="GoldMembers">
        <xsl:for-each select="Member[@level='gold']">
          <xsl:apply-templates select="Name" mode="footnote"/>
        </xsl:for-each>
      </xsl:element>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Member">
    <xsl:element name="Member">
      <xsl:for-each select="@*">
        <xsl:attribute name="{name(.)}">
          <xsl:value-of select="."/>
        </xsl:attribute>
      </xsl:for-each>
      <xsl:apply-templates mode="Normal"/>
    </xsl:element>
  </xsl:template>
```

```
  <xsl:template match="Name" mode="Normal">
    <xsl:element name="Name">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Name" mode="footnote">
    <xsl:element name="Name">
      <xsl:text>***</xsl:text>
        <xsl:apply-templates/>
      <xsl:text>***</xsl:text>
    </xsl:element>
  </xsl:template>

  <xsl:template match="Phone" mode="Normal">
    <xsl:element name="Phone">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

  <xsl:template match="FavoriteColor" mode="Normal">
    <xsl:element name="FavoriteColor">
      <xsl:apply-templates/>
    </xsl:element>
  </xsl:template>

</xsl:stylesheet>
```

Do Lab5,
Part 1

Stylesheet Reuse via xsl:include and xsl:import

- The elements xsl:include and xsl:import enable you to reuse other stylesheets.
- These elements are “top-level elements”. This means that they must be immediate children of the xsl:stylesheet element (i.e., they cannot be within a template rule)
- The xsl:include element is basically a macro substitution - the element is replaced by the contents of stylesheet it references

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

   <xsl:include href="file://localhost/xml-course/new-xsl/toUpperCase.xsl"/>

   <xsl:template match="FitnessCenter">
      ...
   </xsl:template>
   ...
</xsl:stylesheet>
```

Replace the xsl:include element with the contents of the referenced stylesheet (i.e., all the children of xsl:stylesheet)

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

<xsl:variable name="lcase" select="'abcdefghijklmnopqrstuvwxyz'"/>
<xsl:variable name="ucase" select="'ABCDEFGHIJKLMNOPQRSTUVWXYZ'"/>

<xsl:template match="*">
  <xsl:apply-templates select="@* | * | text() | comment() | processing-instruction()"/>
</xsl:template>

<xsl:template match="@*">
  <xsl:value-of select="translate(.,$lcase, $ucase)"/>
</xsl:template>

<xsl:template match="text()">
  <xsl:value-of select="translate(.,$lcase, $ucase)"/>
</xsl:template>

