Introduction to XML
Section 1 - Introduction

```
<course title='Introduction to XML'>
  <section num='1' title='Introduction'/>
  <section num='2' title='Basics'/>
  <section num='3' title='Namespaces'/>
  <section num='4' title='Schemas'/>
</course>
```

Course Description

- This course provides a technical introduction to XML
- There are hands-on lab exercises that allow you to practice what we discuss in class
- It is thorough in its discussion of the most important components of XML
- It will not cover those XML constructs that are rarely used in practice

Course Objectives

- Understand what XML is, why it is useful, and how it is used
- Understand the rules of XML and use the building blocks of XML to create well-formed XML documents
- Create schemas for XML documents and understand how an XML document is validated against a schema
- Understand XML namespaces

Course Sections

- Section 1 - **Introduction to XML**
- Section 2 - **XML Basics**
- Section 3 - **Namespaces**
- Section 4 - **Schemas**

- Section Outline -

- What is XML?
- Origins of XML
- Uses of XML

What is XML?

Yet Another Markup Language

- XML Defined
- Benefits of Using XML
- Comparison to HTML

What is XML?

- XML stands for ***eXtensible Markup Language***

- XML is a ***meta-markup language*** for representing **data**
  - XML ***documents*** therefore contain markup and data **--** the markup is generally in the form of ***tags***
  - Both markup and data are in plain text

- XML is **extensible** because there is no predefined set of tags
  - Instead, you create your own tags to represent your own data
  - This is why we call it a meta-markup language instead of a markup language (like HTML, which has a fixed set of tags)

Benefits of XML - Example XML Document

- Without knowing anything about XML, you can probably figure out what this XML document is about

```
<?xml version='1.0'?>

<customer ID='67183625'>
  <name>     Leanne Ross          </name>
  <street>   1475 Cedar Avenue    </street>
  <city>     Fargo                </city>
  <state>    ND                   </state>
  <zipcode>  58103                </zipcode>
</customer>
```

- Okay, so it holds customer data
  - But how shall this information be displayed to a human?
  - **XML is not about format or display --** what other markup language can you think of that **is** concerned with format and display?

Benefits of XML

- XML documents describe the data that they contain
  - To contrast, what is the following flat file describing?

```
67183625,Leanne Ross,1475 Cedar Avenue,Fargo,ND,58103
```

  - What about this one?

```
CD516/90125/Yes/1983-10-16/11.97/11.97
```

- XML is platform-neutral, standardized, and widely adopted as a mechanism for representing data
  - There is a set of rules that apply to XML documents
  - Since the data format is standardized, heterogeneous systems can exchange XML data without knowing anything about each other
  - We thus consider XML documents to be ***portable***

Comparison of XML to HTML

- XML and HTML serve different purposes
  - They are not competitors, but rather cooperators

- HTML describes how information should be **displayed**
  - It says nothing about the data, just how to display it
  - What do the numbers mean in this HTML fragment?

```
<table border=1>
  <tr><td>   12    </td><td>  35    </td></tr>
  <tr><td><b>10</b></td><td><b>0</b></td></tr>
  <tr><td>   6     </td><td>   9    </td></tr>
</table>
```

- XML **describes** that **information** in the first place
  - But says nothing about how to display it

Origins of XML

Not Invented from Scratch

- SGML
- The XML 1.0 Recommendation and the W3C
- XML Design Goals
- Related Standards

Where Did XML Come From?

- XML is a subset of SGML - ***Standard Generalized Markup Language***
  - SGML has been around since the 1980s, and was designed to encode text documents in a portable, self-describing way
  - It is very complex and "large"
  - It achieved some success in the government and aerospace sectors, and other industries which needed a mechanism to manage massive amounts of documentation (sometimes measured in "shelf-feet")
    - In other words, the "User's Guide" for an F-16 aircraft might require two 6-foot shelves of bookcase storage -- that would be 12 shelf-feet!
- SGML's biggest success is HTML, which is an application of SGML for the distribution and rendering of Web pages
  - HTML is simply a markup vocabulary defined via SGML

Why Not Just Use SGML?

- The main problems are its complexity and its design goals
  - Some SGML features are redundant, and some are too complex to be easily implemented correctly in software
    - SGML software vendors left out the parts not needed by their customers
    - What resulted were incompatible SGML implementations
  - It was designed to represent and manage massive amounts of documentation
  - Document **management** was as important as document representation
  - Primarily aimed at books, technical manuals, etc.

- We don't need (or want!) this for data transfer over a network
  - A purchase order would not require 12 shelf-feet of paper storage

Then Why Derive XML from SGML?

- Because the basic idea is great - extensibility and portability
  - Create your own markup vocabularies for your own data
  - But remove the complicated stuff that wasn't needed anyway
    - Some argue that the XML designers could have scaled it down even more
- Because markup (e.g., HTML) was well understood
  - XML was developed between 1996-98 -- basing it on something that many people already knew would facilitate its adoption
- Because software and tools already existed
  - Since XML is a strict subset of SGML, just about any SGML software could already work with XML documents
  - Again, the XML developers sought to facilitate its quick and easy adoption

The XML 1.0 Recommendation

- The World Wide Web Consortium (**W3C**) finalized the XML specification in February, 1998
  - W3C specifications are called ***Recommendations*** and are developed collaboratively by industry participants
  - The XML Recommendation is at ***http://www.w3.org/TR/REC-xml***

- Note what XML is **not**:
  - A transport protocol -- HTTP, etc. can be used to transport XML data
  - A programming language
  - A formatting language -- that is left to things like HTML and XSL (***eXtensible Stylesheet Language***)

Related XML Standards

- **XML Namespaces** - a mechanism for qualifying element and attribute names in XML documents
  - To prevent name collisions
  - Java packages are used for this purpose, as well

- **XLink** - a syntax for creating links between XML documents
  - Like an `<a href="">` in HTML, but with more functionality

- **XML Schema** - an XML vocabulary for creating ***schemas*** for XML documents
  - A schema defines an XML document's structure -- what things are required to be in it, in what order, etc.
  - The W3C XML Schema language is a replacement for SGML's DTD (Document Type Definition) syntax

Related XML Standards

- **XPath** - a query language for XML documents
  - Think of it as the SQL of XML
  - Used in conjunction with other technologies, like XSLT

- **XSL** (***eXtensible Stylesheet Language***) - a language for expressing stylesheets -- it consists of two parts:
  - **XSLT** (***XSL Transformations***) - an XML vocabulary for transforming XML documents into other forms
  - Including XML, HTML, etc.

  - **XSL-FO** (***XSL Formatting Objects***) - an XML vocabulary for specifying formatting semantics
  - For print media, such as PDF

Uses of XML

XML is Everywhere (or Will Be)

- XML's Reason for Being
- Examples of its Use

The Underlying Theme of XML

- XML's main reason for existence is to **represent data**
  - In a portable way
- This "block of data" can be retrieved, modified, stored, etc.
  - By heterogeneous applications -- Application A can generate XML and Application B can read it
  - And Applications A and B don't even know about each other
- Or it can be exchanged between systems, across a network
  - Systems that are written in different languages, running on different platforms, and are not coupled to one another
  - This **ubiquitous data interchange format** is probably the most important benefit that XML provides

Configuration Files

- You may have already seen some XML in configuration files
  - Since XML is a simple, self-describing way to represent data, software applications are increasingly using it for configuration
  - This way, you don't have to learn each vendor's specific configuration file syntax and format

- The Sun J2EE specifications all use XML for the "deployment descriptors" (which are basically configuration files)
  - Web applications are configured with *web.xml*
  - EJBs are configured with *ejb-jar.xml* and J2EE applications are configured with *application.xml*

Business Examples

- **ebXML** is a B2B XML-based standard
  - The intent of ebXML is to create a world-wide standard for electronic commerce, built around standardized XML documents
- **Web Services** use XML as the data format in transmitted messages to/from service providers
- **SOAP** (***Simple Object Access Protocol***) specifies a message format in XML
  - A SOAP message is simply an XML document that uses the SOAP vocabulary

Other Industry Examples

- **MathML** - Mathematical Markup Language
- **CML** - Chemical Markup Language
- **SVG** - Scalable Vector Graphics
- These "languages" are really just XML vocabularies that are specific to an industry or technology
  - An XML vocabulary is an agreed-upon set of names and document structure, i.e., how the names are used to create a document
  - For example, MathML might use *<equation>*, CML might use *<molecule>*, SVG might use *<animate>*, etc.

Resources

- **W3C** - World Wide Web Consortium
  - ***http://www.w3.org***
- **OASIS** - Organization for the Advancement of Structured Information Standards
  - ***http://www.oasis-open.org***
- **XML.org** - an industry Web portal formed by OASIS
  - ***http://www.xml.org***

- Lab 1.1 – Setting up the Environment -

- **Purpose**: To familiarize you with the lab environment
  - Become familiar with the lab structure and Eclipse
  - Start up Eclipse, and creating/using a Eclipse project
- You will also get a brief introduction to Eclipse's capabilities
  - To learn enough to be able to work comfortably with Eclipse
  - It is not an in-depth coverage
  - We'll start Eclipse, make a simple project, and work with XML files
- **Builds on previous labs**: None
- **Approximate Time**: 30-40 minutes

Extract the Lab Setup Zip File

- To set up the labs, you'll need the course setup zip file *
  - It has a name like: ***XML_LabSetup_20100111.zip***
- Our base working directory for this part of the course will be **C:\StudentWork\XMLIntro**
  - This directory will be created when we extract the Setup zip
  - It includes a directory structure and files (e.g., Java files, XML files, other files) that will be needed in the labs
  - All instructions assume that this zip file is extracted to C:\. **If you choose a different directory, please adjust accordingly**

Tasks to Perform

- **Unzip** the lab setup file to **C:**
  - This will create the directory structure, described in the next slide, containing files that you will need for doing the labs

General Instructions

- **Lab Directory Structure**: Your labs will be in the directory: **StudentWork\XMLIntro\workspace**
- The root lab directory where you will do your work for this lab is: **C:\StudentWork\XMLIntro\workspace\Lab01.1**
  - This directory already exists in your workspace – you'll do your work in this directory
  - Generally, the files you work on for a lab will be under the root directory (and instructions are given relative to this directory)
- Detailed instructions are included in this lab
  - They include complete instructions for working in the **Eclipse** environment, as well as details about the lab requirements
- Subsequent labs require you to do the same thing as this lab to build/run, so they include fewer detailed instructions

The Eclipse Development Environment

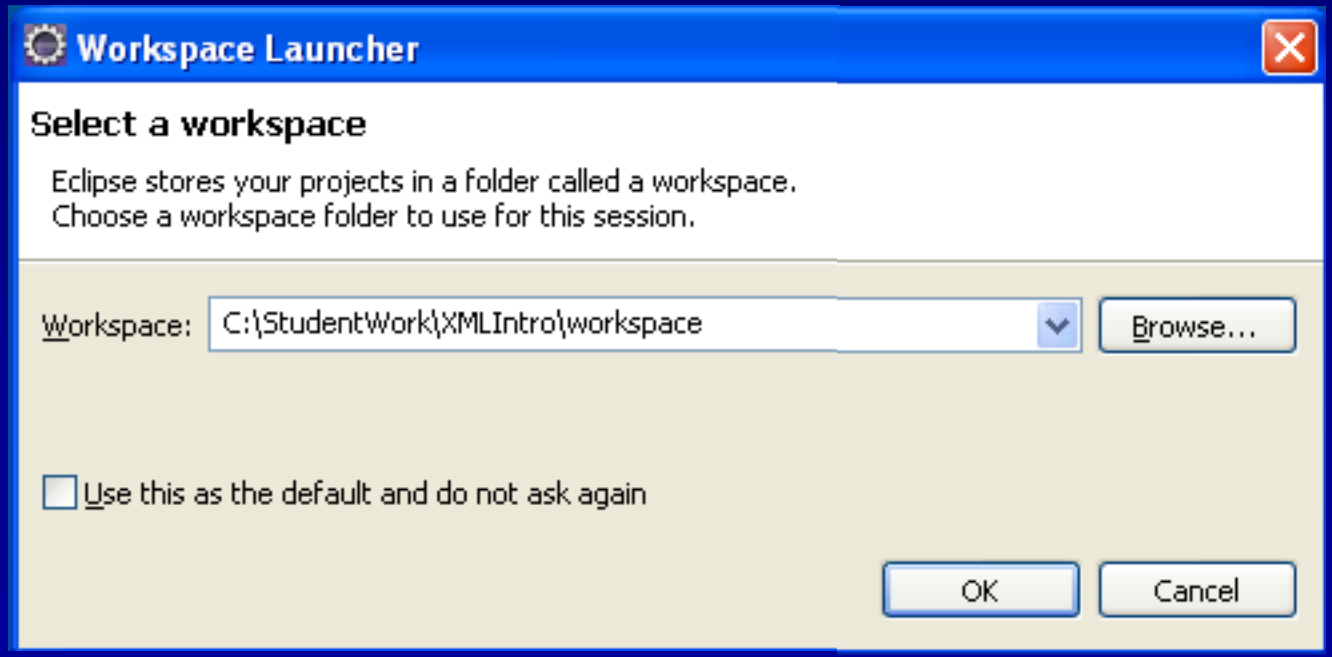
- **Eclipse** is an open source platform for building integrated development environments (IDEs)
  - Used mainly for Java development
  - Can be extended via plugins to create applications useful in many areas (e.g. C# programming)
  - **http://www.eclipse.org** is the main website
- The remainder of this lab gives detailed instructions on using Eclipse to run the labs
  - Starting it, creating and configuring projects, etc.
- The other labs in the course include fewer specific details regarding Eclipse – they may just say build/run as previously
  - For these labs, you should use the same procedures to build/run as in this lab
  - Refer back to these lab instructions as needed

The Eclipse Development Environment

Lab

Tasks to Perform

- To launch eclipse, go to **c:\eclipse** and run **eclipse.exe**
  - A dialog box should appear prompting for workbench location
  - Set the workbench location to **C:\StudentWork\XMLIntro\workspace**
  - If a different default Workbench location is set, change it
  - Click **OK**
  - In the window that opens, click the **Workbench** icon (see notes)

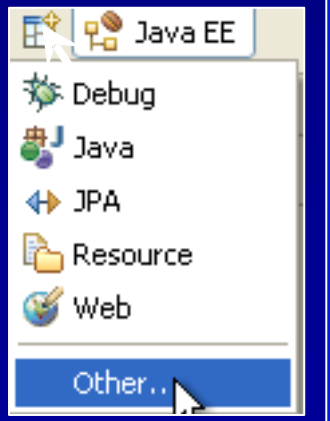


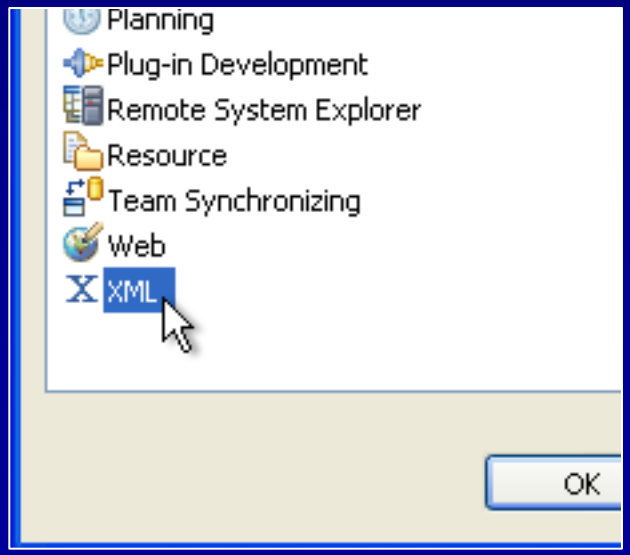
Workbench and XML Perspective

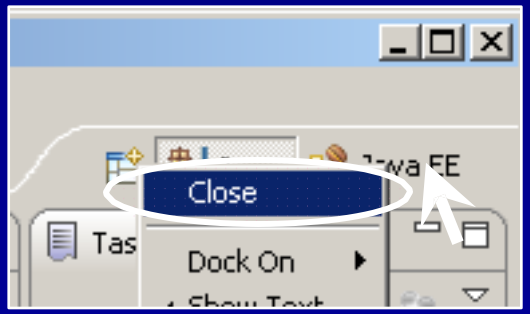
Lab

Tasks to Perform

- You'll likely be in a Java EE perspective *
- If in a **Java EE** perspective, **open an XML one** by clicking the Perspective icon at the top right of the Workbench, and select Other, then XML (as shown below left)
  - Close the Java EE perspective by right clicking its icon, and selecting close (as shown below right)
  - If you were in am XML perspective, then just remain in it





