

# ***Fast Track to XML, XSLT, and Java***

---

Version: 20211118

# Course Description

---

- ◆ This is a hands-on course that covers, in depth, core technologies for working with XML, including
  - **Introduction to XML**
  - **The Java APIs for XML**
  - **XPath and XSLT**
- ◆ The workshop contains numerous hands-on lab exercises
  - The labs follow a common fictional case study: **JavaTunes**, an online music store
  - You'll work with XML documents from the JavaTunes system

# 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
  - Understand and use XML namespaces
- ◆ Understand what an **XML parser** is and how to use one
  - Understand the different models of parsing XML documents (push, pull, and tree-based parsers)
  - Use JAXP to parse XML documents using the different parsing models
- ◆ Use **XPath and XSLT** to process XML documents
  - Understand the XPath data model and its representation of an XML document as a tree structure, use XPath expressions and axes
  - Understand the XSLT architecture and how to create and use XSLT stylesheets

- ◆ The workshop has a large number of hands-on labs
  - Many of the labs follow a common fictional case study - JavaTunes, an online music store
  - The labs are contained directly in the course book, and have detailed instructions on what needs to be done
- ◆ We provide setup files containing skeleton code for the labs
  - Students just need to add code for the particular capabilities that they are working with
  - There is also a solution zip file that contains completed lab code
- ◆ Labs details are separate from the lecture manual pages
  - There are placeholder slides in the lecture manual with icons like the Lab icon in the top right corner of this slide

# Course Contents

---

- ◆ Section 1 - **Introduction to XML**
- ◆ Section 2 - **XML Basics**
- ◆ Section 3 - **Namespaces**
- ◆ Section 4 - **Schemas**
- ◆ Section 5 - **JAXP (Java API for XML Processing)**
- ◆ Section 6 - **SAX (Simple API for XML)**
- ◆ Section 7 - **DOM (Document Object Model)**
- ◆ Section 8 - **XML and Java eXtra Topics**
- ◆ Section 9 - **XPath**
- ◆ Section 10 - **XSLT**

# *Introduction to XML*

## *Section 1 - Introduction*

---

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

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

# ***What is XML?***

---

Yet Another Markup Language

# 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

# 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>
  - Currently release is Fifth Edition – Nov. 2008
  - XML 1.1 (2004) - current release Second Edition (2006)
    - Specialized uses only – XML 1.0 is widely used standard
- ◆ 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)

# 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.

# Lab 1.1: Setting up the Environment

---

In this lab, we'll set up Eclipse to work with our XML labs



# ***Introduction to XML***

## ***Section 2 - Basics***

---

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

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

# ***Building Blocks of XML***

---

What's in an XML Document?

# JavaTunes Purchase Order Document - Body

```
<!-- JavaTunes order XML document -->  ← comment
<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>      <!-- must use abbreviation -->
    <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>
```

**Diagram Labels:**

- comment**: points to `<!-- JavaTunes order XML document -->`
- attribute**: points to `dateTime='2001-10-03 09:50'`
- element**: points to the `<item ID='CD509'>` element
- text node**: points to the text `13.99` inside the `<price>` element

# 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**

<code>&lt;br&gt;</code>	<code>&lt;!-- ok in HTML - not ok in XML --&gt;</code>
<code>&lt;br/&gt;</code> or <code>&lt;br&gt;&lt;/br&gt;</code>	<code>&lt;!-- these are equivalent --&gt;</code>

- ◆ Elements must nest properly

<code>&lt;b&gt;&lt;i&gt;no!&lt;/b&gt;&lt;/i&gt;</code>	<code>&lt;!-- ok in HTML - not ok in XML --&gt;</code>
<code>&lt;b&gt;&lt;i&gt;yes&lt;/i&gt;&lt;/b&gt;</code>	

- ◆ 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

**item element**

`<item ID='CD501'>` *start-tag*

```
<name>Diva</name>
<artist>Annie Lennox</artist>
<releaseDate>1992-01-4</releaseDate>
<listPrice>17.97</listPrice>
<price>13.99</price>
```

`</item>` *end-tag*

*content of item*

# 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 -->
   <!-- ok -->
<img src='logo.gif'/>   <!-- ok -->
   <!-- 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  $\mu$ )
  - 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

---

In this lab, we will create a JavaTunes purchase order XML document from an order form

# ***More Building Blocks of XML***

---

## The Finishing Touches

# 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 **;**

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

- ◆ **<** 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 & 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 2.2: Adding a PI

---

In this lab, we will add a processing instruction that transforms our XML into HTML



# *Introduction to XML*

## *Section 3 - Namespaces*

---

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

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

# ***The Motivating Problem***

---

Why Do We Need Namespaces?

# 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

<code>&lt;product&gt;</code> <code>&lt;lowlimit&gt;</code> <code>&lt;uplimit&gt;</code>	<code>&lt;!-- MathML product --&gt;</code>
<code>&lt;product partNumber='A678'&gt;</code> <code>&lt;stock-level&gt;</code>	<code>&lt;!-- your product --&gt;</code>
<code>&lt;formula concise='H 3 N 1'/&gt;</code>	<code>&lt;!-- CML formula --&gt;</code>
<code>&lt;formula&gt;</code> <code>&lt;price&gt;</code> <code>&lt;quantity&gt;</code>	<code>&lt;!-- your formula --&gt;</code>
<code>&lt;MessageHeader&gt;</code> <code>&lt;From&gt;</code>	<code>&lt;!-- ebXML From --&gt;</code>
<code>&lt;Shipment&gt;</code> <code>&lt;From&gt;</code>	<code>&lt;!-- your From --&gt;</code>

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

# ***The Namespace Solution***

---

“Area Codes” for XML Names

# 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

<b>cust:name</b>	=	qualified name
<b>cust</b>	=	prefix
<b>name</b>	=	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

---

- ◆ 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 3.1: Namespaces

---

In this lab, we will add namespace definitions to an XML document

# ***Default Namespaces***

---

Those Prefixes are a Pain

# 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

# 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>
    ...
  </customer>
</order>
```

## Lab 3.2: Default Namespaces

---

In this lab, we will use default namespaces in an XML document

# *Introduction to XML*

## *Section 4 - Schemas*

---

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

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

# ***Valid XML Documents***

---

eXtensible Doesn't Mean Anarchy

# 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

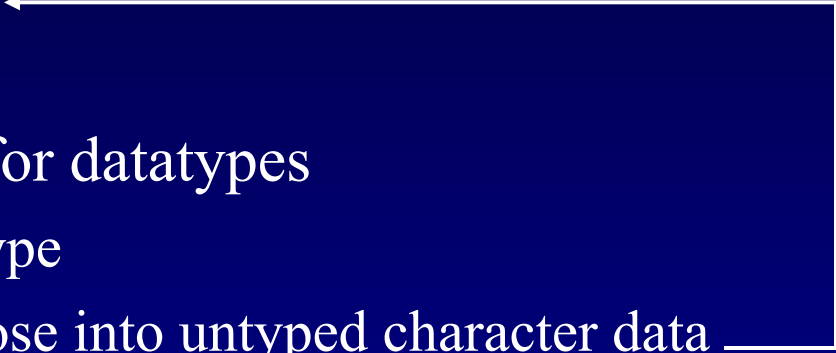
# Schema Languages

---

- ◆ XML inherited the *Document Type Definition* (DTD) syntax from SGML
  - Still being 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 -- now widely 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?

---

- ◆ Much more verbose than DTDs
  - But reducing 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

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

XML Schema  
namespace

```
<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:gYear**      **xsd:gYearMonth**

  - xsd:date**      **xsd:gMonth**      **xsd:gMonthDay**

  - xsd:time**      **xsd:gDay**      **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:long**

**xsd:int**

**xsd:short**

**xsd:byte**

**xsd:positiveInteger**

**xsd:negativeInteger**

**xsd:nonPositiveInteger**

**xsd:nonNegativeInteger**

**xsd:unsignedLong**

**xsd:unsignedInt**

**xsd:unsignedShort**

**xsd:unsignedByte**

## Lab 4.1: Simple Schema

---

In this lab, we will create a very simple XML Schema and validate an instance document against that schema

# 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

---

In this lab, we will create a schema that has complex content



# 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

---

In this lab, we will learn to use occurrence constraints and attributes, and use them to create a complete order schema

# 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
```

XML Schema  
namespace

```
  xmlns:xsd='http://www.w3.org/2001/XMLSchema'
```

```
  xmlns='http://www.javatunes.com/order'
```

JavaTunes order  
namespace  
(default)

```
  targetNamespace='http://www.javatunes.com/order'
```

```
  elementFormDefault='qualified'>
```

target namespace

```
<!-- 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 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)

## [Optional] Lab 4.4: Schema Namespace Support

---

In this lab, we will add namespace support to our schema and provide context-sensitive element definitions for the name elements



# ***Data Modeling with XML Schema***

---

Elements or Attributes?

# 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  $\Rightarrow$  element
  - Flat  $\Rightarrow$  attribute or element (though attribute might win here)
- ◆ Is the information ordered or unordered?
  - Ordered  $\Rightarrow$  element
  - Unordered  $\Rightarrow$  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]***

## ***Overview of Document Type Definitions***

---

Living with the Legacy of 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>
```

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

```
  <age>25</age>
```

```
</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 **d**ata (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** employees
  - An employee has a salary and **0 or more** dependents
  - 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)>  
<!ATTLIST person ssn NMTOKEN #REQUIRED  
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

---

```
<!-- name element already defined - cannot redefine -->
<!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]***

## ***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:minLength**
  - xsd:maxLength**
  - xsd:pattern**
  - xsd:enumeration**
  - xsd:whiteSpace**
  - xsd:minInclusive**
  - xsd:maxInclusive**
  - xsd:minExclusive**
  - xsd:maxExclusive**
  - xsd:totalDigits**
  - 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

---

In this lab, we will use derivation of simple types to refine some of the datatypes in our JavaTunes order schema

# 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>*

# *XML and Java*

## *Section 5 - JAXP*

---

```
<topic title='XML and Java'>
  <section num='5' title='JAXP' />
  <section num='6' title='SAX' />
  <section num='7' title='DOM' />
  <section num='8' title='eXtra Topics' />
</topic>
```

- ◆ XML and Java
- ◆ Generating XML from Java Objects
- ◆ Parsing XML
- ◆ JAXP
- ◆ Instantiating Parsers with JAXP

# ***XML and Java***

---

Portable Makes Things More Potable

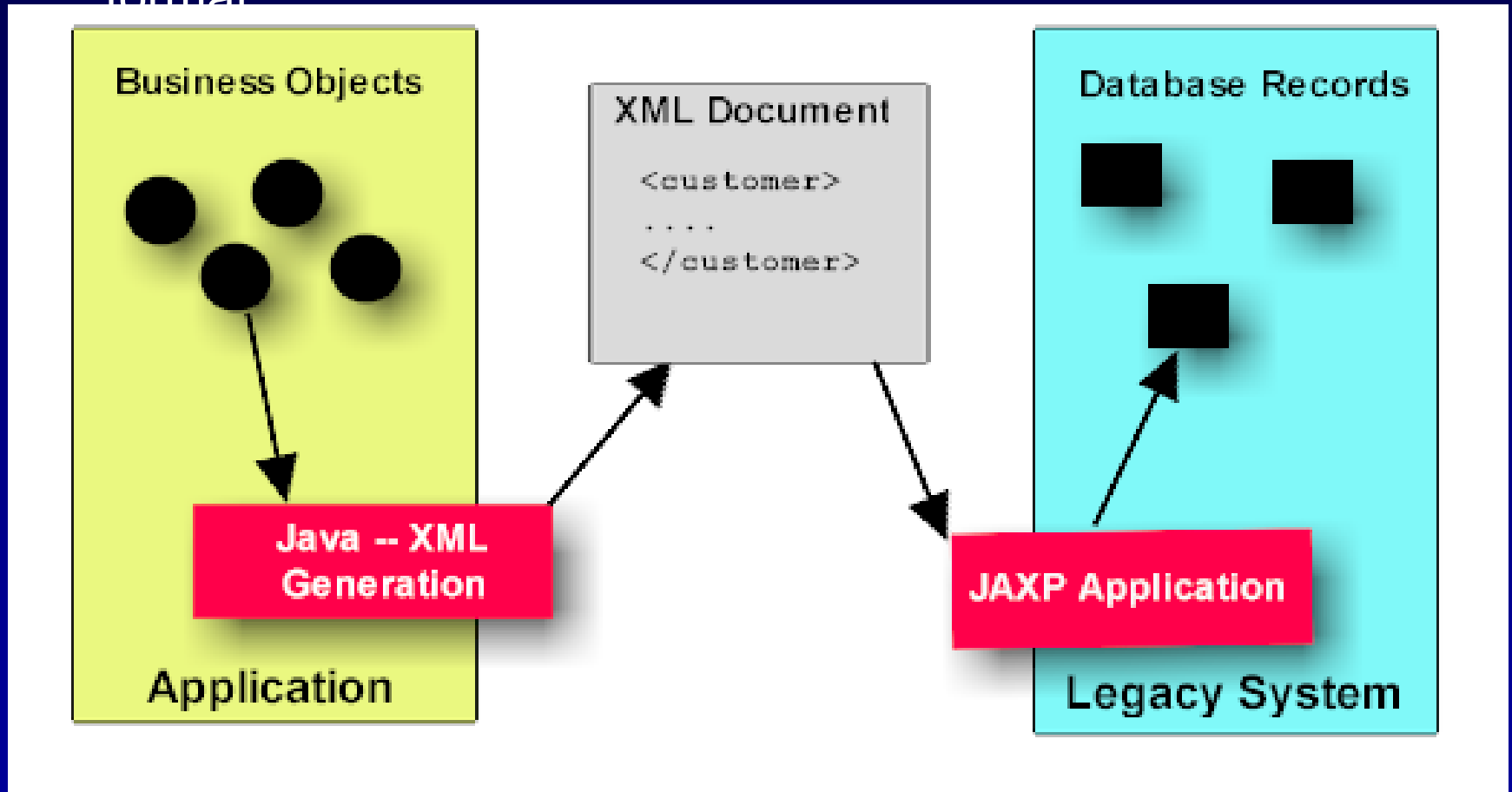
# Portable Code + Portable Data

---

- ◆ Java tends to be the programming language most closely associated with XML
- ◆ Both XML and Java are intended to be used in environments where platform-neutral exchange of information is a requirement
- ◆ Java is often used to “Internet enable” existing legacy applications -  
- these “portals” through which XML will be moved will probably be written in Java

# Application Data Exchange

- ◆ Disparate applications can exchange data via XML
  - XML data are in both an application- and platform-neutral format





# Bridging Application Data and XML

---

- ◆ Java “bridges” application data and its representation in XML
  - Java can **generate** XML, converting application-specific data to application- and platform-neutral XML
  - Java can **parse** XML into data which is used by an application
- ◆ Support for parsing XML documents using Java is via Sun Microsystems’ **JAXP - Java API for XML Processing**
  - **NOTE** that there is no API for **generating** XML

# ***Generating XML from Java Objects***

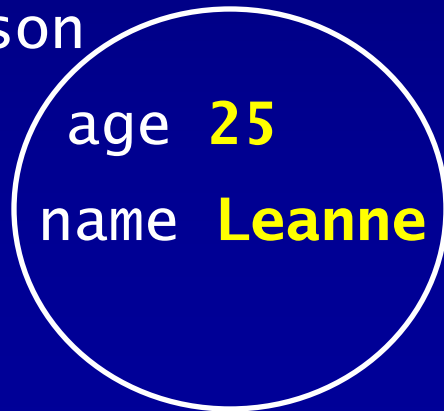
---

Where Do XML Documents Come From?

# Generating XML is Nondeterministic

- ◆ As mentioned earlier, there is no Java “XML generation” API
- ◆ Parsing an XML document is a deterministic process -- there is only one possible way to interpret the markup
  - This allows the construction of a standard API
- ◆ But generating a XML document is nondeterministic -- different kinds of XML can be generated from the same data

Person



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

```
<person age='25'  
  name='Leanne' />
```

# toXML() - Taking a Cue from toString()

---

- ◆ One way to generate XML from Java is to make each object responsible for writing its data into an XML format
  - This idea is similar to the **toString()** method of class `Object`
- ◆ The approach we will use in the following examples is:
  - Each object has a **toXML()** method, in which the object writes itself as an element
  - **Scalar data** (`int`, `String`, etc.) become **attributes** or **child elements**
  - **Complex contained objects** become **child elements**
- ◆ **NOTE** that the objects are writing their data with respect to a specific schema

# Generating XML - Example

---

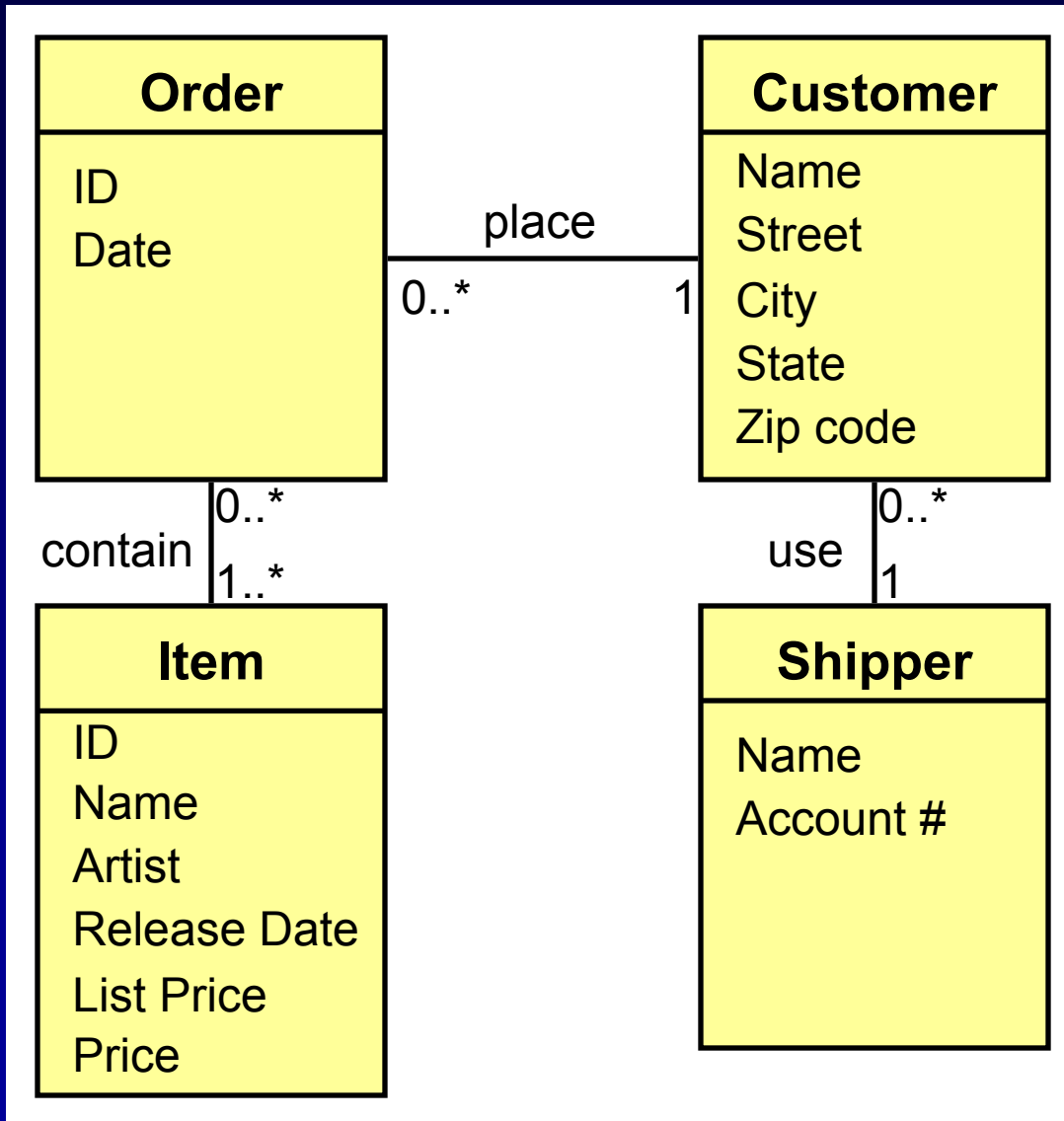
- ◆ We will take a set of JavaTunes application objects that represent an order and generate the appropriate XML
- ◆ First, we need someplace to put each object's XML
  - Each object can return a string containing the XML, like the **toString()** method does, or it can write it to a given stream, e.g., a `FileWriter` -- we will use the first option
- ◆ Then, we need a technique for ensuring that all the objects write their data at the appropriate place, so that we get an XML document with the proper structure
  - Each object will “manage” its contained objects, which correspond to child elements in its content model

# About JavaTunes

---

- ◆ The system domain we will be working with is part of the JavaTunes online music store
  - We will be working with XML documents that carry data pertaining to the operations of JavaTunes
  - Primarily, we will work with an order document that contains data representing a JavaTunes purchase order
  - See the next slides for details

# Purchase Order Data Model - Revisited



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

# Sample JavaTunes Order XML Document

---

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

<!-- JavaTunes order XML document

<order ID='_01170302' dateTime='2002-03-20T05:02:00'
  xmlns:xsi='...' xsi:noNamespaceSchemaLocation='...'>
  <customer>
    <name>Susan Phillips</name>
    <street>763 Rodeo Circle</street>
    <city>San Francisco</city>
    <state>CA</state>
    <zipcode>94109</zipcode>
    <shipper name='UPS' accountNum='343-9080-1' />
  </customer>
  ...
```



# Sample JavaTunes Order XML Document

```
<?xml version='1.0'?>
<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='CD517'>
  <name>1984</name>
  <artist>Van Halen</artist>
  <releaseDate>1984-08-19</releaseDate>
  <listPrice>11.97</listPrice>
  <price>11.97</price>
</item>
<item ID='CD503'>
  <name>Trouble is...</name>
  <artist>Kenny Wayne Shepherd Band</artist>
  <releaseDate>1997-08-08</releaseDate>
  <listPrice>17.97</listPrice>
  <price>14.99</price>
</item>
</order>
```

# JavaTunes Order Schema

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

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

  <xsd:element name='order' type='orderType' />

  <xsd:complexType name='orderType'>
    <xsd:sequence>
      <xsd:element name='customer' type='customerType' />
      <xsd:element name='item' type='itemType' maxOccurs='unbounded' />
    </xsd:sequence>
    <xsd:attribute name='ID' type='xsd:ID' use='required' />
    <xsd:attribute name='dateTime' type='xsd:dateTime' use='required' />
  </xsd:complexType>

  <xsd:complexType name='customerType'>
    <xsd:sequence>
      <xsd:element name='name' type='xsd:string' />
      <xsd:element name='street' type='xsd:string' />
      <xsd:element name='apt' type='xsd:string' minOccurs='0' />
      <xsd:element name='city' type='xsd:string' />

      <!-- customerType continued ... -->
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>
```

# JavaTunes Order Schema

```
<!-- customerType continued ... -->

<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:element name='shipper'>      <!-- uses an anonymous type -->
  <xsd:complexType>
    <!-- no content model, shipper is an empty element (<shipper/>) -->
    <xsd:attribute name='name' type='xsd:NMTOKEN' default='USMail' />
    <xsd:attribute name='accountNum' type='xsd:string'
                                   use='optional' />
  </xsd:complexType>
</xsd:element>
</xsd:sequence>
</xsd:complexType>      <!-- end customerType ... -->

<!-- ... -->
```

# JavaTunes Order Schema

---

```
<!-- ... -->

<xsd:complexType name='itemType'>
  <xsd:sequence>
    <xsd:element name='name' type='xsd:string' />
    <xsd:element name='artist' type='xsd:string' maxOccurs='unbounded' />
    <xsd:element name='releaseDate' type='xsd:date' />
    <xsd:element name='listPrice' type='xsd:decimal' />
    <xsd:element name='price' type='xsd:decimal' default='9.99' />
  </xsd:sequence>
  <xsd:attribute name='ID' type='xsd:ID' use='required' />
  <xsd:attribute name='type' type='xsd:NMTOKEN' default='CD' />
</xsd:complexType>

</xsd:schema>
```

# Some Simplifications for the Example and Lab

---

- ◆ To keep things a bit more simple and compact, we are making the following simplifications:
- ◆ The Customer class does not contain apartment information
  - In the schema, the customer apt element was optional
- ◆ The Customer class only provides for US addresses
  - In the schema, the customer element allowed a choice of state and zipcode or prov and pcode
- ◆ The Item class only provides for a single artist
  - In the schema, an item element could have one or more artists

# toXML() Method - Shipper

- ◆ The Shipper object generates a shipper element

```
class Shipper {  
  
    // instance variables  
    private String m_name = null;  
    private String m_accountNum = null;  
  
    // constructor  
    public Shipper(String name, String accountNum) {  
        m_name = name;  
        m_accountNum = accountNum;  
    }  
  
    // xml generation method  
    public String toXML() {  
        return "<shipper name='" + m_name + "'" +  
               " accountNum='" + m_accountNum + "'/>";  
    }  
}
```

# toXML() Method - Customer

- ◆ The Customer object generates a customer element

```
class Customer {  
  
    // instance variables  
    private String  m_name = null;  
    private String  m_street = null;  
    private String  m_city = null;  
    private String  m_state = null;  
    private String  m_zip = null;  
    private Shipper m_shipper = null;  
  
    // constructor  
    public Customer(String name, String street, String city,  
                    String st, String zip, Shipper ship) {  
        m_name = name;  
        m_street = street;  
        // likewise for the other parameters  
    }  
    // continued ...  
}
```

# toXML() Method - Customer

```
// ... continued
// xml generation method
public String toXML() {
    StringBuffer buffer = new StringBuffer();
    buffer.append("<customer>");
    buffer.append("<name>"      + m_name      + "</name>");
    buffer.append("<street>"    + m_street    + "</street>");
    buffer.append("<city>"      + m_city      + "</city>");
    buffer.append("<state>"     + m_state     + "</state>");
    buffer.append("<zipcode>"  + m_zip      + "</zipcode>");

    // add the shipper's XML, calling its toXML() method
    buffer.append(m_shipper.toXML());

    // close the customer element and return the XML
    buffer.append("</customer>");
    return buffer.toString();
}
}
```



# **[Optional] Lab 5.1 : Generating XML**

---

In this lab, we will create a set of Java classes that can generate XML for an order document

# ***Parsing XML***

---

Getting the Goods out of an XML Document

# Parsing Defined

---

- ◆ The process of determining a document's structure and content from its markup is called *parsing*
  - A parser reports the document information to an application
  - It can do this in different ways
  
- ◆ The XML Recommendation only specifies two things:
  1. How the parser must interpret what it finds in the input stream
  2. What sort of information must be passed to the application
    - It does **not** specify **how** the information is to be returned to the application

# Parser Types

---

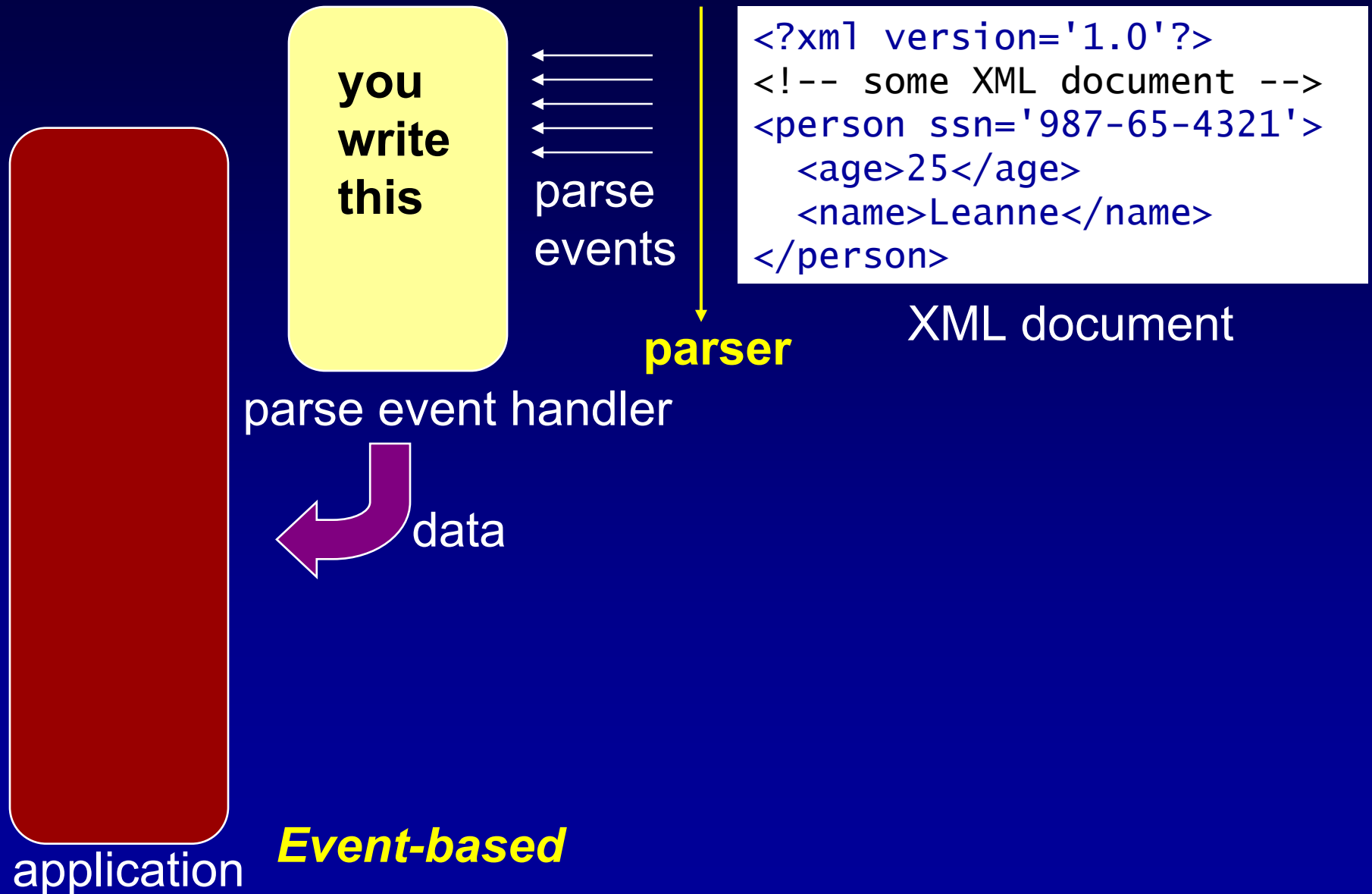
- ◆ There are two basic types of XML parsers
  - They differ in how they report the content of an XML document
- ◆ **Stream-based** parsers parse documents serially at runtime
  - There are two types of stream-based parsers:
  - **Push-based** parsers send (push) XML data to the client (e.g. SAX)
    - Usually these are event-based, where the parsers generate a series of *parse events* as they scan through the document
  - **Pull-based** parsers iterate through a document (e.g. StAX)
    - Generally, the application calls methods on the parser when it needs to access the XML
- ◆ **DOM (Document Object Model)** or *tree-based* parsers return a singly-rooted *tree of nodes* representing the document

# Push (Event) based Streaming Parsers (SAX)

---

- ◆ The parse events are reported to the application, in the order encountered, and then forgotten by the parser
  - The application must implement a *handler* for the parse events and preserve the data associated with those events
  - Generally, each item of markup or chunk of content generates a specific parse event, e.g., finding a start-tag or an end-tag
- ◆ Event-based parsers tend to be very fast and use little overhead, even for very large documents
  - Event-based parsers are ideal for scanning through documents to extract specific information

# Parsing Illustrated – Push-Based /SAX (Event)

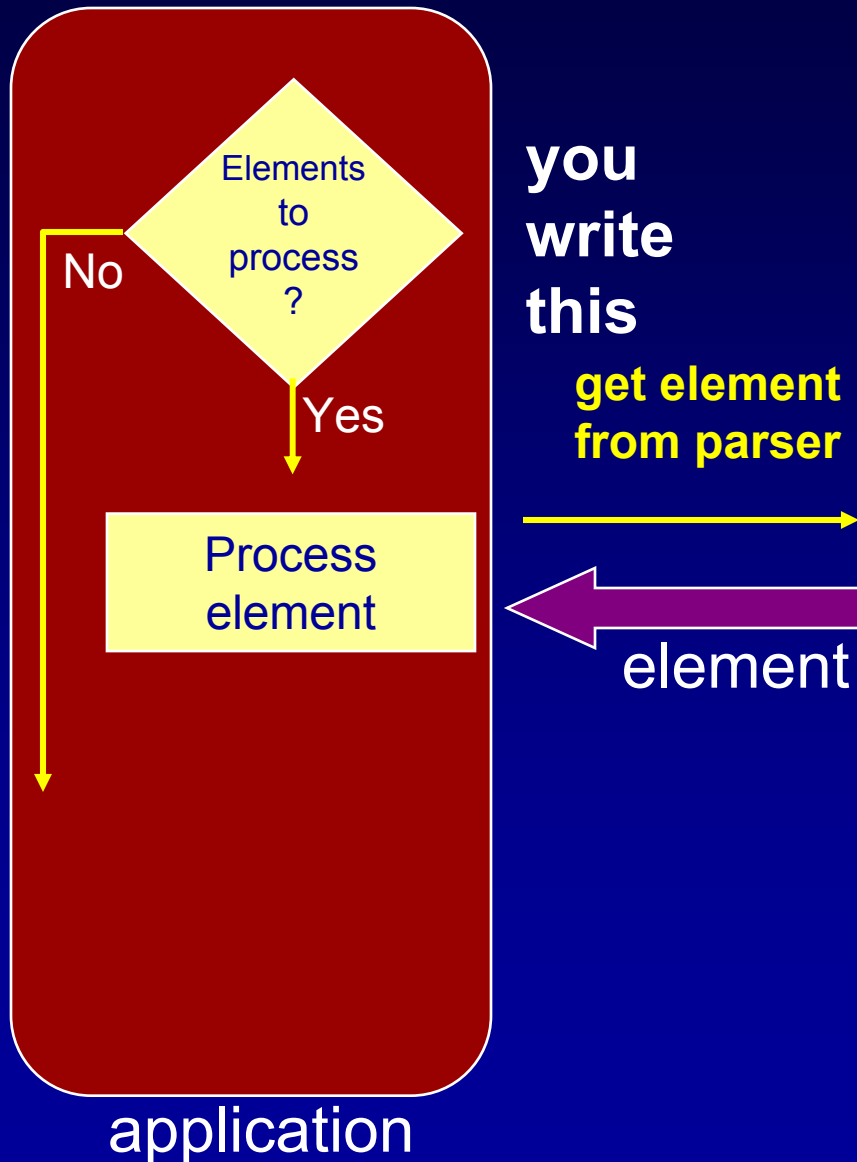


# Pull-based Streaming Parsers (StAX)

---

- ◆ The application calls methods on an XML parsing library when it is ready to interact with the data
  - The client only gets (pulls) information when it explicitly asks for it
  - The client controls movement through the document
  - When asked, the parsing library returns the current item of markup or chunk of content (a start-tag, content, an end-tag)
- ◆ Pull-based parsers tend to be very fast and use little overhead, even for very large documents
  - Usually faster and have less overhead than event-based
  - Pull-based parsers are often the easiest for scanning through documents to extract specific information

# Parsing Illustrated – Pull-Based / StAX (Iteration)



