

*Introduction to
Programming XML and
Related Technologies*

(Course Code XM341)

Instructor Exercises Guide

ERC 4.1

IBM Certified Course Material

Trademarks

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

ClearCase
Notes

DB2
WebSphere

Everyplace

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

Other company, product and service names may be trademarks or service marks of others.

August 2004 Edition

The information contained in this document has not been submitted to any formal IBM test and is distributed on an “as is” basis without any warranty either express or implied. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer’s ability to evaluate and integrate them into the customer’s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will result elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

© Copyright International Business Machines Corporation 2002, 2004. All rights reserved.

This document may not be reproduced in whole or in part without the prior written permission of IBM.

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Trademarks	v
Instructor Exercises Overview	vii
Exercises Configuration	ix
Exercises Description	xi
Exercise 1. XML Basics	1-1
Exercise 2. Introduction to WebSphere Studio Application Developer	2-1
Exercise 3. DTD	3-1
Exercise 4. XML Namespaces	4-1
Exercise 5. XML Schema	5-1
Exercise 6. XPath	6-1
Exercise 7. XSLT Part 1 - Simple XSL Transforms	7-1
Exercise 8. XSLT Part 2 - Conditional XSL Transforms	8-1
Exercise 9. SAX Parser Programming	9-1
Exercise 10. DOM Parser Programming	10-1
Exercise 11. Generating XML from Java Objects	11-1
Exercise 12. XSLT and Java	12-1
Appendix A. Exercise Solutions	A-1

Trademarks

The reader should recognize that the following terms, which appear in the content of this training document, are official trademarks of IBM or other companies:

IBM® is a registered trademark of International Business Machines Corporation.

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

ClearCase®

DB2®

Everyplace®

Notes®

WebSphere®

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States and other countries.

Other company, product and service names may be trademarks or service marks of others.

Instructor Exercises Overview

The objectives of the Introduction to Programming XML and Related Technologies exercises is to have the students successfully configure an XML development environment and execute the given tasks and assignments. All exercises, with the exception of the first, are completed using WebSphere Studio Application Developer which includes the products of several open source projects, such as Apache Xerces. J2SE 1.4 (or later) is also required; some exercises make use of features that are absent from earlier versions of the J2SE.

Each of the exercises contain source XML documents and some also contain Java source code. None of the exercises have any dependencies other than the specific tools listed above and any possible requirements listed in the exercises introduction.

Ensure that all students have finished exercises and can prove their proficiency in each of the exercises XML technology scope.

Exercises Configuration

It is assumed that the workstation preparation instructions in the XM341 Lab Setup Guide have been followed. There is no allowance made in this course for the students to set up their own workstations.

Exercises Description

The exercises contained within this guide are designed to test and reinforce your knowledge of the topics taught in the lecture. You are given the opportunity to work through each of the exercises given what you learned in the unit presentation.

Exercise 1. XML Basics

What This Exercise Is About

In this exercise, you will begin your introduction to XML by analyzing a business document and creating an XML document from its content. The intent is to get you to start thinking of XML in terms of business applications, and to get you comfortable with writing basic XML documents. This exercise is broken down into three sections.

In the first section you will analyze a business document and identify the logical components in its structure. These components will form the basis of our first XML document.

In the second section you will create a well-formed XML document based on the components you identified in the first section.

In the third section you will test your new document to ensure that it is well-formed.

The decision to model information as content or as an attribute is often arbitrary and therefore a matter of opinion. For this reason and because the choices we made form the basis for the structure used in subsequent exercises, we provide a table that conveys how we chose to model the information we selected to use to describe the problem domain.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Decompose a business document into logical elements
- Create a well-formed XML file from business data
- Verify that an XML document is well-formed using Internet Explorer

Required Materials

- Text Editor
- XML Parser (Internet Explorer 6.x or later)
- File C:\xml\labs\lab1-XMLBasics\exam.xml

Applicability

- XM301
- XM341

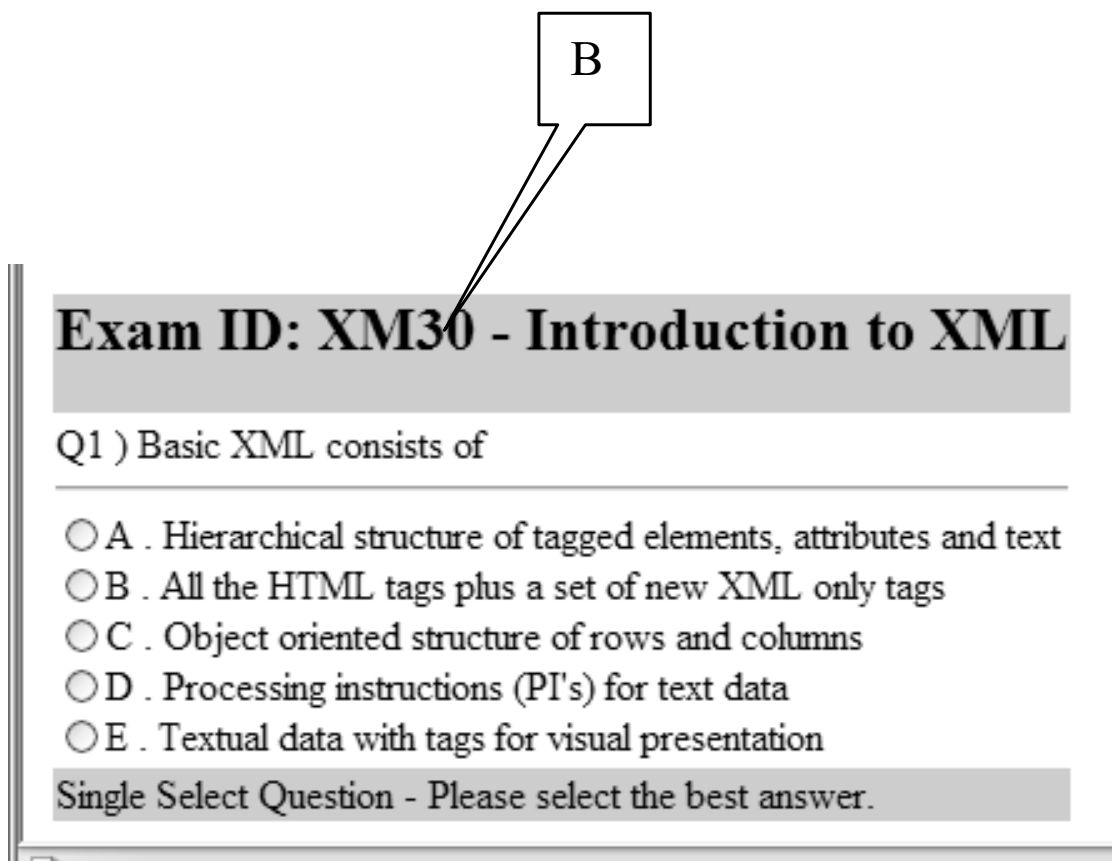
Exercise Instructions

Section 1 - Business Document Analysis

In this section of the lab, you will examine a business document and identify information that will be encoded in XML. Focus on *information* not *presentation* and think in terms of the logical relationships between the information items in the document.

___ 1. Examine the sample exam question shown in this screen capture:

Sample Exam Question



___ 2. For each item listed in the table on the following page:

- ___ a. Identify the item on the screen capture by drawing a circle or square around it.
- ___ b. Mark the circle or square with the letter of the matching item.

Notes: Some items serve to group other items; this helps apply a hierarchical structure to the data.

Some items occur multiple times on the sample; just identify one of the occurrences.

___ 3. Be prepared to justify how you divided and subdivided your data to the class.

The first identifiable item is completed for you as an example.

Letter	Item Name	Type	Description
A	test	Element	Root. Contains all other elements. This represents the entire sample
B	ID	Attribute of test	Test identifier. Value = "XM30"
C	description	Element	Description of test. Value = "Introduction to XML"
D	testQuestions	Element	Grouping element. Contains all the question elements.
E	question	Element	Grouping element. Contains everything related to a single question. Contains items: ID, questionText, and the choices group.
F	ID	Attribute of question	Identifier of a single question. Value = "Q1" (the question number).
G	questionText	Element	Text of the actual question. Value = "Basic XML consists of".
H	choices	Element	Grouping element. Contains all the choices for a given question. Contains items: allowMultiple, and choice
I	allowMultiple	Attribute of choices	Yes or No value: are multiple selections allowed for this question?
J	choice	Element	Grouping element. Contains all information in a particular answer. Note: There are five instances of the element named "choice".
K	ID	Attribute of choice	Identifier for a particular answer.
L	choiceText	Element	Text for the given choice. Value is the text of each possible answer.
M	correct	Element	Yes or No value: is this the correct answer? [The allowMultiple attribute determines if this is a radio button or check box when converting to HTML.]

Section 2 - Create a Well-formed XML Document from Business Data

- ___ 1. In this part of the exercise, you will create an XML document from the information identified in section 1. The raw text from the exam has been provided in the file:

C:\xml\labs\Lab 1 - XMLBasics\exam.xml

- ___ a. Open this file using **Open With > ...** and select **Internet Explorer** (if offered) or **Choose Program...** and select **Internet Explorer** from the pick list.
- ___ b. When the file opens in the browser you should see errors. In the **View** selection on the tool bar, select **Source**. A **Notepad** editor will open with the XML source code already there. This is often the quickest way to access the raw XML code in an editable format. An alternate approach would be to open the XML file directly using the **Open with...** command, but then you will have to make the association yourself.

Note: Having both a browser and an editor open allows you to use the Save command in the editor and the refresh/reload command on the browser to easily and quickly evaluate your work. As you see in the following, it is not necessary to have a browser open; however, doing so will ultimately prove efficient.

- ___ 2. Use the figure and table from the previous two pages to help you identify the *elements* and *attributes* to model. You should already have identified **test** as the *root element*.
- ___ a. Open the **exam.xml** file indicated above using a text editor like **notepad**.
- ___ b. *Best practices* dictate we begin with the XML Declaration **<?xml version="1.0"?>**.
- ___ c. Create the appropriate root element for the document, **test** remembering that every opening tag must have a corresponding closing tag (unless it is an *empty tag*).
- ___ d. Complete the document by adding the rest of the elements and attributes you identified.
- ___ e. Your document should be a well-formed XML file; we show you how to prove this in Section 3.
- ___ f. When you have completed your document, save it with the same file name of **"exam.xml"**.