</xsl:stylesheet>
```

toUpperCase.xsl

xsl:import

- xsl:import acts just like xsl:include - the stylesheet that it references is macro-substituted. However, there is a difference:
  - With xsl:include the stuff that is macro-substituted into the stylesheet has the same precedence as the rest of the stylesheet. It is as though you had one stylesheet.
  - With xsl:import the stuff that is macro-substituted into the stylesheet has lower precedence than the rest of the stylesheet. Also, all xsl:import elements must come first in the stylesheet.

JSON basics

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

JSON

Data Interchange

- The key idea in Ajax.
- An alternative to page replacement.
- Applications delivered as pages.
- How should the data be delivered?

History of Data Formats

- Ad Hoc
- Database Model
- Document Model
- Programming Language Model

JSON

- JavaScript Object Notation
- Minimal
- Textual
- Subset of JavaScript

JSON

- A Subset of ECMA-262 Third Edition.
- Language Independent.
- Text-based.
- Light-weight.
- Easy to parse.

JSON Is Not...

- JSON is not a document format.
- JSON is not a markup language.
- JSON is not a general serialization format.
  - No cyclical/recurring structures.
  - No invisible structures.
  - No functions.

History

- 1999 ECMAScript Third Edition
- 2001 State Software, Inc.
- 2002 JSON.org