```
<?xml version='1.0'?>
<!-- some XML document -->
<person ssn='987-65-4321'>
  <age>25</age>
  <name>Leanne</name>
</person>
```

XML document

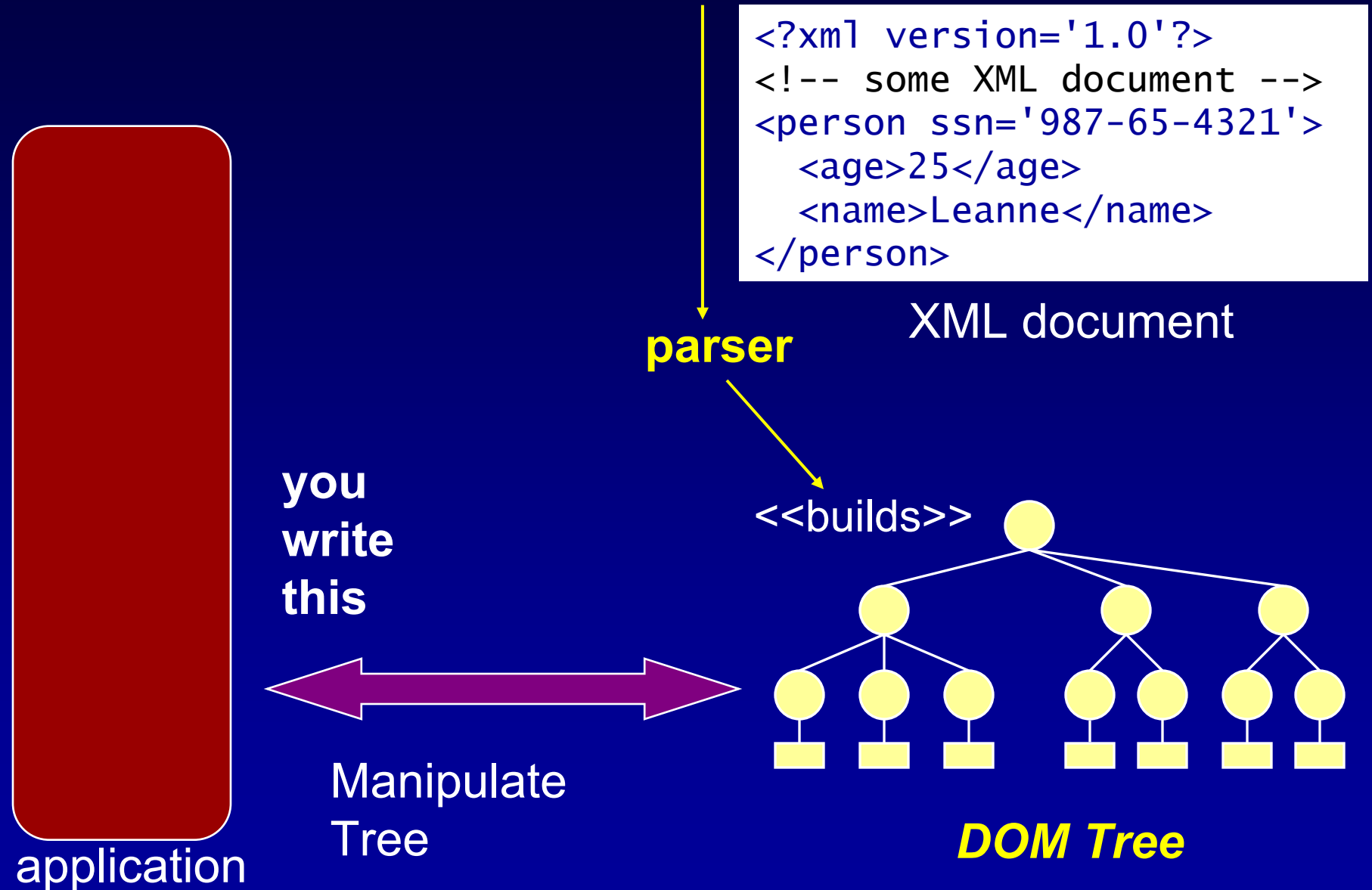


# DOM-Based (Tree) Parsers

---

- ◆ While parsing, they build a tree representation of the document in memory
  - The document is then available for manipulation through an API, and a new XML document can be generated from the modified tree
- ◆ For large documents, tree-based parsers can be resource intensive, both in terms of memory and speed
  - But they provide read/write access to the data

# Parsing Illustrated – DOM-Based (Tree)



# Validating Parsers

---

- ◆ Both stream-based and tree-based parsers can be validating or nonvalidating
  - Validating parsers ensure that a document is well-formed **and** valid
  - Nonvalidating parsers only ensure that the document is well-formed, and may or may not consider the schema, if one is provided
- ◆ Many parsers have the option to be run in validating mode or nonvalidating mode
  - Nonvalidating mode is usually faster -- why?

# ***JAXP***

---

## A Toolbox for Processing XML

# What is JAXP?

---

- ◆ JAXP is Sun Microsystems' ***Java API for XML Processing***
  - It is a collection of several APIs, along with implementations of each
  - It also provides an implementation-neutral mechanism for working with other XML parsers, allowing **parser independence**
- ◆ JAXP 1.6 (available in Java 8+) currently supports:
  - **XML 1.0 / XML 1.1**
  - **SAX 2.0.2 event-based** parser      interface and implementation
  - **StAX 1.2 pull** parser      interface and implementation
  - **DOM Level 3 tree-based** parser      interface and implementation
  - **XSLT 1.0** transformer      interface and implementation
- ◆ JAXP is bundled with the Java JDK

# SAX 2.0

---

- ◆ JAXP supports **SAX** - ***Simple API for XML***
- ◆ SAX is a de facto standard that defines a set of interfaces for **event-based** parsers
  - It establishes **how** event-based parsers must return document information to an application
  - It is defined by its Java interfaces
- ◆ There is no published SAX standard – it is defined solely by the widely accepted Java interfaces
  - But everybody supports and uses it
- ◆ The SAX Web site is ***<http://www.saxproject.org>***

# StAX 1.x

---

- ◆ JAXP supports **StAX** - ***Steaming API for XML***
- ◆ StAX is Java standard defining interfaces for a **pull** parser
  - It establishes **how** applications pull document information from the parser
  - It is defined by a specification and a set of Java interfaces (JSR-173)
- ◆ StAX was designed as a median between the two opposites of event-based and tree based parsers
  - It is fast and small
  - It is easy to used, as processing is under application control, and not event driven

# DOM

---

- ◆ JAXP supports **DOM** - *Document Object Model*
- ◆ DOM is a W3C Recommendation that defines a set of interfaces for **tree-based** parsers
  - It establishes **how** tree-based parsers must return document information to an application
  - It describes what the tree representation of the document looks like, and how it may be manipulated



# Differences between Stream and DOM Based

---

## Stream-based

- ◆ Scans the document from beginning to end -- does not store document data in memory
- ◆ Read-only -- no support for modifying the document
- ◆ Document data is available as soon as it is parsed

## DOM-based

- ◆ Builds an in-memory tree structure representing the document
- ◆ Read/write -- in-memory tree can be randomly accessed and modified
- ◆ Tree is not made available until after the entire document has been parsed

# XSLT 1.0

---

- ◆ JAXP supports **XSLT** - *eXtensible Stylesheet Language for Transformations*
- ◆ XSLT is a W3C Recommendation for XML transformations
  - It defines a language for transforming XML documents into other forms, including XML and HTML
- ◆ As with parsers, there are a number of XSLT processors
  - JAXP provides a default XSLT processor, and is designed to allow the use of other processors

# ***Instantiating Parsers with JAXP***

---

Getting Parsers in a Portable Way

# Pluggable Layers

---

- ◆ **GOAL** - switch parsers or XSLT processors in an application without having to change any code
  - You specify what you want via a *system property*
- ◆ To achieve this, JAXP introduces a layer of indirection using *factory methods* to instantiate a parser or XSLT processor
- ◆ Not all parsers and XSLT processors support this, but most do
  - It is not required in any of the standards
  - Some parsers may have to be instantiated directly
    - Ask the vendor when they will support JAXP!

# JAXP Specification

---

- ◆ JAXP is both a **specification** and an **implementation** that became a standard part of Java with the Java 5 release
  - Including the SAX, StAX, DOM, and XSLT APIs, as well as support for XML Schema and validating against a schema
  - It also specifies how to achieve parser pluggability
- ◆ The JDK includes both the API classes, and the Xerxes parser implementation
  - This implementation supports all the JAXP supported APIs
  - The JDK also includes a fast XSLT processor (Xalan in Java 6)
- ◆ All of these are bundled with the Java JDK
  - It is also possible to supply your own parser, and configure JAXP to use it via system properties

# Other Parsers and XSLT Processors

---

- ◆ The JAXP default implementations are good enough for most XML work -- however, it may become necessary or desirable to use other parsers or XSLT processors
  - Most application servers provide XML parsers -- using them may provide platform-specific advantages
- ◆ To set up another parser:
  - Obtain a copy of the parser
  - Most parsers are packaged as a JAR file -- place the JAR file on the classpath
  - Check the documentation to see if the parser supports the JAXP pluggability layer -- if not, write the code to instantiate it properly (and ask the vendor when their parser will support JAXP!)

# Factory Objects

---

- ◆ To instantiate a parser in JAXP, we use a *factory object*, which creates a parser object for us
- ◆ A factory object can produce only one kind of parser
  - For example, a Xerces factory can only produce a Xerces parser
- ◆ There are two kinds of factories
  1. **SAXParserFactory**, which makes a **SAXParser**
  2. **DocumentBuilderFactory**, which makes a **DocumentBuilder** (the JAXP name for a DOM-compliant parser)
- ◆ All four of these classes are in package **javax.xml.parsers**

# Instantiating a SAXParser - Example

```
import javax.xml.parsers.*;
class SAXExample {
    public static void main(String[] args) {
        SAXParserFactory factory = null;
        SAXParser parser = null;
        try {
            factory = SAXParserFactory.newInstance();

            // want a validating and namespace-aware SAX parser
            factory.setValidating(true);
            factory.setNamespaceAware(true);

            // get a parser from the factory
            parser = factory.newSAXParser();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```



# Instantiating a `DocumentBuilder` - Example

```
import javax.xml.parsers.*;
class DOMExample {
    public static void main(String[] args) {
        DocumentBuilderFactory factory = null;
        DocumentBuilder parser = null;
        try {
            factory = DocumentBuilderFactory.newInstance();

            // want a validating and namespace-aware DOM parser
            factory.setValidating(true);
            factory.setNamespaceAware(true);

            // get a parser from the factory
            parser = factory.newDocumentBuilder();
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

# Instantiating an XSLT Transformer - Example

```
import javax.xml.transform.*;

class XSLTExample {
    public static void main(String[] args) {
        TransformerFactory factory = null;
        Transformer xformer = null;

        try {
            factory = TransformerFactory.newInstance();

            // get an XSLT transformer from the factory
            // need to specify its XSLT stylesheet at this point
            xformer = factory.newTransformer(/* stylesheet */);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

# Validating against a W3C XML Schema

---

- ◆ The JAXP specification includes a standardized way to specify validation against a W3C XML Schema
  - It uses a property-based mechanism to accomplish this
  - The property name and value are, respectively:
    - *<http://java.sun.com/xml/jaxp/properties/schemaLanguage>*
    - *<http://www.w3.org/2001/XMLSchema>*
- ◆ There is a similar mechanism for specifying schema source
  - *<http://java.sun.com/xml/jaxp/properties/schemaSource>*
  - **The schema's URI, InputStream, InputSource, or File**
- ◆ Both of these properties take effect only if:
  - Validation has been turned on
  - The parser supports W3C XML Schema

# Validating against an XML Schema - Example

```
// NOTE that for SAX you use setProperty() on the parser
```

```
factory.setValidating(true);
```

```
parser = factory.newSAXParser();
```

```
parser.setProperty(  
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",  
    "http://www.w3.org/2001/XMLSchema");
```

```
// NOTE that for DOM you use setAttribute() on the factory
```

```
factory.setValidating(true);
```

```
factory.setAttribute(  
    "http://java.sun.com/xml/jaxp/properties/schemaLanguage",  
    "http://www.w3.org/2001/XMLSchema");
```

```
parser = factory.newDocumentBuilder();
```

# Using Both a DTD and a W3C XML Schema

- ◆ You can provide both of these to a parser
  - Use both a `<!DOCTYPE>` and a schema location attribute
  - You might do this while in transition from DTDs to XML Schemas
  - You might also use DTDs solely for your own general entities, e.g.,  
DTD: `<!ENTITY copyright 'This is OURS! '>`  
XML document: `<notice>&copyright;</notice>`
- ◆ The JAXP specification states that the DTD may **not** be used for **validation**
  - However, it might be used for general entity values and for attribute defaults (see notes)

```
<!-- this document refers to a DTD and an XML Schema -->
<!DOCTYPE order SYSTEM '... '>
<order xmlns:xsi='...' xsi:noNamespaceSchemaLocation='...'>
  ...
```

# Factories and System Properties

---

- ◆ A **system property** is consulted in order to figure out which type of factory to make
  - There is a property for SAX and also one for DOM
  - **javax.xml.parsers.SAXParserFactory**
  - **javax.xml.parsers.DocumentBuilderFactory**
  - The value of the system property is the **fully-qualified name of the parser implementation class**
  - We will use the standard parsers for JAXP, and so won't use these
  - If you wanted a different parser, you'd need to check their documentation to find out the correct value for these
- ◆ Once the factory object is created, we can then set a number of properties on the **factory** to specify parser capabilities
  - For example, we can tell the factory whether or not to create a validating parser

# Exception Handling

---

- ◆ A factory's **newInstance()** method can throw a **FactoryConfigurationError**
  - Usually thrown when the class name of the factory specified in the system property cannot be found -- the class name is probably incorrect or the parser's JAR file is not on the classpath
- ◆ A factory's **newSAXParser()** or **newDocumentBuilder()** method can throw a **ParserConfigurationException**
  - Parser configuration cannot be honored, e.g., you request a validating parser from a factory that produces only nonvalidating parsers
- ◆ Both of these classes are in package `javax.xml.parsers`

## Lab 5.2: Creating Parsers

---

In this lab, we will write classes that instantiate SAX and DOM parsers



# *XML and Java*

## *Section 6 - SAX*

---

```
<topic title='XML and Java'>
  <section num='5' title='JAXP' />
  <section num='6' title='SAX' />
  <section num='7' title='DOM' />
  <section num='8' title='eXtra Topics' />
</topic>
```

- ◆ Primary Components of SAX
- ◆ Setting up for a Parse
- ◆ Handling Document Errors
- ◆ Handling Document Content
- ◆ SAX Features and Properties
- ◆ SAX Extensions

# ***Primary Components of SAX***

---

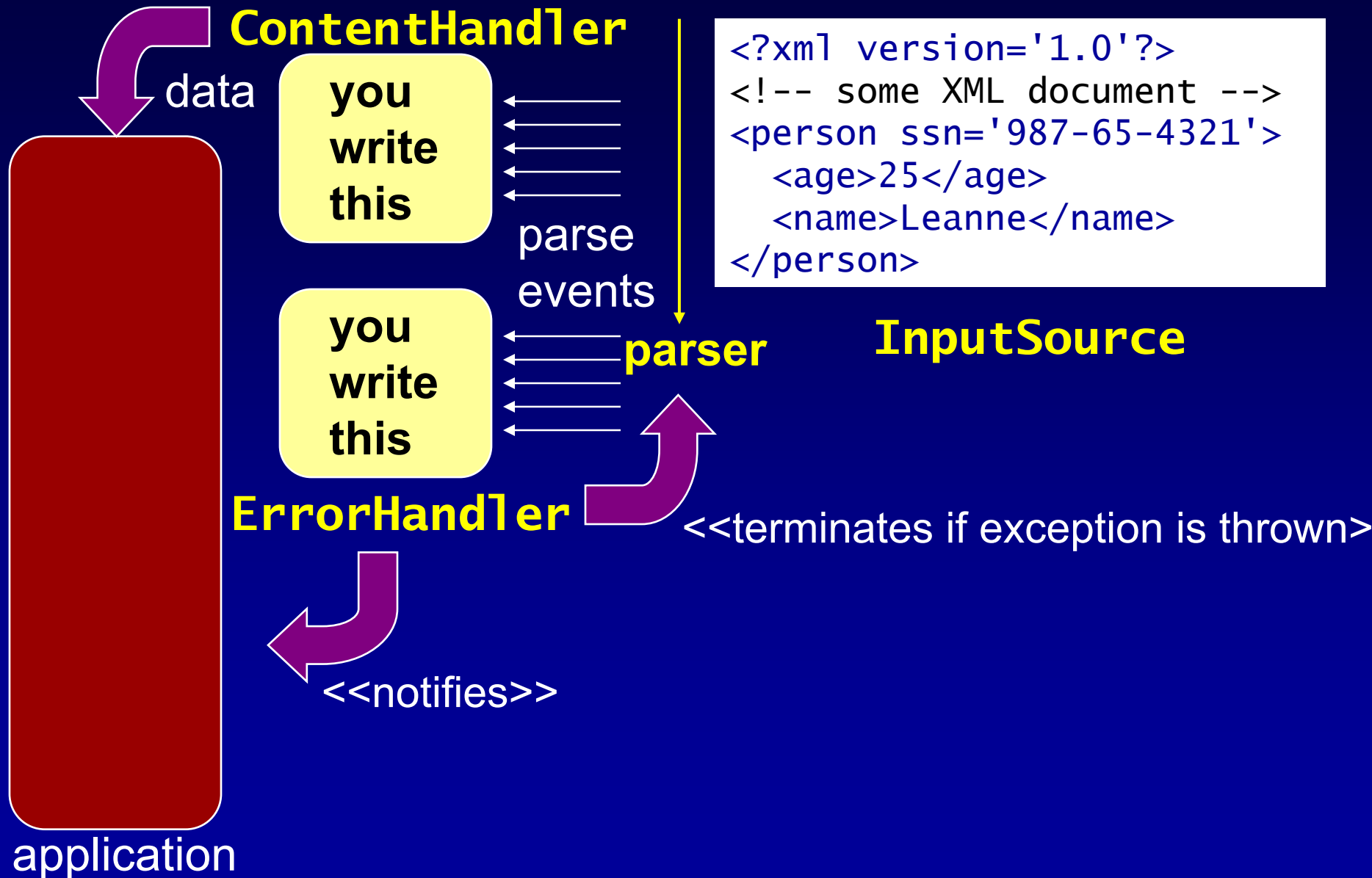
80% of the Work is Done with 20% of the Stuff

# SAX Components and Packages

---

- ◆ SAX is divided up into two sets of components
- ◆ Package **org.xml.sax** contains the SAX **core** components
  - These **must** be implemented by a SAX-compliant parser
- ◆ Package **org.xml.sax.ext** contains the SAX **extensions**
  - These may optionally be supported -- most parsers implement them
- ◆ Package **org.xml.sax.helpers** contains helper classes
  - We will use **DefaultHandler** as a base class for our document handling code

# SAX Parsing Illustrated - Main Components



# Primary SAX Components - Provided for You

---

## ◆ XMLReader (via SAXParser)

- XMLReader is implemented by the parser
- It has methods to register our handlers and to start parsing
- It also has methods to set and query various features and properties, such as validation and namespace-awareness
- **SAXParser is a wrapper around an underlying XMLReader** -- we will generally use SAXParser instead of using XMLReader directly

## ◆ InputSource

- A class which is used to encapsulate the various sources of an XML document into a common input type for an XMLReader

# Primary SAX Components - You Implement

---

## ◆ **ContentHandler**

- An object implementing ContentHandler is delivered the document content by the parser, in the form of *parse events* (method callbacks)
- It has code to do something with the content of an XML document
- You will almost always implement this interface

## ◆ **ErrorHandler**

- An object implementing ErrorHandler is delivered parse-related errors, which could be due to well-formedness or validity violations
- It is strongly recommended that you implement this interface

- ◆ Objects implementing these interfaces are called *event handlers*

# The DefaultHandler Class

- ◆ In order to do a simple parse, we would have to write classes that implement the handler interfaces
  - ContentHandler has 11 methods!
- ◆ To make life easier, you can use a SAX helper class which implements all of the handler interfaces by providing default implementations of the interface methods
  - Subclass DefaultHandler and override the methods you care about

```
public class DefaultHandler
implements ContentHandler, ErrorHandler,
           EntityResolver, DTDHandler    // covered later
{
    public void startDocument() { }    // from ContentHandler
    // likewise for all methods defined in the interfaces
}
```

# SAX Exceptions

---

- ◆ SAX provides a base exception class called **SAXException**, which extends `Exception`
- ◆ **SAXParseException**
  - This is the primary exception class that we will be dealing with
  - It is thrown when the parser encounters errors in a document
  - It extends `SAXException` and allows for additional information to be reported, such as line numbers in the document
- ◆ **SAXNotRecognizedException**  
**SAXNotSupportedException**
  - These are thrown during the configuration of the parser's features or properties
  - Both extend `SAXException`



# ***Setting up for a Parse***

---

Give the Parser What it Wants

# Ingredients for a Parse

---

- ◆ We need a **SAXParser**, an **InputSource**, and a **handler**
  - Instantiate a SAXParser
  - Create an InputSource object representing the XML
  - Write a handler class and instantiate it
  - Call SAXParser's **parse()** method, passing in the InputSource and handler objects

```
public void parse(InputSource in,  
                  DefaultHandler handler)  
    throws SAXException, IOException
```

- ◆ In calling **parse()**, the parser reads the document and sends the parse events to your handler
  - Get ready handler, here comes the goods

# Setting up a Handler and Input Source

- ◆ To write a handler, you can subclass `DefaultHandler` and override the methods you care about

```
import org.xml.sax.helpers.DefaultHandler;
import org.xml.sax.*;

class MyHandler extends DefaultHandler
{
    // we'll override the parse event callback methods soon
}
```

- ◆ To create an `InputSource`, just pass in the actual source of the XML to one of its constructors
  - `InputSource(InputStream byteStream)`
  - `InputSource(Reader charStream)`
  - `InputSource(String uri)`

```
InputSource in = new InputSource(new FileReader("f.xml"));
```

# Parsing an XML Document - Example

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.FileReader;

class SAXExample {
    public static void main(String[] args) {
        // NOTE - factory and parser code not shown in example
        // declare handler and XML source
        MyHandler handler = null;
        InputSource source = null;
        try {
            // instantiate handler and XML source
            handler = new MyHandler();
            source = new InputSource(new FileReader(args[0]));

            // parse the XML document - pass in source & handler
            parser.parse(source, handler);
            System.out.println("\nParse successful.");
        }
        catch (SAXException e) ...
    }
}
```

# Handling Setup and Parse Exceptions

---

- ◆ The factory's **newSAXParser()** method can throw a **ParserConfigurationException**
- ◆ The parser's **parse()** method can throw a **SAXException** or an **IOException**
  - We'll soon see that parse-related errors are sent to your `ErrorHandler` as SAX error parse events
- ◆ These are all checked exceptions, so you will have to use a `try` block

## Lab 6.1: Your First Parse

---

In this lab, we will write some simple handlers for the SAX parser

# ***Handling Document Errors***

---

What to Do When Things Go Wrong

# Parsing Exceptions - Observations from the Lab

---

- ◆ When the parser encountered XML that was **not well-formed** (*testFatalError.xml*), it threw a `SAXParseException` at the method that invoked `parse()`
  - In our case, that was the `main()` method of `SAXTest`
- ◆ However, when the parser parsed an **invalid** document (*testError.xml*), no exception was thrown, even though we were using a validating parser
  - Actually, **our handler was told about the validity error**, but it doesn't do anything with the parse events yet
- ◆ We define how to handle parse errors by implementing the methods in the **ErrorHandler** interface



# SAX Errors and SAX Warnings

---

- ◆ A SAX **error** is almost always caused by the **violation of a validity constraint**
  - This parse event is delivered to our ErrorHandler
  - Since the document can still be parsed we do not have to abort the parse on an error -- we can ignore the error
  - If we do want to abort the parse, our ErrorHandler throws an exception at the parser
- ◆ A **warning** in SAX is any exceptional condition that is not defined in the XML Recommendation as a fatal error or error
  - Again, this parse event is delivered to our ErrorHandler
  - We may try to recover and not abort the parse
  - If we do want to abort the parse, our ErrorHandler throws an exception at the parser

# SAX Fatal Errors

---

- ◆ A SAX *fatal error* is generally caused by a **well-formedness violation**
  - This parse event is first delivered to our ErrorHandler
  - BUT**
  - **The parse will abort** after our ErrorHandler is invoked
  - Even if the handler does not throw an exception at the parser
- ◆ The ErrorHandler can do application-specific processing before the parse terminates
  - For example, it can update an error log
  - It **cannot** signal to the parser to “forget about it and continue”

# ErrorHandler Interface

```
public void warning(SAXParseException e)  
throws SAXException
```

```
public void error(SAXParseException e)  
throws SAXException
```

```
public void fatalError(SAXParseException e)  
throws SAXException
```

- ◆ There is a method defined in the `ErrorHandler` interface for each type of SAX error
- ◆ If you want to abort the parse on an error or warning, throw the `SAXParseException` parameter back at the parser
  - Or, you can create and throw your own `SAXException`
  - The parse **will abort** after `fatalError()` is called

# SAX Error Events Illustrated

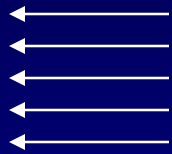
```
warning(... e)
    throw e; // abort

error(... e)
    throw e; // abort

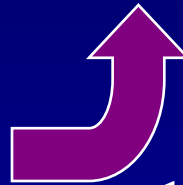
fatalError(... e)
    // always aborts
```

## ErrorHandler

error  
parse  
events



**parser**



```
<?xml version='1.0'?>
<!-- some XML document -->
<person ssn='987-65-4321'>
    <age>25</age>
    <name>Leanne</name>
</person>
```

<<terminates if exception is thrown>

- ◆ To abort the parse on a warning or an error, we have the ErrorHandler throw an exception back at the parser
- ◆ If a fatal error is encountered, the parser will abort

## Lab 6.2: Sax Error Handling

---

In this lab, we will write error handlers for the SAX parser

# ***Handling Document Content***

---

How is the Document Data Delivered?

# ContentHandler Interface

---

```
public void startDocument()  
throws SAXException
```

```
public void endDocument()  
throws SAXException
```

```
public void startElement(String nsURI, String localName,  
                        String qName, Attributes atts)  
throws SAXException
```

```
public void endElement(String nsURI, String localName,  
                      String qName)  
throws SAXException
```

```
public void setDocumentLocator(Locator locator)
```

- ◆ You can probably figure out when these methods are called
  - We will discuss the document locator shortly

# ContentHandler Interface

---

```
public void characters(char[] data, int start, int length)
throws SAXException
```

```
public void ignorableWhitespace(char[] data,
                                int start, int length)
throws SAXException
```

```
public void processingInstruction(String target,
                                   String data)
throws SAXException
```

```
public void startPrefixMapping(String prefix, String uri)
throws SAXException
```

```
public void endPrefixMapping(String prefix)
throws SAXException
```

```
public void skippedEntity(String name)
throws SAXException
```



# Document Locator

---

- ◆ One item of information that is not passed to the interface methods is **where the parse event occurred**
  - There is no reference to line number or which physical entity the event occurred in
  - For errors, we do have this information available in the `SAXParseException` object (which you saw in the previous lab)
- ◆ A **Locator** is used to access location data about a parse event
  - The line number and column number where the event occurred
  - The public-identifier or system-identifier of the entity in which the event occurred

# Locator Interface

---

```
public int      getColumnNumber()  
  
public int      getLineNumber()  
  
public String   getPublicId()  
  
public String   getSystemId()
```

- ◆ To use the document locator, you store it in an instance variable in your handler when the parser gives it to you

```
private Locator m_locator = null;  
...  
public void setDocumentLocator(Locator locator)  
{  
    m_locator = locator;  
}
```

# SAX Content Events Illustrated

**startDocument()**

**startElement(person)**

**startElement(age)**

**characters(25)**

**endElement(age)**

**startElement(name)**

**characters(Leanne)**

**endElement(name)**

**endElement(person)**

**endDocument()**

```
<?xml version='1.0'?>
<!-- some XML document -->
<person ssn='987-65-4321'>
  <age>
    25
  </age>
  <name>
    Leanne
  </name>
</person>
```

## ContentHandler

- ◆ The method calls above are (obviously) shortened
  - Each method contains the relevant data associated with the parse event in its parameter list
  - e.g., `startElement()` is sent any attributes, plus namespace info

# Document Events

---

- ◆ **startDocument()** is called at the start of a document
  - Called before any other parse event
  - Allows the application to do any preliminary initialization
  - **NOTE** that `setDocumentLocator()` is called **before** `startDocument()` but it is not considered to be a parse event
- ◆ **endDocument()** is called at the end of a document
  - Called last in the sequence of all the parse events
  - Allows the application to do any post-processing cleanup

# Element Events

---

- ◆ **startElement()** is called when a start-tag is read and contains the following parameters:
  - The **URI** of the element's namespace *http://www.javatunes.com*
  - The **local part** of the element name `customer`
  - The **fully-qualified** element name `cust:customer`
  - An **Attributes** object containing the attribute information
- ◆ **endElement()** is called when an end-tag is read and contains the same information (except no **Attributes**)
- ◆ Empty elements generate both events

# Attributes Interface

```
public int      getLength()
```

```
public String getValue(int index)
```

```
public String getValue(String qName)
```

```
public String getType(int index)
```

```
public String getType(String qName)
```

```
public String getURI(int index)
```

```
public String getLocalName(int index)
```

```
public String getQName(int index)
```

```
public int      getIndex(String qName)
```

variations using namespace URI and local name pairs

```
public String getValue(String uri, String localName)
```

```
public String getType(String uri, String localName)
```

```
public int      getIndex(String uri, String localName)
```

# Notes on the `Attributes` Object

---

- ◆ The order of attributes in the list is not guaranteed to match the order in which they appear in the document
- ◆ An optional attribute which does not appear on an element is not in the `Attributes` object
  - That element simply does not have that attribute
  - Attributes with fixed and default values **are** included
- ◆ An attribute that does not have a declaration in the schema is assumed to be of type `CDATA`
  - `CDATA` is the DTD attribute type corresponding to the XML Schema type `xsd:string`
  - **NOTE** that Xerces 2 may simply report all attribute types as `CDATA`

## Lab 6.3: Handling Parse Events

---

In this lab, we will start parsing the contents of the document



# Data Events

---

- ◆ **characters()** reports an element's character content
- ◆ **ignoreWhitespace()** reports whitespace
  - Whitespace between elements
- ◆ Both methods contain the following parameters:
  - A **char[]** buffer containing character data
  - An **int** marking the start of the relevant data in the buffer
  - An **int** indicating the length of the relevant data in the buffer
- ◆ Both function similarly -- what we say about **characters()** is also applicable to **ignoreWhitespace()**
  - This issue is when these are called when whitespace is involved

# Dealing with the `char[]` Buffer

- ◆ For efficiency, a parser may read the **entire document** into the `char[]` buffer and pass that **same buffer** into each call to `characters()` and `ignoreWhitespace()`
  - The **start** and **length** parameters allow you to build a `String` containing just the relevant data -- **it is imperative that you do this**

```
characters()  
  start: 105  
  length: 2
```

```
characters()  
  start: 122  
  length: 6
```

**`char[]`.length=2048**

```
<?xml version='1.0'?>  
<!-- some XML document -->  
  
<person ssn='987-65-4321'>  
  <age>25</age>  
  <name>Leanne</name>  
</person>
```

- ◆ Or, the parser may create a new `char[]` buffer for each call
  - Doesn't matter -- it provides the right **start** and **length**

# Multiple Calls to `characters()`

- ◆ The parser may also split long chunks of character data into multiple calls to `characters()`
  - `characters()` is called (at least twice) if the **parser's internal buffer fills** before it reaches the end of the character data
  - Elements with **mixed content** cause multiple calls to `characters()`

```
startElement(quote)
characters(XML )
startElement(b)
characters(and)
endElement(b)
characters( Java)
endElement(quote)
```

```
<quote>
  XML <b>and</b> Java
</quote>
```

- ◆ This means you may have to store each buffer passed to `characters()` in an instance variable and then assemble them after `endElement()` is called

# characters() vs. ignoreWhitespace()

---

- ◆ **ignoreWhitespace()** reports insignificant whitespace
  - Ignorable, a.k.a. *insignificant* whitespace is added to a document for readability, e.g., carriage returns and tabs/spaces between elements
- ◆ When is whitespace ignorable or not?
  - In other words, when should it be reported by **characters()** and when should it be reported by **ignoreWhitespace()**?
- ◆ The answer depends on two factors:
  1. The element's content model
  2. Whether or not the parser reads the schema (if there even is one)
- ◆ **NOTE - Xerces may report all whitespace as significant with XML Schema -- but treats it intelligently with DTDs**

# When is Whitespace Significant?

---

- ◆ For elements with **element content**, carriage returns and tabs/spaces added for readability are **insignificant whitespace**
  - For **mixed content** elements, **whitespace is significant**
- ◆ Recall that a validating parser **must** read the schema
  - A nonvalidating parser **may** read the schema, but it performs no validation
- ◆ **If there is no schema**, both validating and nonvalidating parsers report all whitespace by calling **characters()**
  - In the absence of a schema, there is no way to determine an element's content model, so the most conservative option is taken
  - All whitespace is considered to be significant

# Whitespace Illustrated - DTD

```
<!-- element with element content -->
<!ELEMENT person (age, name)>
```

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

```
<!-- element with mixed content -->
<!ELEMENT quote (#PCDATA | b)*>
```

```
<quote>␣
␣␣XML␣<b>and</b>␣Java␣
</quote>
```

- ◆ If these DTDs are read by the parser:
  - The **person** element's whitespace (␣ and ␣␣) is reported by **ignorableWhitespace()**
  - The **quote** element's whitespace is reported by **characters()**
- ◆ Is the space between Leanne and Ross significant or not?

# Whitespace Illustrated - XML Schema

```
<!-- element with element content -->
<xsd:element name='person'>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name='age' type='xsd:positiveInteger' />
      <xsd:element name='name' type='xsd:string' />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

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

- ◆ **If** this schema is read by the parser:
  - The **person** element's whitespace (␣ and ␣␣) may be reported by **characters()**
- ◆ Is the space between **Leanne** and **Ross** significant or not?