Section 3 - Verify that an XML document is well-formed using Internet Explorer

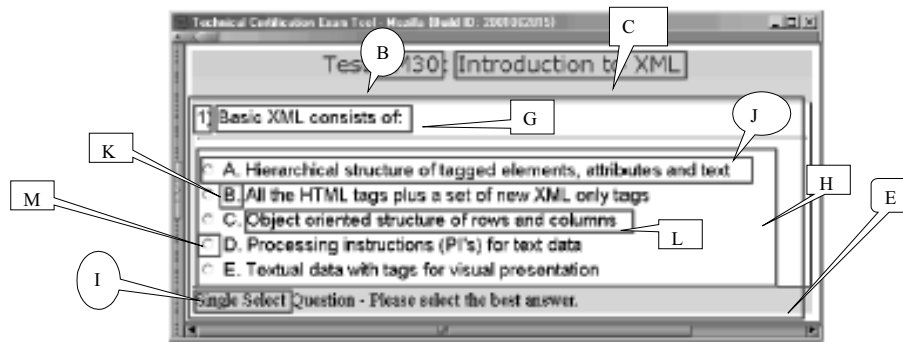
- ___ 1. In this last section of the exercise, you will test the well-formedness of your XML file by applying an XML parser to it. Internet Explorer 6.x is such a parser. Open your XML file in Internet Explorer 6.x. **Internet Explorer 6.x** contains a built-in XML parser. If you made any errors in your document, IE will not be shy about telling where the likely problem is located.
 - ___ a. Open Internet Explorer. In the *address bar*, type in the *full* URL to your new XML file: **file:///c:/xml/labs/Lab 1 - XMLBasics/exam.xml**.
Note: There are three slashes before the drive letter in this URL.
 - ___ b. This is another way: open Internet Explorer; use **Files->Open...->Browse...** Change the *Files of Type:* to *All Files* and work your way through the directory tree down to the "Lab 1 - XMLBasics/exam.xml" file.
 - ___ c. Here is a third way: select your XML file and use **Open With** and selecting **Internet Explorer 6.x** as you did in **Section 1**.
- ___ 2. Fix any errors found in your document. If you did it right the first time, try and force errors into your document to see how Internet Explorer 6 reacts to different types of errors.

When Internet Explorer 6 finds no errors and displays your XML document, then you have successfully built your first well-formed XML document.

[Note: We wrote this as a separate section so that you will have notes to refer to until you become comfortable with navigating between a text editor and a browser/well-formedness tester. It obviously repeats some of the information in Section 2!]

These are our approaches:

1. This shows how we identified the lettered items A through M.



2. Here is our solution to creating the XML schema. (We have dropped the comments only to save space.)

```
<?xml version="1.0"?>
<test id="XM30">
  <description>Introduction to XML</description>
  <testQuestions>
    <question id="Q1">
      <questionText>Basic XML consists of</questionText>
      <choices allowMultiple="No">
        <choice id="A">
          <choiceText>
            Hierarchical structure of tagged elements,
            attributes and text
          </choiceText>
          <correct>Yes</correct>
        </choice>
        <choice id="B">
          <choiceText>
            All the HTML tags plus a set of new XML only tags
          </choiceText>
          <correct>No</correct>
        </choice>
        . . . C., D., and E. omitted to save space..
      </choices>
    </question>
  </testQuestions>
</test>
<!-- End of File -->
```

END OF LAB

Exercise 2. Introduction to WebSphere Studio Application Developer

What This Exercise is About

In this exercise, you will be introduced to WebSphere Studio Application Developer (Application Developer), IBM's integrated development environment. This product is capable of developing several different environments using a single tool. This lab focuses on the XML features but we will only touch on a small fraction of the capabilities available. Nevertheless, it should give you the knowledge required to become immediately productive.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Navigate and configure the Application Developer environment
- Import XML resources into the Application Developer
- Edit XML documents using the Application Developer
- Validate XML documents using Application Developer

Required Materials

- IBM's WebSphere Studio Application Developer 5.x.
- The zip file, C:\XML\labs\XML_Intro.zip, containing the lab resources required to complete the XML Introductory exercises in this course
- The jar file, C:\XML\labs\XML Programming.jar, containing the lab resources required for the XML Programming exercises in this course.
- The files in C:\XML\XM301Lectures folder.
- The files in C:\XML\Startup Files.
- The DB2 XML Extender folder, C:\XML\DXX for the optional DXX lab.
- Sun J2SE v1.4.x or higher (with source)

Applicability

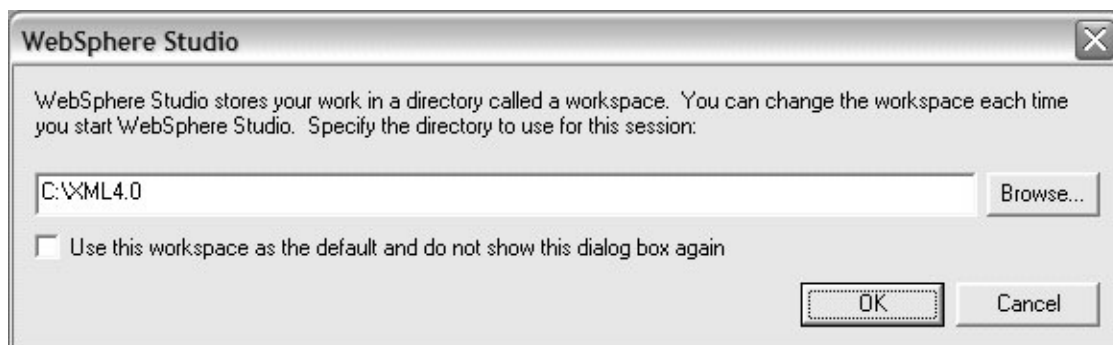
- XM301 Sections 1 and 5, 6, and 7
- XM321 Sections 1 through 4
- XM341 All Sections

Exercise Instructions

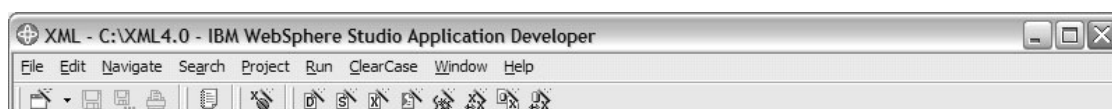
Section 1 - Navigating and Customizing the Application Developer

In this section of the exercise, you will learn how to navigate and customize the different perspectives within the Application Developer.

- ___ 1. Navigating Application Developer. Open the Application Developer environment. (**Start->Programs->IBM WebSphere Studio->Application Developer 5**)¹ You will be presented with the following dialog box asking you to specify the location you wish to use for your workspace, the storage area to use for your working files.



You may want to have workspaces for different projects or you may want to specify a location that is subject to regular backup by your IT infrastructure. In this particular case, click OK to accept this default. **Note:** An easy way to remember where you are storing your files is to modify **Properties Target:**² by appending **—showlocation**. The result is captured on the title bar:³



Note: If the location you select involves many characters, an alternate way is to begin to open another instance: the 1st screen is the same as the one above. Once you have used the information, simply select **Cancel**.

Warning: DO NOT check the Use this workspace. It defeats the purpose of this new feature. IF you accidentally turn off the select workspace feature you can reset it by

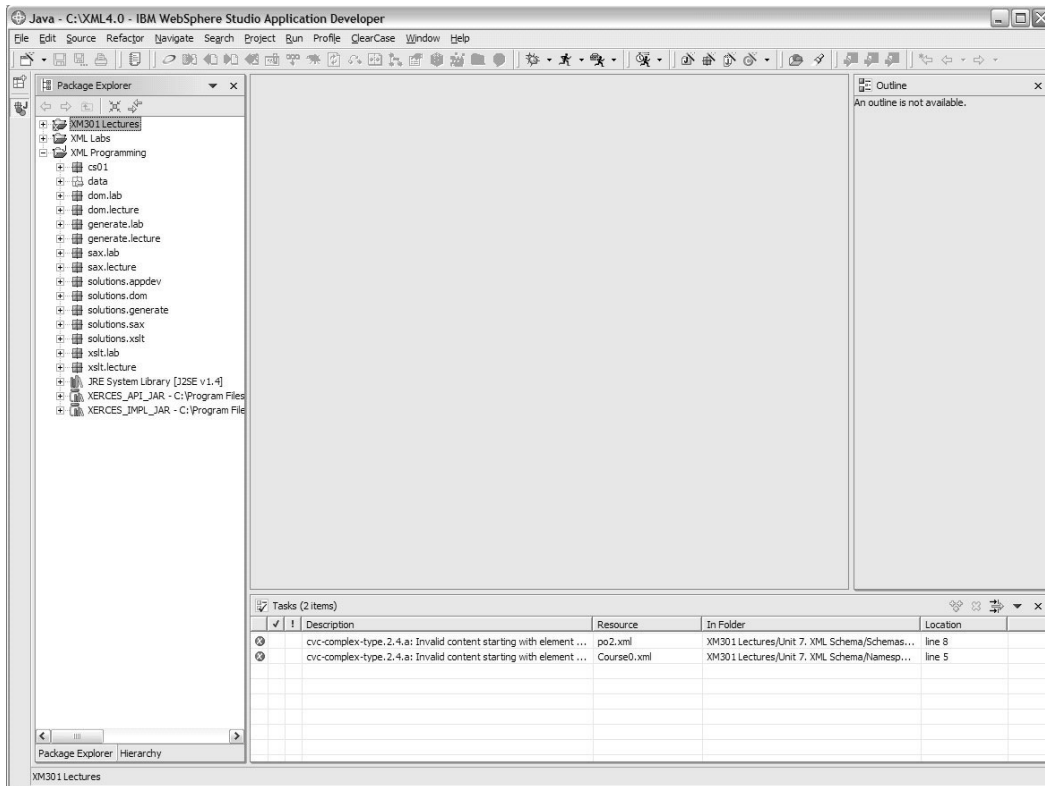
wsappdev.exe —setworkspace

in a command prompt (use Search to find the full pathname to wsappdev.exe). **Note:** you can accomplish the same purpose by appending the **—setworkspace** to the **Target:** in **Properties**.

¹ Before you select this, use the context (right-click) menu to Create Shortcut which you can place on your desktop or in your *tray* (or both).

² Hover over either the WSAD icon or the Application Developer 5.1 entry in the Start/Programs menu. Right-click and select Properties (at the bottom of the pop-up menu) to display useful information including the Target:.

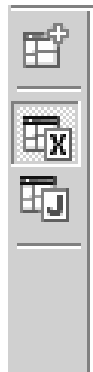
³ Version 5.0. x will put the pathname at the end of the line; the appearance of the icons is slightly different, too.



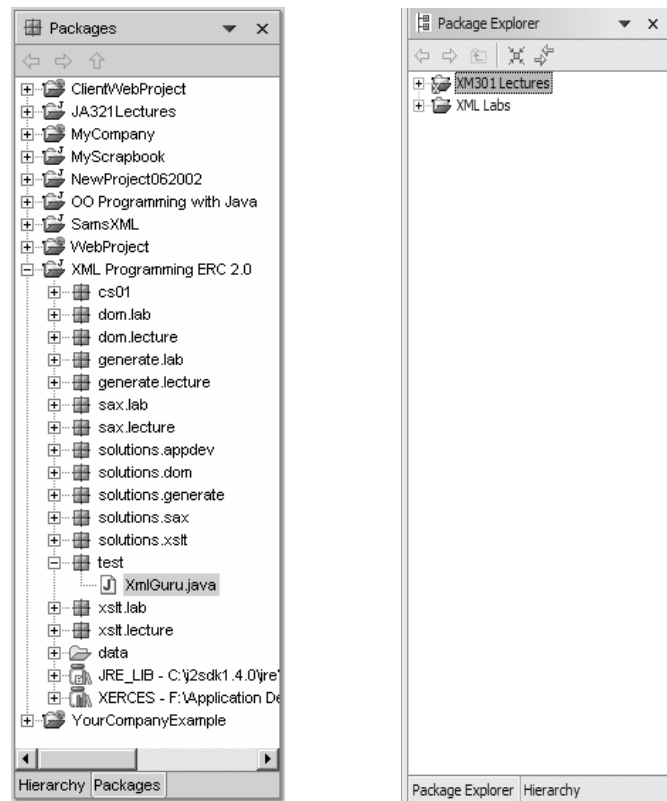
You will be presented with the basic Application Developer workspace as shown above. (The Java *perspective* is shown – observe Java immediately to the right of the *Studio* icon.)