- 2005 Ajax
- 2006 **RFC** 4627

Languages

- Chinese
- English
- French
- German
- Italian
- Japanese
- Korean

Languages

- ActionScript
- C / C++
- C#
- Cold Fusion
- Delphi
- E
- Erlang
- Java
- Lisp
- Perl
- Objective-C
- Objective CAML
- PHP
- Python
- Rebol
- Ruby
- Scheme
- Squeak

Object Quasi-Literals

- JavaScript
- Python
- NewtonScript

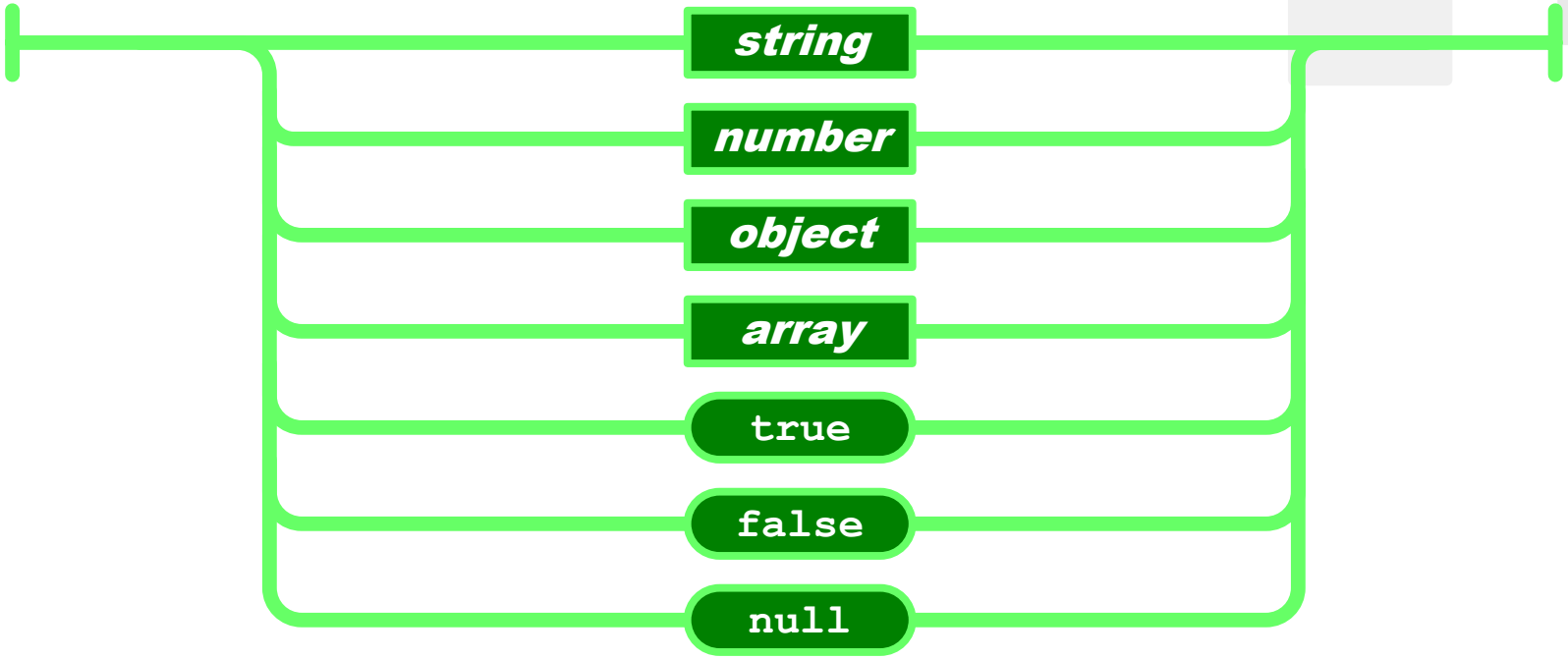
Values

- Strings
- Numbers
- Booleans

- Objects
- Arrays

- `null`

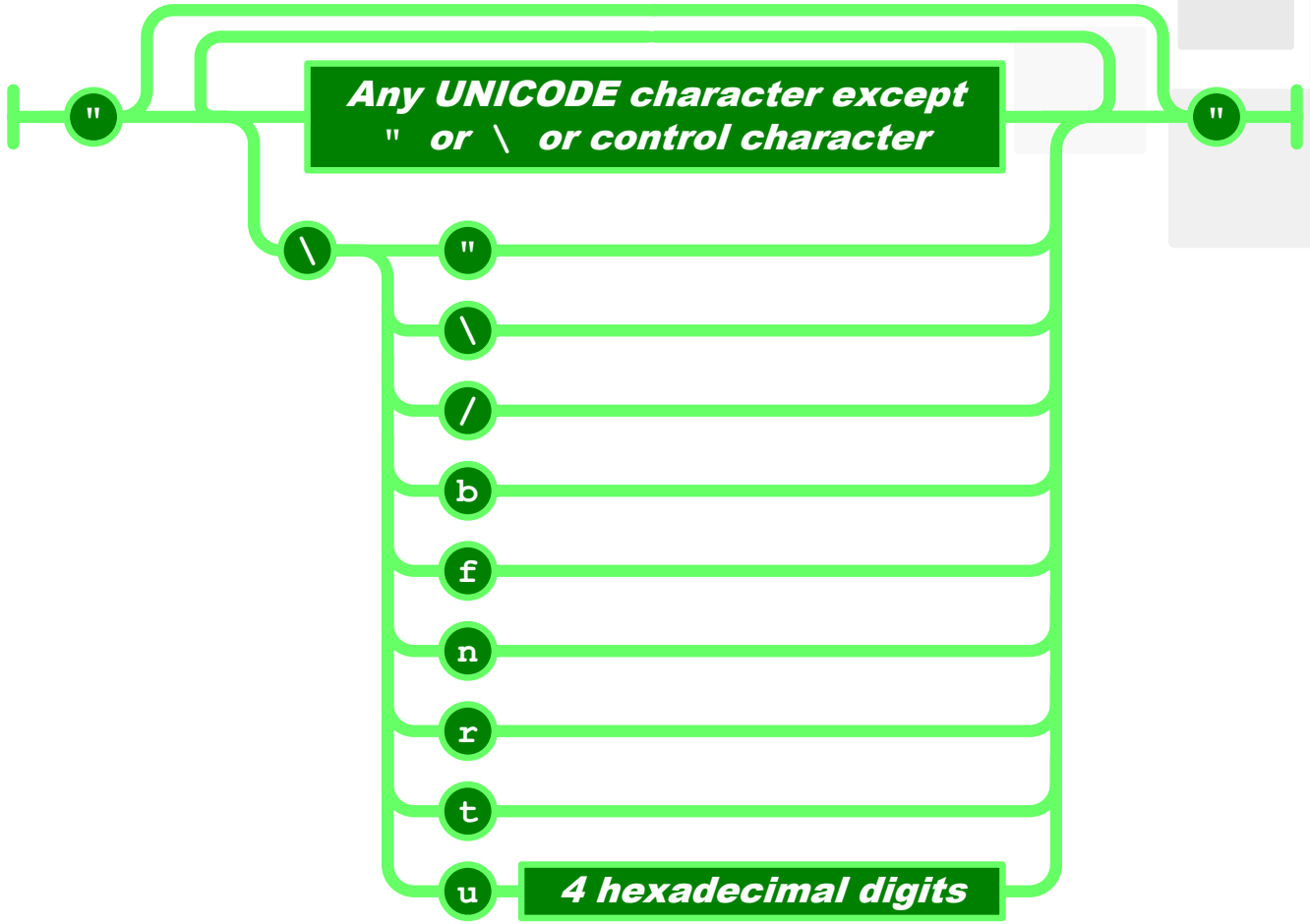
Value



Strings

- Sequence of 0 or more Unicode characters
- No separate character type
  - A character is represented as a string with a length of 1
- Wrapped in "double quotes"
- Backslash escapement

String

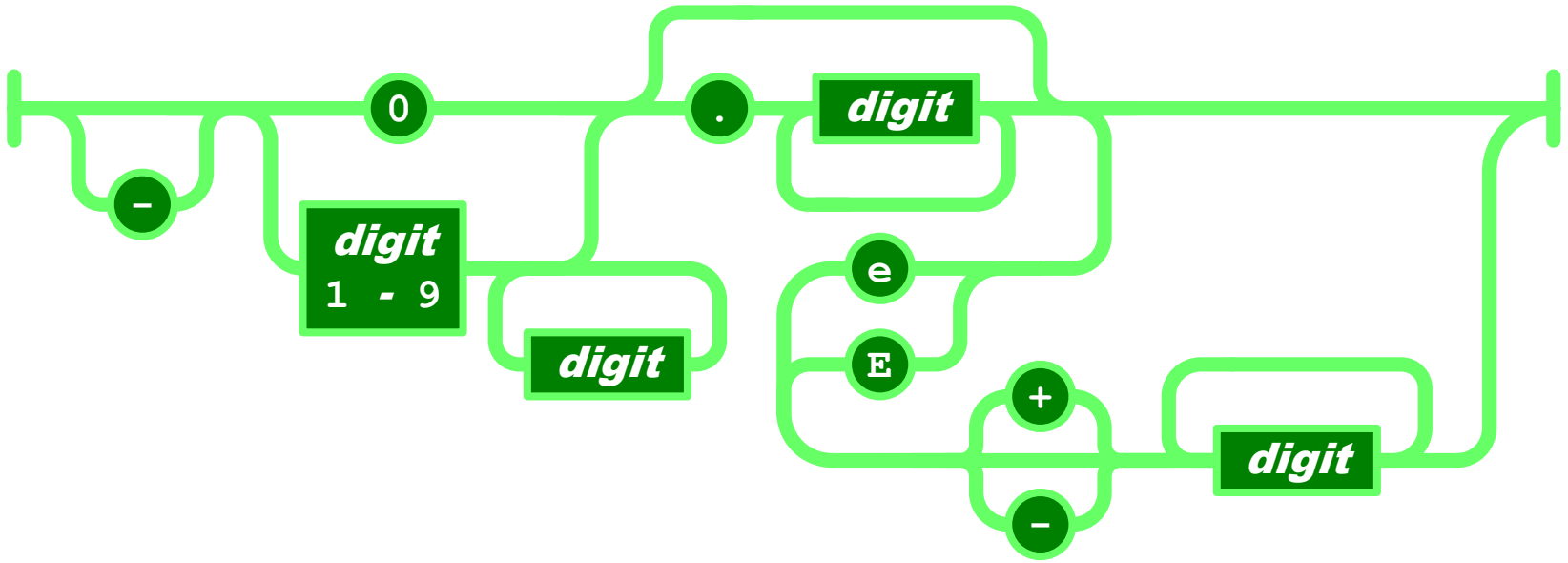

"
Any UNICODE character except
" or \ or control character
"
\
"
\
/
b
f
n
r
t
u
4 hexadecimal digits

Numbers

- Integer
- Real
- Scientific

- No octal or hex
- No **`NaN`** or **`Infinity`**
  - Use **`null`** instead

Number



Booleans

- `true`
- `false`

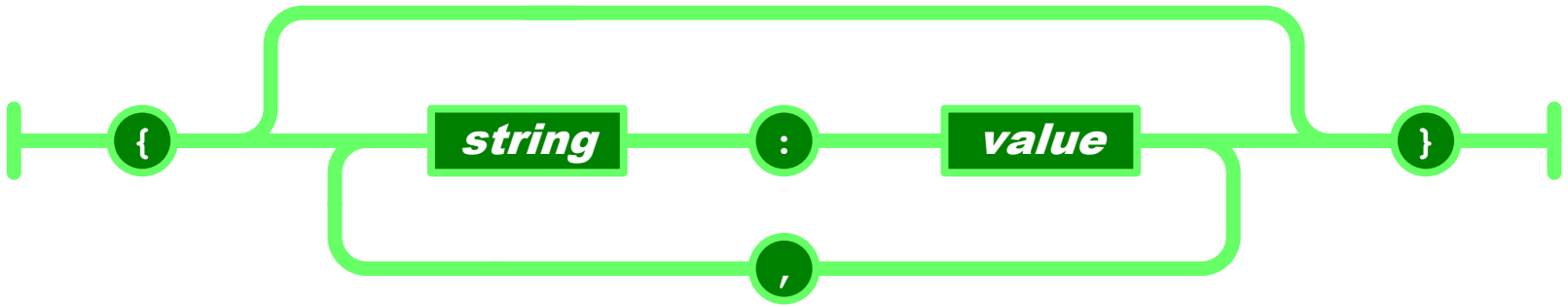
`null`

- A value that isn't anything

Object

- Objects are unordered containers of key/value pairs
- Objects are wrapped in { }
- , separates key/value pairs
- : separates keys and values
- Keys are strings
- Values are JSON values
  - struct, record, hashtable, object

Object



Object