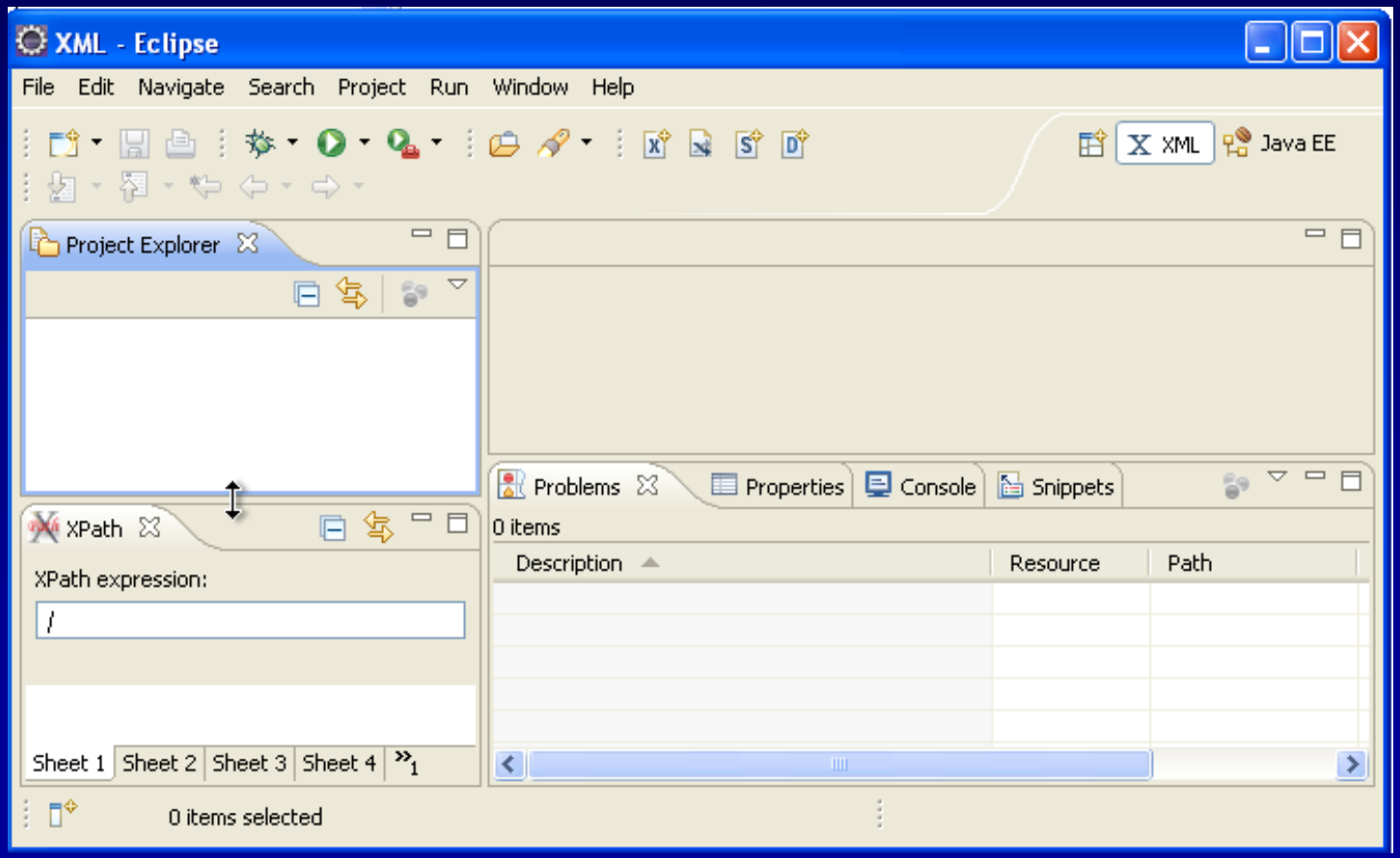


Unclutter the Workbench

Lab

Tasks to Perform

- Let's unclutter the Perspective by closing some views
  - Close the Outline and Templates views (click on the X)
  - You can save this as the default if you want (see note)



Create a Project for our Lab

Lab

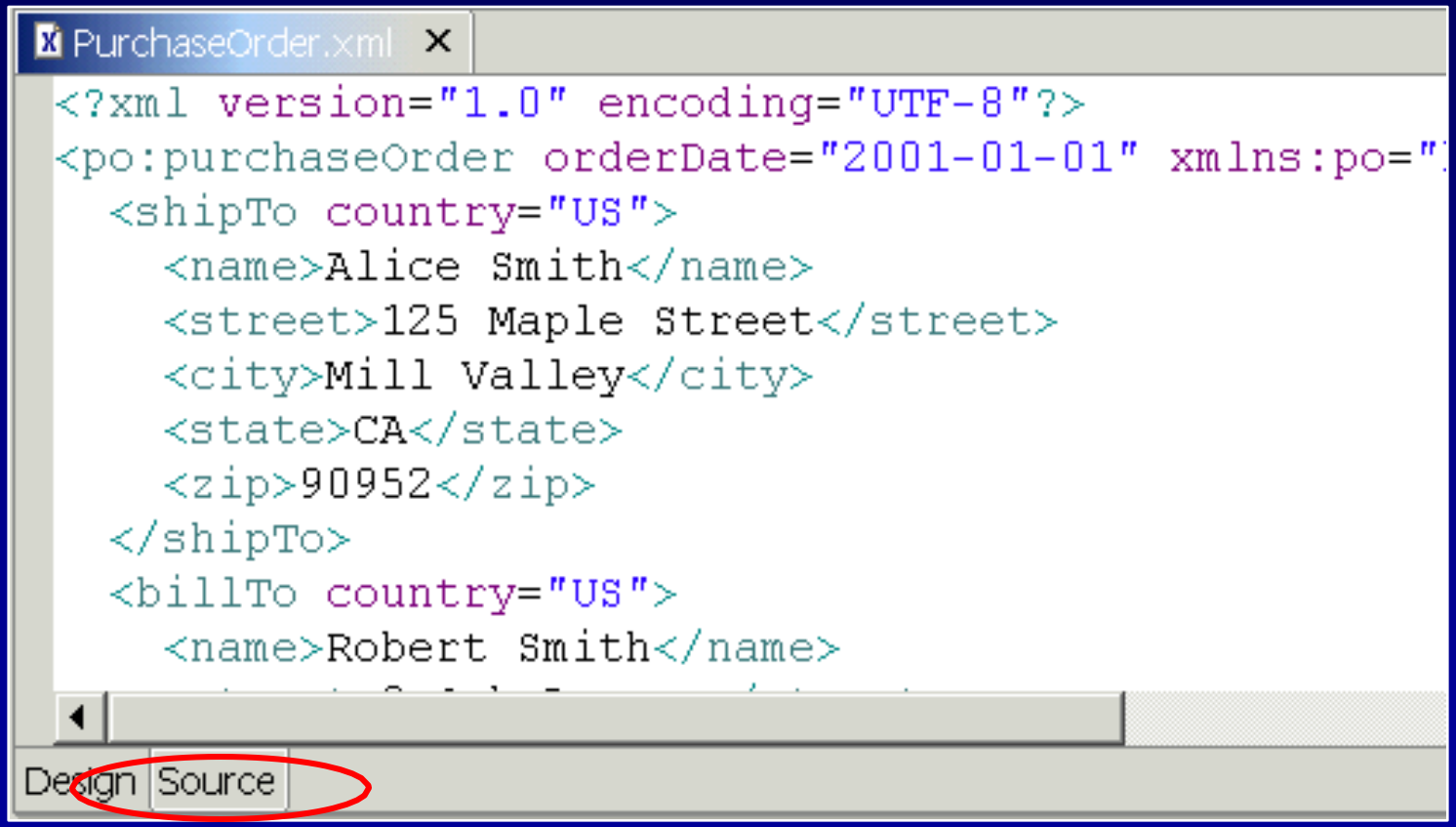
Tasks to Perform

- Create a **Project**
  - To create a new Project, use the menu item: **File | New | Project | General| Project** (see notes)
  - Call the project **Lab01.1**
  - Eclipse will then automatically set the project directory to *Lab01.1*
  - Click **Finish**
- Create a new XML file within the project you just created
  - There are multiple ways to do this – we mention one way here
  - Right click on the Lab01.1 project icon in **Project Explorer** and select **New | XML** to create a new XML file
  - Call the file *order.xml*, click **Next**, and in the next dialog, choose **Create XML file from an XML template** *
  - Click **Finish** – this will create and open the XML file

Editors

Lab

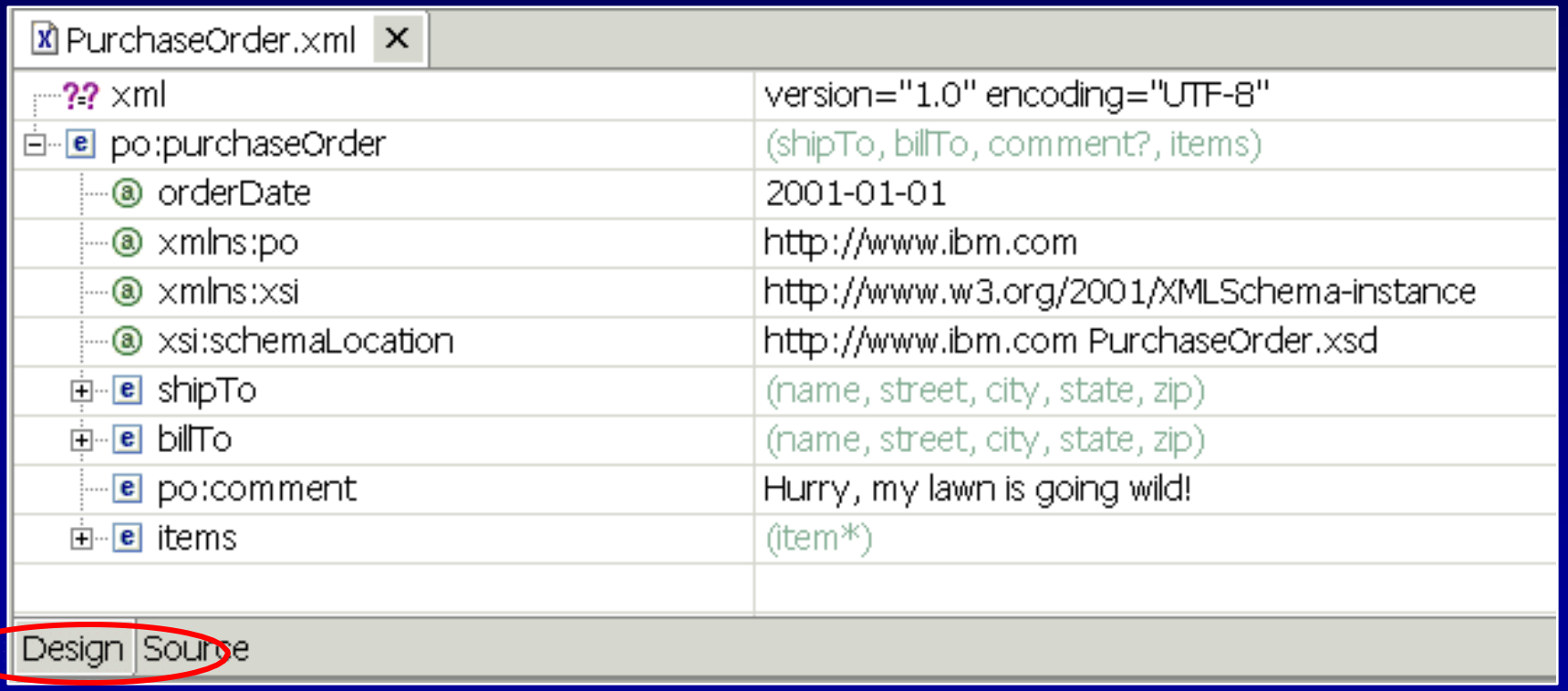
- There is a source editor like this one for a .xml file for all character files. (.java, .jsp, .html, etc.)
  - This is seen in the **Source** tab of the editor



Editors

Lab

- There are design editors like this one for an xml file and for many other types of files. (JSP, HTML etc.)
  - This is seen in the **Design** tab of the editor

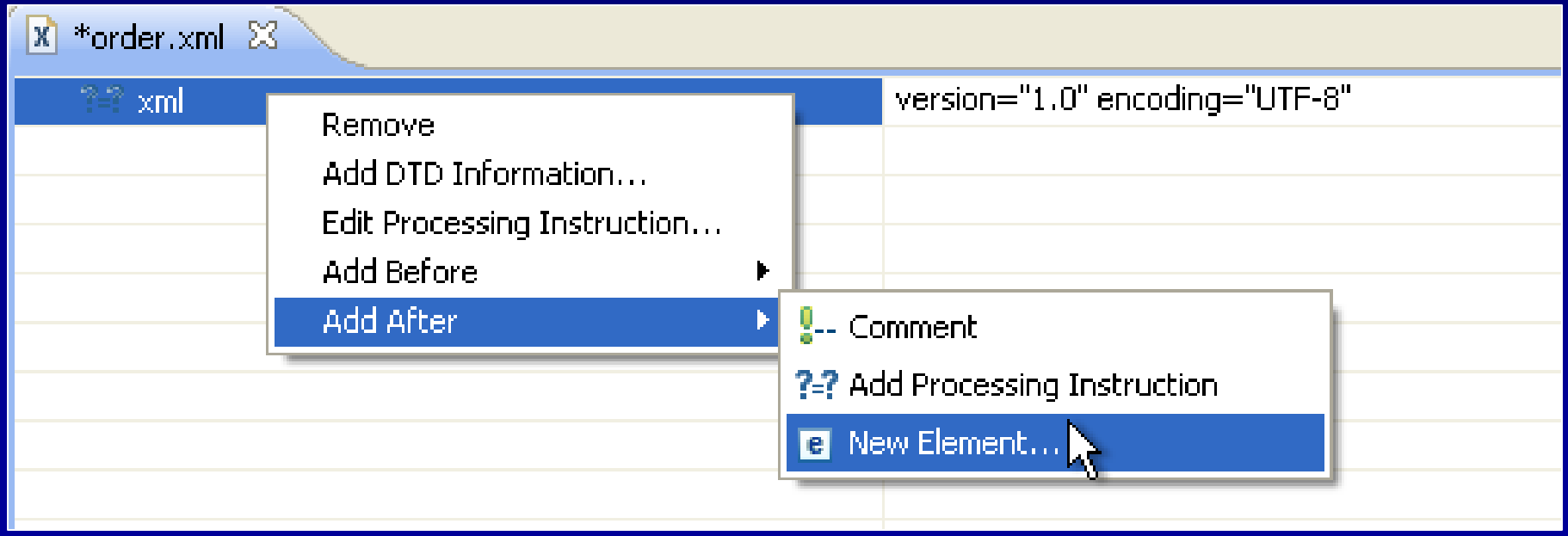


Add in an Element

Lab

Tasks to Perform

- Add in an "order" element using the design view
  - Right click on the XML declaration, choose **Add After -> New Element**
  - Call the element **order**
  - Look at the document in the design and source views
  - You can also type the element directly in the source view



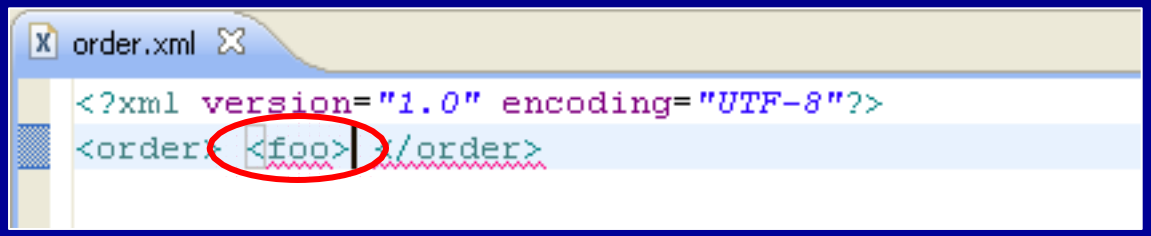
Errors and The Task View

Lab

- Let's add an error in the XML file
  - We'll then validate the file, and see that Eclipse can find XML errors

Tasks to Perform

- View the file in **Source** view, and add a ***<foo>*** tag with no matching end tag, as shown below
  - This is not valid XML (we'll talk about what that means later)
  - You'll have to remove the end tag manually, because Eclipse automatically adds it in when you create a new element
  - **Save** the file

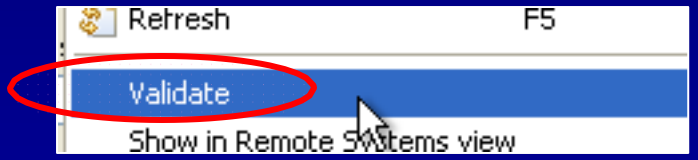


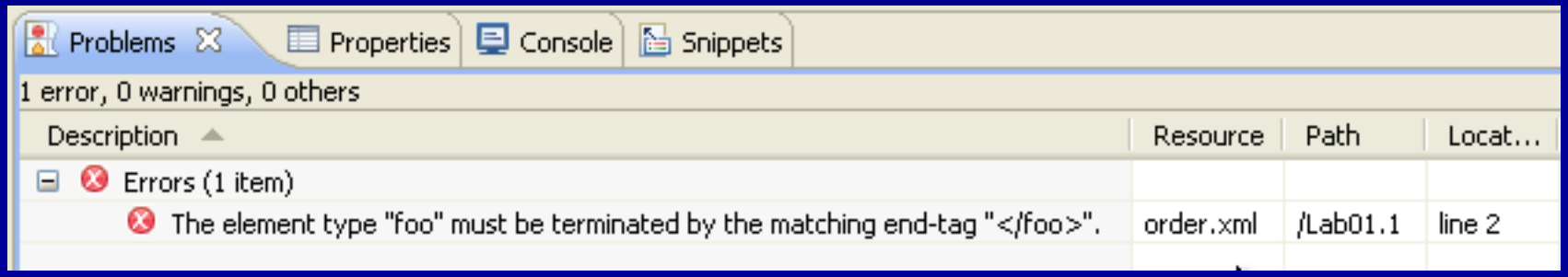
Validating a file

Lab

Tasks to Perform

- Validation will check if the file is good (well formed) XML
- **Validate** the file by right clicking on it in Project Explorer, and selecting Validate (see image below)
  - This will check the document for well formedness
    - The error should show up in the Problems view (see bottom image)
- The rest of this lab describes the structure of Eclipse, for those that haven't used it