Identify the following areas of the workbench.

- ___ a. Perspective bar (vertical bar at the far left of the workbench screen) showing the XML perspective selected:



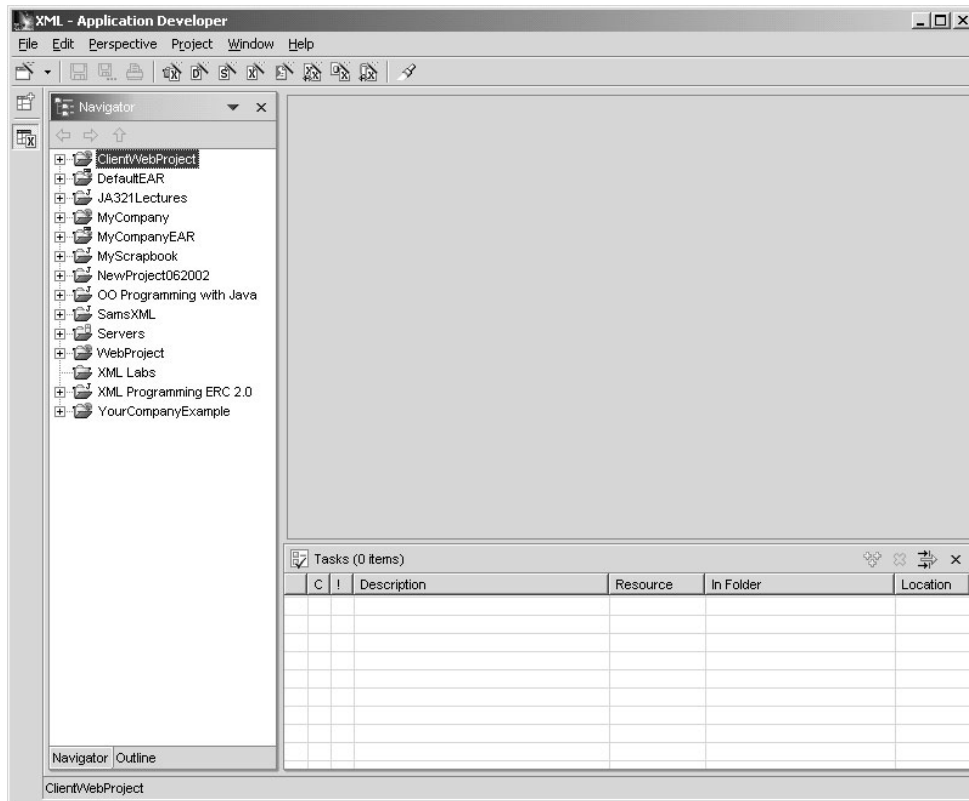
- ___ b. **Package Explorer** view (Tree view of all projects, packages, folders, and source files. Yours will be empty.) A version 5.0 screen capture is shown on the left; version 5.1.0 is on the right.



- ___ c. Source view (currently empty)
 - ___ d. Outline view (currently empty)
 - ___ e. Task List area (displaying two errors)
- ___ 2. **Customizing a Perspective.** Open the XML perspective:
- ___ a. If you don't currently have the XML perspective open (indicated with an X symbol in the perspective list), left-click the master perspective icon (top icon in the perspective list area, with a plus symbol (+)),  and select Other... from the pop-up menu.
 - ___ b. Select XML from the list on the resulting dialog and click OK.
- ___ 3. Customizing your current perspective. All of the views in each perspective may be reorganized to your liking. In this step, you will move the **Outline** view to another location. For the purpose of this lab, the new location is a tab next to the **Navigator** view.
- ___ a. Grab the title bar of the **Outline** view and drag it over to the **Navigator** view. As you move the view around, the dragging icon will change to one of the following: An *up-arrow* icon (places view above current view), a *down-arrow* icon (places view below current view), a *window*, or a *set-of-folders* icon (view will become a tab next to current view).

- ___ b. Drop the **Outline** view on top of the **Navigator** view when the icon shows as a *set-of-folders*.

Your workbench should now resemble the following:



- ___ c. You can now switch between **Navigator** and **Outline** by selecting the appropriate tab.
- ___ d. You can reset the perspective to the arrangement you started with by selecting '**Reset Perspective**' item in the **Window** menu. Do this now.

All views and Perspectives work in this fashion. Once you get accustomed to the Perspective(s) you use most frequently, you can use this technique to arrange the views to suit the way you work.

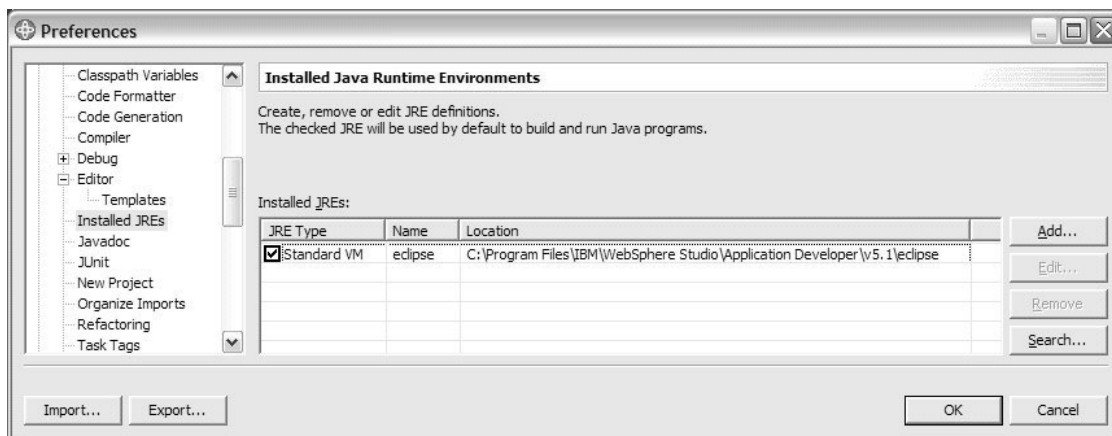
Section 2 - Configure Application Developer to use J2SE 1.4

J2SE version 1.4 contains XML features that are not available in the 1.3.1 SDK version included with Application Developer. Application Developer supports development with multiple versions of the J2SE, and you will use that capability in this lab to gain access to the enhanced features of the newer SDK.

Why is this important? Because the runtime environment determines the rules under which file imports will be compiled.

- ___ 1. Select the **Windows** menu and choose **Preferences** to open the Application Developer **Preferences** dialog.

- ___ 2. Use the navigation tree on the left side of the dialog to find the **Installed Java Runtime Environments (Installed JREs)** configuration as shown below.⁴

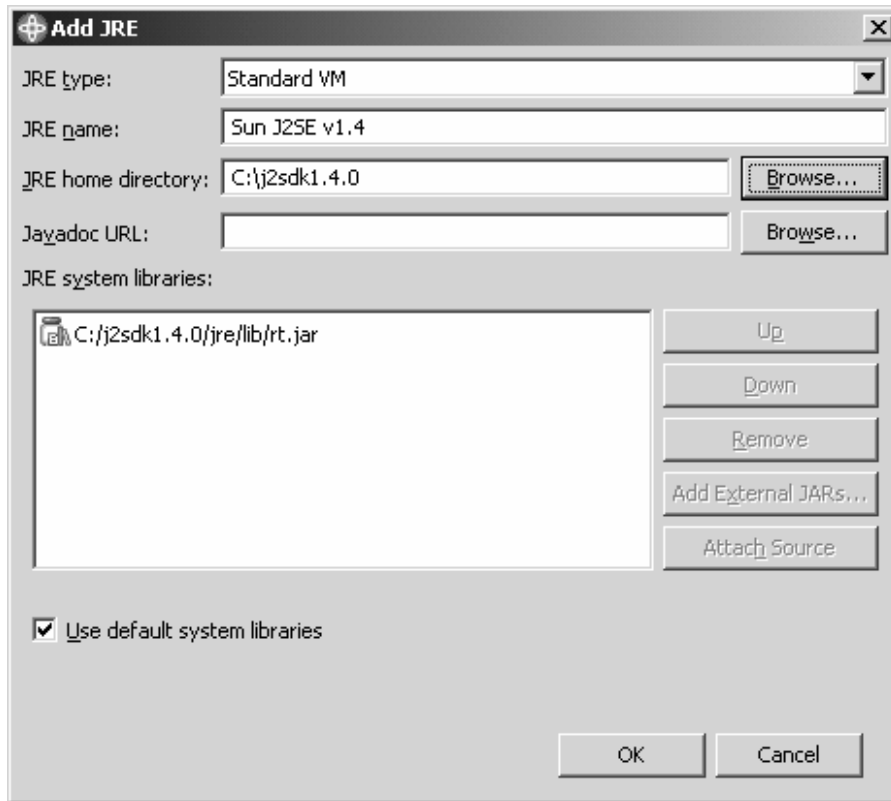


- ___ 3. Click **Add...** to display the **Add JRE** dialog.
- ___ 4. Type `sun_j2se_v1.4` for the **JRE name**, then click (**JRE home directory**) **Browse...** to locate the J2SE 1.4.x. Note that it has been installed in **C:\J2sdk1.4.x.**^{5 6}

⁴ The screen capture is for *Studio 5.1*. In the leftmost portion of the screen, Templates, which is where you can add your own auto-complete shortcuts to those already built-in, is now a subpart of Editor rather than a separate entry as it was in *Studio 5.0*. In the center portion of the screen *Studio 5.0* displays Detected VM. *Studio 5.0* does not have a Search button.