# Processing Instruction Events

- ◆ **processingInstruction()** reports PIs
  - XML declarations do not trigger this parse event because **they are not PIs**
- ◆ It contains two parameters, one each for the **target** and **instruction** in the PI
  - We must decide what to do with these strings, i.e., determine what these strings “mean” and what action to take
  - For example, invoke an external application with this information

```
processingInstruction()  
target: pager  
instruction: UPS 544-8775-1
```

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



## Lab 6.4: Getting Content Out of Elements

---

In this lab, we will get content from the elements in the document

# Selectively Processing Portions of a Document

---

- ◆ SAX excels at allowing you to quickly grab only the pieces of a document you are interested in
  - This is one of the main uses of SAX
  - You ignore all other parse events as they are passed to your handler
- ◆ To do this, your handler needs to keep track of *state information* -- basically, which element the parser is in
  - The same **characters()** method is called for all content
  - You need to use **startElement()** and **endElement()** to keep track of state, and you need to query the state in **characters()**
- ◆ Depending on the problem at hand, this can get complex rather quickly

# Adding State to a Handler - Strategy

---

- ◆ Use a `boolean` instance variable to keep track of each element you are interested in
  - Turn the `boolean` **on** in **`startElement()`** **if** it's the target element
  - Turn the `boolean` **off** in **`endElement()`** **if** it's the target element
  - Query the `boolean` in **`characters()`**, processing content only **if** the `boolean` is **on**
- ◆ Use a `String` instance variable for each target element
  - You can initialize them in a constructor
- ◆ If the target element has child element content, you may want that data, as well -- the following example assumes we do

# Adding State to a Handler - Example

```
// instance variables for state
private String m_target;
private boolean m_inTarget;

// constructor to initialize target element name
public MyHandler(String target) { m_target = target; }

public void startElement(String nsURI, String localName,
                        String qName, Attributes atts)
    throws SAXException
{
    // key in on target element - set boolean switch on
    if (qName.equals(m_target)) { m_inTarget = true; }

    // if in target element, do desired processing
    if (m_inTarget) {
        /* for example, process attributes */
    }
}
// continued ...
```

# Adding State to a Handler - Example

```
// ... continued
public void endElement(String nsURI, String localName,
                        String qName)
    throws SAXException
{
    // key in on target element - set boolean switch off
    if (qName.equals(m_target)) { m_inTarget = false; }
}

public void characters(char[] data, int start, int length)
    throws SAXException
{
    // if in target element, process content
    if (m_inTarget) {
        String content = new String(data, start, length);
        // do something with the content
    }
}
```

# Terminating the Parse Early

---

- ◆ Recall that an event handler can abort the parse at any time
  - By throwing an exception at the parser
- ◆ We did this in our `ErrorHandler` code
  - Abort parse on a validity error
- ◆ We can also do this from a `ContentHandler` method
  - As soon as you have everything you want out of the document, why continue parsing?
  - Throw a “friendly” exception at the parser to terminate the parse
- ◆ **CAREFUL** - if you are relying on an `endDocument()` call for cleanup/post-processing, you won't get it (see notes)

# Terminating the Parse Early - Example

```
public void endElement(String nsURI, String localName,  
                        String qName)  
throws SAXException  
{  
    // key in on target element - set boolean switch off  
    if (qName.equals(m_target))  
    {  
        m_inTarget = false;  
  
        // we are all done - finish up and terminate parse  
        // we can do post-processing/cleanup in our own method  
        this.finishUp();    // or just call this.endDocument()  
  
        // now we throw an exception at the parser  
        throw new SAXException("all done - successful parse");  
    }  
}
```

## Lab 6.5: State Dependent Processing

---

In this lab, we will get content from the elements in the document



# Other ContentHandler Methods

---

- ◆ **skippedEntity()** reports when the parser skips an entity
  - By “entity” we mean the document’s (external) DTD or XML Schema, general entities, and parameter entities
    - General entities and parameter entities apply only to DTDs
  - Recall that nonvalidating parsers are not required to read anything other than the document entity (the document itself) -- though many will read all referenced entities (including Crimson and Xerces)
- ◆ **start/endPrefixMapping()** report the binding/unbinding of a namespace prefix mapping
  - **These do not need to be used to get regular namespace support** -- the qName passed to start/endElement() includes any namespace prefix (if namespace-awareness is enabled)
  - These are rarely used and mentioned only for completeness

# SAXParser and XMLReader

---

- ◆ Recall that XMLReader is implemented by the parser and that **SAXParser is a wrapper around an underlying XMLReader**
- ◆ Using the JAXP SAXParserFactory and SAXParser instead of using an XMLReader directly provides:
  - An easy mechanism for pluggable parsers, via setting a system property (though you can get this with XMLReader, too)
  - More parse() methods (via overloading), and they allow you to specify a DefaultHandler for all parse event handling
- ◆ The key point is that you are using SAX, but are doing so in a parser-independent way

# Using Individual Handlers with XMLReader

- ◆ **XMLReader** allows us to set individual handlers instead of grouping all our handler code into a single `DefaultHandler`
  - These are sometimes called *registration* methods

```
public void setErrorHandler(ErrorHandler handler)
public void setContentHandler(ContentHandler handler)

// these two handlers are not used as much
public void setEntityResolver(EntityResolver resolver)
public void setDTDHandler(DTDHandler handler)
```

```
// you set a system property for XMLReaderFactory, also
// org.xml.sax.driver=class-name-of-XMLReader-impl
XMLReader reader = XMLReaderFactory.createXMLReader();
ContentHandler orderhandler = new OrderHandler();
ErrorHandler errhandler = new JavaTunesErrorHandler();
reader.setContentHandler(orderhandler);
reader.setErrorHandler(errhandler);
InputSource source = new InputSource(...);
reader.parse(source);
```

# Using Individual Handlers with SAXParser

- ◆ SAXParser allows us to get its contained XMLReader via **getXMLReader()**
  - We can then use the registration methods to set individual handlers

```
// set up SAXParserFactory and SAXParser in the usual way
// turn on namespace-awareness, validation (if desired)
// get the XMLReader contained inside the SAXParser
XMLReader reader = parser.getXMLReader();

// instantiate individual handlers
ContentHandler orderhandler = new OrderHandler();
ErrorHandler errhandler = new JavaTunesErrorHandler();

// register the handlers
reader.setContentHandler(orderhandler);
reader.setErrorHandler(errhandler);
InputSource source = new InputSource(...);
reader.parse(source);
```

# ***SAX Features and Properties***

---

Specifying What You Want in a Parser

# SAX Features

---

- ◆ *SAX features* control the behavior of the parser
  - We have already used two such features -- “validating” and “namespace-aware”
- ◆ **Validation** and **namespace-awareness** are popular features
  - They can be specified in JAXP with SAXParserFactory’s **setValidating()** and **setNamespaceAware()** methods
- ◆ Other features are specified with a more general mechanism
  - Providing **setXXX()** methods for every feature is not practical

# Setting SAX Features

---

- ◆ Features are set and queried with the following **SAXParserFactory** methods
  - These methods are also available in XMLReader

```
// sets/queries the feature in the underlying XMLReader
public void      setFeature(String name, boolean value)
public boolean getFeature(String name)
```

- ◆ SAX feature names are URIs
  - As with namespace URIs, these are just names and you should not expect to find anything “there” if you point your browser at them
- ◆ Generally, features cannot be changed while the parser is performing a parse -- they should be modifiable before or after a parse

# SAX Core Features

---

- ◆ *<http://xml.org/sax/features/validation>*
  - **true** - report validation errors -- implies *external-general-entities* and *external-parameter-entities*
  - **false** - do not report validation errors
  
- ◆ *<http://xml.org/sax/features/external-general-entities>*
  - **true** - include external general entities
  - **false** - do not include external general entities
  
- ◆ *<http://xml.org/sax/features/external-parameter-entities>*
  - **true** - include external parameter entities, including the external DTD subset
  - **false** - do not include external parameter entities, even the external DTD subset



# SAX Core Features

---

- ◆ *<http://xml.org/sax/features/namespaces>*
  - **true** - perform namespace processing, i.e., `prefix:localName`
  - **false** - ignore namespaces, i.e., the colon is not a special character
- ◆ *<http://xml.org/sax/features/namespace-prefixes>*
  - **true** - report attributes used for namespace declarations (`xmlns` and `xmlns:prefix`)
  - **false** - do not report attributes used for namespace declarations
- ◆ *<http://xml.org/sax/features/string-interning>*
  - **true** - element names, prefixes, attribute names, namespace URIs, and local names are interned using the `intern()` method of `String`
  - **false** - names are not necessarily interned
  - Interned `Strings` compare equal when using `==`, which is faster than using `equals()`

# SAX Properties

---

- ◆ *SAX properties* provide for a general extension mechanism
  - Two additional handlers, **LexicalHandler** and **DeclHandler**, are defined in package **org.xml.sax.ext**
  - They are registered with the parser using this mechanism -- the handler objects **are** the properties
- ◆ Properties are set and queried with the following **SAXParser** methods (**not** SAXParserFactory)
  - These methods are also available in XMLReader

```
// sets/queries the property in the underlying XMLReader
// NOTE that the property is an Object, not a boolean
public void    setProperty(String name, Object value)
public Object getProperty(String name)
```

# SAX Core Properties

---

- ◆ *<http://xml.org/sax/properties/lexical-handler>*
  - Used to register a `LexicalHandler` with the parser
  - The `Object` must implement `LexicalHandler`
- ◆ *<http://xml.org/sax/properties/declaration-handler>*
  - Used to register a `DeclHandler` with the parser
  - The `Object` must implement `DeclHandler`
- ◆ *<http://xml.org/sax/properties/xml-string>*
  - This value is a `String` containing the source of a parse event
- ◆ *<http://xml.org/sax/properties/dom-node>*
  - This value is a `DOM Node` representing the parser's current node

# Exceptions for Features and Properties

---

- ◆ **get/setFeature()** and **get/setProperty()** can throw exceptions
- ◆ **SAXNotRecognizedException**
  - Parser doesn't recognize the feature or property
  - **Check the URI and make sure it's correct**, so you know you are not getting this exception due to a programming error
- ◆ **SAXNotSupportedException**
  - Parser recognizes the feature or property, but cannot honor the request
    1. It doesn't provide the functionality
    2. It provides the functionality, but cannot honor the request at this timeThis is likely because the parser is in the middle of a parse

# ***Optional Material***

## ***Other Handlers and SAX Extensions***

---

Useful but Not Often Used

NOTE - this section for reference only - it may be skipped  
in class

# EntityResolver Interface - Used with DTDs

```
public InputSource resolveEntity(String publicId,  
                                   String systemId)  
throws SAXException, IOException
```

- ◆ Allows you to intercept the parser's request for external entities (such as an external DTD)
  - The parser calls this method when it needs an external entity
  - You can return a customized `InputSource` for the entity or return `null` and the parser will use the default entity location mechanisms
  - If there is no `EntityResolver`, the default mechanisms are used
  - The default behavior is for the parser to open a regular URI connection to the system-identifier
- ◆ You would use this to resolve a public-identifier, e.g., do a catalog lookup of a DTD

# DTDHandler Interface - Used with DTDs

```
public void unparsedEntityDecl(String name,  
                                String publicId, String systemId,  
                                String notationName)  
throws SAXException  
  
public void notationDecl(String name,  
                           String publicId, String systemId)  
throws SAXException
```

- ◆ These methods receive *unparsed entity* (<!ENTITY>) and *notation* (<!NOTATION>) declarations
  - An unparsed entity is non-XML data and a notation specifies information about an unparsed entity's format
- ◆ The application is responsible for managing non-XML data
  - These constructs are not often used and we did not cover them

# SAX Extensions

---

- ◆ The XML Recommendation does not require the parser to report certain things back to the application, e.g., comments
- ◆ Similarly, the parser has to process DTD declarations correctly, but is not required to report them
- ◆ The SAX extensions interfaces provide extra functionality beyond the XML Recommendation for the handling of such parse events
  - Most applications do not need to implement these interfaces -- they are mentioned here only for completeness/reference
  - Parsers are not required to support these extensions -- Crimson and Xerces both support them



# LexicalHandler Interface

---

```
public void comment(char[] data, int start, int length)  
throws SAXException
```

```
public void startCDATA() throws SAXException
```

```
public void endCDATA()    throws SAXException
```

```
public void startDTD(String name,  
                    String publicId, String systemId)  
throws SAXException
```

```
public void endDTD() throws SAXException
```

```
public void startEntity(String name) throws SAXException
```

```
public void endEntity(String name)    throws SAXException
```

- ◆ This can be thought of as a second set of callbacks to supplement ContentHandler

# DeclHandler Interface - Used with DTDs

```
public void elementDecl(String name, String model)  
throws SAXException  
  
public void attributeDecl(String eName, String aName,  
                           String type,  
                           String valueDefault, String value)  
throws SAXException  
  
public void internalEntityDecl(String name, String value)  
throws SAXException  
  
public void externalEntityDecl(String name,  
                                String publicId, String systemId)  
throws SAXException
```

- ◆ The parser uses these methods to report on declarations in the DTD that are otherwise unavailable to the application

# Registering Extension Handlers with the Parser

```
class MyLexHandler implements LexicalHandler { /* ... */ }
```

```
class MyDeclHandler implements DeclHandler { /* ... */ }
```

```
// set up factory and parser as usual
```

```
// register extension handlers with parser
```

```
// SAXParser and XMLReader both have setProperty() method
```

```
parser.setProperty(
```

```
    "http://xml.org/sax/properties/lexical-handler",
```

```
    new MyLexHandler());
```

```
parser.setProperty(
```

```
    "http://xml.org/sax/properties/declaration-handler",
```

```
    new MyDeclHandler());
```

# *XML and Java*

## *Section 7 - DOM*

---

```
<topic title='XML and Java'>
  <section num='5' title='JAXP' />
  <section num='6' title='SAX' />
  <section num='7' title='DOM' />
  <section num='8' title='eXtra Topics' />
</topic>
```

- ◆ What is DOM?
- ◆ XML Document as a DOM Tree
- ◆ Getting a DOM Tree from the Parser
- ◆ Manipulating the DOM Tree
- ◆ Transforming a DOM Tree Back to XML

# ***What is DOM?***

---

The Champagne of XML Processing  
(Okay, that's stretching it a bit)

# Defining DOM

---

- ◆ **DOM** (*D*ocument *O*bject *M*odel) is an **API** and **logical model** for representing structured documents
  - A *tree-based* parser builds a **DOM tree** from a structured document
  - **DOM applications** can then manipulate the tree with the DOM API
  - DOM is a W3C Recommendation -- [\*http://www.w3.org/DOM\*](http://www.w3.org/DOM)
- ◆ DOM is organized into *levels*
  - The levels are like layers -- DOM Level 2 extends DOM Level 1
  - Level 1 contains the *Core*, *HTML*, and *XML* features
  - Level 2 contains support for namespaces (among other things)
  - Level 3 contains support for the XML Information Set and a standard way to load and save XML (into and from a DOM doc)
  - We are interested in the *Core* and *XML* features, with support for namespaces

# The DOM API and Language Bindings

---

- ◆ The DOM API is a **set of interfaces** which can be implemented in any programming language
  - A particular programming language implementation of the interfaces is called a *language binding*
- ◆ The DOM API is defined in CORBA IDL
  - This provides for both non-OO and OO language bindings
- ◆ There are thus two “views” of the API
  - *Simplified* or *flattened*, with one interface -- **Node**
  - *Object-oriented* or *inheritance hierarchy*, with several more specific interfaces -- **Element**, **Attr**, **Text**, **Comment**, etc.
  - We will look at both, as they both have merits

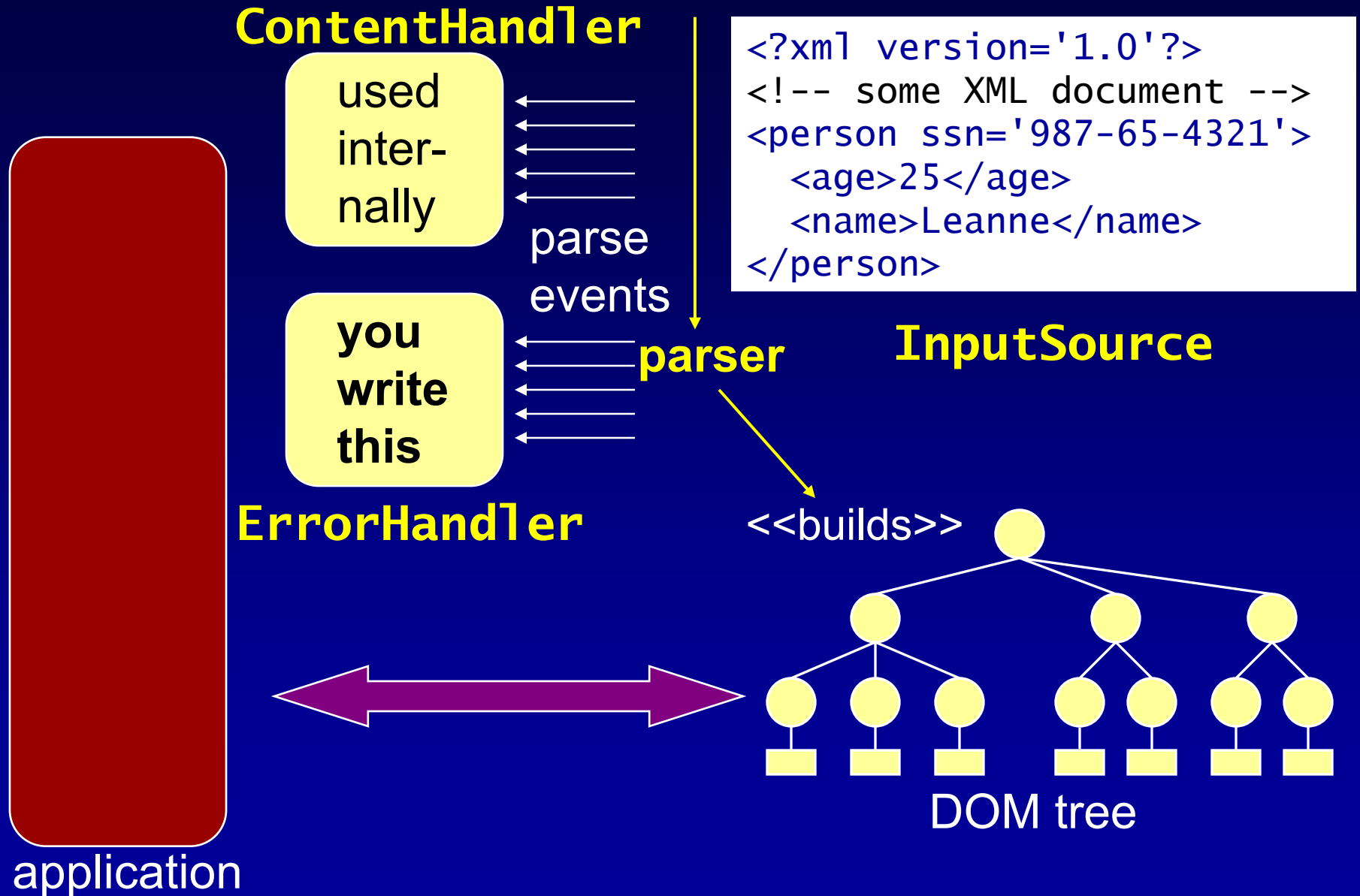
# The DOM Implementation

---

- ◆ A Java XML DOM implementation consists of:
  1. A tree-based parser that returns a DOM tree from a parse
    - This is the JAXP **DocumentBuilder** (implemented in Xerces)
  2. A set of Java methods that implement the DOM interfaces for accessing and manipulating the DOM tree
    - The DOM API is contained in package **org.w3c.dom** and is implemented in Xerces
- ◆ Many DOM implementations use a SAX parser and build the DOM tree in an internal **ContentHandler**
  - The JAXP Xerces parser does this



# DOM Parsing Illustrated - SAX Used Internally



# ***XML Document as a DOM Tree***

---

XML **Does** Grow on Trees

# JavaTunes Order XML Document

---

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

<!-- JavaTunes order XML document

    see notes about the document type declaration -->
<!DOCTYPE order SYSTEM '... '>

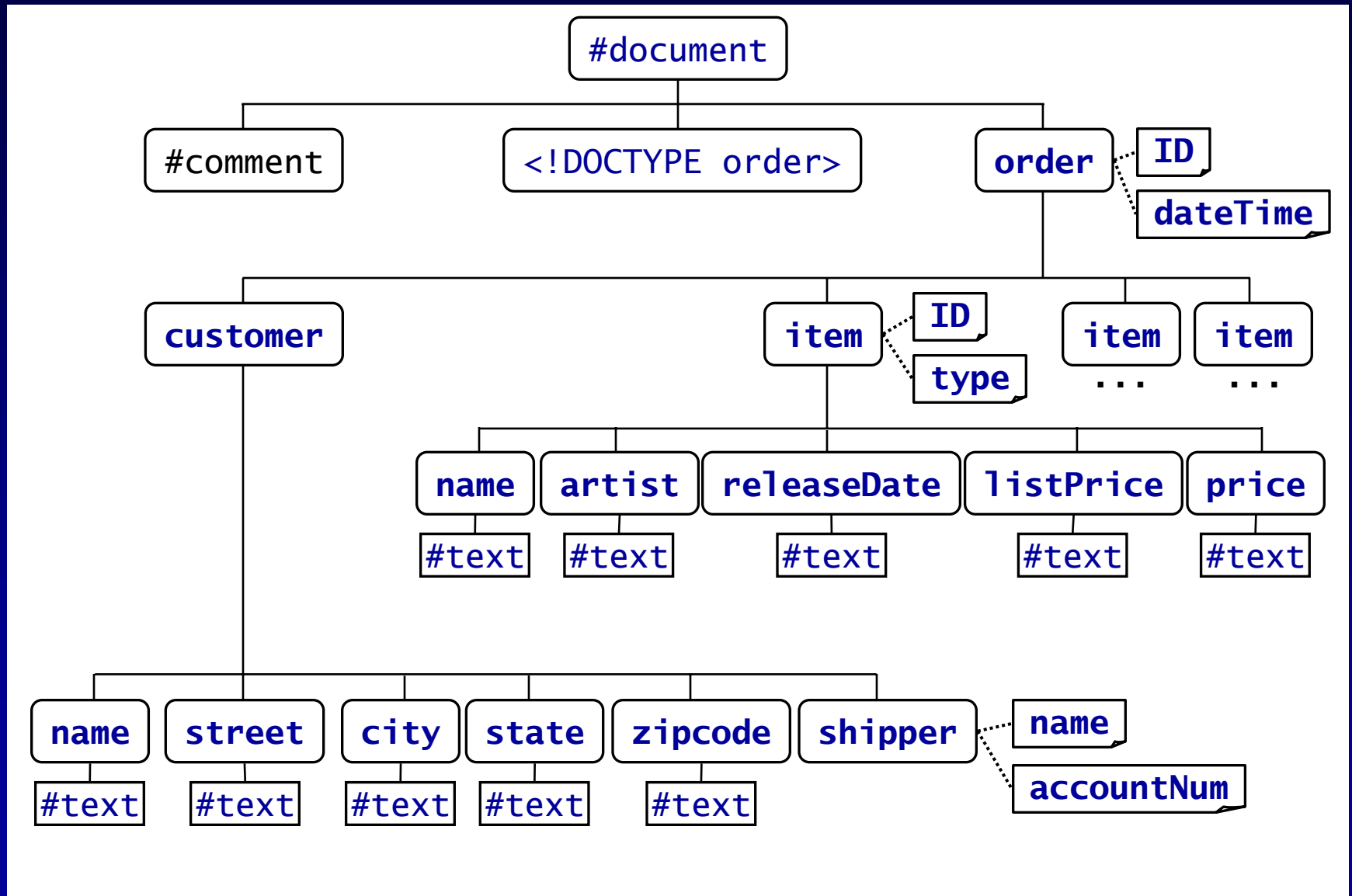
<order ID='_01170302' dateTime='2002-03-20T05:02:00'
  xmlns:xsi='...' xsi:noNamespaceSchemaLocation='...'>
  <customer>
    <name>Susan Phillips</name>
    <street>763 Rodeo Circle</street>
    <city>San Francisco</city>
    <state>CA</state>
    <zipcode>94109</zipcode>
    <shipper name='UPS' accountNum='343-9080-1' />
  </customer>
  ...
```

# JavaTunes Order XML Document

---

```
<?xml version='1.0'?>
<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='CD517'>
  <name>1984</name>
  <artist>Van Halen</artist>
  <releaseDate>1984-08-19</releaseDate>
  <listPrice>11.97</listPrice>
  <price>11.97</price>
</item>
<item ID='CD503'>
  <name>Trouble is...</name>
  <artist>Kenny Wayne Shepherd Band</artist>
  <releaseDate>1997-08-08</releaseDate>
  <listPrice>17.97</listPrice>
  <price>14.99</price>
</item>
</order>
```

# JavaTunes Order as a DOM Tree



# Everything is a Node

---

- ◆ A DOM tree is comprised of *nodes*
  - **Everything** in the XML document is a node in the tree
  - You would expect things like elements and attributes to be in the tree
  - But the tree also includes comments, PIs, the document type declaration (<!DOCTYPE>), etc.
- ◆ Each node has a *type*, a *name*, and a *value*
  - The DOM tree on the previous page shows the nodes and their names
  - Some nodes don't have explicit names, so they are given pseudo-names, e.g., #document, #comment, and #text
- ◆ We will take a closer look at the node types later

# ***Getting a DOM Tree from the Parser***

---

I'll Trade You an XML Document for a DOM Tree

# Ingredients for a Parse - Similar to SAX

---

- ◆ We need a **DocumentBuilder**, an **InputSource**, and an **ErrorHandler**
  - Instantiate a **DocumentBuilder**
  - Create an **InputSource** object representing the XML
  - Write an **ErrorHandler** class and instantiate it
  - Register the **ErrorHandler** with the parser
  - Call **DocumentBuilder**'s **parse()** method, passing in the **InputSource** object

```
public Document parse(InputSource in)  
throws SAXException, IOException
```
- ◆ In calling **parse()**, the parser reads the document and builds the DOM tree
  - **NOTE** that the return type is **Document** this time, not **void**



# The Document Object is the DOM Tree

---

- ◆ It represents the entire XML document, in tree form
  - It is called the **root node** and it provides your entry point into the XML document (via the tree)
  - In the JavaTunes DOM tree shown earlier, it is the node named **#document**
- ◆ **NOTE** that the root node is **not** the root element
  - The root node contains the entire document
  - It is the parent of the root element, and is also the parent of anything contained in the prolog -- comments, PIs, <!DOCTYPE>
  - In our JavaTunes order, the root node has 3 child nodes: a comment, the document type declaration, and the order element
  - This can be a source of confusion, which is why we prefer to call the order element the *document element*, **not** the *root element*

# Specifying Behavior of the Parser

- ◆ As with SAX, we can set a number of properties on the **factory** to specify parser capabilities
  - We already know how to set validation and namespace-awareness
  - We did this in an earlier JAXP lab
- ◆ DOM specifies 4 more properties (see notes for details):
  - **ignoringElementContentWhitespace**
  - **ignoringComments**
  - **expandEntityReferences**
  - **coalescing**

```
<!-- example of how to set/query the properties -->  
public void      setIgnoringComments(boolean ignore);  
public boolean isIgnoringComments();
```

# Setting SAX Handlers for the Parse

---

- ◆ Since SAX is used internally, we can register certain handlers for things that could occur during the parse
  - Such as warnings, validity errors, and well-formedness violations -- these are related to parsing, not DOM
- ◆ We can receive and respond to parse error events by registering an `ErrorHandler`
  - It is **strongly recommended** that you register an `ErrorHandler`
  - We can also register our own `EntityResolver`
    - Used only for DTDs, e.g., to resolve a public-identifier
- ◆ The registration methods are on `DocumentBuilder`

```
public void setErrorHandler(ErrorHandler handler)  
public void setEntityResolver(EntityResolver resolver)
```

# Getting a DOM Tree from a Parser - Example

```
import javax.xml.parsers.*;
import org.xml.sax.*;
import org.w3c.dom.*;
import java.io.FileReader;

class DOMExample {
    public static void main(String[] args) {
        // NOTE - factory and parser code not shown in example

        // declare error handler, XML source, and Document
        ErrorHandler errhandler = null;
        InputSource source = null;
        Document doc = null;
        try {
            // instantiate error handler and XML source
            errhandler = new MyHandler();
            source = new InputSource(new FileReader(args[0]));

            // continued ...
        }
    }
}
```

# Getting a DOM Tree from a Parser - Example

```
// ... continued

// register error handler with parser
parser.setErrorHandler(errhandler);

// get DOM tree (Document object) from XML source
doc = parser.parse(source);

    System.out.println("\nParse successful.");
}
catch (Exception e) {
    System.out.println(e);
    System.out.println("\nParse aborted.");
}
}
}
```

# Handling Setup and Parse Exceptions

---

- ◆ The factory's **newDocumentBuilder()** method can throw a **ParserConfigurationException**
- ◆ The parser's **parse()** method can throw a **SAXException** or an **IOException**
  - As before, parse-related errors are sent to your `ErrorHandler` as SAX error parse events
- ◆ These are all checked exceptions, so you will have to use a `try` block

# Lab 7.1: Getting a DOM Tree From a Parser

---

In this lab, we will configure our DOM parser, and get a DOM tree from the parser

# ***Manipulating the DOM Tree***

---

Walking the Tree and Taking the Fruit

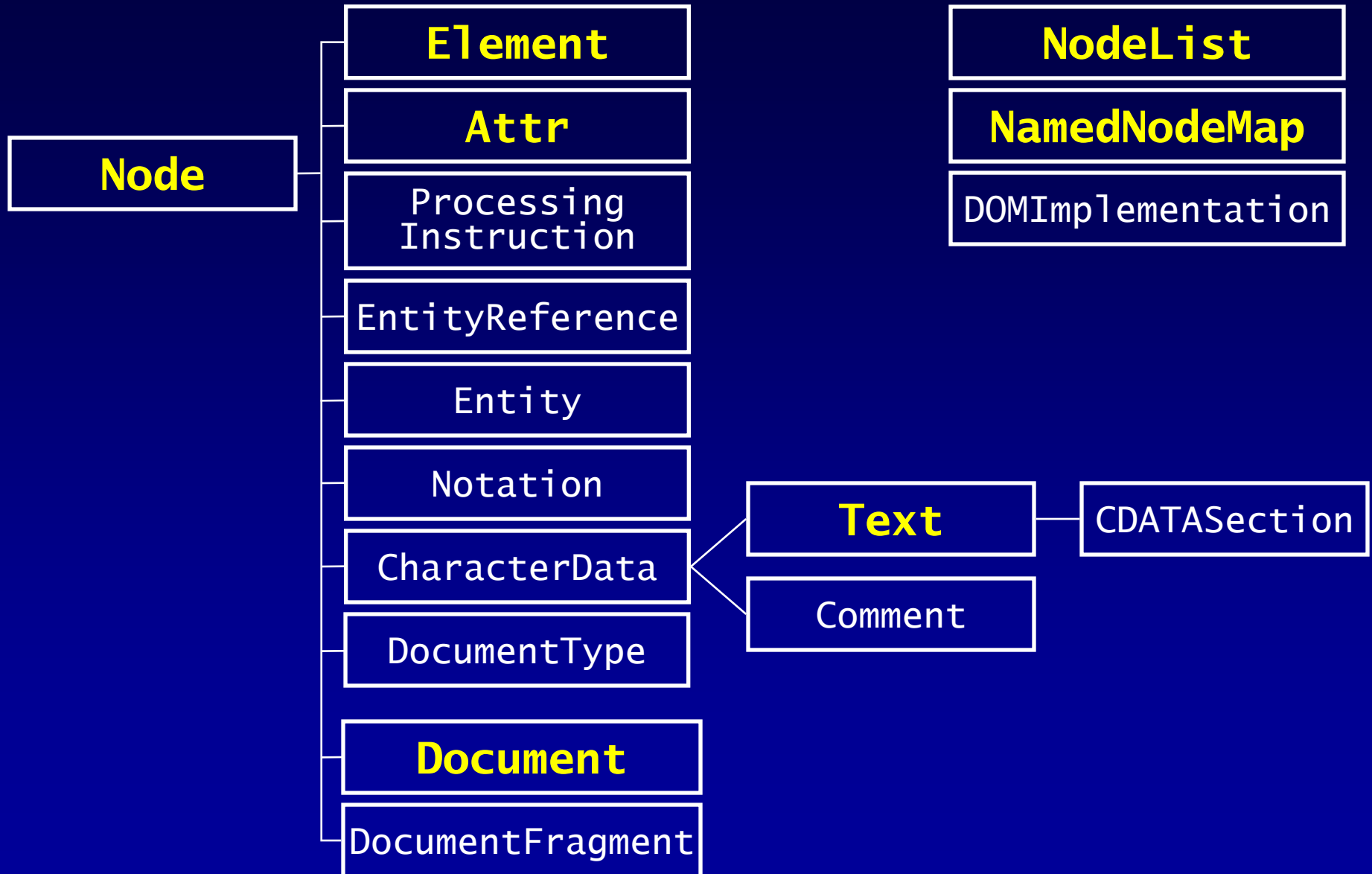


# The DOM Interfaces

---

- ◆ DOM defines an *interface hierarchy*
- ◆ **Node** is the root interface type
  - Some derived interfaces represent the things in an XML document -- **Element**, **Attr**, **Text**, **Comment**, **ProcessingInstruction**, etc.
  - **Document** and **DocumentFragment** represent the DOM tree and a subtree, respectively
- ◆ **NodeList** and **NamedNodeMap** are collections
  - They allow Nodes to be kept in a list or an associative array
- ◆ **DOMImplementation** is the underlying implementation
  - It provides operations that are independent of a particular Document and allows you to query the implementation for specific capabilities

# DOM Interface Hierarchy



# Node Interface

---

- ◆ A DOM tree is a tree of Nodes
  - Really, the Nodes are more specific subtypes, but they can be considered simply as a tree of Nodes, which means that we can walk the tree recursively with very simple code
- ◆ Node is the *simplified* or *flattened* view of the DOM API
  - We can navigate the tree by treating all the nodes as Node objects
- ◆ The tree model that DOM uses is a *familial* model
  - Because there is an **ordering of sibling nodes** -- when siblings are not ordered we call it an *arboreal* model
  - For every collection of sibling nodes, there is a first sibling and a last sibling, and for any two siblings, one comes before the other

# Node Interface - Getting Node Data

- ◆ Every node has 4 properties: **type**, **name**, **value**, **attributes**

```
public short      getNodeTypes()           // see notes
public String     getNodeName()            // see chart*
public String     getNodeValue()           // see chart*
public NamedNodeMap getAttributes()        // see chart*
```

- ◆ Node types are given as integer constants in Node
  - For example, **Node.ELEMENT\_NODE**
- ◆ \*The chart on the next slide details name, value, and attributes
- ◆ **Only element nodes have attributes**

# Node Data - Details by Node Type

Node Type	Name	Value	Attributes
<b>Element</b>	tag name	null	<b>NamedNodeMap</b>
<b>Attr</b>	attribute name	attribute value	null
<b>Text</b>	#text	content of node	null
<b>CDATA Section</b>	#cdata-section	content of section	null
<b>Entity Reference</b>	name of entity referenced	null	null
<b>Entity</b>	entity name	null	null
<b>Processing Instruction</b>	target	instruction	null
<b>Comment</b>	#comment	content of comment	null
<b>Document</b>	#document	null	null
<b>Document Type</b>	document type name	null	null
<b>Document Fragment</b>	#document-fragment	null	null
<b>Notation</b>	notation name	null	null

# Node Data - Getting Attributes from Elements

```
public NamedNodeMap getAttributes()
```

- ◆ **NamedNodeMap** contains iteration methods and methods to read, add, and remove nodes
  - We can iterate through the attribute nodes, using Node methods to get the name and value of each attribute

```
public int getLength()
```

```
public Node item(int index)
```

```
// gets a node by name
```

```
public Node getNamedItem(String name)
```

```
// adds or replaces a node by its node name
```

```
public Node setNamedItem(Node arg)
```

```
// removes a node by name
```

```
public Node removeNamedItem(String name)
```

# Node Interface - Navigating the DOM Tree

```
public Node      getParentNode()
public NodeList getChildNodes()
public Node      getFirstChild()
public Node      getLastChild()
public boolean   hasChildNodes()
public Node      getPreviousSibling()
public Node      getNextSibling()
```

- ◆ **NodeList** simply contains iteration methods

```
public int      getLength()
public Node     item(int index)
```

# DOM Exceptions

---

- ◆ The DOM Recommendation has a single exception type
  - The **DOMException** class in Java is used to encapsulate a **DOM error code** and a descriptive message
  - The error code is provided as a **public instance variable** called **code** in a DOMException
- ◆ The DOM Recommendation specifies methods that return error codes rather than throw exceptions because it has to support language bindings that do not have exceptions
  - In Java, these specified codes are constants in DOMException

```
// constructor - exception contains error code + message  
public DOMException(short code, String message)
```



# Handling DOM Exceptions

- ◆ DOMException extends **RuntimeException**
  - Therefore, you do not have to try/catch the DOM code you write

```
catch (DOMException e)
{
    switch (e.code) // code is a public instance variable
    {
        case DOMException.INDEX_SIZE_ERR:
            ...
        case DOMException.NOT_FOUND_ERR:
            ...
    }
}
```

- ◆ DOM API javadoc is helpful -- methods list the codes that they may include in a thrown DOMException
  - And specifically why the exception is being thrown

## Lab 7.2: Getting Node Data

---

In this lab, we will configure our DOM parser, and get a DOM tree from the parser

# Document Interface

---

- ◆ The Document node represents the *root* of the XML document
  - **Not the root element, but the root**
  - The prolog and body are branches off of the root
- ◆ When the parser is finished parsing, it returns a reference to the root of the DOM tree -- the Document node
  - There can obviously be only one root node in the tree
  - The Document node may have Comment and ProcessingInstruction nodes as children, and exactly one Element node, which corresponds to the root element or document element
  - The Document node may also have only one DocumentType node among its children, corresponding to the <!DOCTYPE> declaration

# Document as Query Tool

- ◆ The Document node provides useful query methods

```
// document element node
public Element      getDocumentElement()

// element node whose ID is elementId
public Element      getElementById(String elementId)

/* all element nodes with a given tagname, in the order
   in which they are encountered in the entire tree */
public NodeList      getElementsByTagName(String tagname)

// document type declaration node (<!DOCTYPE>)
public DocumentType getDoctype()

// specific (vendor) implementation of the DOM API
public DOMImplementation getImplementation()
```

# Document as Node Factory

---

- ◆ The **Document** node also plays the role of a “node factory”
  - The return values are the newly created node objects

<code>public Element</code>	<code><b>createElement</b>(String <i>tagname</i>)</code>
<code>public Attr</code>	<code><b>createAttribute</b>(String <i>name</i>)</code>
<code>public Text</code>	<code><b>createTextNode</b>(String <i>data</i>)</code>
<code>public Comment</code>	<code><b>createComment</b>(String <i>data</i>)</code>
<code>public ProcessingInstruction</code>	<code><b>createProcessingInstruction</b> (String <i>target</i>, String <i>data</i>)</code>

# Element Interface

- ◆ Element extends Node to add specific methods for attributes
  - Attributes can be treated as Strings or as Attr objects