```
{"name":"Jack B. Nimble","at large":
true,"grade":"A","level":3,
"format":{"type":"rect","width":1920,
"height":1080,"interlace":false,
"framerate":24}}
```

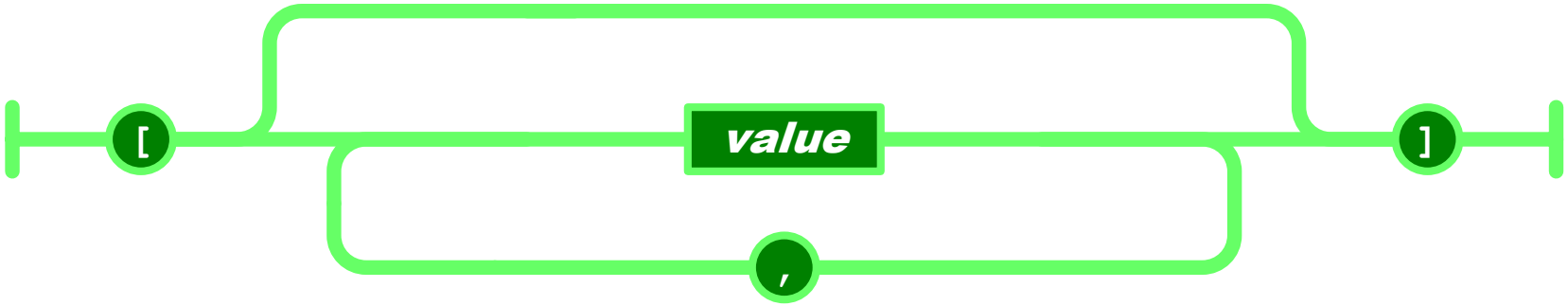
Object