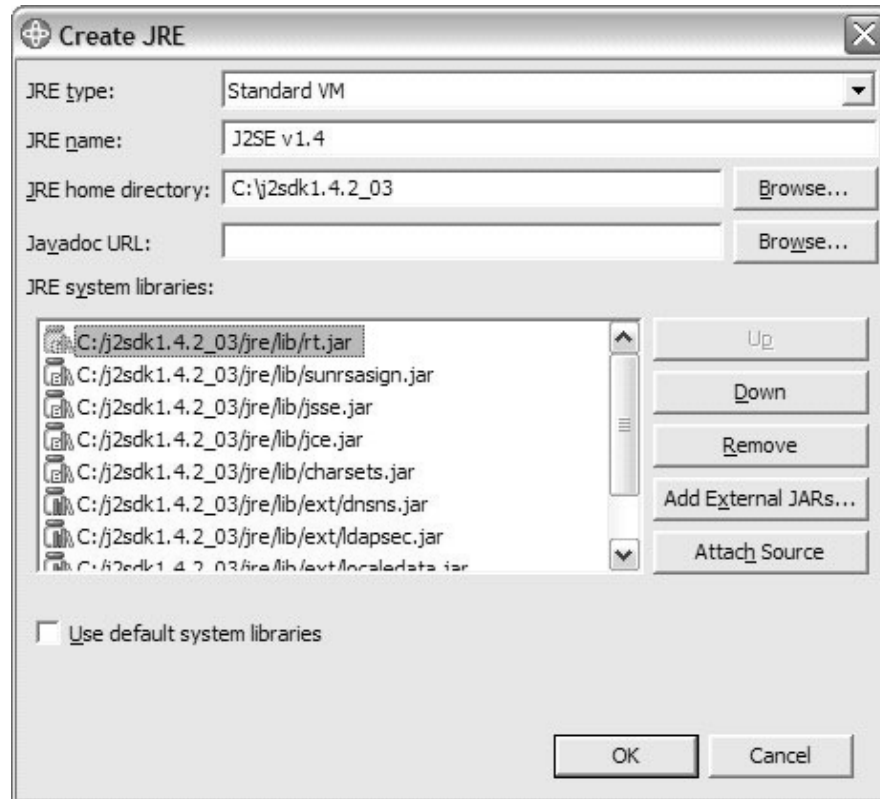
⁵ If there is only one choice, select it. If there is more than one choice, select that with the highest version number.

⁶ **Back at your work site you would use whatever version your organization has identified – provided it is at least version 1.3.1.** If you have a choice, use J2SEE version 1.4.2 or higher **if you will be creating java code** because 1.4.2+ supports hot code swapping. The URL for downloading the latest source was (as of January 20, 2004) <http://java.sun.com/j2se/1.4.2/download.html>; this is subject to change. The latest version at this writing was J2SE v 1.4.2_03; select the SDK file not the JRE file. DOWNLOAD the *Windows* version; Accept the License Agreement and Continue. You have a choice of whether to download a very large file for offline installation, or a much smaller file that requires a reasonable (broadband) connection. That applies at this point to the 32-bit version; the 64-bit version is near the bottom of this download list; **it** is smaller because at this point **it** does not **yet** have all the features of the 32-bit version – expect this to change with time. In the install, typically we do accept the defaults but we do not register either browser.

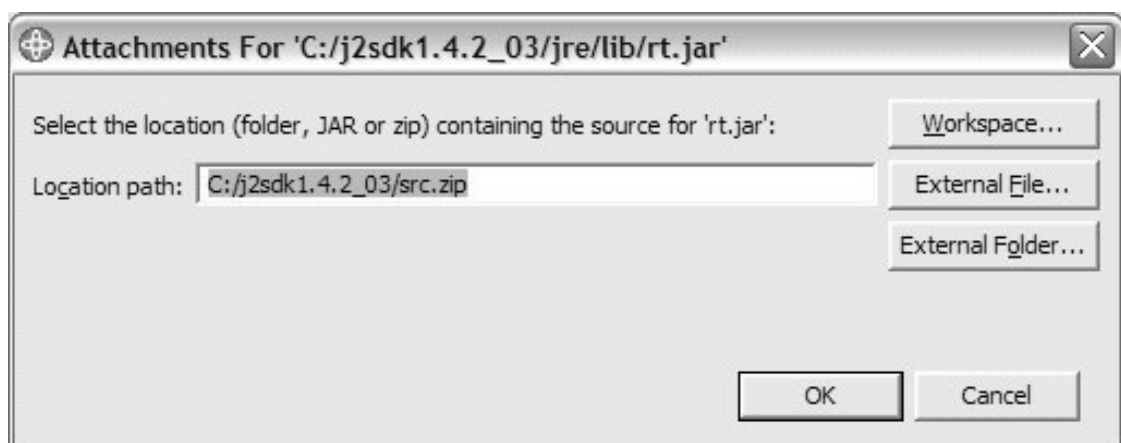


- ___ 5. Now you will make the source code for the Java classes in this JRE available to Application Developer. In doing this, you are making it possible to examine the source for classes in the JRE by opening them in the IDE and it will provide source level trace into the JRE during debugging. Refer to the screen shots, above.
- ___ a. Uncheck the **Use default library** item.

- ___ b. Click on C:/j2sdk1.4.0/jre/lib/rt.jar to make it active with this result:⁷



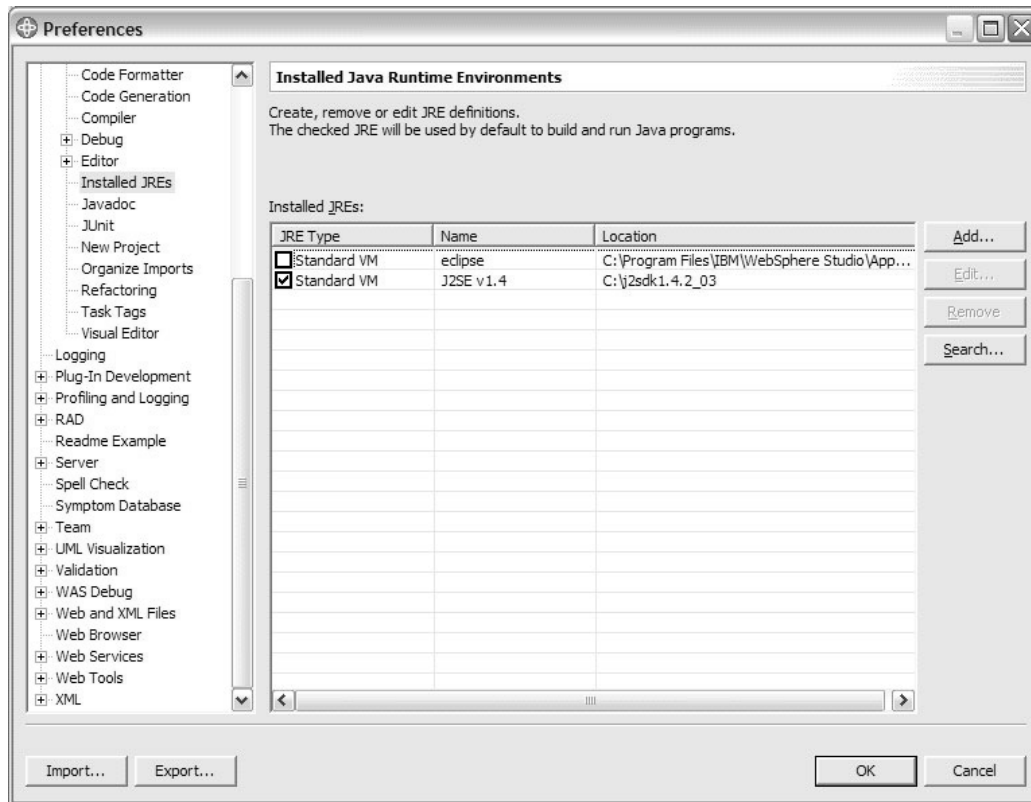
- ___ c. Click **Attach Source** to display the source **Attachments For** [the JRE rt.jar file] shown in the following screen capture.
- ___ d. *Studio* will automatically find the source zip file. Confirm that it does contain the path name of the src.zip file as shown here:



- ___ e. Click **OK** to close the Attachments window. (*Studio 5.0* has a slightly different look but the result is the same.)
- ___ 6. Click **OK** as necessary to return to the **Installed Java Runtime Environments** in the **Preferences** windows.

⁷ The 1st screen capture is *Studio v 5.0*; the one immediately above is v 5.1.

- ___ 7. We are going to use the most up-to-date J2SE 1.4.x for the labs in this course. Make it the default JRE by selecting the check box⁸ beside it in the list of available JREs. The **Installed Java Runtime** Environments dialog should now look like the following:



- ___ 8. Click **OK** on the **Preferences** dialog. The new JRE is now configured and will be used by default when we create Java projects in Application Developer.

Section 3 - Create a New Java Project

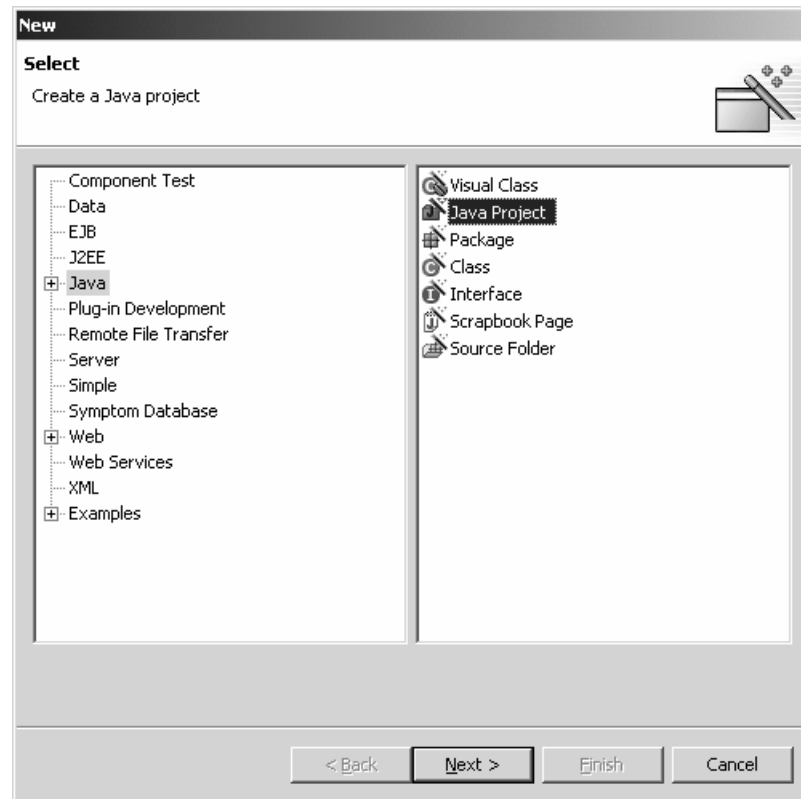
In this section, you will create a new project to hold the material for the Java programming exercises in the course.

- ___ 1. Click the New wizard in the upper left corner of the workspace.

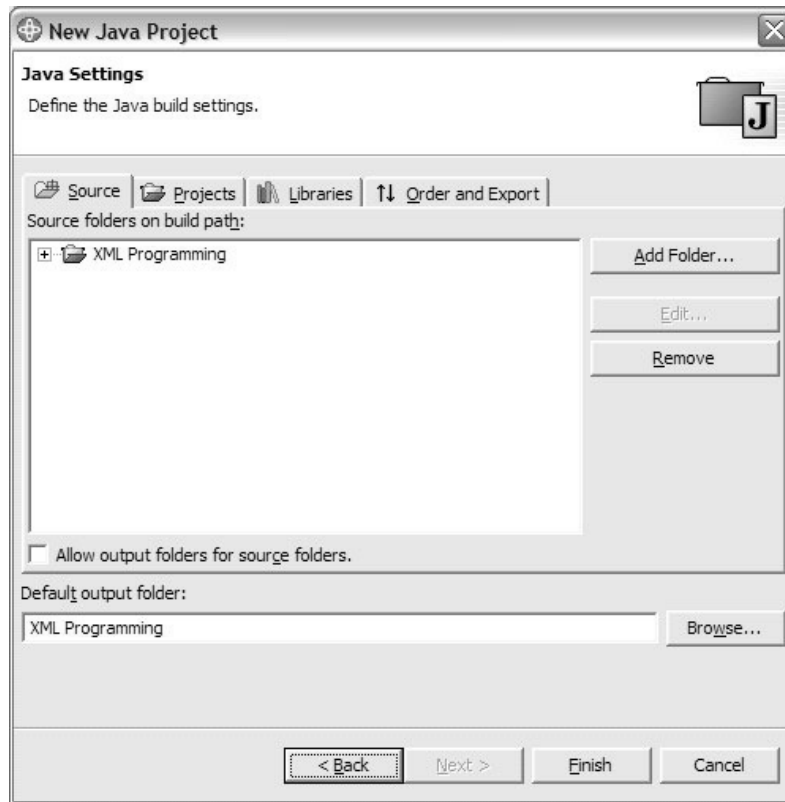


⁸ This checkbox acts like a radio button!

