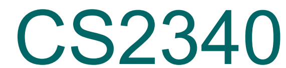
Specialized Parsers – XML and YAXO



XML Vocabulary

- XML – Extensible Markup Language
  - Designed to describe data and focus on what the data is
  - Vs. HTML – display data and focus on how data looks.
  - It doesn’t do anything, it describes data via tags and values.
  - Tutorial: http://www.w3schools.com/xml/xml_whatis.asp

XML

- Must have open/close tags
- Must be properly nested
- Always have a root element
  - Parsed document forms a tree structure
- Can be commented
  - <!-- This is a comment -->
- Is case sensitive <Name> != <name>
- Can have attributes <person sex=“male”>

Sample XML Description

<CustomerList>
<CompanyName>Extroon Incorporated</CompanyName>
<CompanyPhone>770-555-1212</CompanyPhone>

<customer>
<name>Bob Waters</name>
<id>126423</id>
<addr> 1313 MockingBird Lane </addr>
</customer>
<customer>
<name>Sally Smith</name>
<id>559382</id>
<addr> 1212 Sunnyvale Retirement Home</addr>
</customer>
</CustomerList>

Well-Formed vs. Valid XML

- Just because it is well-formed (syntactically correct) doesn’t mean the data is correct.
- Need to specify what the data is supposed to look like for the information to be valid.
- Can use either schemas or Document Type Definition (DTD).

Sample DTD

```
<!DOCTYPE CustomerList [
  <!ELEMENT CompanyName  (#PCDATA)>
  <!ELEMENT CompanyPhone (#PCDATA)>
  <!ELEMENT customers (customer+)>
  <!ELEMENT customer (name,id,addr)>
  <!ELEMENT name (#PCDATA)>
  <!ELEMENT id  (#PCDATA)>
  <!ELEMENT addr (#PCDATA)>
]>
```

Sample Schema

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.cc.gatech.edu/cs2340"
  xmlns="http://www.cc.gatech.edu/cs2340"
  elementFormDefault="qualified">
<xs:element name=“CustomerList">
   <xs:complexType>
     <xs:sequence>
     <xs:element name=“CompanyName"  type="xs:string"/>
     <xs:element name=“CompanyPhone“  type="xs:string"/>
```

Schema Continued

```
<xs:element name=“customer" />
   <xs:complexType>
      <xs:sequence>
        <xs:element name=“name” type=“xs:string”/>
        <xs:element name=“id" type="xs:string"/>
        <xs:element name=“addr" type="xs:string"/>
    </xs:sequence>
</xs:complexType>
</xs:element>
</xs:complexType>
</xs:element>
</xs:schema>
```

Parsing XML

- You could do it yourself…..
- DOM – Document Object Model
  - Tree-Based
  - Parse entire doc into tree, then query
  - www.w3.org/DOM
- SAX – Simple API for XML
  - Event-Based
  - Report parsing events and handle as they happen
  - www.saxproject.org

SAX Example

```
<?xml version="1.0"?>
<doc>
<para>Hello, world!</para>
</doc>
```

start document
start element: doc
start element: para
characters: Hello, world!
end element: para
end element: doc
end document

YAXO - SAX

- Override the class SaxHandler
- Override as necessary the messages:
  - startDocument
  - endDocument
  - startElement: aName attributeList: attributes
  - endElement: aName
  - characters: aString

Some Code