```
// reads attribute
public String      getAttribute(String name)
public Attr        getAttributeNode(String name)

// writes attribute - adds new or replaces existing
public void        setAttribute(String name, String value)
public Attr        setAttributeNode(Attr newAttr)

// removes attribute
public void        removeAttribute(String name)
public Attr        removeAttributeNode(Attr oldAttr)

public boolean     hasAttribute(String name)

/* all element nodes with a given tagname, in the order
   in which they are encountered in the element subtree */
public NodeList    getElementsByTagName(String tagname)
```

# Attr Interface

---

- ◆ `Attr` nodes are different than other nodes since they are technically not on the DOM tree
  - They are not children of their respective `Element` nodes, but rather are “associated with” or “inside” them

```
public String  getName()

public String  getValue()

public void    setValue(String value)

public Element getOwnerElement()

/* was the attribute explicitly given a value
   in the document, or did it take on a default value? */
public boolean getSpecified()
```

# Text Interface

---

- ◆ Text nodes carry the actual content of an XML document
  - The methods listed here are inherited from `CharacterData` (which extends `Node`)

```
public int      getLength()  
  
public String getData()  
  
public void    setData(String data)  
  
public String substringData(int offset, int count)  
  
public void    appendData(String data)  
  
public void    insertData(int offset, String data)  
  
public void    deleteData(int offset, int count)  
  
public void    replaceData(int offset, int count, String s)
```



# Node Surgery - Modifying the DOM Tree

- ◆ The Node interface provides methods that allow nodes to be inserted into and deleted from the tree
  - **NOTE** that you call these methods on the **parent node** of the node to be inserted or deleted

```
// adds newChild after the last child and returns newChild  
public Node appendChild(Node newChild)
```

```
// inserts newChild before refChild and returns newChild  
public Node insertBefore(Node newChild, Node refChild)
```

```
// removes and returns oldChild  
public Node removeChild(Node oldChild)
```

```
// replaces oldChild with newChild and returns oldChild  
public Node replaceChild(Node newChild, Node oldChild)
```

```
// returns a duplicate of this node  
public Node cloneNode(boolean deep)
```

# The DOM Tree is “Live”

---

- ◆ Removal or addition of nodes **immediately changes the tree structure**
- ◆ These changes are **automatically reflected** in `NodeLists` and `NamedNodeMaps`
  - Their size will grow or shrink automatically
  - You don’t have to reinvoke `getChildNodes()` or `getAttributes()` to get updated `NodeLists` and `NamedNodeMaps`, respectively
  - In the case of shrinkage, nodes automatically shift
- ◆ This changes how you have to think about looping constructs
  - See notes below

## Lab 7.3: Modifying the DOM Tree

---

In this lab, we will work with the DOM tree, deleting various elements and then printing out the tree

# ***Transforming a DOM Tree Back to XML***

---

Closing the Circle

# Instantiating an XSLT Transformer - Revisited

```
import javax.xml.transform.*;

class XSLTExample {
    public static void main(String[] args) {
        TransformerFactory factory = null;
        Transformer xformer = null;

        try {
            factory = TransformerFactory.newInstance();

            // get an XSLT transformer from the factory
            // need to specify its XSLT stylesheet at this point
            xformer = factory.newTransformer(/* stylesheet */);
        }
        catch (Exception e) {
            System.out.println(e);
        }
    }
}
```

# Transforming a Source to a Result

- ◆ We can use the *identity* Transformer to convert from one type of data representation to another, e.g., DOM tree to XML
  - No XSLT stylesheet is used
  - We create a **DOMSource** from the DOM tree
  - We create a **StreamResult** to fill an empty output file

```
// get an XSLT transformer from the factory
// NOTE - no stylesheet used for the identity transformer
// newTransformer() is invoked with no stylesheet argument
xformer = factory.newTransformer();

// DOMSource is in package      javax.xml.transform.dom
// StreamResult is in package   javax.xml.transform.stream

// transform source to result
xformer.transform(new DOMSource(doc),
                  new StreamResult(new File("out.xml")));
```

# Extending Your Thinking of a Transformation

---

- ◆ The full signature of the `transform()` method is  
**`public void transform(Source input, Result output)`**  
**`throws TransformerException`**
  - `Source` and `Result` are **interfaces** in package `javax.xml.transform`, representing any type of input source and output result for the `Transformer`
  - `javax.xml.transform.dom`, `javax.xml.transform.sax`, and `javax.xml.transform.stream` are packages containing **classes that implement** these interfaces
- ◆ You can convert from any input form to any other output form
  - DOM tree to SAX events, for example
  - **NOTE** that not all transformer implementations support all types of `Sources` and `Results`

## Lab 7.4: Transforming a DOM Tree to XML

---

In this lab, we will transform the DOM tree back to XML and print it out



# *XML and Java*

## *Section 8 - eXtra Topics*

---

```
<topic title='XML and Java'>
  <section num='5' title='JAXP' />
  <section num='6' title='SAX' />
  <section num='7' title='DOM' />
  <section num='8' title='eXtra Topics' />
</topic>
```

- ◆ StAX Parser Overview
- ◆ Performance Issues
- ◆ JAXB Overview
- ◆ JDOM Overview

# *StAX Parser - Pull Based Parsing*

---

We Deliver the Goods When **YOU** Want Them

# Pull Based API

---

- ◆ With a pull-based parser, the application is in control of when (and if) it needs to interact with an XML document
  - The client gets information when it asks for it
- ◆ There is no standard API for pull-based parsing
  - We'll show some details from the **Java StAX API** which represents the basic model well
- ◆ **XMLStreamReader**
  - One of the main interfaces for pull-based parsing, it provides forward, read-only access to XML
  - It is designed to iterate over XML, and provides methods to get data for the current parser position in the document
  - `XMLStreamReader` is used to represent a cursor with which you can walk a document from beginning to the end
  - It includes accessor methods for all possible information retrievable from the document

# XMLStreamReader Common Methods

---

```
boolean    hasNext()  
           throws XMLStreamException  
int        next()  
           throws XMLStreamException
```

```
int        getEventType()  
QName      getName()  
boolean    hasText()  
String     getText()
```

```
int        getAttributeCount()  
QName      getAttributeName(int index)  
String     getAttributeValue(int index)
```

```
int        getNamespaceCount()  
String     getNamespacePrefix(int index)  
String     getNamespaceURI(int index)
```

# Pull vs Push Based Streaming Parsers

---

- ◆ Pull parsing provides several advantages over push parsing when working with XML streams:
  - Pull parsing libraries can be much smaller and the client code to interact with those libraries much simpler than with push libraries
    - You basically just write a loop, rather than having to implement event handlers
  - The client controls the application thread, and can call methods on the parser when needed
    - With push processing, the parser controls the application thread
  - Pull clients can read multiple documents at one time with a single thread

# Lab 8.1: Trying Out StAX

---

In this lab, we will try out a StAX parser

# *Performance Issues*

---

Nothing is Free

# Parsers

---

- ◆ All Java-based parsers are roughly equal in performance
- ◆ Native parsers, e.g., those written in C, are somewhat faster
  - But don't integrate as well with applications written in Java
- ◆ The JDK implementations are good and have widespread use
  - **Xerces** (A SAX and DOM parser) -- developed initially by Apache and now by both Apache and the Java team
  - **Xalan** is a popular XSLT engine -- developed by Apache and Sun
- ◆ Many application servers include a parser and XSLT engine
  - IBM WebSphere, BEA WebLogic, Oracle9iAS
  - Using these over Sun's may have platform-specific advantages



# Machine Resources

---

- ◆ XML parsers are generally CPU-bound
  - The CPU is the most valuable resource!
- ◆ This is because parsers must perform character-by-character analysis when reading an XML document
  - To look for each **<**, **&**, etc.
- ◆ Conversely, reading “regular” files is I/O bound, not CPU-bound, because data can be read in blocks
- ◆ Best to use XML for **hierarchical data** (where XML shines)
  - If you are processing **list data** or **flat data**, it’s probably better to go with name-value pairs or comma-separated values

# Elements versus Attributes

---

- ◆ From the parser's point of view, an XML document consisting of mostly elements is a much easier data structure to traverse and manipulate
- ◆ Internally, most parsers do this:
  - **Elements** are stored in a **hashtable**
  - **Attributes** are stored in a **list**
- ◆ Hashed lookups are faster than list lookups
  - A document that contains many attributes may perform more slowly than one with fewer attributes
  - Consider using attributes for metadata and elements for “real” content

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

# Using SAX

---

- ◆ SAX is extremely fast and lightweight
  - All it does is scan the document and throw the events at your listener
- ◆ However, SAX does not solve every problem
  - The data are read by the parser, sent to your listener, then discarded -- storing or manipulating the data in memory is up to you
  - The document is read serially -- the parser does not provide random access to the data contained therein
- ◆ SAX is low-level and fairly “raw” -- more work for you
  - Writing the handler code for something as seemingly straightforward as searching a document for a particular element’s value is not trivial
  - The multiple calls to `startElement()` and `characters()` must be handled by keeping track of state information

# Using DOM

---

- ◆ DOM provides a nice benefit in that an XML document is stored in memory, in a tree structure
  - This provides for random access to its contents and easier searching
  - It allows you to modify the document in memory and write it back out to persistent storage, e.g., a file (using XSLT)
- ◆ However, this memory consumption can be prohibitively large -- the DOM tree also takes considerable time to build
  - A 1MB file could require up to several MB of memory storage -- even a 100KB file is considered pretty large
  - The DOM tree is a massive, reference-laden data structure -- each element node has references to:
    - document, preceding-sibling, following-sibling, parent, children, attributes

# Using XSLT

---

- ◆ Depending on what you are doing, transformations can be costly, both in terms of speed and memory consumption
  - There is a tradeoff between flexibility and scalability
- ◆ It's great to have data stored as an XML document and then transform it as needed to:
  - HTML for Web browsers
  - WML (Wireless Markup Language) for handheld devices
  - XML (in some other form)
- ◆ But if you have to transform XML documents for many concurrent users, this can hurt
  - A transformation can take up to 0.5 seconds --  $0.5 * 200$  concurrent users = 100 seconds per transformation (see notes about XSLTC)

# Validation

---

- ◆ It can be expensive to validate an XML document
  - SAX processing may be 2-3 times slower, a noticeable cost
  - DOM processing is slow enough as it is (to build the tree structure in memory), so the validation cost is less noticeable
- ◆ If there is a solid or tight relationship between the parties exchanging XML documents, validation can be turned off
  - The documents exchanged are assumed and trusted to be valid
  - Of course, this is after all testing has been completed and the exchange of documents has been proven to work correctly
- ◆ However, if there is a many-to-many exchange of XML documents, it is probably best to have validation turned on

# ***JAXB Overview***

---

Raising the Bar on XML Processing with Java

# JAXB - Java Architecture for XML Binding

---

- ◆ Provides a fast, convenient way to create a two-way mapping between an XML document and a set of Java objects
  - You can **unmarshal** an XML document directly into a set of these objects, which is called a **content tree**
  - You can **marshal** these objects directly into an XML document
- ◆ Provides a **schema compiler** that takes a schema as input and generates the mapped Java classes as output
  - **NOTE** that the JAXB specification only supports XML Schema
  - There is also a set of annotation elements that can be used within an XML Schema to customize the generated classes
  - The generated classes can validate input XML documents as well as any XML streams that are output by the content tree



# Why Use JAXB?

---

- ◆ JAXB provides a bridge between XML and Java
  - Just as an XML document is an instance of a schema, a Java object is an instance of a class
  - JAXB allows you to create Java objects at the same conceptual level as the XML data
- ◆ JAXB is easy to use
  - No low-level SAX parse event handler code, no tree-manipulation code with the DOM API
  - The processing code required to bridge XML data with corresponding Java objects is written for you

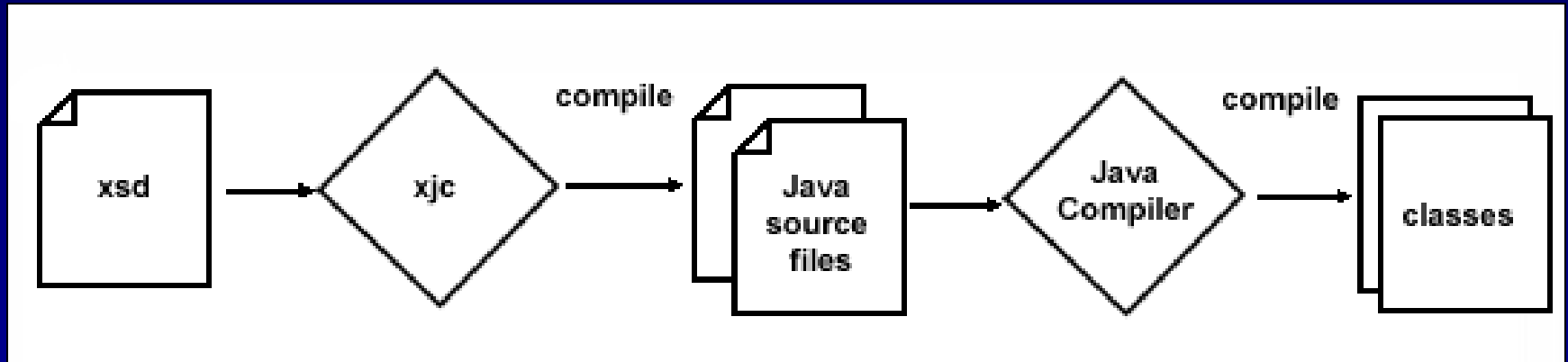
# Benefits of JAXB Over Straight SAX and DOM

---

- ◆ JAXB combines some of the best features of SAX and DOM
  - A SAX parser is fast, but does not store the document in memory -- a DOM parser does, but at considerable cost
- ◆ JAXB stores the document data in a set of Java objects
  - Since the data are now in objects, you can simply call the accessor methods on those objects to read and modify their data
  - And then you can easily marshal those objects back into XML form
- ◆ JAXB can even be faster than SAX
  - JAXB is fast because the generated classes contain the schema logic, avoiding the dynamic interpretation a SAX parser must perform

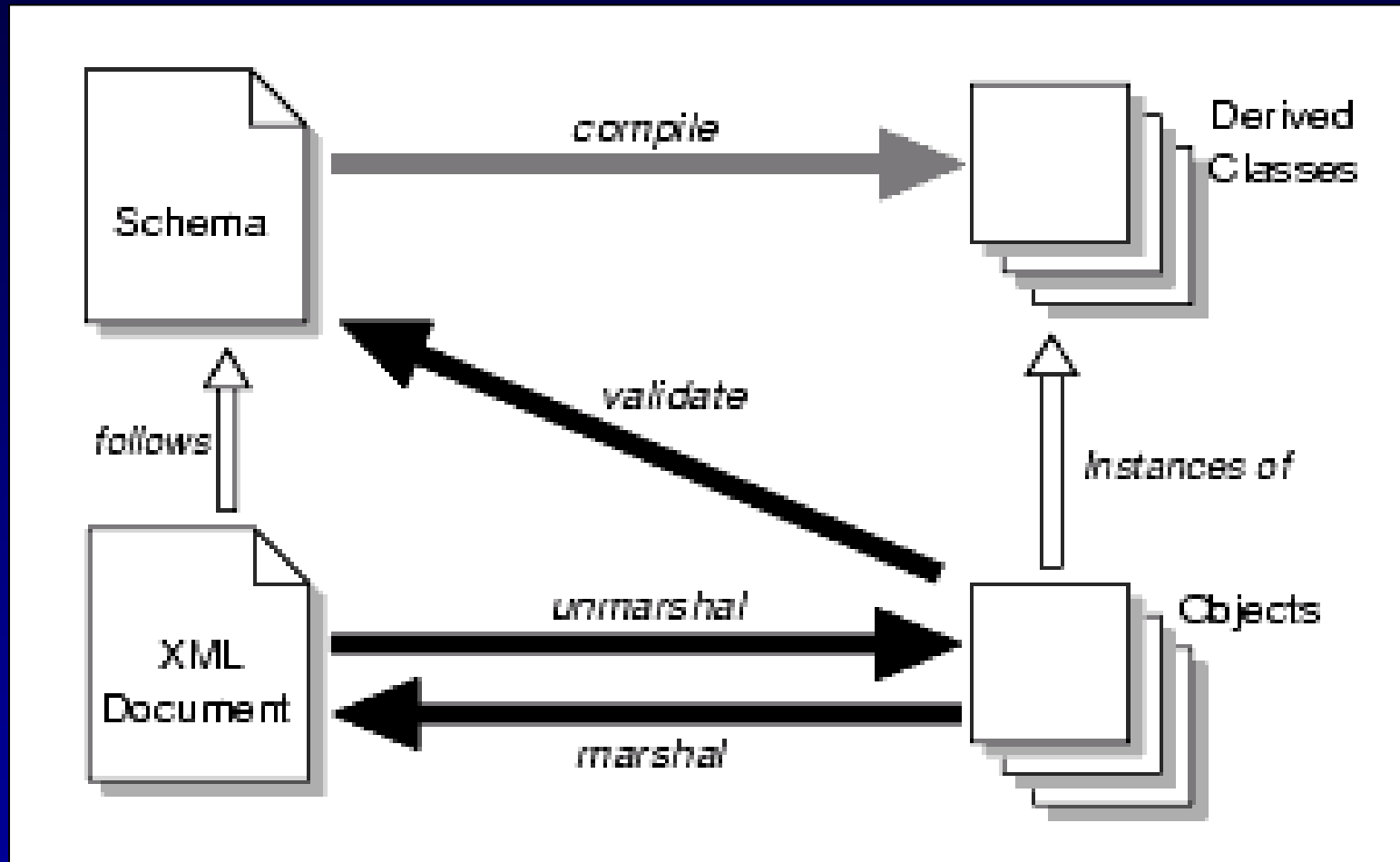
# How it Works - Generating the Classes

- ◆ You need an XML Schema to use JAXB
  - The schema is used to generate the classes
  - See notes below about support for other schema languages
  - The *schema compiler xjc* reads the XML Schema document and generates the source code (which you then compile the normal way)



- ◆ By default, the schema compiler generates a class for every element that contains child elements
  - Inside a class, it generates a property (a pair of get/set methods) to access the content of child elements and the values of attributes

# How it Works – XML > Java > XML - Illustrated



# How it Works – XML > Java > XML - Coding Steps

---

## ◆ XML > Java

- Create an instance of **JAXBContext**
- Create an **Unmarshaller** from a JAXBContext
- Call the Unmarshaller's **unmarshal()** method

## ◆ Java > XML

- Create an instance of **JAXBContext**
- Create a **Marshaller** from a JAXBContext
- Call the Marshaller's **marshal()** method
- You can also start with an “empty” content tree
- You instantiate the generated classes and populate the objects with data -- you can then marshal it into an XML document

# JAXB Can Validate Data

---

- ◆ **Unmarshal-time** validation

- The `Unmarshaller` can validate the XML document as it's creating the content tree

- ◆ **On-demand** validation

- At any time, you can ask a **Validator** to validate the content tree or any subtree of it

- ◆ **Fail-fast** validation

- Operations that make the content tree invalid immediately fail

- ◆ These validations are provided by the JAXB implementation

- They do not necessarily involve an XML Schema
- The generated content tree classes have validation logic in them

# Example - XML Schema

```
<xsd:schema xmlns:xsd='http://www.w3.org/2001/XMLSchema'>
  <xsd:element name='order' type='orderType' />

  <xsd:complexType name='orderType'>
    <xsd:sequence>
      <xsd:element name='customer' type='xsd:string' />
      <xsd:element name='item' type='itemType'
        maxOccurs='unbounded' />
    </xsd:sequence>
    <xsd:attribute name='ID' type='xsd:ID' use='required' />
  </xsd:complexType>

  <xsd:complexType name='itemType'>
    <xsd:sequence>
      <xsd:element name='name' type='xsd:string' />
      <xsd:element name='price' type='xsd:decimal' />
    </xsd:sequence>
    <xsd:attribute name='ID' type='xsd:ID' use='required' />
    <xsd:attribute name='type' type='xsd:NMTOKEN'
      default='CD' />
  </xsd:complexType>
</xsd:schema>
```

# Example - XML Document

---

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

<order ID='_01170302'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='order-jaxb.xsd'>
  <customer>Susan Phillips</customer>
  <item ID='CD514'>
    <name>So</name>
    <price>13.99</price>
  </item>
  <item ID='CD517'>
    <name>1984</name>
    <price>11.97</price>
  </item>
  <item ID='CD503'>
    <name>Trouble is...</name>
    <price>14.99</price>
  </item>
</order>
```



# Example - Generating the Classes

---

- ◆ Run **xjc**, specifying the schema and the package for the generated classes

```
xjc -p com.javatunes.order order-jaxb.xsd
```

- ◆ Compile the generated code

```
javac com\javatunes\order\*.java
```

- ◆ Javadoc the generated code

```
javadoc -d ... com.javatunes.order
```

- ◆ See the notes below for instructions on how to run the examples

# Example - The Generated Classes - ItemType

- ◆ **ItemType** contains the data for each **item** element
- ◆ **item**'s **ID** and **type** attributes and **name** and **price** child elements have simple content
  - They become scalar properties of ItemType
  - String get/set**ID**()
  - String get/set**Type**()
  - String get/set**Name**()
  - BigDecimal get/set**Price**()

```
<item ID='CD514'>           <!-- type='CD' by default -->
  <name>So</name>
  <price>13.99</price>
</item>
```

# Example - The Generated Classes - OrderType

- ◆ **OrderType** contains the data for each **order** element
- ◆ order's **ID** attribute and **customer** child element have simple content
  - They become scalar properties of OrderType
  - String get/set**ID**()
  - String get/set**Customer**()
- ◆ **order**'s **item** child elements have complex content
  - They are stored in a List property, as ItemTypes
  - List get**Item**()

```
<order ID='_01170302'>  
  <customer>Susan Phillips</customer>  
  <item>  
  <item>
```

# Example - Additional Generated Classes

---

- ◆ We have already seen that a class is generated for each **complex type** in the schema
  - In our example, we have an **OrderType** and an **ItemType**
- ◆ A class is also generated for each **top-level element** in the schema
  - In our example, this means an **Order**
  - Order extends OrderType
- ◆ An **ObjectFactory** class is also generated
  - It helps us to create new content tree objects
  - Order **createOrder()**
  - ItemType **createItemType()**

# Example - The Content Tree - Illustrated

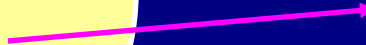
- ◆ **OrderType** is the **root of the content tree**, since it is the root element of the XML document

## OrderType

String ID  
String customer  
List item

## ItemTypes

String ID  
String type  
String name  
BigDecimal price



# Example - Unmarshalling XML into Content Tree

```
import javax.xml.bind.*;
import com.javatunes.order.*;
// with other imports

class OrderTest {
    public static void main(String[] args) {
        try {
            // create JAXBContext object
            JAXBContext jaxbctx =
                JAXBContext.newInstance("com.javatunes.order");

            // create Unmarshaller
            Unmarshaller u = jaxbctx.createUnmarshaller();

            // turn on unmarshal-time validation
            u.setValidating(true);

            // unmarshal XML document into content tree - easy!
            Order o = (Order) u.unmarshal(
                new FileInputStream("order-jaxb.xml"));
        }
    }
}
```

# Example - Modifying the Content Tree

```
// show some order data
System.out.println("Order ID: " + o.getID());
System.out.println("Customer: " + o.getCustomer());

// change the customer information
o.setCustomer("Leanne Ross");

// change the item information
List items = o.getItem();          // the List is "live"

for (Iterator i = items.iterator(); i.hasNext(); ) {
    ItemType item = (ItemType) i.next();

    if (item.getName().equals("So")) {
        item.setPrice(new BigDecimal("8.88"));
    }

    if (item.getName().equals("1984")) {
        items.remove(item);
    }
}
```

# Example - Marshalling Content Tree into XML

```
// perform on-demand validation of content tree
Validator v = jaxbctx.createValidator();
v.validateRoot(o);

// create a Marshaller
Marshaller m = jaxbctx.createMarshaller();

// set schema location attribute
m.setProperty(
    Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION,
    "order-jaxb.xsd");

// marshal content tree into XML document - easy!
m.marshal(o,
    new FileOutputStream("order-jaxb-result.xml"));
}
catch (Exception e) {
    System.out.println(e);
}
}
```



# Example - Creating a New Content Tree

```
import javax.xml.bind.*;
import com.javatunes.order.*;
// with other imports

class OrderCreate {
    public static void main(String[] args) {
        try {
            // create a JAXBContext object
            JAXBContext jaxbctx =
                JAXBContext.newInstance("com.javatunes.order");

            // create factory to create order and its data
            ObjectFactory orderFactory = new ObjectFactory();

            // create a new Order object and fill it with data
            Order o = orderFactory.createOrder();
            o.setID("_02201003");
            o.setCustomer("Connor Morgan");
        }
    }
}
```

# Example - Creating a New Content Tree

---

```
List items = o.getItem(); // empty List

ItemType item1 = orderFactory.createItemType();
item1.setID("CD510");
item1.setName("Hysteria");
item1.setPrice(new BigDecimal("14.99"));
items.add(item1); // type attribute defaults to CD

ItemType item2 = orderFactory.createItemType();
item2.setID("CD518");
item2.setName("Escape");
item2.setPrice(new BigDecimal("11.97"));
items.add(item2); // type attribute defaults to CD
```

# Example - Marshalling Content Tree into XML

```
// perform on-demand validation of content tree
Validator v = jaxbctx.createValidator();
v.validateRoot(o);

// create a Marshaller
Marshaller m = jaxbctx.createMarshaller();

// set schema location attribute
m.setProperty(
    Marshaller.JAXB_NO_NAMESPACE_SCHEMA_LOCATION,
    "order-jaxb.xsd");

// marshal content tree into a new XML document
m.marshal(o,
    new FileOutputStream("order-jaxb-create.xml"));
}
catch (Exception e) {
    System.out.println(e);
}
}
```

# Example - Invalidating the Content Tree

- ◆ Any of the following would invalidate the content tree

```
// Order o
o.setID("123");           // invalid ID (must be XML name)
o.setCustomer(null);      // required element

// ItemType item
item.setID(null);         // required attribute
item.setType("A DVD");    // invalid type (must be NMTOKEN)
item.setName(null);       // required element
item.setPrice(new BigDecimal("abc")); // not a number
```

- ◆ This would be detected when:
  - Performing on-demand validation with the Validator
  - Marshalling content tree to XML with the Marshaller

# JAXP (SAX, DOM, XSLT) vs. JAXB

---

- ◆ If you want to read only a piece of data from an XML document, use SAX because it is fast and works well here
- ◆ If you want to modify an XML document (that is not too large) by inserting or removing objects from the tree, use DOM because of its tree manipulation capabilities
- ◆ If you want to transform XML to another format, use XSLT
  - XSLT is also used to transform a DOM tree to an XML document

# JAXP (SAX, DOM, XSLT) vs. JAXB

---

- ◆ JAXB provides easy access to data in memory, but does not provide the full tree manipulation capabilities of DOM
  - You can't just add nodes into the content tree anywhere you want -- you are limited to the capabilities of the generated classes' methods
  - DOM allows you to add/edit/delete nodes in any fashion desired
  - This is why the JAXB content tree has a much smaller memory footprint than a DOM tree
- ◆ JAXB provides for converting XML data to different types
- ◆ JAXB requires the use of XML Schema and valid data
  - Invalid XML data cannot generate a proper content tree
  - This also allows JAXB to be fast, because the generated classes only work with a known document type, thereby adding efficiency

# A Few Last Points

---

- ◆ JAXB provides a very important component for working with XML data from Java
  - **JAXB raises the level of abstraction** -- instead of writing SAX event handler code or DOM tree manipulation code, you can focus on **working with the data**
  - Having nothing but parsers to work with slows down the application development cycle
  - The JAXB generated classes can be extended to provide additional application-specific functionality
- ◆ If you need a high-level in-memory representation of XML data, think JAXB
  - SAX and DOM are both fairly low-level

# ***JDOM Overview***

---

## The Java DOM



# What is JDOM?

---

- ◆ JDOM allows for in-memory access to an XML document, similar to DOM -- however, JDOM:
  - Has a more straightforward API
  - Performs better than DOM
  - Is optimized for Java
- ◆ JDOM started as an open source project, and is now part of Sun's Java Community Process
  - Java Specification Request JSR-102
  - This means that JDOM will become part of the Java platform
- ◆ Jason Hunter is the main person behind JDOM's development

# JDOM Philosophy

---

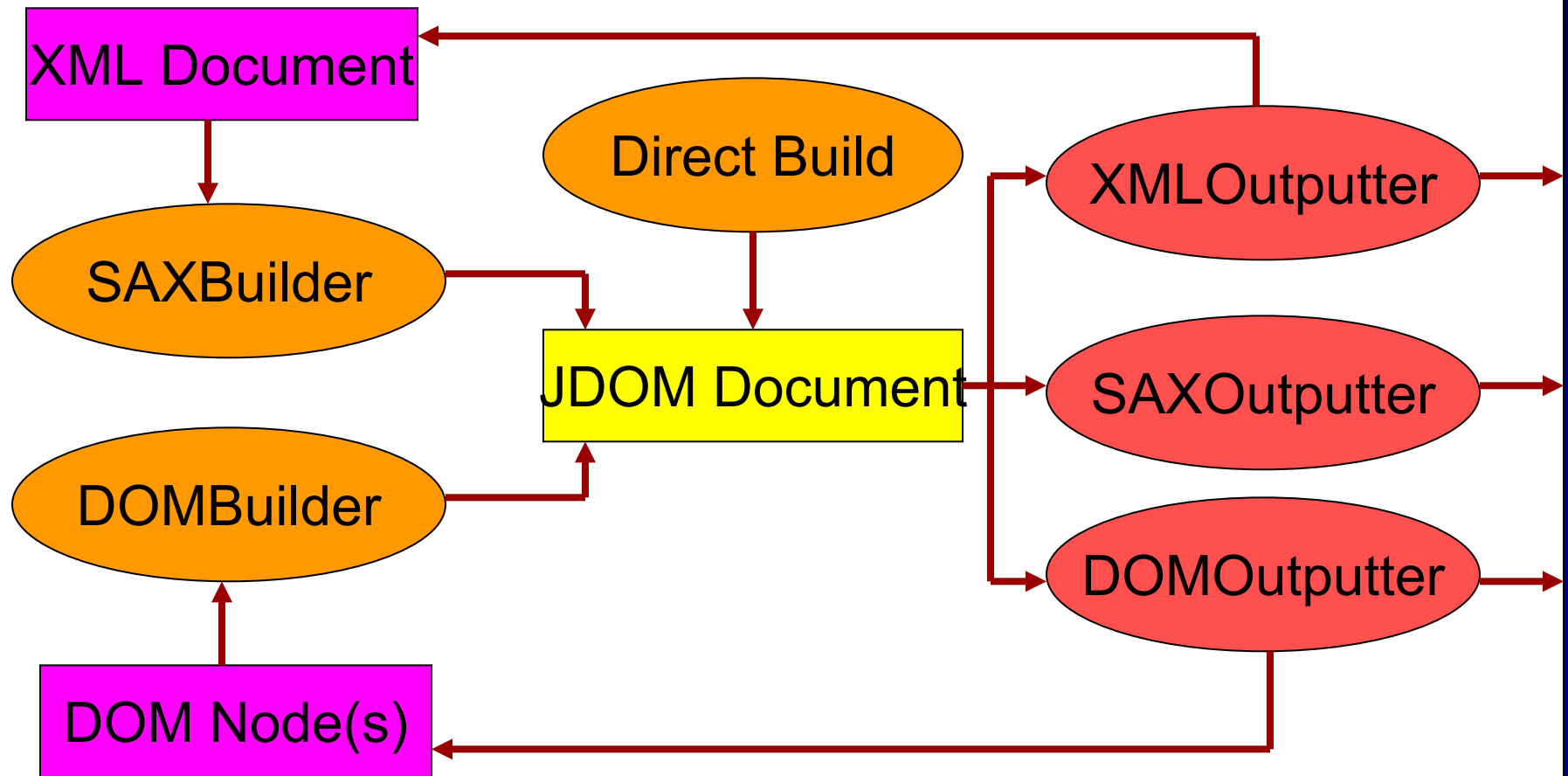
- ◆ Should be straightforward for Java programmers
  - Easier API -- includes method overloading and uses the standard Collections classes
  - Provides for data type conversions
  - Provides Java exception handling -- not DOM error codes
- ◆ Should hide the complexities of XML and stay current with the latest XML standards
  - SAX 2.0, DOM Level 2, XML Namespaces, W3C XML Schema
- ◆ Should integrate well with SAX and DOM
  - Conversion from SAX/DOM to JDOM and vice-versa

# JDOM 2.x Packages

---

- ◆ **org.jdom2** contains the types that represent an XML document and its constructs
  - **Document**, **Element**, **Attribute**, **DocType**, **Comment**, etc.
- ◆ **org.jdom2.input**: Classes for reading XML documents
  - **DOMBuilder**, **SAXBuilder**
- ◆ **org.jdom2.output** contains classes for writing XML documents
  - **DOMOutputter**, **SAXOutputter**, **XMLOutputter**
- ◆ **org.jdom2.transform** supports XSLT transformations

# Typical Scenario



# Building a JDOM Document

- ◆ Documents can be built from scratch -- “direct build”

```
// order is the document element  
Document doc = new Document(new Element("order"));
```

- ◆ Documents can be built from an XML input source

```
SAXBuilder builder = new SAXBuilder();  
Document doc = builder.build(new File("order.xml"));
```

- SAXBuilder is fast and recommended -- uses a SAX parser
- DOMBuilder is used to read an existing DOM tree

- ◆ You can specify the parser used by the builder

```
public SAXBuilder(String parserClass, boolean validate);  
public DOMBuilder(String parserClass, boolean validate);
```

# Outputting the JDOM Document

---

- ◆ **XMLOutputter** writes the document as XML
- ◆ **SAXOutputter** generates SAX events
- ◆ **DOMOutputter** creates a DOM tree
- ◆ To output a document to XML:

```
XMLOutputter outputter = new XMLOutputter();  
outputter.output(  
    doc, new FileOutputStream("order-new.xml"));
```

# The XML Document Used in Our Examples

---

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

<order ID='_01170302'
  xmlns:xsi='http://www.w3.org/2001/XMLSchema-instance'
  xsi:noNamespaceSchemaLocation='order-jaxb.xsd'>
  <customer>Susan Phillips</customer>
  <item ID='CD514'>
    <name>So</name>
    <price>13.99</price>
  </item>
  <item ID='CD517'>
    <name>1984</name>
    <price>11.97</price>
  </item>
  <item ID='CD503'>
    <name>Trouble is...</name>
    <price>14.99</price>
  </item>
</order>
```

# Working with Elements

---

- ◆ Elements are easy to work with in JDOM
  - There is no all-encompassing Node type, as in DOM, where everything in an XML document is a “node”
  - The JDOM API was designed to easily handle the most common cases, e.g., getting the content out of an element
  - JDOM philosophy -- “an element has content”
  - DOM philosophy -- “an element node has a child text node with content”
- ◆ Elements can be retrieved from a document or created from scratch

```
Element order = doc.getRootElement();
```

```
Element customer = new Element("customer");
```



# Retrieving Child Elements

---

```
// returns all of order's child elements
List<Element> allChildren = order.getChildren();

// returns all of order's item child elements
List<Element> items = order.getChildren("item");

// returns order's first item child element
Element item1 = order.getChild("item");

// descendant elements can also be retrieved easily
Element price = order.getChild("item").getChild("price");
```

# Adding and Removing Child Elements

- ◆ You can use the standard `List` methods or convenience methods provided in the JDOM API

```
List<Element> allChildren = order.getChildren();
```

```
/* remove order's customer child element by  
   using the remove() method of List */
```

```
allChildren.remove(order.getChild("customer"));
```

```
// remove customer by using convenience method of Element  
order.removeChild("customer");
```

```
/* add item child element to order by  
   using the add() method of List */
```