- ___ 2. For the project type, select **Java**, and then **Java Project**.



- ___ 3. Click **Next** to open an intermediate window to enter a project name: in the Project Name field, enter **XML Programming**. Make sure **Use default location** is selected, and click **Next** (Do not click Finish) to open the **Java Settings** window for a **New Java Project**.

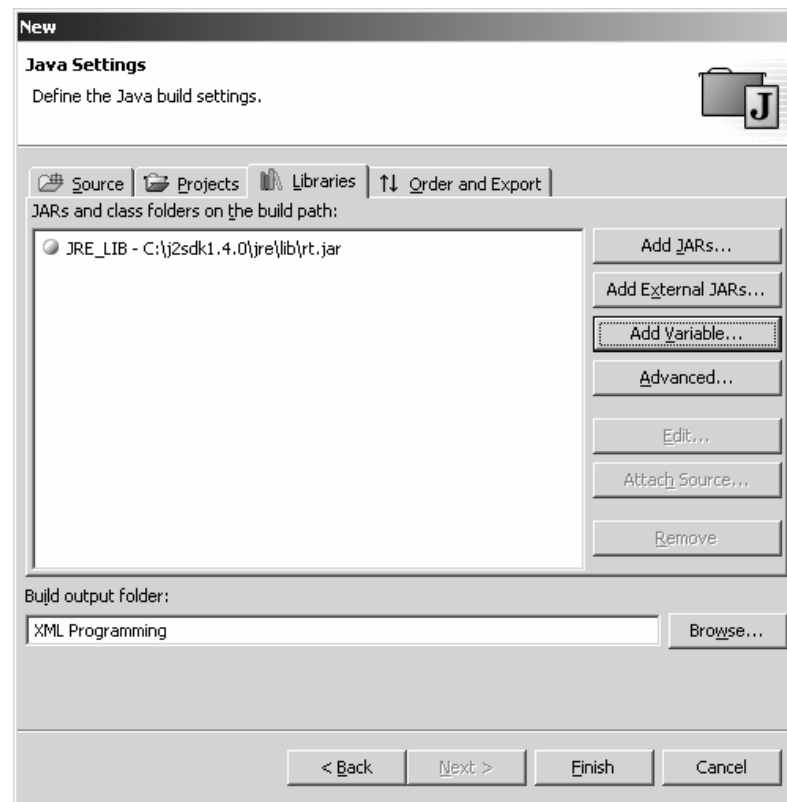


- ___ 4. This screen is where you specify the source directory for the project, references to other projects, what libraries to include, their order on the classpath, and so on. [In v5.0 the center pane is blank.]

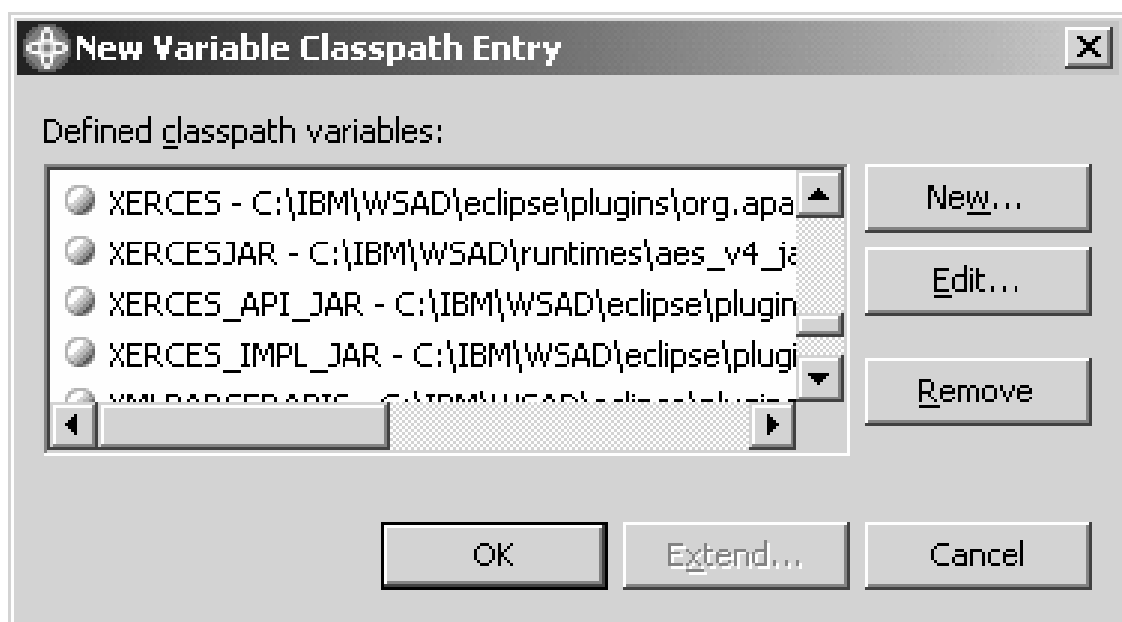
For this project, we will need to add Xerces to the project classpath.

The Xerces implementation is packaged in two jar files, xercesImpl.jar and xmlParserAPIs.jar. Due to their frequent use, Application Developer has predefined classpath variables for these jars, named **XERCES_IMPL_JAR** and **XERCES_API_JAR** respectively. You will now add those classpath variables to your project classpath.

- ___ a. Select the Libraries tab. The following window opens (note the reference to the j2sdk1.4.0 JRE (or whatever version we chose earlier) which we just configured as the default for new projects):

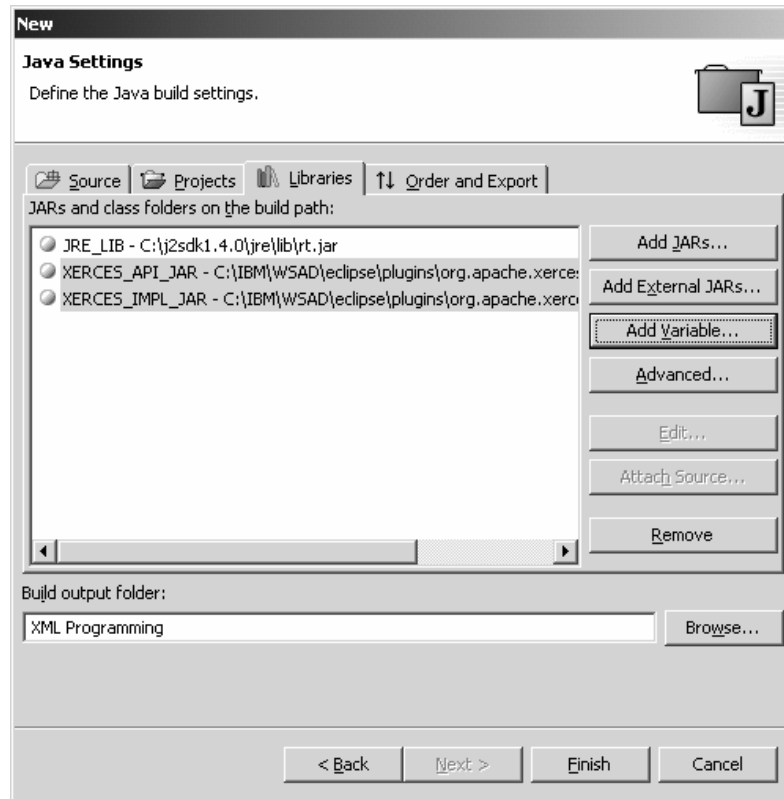


- ___ b. Click **Add Variable...**
- ___ c. Scroll down and locate the two Xerces variables in the list of defined classpath variables and select them (hold down the **<Ctrl>** key to make a multiple, not necessarily sequential, selection).



- ___ d. Click **OK**.

- ___ e. Your Java Settings - Libraries screen should now look like the following:



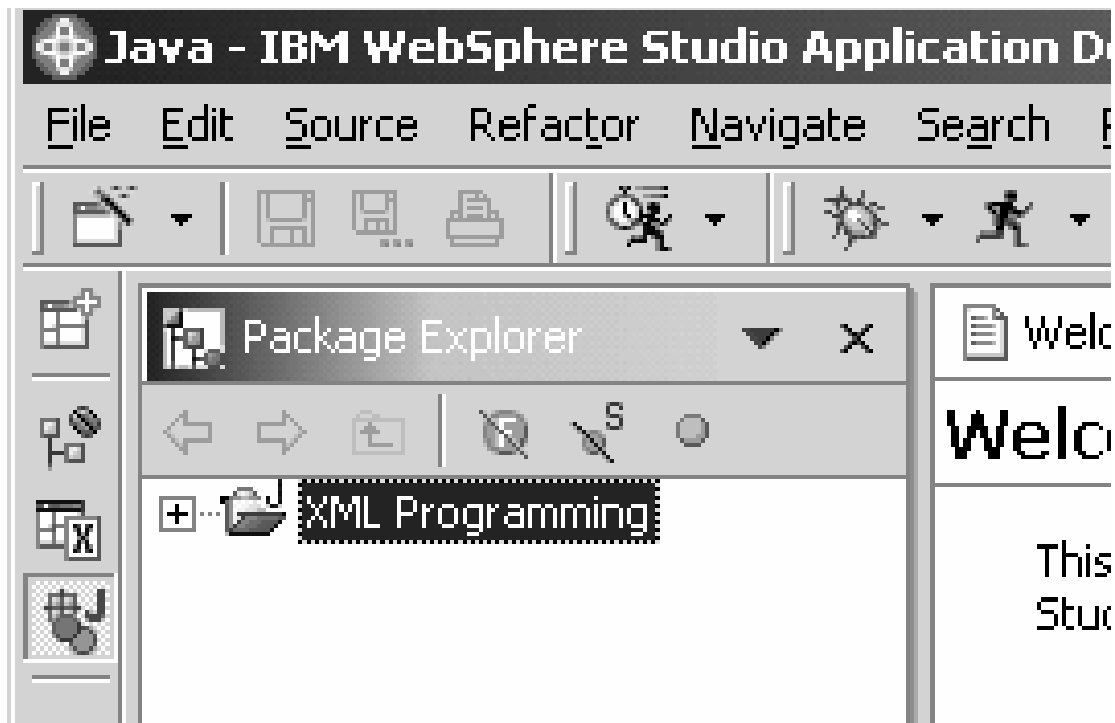
- ___ f. Click **Finish** to create the XML Programming project. Some time will elapse as Studio checks for errors.
- ___ g. Note that Application Developer will open the Java Perspective to present the Java project you just created. If errors were found, they would appear in the Tasks pane to the right of the Package Explorer (or whatever leftmost pane you had open).
- ___ h. The Package Explorer view should now show your XML Programming project. Expand it to show the classpath references: JRE_LIB, XERCES_API_JAR and XERCES_IMPL_JAR.