```
SAXHandler subclass: #MySampleSaxThing
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''
    category: 'XML-Parser'
```

More Code

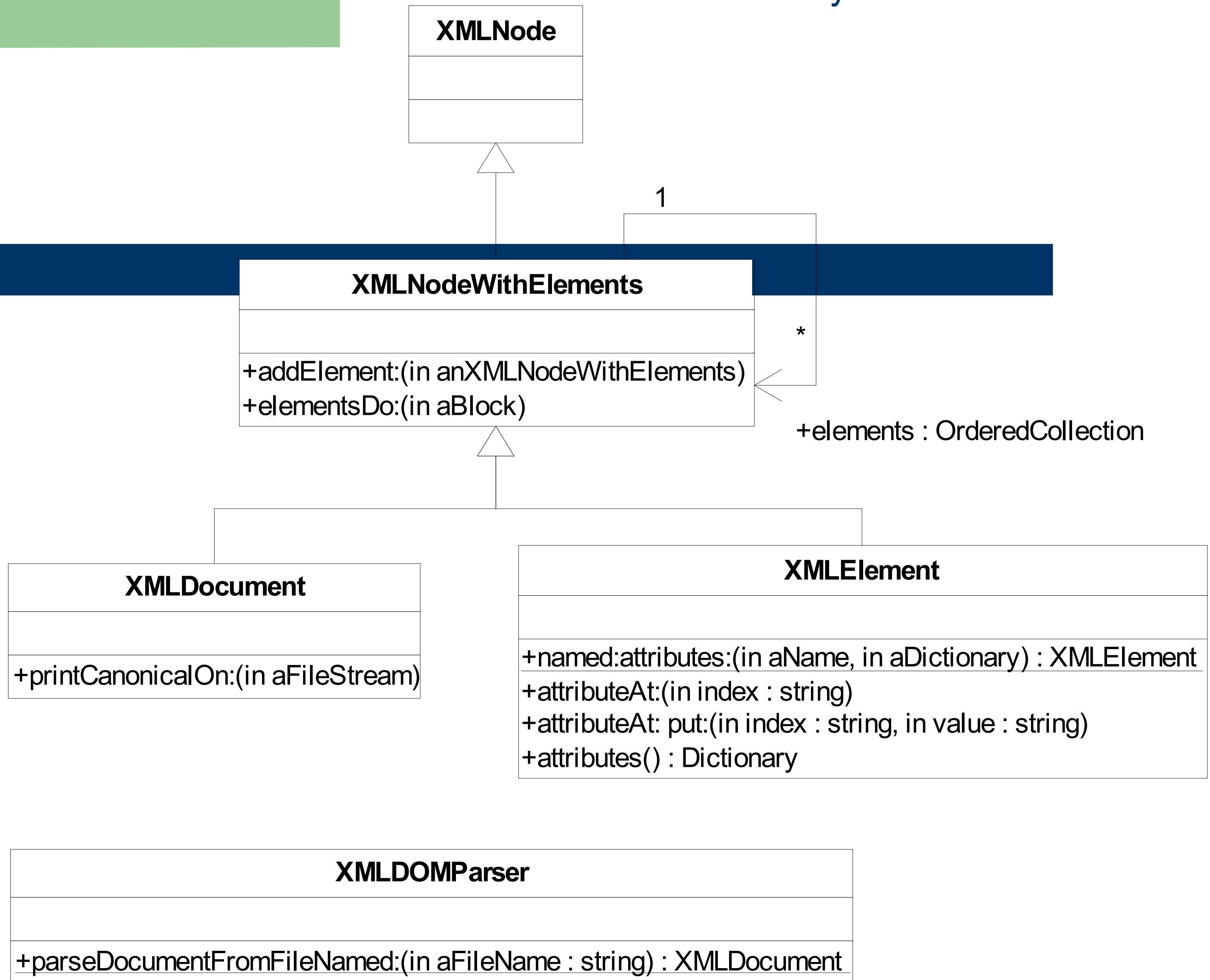
```
startElement: elementName attributeList: attributeList
    Transcript show: 'Processing Element: ';
        show: elementName;
        cr.


characters: aString
    Transcript show: 'Got characters: ';
        show: aString;
        cr
```

Starting it Up

MySampleSaxThing parseDocumentFromFileNamed: 'sample.xml'

By: Jonathan D'Andries



For DOM, we get model first

```
f:=FileStream fileNamed: 'samplexml2.xml'.
x:=XMLDOMParser parseDocumentFrom: f.
```

X now contains an object of type XMLDocument

Note that DOM uses SAX to build the in-memory tree.

Getting elements out

document elements returns an OrderedCollection of elements in the document

(document elements) at: 1 gets us the root XMLElement
document topElement
document elementAt: 'rootElementName'

We can then use the **firstTagNamed: #customer**

We can also use **tagsNamed: #customer do: [aBlock]** to execute the same code for each tag block.

Playing with DOM Directly

```
f:=FileStream fileNamed: 'samplexml2.xml'.
x:=XMLDOMParser parseDocumentFrom: f.
f close.
e:=x elements.
n:=e at: 1.
n name.
n tag.
n contentString.
c:=n firstTagNamed: #customer.

n tagsNamed: #customer do: [ :i | Transcript show: i ; cr. ]
```

Writing a Custom Class -- Looking up specific elements

```
lookup: aName
        | top ele |
        top:=document topElement.
        top tagsNamed: #customer
                do: [:tag |
                    ele:=tag firstTagNamed: #name.
                    Transcript show: 'Examining: "';
                               show: ele characterData;
                               show: '"';
                                cr.
                     ele characterData = aName
                        ifTrue: [Transcript show: 'Found the entry'.
                                 self showData: tag.
                                ^ aName]].
        Transcript show: 'Entry Not Found'.
        ^ 'No such customer'
```

Running Example

```
x:=MyDomThing new.
x openOn: 'samplexml2.xml'.
x showElements: x topElement.
x lookupName: 'Bob Waters'.
```

Making document from scratch

```
createHeader
    | aTopElement |
    document _ XMLDocument new.
    aTopElement _ XMLElement named: 'CustomerList‘
          attributes: Dictionary new.
    aTopElement addElement: (self makeSubElement:
          'CompanyName' content: 'FooBar Inc').
    aTopElement addElement: (self makeSubElement:
          'CompanyPhone' content: '990-555-1345').
    document addElement: aTopElement
```

Making a string subelement

```
makeSubElement: aTagName content: aStringContent
    | anXMLElement |
    anXMLElement := XMLElement named: aTagName
        attributes: Dictionary new.
    anXMLElement
        addContent: (XMLStringNode string: aStringContent).
    ^ anXMLElement
```

Making a subgroup

```
createCustomer: aName id: anId status: aStatus
  | top aCustElement |
  top := document topElement.
  aCustElement := XMLElement named: 'Customer' attributes:
                    Dictionary new.
  aCustElement attributeAt: 'status' put: aStatus.
  aCustElement addElement: (self makeSubElement: 'name‘
                        content: aName).
  aCustElement addElement:
                (self makeSubElement: 'id' content: anId).
  top addElement: aCustElement
```

Running Example

```
x_MyXMLWriter new.
x writeit: 'testxml3.xml'
```