Important Notes for Using Eclipse

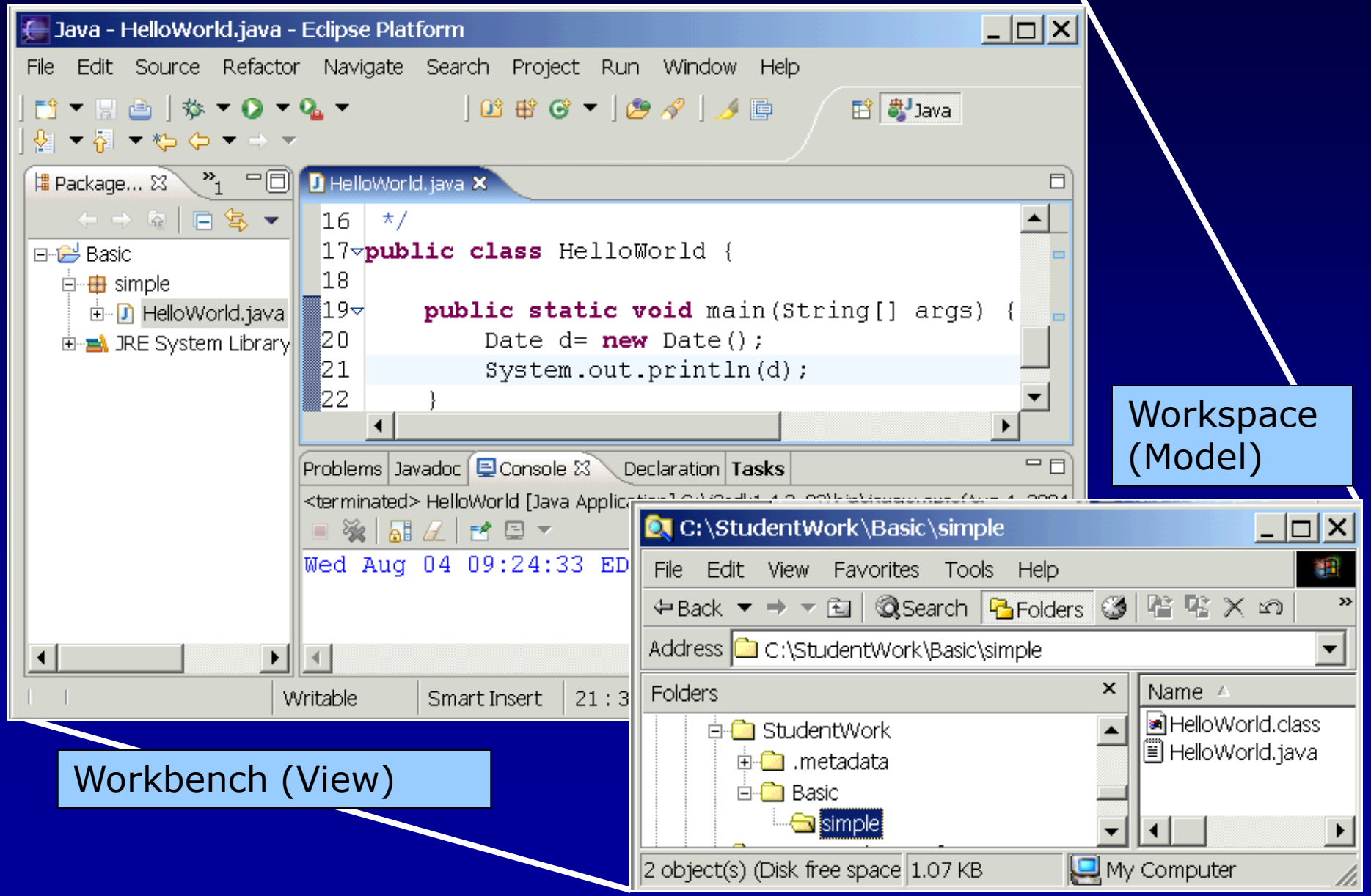
- Each lab that has a **separate lab directory** will require you to create **a new Eclipse project**
  - Sometimes several labs are done one directory, in which case you will use the same project for all of them
- If you **COPY** files (e.g. from the lab setup) into a project directory using Windows Explorer you need to **Refresh** it (Right click on the project, select **Refresh**)
  - You can also copy directly into an Eclipse view – in which case you don't need to refresh
- For anyone not familiar with Eclipse, the next few slides give a (very) brief overview of how Eclipse is structured
  - There is **nothing you need to do** in those slides – they are for information purposes only

Important Notes for Using Eclipse

- Eclipse products have two fundamental layers
  - The **Workspace** – files, packages, projects, resource connections, configuration properties
  - The **Workbench** – editors, views, and perspectives
- The Workbench sits on top of the Workspace and provides visual artifacts that allow you to access and manipulate various aspects of the underlying resources, such as:
  - **Editor** – A component that allows a developer to interact with and modify the contents of a file.
  - **View** – A component that exposes meta-data about the currently selected resource.
  - **Perspective** – A grouping of related editors and views that are relevant to a particular task and/or role.
- You can have multiple perspectives open to provide access to different aspects of the underlying resources

Workbench and Workspace

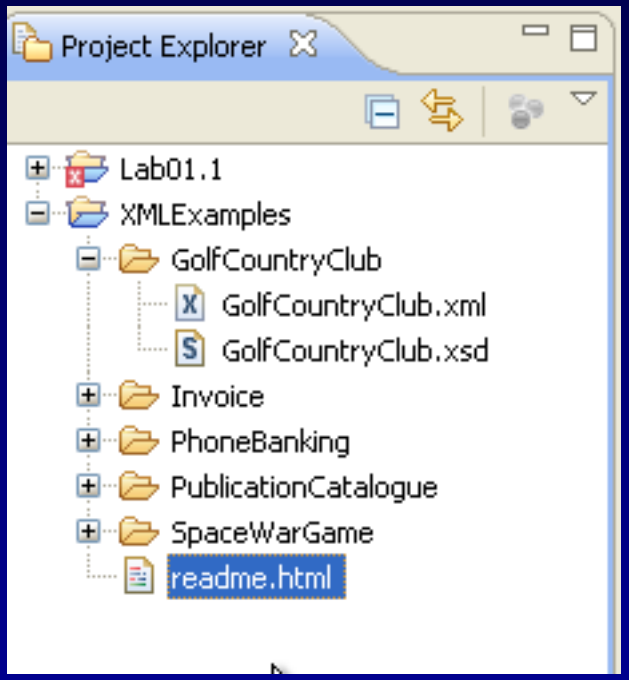
Lab



Project Explorer View

Lab

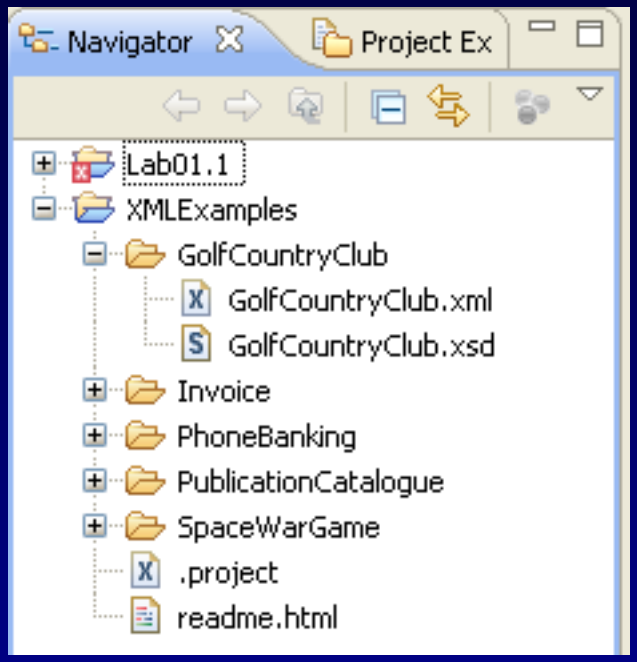
- Shows the resources in a Project
- What is shown may change depending on the type of project
  - In a simple project, there are generally projects, files and folders shown
  - May not show all the files that are in the file system (for example, the .project file which is used by Eclipse to organize the project)



Navigator View

Lab

- Shows how different resources are structured – file system view
  - For a simple project, similar to Project Explorer
- There are three kinds of resources:
  - **Projects**
    - •Used to organize all your resources and for version control.
    - •When you create a new project, you assign a physical location for it on the file system.
    - •A third-party SCM (Source Control Manager) may be used to properly share project files amongst developers.
  - **Folders**
    - •Like directories on the file system
  - **Files**
    - •Correspond to files on the file system



Introduction to XML
Section 2 - Basics

```
<course title='Introduction to XML'>
  <section num='1' title='Introduction'/>
  <section num='2' title='Basics'/>
  <section num='3' title='Namespaces'/>
  <section num='4' title='Schemas'/>
</course>
```

- Section Outline -

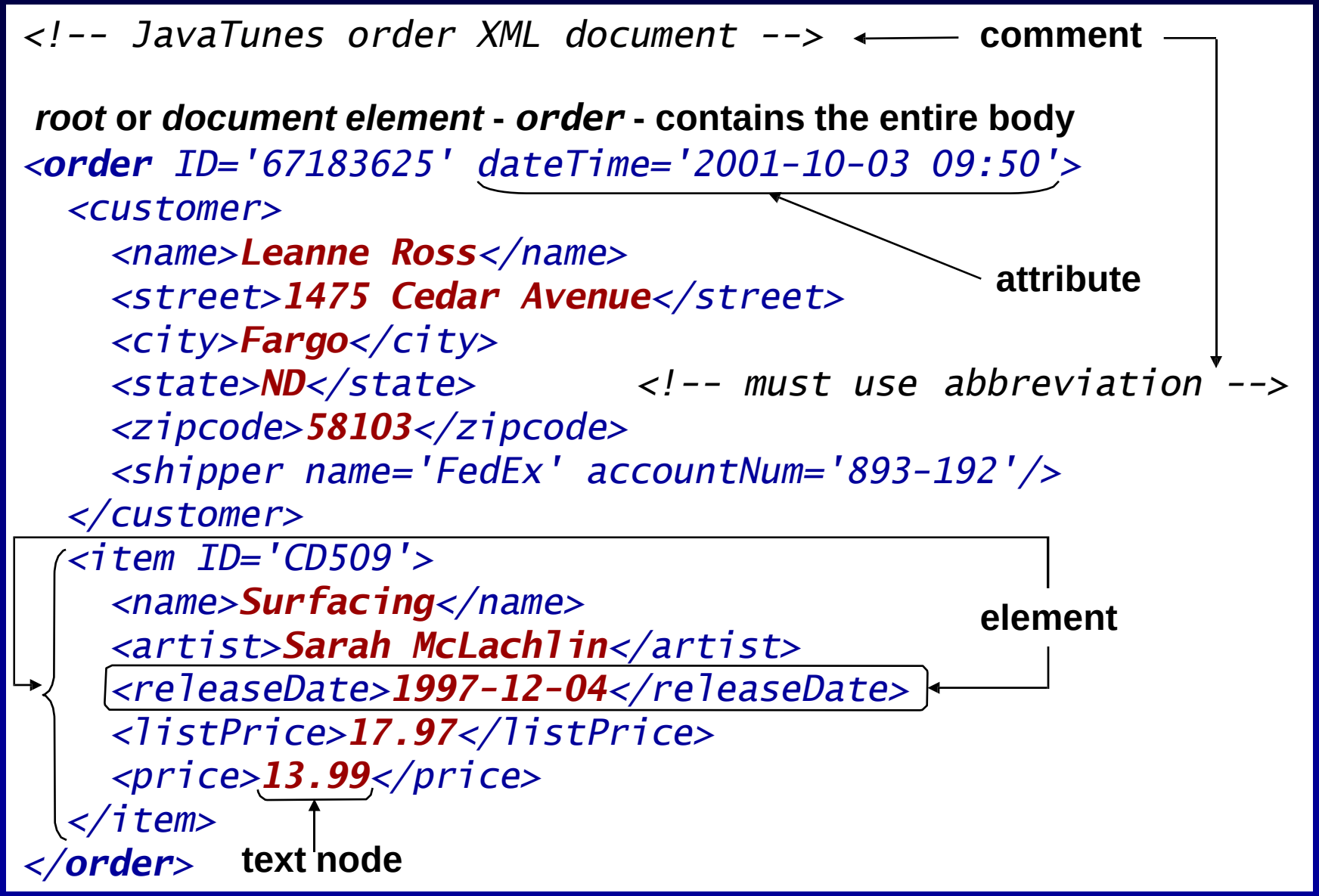
- Building Blocks of XML
- Rules for Well-formed XML Documents

Building Blocks of XML

What's in an XML Document?

- Document Body
- Elements
- Attributes
- Comments

JavaTunes Purchase Order Document - Body



The Document Body

- The “main” part of the document, the body is required
  - The body is contained entirely within the ***document element***, also called the ***root element***
  - In our JavaTunes purchase order, that is the ***`order`*** element
- The fundamental component of the body is the ***element***
- ***Attributes*** can be declared on elements

Elements

```
<!-- the name element contains character data -->
<name>Leanne Ross</name>

<!-- the person element contains child elements -->
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

- Elements are the basic building blocks of an XML document and contain the document's ***content***

- This content is usually ***character data*** or ***child elements***
  - Element content can also be a mixture of character data and child elements, but this is not as common in XML as it is in HTML
  - An element can also be ***empty***, having nothing between its tags

Well-formed Elements

- Every ***start-tag*** must have a matching ***end-tag***
  - An element which has no content is called an ***empty*** element, and can be written with an ***empty-element-tag***

```
<br>                    <!-- ok in HTML - not ok in XML -->
<br/> or <br></br>      <!-- these are equivalent -->
```

- Elements must nest properly

```
<b><i>no!</b></i>       <!-- ok in HTML - not ok in XML -->
<b><i>yes</i></b>
```

- XML documents must contain ***at least one element*** and there must be a ***root element*** which contains all of the other elements
  - Another (better) term for this is the ***document element***

"Element" and "Tag" are Not Synonyms

- An element is delimited by start- and end-tags
  - Elements are not tags and tags are not elements

Attributes

```
<!-- attributes in a start-tag -->
<person ssn='987-65-4321' gender='F'>
  <name>Leanne Ross</name>
  <age>25</age>
</person>

<!-- attributes in an empty-element-tag -->
<shipper name='FedEx' accountNum='893-192'/>
```

- An attribute is additional information **about** or **associated with** an element
  - Attributes can appear in start-tags or empty-element-tags

- **Attributes are considered to be markup**, since they are defined to be information **about** an element, not part of the **content** of an element

Well-formed Attributes

- Attribute values must be quoted
  - You can use single- or double-quotes

```
<img src=logo.gif>      <!-- ok in HTML - not ok in XML -->
<img src="logo.gif"/>   <!-- ok -->
<img src='logo.gif'/>   <!-- ok -->
<img src="logo.gif'/>   <!-- no -->
```

- An element cannot have two attributes with the same name
- Attributes can appear on an element in any order

Elements or Attributes?

- A decision you will often come across is whether to make a piece of data an **element** or an **attribute**

```
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
<!-- or -->
<person name='Leanne Ross' age='25'/>
```

- Elements provide for nested data structures -- attributes are simple name-value pairs

- We will defer this discussion until we know more about elements and attributes

XML Names

- Elements, attributes, etc., must be valid ***XML names***

- XML names can contain:
  - Letters (including non-English letters like µ)
  - Numbers
  - Hyphens -
  - Underscores _
  - Periods .
  - Colons :

- XML names must start with:
  - Letter
  - Underscore

Comments

```
<!-- this is a comment -->
```

- Intended for humans -- ignored by the XML parser
- Can span multiple lines
- Cannot contain `--`
- Can appear anywhere except inside a tag

```
<name> <!-- this is ok --> </name>
<name<!-- this is not -->> </name>
```

XML is Strict

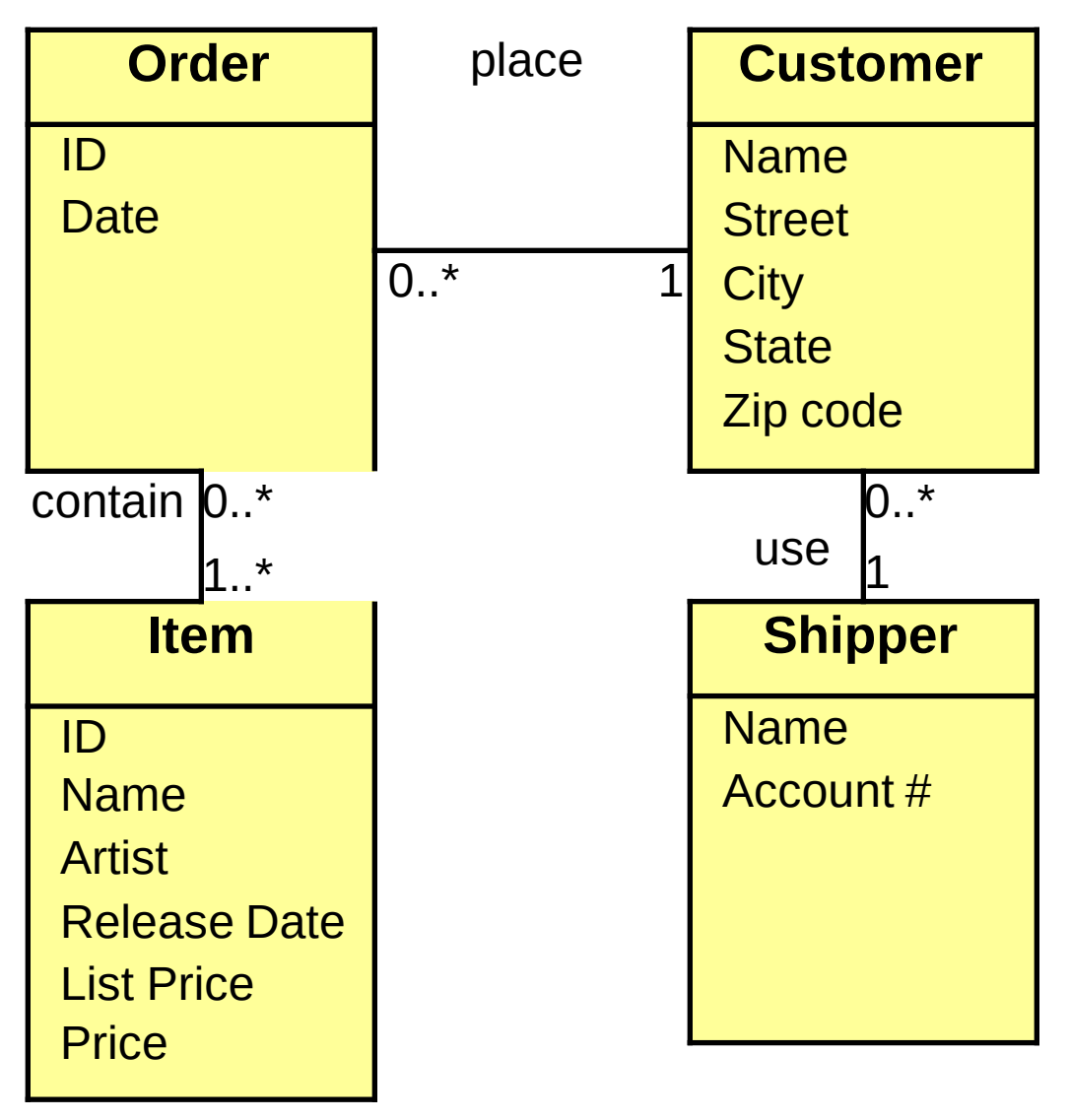
- XML documents **must** be well-formed
  - Well-formedness violations deem a document **unusable** and the XML parser will impolitely abort when it finds one
- This is a good thing
  - The leniency with which Web browsers interpret HTML has led to several compatibility issues and we are **not** about to repeat **that**
- **XML is case sensitive!!!**
  - Reserved words, XML names, **everything**

- Lab 2.1 – Representing Data as XML-

- **Purpose**: In this lab, we will create a JavaTunes purchase order XML document from an order form
  - Continue working in your **Lab01.1** project
- **Objectives**: Learn more about creating XML documents
  - You will create a valid XML document based on the data model of the JavaTunes purchase order seen in the slides
- **Builds on previous labs**: Lab01.1
- **Approximate Time**: 20-30 minutes

Purchase Order - Data Model

Lab



This is a UML class diagram showing the data model for the JavaTunes purchase order

Purchase Order - Sample Document

Lab