While we have created the project and established the environment, we still have to import the files we shall use. We do that now.

Section 4 - Import External Files

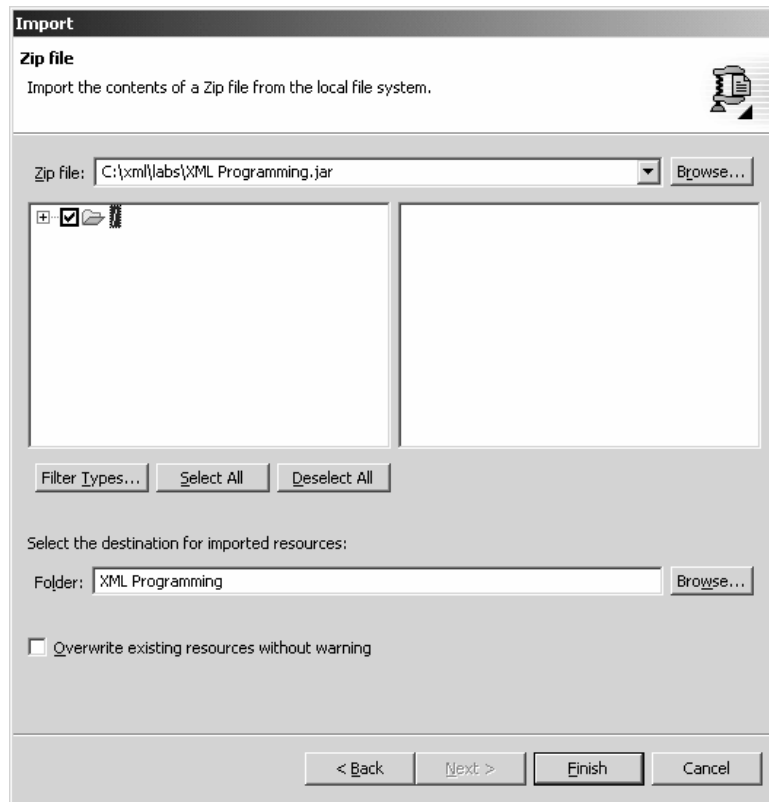
In this section of the lab, you will import the Java source and XML document files used in the Java Programming section of the course. The course files have been supplied in the form of a Java JAR file.

- ___ 1. Make sure you are in the **Java Perspective**, and that you have the **XML Programming** project selected. (This is the upper left-hand corner of how your screen should appear.)

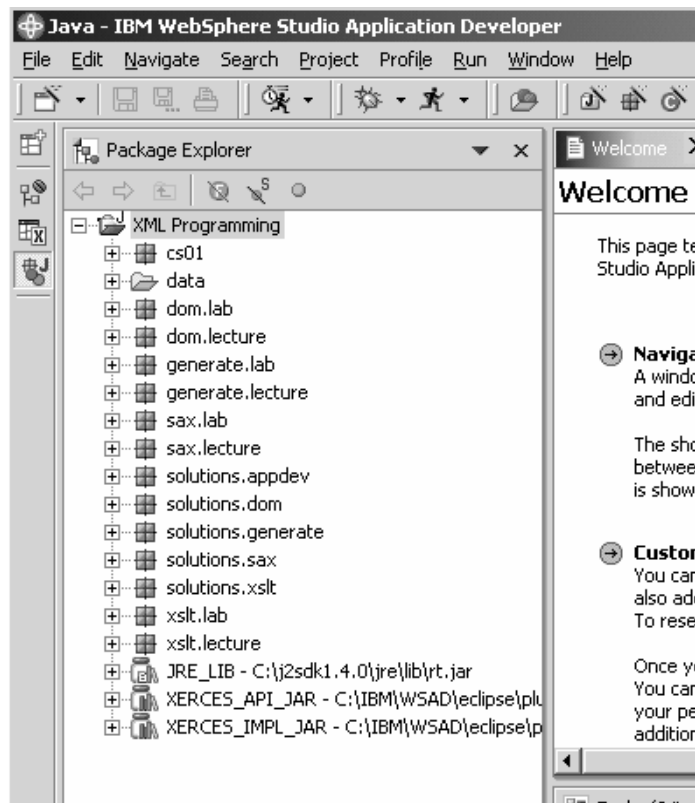


- ___ 2. Select **File -> Import...** to bring up the import dialog.
- ___ 3. Scroll down and select Zip file (Zip and Jar are considered the same type) and click **Next** to bring up the Zip File import screen.
- ___ 4. Click **Browse** next to the Zip file field:
 - ___ a. Locate and select Zip File: C:\XML\labs\XML_Programming.jar.⁹ Click **Open** to open the next, **Import** window. [Note: XML may be lower-case or a mix of cases.]
 - ___ b. We will be importing all the files in the jar, but go ahead and expand folder list to view its contents. Select a folder to view its content as well. When you are finished ensure that the check box is black and white (not gray, which indicates not all subfolders/files are selected) with a check inside the box.
 - ___ c. Make sure that the Folder field contains XML Programming.

⁹ In some images the file is in C:\xml\labs\; if you experience difficulty use Search to find **XML Programming.jar**.

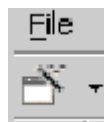


- ___ d. Click **Finish** to import the XML Programming JAR file into your project. The result should now include a list of 14 packages and one data folder like this:



Section 5 - Create a New, Non-java Project

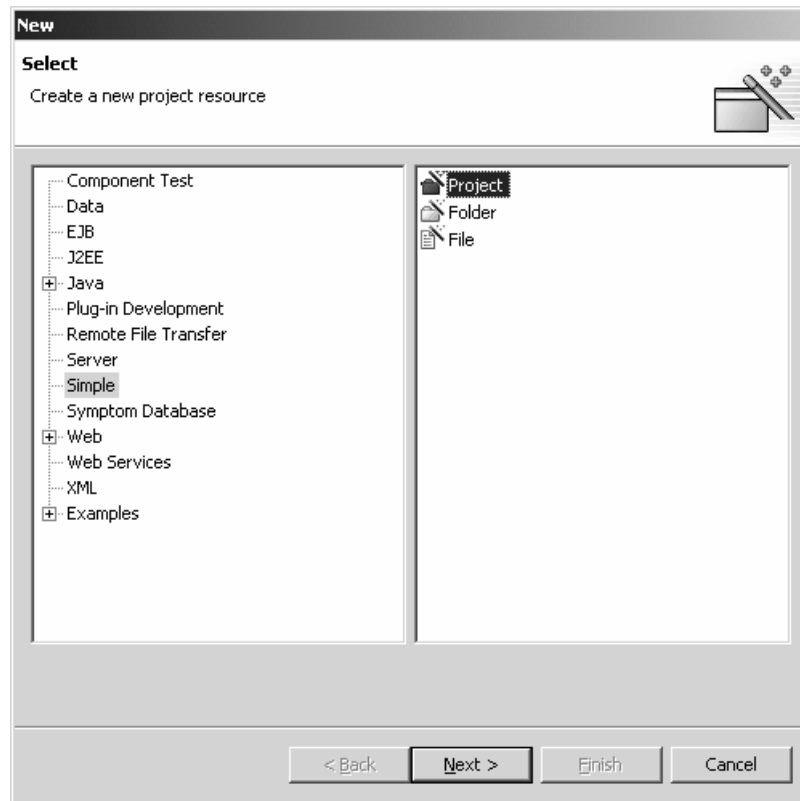
In this section, you will create the project you will use for the non-Java, Introductory Lab exercises. This project will not make use of the Java features in Application Developer so it is not necessary to create it as a Java project.



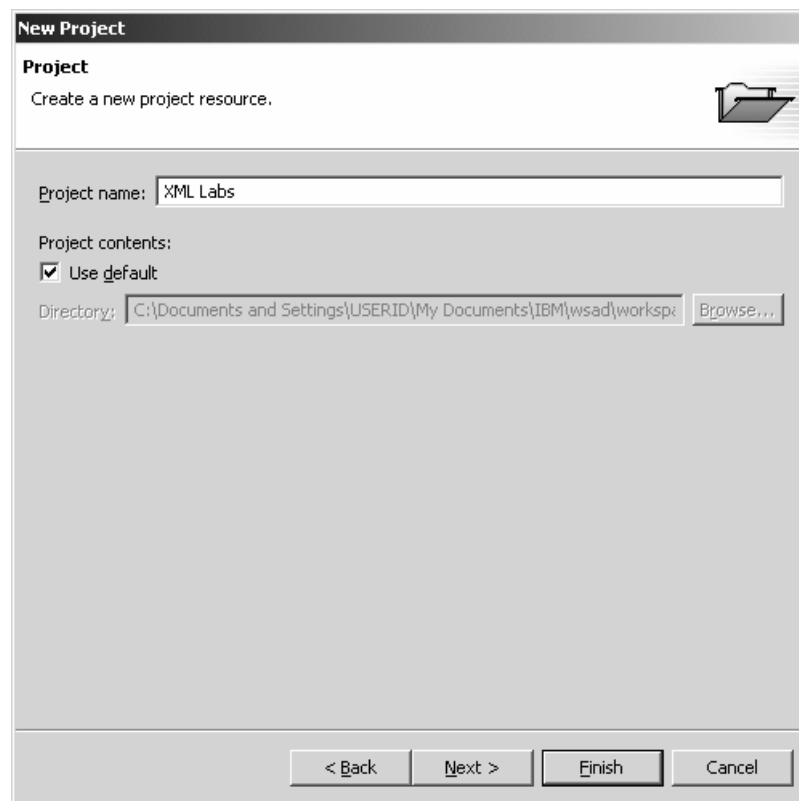
- ___ 1. Click the **New wizard** icon in the upper left corner of the workspace below **File** in order to open the **New** window, shown below.

FYI: You can accomplish the same result via **File New->Project**; you can also bring up a context-sensitive menu from within the package pane and then again use **File New->Project**.

- ___ a. For the project type, select **Simple**, and then **Project**.



- ___ 2. Click **Next** to open the **New Project** window. In the **Project Name:** field, enter **XML Labs**. Make sure **Use default location** is selected, and click **Finish**.¹⁰



- ___ 3. Application Developer will create the new project, open a **Resource** perspective (the default perspective for Simple projects) and select the newly created Project. Since the work in this project will be XML related, open an XML perspective: to return to the XML perspective, simply click on the **XML** perspective icon on the lefthand side of Application developer.
- ___ 4. A set of resource files have also been supplied for this project, this time they are in a ZIP file. Import them now.