```
allChildren.add(new Element("item"));
```

```
// add item by using convenience method of Element  
order.addContent(new Element("item"));
```

# Reading and Writing Element Content

---

```
// get the character content out of an element  
// getTextTrim() and getTextNormalize() methods, also  
Element customer = order.getChild("customer");  
String cust = customer.getText();
```

```
// create or change an element's character content  
Element customer = order.getChild("customer");  
customer.setText("Leanne Ross");
```

# Reading Attributes

- ◆ You can treat an attribute as a (`String`) name-value pair or use the **Attribute** class, which provides some data type conversion

```
<customer number='2002'>Leanne Ross</customer>
```

```
// read attribute as a String
String numberValue = customer.getAttributeValue("number");

// read attribute as an int - use Attribute class
Attribute numberAttr = customer.getAttribute("number");
int numberValueInt = numberAttr.getIntValue();
```

# Writing Attributes

---

```
// set the value of an attribute
customer.setAttributeValue("number", "125");

// add attribute and its value
customer.addAttribute("status", "preferred");

// add attribute using Attribute class
customer.addAttribute(
    new Attribute("status", "preferred"));

// remove an attribute
customer.removeAttribute("number");

// remove all the attributes
customer.getAttributes().clear();    // a List
```

# Working with JDOM is Easy and Powerful

---

- ◆ The JDOM API is closer to what most Java programmers expect when they start exploring how to work with an in-memory representation of an XML document
  - The API uses familiar Collections classes and also provides convenience methods for modifying content
  - The JDOM API is all about Java
  - Basic data type conversions are provided
- ◆ JDOM integrates with SAX, DOM, and XSLT
  - And it will aggressively support emerging XML standards

# Current Status

---

- ◆ JDOM is currently in in version 2.x
  - Software and documentation can be downloaded from:  
*<http://www.jdom.org>*
  - It supports Java 5+
  
- ◆ Some possible extensions to JDOM
  - XPath -- much work on this has been done already
  - XLink/XPointer -- follows XPath
  - Native XSLT support -- currently uses Xalan
  - In-memory validation

# *Introduction to XSLT*

## *Section 9 - XPath*

---

```
<topic title='XPath and XSLT'>  
  <section num='9' title='XPath' />  
  <section num='10' title='XSLT' />  
</topic>
```

- ◆ XPath Data Model
- ◆ XPath Expressions and Location Paths
- ◆ Traversing XML Documents
- ◆ Predicates in XPath Expressions
- ◆ Other Uses of XPath



# Section Overview and Objectives

---

- ◆ This section is an introduction to the XPath language and the associated XPath data model
- ◆ It serves as a prerequisite to the XSLT section
  - XPath is not used in isolation, but rather with other XML technologies such as XSLT and Xpointer
- ◆ The answers to the lab exercise for this section are in the back of the printed course manual

## Objectives

- ◆ Understand the XPath data model and its representation of an XML document as a tree structure
- ◆ Write XPath expressions to locate specific portions of the XPath tree structure
- ◆ Understand the XPath axes that partition the tree structure

# ***XPath Data Model***

---

## The Biology of the XML Tree

# The Concept of a Path

---

- ◆ A *path* is a description of how to find something in a collection of objects
- ◆ Paths only make sense with respect to an underlying model of the collection's structure
  - *C:\doc\letters\bob.doc* is a path that describes how to locate a file object in a Windows hierarchical file system
  - *http://www.w3.org/TR/xpath* describes how to find the XPath Recommendation document on the World Wide Web
- ◆ In the above examples, we cannot make sense of the paths unless we understand the underlying model

# An XPath

---

- ◆ An XPath is a description of how to find “something” in an XML document
- ◆ One important difference between an XPath and the other kinds of paths we are used to is that **an XPath usually identifies a collection of objects** rather than just one
- ◆ This collection of objects described by an XPath is called the **result set** of the XPath
  - We now have to determine what these objects described by the XPath actually are

# XPath Data Model

---

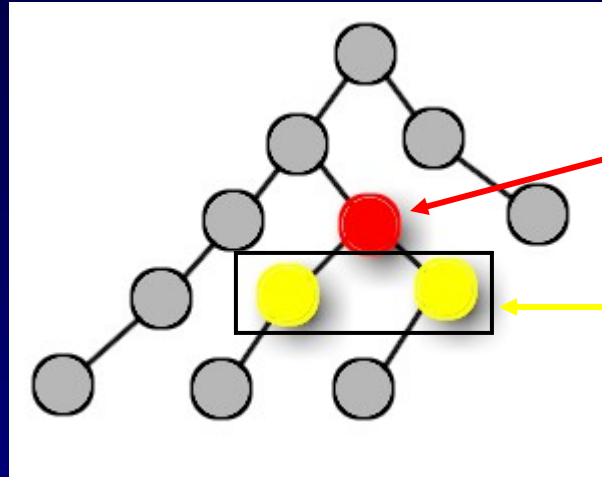
- ◆ XPath operates on an XML document as a **tree of nodes**
  - The XPath Recommendation defines the kinds of nodes and their relationships in what is called the ***XPath data model***
- ◆ The XPath data model defines seven types of nodes:
  - **Root**
  - **Element**
  - **Text**
  - **Attribute**
  - **Comment**
  - **Processing Instruction (PI)**
  - **Namespace**

# XPath Tree

---

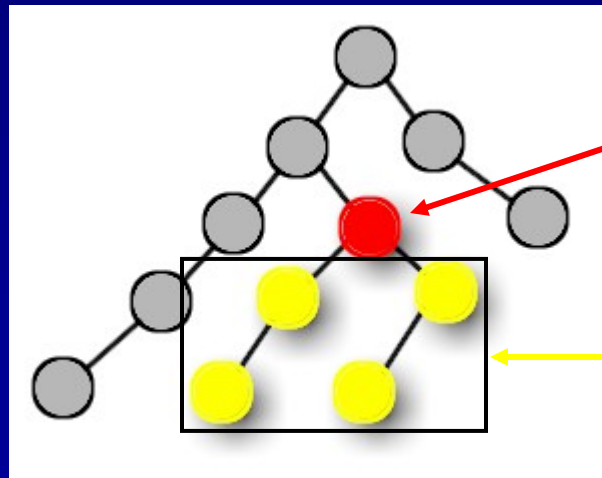
- ◆ There are two kinds of tree models -- the *arboreal* model and the *familial* model
- ◆ In the arboreal model, we speak of roots, branches, and leaves
  - A standard file system uses the arboreal model
  - The ordering of nodes at the same level is not significant
- ◆ In the familial model, we speak of children, parents, and siblings
  - The ordering of siblings is significant
- ◆ The XPath data model uses the **familial** model

# Basic Node Relationships



## ◆ *Children*

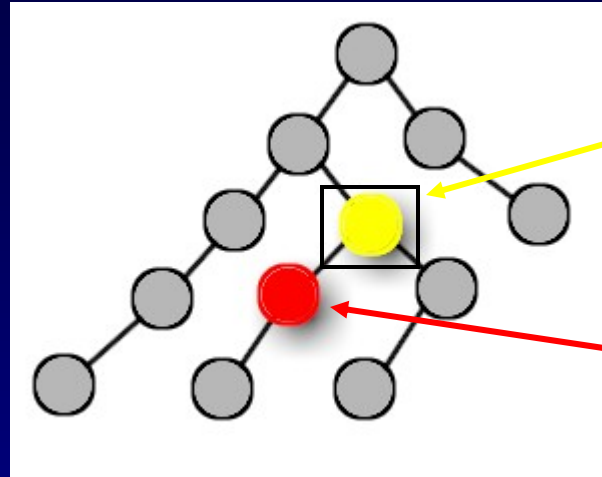
- Nodes directly below a reference node



## ◆ *Descendants*

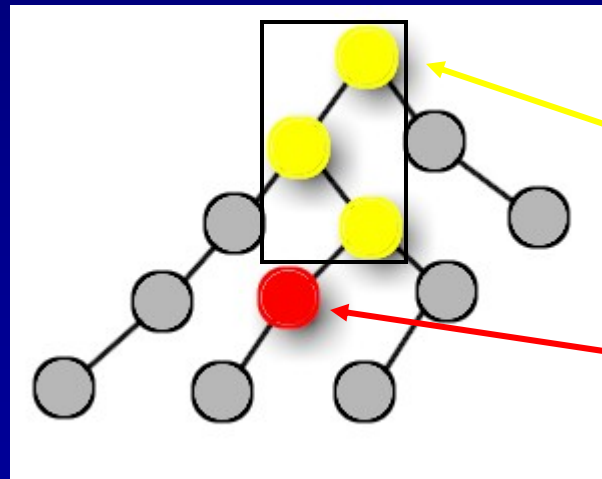
- All nodes to any depth below the reference node

# Basic Node Relationships



## ◆ *Parent*

- Node directly above a reference node



## ◆ *Ancestors*

- All nodes to any depth above the reference node



# More about Nodes

---

- ◆ Nodes have a *string-value* and some node types have *names*
- ◆ **String-values** can be computed for all node types
  - It may incorporate the string-values of its children
  - The string-value is a way of “extracting content” from a node
- ◆ **Names** can be computed for only some node types
  - Names are not meaningful for all kinds of nodes
  - Some node types have an *expanded-name*, which consists of a *local part* and a *namespace URI*

```
<!-- these have the same expanded-name -->  
<jt:order xmlns:jt='http://www.javatunes.com/order'>  
  
<order xmlns='http://www.javatunes.com/order'>
```

# The Root Node

---

- ◆ The *root node* is the root of the tree
  - It cannot occur anywhere else in the tree
- ◆ The root node is **not** the document or root element -- the root node has as children:
  - Any PI and comment nodes in the prolog
  - A single element node corresponding to the document element
- ◆ The string-value of the root node is the string-value of the document element
- ◆ The root node does not have a name -- we refer to it with **/**

# Element Nodes

---

- ◆ There is an *element node* for each element in the XML document
- ◆ Element nodes can have as children:
  - Other element nodes
  - Comment nodes and PI nodes
  - Text nodes, which contain the character data of the element
- ◆ Attribute nodes are associated with elements but **they are not children of the element** they are associated with

# Element Nodes

- ◆ An element node has an expanded-name with
  - The local part = the element tag name
  - The namespace part = the URI that defines the namespace
- ◆ The string-value of an element node is the **concatenation of the string-values of all its text node descendants**, in *document order*
  - Basically, the element's entire text content

```
<!-- name=person string-value=LeanneRoss25 -->
<person ssn='987-65-4321'>
  <name>
    <firstName>Leanne</firstName>
    <lastName>Ross</lastName>
  </name>
  <age>25</age>
</person>
```

# Text Nodes

- ◆ *Text nodes* can be thought of as the **carriers of the content of the elements**
  - Text nodes cannot have children
  - The characters inside comments, processing instructions, and attribute values do not produce text nodes
- ◆ A text node does not have a name -- we refer to them with **text()**
- ◆ The string-value of a text node is the character data content

```
<!-- no name string-value=25 -->  
<age>25</age>
```

# Attribute Nodes

---

- ◆ Each element node has an associated set of *attribute nodes*
  - Attribute nodes cannot have children
  - The set of attribute nodes is **unordered**
- ◆ An attribute node has an expanded-name with
  - The local part = the attribute name
  - The namespace part = the URI that defines the namespace
- ◆ The string-value of an attribute node is the value of the attribute
  - A defaulted attribute is treated the same as an attribute whose value was explicitly specified in the document

```
<!-- name=ssn string-value=987-65-4321 -->  
<person ssn='987-65-4321' />
```

# Comment Nodes

---

- ◆ There is a *comment node* for each comment
  - Comment nodes cannot have children
- ◆ A comment node does not have a name -- we refer to them with **comment()**
- ◆ The string-value of a comment is the content of the comment
  - Not including the opening **<!--** or the closing **-->**

```
<!-- no name string-value= JavaTunes order document -->  
<!-- JavaTunes order document -->
```

# Processing Instruction Nodes

- ◆ There is a *processing instruction node* for each processing instruction
  - PI nodes cannot have children
- ◆ The name of a PI node is the target
- ◆ The string-value of a PI node is the instruction
  - Including any white space, but not including the ?>

```
<!-- name=xml-stylesheet  
      string-value=href='http://...' type='text/xml' -->  
<?xml-stylesheet href='http://...' type='text/xml'?>
```



# Namespace Nodes

---

- ◆ Each element has an associated set of *namespace nodes*
  - One for each namespace prefix that is in scope for the element
  - One for the default namespace if one is in scope for the element
- ◆ A namespace node has an expanded-name with
  - The local part = the namespace prefix, which is empty for the default namespace
  - The namespace part = null
- ◆ The string-value of a namespace node is the namespace URI which is bound to the namespace prefix

```
<!-- name=jt
      string-value=http://www.javatunes.com/order -->
<jt:order xmlns:jt='http://www.javatunes.com/order'>
```

# JavaTunes Order XML Document

---

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

<!-- JavaTunes order XML document

    see notes about the document type declaration -->
<!DOCTYPE order SYSTEM '... '>

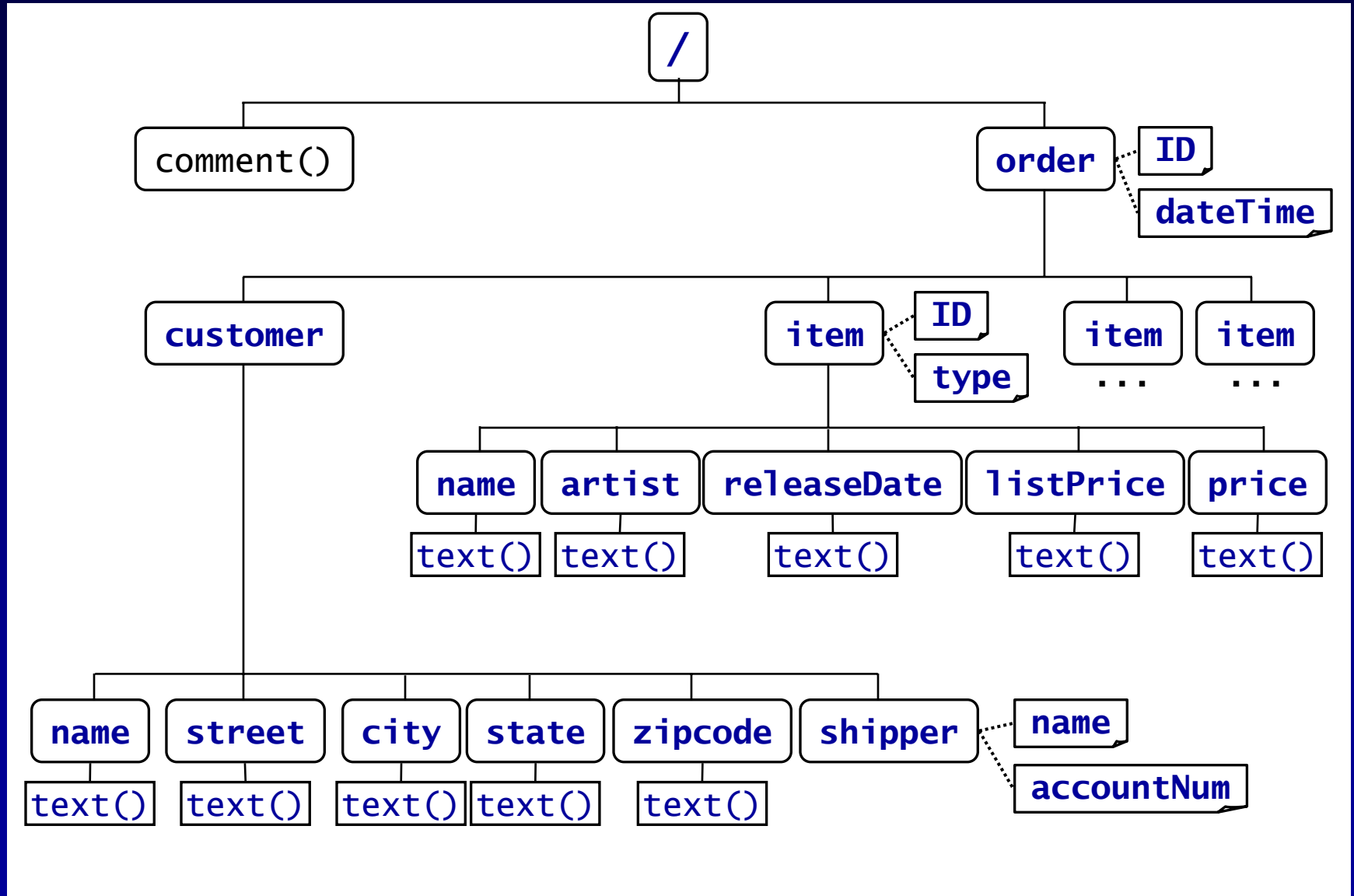
<order ID='_01170302' dateTime='2002-03-20T05:02:00'
  xmlns:xsi='...' xsi:noNamespaceSchemaLocation='...'>
  <customer>
    <name>Susan Phillips</name>
    <street>763 Rodeo Circle</street>
    <city>San Francisco</city>
    <state>CA</state>
    <zipcode>94109</zipcode>
    <shipper name='UPS' accountNum='343-9080-1' />
  </customer>
  ...
```

# JavaTunes Order XML Document

---

```
<?xml version='1.0'?>
<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='CD517'>
  <name>1984</name>
  <artist>Van Halen</artist>
  <releaseDate>1984-08-19</releaseDate>
  <listPrice>11.97</listPrice>
  <price>11.97</price>
</item>
<item ID='CD503'>
  <name>Trouble is...</name>
  <artist>Kenny Wayne Shepherd Band</artist>
  <releaseDate>1997-08-08</releaseDate>
  <listPrice>17.97</listPrice>
  <price>14.99</price>
</item>
</order>
```

# JavaTunes Order as an XPath Tree



## ***Lab 9.1 - Working with Node Trees***

---

- ◆ In this lab, we will work with some XPath node trees and nodes
- ◆ Note: All the XPath labs are pencil/paper, and do not use a computer
  - We'll use it in computer-based labs when we work with XSLT

# ***XPath Expressions***

---

How to Ask for What You Want  
(in an XML Document)

# XPath Expressions - Location Path

---

- ◆ An *XPath expression* is an instance of the XPath language
  - XPath has a number of constructs like *predicates* and *functions* that allow for complex and powerful expressions
  - We will start with a simple type of expression called a *location path*
- ◆ There are two kinds of location paths -- *relative* location paths and *absolute* location paths
  - Absolute location paths start with */* and are computed starting from the root node
  - Relative location paths are computed relative to a context node

# XPath Expressions - Location Path

---

- ◆ A location path is a **series of steps separated by /**
- ◆ Every location path has a ***context node***
  - Initially, the context node is the starting point of the location path
  - As we step through the location path, the context node is recomputed at each step
  - As each step is evaluated, we get a ***node-set*** that matches that step
  - Each node in the node-set is used in turn as the context node to compute the next step
  - These resulting node-sets are combined to produce the resulting node-set for the location path

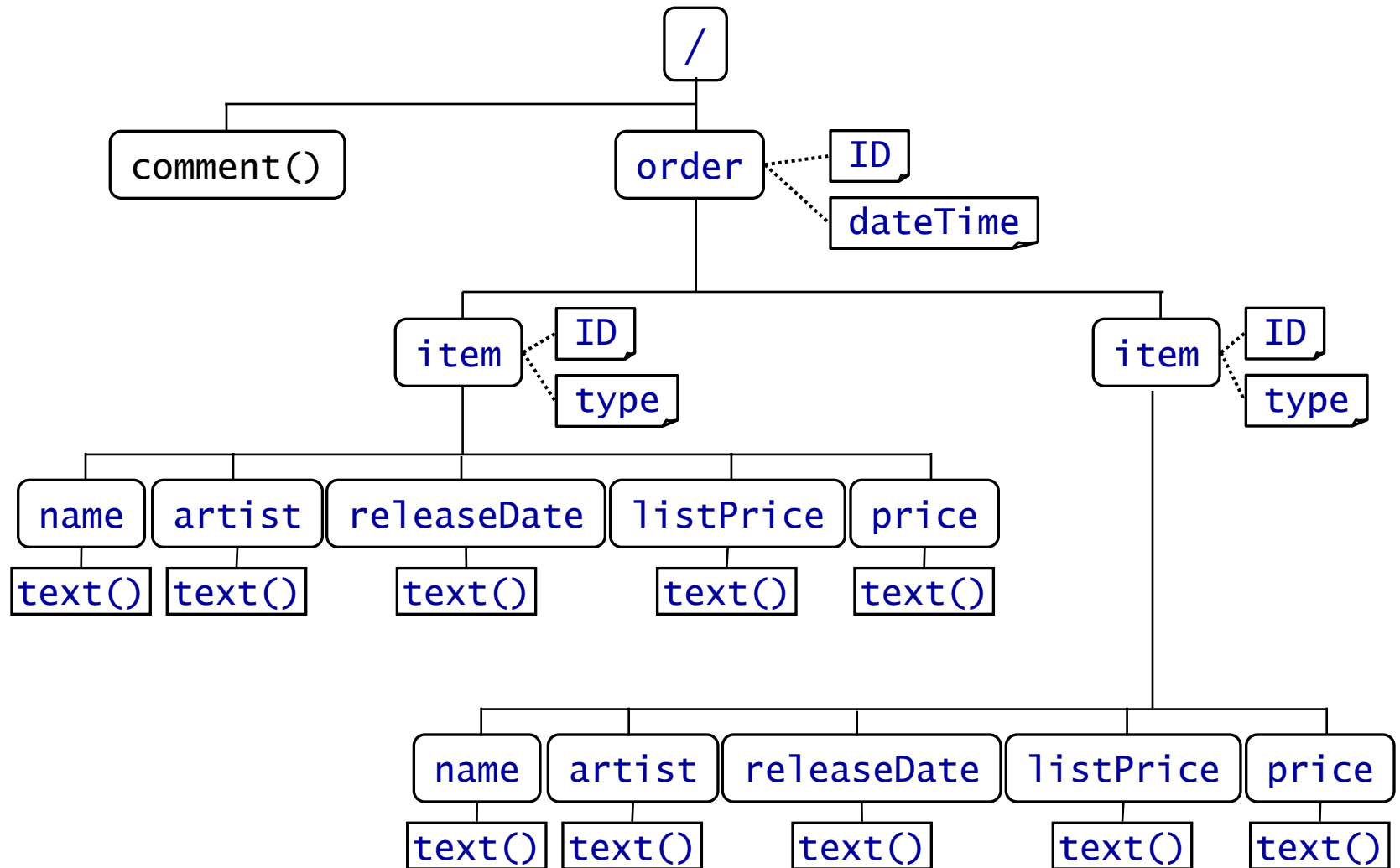


# Location Path - Example

- ◆ The easiest way to get the idea of a location path and the node-sets is to walk through an example

```
<?xml version='1.0'?>
<!-- customer data has been removed for the example -->
<order ID='_01170302' dateTime='2002-03-20T05:02:00'>
  <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='CD517'>
    <name>1984</name>
    <artist>Van Halen</artist>
    <releaseDate>1984-08-19</releaseDate>
    <listPrice>11.97</listPrice>
    <price>11.97</price>
  </item>
</order>
```

# Node Tree for the Example

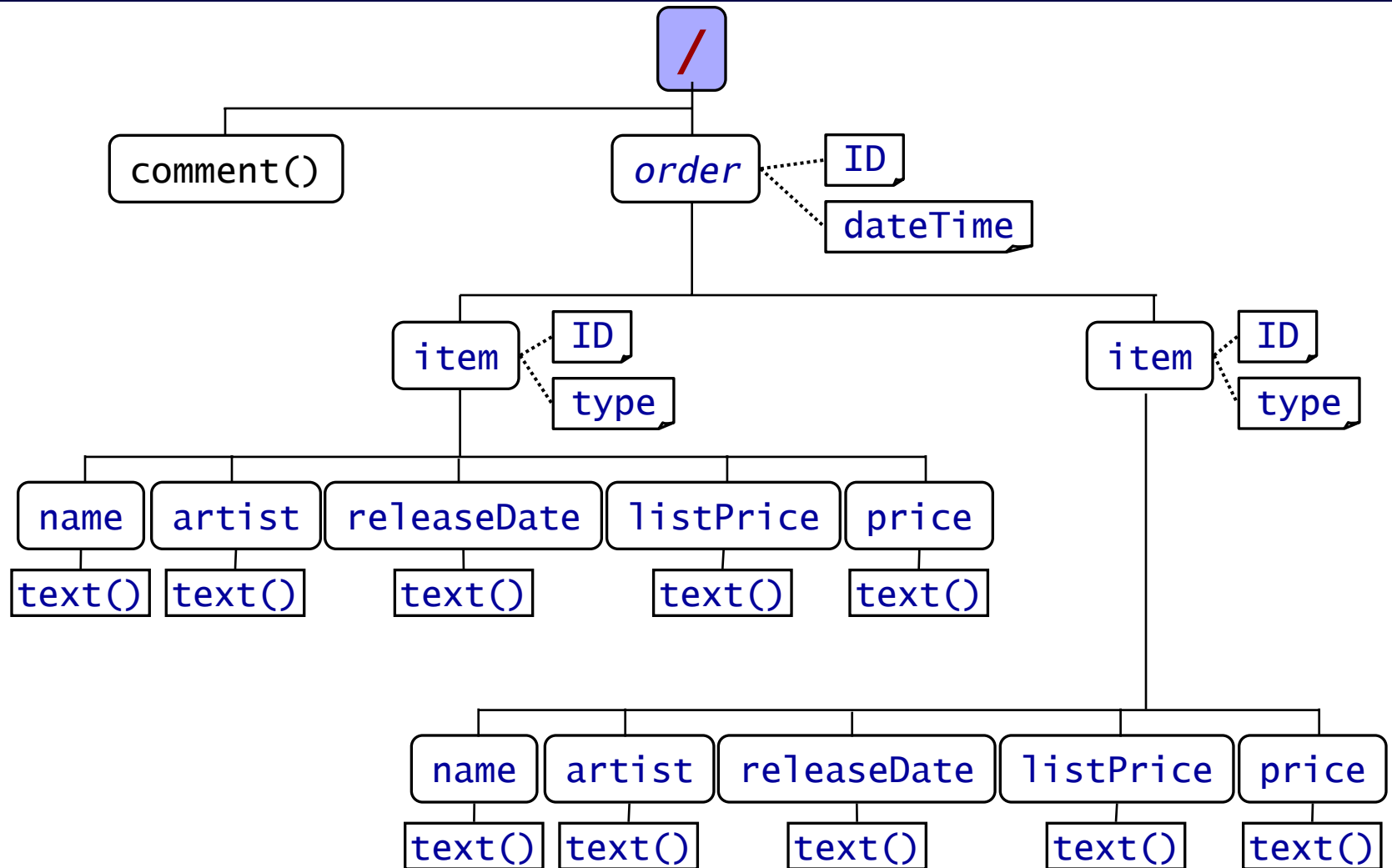


# Location Path Example - Initial Context Node

---

- ◆ The simplest location path is the specification of the root node  
  
/
- ◆ This is the **initial context node** for the location path
  - This is **not** the document element, but the root node, which includes the nodes in the prolog and body

# JavaTunes Node Tree - /



# Location Path Example - First Step

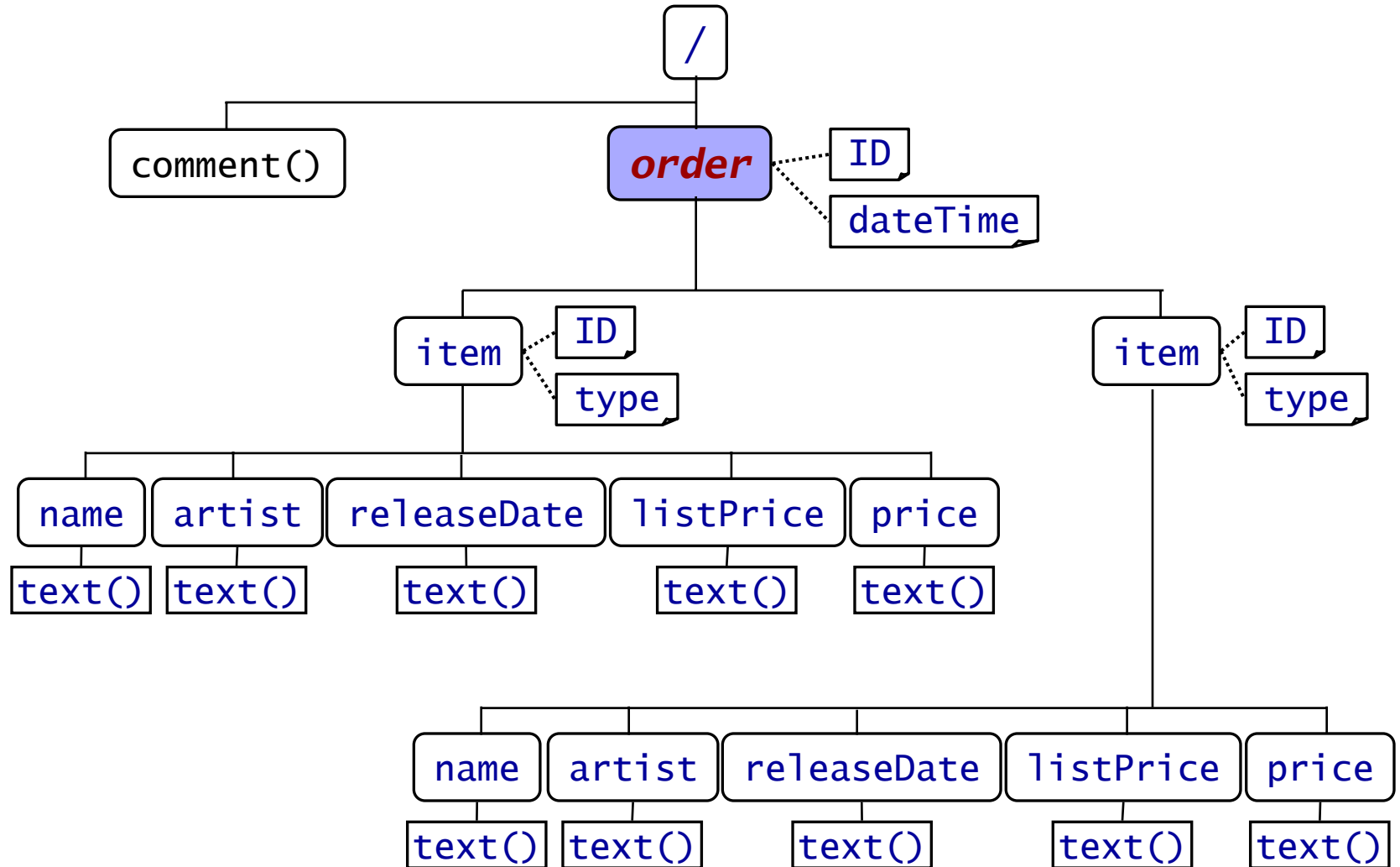
---

- ◆ Next we can step down to the document element with the location path

**/order**

- ◆ Notice that **the document element is now the context node** and that we have one node in our node-set for this step

# JavaTunes Node Tree - /order



# Location Path Example - Second Step

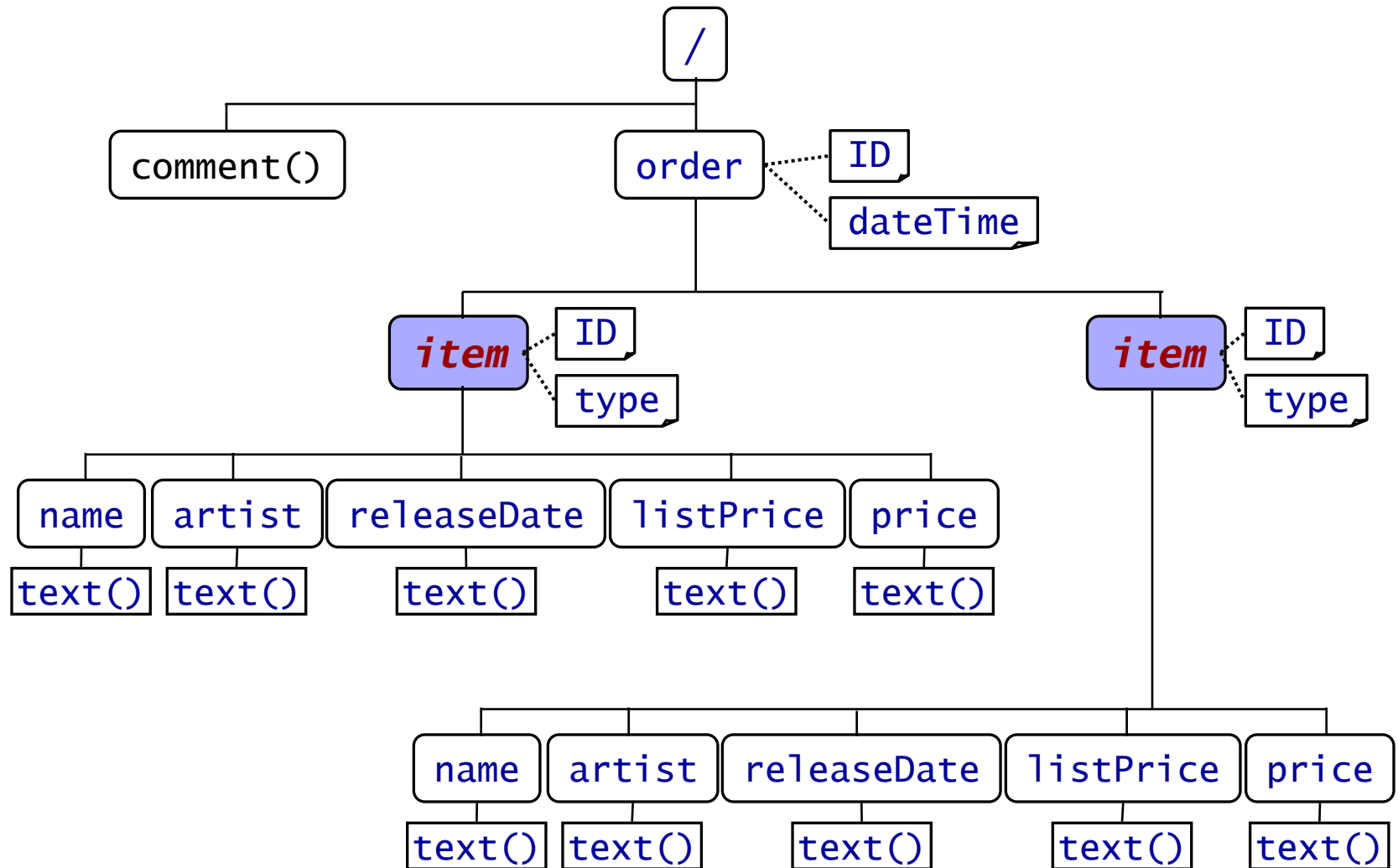
---

- ◆ Now we want to step further down the tree and select the **item** element nodes

**/order/item**

- ◆ **NOTE** that this step produces a node-set with **two nodes**
  - Because there are **two** **item** nodes under the **order** node