```
<!-- JavaTunes order XML document -->

<order ID='67183625' dateTime='2001-10-03  09:50'>
  <customer>
    <name>Leanne Ross</name>
    <street>1475 Cedar Avenue</street>
    <city>Fargo</city>
    <state>ND</state>
    <zipcode>58103</zipcode>
    <shipper name='FedEx' accountNum='893-192'/>
  </customer>
  <item ID='CD509'>
    <name>Surfacing</name>
    <artist>Sarah McLachlin</artist>
    <releaseDate>1997-12-04</releaseDate>
    <listPrice>17.97</listPrice>
    <price>13.99</price>
  </item>
</order>
```

Purchase Order - Data

Lab

JavaTunes Order

Order ID: 12050826

Order Date: February 7, 2002 4:20PM

Customer Info

James Heft
455 Meadow St.
Lodi
CA
95112

Purchase Info

Item ID	Name	Artist	Release Date	List Price	Price
CD513	My, I'm Large	Bobs	1987-02-20	11.97	11.97
CD518	Escape	Journey	1981-02-25	11.97	11.97

Shipping Info

UPS
544-8775-1

Testing for Well-formedness

Tasks to Perform

- Modify the *order.xml* document from the previous lab so it contains purchase order information as well-formed XML
  - Use the structure of the example JavaTunes purchase order XML document earlier in this section (also shown in these lab instructions)
  - You can right click on the ***order.html*** file in the project, and select ***Open With | Web Browser*** to have a nice visual view of the data *
- **Validate** *order.xml* when you're done
  - Right click on *order.xml*, and select **Validate**
  - If you have any errors, they should show up in the Problems view
  - Correct any errors that you find
- Once you have a valid XML document with the data, you're done

More Building Blocks of XML

The Finishing Touches

- Prolog
- XML Declaration
- Document Type Declaration
- Processing Instructions
- Predefined Entities and *CDATA* Sections

JavaTunes Purchase Order Document - Prolog

```
<?xml version='1.0'?>
```
XML declaration

```
<!DOCTYPE order SYSTEM 'order.dtd'>
```
document type declaration

```
<?xml-stylesheet type='text/xsl' href='order.xsl'?>
```
processing instruction

```
<!-- JavaTunes order XML document -->
```
comment

begin document body

```
<order ID='67183625' dateTime='2001-10-03 09:50'>
  <customer>
  <!-- rest of body follows ... -->
```

Prolog

- Consists of everything before the body
  - All prolog components are optional, thus the prolog is optional
- The prolog can contain
  - XML declaration — can only be in prolog
  - Document type declaration — can only be in prolog
  - Processing instructions — can be in prolog or body
  - Comments — can be in prolog or body
- The prolog contains document ***metadata*** -- information about the document

XML Declaration

```
<?xml version='1.0' encoding='UTF-8' standalone='no'?>
```

- Optional, but strongly recommended
  - If present, it must be the **first** thing in the document
- Specifies up to three properties of the document:
  - XML version — **REQUIRED** - ***1.0***
  - Character encoding — OPTIONAL - default is ***UTF-8***
  - External dependencies — OPTIONAL - parser will determine anyway
- These three things **must** appear in the above order
  - You must use quotes around values -- ***''*** and ***""*** are both allowed

Document Type Declaration

```
<!DOCTYPE order SYSTEM 'order.dtd'>
<order>
  <!-- we say this is a "document of type order" -->
</order>
```

- We will cover this in the Schemas section, when we briefly discuss ***document type definitions*** (DTDs) -- for now:
  - It is part of the prolog and is optional (like everything in the prolog)
  - It's purpose is to reference a document type definition (DTD), which specifies the rules for what can be in the document, in what order, etc.
  - It specifies the document's ***type***

- XML documents have a ***type***
  - Denoted by the ***root*** or ***document element***, which is the top-level element, containing everything else

Processing Instructions

```
<?target instruction?>

<!-- some PIs have a well-known format and meaning -->
<?xml-stylesheet type='text/xsl' href='order.xsl'?>
```

target

instruction

```
<!-- others you create with your own format/meaning -->
<?notepad file://venus/documents/resume.txt?>
```

- PIs are directives to the application to "do something"
  - What that is or means is completely up to you -- the XML parser knows nothing about the target nor the instruction, it simply passes these things to the application
  - The target could be the name of a program to be invoked -- the instruction could be an argument(s) to that program
- PIs can appear anywhere in the document except inside a tag

Predefined Entities - Escaping Markup

- What if a document's content contains markup characters?
  - You may need to escape them, so they are not treated as markup

- XML provides five predefined ***entity references***
  - All entity references are delimited by ***&*** and ***;***

<lt;	<
&gt;	>
&quot;	"
&apos;	'
&amp;	&

- **<** and ***&*** **must** be escaped in elements and attributes

- **"** and **'** may need to be escaped in attribute values

Escaping Markup in Content - Examples

```
<condition>a < b</condition>                <!-- no -->
<condition>a &lt; b</condition>             <!-- ok -->

<if condition='a & b'/>                    <!-- no -->
<if condition='a &amp; b'/>                <!-- ok -->


<!-- if attribute value contains ', delimit it with " -->
<person quote="What's up, Doc?"/>          <!-- ok -->

<!-- however, this is not always possible -->
<person quote="What's up, "Doc?""/>              <!-- no -->
<person quote="What's up, &quot;Doc?&quot;"/>    <!-- ok -->
```

CDATA Sections - Escaping LOTS of Markup

```
<![CDATA[none of this will be parsed]]>
```

- A ***CDATA*** section is a block of text that is entirely escaped
  - This can be easier than escaping individual characters
  - Especially useful if your content is XML or HTML code
  - The only thing that cannot appear in a *CDATA* section is ***]]>***

```
<item ID='CD520'>
  <name><![CDATA[<XML-Singalong> &amp; <company>]]></name>
  <artist>The New Tags</artist>
  <releaseDate>2002-02-04</releaseDate>
  <listPrice>9.98</listPrice>
  <price>3.99</price>
</item>
```

Lab

- Lab 2.2 – Adding a PI -

- **Purpose**: In this lab, we will add a processing instruction that transforms our XML into HTML
  - We'll also experiment with some of the other XML syntax
  - Continue working in your **Lab01.1** project
- **Objectives**: Work with Processing Instructions, and with other XML syntax
- **Builds on previous labs**: Lab02.1
- **Approximate Time**: 20-30 minutes

Adding a PI to Our Document

Lab

Tasks to Perform

- Add an *xml-stylesheet* PI to *order.xml*
  - Have it specify ***customer.xsl*** as the stylesheet (this is an XSLT stylesheet already supplied in your lab directory)
  - The *xml-stylesheet* PI must be in the prolog - See the example in this section
  - The PI directs an XML-enabled browser to display the document according to what's in *customer.xsl*
- **Load** *order.xml* into an XML-enabled browser -- what do you see?
  - Open the supplied *customer.html* file in a browser – they should be similar (This file is in the lab directory)
  - After you see what it does, comment out the PI
- Internet Explorer 6.0+ and Firefox both support XSLT 1.0

Adding PIs and Comments

Tasks to Perform

- Create a PI that directs the application to page the shipper
  - Put it somewhere in the document body, e.g., right after the *shipper* element -- we're not going to use it, so you can make it up -- example:

    ```
    <?pager UPS 544-8775-1?>
    ```

- If you wish, add some comments to the document
  - Comments can go in the prolog and the body

- OPTIONAL - change some of the purchase order data to use markup characters and then escape those characters
  - For example, `<artist>Seals & Crofts</artist>`

Introduction to XML
Section 3 - Namespaces

```
<course title='Introduction to XML'>
  <section num='1' title='Introduction'/>
  <section num='2' title='Basics'/>
  <section num='3' title='Namespaces'/>
  <section num='4' title='Schemas'/>
</course>
```

- Section Outline -

- The Motivating Problem
- The Namespace Solution
- Namespace Scope and Overriding
- Default Namespaces
- Namespaces and Attributes

The Motivating Problem

Why Do We Need Namespaces?

- Name Collisions
- Possible Solutions

Name Collision - Example

```
<!-- JavaTunes order document - fragment -->

<!-- from the JavaTunes customer vocabulary -->
<customer>
  <name title='Ms.'>
    <firstName>Leanne</firstName>
    <lastName>Ross</lastName>
  </name>
  ...

<!-- from the JavaTunes item vocabulary -->
<item ID='CD509'>
  <name>Surfacing</name>
  ...
```

- We have a potential problem here -- we need to distinguish between the two kinds of ***name*** elements
  - Because they have different ***content models***

JavaTunes Name Collision - Possible Solutions

```
<customer>
  <customer-name title='Ms.'>
    <firstName>Leanne</firstName>
    <lastName>Ross</lastName>
  </customer-name>
  ...
<item ID='CD509'>
  <item-name>Surfacing</item-name>
  ...
```

- Problem - we have two types of content for names
- This can be handled by XML Schema (but not by DTDs)
- Or, we could use unique names:
  customer-name and ***item-name***

Inter-Organization Name Collisions - Example

```
<product>                                  <!-- MathML product -->
  <lowlimit>
  <uplimit>

<product partNumber='A678'>                <!-- your product   -->
  <stock-level>
```

```
<formula concise='H 3 N 1'/>               <!-- CML formula  -->

<formula>                                  <!-- your formula -->
  <price>
  <quantity>
```

```
<MessageHeader>
  <From>                                   <!-- ebXML From -->
```

- You **could** rename your elements to avoid collisions, but ...

The Namespace Solution

"Area Codes" for XML Names

- Definition
- Terminology
- URIs and Prefixes

The Namespace Solution

- To disambiguate names, we define a ***namespace***

- Suppose we define two namespaces -- ***`customer`*** and ***`item`***
  - We can now have a ***`name`*** element in both namespaces
  - We distinguish between them by using the labels of the namespaces as ***prefixes*** to the *`name`* elements, i.e.,

    `customer:name` and ***`item:name`***

- The ***fully-qualified name*** consists of two parts -- a ***namespace prefix*** and a ***local base name***, with a colon (***`:`***) delimiting the two parts

Namespace Terminology

- An XML name is said to be a ***qualified name*** or ***QName*** when it consists of a ***prefix***, followed by a colon (***:***), followed by a ***local part*** -- and the prefix is bound to a namespace

cust:name	=	qualified name
cust	=	prefix
name	=	local part

- **NOTE** - colons should **never** be used in XML names except when used as a delimiter between the prefix and the local part of a qualified name

Namespace Overlap

- If we allow arbitrary strings as namespace names, we could easily get collisions -- which is the problem we are trying to solve in the first place(!)
  - Two organizations could both use ***`customer`*** for a namespace
  - If you are using vocabularies from both of these organizations, you could end up with ***`customer:name`*** and ***`customer:name`***
- Namespaces should satisfy several criteria:
  - **Universally unique**
  - **Conform to XML naming rules**
  - Simple in structure, so that documents remain readable even when the namespace prefixes are used

The URI + Prefix Solution

- To meet these criteria, we define a namespace in two steps

1. The namespace name is defined as a URI (often a URL)
   - This is designed to guarantee uniqueness
   - But URIs are long and generally cannot be used as XML names, which cannot contain a slash (/) character

2. We then bind a ***local prefix***, or abbreviation, to the URI
   - This prefix can follow the XML naming rules
   - It only has to be unique within the immediate document
   - Think of the prefix as a local "nickname" for the namespace

- **The URI is just a name and does not point to anything**

Defining a Namespace - Binding Prefixes to URIs

```
<element-name xmlns:prefix='namespaceURI'>
```

```
<cust:customer
 xmlns:cust='http://www.javatunes.com/customer'>
  <cust:name title='Ms.'>
    <cust:firstName>Leanne</cust:firstName>
    <cust:lastName>Ross</cust:lastName>
  </cust:name>
  ...
<item ID='CD509'>
  <name>Surfacing</name>
  ...
```

- The **`xmlns:cust`** attribute binds the prefix **`cust`** to the namespace **http://www.javatunes.com/customer**

Namespace Scope and Overriding

"Inheritance" for Namespaces

- Namespace Scope
- Rules for Namespace Definitions
- Overriding Namespace Prefixes

Namespace Scope

- Namespace definitions are ***scoped*** by their declaring elements
  - The namespace definition is available to the declaring element, and to all of its descendant elements
- Namespaces which are used throughout the whole document should be defined in the document element
  - **The scope of a namespace defined in the document element is the entire document**, which makes the namespace available to all the elements and attributes in that document

Namespace Definitions - Rules

- A namespace definition **can** be applied to:
  - The element in which the namespace is defined
  - In the previous example, ***cust:customer*** is in the namespace
  - The elements contained in that element (its descendants)
  - In the previous example, ***cust:name***, ***cust:firstName***, and ***cust:lastName*** are also in the namespace
- **NOTE** that a namespace definition **allows** the use of the namespace but does **not require** that an element or attribute be in that namespace

Namespace Definitions - Rules

- An element's attributes are **not** automatically in the same namespace as the element itself
  - The prefix **must** appear on an attribute to indicate that it belongs to a namespace
- More than one namespace can be declared in the same element
  - This commonly occurs in **document elements**
- A prefix may not be bound to two different namespaces simultaneously
  - Doing so would make the prefix ambiguous

Namespace Scope - Example

```
<!-- two namespaces defined for the whole document -->
<order ID='67183625' dateTime='2001-10-03 09:50'
       xmlns:cust='http://www.javatunes.com/customer'
       xmlns:item='http://www.javatunes.com/item'>
  <customer>
    <cust:name title='Ms.'>
      <firstName>Leanne</firstName>
      <lastName>Ross</lastName>
    </cust:name>
    ...
  </customer>
  <item ID='CD509'>
    <item:name>Surfacing</item:name>
    ...
  </item>
</order>
```

- Only the *cust:name* and *item:name* elements belong to a namespace

Overriding Namespace Prefixes

- A prefix binding acts much like a variable definition
  - A prefix is bound to a namespace URI throughout the scope of the namespace definition, as we've discussed
  - **Except for** descendant elements in which the **same prefix** is bound to a **different namespace**
  - The second prefix binding ***overrides*** (or shadows) the first binding, but **only** within the scope of the second namespace definition

Overriding Namespace Prefixes - Example

```
<order ID='67183625' dateTime='2001-10-03 09:50'>

  <cust:customer
   xmlns:cust='http://www.javatunes.com/customer'>

    <cust:name
     xmlns:cust='http://www.javatunes.com/names'
     title='Ms.'>
      <cust:firstName>Leanne</cust:firstName>
      <cust:lastName>Ross</cust:lastName>
    </cust:name>                    Scope of second cust

    <cust:street>1475 Cedar Avenue</cust:street>

  </cust:customer>                  Scope of first cust
  ...
</order>
```

Lab

- Lab 3.1 – Namespaces -

- **Purpose**: In this lab, we will add namespace definitions to an XML document
  - We'll declare the namespaces in two different ways, and compare them
- **Objectives**: Work and become more familiar with namespaces
- **Builds on previous labs**: None
  - The root lab directory where you will do your work for this lab is:

  C:\StudentWork\XMLIntro\workspace\Lab03.1
- **Approximate Time**: 30-40 minutes

Namespaces

Lab

Tasks to Perform

- Create a **Project** (**File | New | Project | General| Project**)
  - Call the project **Lab03.1**
- You will work on the file ***orderns.xml***, which is already in the lab directory, and is shown on the next two slides
- Verify your solution **after each step** by validating it and making sure that there are no errors
  - You can also load your *orderns.xml* file into Internet Explorer, to see how it formats namespace definitions

ordersns.xml

Lab

```
<?xml version='1.0'?>

<!-- JavaTunes order XML document -->

<order ID='03230413' dateTime='2002-03-24 01:20'>
  <customer>
    <name title='Ms.'>
      <firstName>Rachel</firstName>
      <lastName>Jacobs</lastName>
    </name>
    <street>1408 Fell St.</street>
    <city>Oneida</city>
    <state>NY</state>
    <zipcode>14180</zipcode>
    <shipper name='FedEx' accountNum='77-63-2478'/>
  </customer>
  ...
```

ordersns.xml

Lab

```
  ...
  <item ID='CD514'>
    <name>So</name>
    <artist>Peter Gabriel</artist>
    <releaseDate>1986-10-03</releaseDate>
    <listPrice>17.97</listPrice>
    <price>13.99</price>
  </item>
  <item ID='CD506'>
    <name>Seal</name>
    <artist>Seal</artist>
    <releaseDate>1991-08-18</releaseDate>
    <listPrice>17.97</listPrice>
    <price>14.99</price>
  </item>
</order>
```

Part A - Namespaces and Prefix Bindings

Lab

Tasks to Perform

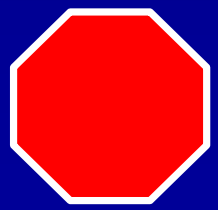
- Define a namespace prefix for **items** so that its scope is confined to **only** each *item* element (see notes)
  - Place each *item* element and each one's child elements in the namespace -- do not put *item*'s *ID* attribute in the namespace
- Define a namespace prefix for **customers** so that its scope is **only** the *customer* element (see notes)
  - Place the *customer* element and all of its descendant elements in the namespace -- do not put *name*'s *title* attribute in the namespace
  - Do not put the *shipper* element's attributes in the namespace
- After validating *orderns.xml*, look at it in design view -
  - Open the nodes for the order, customer, and item elements, and notice where the namespace declarations lie
- Copy *orderns.xml* to ***ordernsA.xml*** (within Eclipse *) for use later

Part B - Namespace Scope

Lab

Tasks to Perform

- Define the namespace prefix for items so that it includes both *item* elements, but only define the namespace in one place
  - What are your choices for where this namespace definition can go?
- Where else can you define the namespace prefix for customers?
  - Define it there
- After making these changes, validate your document again
  - Open *ordernsA.xml* also, and compare it to your current *orderns.xml* looking at both of them in design view
  - It's easy to see in this view where the namespaces are declared, and how the contained elements are effected by the declarations



Default Namespaces

Those Prefixes are a Pain

- Defining and Using the Default Namespace
- Using Both Default and Explicit Namespaces
- Overriding the Default Namespace

Default Namespaces

- Since using a namespace-prefixed name can be difficult to maintain, and make a document more difficult to read, we have another option -- the ***default namespace***
  - A default namespace is bound to a **zero-length string** prefix
  - Within the scope of the default namespace, **all elements without a prefix are in that namespace**
- **This only applies to elements, not attributes!**
  - Attributes **must** have a namespace prefix if they are to be in a namespace, even if they are in the scope of a default namespace

Default Namespaces - Example