```
{
    "name":      "Jack B. Nimble",
    "at large": true,
    "grade":     "A",
    "format": {
        "type":       "rect",
        "width":      1920,
        "height":     1080,
        "interlace": false,
        "framerate": 24
    }
}
```

Array

- Arrays are ordered sequences of values
- Arrays are wrapped in `[]`
- `,` separates values
- JSON does not talk about indexing.
  - An implementation can start array indexing at 0 or 1.

Array



Array

```
["Sunday", "Monday", "Tuesday", "Wednesday", "Thursday",
 "Friday", "Saturday"]
```

```
[
   • [0, -1, 0],
   • [1, 0, 0],
   • [0, 0, 1]
]
```

Arrays vs Objects

- Use objects when the key names are arbitrary strings.
- Use arrays when the key names are sequential integers.
- Don't get confused by the term Associative Array.

MIME Media Type

- `application/json`

Character Encoding

- Strictly UNICODE.
- Default: UTF-8.
- UTF-16 and UTF-32 are allowed.

Versionless

- JSON has no version number.
- No revisions to the JSON grammar are anticipated.
- JSON is very stable.

Rules

- A JSON decoder must accept all well-formed JSON text.
- A JSON decoder may also accept non-JSON text.
- A JSON encoder must only produce well-formed JSON text.
- *Be conservative in what you do, be liberal in what you accept from others.*

Supersets

- YAML is a superset of JSON.
  - A YAML decoder is a JSON decoder.
- JavaScript is a superset of JSON.
  - A JavaScript compiler is a JSON decoder.
- New programming languages based on JSON.

JSON is the X in Ajax

- JSON is the X in Ajax

JSON in Ajax

- HTML Delivery.

- JSON data is built into the page.
  - `<html>...`
  - `<script>`
  - `var data = { ... JSONdata ... };`
  - `</script>...`
  - `</html>`

JSON in Ajax

- XMLHttpRequest
  - Obtain **`responseText`**
  - Parse the **`responseText`**

```
responseData = eval(
    '(' + responseText + ')');

responseData =
    responseText.parseJSON();
```

JSON in Ajax

- Is it safe to use **`eval`** with XMLHttpRequest?
- The JSON data comes from the same server that vended the page. **`eval`** of the data is no less secure than the original html.
- If in doubt, use ***`string`*`.parseJSON`** instead of **`eval`**.

JSON in Ajax

- Secret **<iframe>**
- Request data using **form.submit** to the **<iframe>** target.
- The server sends the JSON text embedded in a script in a document.
  - **<html><head><script>**
  - **document.domain = 'penzance.com';**
  - **parent.deliver({ ... JSONtext ... });**
  - **</script></head></html>**
- The function **deliver** is passed the value.

JSON in Ajax

- Dynamic script tag hack.
- Create a script node. The **`src`** url makes the request.
- The server sends the JSON text embedded in a script.
  - **`deliver({ ... JSONtext ... });`**
- The function **`deliver`** is passed the value.
- The dynamic script tag hack is insecure.

JSONRequest

- A new facility.
- Two way data interchange between any page and any server.
- Exempt from the Same Origin Policy.
- Campaign to make a standard feature of all browsers.

JSONRequest