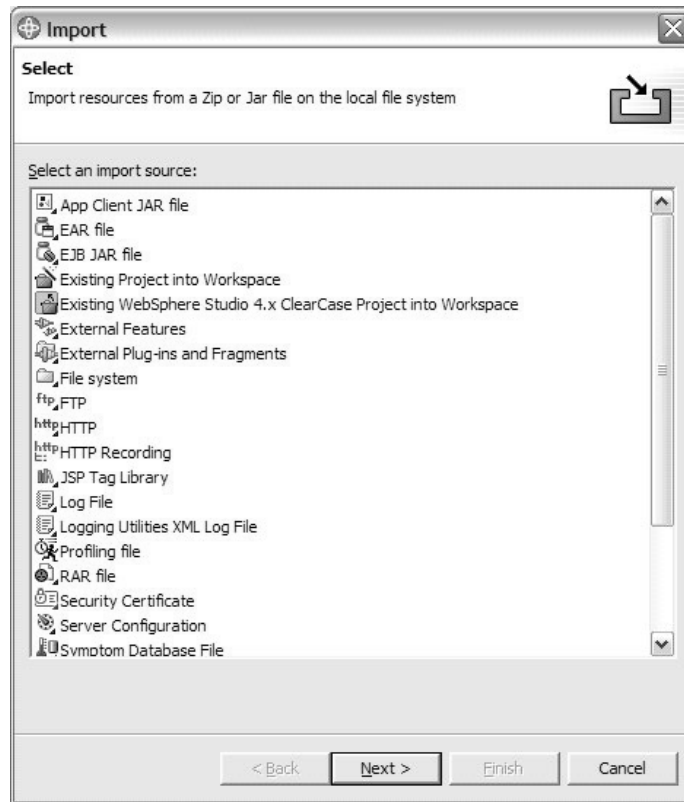
Either (1) follow the same steps you followed in Section 4 but this time, the file you are looking for is: C:\xml\labs\XML_Intro.zip. Import all the files and don't forget to make sure that the Project Folder field contains XML Labs; or (2) follow the steps in **Section 6**, below.

Section 6 - Import External Files

In this section of the lab, you will import the XML and other resource files used throughout the rest of the course. The course files have been supplied in the form of a ZIP file. *[Should you wish to practice at your office, these are the files you will wish to capture on a floppy.]*

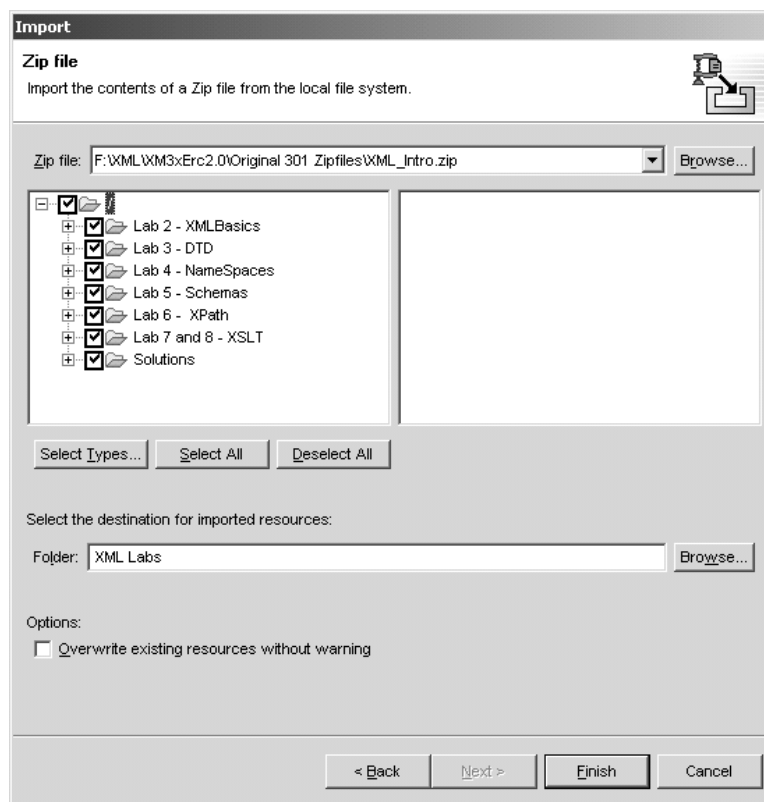
- ___ 1. Make sure you are in the **XML Perspective**, and that you have the **XML Labs** project selected.
- ___ 2. Select **File -> Import...** to bring up the import dialog:¹¹

¹⁰ Beginning with ERC4.0 our default pathname is considerably shorter than the one shown above.

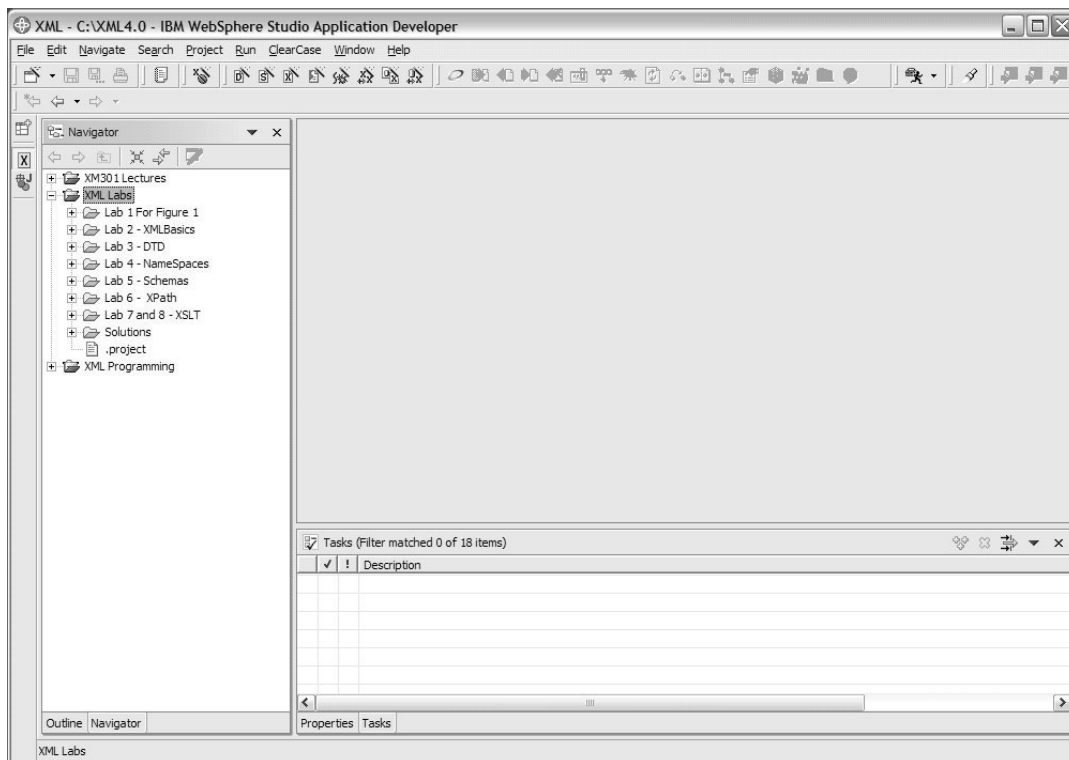


- ___ 3. Select Zip file (Zip and Jar are considered the same type) and click **Next** to bring up the Zip File import screen.

¹¹ You could context-select Import as well. This is a 5.1 screen capture: 5.0 only lists eleven choices compared to 5.1's 24 import source choices.



- ___ 4. Click **Browse** next to the Zip file field:
 - ___ a. Locate and select Zip File: xxx\XML_Intro.zip. (Yours will *probably* be at C:\xml\labs\ but you may have to **Find/Search** for it.) **Open** it.
 - ___ b. You will import all the files in this ZIP file, but go ahead and *expand* (*click on the + sign in the check box*) the folder list to view its contents. Select a folder name to view its content as well.
 - ___ c. Make sure that the **Folder:** field contains XML Labs.
 - ___ d. Click **Finish** to import the XML_Intro.zip file into your project. With the XML perspective selected and the XML Labs project expanded, your workspace should now look like this (the left pane may be split if you reset your perspective):



- ___ 5. **New with ERC4.0** is an **XM301 Lectures** folder that contains sample files to accompany points made in the lectures. This folder is contained in the same directory as the previous folders; this material is a **File system** source. Using what you have learned so far, import the contents into an **XM301 Lectures** folder that you will create.
- ___ 6. There is also a folder, **Startup Files**, that contains html snippets you will use in Labs 7 and 8. You may import them or you may find it easier to open them externally and cut and paste as necessary.

Section 7 - Editing XML Files with Application Developer

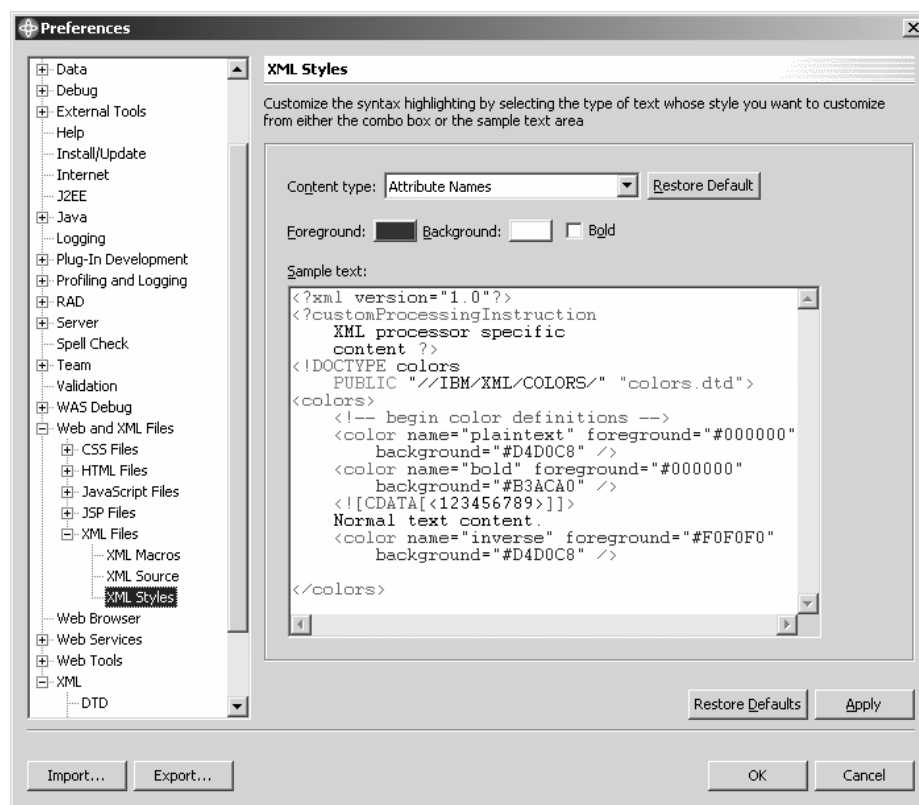
In this section, you will explore the XML Editor in WebSphere Studio Application Developer. To do this, you will use a document we have provided that is similar to the XML document you created in Lab 1.