```
<order ID='10161984' dateTime='2002-03-28 06:30'>
  ...
  <!-- this element is explicitly in a namespace -->
  <i:item ID='CD508' xmlns:i='http://javatunes.com/item'>
    <i:name>So Much for the Afterglow</i:name>
    <i:artist>Everclear</i:artist>
    <i:releaseDate>1997-01-19</i:releaseDate>
    <i:listPrice>16.97</i:listPrice>
    <i:price>13.99</i:price>
  </i:item>

  <!-- this element is in a namespace by default  -->
  <item ID='CD510' xmlns='http://javatunes.com/item'>
    <name>Hysteria</name>
    <artist>Def Leppard</artist>
    <releaseDate>1987-06-20</releaseDate>
    <listPrice>17.97</listPrice>
    <price>14.99</price>
  </item>
</order>
```

Using Both the Default and Explicit Namespaces

```
<!-- default and explicit namespaces for document -->
<order ID='32450227' dateTime='2002-01-15 09:35'
       xmlns='http://www.javatunes.com/order'
       xmlns:item='http://www.javatunes.com/item'>

  <!-- this element is in the default namespace -->
  <customer>...</customer>

  <!-- this element is in the explicit namespace -->
  <item:item ID='CD512'>
    <item:name>Human Clay</item:name>
    <item:artist>Creed</item:artist>
    <item:releaseDate>1999-10-21</item:releaseDate>
    <item:listPrice>18.97</item:listPrice>
    <item:price>13.28</item:price>
  </item:item>
</order>
```

Overriding Default Namespaces

- Earlier, we saw how a namespace prefix can be overridden
  - The same process can be used to override the default namespace
- We can also "remove" the default namespace by setting the *xmlns* attribute to the empty string (`''`)
  - This is the only time we would ever want to use the empty string as a namespace name
- Attempting to bind a **prefix** to the empty string as a namespace name will produce unpredictable results, e.g., *xmlns:item=''*

Overriding Default Namespaces - Example

```
<!-- default namespace for document -->
<order ID='32450227' dateTime='2002-01-15 09:35'
       xmlns='http://www.javatunes.com/order'>

  <!-- this element overrides the default namespace -->
  <customer xmlns='http://www.javatunes.com/customer'>
    <name title='Mr.'>
      ...
  </customer>

  <!-- this element is not in any namespace -->
  <item ID='CD512' xmlns=''>
    <name>Human Clay</name>
    <artist>Creed</artist>
    <releaseDate>1999-10-21</releaseDate>
    <listPrice>18.97</listPrice>
    <price>13.28</price>
  </item>
</order>
```

Namespaces and Attributes

No “Free Ride” for Attributes

- Recap of Namespaces and Attributes
- Default Namespaces and Attributes

Namespaces and Attributes - Recap

- Attributes can also be in a namespace
  - This is indicated in the usual way, with a prefix
- An element and its attribute(s) may belong to different namespaces
  - An attribute is **not** automatically in the same namespace as its element
  - A example is XLink, which uses a namespace to allow XLink attributes to appear on elements which themselves are not in the XLink namespace

```
<c:customer
 xmlns:xlink='http://www.w3.org/1999/xlink'
 xmlns:c='http://www.javatunes.com/customer'>
  <c:name xlink:href='http://www.verisign.com/verify'
          xlink:type='simple'
          title='Ms.'>
    ...
```

Default Namespaces and Attributes

- The default namespace **never** applies to attributes
  - Attributes **must always have a prefix** to be in a namespace
  - Therefore, if a default namespace is active and you want an attribute to be in that namespace, you have to have a prefix binding also

```
<!-- default namespace for document -->
<order ID='67183625' dateTime='2001-10-03 09:50'
       xmlns='http://www.javatunes.com/order'>

  <!-- a prefix is also bound to this namespace -->
  <customer xmlns:order='http://www.javatunes.com/order'>
    <name order:title='Ms.'>
      <firstName>Leanne</firstName>
      <lastName>Ross</lastName>
    </name>
    <street>1475 Cedar Avenue</street>
    ...
</order>
```

- Lab 3.2 – Default Namespaces -

- **Purpose**: In this lab, we will use default namespaces in an XML document
  - We'll work with our order document, as well as with a new document, ***purchase-requestns.xml***, which is also in the lab dir
- **Objectives**: Work with default namespaces
- **Builds on previous labs**: Lab 3.1
  - Continue working in your Lab03.1 project
- **Approximate Time**: 30-40 minutes

purchase-requestns.xml

Lab

```
<?xml version='1.0'?>

<!-- JavaTunes purchase request XML document -->

<purchase-request>
  <purchase>
    <amount currency='USD'>1016.84</amount>
    <dateTime>2002-01-04 14:21</dateTime>
  </purchase>
  <merchant>
    <merchant-name>JavaTunes</merchant-name>
    <business-number>190973</business-number>
  </merchant>
  <credit-card type='Visa'>
    <name-on-card>Bob Smith</name-on-card>
    <card-number>198798739991827 7</card-number>
    <exp-date>01/04</exp-date>
  </credit-card>
</purchase-request>
```

Default Namespaces

Lab

Tasks to Perform

- Use a namespace for purchase requests (see notes)
  - Make it the default namespace for the entire document
  - Where do you make this namespace definition?
- In the scope of the `credit-card` element, reset the default namespace to be the one for credit cards (see notes)
  - This element and all of its child elements should belong to this "JavaTunes credit card" namespace
  - The `type` attribute should not be in any namespace

Default Namespaces

Tasks to Perform

- Use a namespace for currencies (see notes)
  - Use the URI *http://www.monetary.org*
  - Put the `amount` element's **`currency`** attribute in this namespace
- Validate your document, and view it in Design View
  - Notice the namespaces
  - You can also load the document into Internet Explorer to view it
- Use a default namespace for orders in the *orderns.xml* document from the last lab (see notes)
  - All the elements in the order should be in this one namespace (see the notes for an explanation as to why this is okay)
  - As before, none of the attributes should be in the namespace

Introduction to XML
Section 4 - Schemas

```
<course title='Introduction to XML'>
  <section num='1' title='Introduction'/>
  <section num='2' title='Basics'/>
  <section num='3' title='Namespaces'/>
  <section num='4' title='Schemas'/>
</course>
```

- Section Outline -

- Valid XML Documents
- XML Schema Basics
- Data Modeling with XML Schema
- Overview of Document Type Definitions (DTDs)
- Advanced Topics - OPTIONAL

Valid XML Documents

eXtensible Doesn’t Mean Anarchy

- Definition of Validity
- Definition of Schema
- Schema Languages

Definition of Validity

- A XML document is ***valid*** if its **structure and content** are in compliance with the rules set forth in its ***schema***
  - Schemas allow us to validate XML documents
  - Valid documents must also be well-formed -- **all** XML documents must be well-formed

- XML documents without schemas:
  - If we have an XML document that we say is of type ***order***
  - But we have no schema that defines what an order really is
  - How many items can an order have? Must it have items at all? What comprises an item? What comprises a customer?

- We need answers…we need a schema!

Definition of Schema

- A ***schema*** is a document that **describes the structure and content** of an XML document
  - It is a blueprint or definition for an XML document
- It contains a set of rules for a **type** of document -- rules that dictate things such as:
  - Which elements are permitted, required, in what order, etc.
  - What each element's content must be
  - The attributes that are permitted/required on elements, default values for attributes

Analogy - Schema::Document as Class::Object

- A **schema** defines a **type** of XML document
  - XML documents of that type are said to be ***instances*** of the schema
  - A schema is like a contract -- valid documents of this type must adhere to the rules in the schema
- In OOP, a **class** defines a **type** of object
  - Objects of that type are said to be instances of the class

Schema Languages

- XML inherited the ***Document Type Definition*** (DTD) syntax from SGML
  - Still commonly used, DTDs will be with us for quite awhile
  - Does not use XML syntax and has little support for datatypes
  - But all XML software supports it -- and we will study it briefly

- The future (and present) is ***W3C XML Schema***
  - Uses XML syntax -- a W3C XML Schema is an XML document
  - Strong support for datatypes
  - W3C Recommendation in May, 2001 -- becoming readily adopted

- There are other schema languages
  - **RELAX/NG** has been developed under the endorsement of OASIS

DTD Weaknesses

- DTDs specify document ***structure***, but they don't help much in the way of ***content***
  - Example: `<age>green</age>`
- DTDs have very limited support for datatypes
  - Attributes have some notion of type
  - But elements ultimately decompose into untyped character data
- DTDs do not use XML syntax
- DTDs do not support namespaces
- Content models cannot be defined flexibly
  - And no default values for elements

W3C XML Schemas

- Can define both ***content* and *structure***
  - Focus is on (reusing) datatypes
  - Many predefined datatypes for integers, floats, dates, strings, etc.
  - You can create (and reuse) your own datatypes
  - You can extend/refine the existing datatypes
  - You can specify rich, flexible content models
- Incredibly powerful
  - You can express anything you want with this schema language
- Uses XML syntax and supports namespaces
  - An XML Schema is an XML document
  - XML Schema is a replacement for DTDs

What's the Catch?

- XML Schema support is not yet widespread (but will be)
  - Specification finalized in May, 2001
  - Apache's **Xerces 2** parser supports validation against XML Schemas
  - Sun's **JAXP 1.2** (Java API for XML Processing) uses Xerces 2, and so supports XML Schema
- Much more verbose than DTDs
  - But verbosity was not a design goal of XML
  - Verbosity is sacrificed in favor of precision
- More difficult to learn and use
  - Specification is in three parts, totaling several hundred pages
  - ***http://www.w3.org/TR/xmlschema-0***, *.../xmlschema-1*, and *.../xmlschema-2*

XML Schema Basics

Getting Started with XML Schema

- General Form of an XML Schema
- Element Definitions
- Simple Types
- Complex Types
- Attribute Definitions
- Using XML Schema with Namespaces

General Form of an XML Schema

```
<?xml version='1.0'?>

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <!-- elements and their attributes are defined here -->

</xsd:schema>
```

- An XML Schema is an XML document
  - The document element is the ***xsd:schema*** element
- XML Schema elements belong to the namespace ***http://www.w3.org/2001/XMLSchema***

Elements, Attributes, and Types

- The four primary components of schema definitions are:
- **Element definitions** `xsd:element`
- **Attribute definitions** `xsd:attribute`
- **Simple type definitions** -- specify restrictions on character data content, e.g., `xsd:string` and `xsd:integer`
- **Complex type definitions** -- specify an element's child elements and/or its attributes

Simple Schema and XML Document - Example

```
<?xml version='1.0'?>

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='simple' type='xsd:string'/>

</xsd:schema>
```

```
<?xml version='1.0'?>

<!-- can't get much simpler than this -->

<simple>Hey, I'm an XML Document</simple>
```

Another Simple Schema and XML Document

```
<?xml version='1.0'?>

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='age' type='xsd:positiveInteger'/>

</xsd:schema>
```

```
<?xml version='1.0'?>

<!-- age must contain an integer greater than 0 -->

<age>28</age>
```

- If *age* contained a value of *green*, this XML document would be invalid

Referencing a Schema in an XML Document

```
<?xml version='1.0'?>

<!-- schema stored in file age.xsd -->
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='age' type='xsd:positiveInteger'/>

</xsd:schema>
```

```
<?xml version='1.0'?>

<!-- point to schema with a schema location attribute -->
<age
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xsi:noNamespaceSchemaLocation='age.xsd'>28</age>
```

XML Schema namespace for instance documents

- This is a **hint** to the parser to tell it where to find the schema
  - **NOTE** the use of a different namespace for this attribute -- the **XML Schema namespace for *instance documents*** (XML documents)

Element Definitions

- **`xsd:element`** defines an XML element within a schema
  - At a minimum, it defines the **name** and **type** of the element

- Elements must have a **`name`** attribute
  - Its value is what appears in the start/end tags in the XML document
  - It is an unqualified local name, i.e., it cannot specify a namespace

- Elements must have a **`type`** attribute
  - The type describes the element content and allows for its validation
  - The type of an element is described by **either**:
    - A **type attribute** which is a reference to a defined type
    - A **type definition** within the element definition
  - If no type is supplied, then the element's type is **`xsd:anyType`**

Simple Types

- **Simple types** describe **values** or document content
  - Each type is some kind of restriction on character content
  - Simple types provide for validation of values within the document
- The fundamental type is ***xsd:anyType***, known as the ***ur-type***
  - Basic types are restrictions on the ur-type
- Simple types do not describe document structure

Simple Types

- XML Schema provides a wide selection of built-in simple types
  - **String** types - character strings
  - **Numeric** types - numeric values
  - **Date** and **time** types - ISO 8601 date and time values
  - **Legacy** types - attribute types described in the XML1.0 spec
  - **Other** types - miscellaneous types such as *`xsd:anyURI`* and *`xsd:boolean`*

- These built-in datatypes fall into two categories:
  - **Primitive** datatypes
  - **Derived** datatypes

Simple Types - Primitive Datatypes

- **Primitive datatypes** exist on their own
  - Not derived from another datatype

- XML Schema defines several primitive datatypes
  - Representations for strings, numbers, date/time values, binary data

- String datatypes
  - ***xsd:string***
  - ***xsd:anyURI***
  - ***xsd:QName***

Simple Types - Primitive Datatypes

- Numeric datatypes
  - ***xsd:boolean (true | false)***
  - ***xsd:decimal***
  - ***xsd:float***
  - ***xsd:double***

- Date/time datatypes
  - ***xsd:dateTime***
  - ***xsd:date***
  - ***xsd:time***
  - ***xsd:gYear***
  - ***xsd:gMonth***
  - ***xsd:gDay***
  - ***xsd:gYearMonth***
  - ***xsd:gMonthDay***
  - ***xsd:duration***

- Binary datatypes
  - ***xsd:hexBinary***
  - ***xsd:base64Binary***

Simple Types - Derived Datatypes

- **Derived datatypes** are defined in terms of restrictions or compositions of other datatypes

- XML Schema defines 13 derived numeric datatypes and 12 derived string datatypes

xsd:integer	***xsd:positiveInteger***
xsd:long	***xsd:negativeInteger***
xsd:int	***xsd:nonPositiveInteger***
xsd:short	***xsd:nonNegativeInteger***
xsd:byte	
xsd:unsignedLong	***xsd:unsignedShort***
xsd:unsignedInt	***xsd:unsignedByte***

Lab

- Lab 4.1 – Simple Schema -

- **Purpose**: In this lab, we will create a very simple XML Schema and validate an instance document against that schema
- **Objectives**: Become familiar with XML Schema basics
- **Builds on previous labs**: None
  - The root lab directory where you will do your work for this lab is:

  C:\StudentWork\XMLIntro\workspace\Lab04.1
- **Approximate Time**: 20-30 minutes

Create XML and Schema Files

Lab

Tasks to Perform

- Create a **Project** (**File | New | Project | General| Project**)
  - Call the project **Lab04.1**
- In *your project*, create a file named ***simple.xsd*** which has the schema for documents of type `simple`
  - Create a new Schema doc via **File | New | XML Schema**
  - Next, **modify the namespace declaration** for the XML schema namespace to use the **xsd prefix -** Instead of the default prefix
  - Also **remove the namespace declarations** for the target and "simple" namespaces **-** We'll add in namespaces later

*simple.xsd

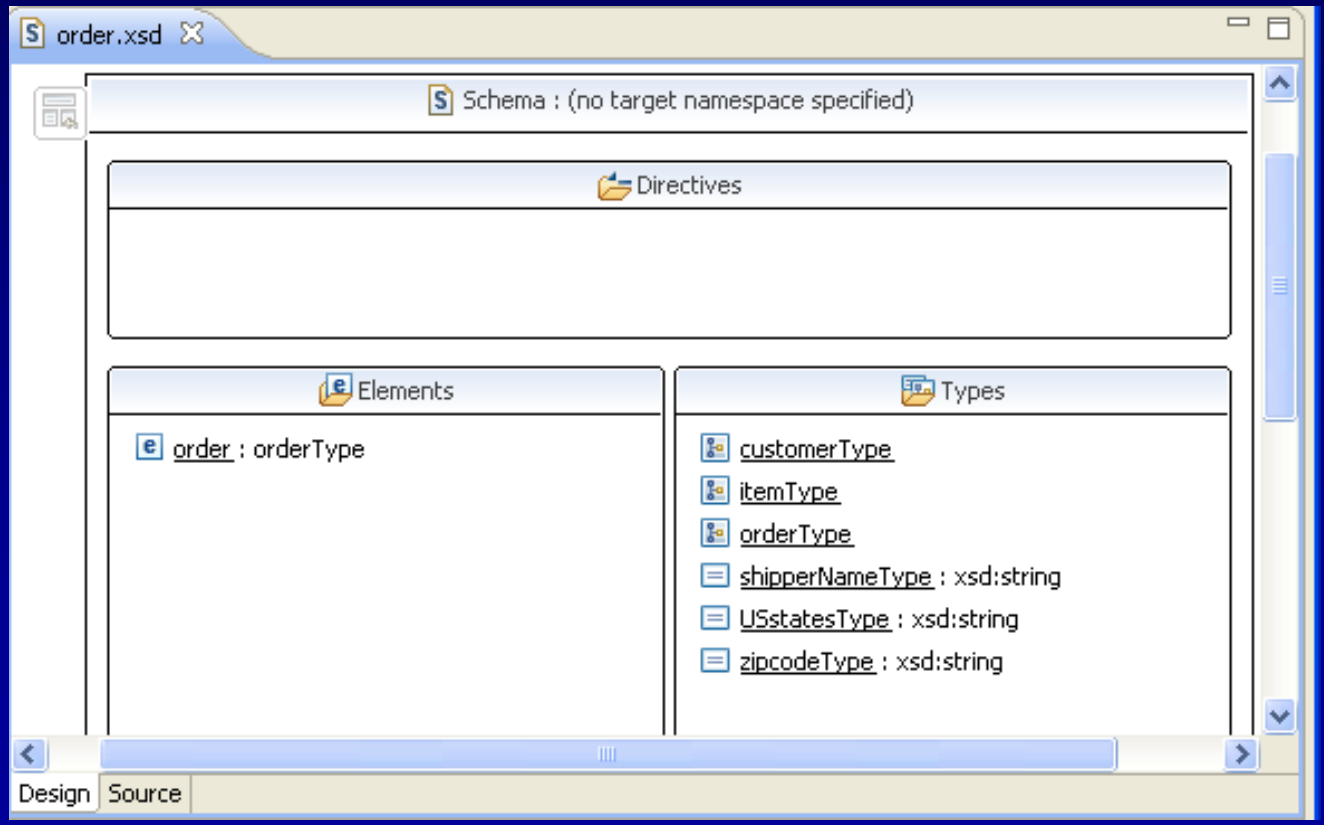
```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