# JavaTunes Node Tree - /order/item





# Location Path Example - Third Step

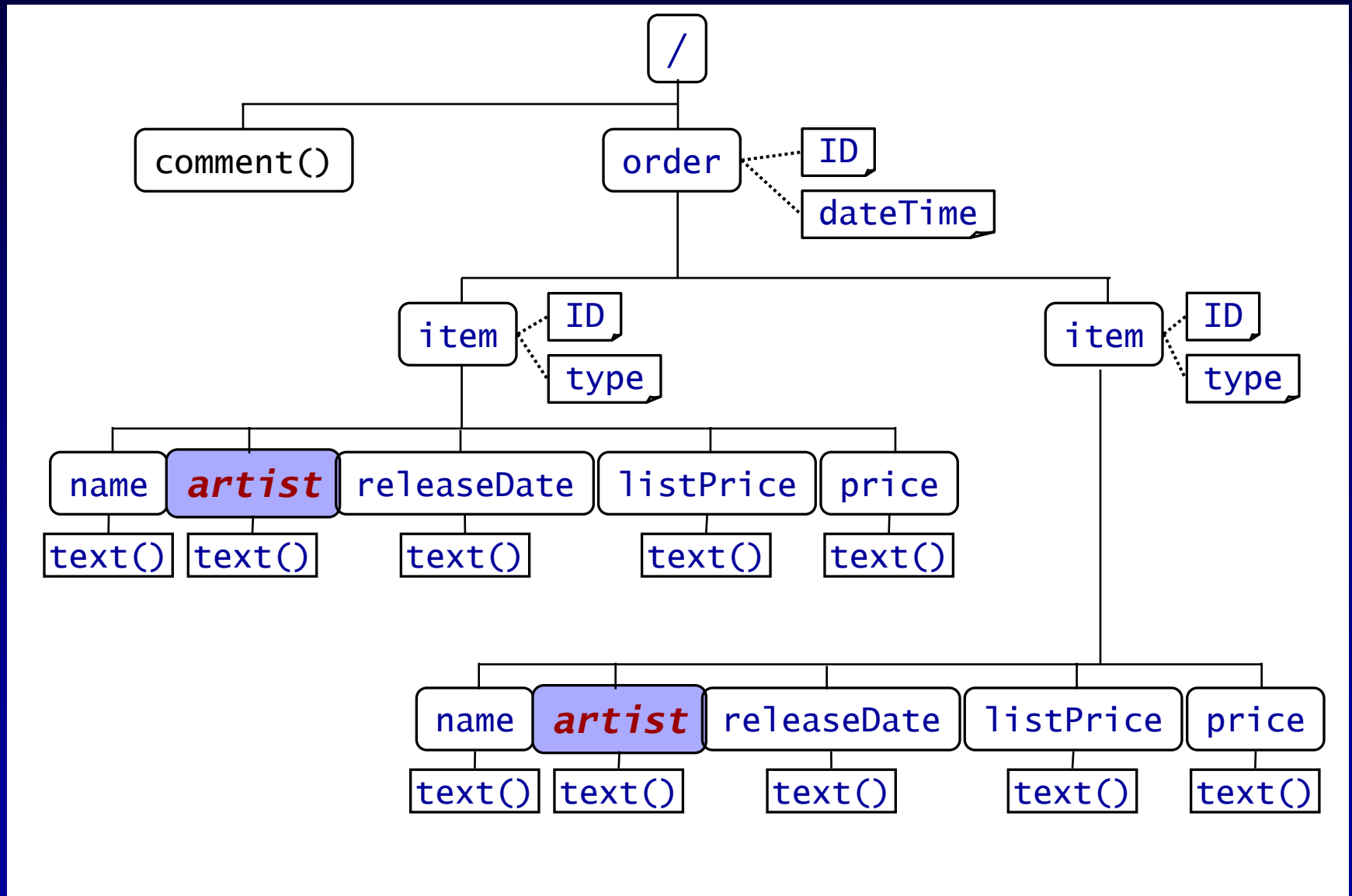
---

- ◆ Now we want to select the **artist** element nodes

**/order/item/artist**

- ◆ To compute the node-set, we let each **item** node we found in the preceding step be the context node, in turn
  - Then, **for each item context node**, we calculate the resulting node-set for this step -- a node-set containing one **artist** node
- ◆ The complete node-set for this step is the union of these two node-sets
  - Two **artist** nodes

# JavaTunes Node Tree - /order/item/artist



# Location Path Example - Getting Attributes

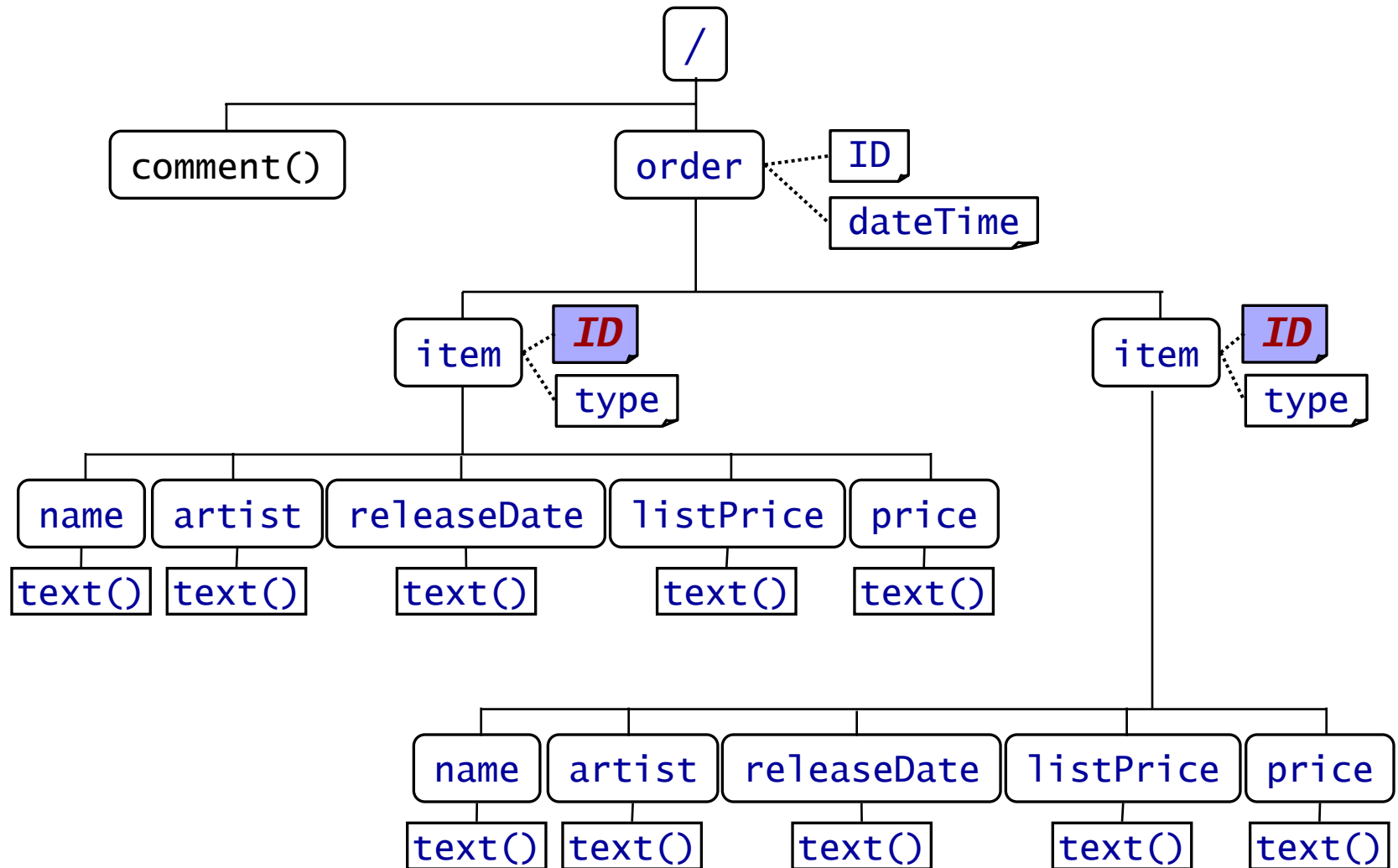
---

- ◆ So far, we have been able to specify element nodes in a location path by using their names
- ◆ Using the syntax **@attribute-name**, we can specify attribute nodes
- ◆ If our location path is:

**/order/item/@ID**

- This results in a node-set containing the two attribute nodes named **ID** that are associated with the **item** elements

# JavaTunes Node Tree - /order/item/@ID



# Location Path Example - Wildcards

---

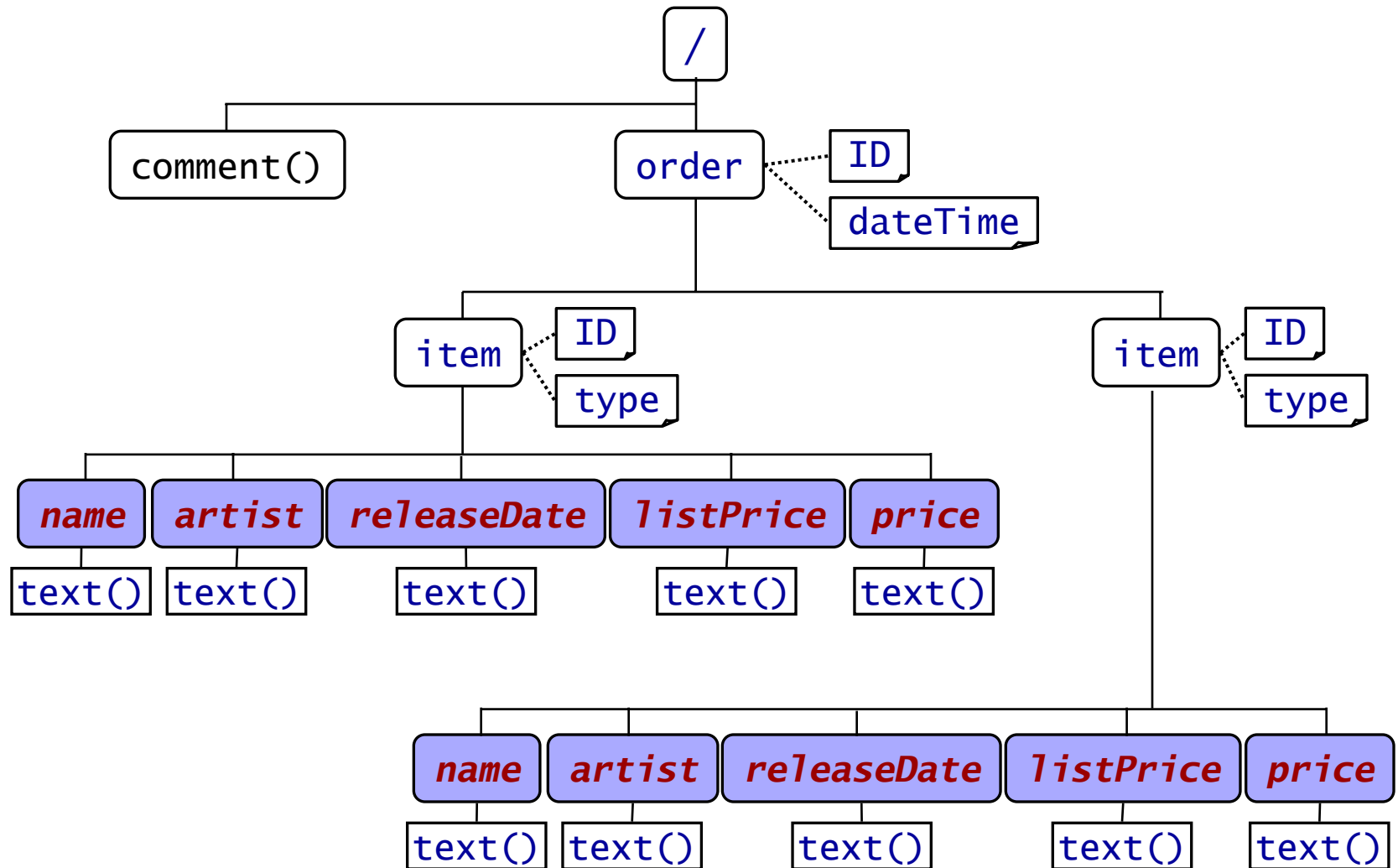
- ◆ XPath allows wildcards to be used in location paths
  - Similar, but not identical, to wildcards in other path languages
  - The wildcard **\*** matches any **name**
- ◆ This location path matches **all the child element nodes** of the **item** nodes

**/order/item/\***

- ◆ **\*** cannot be used for partial matches
  - This location path is illegal

**/order/item/\*price\***

# JavaTunes Node Tree - /order/item/\*



# More on Wildcards

---

- ◆ Wildcards can appear anywhere in the location path
- ◆ This selects element nodes that are children of the root node
  - Is there something special about this location path?

**/\***

- ◆ This selects grandchildren element nodes of the root node

**/\*\***

- ◆ **NOTE** that these wildcards refer only to **element nodes** in the node tree -- no other node type

# More on Wildcards

---

- ◆ This selects **artist** nodes that are great-grandchildren of the root node

**/\*/\*/**artist****

- ◆ These location paths both have an **empty node-set**

**/order/**artist****

**/\*/**artist****

- There are no **artist** child nodes of the **order** node (which is the node matching **/order** and **/\***)
- ◆ The **\*** wildcard refers to only one generation or step in the XPath node tree



# Wildcards and Attributes

---

- ◆ Attributes also have names, and therefore can also be referred to with wildcards
- ◆ This selects all attributes of the **item** nodes

**/order/item/@\***

- How many attributes will the path above select on a JavaTunes **item**?  
**<item ID='CD514'> ...**

- ◆ This selects all attributes of **order**

**/order/@\***

# Location Path Abbreviations

---

- ◆ **▪** (single dot)      **context node**
  
- ◆ **▪ ▪** (double dot)      **parent of context node**
  - Analogous to what we find in a UNIX or Windows file system
  
- ◆ **//**      **context node and its descendants**
  - A location path starting with **//** starts at the root and searches all descendants of it
    - //name** selects customer name **and** item name in a JavaTunes order
    - /name** selects nothing
  - A location path can also include **//** in the middle
    - /article//figure** selects figure descendants of articles, but not of books

# Using an Explicit Leading . (dot)

---

- ◆ You can use this to add clarity to the location path
  - It emphasizes that the rest of the path is relative to the context node
  - Analogous to using it in a file system **cd** command
    - cd XML is the same as cd ./XML (UNIX)
- ◆ **./street**
- ◆ **./\***
- ◆ **./name**
- ◆ **./@ID**
- ◆ **./item/releaseDate**

# Context Node in Action - XSLT Example

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

<xsl:stylesheet
  xmlns:xsl='http://www.w3.org/1999/XSL/Transform'
  version='1.0'>

  <xsl:template match='/order'>
    <xsl:apply-templates select='item' />
    <!-- or -->
    <xsl:apply-templates select='./item' />
  </xsl:template>

</xsl:stylesheet>
```

- ◆ The `match` attribute specifies the context node for the template
  - Therefore, the `select` attribute selects **item children of order**

# Node Tests - Specifying Nodes by Type

---

- ◆ So far, we have been able to specify our node-sets because we were able to identify them by name (and with wildcards)
- ◆ But how do we specify nodes which do not have names -- like text nodes? How do we specify a collection of possibly heterogeneous nodes?
- ◆ A *node test* is a way of referring to nodes by type

# Node Tests Available in XPath

---

- ◆ **node()**
  - Any type of node
- ◆ **text()**
  - Any text node
- ◆ **comment()**
  - Any comment node
- ◆ **processing-instruction()**
  - Any PI node
- ◆ **processing-instruction(' *target-name*')**
  - Any PI node with target ***target-name***

# Node Tests - Examples

---

- ◆ **/order/item/artist/text()**

- Text node children of the **artist** element

- ◆ **/comment()**

- All comments in the prolog

- ◆ **//comment()**

- All comments in the document

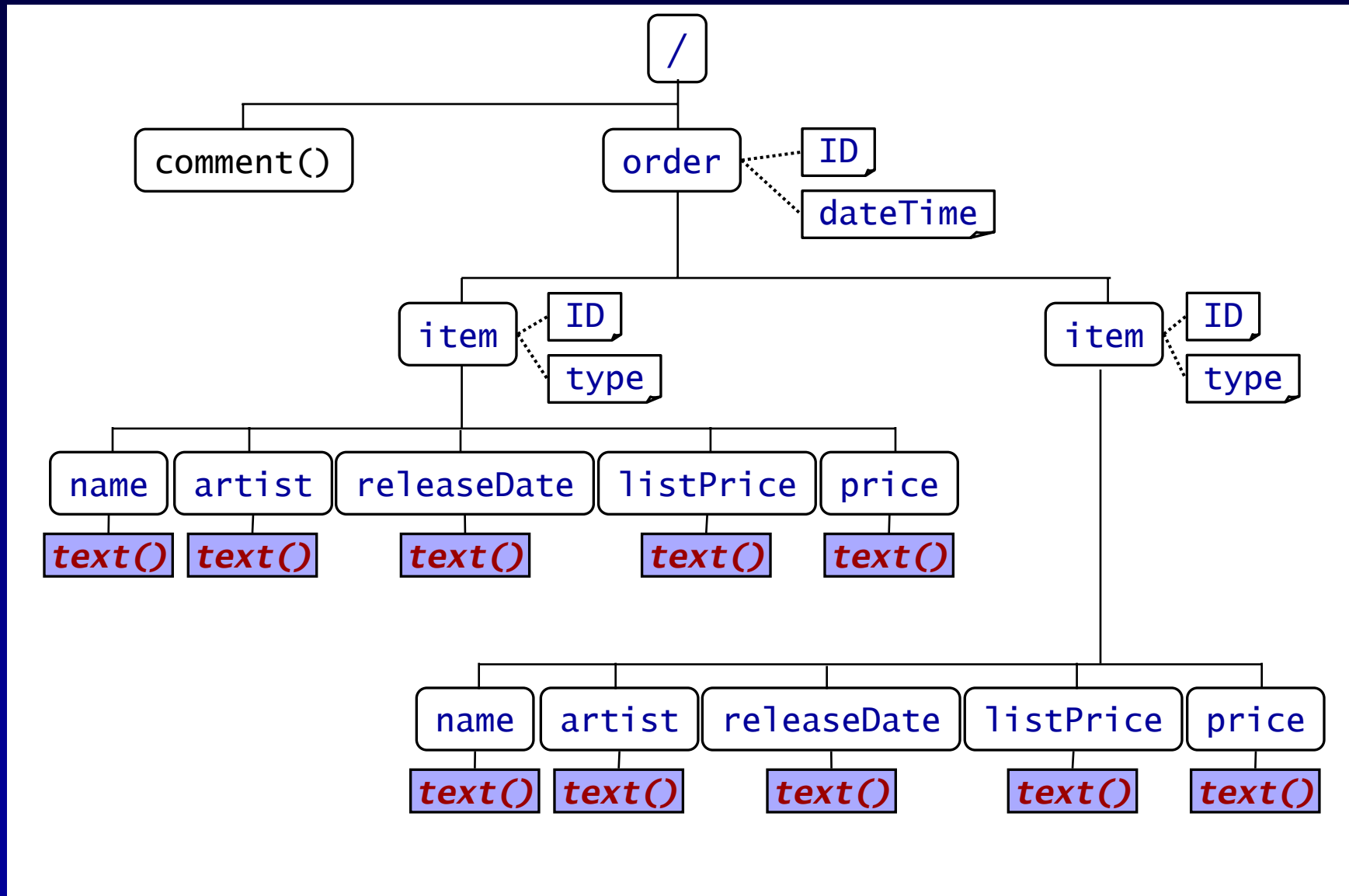
- ◆ **/order/node()**

- All children of the **order** element
- What is the difference between **/order/\*** and **/order/node()**?

- ◆ **/order/customer//text()**

- Text node descendants of the **customer** element

# JavaTunes Node Tree - /order/item/\*/text()





## ***Lab 9.2 - Location Paths***

---

- ◆ In this lab, we will work with a slightly more complex document, and use XPath location paths on it

# *Traversing XML Documents*

---

Which Way Shall We Go?

# Axes

---

- ◆ So far we have been stepping through our trees from parent node to child node
  - Moving down the tree from the root to the branches
- ◆ XPath allows us to specify one of several other “directions of travel” at each step in the location path
  - These directions are called *axes*
  - There are four basic axis types: *element forward*, *element reverse*, *attribute*, and *namespace*
  - These are combined to produce the 13 axes in XPath
- ◆ The axes that are most often used each have an **abbreviated form**, which is shown as we illustrate each one

# Axes Illustrated

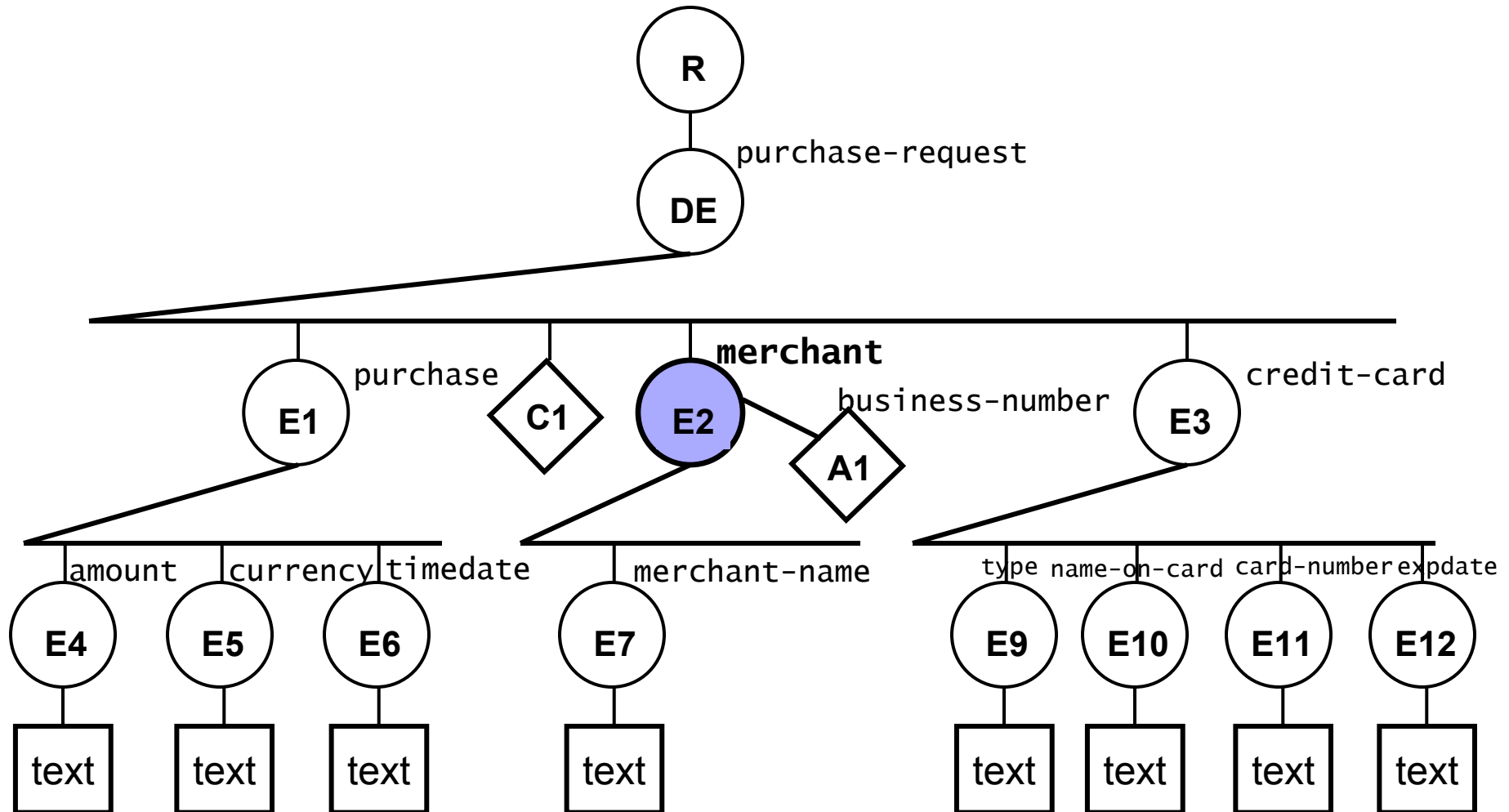
---

- ◆ To better see what these axes look like, let's examine our JavaTunes purchase request document
- ◆ We will identify the nodes in each of the axes
- ◆ The context node will be the **merchant** element node

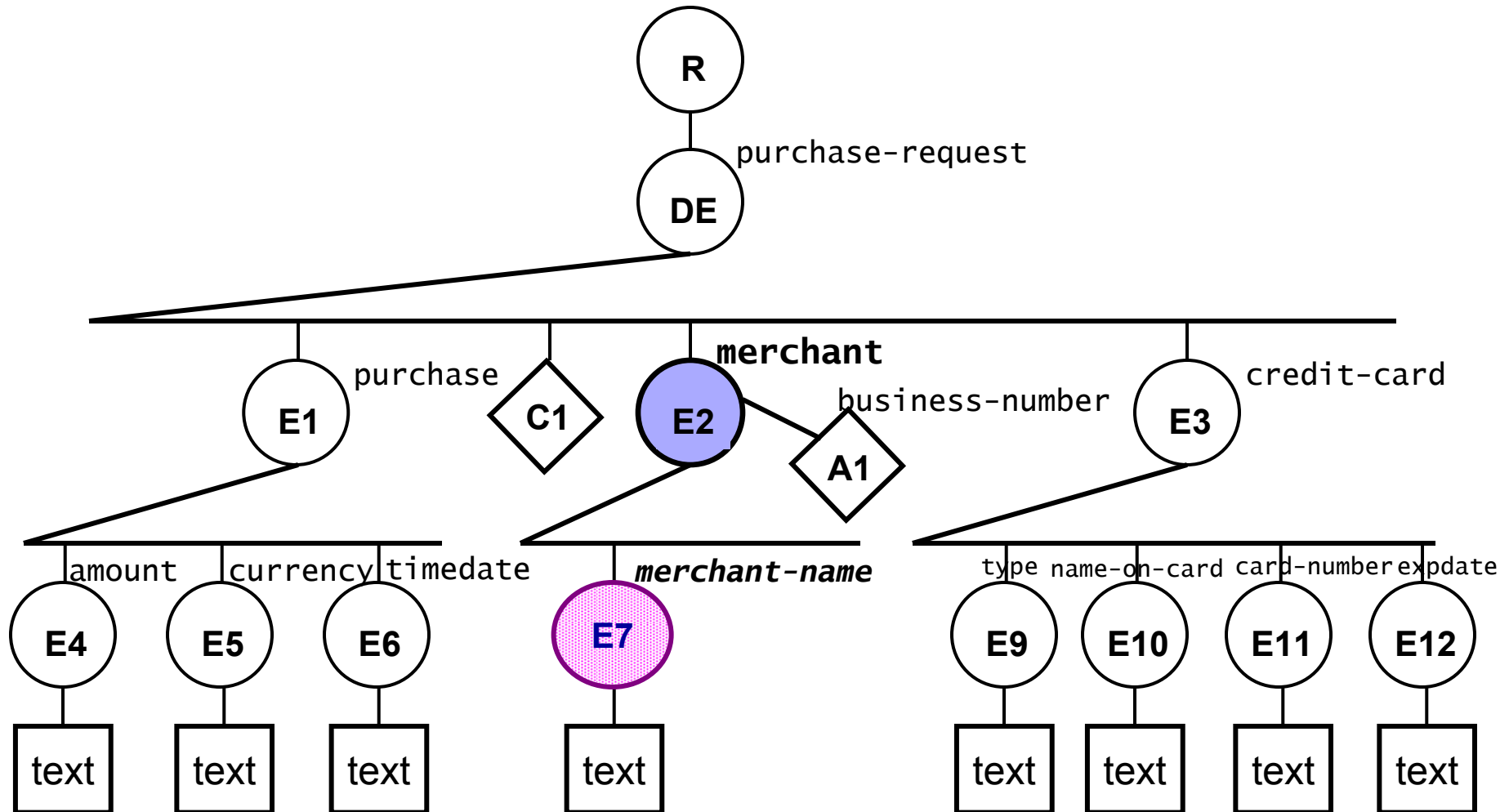
# Axes Illustrated - XML Document