```
function done(requestNr, value, exception) {
    …
}

var request =
    JSONRequest.post(url, data, done);

var request =
    JSONRequest.get(url, done);
```

- No messing with headers.
- No cookies.
- No implied authentication.

JSONRequest

- Requests are transmitted in order.
- Requests can have timeouts.
- Requests can be cancelled.
- Connections are in addition to the browser's ordinary two connections per host.
- Support for asynchronous, full duplex connections.

ECMAScript Fourth Ed.

- New Methods:
  - `Object.prototype.toJSONString`
  - `String.prototype.parseJSON`
- Available now: `JSON.org/json.js`

supplant


```
var template = '<table border="{border}">' +
    '<tr><th>Last</th><td>{last}</td></tr>' +
    '<tr><th>First</th><td>{first}</td></tr>' +
    '</table>';

var data = {
    "first":  "Carl",
    "last":   "Hollywood",
    "border": 2
};

mydiv.innerHTML = template.supplant(data);
```

supplant

```
String.prototype.supplant = function (o) {
    return this.replace(/{([^{}]*)}/g,
        function (a, b) {
            var r = o[b];
            return typeof r === 'string' ?
                r : a;
        }
    );
};
```

JSONT

```
var rules = {
    self:
'<svg><{closed} stroke="{color}" points="{points}" /></svg>',
    closed: function (x) {return x ? 'polygon' : 'polyline';},
    'points[*][*]': '{$} '
};

var data = {
    "color":  "blue",
    "closed": true,
    "points": [[10,10], [20,10], [20,20], [10,20]]
};

jsonT(data, rules)

<svg><polygon stroke="blue"
     points="10 10 20 10 20 20 10 20 " /></svg>
```

http://goessner.net/articles/jsont/

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

```
function jsonT(self, rules) {
   var T = {
      output: false,
      init: function () {
         for (var rule in rules) if (rule.substr(0,4) != "self") rules["self." + rule] = rules[rule];
         return this;
      },
      apply: function(expr) {
         var trf = function (s) {
            return s.replace(/{([A-Za-z0-9_\@\.\[\]\'@\(\)]+)}/g, function ($0, $1){
               return T.processArg($1, expr);
            })
         },  x = expr.replace(/\[[0-9]+\]/g, "[*]"),  res;
         if (x in rules) {
            if (typeof(rules[x]) == "string") res = trf(rules[x]);
            else if (typeof(rules[x]) == "function") res = trf(rules[x](eval(expr)).toString());
         } else res = T.eval(expr);
         return res;
      },
      processArg: function (arg, parentExpr) {
         var expand = function (a, e) {
            return (e = a.replace(/^\$/,e)).substr(0, 4) != "self" ? ("self." + e) : e;
         },  res = "";
         T.output = true;
         if (arg.charAt(0) == "$") res = eval(arg.replace(/${([A-za-z0-9_]+)\([A-Za-z0-9_\@\.\[\]\']+\)\}/, function($0, $1, $2){
            return "rules['self." + $1 + "'](" + expand($2,parentExpr) + ")";
         }));
         else if (arg != "$") res = T.apply(expand(arg, parentExpr));
         else res = T.eval(parentExpr);
         T.output = false;
         return res;
      },
      eval: function (expr) {
         var v = eval(expr), res = "";
         if (typeof(v) != "undefined") {
            if (v instanceof Array) {
               for (var i = 0; i < v.length; i++) if (typeof(v[i]) != "undefined") res += T.apply(expr + "[" + i + "]");
```

Some features that make *it* well-suited for data transfer

- It's simultaneously human- and machine-readable format;
- It has support for Unicode, allowing almost any information in any human language to be communicated;
- The self-documenting format that describes structure and field names as well as specific values;
- The strict syntax and parsing requirements that allow the necessary parsing algorithms to remain simple, efficient, and consistent;
- The ability to represent the most general computer science data structures: records, lists and trees.

JSON Looks Like Data

- JSON's simple values are the same as used in programming languages.
- No restructuring is required: JSON's structures look like conventional programming language structures.
- JSON's object is record, struct, object, dictionary, hash, associate array...
- JSON's array is array, vector, sequence, list...

Arguments against JSON

- JSON Doesn't Have Namespaces.
- JSON Has No Validator.
- JSON Is Not Extensible.
- JSON Is Not XML.

JSON Doesn't Have Namespaces

- Every object is a namespace. Its set of keys is independent of all other objects, even exclusive of nesting.

- JSON uses context to avoid ambiguity, just as programming languages do.

Namespace

- **http://www.w3c.org/TR/REC-xml-names/**

- In this example, there are three occurrences of the name title within the markup, and the name alone clearly provides insufficient information to allow correct processing by a software module.