</xsd:schema>
```

Design Schema View

Lab

- Shows the structure of the schema
  - You can edit the schema here, or in the source (they'll stay in sync)
  - Double clicking zooms the editor in on the clicked element, right clicking allows you to modify elements



Create XML and Schema Files

Lab

Tasks to Perform

- Add a single element to the schema

```
<xsd:element name='simple' type='xsd:string'/>
```

  - You can use design or source view – as you prefer
  - Changes in one are reflected in changes in the other

- **Create an XML document** based on *simple.xsd* - it's easy to use Eclipse to create an XML file based on a schema
  - Right click on the project and select: **New | XML**
  - Name the file ***simple.xml*** in the first dialog, and in the second dialog, select "**Create XML file from an XML schema file**" and click **Next**
  - In the next dialog, select simple.xsd from the Lab04.1 project and click **Finish**
  - Eclipse will create the file with most of the content you need

Validate the XML Document

Lab

Tasks to Perform

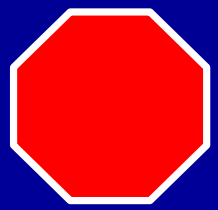
- Review ***simple.xml*** which has our very simple XML document in it (example shown at bottom)
  - Note the schema location element *
- Validate your XML file
  - Since you declare a schema file, it will validate against the schema
- If the document is valid, you will get no error messages
- If the document is invalid, you will see an error message(s)

Use Different Types

Lab

Tasks to Perform

- Experiment with some of the other simple datatypes
  - *xsd:integer*, *xsd:float* , *xsd:date* are common ones to use
  - The easiest way is to use them one at a time, changing the XML and schema documents each time
- Make sure you try some content that should be invalid
  - For example, use a schema type of xsd:integer, and leave the content as the original text string
  - Or use an element with a name different from <simple>
  - You should get an error(s)



Complex Types

- **Complex types** describe types (content models) containing **child elements** and/or **attributes**
  - Simple content allows character data only, with no child elements and no attributes
  - Complex content allows child elements and/or attributes

- Examples of elements that would need complex types:

```
<!-- this element has child element content -->
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

```
<!-- this element has attributes -->
<shipper name='FedEx' accountNum='893-192'/>
```

Defining Complex Types

- ***xsd:complexType*** is used to define a complex type
  - A named *xsd:complexType* appears at the **top level** of the schema
- Complex types are defined in terms of ***model groups***, also called ***compositors***
  - Model groups allow you to group child elements together to construct higher level content models
  - Every complex type has exactly one model group
- Model groups include:
  - ***xsd:sequence***
  - ***xsd:choice***
  - ***xsd:all***

xsd:sequence

- Specifies a group of child elements **in a specific order**

```
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

```
<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'/>
  </xsd:sequence>
</xsd:complexType>
```

- The following instance of *personType* is not valid

```
<person>
  <age>25</age>
  <name>Leanne Ross</name>
</person>
```

Using Complex Types

- You can define a complex type as a direct child of *xsd:schema* and then use it like any other type
  - In defining an element, you reference this complex type by name

```
<?xml version='1.0'?>

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='person' type='personType'/>

  <xsd:complexType name='personType'>
    <xsd:sequence>
      <xsd:element name='name' type='xsd:string'/>
      <xsd:element name='age' type='xsd:positiveInteger'/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Referencing Top-Level Elements

- You can also define elements at the top level and reference them from within a complex type

```
<?xml version='1.0'?>

<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>

  <xsd:element name='age' type='xsd:positiveInteger'/>

  <xsd:element name='person' type='personType'/>

  <xsd:complexType name='personType'>
    <xsd:sequence>
      <xsd:element name='name' type='xsd:string'/>
      <xsd:element ref='age'/>
    </xsd:sequence>
  </xsd:complexType>

</xsd:schema>
```

Anonymous Types

- There are often times when you only want to use a complex type definition once
  - As opposed to global definitions at the top level of the schema

- Define the type as part of an element definition
  - It is only used once, by that definition
  - We say that the element's type is defined ***locally***

```
<xsd:element name='part'>      <!-- no type attribute -->
  <xsd:complexType>            <!-- no name attribute -->
    <xsd:choice>
      <xsd:element name='name'   type='xsd:string'/>
      <xsd:element name='number' type='xsd:string'/>
    </xsd:choice>
  </xsd:complexType>
</xsd:element>
```

Adding Flexibility to Content Models

- What if we had an element with different possible content scenarios?
  - For example, a part that has either a name or a part number, but not both

```
<part>
  <name>Left Threaded 1/4" Widget</name>
</part>
```

```
<part>
  <number>LT-1/4-W</number>
</part>
```

- What if the order of child elements does not matter?
  - For example, a *person* element has *name* and *age* child elements, but we don't care in which order they occur

xsd:choice

- Specifies a **choice** of one of a group of elements
  - The type below defines an element that will have one child element
  - That child can be **either** a *name* or a *number*
  - Which is exactly what we want

```
<xsd:element name='part' type='partType'/>

<xsd:complexType name='partType'>
  <xsd:choice>
    <xsd:element name='name'   type='xsd:string'/>
    <xsd:element name='number' type='xsd:string'/>
  </xsd:choice>
</xsd:complexType>
```

More Complex Choices

- What if we had different kinds of addresses?
  - For example, US and Canadian

```
<address>
  <street>1475 Cedar Avenue</street>
  <apt>3</apt>
  <city>Fargo</city>
  <state>ND</state>
  <zipcode>58103</zipcode>
</address>
```

```
<address>
  <street>992 Red Oak Blvd.</street>
  <apt>5F</apt>
  <city>Winnipeg</city>
  <prov>MB</prov>
  <pcode>R2M 2T2</pcode>
</address>
```

Examining a More Complex Choice

- Our *address* element consists of the following:
  - A sequence of *street*, *apt*, *city*
  - Followed by either of:
    - A sequence of *state*, *zipcode*
    - A sequence of *prov*, *pcode*

- We use an *xsd:choice* where we have the choice of one of two sequences

- You can nest *xsd:choice* and *xsd:sequence* elements to build your required content model

Type for More Complex Choice

```
<xsd:complexType name='addressType'>
  <xsd:sequence>

    <xsd:element name='street' type='xsd:string'/>
    <xsd:element name='apt'    type='xsd:string'/>
    <xsd:element name='city'   type='xsd:string'/>

    <xsd:choice>
      <xsd:sequence>
        <xsd:element name='state'   type='xsd:string'/>
        <xsd:element name='zipcode' type='xsd:string'/>
      </xsd:sequence>
      <xsd:sequence>
        <xsd:element name='prov'    type='xsd:string'/>
        <xsd:element name='pcode'   type='xsd:string'/>
      </xsd:sequence>
    </xsd:choice>

  </xsd:sequence>
</xsd:complexType>
```

xsd:all

- Specifies a list of child elements, all of which must be found, **in any order**
  - Only element definitions and element references may be used within *xsd:all*

```
<xsd:complexType name='personType'>
  <xsd:all>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element ref='age'/>
  </xsd:all>
</xsd:complexType>
```

- Both of these instances of *personType* are valid

```
<person>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

```
<person>
  <age>25</age>
  <name>Leanne Ross</name>
</person>
```

- Lab 4.2 – More Complex Schemas -

- **Purpose**: In this lab, we will create a schema that has complex content
  - We will create an XML Schema for a simplified JavaTunes order
- **Objectives**: Work with a more complex schema
- **Builds on previous labs**: Lab 4.1
  - Continue working in your Lab04.1 project
- **Approximate Time**: 30-40 minutes

Element Definitions in a Schema

- Here are the rules for the content models for our simplified order document
  - An ***order*** has a *customer* (we will ignore *items* for now)
  - A ***customer*** has a *name*, a *street*, an *apt*, a *city*, a *state*, a *zipcode*, and a *shipper*
  - ***shipper*** is an empty element
  - All other elements have character content
- Ignore the attributes for now
  - We will define them in the next lab

Simple Order XML Document


```
<?xml version='1.0'?>

<!-- JavaTunes order XML document (simplified) -->

<order>
  <customer>
    <name>Susan Phillips</name>
    <street>763 Rodeo Circle</street>
    <apt>1A</apt>
    <city>San Francisco</city>
    <state>CA</state>
    <zipcode>94109</zipcode>
    <shipper/>
  </customer>
</order>
```

Element Definitions in a Schema

Lab

Tasks to Perform

- Create a schema in file ***order.xsd***, with the following types:
  - ***`orderType`*** **-** a sequence of one element, *`customer`*
  - ***`customerType`*** **-** a sequence of the 7 *`customer`* child elements
  - Use an anonymous type for the ***`shipper`*** element (see notes)
- Then define the document element *`order`*
- Test your schema by validating the *simpleorder.xml* file
  - We supply this in the project (without the schema location)
  - **NOTE -** be sure to add the schema location attribute to the *`order`* element, referring to the schema in *order.xsd* (see notes)

Adding a Choice

Lab

Tasks to Perform

- Enhance the content model for *customerType* to support both US and Canadian customers -- allow a choice of either:
  - *state* and *zipcode*

    OR
  - *prov* and *pcode*
  - This is a choice of two sequences
- Again, experiment with the document, checking its validity

Element Occurrence Constraints

- You can constrain the number of times an element occurs within a complex type
- The attributes ***minOccurs*** and ***maxOccurs*** are used to define the concurrence constraints
- *minOccurs* - minimum number of times element can appear
  - The default is 1
  - Use 0 to make an element optional
- *maxOccurs* - maximum number of times element can appear
  - The default is 1
  - Use *unbounded* to indicate as many as you want

Optional Element - Example

- Make age optional for a person

```
<xsd:element name='person' type='personType'/>

<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'
                 minOccurs='0'/>
  </xsd:sequence>
</xsd:complexType>
```

- The following instance of *personType* is valid
  - A person doesn't need to have an age now

```
<person>
  <name>Leanne Ross</name>
</person>
```

Multiple Occurrences of an Element - Example

- Allow persons to have multiple addresses
  - A person has one or more addresses -- here we are using the previously defined *addressType*

```
<xsd:element name='person' type='personType'/>

<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'
                 minOccurs='0'/>
    <xsd:element name='address' type='addressType'
                 minOccurs='1' maxOccurs='unbounded'/>
  </xsd:sequence>
</xsd:complexType>
```

Multiple Occurrences of an Element - Example

- The following instance of *personType* is valid
  - A person can have many addresses now

```
<person>
  <name>Leanne Ross</name>
  <age>25</age>
  <address>
    <street>125 Fell St.</street>
    ...
  </address>
  <address>
    <street>521 Oak St.</street>
    ...
  </address>
</person>
```

Element Default and Fixed Values

- Default values may be specified using the ***default*** attribute
  - The **element must appear and must be empty** for the default value to be used, e.g., *<age/>*

```
<xsd:element name='age' type='xsd:positiveInteger'
             default='29'/>
```

- Fixed values may be specified using the ***fixed*** attribute
  - Works the same as a default value if the element is empty
  - If the element is present, the only value allowed is that given by the fixed value

```
<xsd:element name='age' type='xsd:positiveInteger'
             fixed='29'/>
```

Element Default and Fixed Values - Example

```
<xsd:element name='age'    type='xsd:positiveInteger'
             default='29' minOccurs='0'/>
```

```
<person>                    <!-- age is provided and is 25 -->
  <name>Leanne</name>
  <age>25</age>
</person>
```

```
<person>                    <!-- age is defaulted to 29 -->
  <name>Leanne</name>
  <age/>
</person>
```

```
<person>                    <!-- age exists but has no value -->
  <name>Leanne</name>
  <age></age>               <!-- age element is not empty -->
</person>
```

```
<person>                    <!-- this person has no age -->
  <name>Leanne</name>
</person>
```

Attribute Definitions

- ***xsd:attribute*** defines attributes for an element
  - **Must occur within a complex type definition**, **after** any child elements have been specified (e.g., in a sequence)
  - Attribute occurrence may also be specified -- required, optional, etc.
  - Attribute is usually defined locally with a simple type
  - May also be a reference to a globally defined (top-level) attribute

```
<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'/>
  </xsd:sequence>
  <xsd:attribute name='gender' type='xsd:string'/>
</xsd:complexType>
```

```
<person gender='F'>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

Attribute Occurrence Constraints

- The ***use*** attribute specifies the attribute's occurrence

  Allowed values are:

  - ***optional*** - attribute may optionally appear
  - ***required*** - attribute must appear
  - ***prohibited*** - attribute must not appear
    - Useful when deriving types from other types, which we cover later
  - Default value is ***optional***

```
<xsd:complexType name='personType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'/>
  </xsd:sequence>
  <xsd:attribute name='gender' type='xsd:string'
                 use='required'/>
</xsd:complexType>
```

Attribute Default and Fixed Values

- Default values may be specified using the ***default*** attribute
  - The default value is inserted when the attribute is missing in a validated document

```
<xsd:attribute name='citizen' type='xsd:string'
               default='US'/>
```

- Fixed values may be specified using the ***fixed*** attribute
  - Works the same as a default value if the attribute is missing
  - If the attribute is present, the only value allowed is that given by the fixed value

```
<xsd:attribute name='verified' type='xsd:string'
               fixed='yes'/>
```

Attribute Occurrence - Example

```
<xsd:complexType name='employeeType'>
  <xsd:attribute name='name'     type='xsd:string'
                 use='required'/>
  <xsd:attribute name='gender'   type='xsd:string'/>
  <xsd:attribute name='verified' type='xsd:string'
                 fixed='yes'/>
  <xsd:attribute name='citizen'  type='xsd:string'
                 default='US'/>
</xsd:complexType>
```

```
<!-- this is valid -->
<employee name='Bob' gender='M' verified='yes'/>
```

```
<!-- this is valid -->
<employee name='Robin' citizen='UK'/>
```

```
<!-- is this valid?  if not, what's wrong? -->
<employee verified='no' name='Leanne' gender='F'/>
```

Legacy Attribute Types from DTDs

- These built-in simple types correspond to the attribute types specified in the XML 1.0 Recommendation
  - It is recommended that these types only be used for attributes

xsd:NMTOKEN xsd:NMTOKENS
xsd:ID xsd:IDREF xsd:IDREFS

- You might use these if your organization is transitioning from DTDs to XML Schema
- You might also use them because they work well in some situations

xsd:NMTOKEN(S) Attributes

- **xsd:NMTOKEN**
  - An XML 1.0 ***NMTOKEN*** is a valid ***XML name token***, i.e., each character must be a **letter**, **number**, **-**, **_**, **.**, or **:**
- **xsd:NMTOKENS**
  - The attribute value is a **list** of name tokens
  - The individual name tokens are delimited by a whitespace character

xsd:NMTOKEN(S) Attributes - Example

```
<xsd:complexType name='employeeType'>
  <xsd:attribute name='name'      type='xsd:string'/>
  <xsd:attribute name='clearance' type='xsd:NMTOKEN'/>
  <xsd:attribute name='candies'   type='xsd:NMTOKENS'/>
</xsd:complexType>
```

```
<!-- these are valid -->
<employee name='Bob' clearance='secret'/>
<employee name='Robin' clearance='ultra'
          candies='blue red green'/>
<employee name='~6@2 *?#^ 6%3!'/>
```

```
<!-- these are not valid - what's wrong? -->
<employee name='Igor' clearance='top secret'/>
<employee name='Ann' clearance='ultra@entrance'
          candies='orange, green, purple'/>
```

xsd:ID-xsd:IDREF(S) Attributes

- ***xsd:ID***
  - An XML 1.0 ***ID*** attribute uniquely identifies its element
  - An element can have no more than one *ID* type attribute
  - The value must be a **valid XML name** (**not** name token) **and** be **unique in the document** (amongst other *ID* attributes)
  - *ID* attributes are often specified as required
  - Supplying a default or fixed value is pointless and not allowed

- ***xsd:IDREF(S)***
  - The value(s) of an ***IDREF(S)*** must be the value(s) of an ***ID*** attribute(s) of other element(s) in the document
  - Used to create internal links between elements
  - *IDREF(S)* type attributes are either required or optional

xsd:ID-xsd:IDREF(S) Attributes - Example

```
<xsd:complexType name='employeeType'>
  <xsd:attribute name='ID'       type='xsd:ID'
                 use='required'/>
  <xsd:attribute name='manager' type='xsd:IDREF'/>
</xsd:complexType>

<xsd:complexType name='deptType'>
  <xsd:attribute name='members' type='xsd:IDREFS'/>
</xsd:complexType>
```

```
<!-- these elements are in one document - it is valid -->
<employee ID='_45910'/>
<employee ID='_83910' manager='_45910'/>
<dept members='_45910 _83910'/>
```

```
<!-- this document is not valid - what's wrong? -->
<employee ID='_1205' manager='_1205'/>
<employee ID='_1003'/>
<employee ID='1205' manager='_1003'/>
<employee ID='_1205'/>
<dept members='_1003 _1205'/>
```

Defining Attributes on a Simple Element

- How can we define the type for the following element?

```
<amount currency='USD'>100.00</amount>
```

- *xsd:complexType* is used to define attributes on an element
  - **But** all of the complex types we've seen so far involve child elements (except *shipper*, which we defined to be empty)
  - We need a way to specify that *amount* has a *currency* attribute and **character content** rather than child element content

- We use ***xsd:simpleContent*** inside *xsd:complexType*, instead of a model group such as *xsd:sequence*
  - Model groups specify child elements -- we don't want that here

Attributes on a Simple Element - Example

```
<!-- anonymous type - could also use a named type -->
<xsd:element name='amount'>
  <xsd:complexType>
    <xsd:simpleContent>
      <xsd:extension base='xsd:decimal'>
        <xsd:attribute name='currency' type='xsd:string'/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:element>
```

- We use ***xsd:extension*** to ***extend*** the base type *xsd:decimal*
  - We are extending the meaning of *xsd:decimal* to say that this element has decimal content **and** an attribute

```
<amount currency='USD'>100.00</amount>
```

- Lab 4.3 – A Complete Order Schema -

- **Purpose**: In this lab, we will learn to use occurrence constraints and attributes, and use them to create a complete order schema
  - We will create a complete Schema for a JavaTunes order
- **Objectives**: Work with occurrence constraints and attributes
- **Builds on previous labs**: Lab 4.2
  - Continue working in your Lab04.1 project
- **Approximate Time**: 40-50 minutes

A Complete Order Schema

- **Purpose** - learn to use occurrence constraints and attributes, and use them to create a complete order schema
- In this lab, we will create a complete schema for a JavaTunes order
- The JavaTunes order schema is *order.xsd*, the one you created in the previous lab
- The JavaTunes order document is *order.xml*, which you used in earlier labs, and which we supply in the lab dir
  - **NOTE** - be sure to add the schema location attribute to the `order` element, referring to the schema in *order.xsd* (see notes)

Element Content Models

- Here are the rules for the element content models:
  - An ***order*** has a *customer* and **1 or more** *items*
  - A ***customer*** has a *name*, a *street*, an **optional** *apt*, a *city*, a **choice** of *state* and *zipcode* or *prov* and *pcode*, and a *shipper*
  - ***shipper*** is an empty element
  - An ***item*** has a *name*, **1 or more** *artists*, a *releaseDate*, a *listPrice*, and a *price*
  - *releaseDate* has *xsd:date* content
  - *listPrice* and *price* have decimal content; *price* has a **default value** of *9.99*
  - All other elements have character content

Attributes

Lab

- Here are the attribute definitions: (see notes for type formats)
- ***order***
  - ***ID*** is type *xsd:ID* and is required
  - ***dateTime*** is type *xsd:dateTime* and is required
- ***shipper***
  - ***name*** is type *xsd:NMTOKEN* and is defaulted to *USMail*
  - ***accountNum*** is type *xsd:string* and is optional with no default
- ***item***
  - ***ID*** is type *xsd:ID* and is required
  - ***type*** is type *xsd:NMTOKEN* and is defaulted to *CD*

Create the Schema

Lab

Tasks to Perform

- Based on the content model in the previous slides, extend your *order.xsd* schema to conform to it
  - We supply a sample order XML document, *order.xml*, that you can validate against
  - Finish the schema, then try validating *order.xml* against it
  - If your schema is correct, then you should not get any errors.
  - See notes for some of the attribute data

Testing

Lab

Tasks to Perform

- Experiment with the order document and check its validity
  - Give an item two artists -- is the document valid?
  - This order's customer has no apartment -- is that valid?
  - Remove a required attribute -- is the document valid?
  - Remove an optional attribute -- is the document valid?
  - Change the shipper name to *UPS Ground* -- is that value okay?