```
<?xml version='1.0'?>
<purchase-request>
  <purchase>
    <amount>10.00</amount>
    <currency>USD</currency>
    <timedate>2003-01-18T14:21:00</timedate>
  </purchase>
  <!-- we changed business-number to be an attribute -->
  <merchant business-number='987676257625'>
    <merchant-name>JavaTunes</merchant-name>
  </merchant>
  <credit-card>
    <type>Visa</type>
    <name-on-card>Bob Smith</name-on-card>
    <card-number>1987987399918277</card-number>
    <expdate>01/04</expdate>
  </credit-card>
</purchase-request>
```

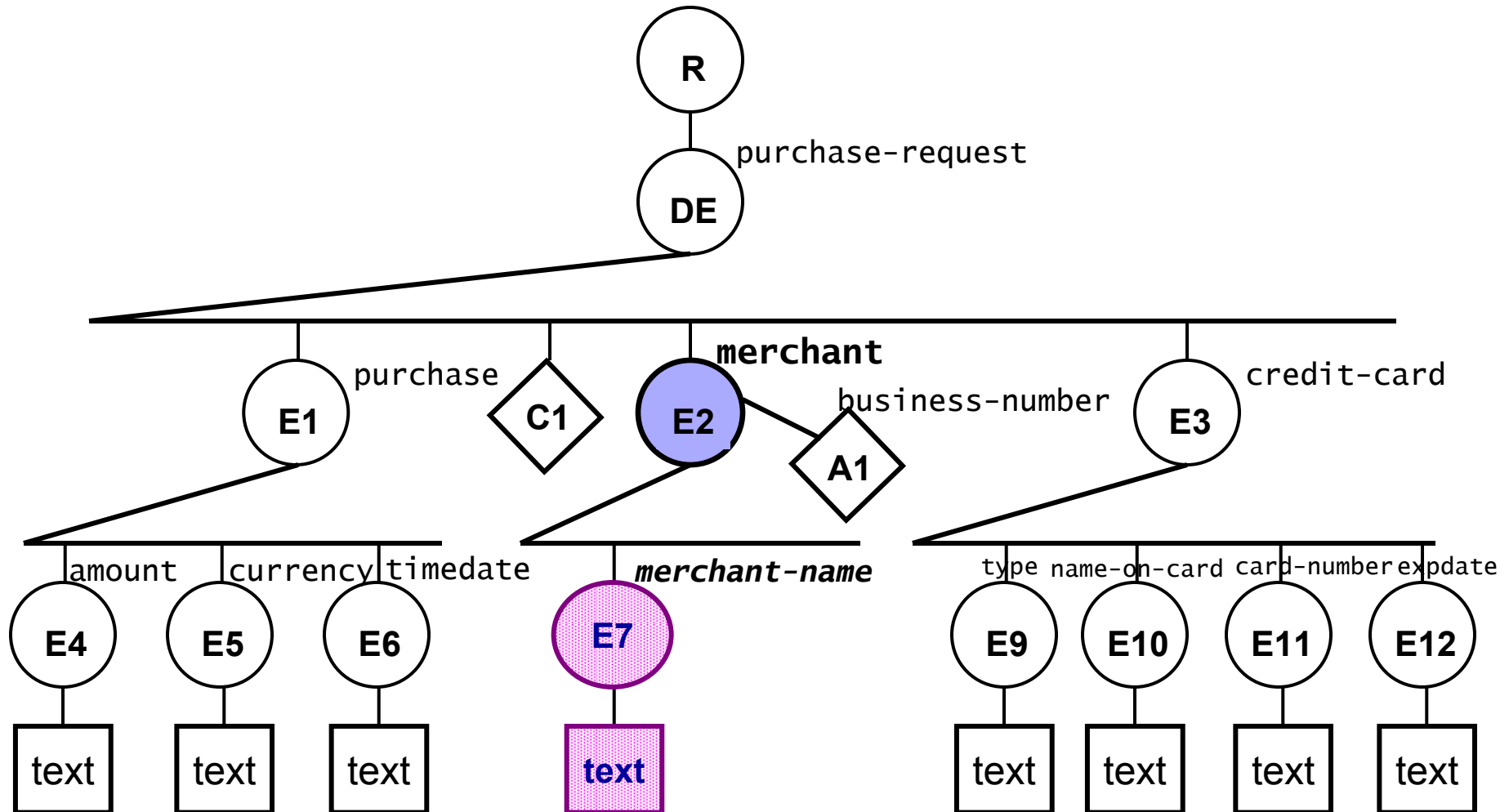
# Axes Illustrated - Node Tree



# Axes Illustrated - child

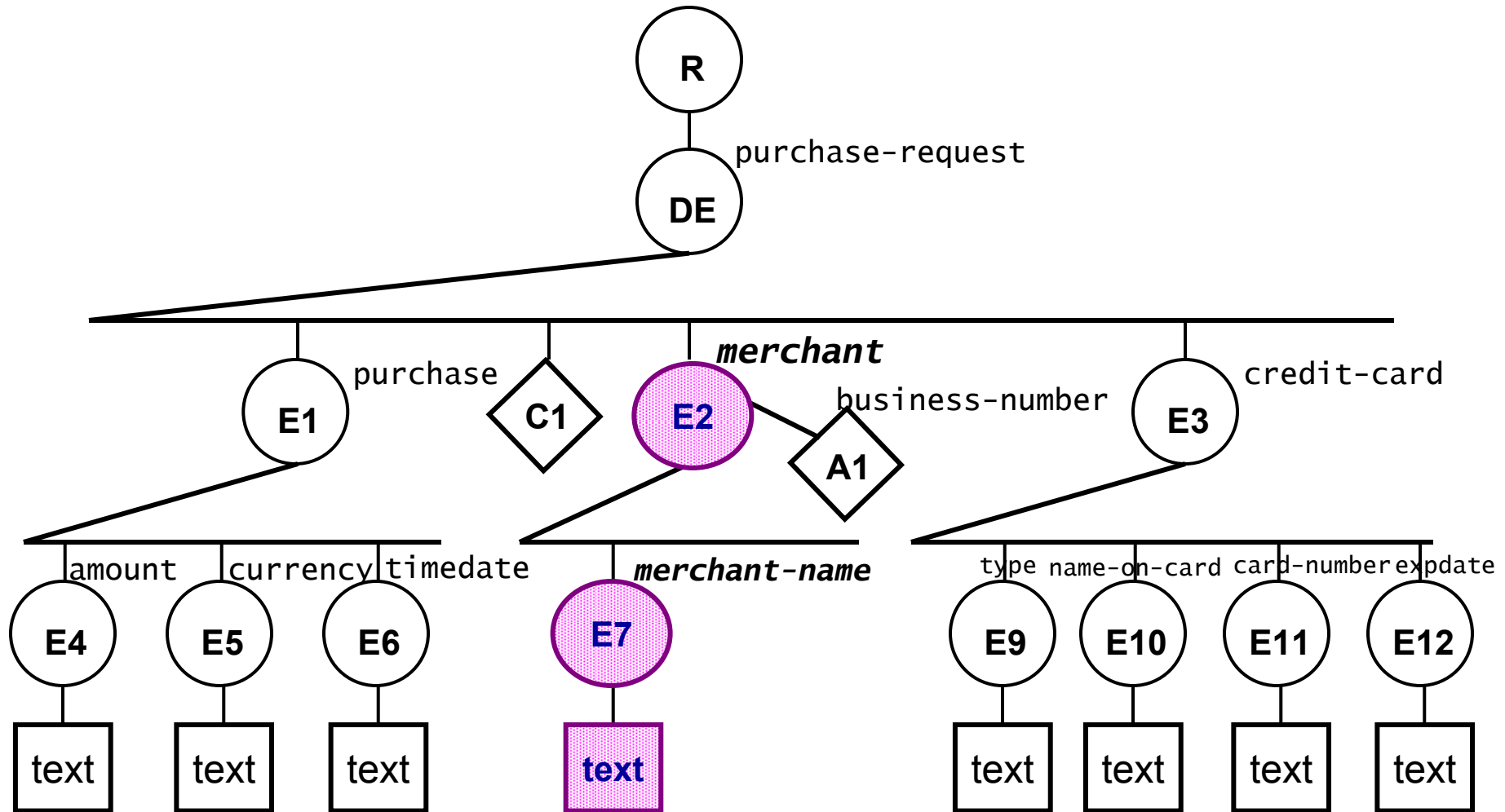


# Axes Illustrated - descendant

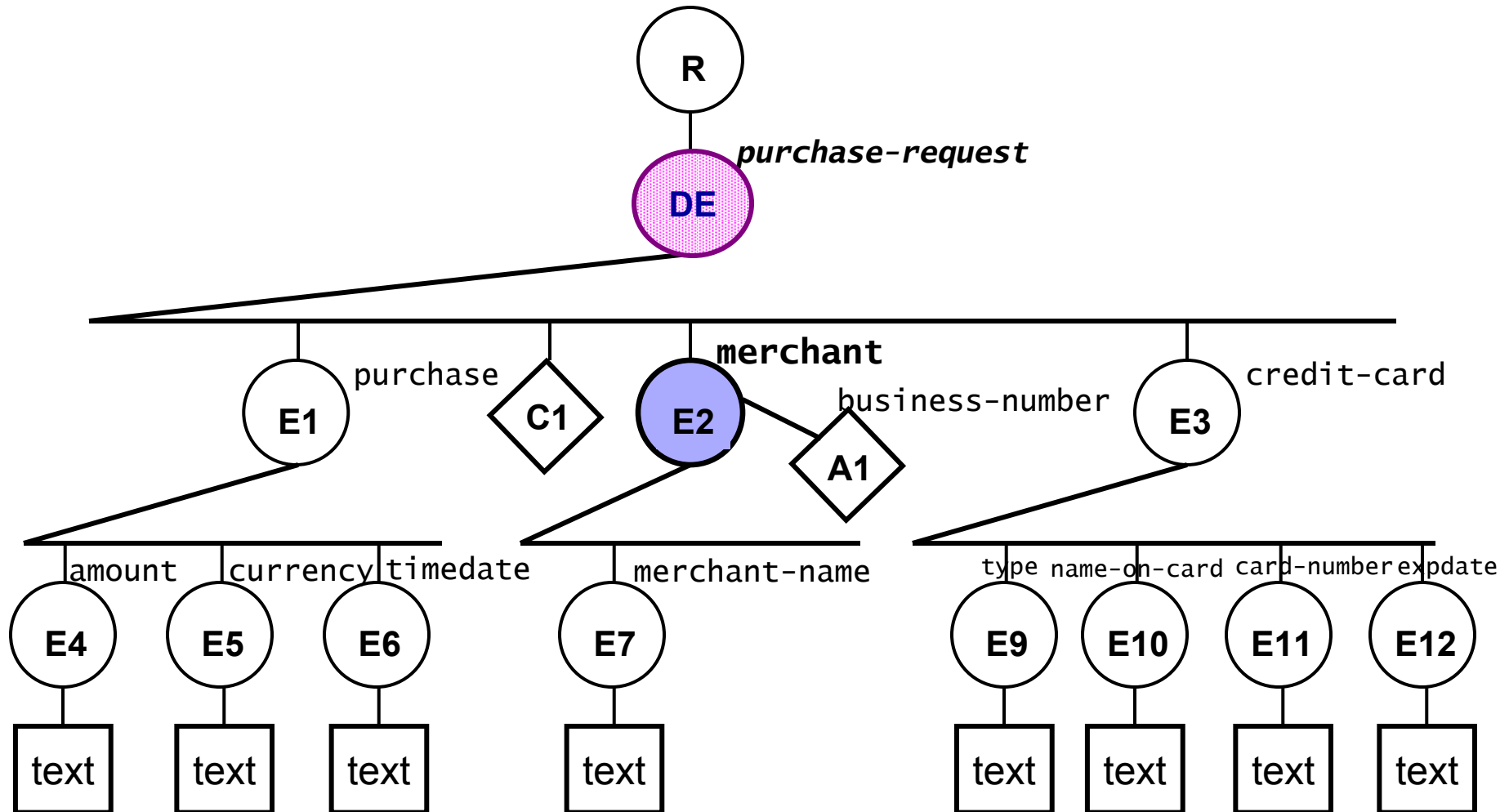




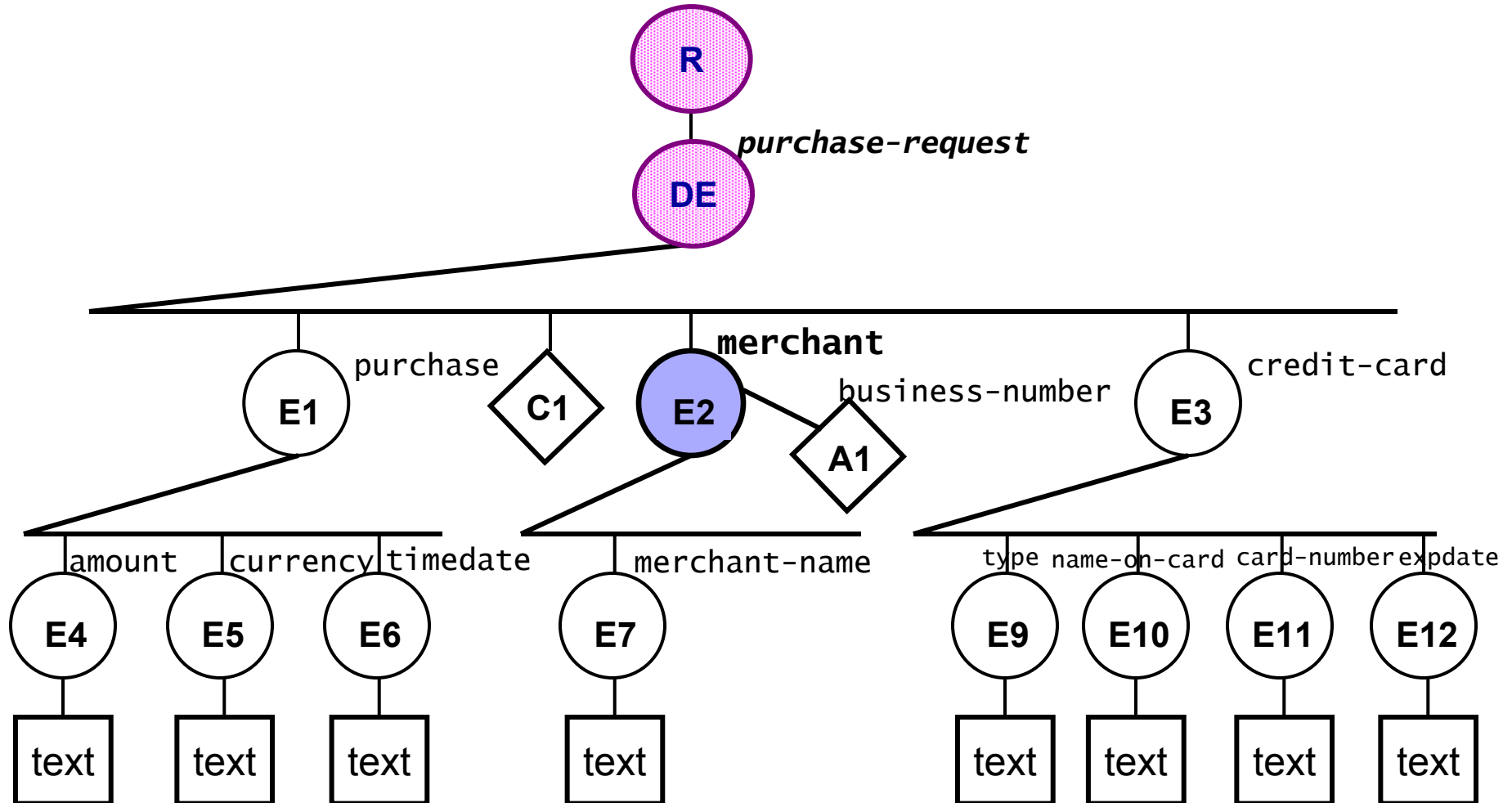
# Axes Illustrated - descendant-or-self //



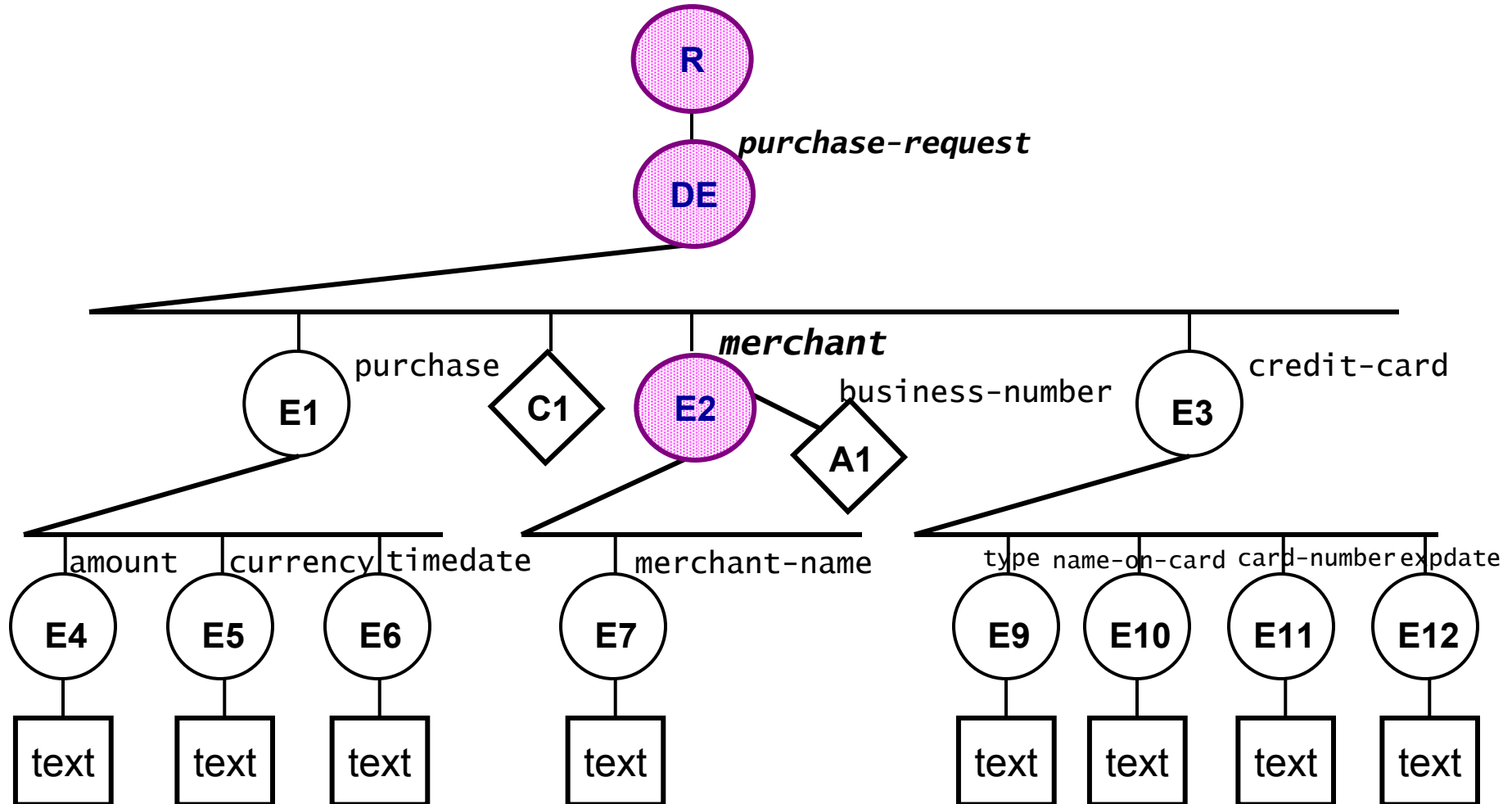
# Axes Illustrated - parent ..



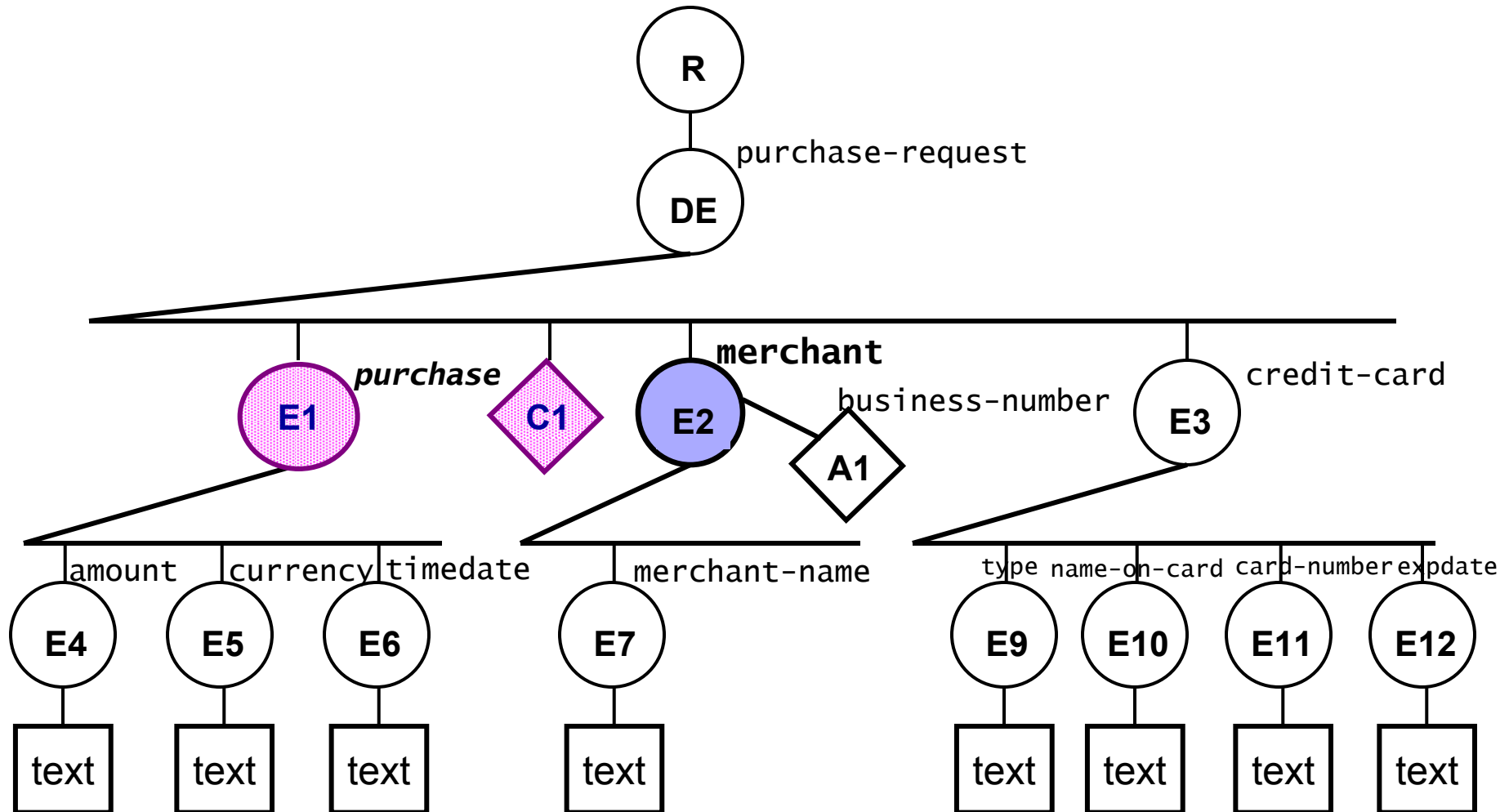
# Axes Illustrated - ancestor



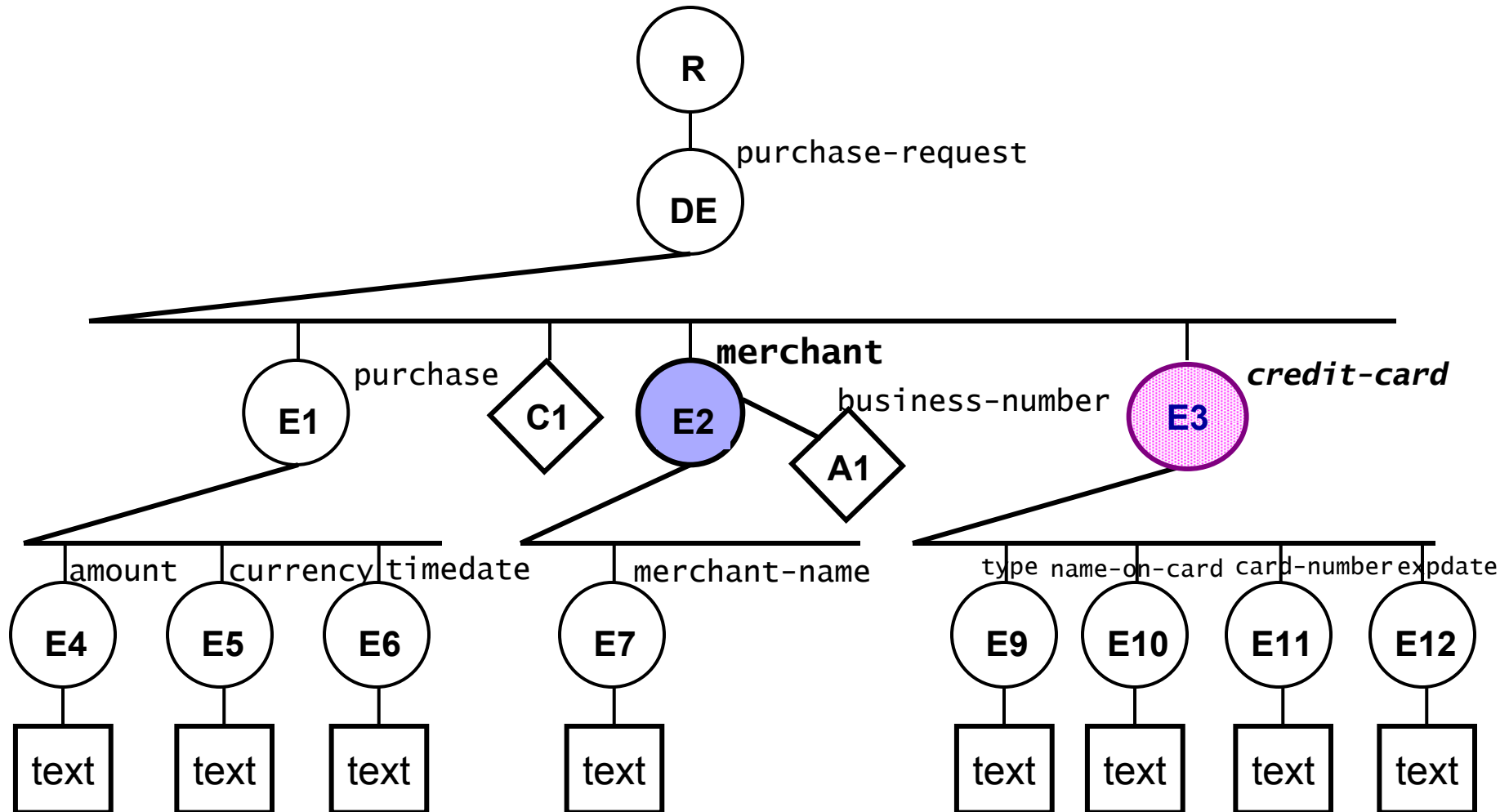
# Axes Illustrated - ancestor-or-self



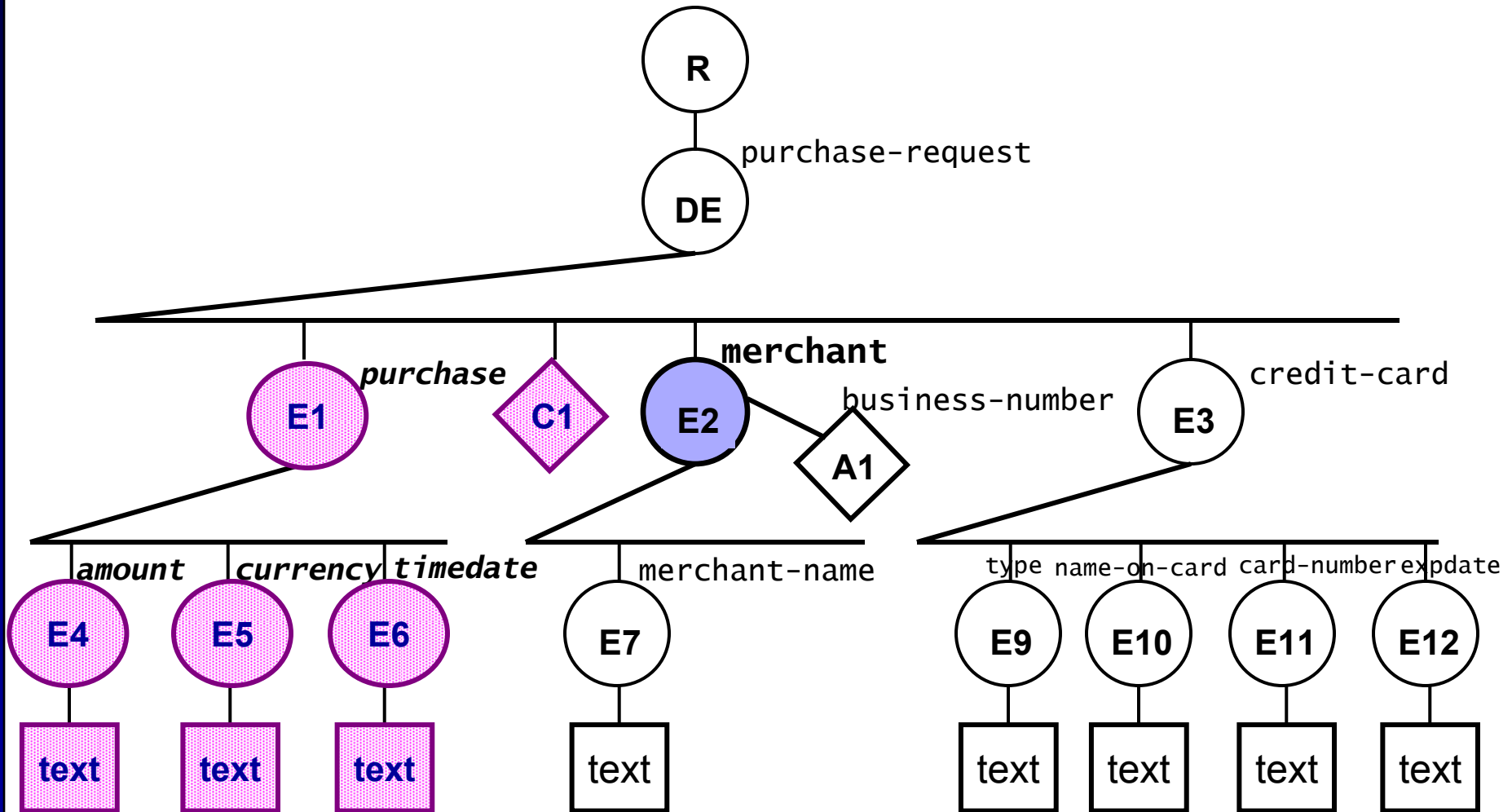
# Axes Illustrated - preceding-sibling



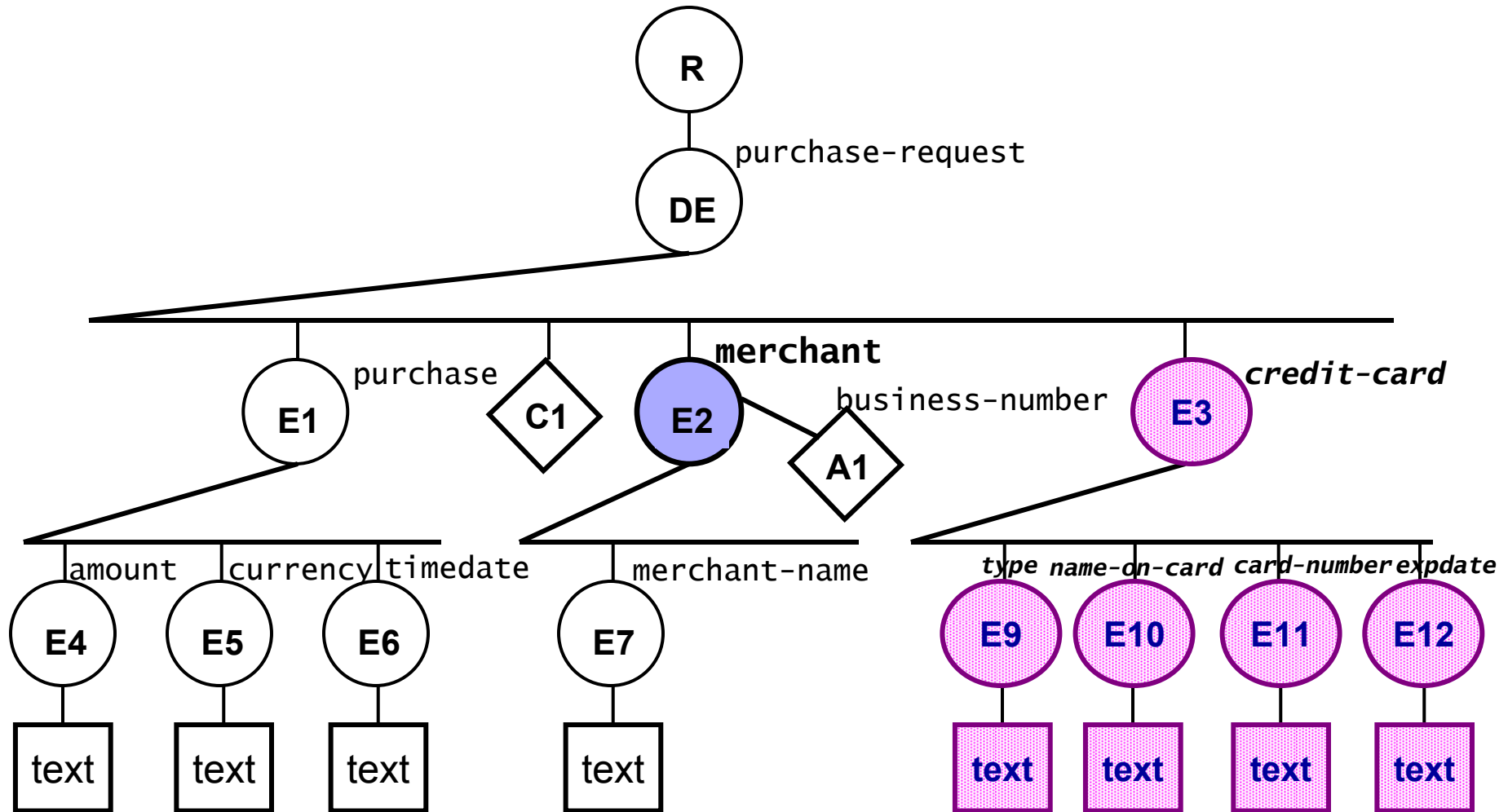
# Axes Illustrated - following-sibling



# Axes Illustrated - preceding

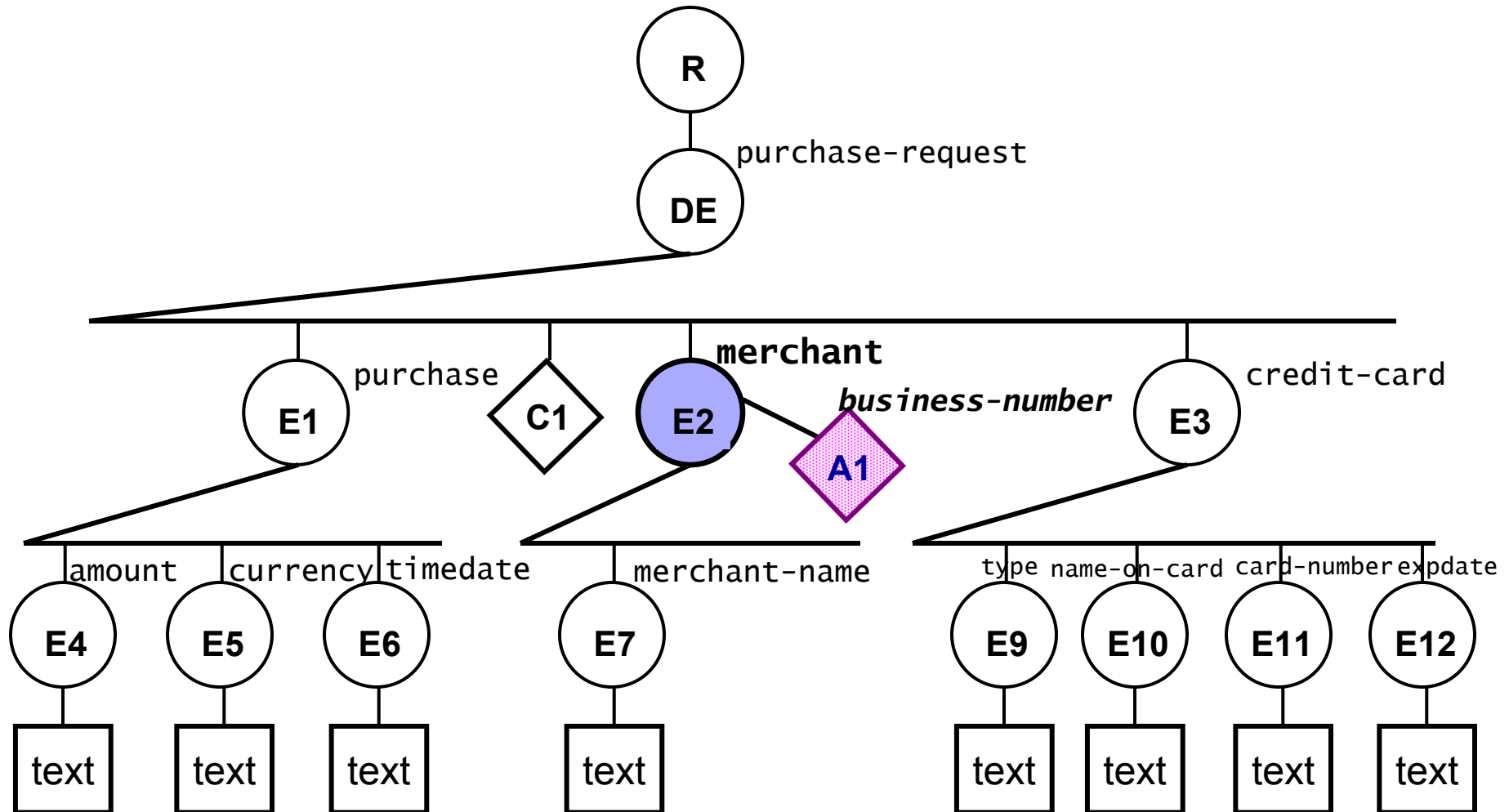


# Axes Illustrated - following

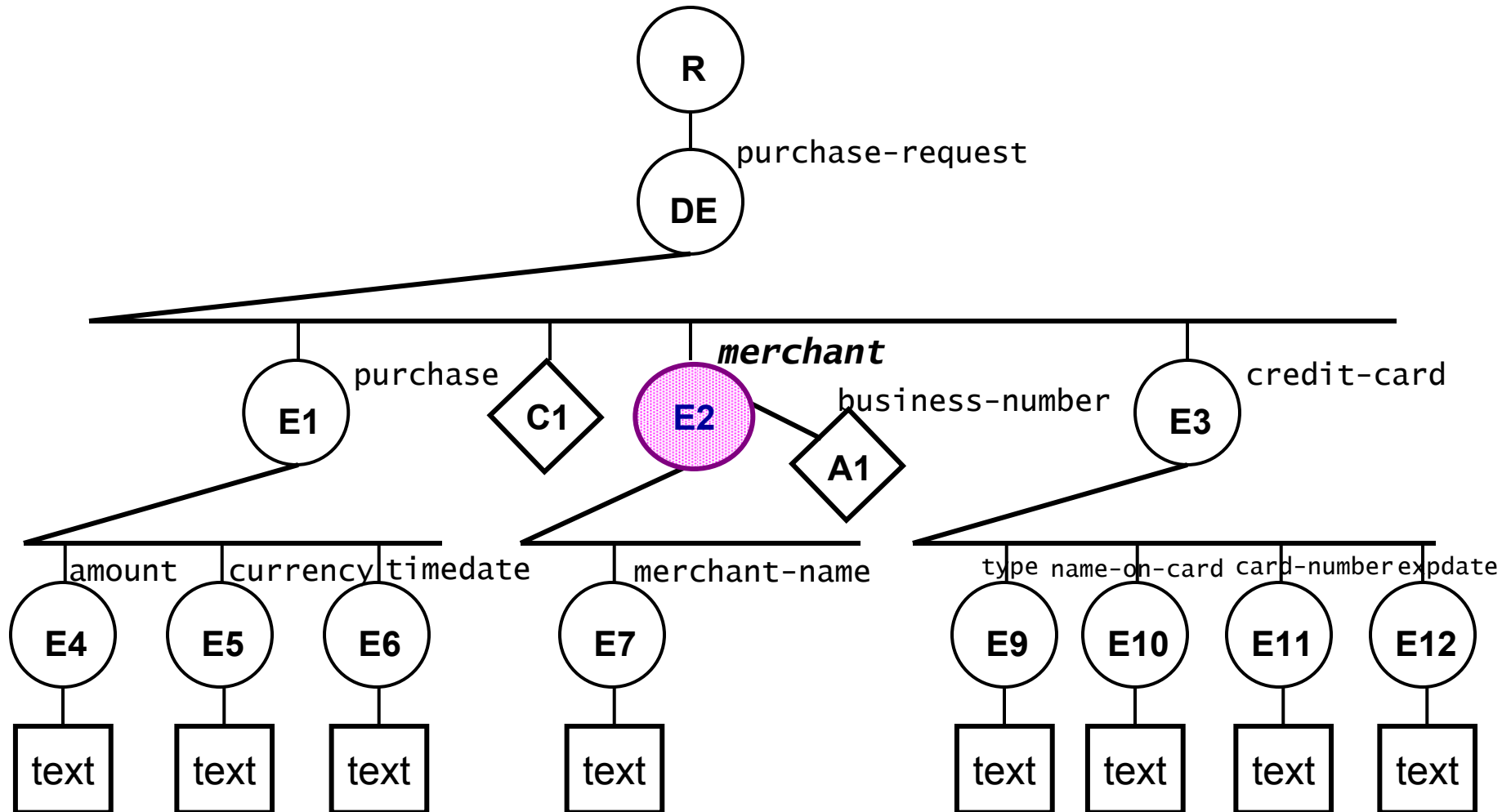




# Axes Illustrated - attribute @



# Axes Illustrated - self



# Axes in Location Paths

---

- ◆ The “direction” or axis to be taken at each step is indicated by a **::** prefix before the node test
- ◆ Thus, the full specification of each step in a location path is ***axis::node-test***
- ◆ As before, these steps can be combined, separated by a **/**
- ◆ **When no axis is explicitly stated, the child axis is used**
  - That is, **/order** is the same as **/child::order**
  - Since the **child** axis is used most often, it is the **default axis**

# Axes in Location Paths - Examples

---

- ◆ `/child::order`  
`/order`
- ◆ `/child::order/attribute::*`  
`/order/@*`
- ◆ `/child::credit-card/attribute::type`  
`/credit-card/@type`
- ◆ `descendant::text()`
- ◆ `parent::name/attribute::title`  
`../name/@title`

# Axes in Location Paths - Examples

---

- ◆ `following::text()`
- ◆ `preceding::comment()`
- ◆ `following-sibling::* / descendant::text()`
- ◆ `ancestor::name`
- ◆ `preceding-sibling::purchase`
- ◆ `descendant::node()`

# Abbreviated Syntax for Popular Axes

---

◆ Now that we know what axes are, let's recap the abbreviations

◆ **no axis specified**                      **child**

◆ **.**                                      **self**

◆ **..**                                    **parent**

◆ **//**                                    **descendant-or-self**

◆ **@**                                    **attribute**

# Don't Get All Chopped up over Axes

---

- ◆ The unabbreviated axes are not used much in practice
- ◆ They aren't even allowed in XSLT template match patterns
- ◆ However, they do provide directions of travel that you may need someday
- ◆ The most commonly used axes are those with abbreviated forms
  - The child axis being most popular, it is the default axis

# ***Predicates***

---

## The WHERE Clause of Xpath



# Predicates

---

- ◆ Predicates are used to filter node sets
  - For example, we might filter out all of the elements that don't have a particular attribute or a particular descendant
  - Or, we might keep only the last node in a node-set
- ◆ A *predicate* in XPath is an expression that can be evaluated to `true` or `false` for a node
  - Predicates are enclosed by `[ ]`
  - Only nodes for which the predicate is `true` are kept
- ◆ Predicates are applied to the resulting node-set of each step
  - Predicates can appear in each step, and more than one predicate can be chained together in a particular step

# Position Predicates

---

- ◆ A *position predicate* refers to the position of the node in the resulting node-set
  - We use the XPath **position()** and **last()** functions to write them
- ◆ **item[position()=1]**  
**item[1]**
  - First `item` child of the context node
- ◆ **item[position()=last()]**
  - Last `item` child of the context node

# Position Predicates - Examples

---

- ♦ **`/doc/chapter[5]/section[2]`**
  - 2nd section of the 5th chapter of the doc
- ♦ **`/doc/chapter[position()=last()]/section[1]`**
  - 1st section of the last chapter of the doc
- ♦ **`/comment()[1]`**
  - 1st comment in the prolog

# More Predicates - Examples

---

## ♦ **chapter[title]**

- All chapter children of the context node that have a `title` child
- Without predicates, we would have to use **chapter/title/..**

## ♦ **chapter[title='Introduction']**

- All chapter children of the context node that have a `title` child with string-value equal to `Introduction`

## ♦ **employee[@secretary and @assistant]**

- All the `employee` children of the context node that have both a `secretary` attribute and an `assistant` attribute

# Chained Predicates and Predicate Ordering

---

- ◆ More than one predicate can be chained together in a particular step
  - **The order of predicates can be meaningful**
- ◆ **//chapter[3][not(title)]**
  - Selects the 3rd chapter node and then keeps it if does not contain a `title` child
- ◆ **//chapter[not(title)][3]**
  - Selects all the chapters which do not contain a `title`, and **then** returns the 3rd one in that set

# Chained Predicates - Examples

---

## ◆ `message[@type='warning']`

- **All** message children of the context node **that** have a `type` attribute with string-value equal to `warning`

## ◆ `message[@type='warning'][5]`

- **5th** message child of the context node **that** has a `type` attribute with string-value equal to `warning`

## ◆ `message[5][@type='warning']`

- **5th** message child of the context node **if** that child has a `type` attribute with string-value equal to `warning`

# XPath Functions

---

- ◆ XPath provides a number of useful functions
- ◆ The format of these function signatures is  
***return-type function-name(type argument, ...)***
- ◆ **number position()**
  - Returns the position of the current node in the context node-set
- ◆ **number count(node-set set)**
  - Returns the number of nodes in *set*
- ◆ **number sum(node-set set)**
  - Returns the sum of the numeric values of the nodes in *set*

# XPath Functions

---

- ◆ **boolean contains(string *s1*, string *s2*)**
  - Returns true if *s2* is a substring of *s1*, i.e., *s1* contains *s2*
- ◆ **boolean starts-with(string *s1*, string *s2*)**
  - Returns true if *s1* starts with *s2*
- ◆ String-values are case sensitive in XPath
  - This should come as no surprise



# XPath Operators

---

- ◆ XPath provides all the usual comparison operators

**=   !=   <   <=   >   >=**

- ◆ And the operators used to combine boolean expressions

**and   or   not()**

- ◆ And the basic arithmetic operators

**+**            addition

**-**            subtraction

**\***            multiplication

**div**          division

**mod**        remainder

# XPath Functions and Operators - Examples

---

## ◆ `/order[count(item)>=5]`

- Selects the order if its number of `item` children is 5 or more
- In English: the order contains at least 5 items

## ◆ `/order[count(item)>=5 or sum(item/price)>=75]`

- Selects the order if its number of `item` children is 5 or more **or** if the sum of its `price` grandchildren' values is 75 or higher
- In English: the order contains at least 5 items or the total cost is at least \$75

# XPath Functions and Operators - Examples

---

## ◆ `//customer[city='New York' and state='NY']`

- Selects the customer if the string-value of its `city` child is New York **and** the string-value of its `state` child is NY
- In English: the customer lives in New York City

## ◆ `//customer[starts-with(zipcode, '94')]`

- Selects the customer if the string-value of its `zipcode` child starts with 94
- In English: the customer lives in the San Francisco Bay Area

## ***Lab 9.3 - Predicates, Functions, Operators***

---

- ◆ In this lab, we will work with the JavaTunes order document, and use more complex XPath location paths

# *Other Uses of XPath*

---

XLink and Xpointer

# XPath is Used in XLink and XPointer

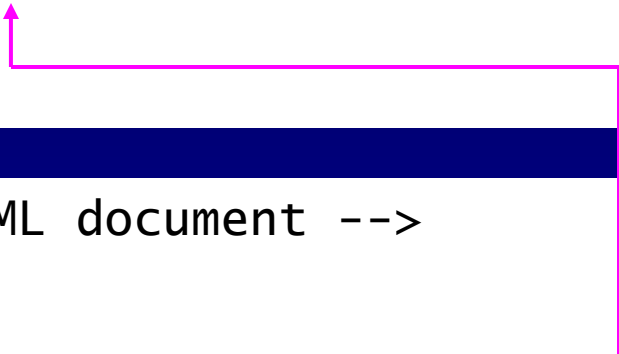
---

- ◆ XPath is a general query language for XML documents
  - Anytime you need to specify a portion of an XML document, think XPath
- ◆ XLink and XPointer are used to create links between documents
  - HTML to XML
  - XML to XML
- ◆ XPath expressions are used to specify which portion of an XML document you want to link to

# Linking from HTML to HTML

- ◆ The **a** element creates links between HTML documents