```
<section>
    <title>Book-Signing Event</title>
    <signing>
        <author title="Mr" name="Vikram Seth" />
        <book title="A Suitable Boy" price="$22.95" />
    </signing>
    <signing>
        <author title="Dr" name="Oliver Sacks" />
        <book title="The Island of the Color-Blind"
              price="$12.95" />
    </signing>
</section>
```

Namespace

```
{"section":
    "title": "Book-Signing Event",
    "signing": [
        {
            "author": { "title": "Mr", "name": "Vikram Seth" },
            "book": { "title": "A Suitable Boy",
                      "price": "$22.95" }
        }, {
            "author": { "title": "Dr", "name": "Oliver Sacks" },
            "book": { "title": "The Island of the Color-Blind",
                      "price": "$12.95" }
        }
    ]
}}
```

- section.title
- section.signing[0].author.title
- section.signing[1].book.title

JSON Has No Validator

- Being well-formed and valid is not the same as being correct and relevant.
- Ultimately, every application is responsible for validating its inputs. This cannot be delegated.
- A YAML validator can be used.

JSON is Not Extensible

- It does not need to be.
- It can represent any non-recurrent data structure as is.
- JSON is flexible. New fields can be added to existing structures without obsoleting existing programs.

JSON Is Not XML

- objects
- arrays
- strings
- numbers
- booleans
- **`null`**

Data Interchange

- JSON is a simple, common representation of data.
- Communication between servers and browser clients.
- Communication between peers.
- Language independent data interchange.

Why the Name?

- XML is not a good data interchange format, but it is a document standard.
- Having a standard to refer to eliminates a lot of squabbling.

Going Meta

- By adding one level of meta-encoding, JSON can be made to do the things that JSON can't do.
- Recurrent and recursive structures.
- Values beyond the ordinary base values.

Going Meta

- Simply replace the troublesome structures and values with an object which describes them.

```
{
    "$META$": meta-type,
    "value": meta-value
}
```

- Possible meta-types:

`"label"`	**Label a structure for reuse.**
`"ref"`	**Reuse a structure.**
`"class"`	**Associate a class with a structure.**
`"type"`	**Associate a special type, such as Date, with a structure.**

Browser Innovation

- During the Browser War, innovation was driven by the browser makers.
- In the Ajax Age, innovation is being driven by application developers.
- The browser makers are falling behind.

The Mashup Security Problem

- Mashups are an interesting new way to build applications.
- Mashups do not work when any of the modules or widgets contains information that is private or represents a connection which is private.

The Mashup Security Problem

- JavaScript and the DOM provide completely inadequate levels of security.
- Mashups require a security model that provides cooperation under mutual suspicion.

The Mashup Security Solution

```
<module id="NAME" href="URL" style="STYLE" />
```

- A module is like a restricted iframe. The parent script is not allowed access to the module's window object. The module's script is not allowed access to the parent's window object.

The Mashup Security Solution