- Are your default values being used?
  - Is the *type* attribute of *item* defaulting to *CD*?
  - Is the *name* attribute of *shipper* defaulting to *USMail*?
  - Is the *price* child element of *item* defaulting to *9.99*? Remember that it must appear as *<price/>* to take the default value

Context-Sensitive Element Definitions

- In XML Schema, we can define elements **specific to a parent element** context

```
<!-- we have two different contexts for name -->
<customer>
  <name title='Ms.'>
    <firstName>Leanne</firstName>
    <lastName>Ross</lastName>
  <name>
</customer>
<item>
  <name>Surfacing</name>
</item>
```

- Thus, we can easily avoid the name collision between the two different JavaTunes order ***name*** elements that we discussed in the Namespaces section
  - A *customer name* is defined differently from an *item name*

Context-Sensitive Element Definitions - Example

```
<xsd:complexType name='customerType'>
 <xsd:sequence>
   <xsd:element name='name'>
     <xsd:complexType>                    <!-- anonymous type -->
       <xsd:sequence>
         <xsd:element name='firstName'  type='xsd:string'/>
         <xsd:element name='lastName'   type='xsd:string'/>
       </xsd:sequence>
       <xsd:attribute name='title'      type='xsd:string'/>
     </xsd:complexType>
   </xsd:element>
 </xsd:sequence>
</xsd:complexType>
```

```
<xsd:complexType name='itemType'>
 <xsd:sequence>
   <xsd:element name='name' type='xsd:string'/>
 </xsd:sequence>
</xsd:complexType>
```

Using XML Schema with Namespaces

- XML Schema has full support for namespaces
- You can specify a ***target namespace*** that a schema is to be used for
  - In XML documents, the elements belonging to the target namespace can be validated against this schema
  - Elements belonging to **another namespace** are **not** validated against this schema -- because this schema does not apply to them
- Schemas are namespace-specific
  - A schema provides definitions for a single target namespace

Schema with Namespace Support - Example

```
<?xml version='1.0'?>

<!-- schema stored in file order.xsd -->
<xsd:schema
 xmlns:xsd='http://www.w3.org/2001/XMLSchema'
 xmlns='http://www.javatunes.com/order'
 targetNamespace='http://www.javatunes.com/order'
 elementFormDefault='qualified'>

  <!-- top-level elements always in target namespace -->
  <xsd:element name='order' type='orderType'/>

  <xsd:complexType name='orderType'>
    <xsd:sequence>
      <!-- locally defined elements are in the target
           namespace IF elementFormDefault='qualified' -->
      <xsd:element name='customer' type='customerType'/>
      ...

</xsd:schema>
```

XML Schema namespace

JavaTunes order namespace (default)

target namespace

XML Document with Namespaces - Example

```
<?xml version='1.0'?>

<!-- point to schema with a schema location attribute -->
<jt:order
 xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
 xmlns:jt='http://www.javatunes.com/order'
 xsi:schemaLocation='http://www.javatunes.com/order
                     file:///StudentWork/XML/order.xsd'>
  <jt:customer>
    ...
  <jt:item>
    ...
</jt:order>
```

XML Schema namespace for instance documents

JavaTunes order namespace

location of schema

for this namespace

- ***xsi:schemaLocation*** specifies a ***namespace URI*** and the ***physical location of the schema*** for that namespace
- We use the ***jt*** prefix to place the elements in the JavaTunes order namespace (or, we could use a default namespace)

- Lab 4.4 – Schema Namespace Support -

- **Purpose**: In this lab, we will add namespace support to our schema and provide context-sensitive element definitions for the name elements
- **Objectives**: Work with Schema and namespaces
- **Builds on previous labs**: Lab 4.3
  - Continue working in your Lab04.1 project
- **Approximate Time**: 40-50 minutes

Namespace Support - Optional

Tasks to Perform

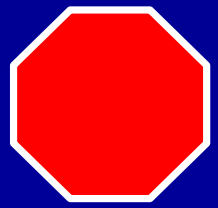
- Make a copy of *order.xsd* and name it ***orderns.xsd***
  - Work in *orderns.xsd* for this lab – this way we'll preserve the non-namespace version

- Add support for the JavaTunes order namespace to the new namespace-based schema
  - You can use the example just shown as your model
  - Recall that in our earlier namespace lab, we used a single order namespace – look at ***orderns.xml*** in this project to see the namespace declaration
  - Recall also that you'll need to use an `elementFormDefault` attribute (see notes)

Context-Sensitive *names* - Optional

Tasks to Perform

- Create a complex type for the ***customer name*** element
  - You can use the example shown a few slides back as your model
  - No change is necessary to the *item name* element definition
- Add the necessary attributes to the XML document to refer to the schema
  - You will use ***xsi:schemaLocation*** instead of *xsi:noNamespaceSchemaLocation*
  - And recall that the value of *xsi:schemaLocation* is a **pair**: ***namespace-URI location-of-schema*** (separated by a space)
  - You can use the example just shown as your model
- Validate the supplied orderns.xml document
  - This document uses the order namespace



Data Modeling with XML Schema

Elements or Attributes?

- The SGML View
- The OO View
- The Middle View
- Pragmatic Considerations
- The Metadata View

Elements or Attributes?

- A decision you will often come across is whether to make a piece of data an **element** or an **attribute**
  - Sometimes this decision is easy and practically made for you
  - Other times it's more difficult
- We will examine several viewpoints on the issue and present some ideas that may help you decide

The SGML View (Noun-Adjective Model)

- SGML has a very clear rule for what should be an element and what should be an attribute
- Elements (nouns) contain the **content** of the document
  - The content is what the author has written, and as such, it should not be changed in any way by the process of being marked up
- Attributes (adjectives) contain information **about** the document -- we refer to this "data about the data" as ***metadata***
  - Things like revision date, author, security classification, draft status, or formatting instructions

Problem with the SGML View

- The problem with this view is that XML is not SGML
  - The primary focus of SGML is documents which are usually going to be read by someone
  - Thus, the SGML view is often called the “visibility” constraint -- if someone is going to see or read it, it should be an element
- XML documents are usually going to be used for transmitting information from system to system
  - They will usually not be read by anyone directly
  - Visibility constraint is less applicable

The OO View (Object-Instance Variable Model)

- In contrast, the OO view looks at the issue in terms of objects and properties
  - In this view, **elements** correspond to **objects** and **attributes** to the (scalar) **instance variables** of those objects

- This is also called the ***container and contents*** view
  - An element corresponds to an object
  - If an object has instance variables which are **complex objects**, these are represented in a natural way as **child elements**
  - If an object has instance variables which are **scalars** (e.g., `int`, `String`), these map in a natural way to **attributes**

The Middle View

- In actual practice, we usually choose a position between the SGML and OO views
  - There is no “right way”
  - One consideration which is often underplayed is, in the intuition of the designer, what representation “feels right” for a data item
  - This is one of the most debated issues in XML -- there is an on-going discussion of this topic at *http://www.oasis-open.org/cover/elementsAndAttrs.html*
- There are also a number of pragmatic considerations which can guide our decision
  - Leave the philosophizing to the academics
  - Let the “physics” of the situation tell you what to do

Pragmatic Considerations - Attributes

- Advantages of attributes include:
  - Most compact way to represent simple name-value pairs
  - Unique identifier types, i.e., *xsd:ID* and *xsd:IDREF*(*S*)
  - More intuitive default value behavior

- Disadvantages of attributes include:
  - Can only represent simple name-value pairs, i.e., scalar data
  - Aren't as convenient for large chunks of data, e.g., a paragraph
  - Can only have zero or one occurrence, i.e., you can't have two attributes with the same name on an element
  - Order cannot be constrained, i.e., attributes on an element can appear in any order

Pragmatic Considerations - Elements

- Advantages of elements include:
  - Substructures and nesting -- attributes cannot represent structure
  - Can flexibly constrain order and occurrence
    - 0 or 1, 0 or more, 1 or more, 1 to 10, exactly 1, exactly 3, etc.
  - More convenient for large chunks of data, e.g., a paragraph

- Disadvantages of (character data) elements include:
  - More verbose (start- and end-tags) than attributes
  - Less intuitive default value behavior

Asking Certain Questions Can Make it Easier

- Is the information hierarchical or flat (scalar)?
  - Hierarchical ⇒ element
  - Flat ⇒ attribute or element (though attribute might win here)

- Is the information ordered or unordered?
  - Ordered ⇒ element
  - Unordered ⇒ attribute

The Metadata View

- If the pragmatic considerations don't answer this question for you, consider the ***metadata*** view
  - If the data is **about** the content, make it an **attribute**
  - If the data **is** the content, make it an **element**

In this example, the *amount* **is** *100.00*, whereas *currency* **describes** the *100.00*

```
<amount currency='USD'>100.00</amount>
```

In this example, *ID* and *date* are data **about** the *order* -- the *items* are the **contents** of the *order*

```
<order ID='_1234' date='2001-02-20'>
  <item partNumber='VT-2112'>
    ...
  <item partNumber='TS-1002'>
    ...
```

Optional Section
Overview of Document Type Definitions

Living with the Legacy of DTDs

- Document Type Declaration
- Defining Elements and Attributes
- Defining General Entities
- Comparison of XML Schema to DTDs

Document Type Declaration

```
<!DOCTYPE type location-of-DTD>
```

```
<?xml version='1.0'?>

<!DOCTYPE person location-of-DTD>

<person>
  <!-- this is a document of type person -->
</person>
```

- Located in the prolog, it specifies the document's ***type***
  - And therefore the document element -- in this case ***person***
- Points to the document's schema, which must be a DTD

Referencing an External DTD - *SYSTEM*

```
<!DOCTYPE type SYSTEM 'system-identifier'>
```

```
<!-- DTD is on an HTTP (Web) server -->
<!DOCTYPE person
  SYSTEM 'http://www.javatunes.com/dtds/person.dtd'>
```

```
<!-- DTD is on a network file system -->
<!DOCTYPE person
  SYSTEM 'file://venus/XML/dtds/person.dtd'>
```

```
<!-- DTD is on the local file system -->
<!DOCTYPE person
  SYSTEM 'file:///StudentWork/XML/dtds/person.dtd'>
```

```
<!-- DTD is in the current directory -->
<!DOCTYPE person SYSTEM 'person.dtd'>
```

- ***system-identifier*** indicates the physical location of the DTD

Referencing an External DTD - *PUBLIC*

```
<!DOCTYPE type PUBLIC 'public-identifier'
                      'system-identifier'>
```

```
<!-- DTD's location is determined, maybe by lookup -->
<!-- if this fails, the system-identifier is used  -->
<!DOCTYPE person
  PUBLIC '-//JavaTunes//Order//EN'
         'http://www.javatunes.com/dtds/person.dtd'>
```

- The application can use the ***public-identifier*** to determine the physical location of the DTD
  - Provides location transparency -- DTD's location can change and the application can still find it
  - How this is handled is completely up to the application
- The system-identifier is used as a backup mechanism

Using an Internal DTD

```
<!DOCTYPE type
[
  definitions
]>
```

```
<!-- DTD is embedded in the XML document -->
<!DOCTYPE person
[
  <!ELEMENT person (name, age)>   <!-- discussed soon -->
```

- Useful when first developing a DTD
  - The DTD and a document of its type are in the same file
- Not very reusable
  - Once the DTD is done, it's generally moved out of the document

Combining an External and Internal DTD

```
<!-- total DTD = internal DTD + external DTD -->
<!DOCTYPE person
  SYSTEM 'http://www.javatunes.com/dtds/person.dtd'
[
  <!ELEMENT person (name, age)>  <!-- discussed soon -->
]>
```

- The internal DTD can provide additional definitions
- It can also **redefine** or **override** certain external definitions
  - Internal DTD has precedence over external DTD
- Internal and external DTDs must be compatible
  - Element definitions cannot be overridden, for example

Defining Element Content in a DTD

```
<!ELEMENT element-name content-model>
```

```
<!-- person has element content - name followed by age -->
<!ELEMENT person (name, age)>

<!-- name and age both have character (text) content -->
<!ELEMENT name (#PCDATA)>
<!ELEMENT age  (#PCDATA)>
```

```
<!-- this is valid -->
<person>
```

- The content model *(name, age)* is called a ***sequence***
  - Each child element **must** appear, and **in this order**
- ***#PCDATA*** indicates **p**arsed **c**haracter **data** (text)

Defining Element Content - *EMPTY* and *ANY*

```
<!ELEMENT element-name EMPTY>
```

```
<!ELEMENT shipper EMPTY>
```

```
<!-- empty elements often have attributes -->
<shipper name='FedEx' accountNum='893-192'/>
```

```
<!ELEMENT element-name ANY>
```

- Elements with *ANY* content can contain anything (or nothing)
  - Usually only used while DTD is still under development
  - You may have decided on the content models for some elements but not all -- leave the unfinished ones as *ANY* for the time being
  - Validation is still possible against this in-progress DTD

Element Occurrence Constraints in a DTD

- When defining an element with **element content**, you can specify the occurrences of its child elements

- In the content model, an ***occurrence indicator*** is appended to the element name
  - ***?*** means **0 or 1** occurrence
  - ******* means **0 or more** occurrences
  - ***+*** means **1 or more** occurrences
  - no indicator means **exactly one** occurrence

Element Occurrence Constraints - Example

```
<!ELEMENT company     (employee+)>

<!ELEMENT employee    (salary, dependent*)>
<!ELEMENT salary      (#PCDATA)>
<!ELEMENT dependent   (firstName, middleName?, lastName)>
<!ELEMENT firstName   (#PCDATA)>
<!ELEMENT middleName  (#PCDATA)>
<!ELEMENT lastName    (#PCDATA)>
```

- In this example:
  - A *company* has **1 or more** *employee*s
  - An *employee* has a *salary* and **0 or more** *dependent*s
  - A *dependent* has a *firstName*, **optionally** a *middleName*, and a *lastName*
  - **NOTE** - these content models are sequences, so order matters

- A valid XML document of type *company* is in the notes

Defining Element Choice in a DTD

```
<!-- a part has a name OR a number -->
<!ELEMENT part (name | number)>

<!-- an address can be US OR Canadian -->
<!ELEMENT address (street, apt?, city,
  (state | prov), (zipcode | pcode))>
```

- The vertical bar **|** is used to indicate a choice between two or more elements
  - Use parentheses for grouping and nesting

- In the second example, we provide for Canadian addresses:
  - The *apt* is optional
  - A *state* or *prov* must be supplied
  - A *zipcode* or *pcode* must be supplied

Occurrence Indicators and Choice

```
<!ELEMENT contacts (name, email, phone)*>
<!ELEMENT contacts (name, email?, phone+)*>
<!ELEMENT contacts (name, (email | phone)+)*>
<!ELEMENT contacts (name | (email, phone))*>
```

- The occurrence indicators (*?*, ***, *+*) can be combined with parentheses to give lots of flexibility
  - A sequence or a choice can be enclosed in parentheses and you can nest these inside other sequences or choices
  - Satisfy the content model in the parentheses, then apply the occurrence indicator

Defining Attributes in a DTD

```
<!ATTLIST element-name attribute1 type occurrence default
                       ...
                       attributen type occurrence default>
```

```
<!ELEMENT person (name, age)>      #REQUIRED
<!ATTLIST person ssn      NMTOKEN
                 gender   (M | F)    #IMPLIED
                 dob      CDATA      #IMPLIED
                 donor    (yes | no) 'yes'>
```

```
<!-- this is valid -->
<person ssn='987-65-4321' gender='F'
        dob='March 2, 1977' donor='yes'>
  <name>Leanne Ross</name>
  <age>25</age>
</person>
```