```
<!-- target HTML document -->
<!-- http://.../chap.html -->
<html>
  ...
  <a name='section1'>This is Section 1</a>
  ...
</html>
```



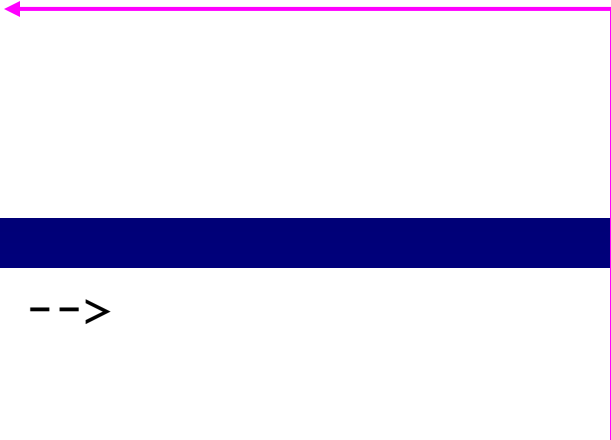
```
<!-- source HTML document -->
<html>
  ...
  <a href='http://.../chap.html#section1'>
    Click here for Section 1
  </a>
  ...
</html>
```

# Linking from HTML to XML

- ◆ The **a** element can also create links between HTML and XML

```
<!-- target XML document -->
<!-- http://.../chap.xml -->
<chap>
  <section>...</section>
  <section>...</section>
  <section>...</section>
</chap>
```

```
<!-- source HTML document -->
<html>
  ...
  <a href='http://.../chap.xml#xpointer(//section[1])'>
    Click here for Section 1
  </a>
  ...
</html>
```





# Linking from XML to XML

- ◆ We can also create links between XML documents

```
<!-- target XML document -->
<!-- http://.../chap.xml -->
<chap>
  <section>...</section>
  <section>...</section>
  <section>...</section>
</chap>
```

```
<!-- source XML document -->
<biblio xmlns:xlink='http://www.w3.org/1999/xlink'>
  <ref xlink:type='simple'
    xlink:href=
      'http://.../chap.xml#xpointer(//section[1])' />
</biblio>
```



- ◆ We use XLink attributes instead of <a href=' '>

# *Introduction to XSLT*

## *Section 10 - XSLT*

---

```
<topic title='XPath and XSLT'>  
  <section num='9' title='XPath' />  
  <section num='10' title='XSLT' />  
</topic>
```

- ◆ What are XSL and XSLT?
- ◆ The XSLT Process
- ◆ Transforming XML to XML
- ◆ Transforming XML to HTML
- ◆ Conditional Processing
- ◆ Client-Side and Programmatic Transforms

# ***What are XSL and XSLT?***

---

Establishing What's What

# What is XSL?

---

- ◆ **XSL** (*eXtensible Stylesheet Language*) is 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

# What is XSLT?

---

- ◆ XSLT is primarily a **language for transforming XML documents into other XML documents**
- ◆ XSLT was originally conceived to transform an XML document into an ***XML-FO document***
  - Which specifies the formatting semantics of how the XML document's content should be displayed
  - An XML-FO document is itself an XML document
- ◆ Completed first, XSLT became a separate recommendation, just like XPath

# XSL and XSLT - Original Plan - Illustrated

```
<?xml version='1.0'?>  
<!-- XML document -->  
<person>  
  ...
```

```
<?xml version='1.0'?>  
<!-- XSLT stylesheet -->  
<xsl:stylesheet>  
  ...
```

**XSLT processor**

**XSL-FO processor**

**Output  
format(s)**

```
<?xml version='1.0'?>  
<!-- XML-FO stylesheet -->  
<fo:root>  
  ...
```

# Uses of XSLT

---

- ◆ The primary goal of XSLT was originally to transform XML documents into other XML documents
  - XSLT converts one XML document structure (or vocabulary) into another XML document structure
  - This is currently a popular use of XSLT, even though the target document is often not an XML-FO document
- ◆ An additional and widespread use of XSLT is to transform XML documents into non-XML document formats
  - **Commonly from XML to HTML, for display in Web browsers**
  - Looking (not far!) ahead, from XML to WML, for wireless devices
- ◆ And there are other ways to do transformations
  - For example, transforming a DOM tree to an XML document

# XSL and XSLT - Current Use - Illustrated

```
<?xml version='1.0'?>  
<!-- XML document -->  
<person>  
...
```

```
<?xml version='1.0'?>  
<!-- XSL stylesheet -->  
<xsl:stylesheet>  
...
```

**XSLT processor**

```
<?xml version='1.0'?>  
<!-- XML document -->  
...
```

```
<!-- HTML document -->  
<html>  
...
```

text document



# The XSLT Processor

---

- ◆ A piece of software that performs the transformation
  - **Input:** XML source document + XSLT stylesheet
  - **Output:** whatever the stylesheet dictates (XML, HTML, plain text)
- ◆ XSLT processors can be run from the command line and from application programs
  - The JAXP **Transformer** class encapsulates the XSLT processor, which can be used in a servlet to transform XML to HTML, for example
- ◆ Web browsers have reasonably good support for XSLT
  - They can do transformations, using the **xml-stylesheet** PI -- we saw this in the XML Introduction

# ***The XSLT Process***

---

Getting from Point A to Point B

# Stylesheets and Templates

---

- ◆ An XSLT transformation is described by a special XML document called a *stylesheet*, which contains *templates*
  - Each template describes how to take a part of the source document and convert it into a part of the result document
  - There are three documents involved -- the *XML source document*, the *XML stylesheet document* and the *result document* (which may or may not be XML)
- ◆ Stylesheets are a collection of template rules that are applied by the XSLT processor as it walks the source tree
  - The order of the templates in the stylesheet is irrelevant to the order in which they are applied, or even if they are applied
  - Think of the stylesheet as a repository or library of templates

# Stylesheet Structure - Basic Form

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

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

```
  <!-- "top-level" elements, which can be in any order -->
```

```
    <xsl:template ...>
```

```
    <xsl:template ...>
```

```
    <!-- other top-level elements -->
```

```
</xsl:stylesheet>
```

- ◆ **version** and the XSLT namespace URI are required
- ◆ The bulk of a stylesheet will be the **xsl:template** elements

# Template Structure - Basic Form

---

```
<xsl:template match='pattern'>
```

```
  <!-- content of template -->
```

```
</xsl:template>
```

- ◆ The ***pattern*** in the **match** attribute determines which node(s) the template applies to
  - And thus determines the context node of the template
  - This pattern is an **XPath location path**
- ◆ The **content** of the template determines:
  - What the processor outputs to the result document
  - Which nodes get processed next

# Flow of the XSLT Processor

---

- ◆ The XSLT processor starts at the root node
  - In XPath, we call this **/**
- ◆ As each node is visited, the processor looks for a template that matches this current **context node**
  - If a template is found, it is applied
  - If not, a default template may be applied (we will look at these default templates soon)
- ◆ In a template, the processor can be directed to apply other templates to the child nodes of this context node
  - To do this, **the processor takes each child node in turn as the context node and repeats the process**

# Flow of the XSLT Processor

---

- ◆ By following this process, the XSLT processor walks the source tree and builds the result tree
- ◆ The key to this tree walking behavior is the directive to process the child nodes of the current context node
  - We may thus have a number of templates active at one time -- one for each level of the recursion
- ◆ If a template is applied to a context node in the source document:
  - And in that template, we apply another template to a child node
  - We now have two templates active
  - The child node template finishes and then the original context node's template resumes

# Processing Child Nodes

- ◆ Child nodes are not automatically processed
  - This needs to be triggered with the **xsl:apply-templates** element

```
<xsl:template match="/">      <!-- processor starts at / -->
  <html>
    <head><title>JavaTunes Order</title></head>
    <body>
      <xsl:apply-templates/>    <!-- process child nodes -->
    </body>
  </html>
</xsl:template>
<!-- other templates - matching on order, customer ... -->
```

- ◆ **Without** `xsl:apply-templates`, the example creates a document with **just** the HTML head and an empty body
  - The processor would stop after outputting `</html>`
  - `xsl:apply-templates` tells processor to visit `/`'s child nodes



# Default Template Rules

---

- ◆ There are three default template rules present in every stylesheet
  - Therefore, a stylesheet with no defined templates (the “empty” stylesheet) **still has these three templates**
- ◆ They direct the processor to:
  - Visit all the element, text, PI, and comment nodes
  - Output the values of the text and attribute nodes that are visited
  - Suppress PIs and comments from the output
- ◆ These default template rules can be overridden by explicitly defining your own templates
- ◆ There is also an internal template rule for namespace nodes

# Default Template for Root and Element Nodes

```
<xsl:template match='/ | *'>

  <!-- process child nodes -->
  <!-- children are element, text, PI, comment nodes -->
  <xsl:apply-templates/>

</xsl:template>
```

- ◆ This template matches **/** and all **element** nodes (\*)
  - Recall that the processor starts at the root node (/)
- ◆ Thus, when the processor begins, this template is activated
  - And it directs the processor to visit the child nodes of /
  - **NOTE** - recall that attribute nodes are **not** children of their elements

# Default Template for Text and Attribute Nodes

```
<xsl:template match='text() | @*'>

  <!-- copy string-value of node to output -->
  <xsl:value-of select='.'/>

</xsl:template>
```

- ◆ This template matches all **text** and **attribute** nodes
  - And it directs the processor to output their string-values
- ◆ Attribute nodes are not visited by default
  - But if you write a template that causes the processor to visit an attribute node, this template rule automatically outputs its value
- ◆ We will discuss the details of `xsl:value-of` later

# Default Template for PI and Comment Nodes

---

```
<xsl:template match='processing-instruction()|comment() '>  
    <!-- empty template - do nothing with these nodes -->  
    <!-- "eat them" - they will not be in the output -->  
  
</xsl:template>
```

- ◆ This template matches all **PI** and **comment** nodes
  - And it directs the processor to do nothing with them
  - Thus, they are suppressed from the output

# Default Template for Namespace Nodes

- ◆ The processor does not copy namespace nodes to the output
  - This rule can't be written in a template -- it must be built in to the processor's source code
  - There is no XPath expression to match namespace nodes
- ◆ The processor automatically inserts necessary namespace declarations (`xmlns*` attributes) into the result document
  - You do not need to write templates to get this behavior (a good thing)

```
<xsl:stylesheet xmlns:xsl=... version='1.0'
                xmlns:jt='http://www.javatunes.com'>
  <xsl:template match=...>
    <jt:order-summary>
      <xsl:apply-templates/>
    </jt:order-summary>
  </xsl:template>
</xsl:stylesheet>
```

## ***Lab 10.1 – Default Template Rules***

---

- ◆ In this lab, you will create an empty stylesheet which uses the default template rules
  - You'll run this stylesheet on your *order.xml* document

# Writing Templates - `xsl:template`

◆ **`xsl:template`** has two main components:

1. The ***match pattern*** -- specifies which node(s) get processed according to the instructions in the template
2. The ***content*** of the template -- specifies what the processor should do
  - There may be instructions to generate output (or not)
  - There may be instructions to direct the processor to visit this node's children (or not)

```
<xsl:template match='order'>
  <!-- "literal result element" - gets output directly -->
  <order-summary>

    <xsl:apply-templates/>    <!-- process child nodes -->

  <!-- "literal result element" - gets output directly -->
  </order-summary>
</xsl:template>
```

# Continuing Processing - `xsl:apply-templates`

- ◆ **`xsl:apply-templates`** triggers all template processing
  - `<xsl:apply-templates/>` -- the context node's children are processed one by one, **in document order**, using template matching
  - `<xsl:apply-templates select='node-set' />` -- the set of nodes in the **select** attribute are processed one by one, **in the order in which they appear in the set**, using template matching patterns

```
<order>
  <customer> ...
  <item> ...
  <item> ...
```

```
<xsl:template match='order'>
  <xsl:apply-templates/>      <!-- customer, item, item -->
  <xsl:apply-templates select='item' />
  <xsl:apply-templates select='customer' />
</xsl:template>
```



# Continuing Processing - `xsl:apply-templates`

- ◆ `xsl:apply-templates` can be used multiple times
- ◆ `select` attribute is not limited to children of the context node
  - Its node-set is specified by an XPath location path
  - In this template, the context node is `order`

```
<xsl:template match='order'>

  <!-- process customer zipcodes specifically -->
  <xsl:apply-templates select='customer/zipcode'/>

  <!-- process item IDs specifically -->
  <xsl:apply-templates select='//item/@ID'/>

  <!-- process all children of context node (order) -->
  <xsl:apply-templates/>

</xsl:template>
```

# Extracting Content - `xsl:value-of`

- ◆ **`xsl:value-of`** inserts the **string-value** of an XPath expression into the result tree
  - The **`select`** attribute holds the XPath expression
  - `select` is a **required** attribute
  - The expression can simply be a node or can use XPath functions

```
<xsl:template match='order'>
  <order-summary>
    <sales-region>
      <xsl:value-of select='customer/zipcode'>
    </sales-region>
    <quantity>
      <xsl:value-of select='count(item)'/>
    </quantity>
  </order-summary>
</xsl:template>
```

# Template Conflict and Template Priorities

- ◆ Only one template can be applied to a given source node
  - Otherwise, we could get unpredictable results
  - But a given source node may match more than one template

```
<person>
  <name>
    <firstName>Leanne</firstName>
    ...
```

```
<!-- all of these match - which one should be applied? -->
<xsl:template match='firstName'>
<xsl:template match='//firstName'>
<xsl:template match='/person/name/*'>
<xsl:template match='*'>
```

- ◆ Every template has a *priority* -- highest priority wins conflicts
  - Templates can be given *explicit priorities*
  - In the absence of an explicit priority, templates have *default priorities*

# Conflict Resolution for Templates

```
<!-- all of these match - which one should be applied? -->
<xsl:template match='firstName'>           <!-- 0 -->
<xsl:template match='//firstName'>         <!-- 0.5 -->
<xsl:template match='/person/name/*'>      <!-- 0.5 winner -->
<xsl:template match='*'>                   <!-- -0.5 -->
```

- ◆ Default priorities are assigned based on the match pattern:
  - Just an element or attribute name **0**
  - Just a node test, e.g., \*, @\*, text() **-0.5**
  - All other patterns **0.5**
- ◆ In the case of a tie (which is technically an error):
  - The processor can abort
  - But usually it applies the one that **occurs last** in the stylesheet
  - Thus, you want to **position your most specific templates last**

# Conflict Resolution for Templates

---

- ◆ For match patterns that are unions, e.g., `name|artist[1]`:
  - Each part of the union has its own priority value
  - `name` has priority 0
  - `artist[1]` has priority 0.5
  - This template has priority 0 when `name` matches and priority 0.5 when `artist[1]` matches
- ◆ The built-in default template rules will never have priority over any explicit templates that you write
  - Regardless of priority conflicts
  - This is because they are loaded first, hence they “appear” before any other templates in your stylesheet

## ***Lab 10.2 – Writing Templates***

---

- ◆ In this lab, you will override some of the default template rules

# *Transforming XML to XML*

---

Transforming Data to Other Data

# Copying Data to the Result - `xsl:copy-of`

---

- ◆ `xsl:copy-of` copies the nodes specified by its **`select`** attribute into the result document
  - Including child nodes and attribute nodes
  - Relevant namespace nodes are also copied
  - Provides a *deep copy*
- ◆ The `select` attribute is **required**
- ◆ `xsl:copy-of` is an empty element
  - That is, nothing can appear between `<xsl:copy-of>` and `</xsl:copy-of>`



# xsl:copy-of - Example

```
<people>
  <person ssn='987-65-4321'>
    <name>
      <firstName>Leanne</firstName>
      <lastName>Ross</lastName>
    ...
  <person ssn='123-45-6789'>
    <name>
      <firstName>Susan</firstName>
      <middleName>Ann</middleName>
      <lastName>Phillips</lastName>
    ...
  ...
</people>
```

```
<!-- copies entire person subtree into the result -->
<xsl:template match='person'>
  <xsl:copy-of select='.'/>
</xsl:template>
```

```
<!-- copies only firstName, middleName, lastName -->
<xsl:template match='person'>
  <xsl:copy-of select='name/*'/>
</xsl:template>
```

# Copying Data to the Result - **xsl:copy**

---

- ◆ **xsl:copy** copies the **context node** into the result document
  - **Child nodes and attribute nodes are not copied**
  - But relevant namespace nodes are copied
  - If the context node is an attribute, it copies the entire attribute, including the name and the value
  - Provides a *shallow copy*
- ◆ No **select** attribute -- it applies only to the context node
- ◆ **xsl:copy** can contain content
  - That is, what appears between `<xsl:copy>` and `</xsl:copy>`
  - This will become the content of the copied node (in the case of an element node)

# xsl:copy - Example

```
<people>
  <person ssn='987-65-4321'>
    <name>
      <firstName>Leanne</firstName>
      <lastName>Ross</lastName>
    ...
  <person ssn='123-45-6789'>
    <name>
      <firstName>Susan</firstName>
      <middleName>Ann</middleName>
      <lastName>Phillips</lastName>
    ...
  ...
</people>
```

```
<!-- copies person element, having only a
      lastName child element as content (see notes) -->
<xsl:template match='person'>
  <xsl:copy>                                <!-- generates <person> -->
    <xsl:copy-of select='name/lastName' />
  </xsl:copy>                                <!-- generates </person> -->
</xsl:template>
```

# xsl:copy/copy-of versus xsl:value-of

- ◆ xsl:copy and xsl:copy-of are used to copy **nodes**, while xsl:value-of is used to copy **string-values**
  - Think about the difference in the templates below

```
<!-- this is a comment -->
<?someTarget this is a PI?>
<people>
  <person ssn='987-65-4321'>
    <name>
      <firstName>Leanne</firstName>
      <lastName>Ross</lastName>
    ...
  </person>
</people>
```

```
<xsl:template match='person'>
  <xsl:copy-of select='name' />
  <xsl:value-of select='name' />
</xsl:template>
```

```
<xsl:template match='processing-instruction()|comment()'>
  <xsl:copy />
  <xsl:value-of select='.' />
</xsl:template>
```

## ***Lab 10.3 – Pruning the Source Tree***

---

- ◆ In this lab, you will generate customized XML output from a JavaTunes order

# Attribute Value Templates

- ◆ Output elements can contain attributes with computed values
- ◆ An *attribute value template* is an XPath expression enclosed by { and }
  - This is placed inside the quotes of an attribute value
  - The expression is evaluated and replaced with its string-value

```
<!-- source document -->
<shipper>
  <name>UPS</name>
  <accountNum>343-9080-1</accountNum>
</shipper>
```

```
<xsl:template match='shipper'>
  <shipping-info name='{name}' account='{accountNum}' />
</xsl:template>
```

```
<!-- result document -->
<shipping-info name='UPS' account='343-9080-1' />
```

# Controlling the Output

---

- ◆ There are some *top-level elements* that control the processor's output semantics
  - Top-level elements appear as children of `xsl:stylesheet`
  - They can appear in any order in a stylesheet
- ◆ We can tell the processor to emit XML, HTML, or plain text
- ◆ We can tell the processor to indent elements in the output
- ◆ We can specify how we want whitespace to be handled
- ◆ For XML output, we can specify things about the XML declaration and the document type declaration

# xsl:output - Specifying Output Format

---

◆ **method**='xml' | 'html' | 'text'

- XML is the default output format
- If **method** is absent and the document element of the result document is **html**, the output format is automatically HTML
- The processor emits HTML that is compatible with Web browsers, e.g., **<br>** instead of **<br/>**
- If the output format is **text**, only text nodes are output

◆ **indent**='yes' | 'no'

- Determines whether elements should be indented with whitespace
- Processor is **not required** to indent if **yes** is specified



# xsl:output - Specifying the XML Declaration

---

◆ **omit-xml-declaration**='yes' | 'no'

- If the output format is XML, the processor provides a default XML declaration

```
<?xml version="1.0" encoding="UTF-8"?>
```

- If the output format is HTML or text, no XML declaration is provided

◆ **encoding**=' *encoding*'

◆ **standalone**='yes' | 'no'

# xsl:output - Specifying a <!DOCTYPE>

---

- ◆ **doctype-public** = '*uri-of-public-identifier*'
  - Specifies the public-identifier for the DTD
  - doctype-system must also be specified
- ◆ **doctype-system** = '*uri-of-system-identifier*'
  - Specifies the system-identifier for the DTD
- ◆ If using an XML Schema, you would specify it by simply including the schema location attribute in the result document
  - Typically in the document element

# Specifying Whitespace Handling

---

- ◆ Two more top-level elements control whitespace handling
- ◆ **xsl:preserve-space** `elements='e1 e2 ...'`
- ◆ **xsl:strip-space** `elements='e1 e2 ...'`
- ◆ In both of them:
  - **elements** is a whitespace-separated list of element names
  - A value of **\*** indicates all elements
  - A value of **prefix:\*** indicates all elements in a namespace
- ◆ Unless specified, whitespace is preserved in elements

# Adding Comments and PIs to the Output

```
<!-- adds a comment - the content becomes a comment -->
<!-- comments are not allowed to contain '--' -->
<xml:comment>
  this is a comment
</xml:comment>
```

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

```
<!-- adds a PI -->
<!-- the name attribute specifies the target -->
<!-- the content specifies the instruction -->
<xml:template match='shipper'>
  <xml:processing-instruction name='pager'>
    <xml:value-of select='@accountNum' />
  </xml:processing-instruction>
</xml:template>
```

```
<?pager 343-9080-1?>
```

## ***Lab 10.4 – Creating an Order Summary***

---

- ◆ In this lab, you will create an order summary document from a JavaTunes order

# *Transforming XML to HTML*

---

Displaying Content in a Web Browser

# Fundamental Approach - Supplying the HTML

---

- ◆ Wrap the desired source document content in HTML elements
  - Explicitly provided elements are called *literal result elements*
  - They are output directly into the result document
  - When transforming to HTML output format, these elements will be standard HTML
- ◆ We used literal result elements in building the order summary
  - We explicitly provided `<order-summary total-cost=' ' ...>` and `</order-summary>` in a template
  - We filled in the attribute values and element content from the JavaTunes order, using `xsl:value-of` and `xsl:copy/copy-of`
  - And we used `xsl:apply-templates` to control the flow of the processor

# Fundamental Approach - What to Match on

---

- ◆ One of the main issues is knowing what to match on to get the desired HTML output
- ◆ For example, when building a table, you want only **one <table>** element, **one <tr>** element **for each row**, and **one <td>** element **for each value** in a row
  - These elements need to go in the appropriate templates so you get the correct occurrences
- ◆ As another example, the `html`, `head`, and `body` elements occur **only once**
  - Output these in a template matching either `/` or the document element
  - Because such a template will only be activated once (get it?)



# Fundamental Approach - Example

```
<xsl:template match='/ '>
  <xsl:copy-of select='comment()[1]'/>  <!-- copyright -->
  <html>
    <head><title>JavaTunes Order</title></head>
    <body>
      <!-- insert order data -->
      <xsl:apply-templates select='order'/>
    </body>
  </html>
</xsl:template>
```

```
<xsl:template match='order'>
  <h1 align='center'>JavaTunes Order</h1>
  <b>Order ID: <xsl:value-of select='@ID'/></b><br/>
  Order Date: <xsl:value-of select='@dateTime'/>
  <hr/>
  <!-- insert customer data -->
  <xsl:apply-templates select='customer'/>
  <!-- insert item data via similarly -->
<xsl:template>
```

# HTML Elements in an XML Document

---

- ◆ Since the stylesheet is an XML document, the XML well-formedness rules apply to these HTML elements
  - All elements must be closed `<p></p>`
  - Elements must nest properly `<b><i></i></b>`
  - Must use XML empty element syntax `<br/>`
  - Attribute values must be quoted `<img src=' '/>`
- ◆ The processor will modify some things for Web browsers
  - When the output format is specified to be HTML
  - For example, it changes `<br/>` in the stylesheet to `<br>` in the output
  - This is because some older browser versions have trouble with “well-formed” HTML

# Basic HTML Tables

---

- ◆ As mentioned, knowing what a template should match on and what to put in that template is key
  - It helps to think about the occurrences of the things you want
- ◆ In the following example, we will put a JavaTunes order's **item** data in a table
  - Take a moment to look at the next slide
  - We want **one <table>** element -- match on something that **occurs once: order**
  - We want **one <tr>** element **for each row** -- match on something that **occurs once for each row: item**
  - We want **one <td>** element **for each value** -- match on something that **occurs once for each piece of data: ID, name, artist, releaseDate, listPrice, price**

# HTML Tables - Example - Source and Result

```
<order ...>                                <!-- only item data is shown -->
  <item ID='CD512'>
    <name>Human Clay</name>
    <artist>Creed</artist>
    <releaseDate>1999-10-21</releaseDate>
    <listPrice>18.97</listPrice>
    <price>13.28</price>
  </item>
  <item ID='CD508'>
    <name>So Much for the Afterglow</name>
    <artist>Everclear</artist>
    <releaseDate>1997-01-19</releaseDate>
    <listPrice>16.97</listPrice>
    <price>13.99</price>
  </item>
</order>
```

Item ID	Name	Artist	Release Date	List Price	Price
CD512	Human Clay	Creed	1999-10-21	18.97	13.28
CD508	So Much for the Afterglow	Everclear	1997-01-19	16.97	13.99

# HTML Tables - Example - Table Framework

```
<!-- one order means we get one set of these elements -->
<xsl:template match='order'>
  <table border='1'>
    <thead>
      <tr>
        <th>Item ID</th>
        <th>Name</th>
        <th>Artist</th>
        <th>Release Date</th>
        <th>List Price</th>
        <th>Your Price</th>
      </tr>
    </thead>
    <tbody>
      <!-- this will insert a row (tr) for each item -->
      <xsl:apply-templates select='item' />
      <!-- close the table tags -->
    </tbody>
  </table>
</xsl:template>
```

Diagram illustrating the structure of the XSLT template for building an HTML table. A large curly brace groups the `<thead>` section (from `<tr>` to `</tr>`) and the `<!-- build the headings -->` comment, indicating that this section is responsible for building the table headings.

# HTML Tables - Example - Table Rows

```
<!-- this will get activated once for each item -->
<xsl:template match='item'>
  <tr>
    <!-- this will insert the data as <td>value</td> -->
    <xsl:apply-templates select='@ID | *' />
  </tr>
</xsl:template>
```

- ◆ Notice that in `xsl:apply-templates` we explicitly selected **@ID | \***
  - We do this to select the ID attribute
  - If an `item`'s data were all in text nodes of child elements, we could simply use `<xsl:apply-templates/>`

# HTML Tables - Example - Table Data

```
<!-- this will get activated once for each value -->
<xsl:template match='item/@ID | item/*'>
  <td>
    <!-- this will insert the value -->
    <xsl:value-of select='.'/>
  </td>
</xsl:template>
```

- ◆ Notice that we didn't use **@\***
  - Recall that our schema specified a **type** attribute with a default value of **CD** -- we do not wish to include that in our table
  - Likewise, we don't use **item/\*** if we don't want all the **item** content
- ◆ **NOTE** that we are leveraging some string-value behavior
  - The string-values of the **item/\*** nodes are the text contents of those elements (see notes)

## ***Lab 10.5 – Transforming to HTML***

---

- ◆ In this lab, you will create an HTML page from a JavaTunes order



# ***Conditional Processing***

---

## Decision Making in Transformations

# Conditional Processing

---

- ◆ XSLT has several constructs that provide for the conditional application of templates
  - These are analogous to the flow of control constructs found in programming languages
  - They allow you to conditionally perform an operation based on some input -- often based on the content of the source document
  - They can be used in generating both XML and HTML output formats
- ◆ **xsl:if**
- ◆ **xsl:choose ... xsl:when ... xsl:otherwise**

# Conditional Processing - `xsl:if`

---

## ◆ `xsl:if test='boolean-expression'`

- The value of **test** is any XPath expression that evaluates to `true` or `false`
- If `test` is `true`, the content of `xsl:if` is processed
- The expression can include XPath functions, and comparison and logical operators (see notes)

## ◆ There is no “else” construct

- Use `xsl:choose ... xsl:when ... xsl:otherwise`

# xsl:if - Example

```
<xsl:template match='order'>
  ...
  <!-- if there are 5 or more items in the order OR
        the total cost of the order is at least 75 -->
  <xsl:if test='count(item)>=5 or sum(item/price)>=75'>
    <!-- output a discount element -->
    <discount rate='.1' />
  </xsl:if>
  ...
</xsl:template>
```

```
<xsl:template match='customer'>
  ...
  <!-- if the customer lives in New York City -->
  <xsl:if test='city="New York" and state="NY"'>
    <!-- output a special message (HTML) -->
    <b>NOTE: 7% surcharge applies to all NYC orders</b>
  </xsl:if>
  ...
</xsl:template>
```

# **xsl:choose ... xsl:when ... xsl:otherwise**

---

- ◆ Using DTD syntax, the content model of **xsl:choose** is (**xsl:when+**, **xsl:otherwise?**)
  - Use for if/else logic -- similar to switch ... case ... default in Java
- ◆ **xsl:when** works just like xsl:if
  - **xsl:when test='boolean-expression'**
- ◆ **xsl:otherwise** provides the else logic
- ◆ **xsl:choose** is processed as follows:
  - Each xsl:when's test expression is evaluated in turn
  - The content of **the first, and only the first** xsl:when whose test is true is processed
  - If none of them are true, xsl:otherwise is processed (if present)

# xsl:choose - Example

```
<xsl:template match='shipper'>
  ...
  <xsl:choose>

    <xsl:when test='@name="FedEx"'>
      <i>Federal Express shipping rate is $10.95</i>
    </xsl:when>

    <xsl:when test='@name="UPS"'>
      <i>UPS shipping rate is $9.99</i>
    </xsl:when>

    <xsl:otherwise>
      Regular shipping charges will apply
    </xsl:otherwise>

  </xsl:choose>
  ...
</xsl:template>
```

## ***[Optional] Lab 10.6 – Conditional Processing***

---

- ◆ In this lab, you will enhance our HTML output from the previous lab

# ***Client-Side and Programmatic Transforms***

---

Life Beyond the Command Line  
OPTIONAL SECTION



# Transforms in a Web Browser

---

- ◆ Web browsers have increasing support for XSLT
  - They can do transformations using the **xml-stylesheet** PI -- we saw this in the XML Introduction course
- ◆ The **xml-stylesheet** PI directs the browser to do the transform

```
<?xml-stylesheet
  href='path-to-stylesheet'
  type='text/xsl'?>
```

  - The browser downloads the stylesheet from the href location and does the transform using its built-in XSLT engine
  - This PI **must appear in the prolog** of the source document

# Transforms in an Application Program

---

- ◆ Transforms can be performed programmatically
  - The JAXP **Transformer** class encapsulates the XSLT processor
- ◆ Server-side applications would likely use programmatic transforms
  - XML into other XML
  - XML into HTML for display in a Web browser
  - XML into WML for display in a wireless device

# Using an XSLT Transformer - Example

```
import javax.xml.transform.*;

class XSLTExample {
    public static void main(String[] args) {
        try {
            TransformerFactory factory =
                TransformerFactory.newInstance();

            // get an XSLT transformer from the factory
            // need to specify its XSLT stylesheet at this point
            Transformer xformer =
                factory.newTransformer(
                    new StreamSource(new File("order-xml.xml")));
            // do the transform - pass in source and result
            xformer.transform(new StreamSource(new File("order.xml")),
                new StreamResult(new File("order-res.html")));
        } catch (Exception e) { System.out.println(e); }
    }
}
```

# ***[Optional] Lab 10.7 – Transforming in Browsers and Java***

---

- ◆ In this lab, you will see how to use XSLT in other environments