```
<module id="NAME" href="URL" style="STYLE" />
```

- The module node presents a **`send`** method which allows for sending a JSON string to the module script.

- The module node can accept a **`receive`** method which allows for receiving a JSON string from the module script.

The Mashup Security Solution

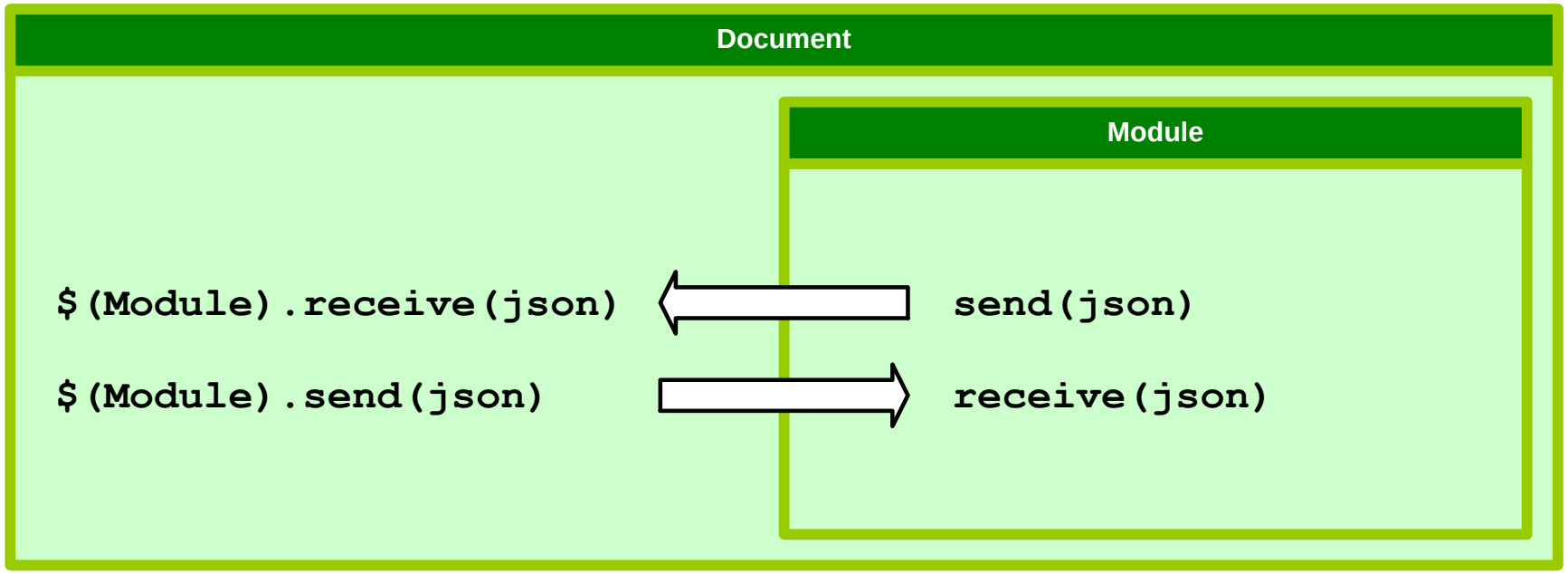
```
<module id="NAME" href="URL" style="STYLE" />
```

- Inside the module, there is a global **`send`** function which allows for sending a JSON string to the outer document's script.

- Inside the module, you can define a **`receive`** method which allows for receiving a JSON string from the outer document's script.

The Mashup Security Solution

```
<module id="NAME" href="URL" style="STYLE" />
```



The Mashup Security Solution

```
<module id="NAME" href="URL" style="STYLE" />
```

- Communiciation is permitted only through cooperating **`send`** and **`receive`** functions.
- The module is exempt from the Same Origin Policy.

The Mashup Security Solution

```
<module id="NAME" href="URL" style="STYLE" />
```

- Ask your favorite browser maker for the **<module>** tag.

XML Vs JSON

Features	JSON	XML
Full Form	JavaScript Object Notation	Extensible Markup Language
Extend	Extended from JavaScript	Extended from SGML(Standard Generalized Markup Language)
Speed	Faster in Execution	Slow compared to JSON in Execution
Arrays Usage	Uses Structured Data Excepting Arrays	Uses Structured Data Including Arrays
Application	For WebService, JSON is better.	For Configuration, XML is better
Example	{ "company": AcadGild, "course": "Android", "price": 16999 }	<education> <company>AcadGild</company> <course>Android</course> <price>16999</price> </education>
Comments	Not Supports Comments	Supports Comments
NameSpace	No support for Namespaces	Support Namespaces

XML Vs JSON

SYNERGETICS GET IT RIGHT | Microsoft Partner Gold Cloud Platform Silver Learning

Features	JSON	XML
APIs	Facebook Graph API, Google Maps API, Twitter API, Pinterest API,Reddit API	Amazon Products Advertising API
Performance	Good Compared to XML	Not as good when Compared to JSON
Purpose	Structured Data Interchange	Document Markup
Database Support	MongoDB, CouchDB, eXistDB, BaseX, Oracle Database, PostgreSQL	MySQL, IBM DB2, Mocrosoft SQL Server, BaseX, MarkLogics
Light Weight	Lighter than XML	Less Light compared to JSON
Schema	For Description and DataType and Structure Validation	For DataType, Structure Validation, It also makes possible to create new DataTypes
Easy to Manipulate	Yes	No
Security	More Secure	Less Secure

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Q & A

Contact: your synergetics email id

SYNERGETICS GET IT RIGHT | Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You