Attribute Occurrence Constraints in a DTD

- ***#REQUIRED***

  ```
  <xsd:attribute ... use='required'/>
  ```

- ***#IMPLIED***

  ```
  <xsd:attribute ... use='optional'/>
  ```

- ***'default-value'***

  ```
  <xsd:attribute ... default=''/>
  ```

- ***#FIXED 'constant-value'***

  ```
  <xsd:attribute ... fixed=''/>
  ```

Attribute Types in a DTD

- Unlike the text content of an element, attributes are typed
  - ***CDATA***
  - ***NMTOKEN NMTOKENS***
  - ***ID IDREF IDREFS***
  - **enumeration**

  - *NOTATION*
  - *ENTITY ENTITIES*

- We will not cover *NOTATION* and *ENTITY* types, because they are rarely used

CDATA and *NMTOKEN(S)* Attributes

- ***CDATA*** - character **data**

```
<xsd:attribute ... type='xsd:string'/>
```

- ***NMTOKEN(S)***

```
<xsd:attribute ... type='xsd:NMTOKEN'/>
<xsd:attribute ... type='xsd:NMTOKENS'/>
```

ID-IDREF(S) Attributes

- ID

```
<xsd:attribute ... type='xsd:ID'/>
```

- IDREF(S)

```
<xsd:attribute ... type='xsd:IDREF'/>
<xsd:attribute ... type='xsd:IDREFS'/>
```

Enumerated Attributes

- An ***enumeration*** is like a ***NMTOKEN*** but the attribute values are restricted to a defined list
  - This allows for value checking
  - The values in the enumeration must be ***valid XML name tokens***

```
<!ELEMENT employee EMPTY>
<!ATTLIST employee class (ceo | manager | grunt) #IMPLIED
                   perks (lots | few)             'few'>
```

```
<!-- this is valid -->
<employee class='ceo' perks='lots'/>

<!-- this is not valid - what's wrong? -->
<employee class='clerical' perks='none'/>

<!-- what are the attribute values here? -->
<employee/>
```

Defining General Entities

- A ***general entity*** is a piece of XML that you can insert into a document, using an ***entity reference***
  - They can be used in both element content and attribute values
  - The parser performs the process of ***entity replacement***

```
<!ENTITY entity-name 'entity'>
```

```
<!ENTITY copyright 'This is OURS!'>
```

```
<!-- somewhere in an XML document -->
<article owner='&copyright;'>
  <notice>Be warned! &copyright;</notice>
</article>

<!-- after entity replacement, we have this -->
<article owner='This is OURS!'>
  <notice>Be warned! This is OURS!</notice>
</article>
```

JavaTunes Order DTD

```
<!ELEMENT order (customer, item+)>
<!ATTLIST order ID       ID    #REQUIRED
                dateTime CDATA #REQUIRED>

<!ELEMENT customer (name, street, apt?, city,
  ((state, zipcode) | (prov, pcode)), shipper)>
<!ELEMENT name    (#PCDATA)>
<!ELEMENT apt     (#PCDATA)>
<!ELEMENT street  (#PCDATA)>
<!ELEMENT city    (#PCDATA)>
<!ELEMENT state   (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>
<!ELEMENT prov    (#PCDATA)>
<!ELEMENT pcode   (#PCDATA)>

<!ELEMENT shipper EMPTY>

<!ATTLIST shipper name       NMTOKEN 'USMail'
                  accountNum CDATA   #IMPLIED>
```

JavaTunes Order DTD

```
<!ELEMENT item (name, artist+, releaseDate,
  listPrice, price)>
<!ATTLIST item ID   ID      #REQUIRED
               type NMTOKEN 'CD'>

<!-- name element already defined - cannot redefine -->
<!ELEMENT artist      (#PCDATA)>
<!ELEMENT releaseDate (#PCDATA)>
<!ELEMENT listPrice   (#PCDATA)>
<!ELEMENT price       (#PCDATA)>
```

Comparison of XML Schema to DTDs

- Element definitions are “flat” and not hierarchical

- This has several ramifications:
  - No required ordering of the definitions in a DTD
  - No nesting of definitions
  - No context-sensitive element definitions, e.g., no way to distinguish between a ***customer name*** and an ***item name***

  - No designation of document or root element -- any element definition can be a document element

Comparison of XML Schema to DTDs

- Attribute values have some types, but element values do not
  - `<price>abc</price>` is valid with respect to this DTD!
  - And there are no numeric or date types for attributes in DTDs
- Attribute values can take defaults, but element values cannot
- DTDs do not use XML syntax and do not support namespaces
- Content models cannot be defined flexibly
  - Order of child elements must be specified, i.e., there is no equivalent of `xsd:all`
  - It is difficult to specify certain child element occurrences, e.g., between 1 and 3

Optional Section
XML Schema Advanced Topics

- Element Groups
- Attribute Groups
- Deriving Simple Types
- Deriving Complex Types
- The *any* Types

Element Group Definitions

- ***xsd:group*** contains a compositor that acts as an atomic set of elements
  - Defines a reusable group of elements
  - Element group definitions are at the **top level, with a name**

- Element group **references** can be used within *xsd:sequence* and *xsd:choice* model groups
  - May use *maxOccurs* and *minOccurs*

- Similar to *xsd:sequence* embedded within a model group
  - However, *xsd:sequence* is unnamed and specific to a parent element context, therefore it cannot appear at the top level

Element Group Definitions - Example

```
<!-- groups must appear at the top level, with a name -->

<xsd:group name='USaddressGroup'>
  <xsd:sequence>
    <xsd:element name='state'    type='xsd:string'/>
    <xsd:element name='zipcode' type='xsd:string'/>
 </xsd:sequence>
</xsd:group>

<xsd:group name='CAaddressGroup'>
  <xsd:sequence>
    <xsd:element name='prov'     type='xsd:string'/>
    <xsd:element name='pcode'    type='xsd:string'/>
  </xsd:sequence>
</xsd:group>
```

Element Group Definitions - Example

- We can now reference the groups to compose other types

```
<!-- reference the named groups with ref='' -->

<xsd:complexType name='addressType'>
  <xsd:sequence>
    <xsd:element name='street' type='xsd:string'/>
    <xsd:element name='apt'    type='xsd:string'
                 minOccurs='0'/>
    <xsd:element name='city'   type='xsd:string'/>
    <xsd:choice>
      <xsd:group ref='USaddressGroup'/>
      <xsd:group ref='CAaddressGroup'/>
    </xsd:choice>
  </xsd:sequence>
</xsd:complexType>
```

Attribute Group Definitions

- ***`xsd:attributeGroup`*** is like an *`xsd:group`* for attributes
  - Defines a reusable group of attributes
  - Attribute group definitions are at the **top level, with a name**
- Attribute group **references** can be used wherever *`xsd:attribute`* can be used

Attribute Group Definitions - Example

```
<!-- groups must appear at the top level, with a name -->

<xsd:attributeGroup name='itemAttrGroup'>
  <xsd:attribute name='ID'   type='xsd:ID'
                 use='required'/>
  <xsd:attribute name='type' type='xsd:NMTOKEN'
                 default='CD'/>
</xsd:attributeGroup>
```

```
<!-- reference the named groups with ref='' -->

<xsd:complexType name='itemType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    ...
  </xsd:sequence>
  <xsd:attributeGroup ref='itemAttrGroup'/>
</xsd:complexType>
```

Deriving New Types

- With XML Schema, we can create new types based on existing types
- We do this by placing ***restrictions*** on certain ***facets*** of the ***base type***
- We use ***`xsd:simpleType`*** and ***`xsd:complexType`*** to create derived types

Facets

- Every basic datatype is defined in terms of a series of ***facets***
- Facets determine the nature of the datatype
- XML Schema defines 12 facets that can constrain a datatype

xsd:length	***xsd:minInclusive***
xsd:minLength	***xsd:maxInclusive***
xsd:maxLength	***xsd:minExclusive***
xsd:pattern	***xsd:maxExclusive***
xsd:enumeration	***xsd:totalDigits***
xsd:whiteSpace	***xsd:fractionDigits***

Facets

- Each datatype has a set of constraining facets
  - Not all facets apply to each datatype
  - Each datatype may interpret a facet differently

- For ***xsd:string***
  - The *length*, *minLength*, and *maxLength* facets refer to the number of characters in the string
  - The *totalDigits* and *fractionDigits* have no meaning

- For ***xsd:list***
  - The *length*, *minLength*, and *maxLength* facets refer to the number of atomic elements in the list
  - We will look at *xsd:list* in detail later

Deriving New Simple Types

- ***xsd:simpleType*** allows us to derive new simple types
  - Recall that a simple type is a datatype for **values**
  - Used for **text values of elements** and **attribute values**

- We derive a type in terms of ***restrictions***, ***lists***, or ***unions***
  - ***xsd:restriction*** specifies the rules for this derived type
  - ***xsd:list*** specifies that this datatype is a list of another named simple type
  - ***xsd:union*** specifies that this datatype may contain one value from a series of possible datatypes

```
<xsd:simpleType name=''>
USE <xsd:restriction>
OR  <xsd:list>
OR  <xsd:union>
</xsd:simpleType>
```

Deriving New Types by Restriction

- ***xsd:restriction*** defines the rules for the derived type
  - The base datatype is specified with the ***base*** attribute
  - One or more **constraining facets** may then be specified

- Each constraining facet contains two attributes
  - ***value*** is the meaningful value of this constraint - **required**
  - ***fixed*** is a boolean value determining if this facet can be constrained further (by other types derived from this type) - default is *false*

```
<!-- restrict a base type by using various facets -->
<xsd:simpleType name=''>
  <xsd:restriction base=''>
    <someFacet value=''>
    <someFacet value=''>
  </xsd:restriction>
</xsd:simpleType>
```

Deriving New Types by Restriction - Example

```
<!-- create a type with integer values from 1 to 150 -->
<xsd:simpleType name='ageType'>
  <xsd:restriction base='xsd:positiveInteger'>
    <xsd:minInclusive value='1' fixed='true'/>
    <xsd:maxInclusive value='150'/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- reference the type as usual, with type='' -->
<xsd:element name='age' type='ageType'/>
```

```
<!-- this is valid -->
<age>25</age>
```

```
<!-- this is not valid -->
<age>199</age>
```

Deriving New Types by Enumeration

- ***xsd:enumeration*** is a **facet** that allows you to specify a **set of valid values** for a type
  - Like all facets, it appears in the body of *xsd:restriction*
  - This facet appears once for each possible value
  - Each value must be unique and valid for the base type
- In XML documents, the value for such a type must be **one of the values** in the set
  - It is a **single-valued** type

Deriving New Types by Enumeration - Example

```
<!-- create a single-valued type with the 50 US states
     as valid values -->
<xsd:simpleType name='USstatesType'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='AK'/>
    <xsd:enumeration value='AL'/>
    ...
    <xsd:enumeration value='WY'/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- reference the type as usual, with type='' -->
<xsd:element name='state' type='USstatesType'/>
```

```
<!-- this is valid - a single value from the set -->
<state>WY</state>
```

```
<!-- this is not valid -->
<state>XX</state>
```

Deriving New Types by List

- ***xsd:list*** defines a list of values from a set
  - *xsd:list* is **not a facet**
  - It is used instead of *xsd:restriction*
  - The type for the list of values is specified by the ***itemType*** attribute

```
<!-- create a type that is a list of values -->
<xsd:simpleType name=''>
  <xsd:list itemType=''/>    <!-- a list of what type? -->
</xsd:simpleType>
```

- In XML documents, the value for such a type can be a **list of values** of type *itemType*, each one separated by whitespace
  - It is a **multi-valued** type

Deriving New Types by List - Example

```
<!-- create a multi-valued string type -->
<xsd:simpleType name='nameListType'>
  <xsd:list itemType='xsd:string'/>
</xsd:simpleType>
```

```
<!-- this is valid - a list of xsd:string values -->
<name>Jackson Jack Jackie Jackbo</name>
```

```
<!-- create a multi-valued type with the 50 US states
     as valid values -->
<xsd:simpleType name='USstatesListType'>
  <xsd:list itemType='USstatesType'/>
</xsd:simpleType>
```

```
<!-- this is valid - a list of USstatesType values -->
<state>WY AK</state>
```

Further Refining a Derived Type - Example

```
<!-- create a multi-valued string type with <=3 values -->
<xsd:simpleType name='threeNameListType'>
  <xsd:restriction base='nameListType'>
    <xsd:maxLength value='3'/>
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- this is not valid - too many values -->
<name>Jackson Jack Jackie Jackbo</name>
```

- Notice how we derive a new type based on our own list type
  - By simply placing a restriction on it
- You can also use anonymous types in deriving new types
  - See notes below for an example

Deriving New Types by Union

- ***xsd:union*** defines a type as a union of other types
  - *xsd:union* is **not a facet**
  - It is used instead of *xsd:restriction* or *xsd:list*
  - The types of valid values are specified by the ***memberTypes*** attribute

```
<!-- create a type that is a union of other types -->
<xsd:simpleType name=''>
   <xsd:union memberTypes=''/> <!-- what types allowed? -->
</xsd:simpleType>
```

- In XML documents, the value for such a type must be **one of the types** in the union
  - It is a **single-valued** type

Deriving New Types by Union - Example

```
<xsd:simpleType name='statusCodeType'>
  <xsd:restriction base='xsd:positiveInteger'>
    <xsd:maxInclusive value='5'/>
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name='statusNameType'>
  <xsd:restriction base='xsd:string'>
    <xsd:enumeration value='COMMITTED'/>
    <xsd:enumeration value='ROLLED-BACK'/>
    ...
  </xsd:restriction>
</xsd:simpleType>
```

```
<!-- a status type that is a union of the other types -->
<xsd:simpleType name='statusType'>
  <xsd:union memberTypes='statusCodeType statusNameType'/>
</xsd:simpleType>
```

```
<!-- this is valid -->
<status>1</status>
```

```
<!-- this is valid -->
<status>COMMITTED</status>
```

Deriving New Complex Types

- ***xsd:complexType*** allows us to derive new complex types
  - Used for **elements with child elements** and/or **attributes**

- Within *xsd:complexType*, ***xsd:complexContent*** defines a complex type in terms of an existing complex type
  - ***xsd:restriction*** is used to restrict the base type
  - By removing or redefining child elements and attributes

  - ***xsd:extension*** is used to extend the base type
  - By adding child elements and attributes

- Restrictions have a different meaning here
  - There are no constraining facets on complex content; facets apply only to simple types

Deriving New Complex Types - Example

- We start with a complex type named *address*
  - We will then extend and restrict that definition

```
<!-- this is the base type -->
<xsd:complexType name='addressType'>
  <xsd:sequence>
    <xsd:element name='city'    type='xsd:string'/>
    <xsd:element name='state'   type='xsd:string'/>
    <xsd:element name='zipcode' type='xsd:string'/>
  </xsd:sequence>
</xsd:simpleType>
```

```
<!-- this is a valid instance of addressType -->
<address>
  <city>Harrisburg</city>
  <state>Pennsylvania</state>
  <zipcode>17109</zipcode>
</address>
```

Deriving New Complex Types by Extension

```
<!-- extend addressType - add element & attribute -->
<xsd:complexType name='zip-plus4addressType'>
  <xsd:complexContent>
    <xsd:extension base='addressType'>
      <xsd:sequence>
        <xsd:element name='zip-plus4' type='xsd:string'/>
      </xsd:sequence>
      <xsd:attribute name='type'  type='xsd:string'/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:simpleType>
```

```
<!-- this is a valid instance of zip-plus4addressType -->
<address type='billing'>
  <city>Harrisburg</city>
  <state>Pennsylvania</state>
  <zipcode>17109</zipcode>
  <zip-plus4>0775</zip-plus4>
</address>
```

Deriving New Complex Types by Restriction

```
<!-- restrict addressType - redefine & remove elements -->
<xsd:complexType name='simpleAddressType'>
  <xsd:complexContent>
    <xsd:restriction base='addressType'>
      <xsd:sequence>
        <xsd:element name='city'  type='xsd:string'/>
        <xsd:element name='state' type='USstatesType'/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:simpleType>
```

```
<!-- this is a valid instance of simpleAddressType -->
<address>
  <city>Harrisburg</city>
  <state>PA</state>
</address>
```

Generic Type for Elements

- ***xsd:any*** is a wildcard schema component for elements
  - May specify a namespace
  - May use occurrence specifiers
  - No *type* is specified

- By default, *xsd:any* will allow any element from any namespace

```
<!-- a sequence of any one element -->
<xsd:sequence>
   <xsd:any/>
</xsd:sequence>
```

```
<!-- a sequence of any three or more elements -->
<xsd:sequence>
   <xsd:any minOccurs='3' maxOccurs='unbounded'/>
</xsd:sequence>
```

Generic Type for Attributes

- ***xsd:anyAttribute*** is a wildcard schema component for attributes
  - May specify a namespace
  - No *type* is specified

- By default, *xsd:anyAttribute* will allow any attribute from any namespace

```
<!-- a complex type having any attribute -->
<xsd:complexType>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string'/>
    <xsd:element name='age'  type='xsd:positiveInteger'/>
  </xsd:sequence>
  <xsd:anyAttribute/>
</xsd:complexType>
```

- [Optional] Lab 4.5 – Advanced Topics -

- **Purpose**: In this lab, we will use derivation of simple types to refine some of the datatypes in our JavaTunes order schema
- **Builds on previous labs**: Lab 4.4
  - Continue working in your Lab04.1 project
- **Approximate Time**: 40-50 minutes

Derived Types

Lab

Tasks to Perform

- Create simple types to provide the following refinements:
  - Restrict the ***shipper name*** attribute to allow the following values: *USMail*, *FedEx*, *UPS*
  - Create a type for zipcodes (see notes)
  - Restrict the ***state*** element content to actual state abbreviations
  - Don't actually do this for 50 states
  - Pick your 3 favorite states and use them

Testing Our Derived Types

- Modify your schema appropriately to use these new types
  - Simply refer to them by name in the appropriate *type* attributes, e.g.,

```
<element name='zipcode' type='zipcodeType'/>
```

- Test your schema by changing some of the data in an order
  - Try invalid values for the *shipper name* attribute
  - Try 5-digit and zip-plus4 values for the *zipcode* element
  - Try invalid values, as well
  - Try invalid values for the *state* element
- You can do this work in *order.xsd* or *orderns.xsd*

Resources

- **W3C** - World Wide Web Consortium
  - ***http://www.w3.org***
- **OASIS** - Organization for the Advancement of Structured Information Standards
  - ***http://www.oasis-open.org***
- **XML.org** - an industry Web portal formed by OASIS
  - ***http://www.xml.org***