- ___ 1. Browse the Lab 2 - XML Basics directory of the XML Labs project. In it, you will find a file called **exam.xml**. This document is similar to the one you created in Lab 1. Open it by double clicking on it.
- ___ 2. Notice that Application Developer uses its XML Editor to open XML files. Similar associations exist for other file types and can be modified using the Workspace preferences. Right-click on **exam.xml** and observe the **open with** possibilities. Notice **Validate XML file** near the bottom of the pop-up menu. Try it.

- ___ 3. XML Editor Source View. The XML Editor has two views, **Source** and **Design**. You can select either view by clicking the appropriate tab. Select the **Source** view if it isn't already selected.
- ___ 4. The Source view is a color-coded, syntax-aware XML editor. The various parts of your XML document are color coded to make it easier to read. This color coding may be customized to suit each developer's preference.
 - ___ a. Select the **Window** menu and choose **Preferences** to open the Application Developer **Preferences** dialog.


Hint: if you prefer single-click over double-click to open files, you can do it in the 1st Preferences pane to open.

- ___ b. Then use the navigation tree on the left side of the dialog to find the **XML Style** configuration as shown below:¹²



- ___ 5. **Content type:** lists the various parts of an XML document that you can change color preferences (background, foreground, bold) for. Adjust your color preferences now. As you change them you will see the sample document reflect your chosen preferences.
- Select OK or Apply when you want to apply the preference changes in the workspace. Select OK or Cancel to exit the preferences Dialog.
- ___ 6. The color coding is applied in real time as you edit an XML document.

¹² Studio 5.0 is shown; Studio 5.1 adds an XML Annotations entry that extends our control of how Studio reports.

- ___ a. Turn the <description> element into an XML comment by placing comment delimiters before and after it.
Hint: you can also use <Ctrl><spacebar> which will popup a window in which you can select a comment template! This is *auto-completion*.
- ___ b. Notice that the color of the description element changes immediately.
- ___ c. Remove the comment delimiters to undo your changes.
- ___ 7. The editor can also reformat XML documents to make them easier to read.
 - ___ a. Place the text cursor at the end of an element and press the Delete key (do this on a number of lines).
 - ___ b. Right-mouse click the document and select **Format->Document**. The editor reformats XML according to the settings in the XML Preferences (currently the default settings).
- ___ 8. Application Developer will also let you know when a document is not well-formed.
 - ___ a. Remove the opening quote (") character from the value of the ID attribute on one of the <question> elements.
 - ___ b. Save the document (by typing CTRL-s). Application Developer will place an entry in the Tasks list indicating that there is a problem with your document.
 - ___ c. Double-click the message in the Tasks list to place the text cursor in the location of the error and correct it. Save the corrected document.
- ___ 9. Application developer can check your document for errors without saving it if you select the Validate the current state of the XML File item on the XML menu, a tool-bar icon is also provided for this purpose. 
- ___ 10. XML Editor Design View. The Design view of the XML editor presents XML documents in a vertically split view with two sections. On the left, the document is shown as a set of nested, expandable tree nodes. The value of each item is displayed on the right as the nodes are expanded and shrunk.
 - ___ a. Browse the document by expanding (+) and collapsing (-) various nodes.
 - ___ b. Element values and attribute values can be edited by selecting the current value and typing a new value. Change the value of the **allowMultiple** attribute to No (exclude the quote characters). Now look at the attribute value in the Source view and note that the editor has placed quotes around the new attribute value. The editor is aware of the well-formedness rules in XML and will use correct syntax according to the context of what you are doing.

END OF LAB

For additional practice, select **Help -> Help Contents -> Search: Tutorial -> Go**.

Exercise 3. DTD

What This Exercise Is About

In this exercise, you'll expand on the previous Online Test XML document to add validation for its contents using the Checkpoint questions for this unit and the notes in the supplied schema. You'll do this in phases:

1. Compare the schema and prewritten, well-formed exam.xml prepared for this lab with those for lab 1; identify the constraints implied by the published questions in the Checkpoint charts to identify all the elements and attributes that make up the XML document and create a DTD based on this information.
2. Determine the constraints associated with each element or attribute and reflect those constraints in an internal DTD and test your work.
3. Move the DTD statements into an external file and test your work.
4. Generate a DTD file from our XML file.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Analyze an XML document and extract DTD components
- Create a DTD based on existing XML documents
- Validate an XML document by using a validating parser
- Define a DTD as an internal subset or as an external subset

Required Materials

- Well formed XML document, **exam.xml**
- WebSphere Studio Application Developer V5.1

Applicability

- XML301
- XML341

HINT: If you are having difficulty identifying icons and such (because of their size) drop the screen resolution down a notch to make them appear larger. . . .but going below 800 x 600 is not recommended.

IMPORTANT: If you wish to do these exercises "on your own" ignore sections that begin with phrases such as "Here's how..." or "Here's how we did it..." If you experience difficulty working independently of our hints, you can always peruse our suggestions.

Exercise Instructions

You will use three approaches to building your DTD:

1. Define the DTD as an internal subset; a DTD within an XML document.
2. Extract the DTD, place it in a file by itself and reference it as an external subset.
3. Use *Studio* to build a DTD from our XML file.

Section 1 - Identify the implied DTD requirements

A. Background

The Checkpoint questions we wish to model are presented here:

1. Which DTD entry correctly depicts phone number, with optional area code?
 - a. `<!ELEMENT phone ((areaCode)*, prefix, body)>`
 - b. `<!ELEMENT phone (areaCode?, prefix, body)>`
 - c. `<!ELEMENT phone? (areaCode, prefix, body)>`
 - d. `<!ELEMENT phone (areaCode, (prefix, body)+)>`
2. Which of the following is a not a limitation of DTD?
 - a. Non-XML syntax.
 - b. Does not allow range of values (that is, 5 to 10 elements).
 - c. Does not provide proper typing of values (that is, integer versus string).
 - d. Does not permit Parameter Entity references.
 - e. All of the above.

To save time, stop after the second question.

- ___ 1. Open Application Developer to the XML perspective and close any files you may have open from previous labs.
- ___ 2. Open the file **exam.xml** found in the **Lab 3 - DTD** folder of the **XML Labs** project. Select the Source view of the XML editor.
- ___ 3. Observe the schema and notes that begin the file:

```
<!-- On-Line Exam, DTD Exercise -->
```

```
<!--
```

```
Listed below are all the elements [and attributes] in this XML
document. The rules for each are also included. Use this
information to create an in-line DTD.
```

```
test [ id ]           (Root Element)
  description
  testQuestions
    question [ id ]: One or more
      questionText: Required
```

```
choices [ allowMultiple ]: allowMultiple must be "Yes" or "No"
  choice [ id ]: One or more
    choiceText
    correct: Must be "Yes" or "No"
```

Notes:

- The "allowMultiple" attribute has a "Yes" or "No" value.
- The choice's "id" attribute has a character value.
- The "correct" element has a "Yes" or "No" value.

The ordering of "choiceText" and "correct" may vary

-->

___ 4. Compare the preceding schema with that from Lab 1:

Basic XML Exercise

Create a "well-formed" XML file using the element and attribute name listed below. Elements requiring Attributes are shown as element[attr]

```
test [ id ]          (Root Element)
  description
  testQuestions
    question [ id ]
      questionText
      choices [ allowMultiple ]
        choice [ id ]
          choiceText
          correct
```

Notes:

- The "allowMultiple" attribute has a "Yes" or "No" value.
- The choice's "id" attribute has a character value.
- The "correct" element has a "Yes" or "No" value.

As you can see, they are virtually identical.

Let's compare the implementation of the first question from each lab by selecting analogous parts of the XML code. We have adjusted the indentations to make comparison simpler. We have put them on facing pages, where possible.

This for Lab 3:

```
<test id="XML30-DTD">
  <description>Document Type Definition</description>
  <testQuestions>
    <question id="Q1">
      <questionText>
        Which DTD entry correctly depicts a phone number, with
        optional area code
      </questionText>
      <choices allowMultiple="No">
        <choice id="A">
          <choiceText>
            <![CDATA[<!ELEMENT phone ( (areaCode)*, prefix, body ) >]]>
          </choiceText>
          <correct>No</correct>
        </choice>
        <choice id="B">
          <choiceText>
            <![CDATA[<!ELEMENT phone ( areaCode?, prefix, body ) >]]>
          </choiceText>
          <correct>Yes</correct>
        </choice>
        <choice id="C">
          <choiceText>
            <![CDATA[<!ELEMENT phone? ( areaCode, prefix, body ) >]]>
          </choiceText>
          <correct>No</correct>
        </choice>
        <choice id="D">
          <choiceText>
            <![CDATA[<!ELEMENT phone ( areaCode, (prefix, body)+ ) >]]>
          </choiceText>
          <correct>Yes</correct>
        </choice>
      </choices>
    </question>
    . . .
  </testQuestions>
</test>
```

___ 5. Compare the preceding XML code with that from Lab 1:

This for Lab 1 [actually, it was copied from the exam.xml file we did for Lab 2 and reformatted to make the comparison easier]:

```
<test id="XML30">
  <description>Intro to XML</description>
  <testQuestions>
    <question id="Q1">
      <questionText>
        Basic XML consists of
      </questionText>
      <choices allowMultiple="No">
        <choice id="A">
          <correct>Yes</correct>
          <choiceText>
            Hierarchical structure of tagged elements, attributes and text
          </choiceText>
        </choice>
        <choice id="B">
          <correct>No</correct>
          <choiceText>
            All the HTML tags plus a set of new XML only tags
          </choiceText>
        </choice>
        <choice id="C">
          <correct>No</correct>
          <choiceText>
            Object oriented structure of rows and columns
          </choiceText>
        </choice>
        <choice id="D">
          <correct>No</correct>
          <choiceText>
            Processing instructions (PI's) for text data
          </choiceText>
        </choice>
        <choice id="E">
          <correct>No</correct>
          <choiceText>
            Textual data with tags for visual presentation
          </choiceText>
        </choice>
      </choices>
    </question>
  </testQuestions>
</test>
```

As you can see, with the exception of the inclusion of "`![CDATA[. . .]]`" in question 1, they are virtually identical.

- ___ 6. Remove the "`![CDATA[. . .]]`" from any of the choices in Q1 in the `exam.xml` you opened in Step 2, above (leave the `<!ELEMENT...>` declaration). What happens?

Answer: The parser tries to use the result as part of the schema rather than a (dumb) value, so you are told that "the content of elements must consist of well-formed character data or markup." This, of course, is what the CDATA does: it tells the parser to ignore the characters it doesn't like.

Put the "`![CDATA[. . .]]`" back in and save; you should now again have no errors in the Task window.

Note: We have not commented on the presence or absence of the XML version declaration that starts an XML file. At this writing, it is optional if the default UTF is used. It is, however, a "best practice" to include it in anticipation of the introduction of a new version, which will make the inclusion mandatory.

- ___ 7. We can see from either the schema or the `.xml` files that there are nine elements and four attributes:

Elements:	Attributes:	Constraints:
test	[id]	(Root Element)
description		
testQuestions		
question	[id]	One or more
questionText		Required
choices	[allowMultiple]	allowMultiple must be "Yes or "No"
choice	[id]	One or more
choiceText		
correct		: Must be "Yes" or "No"

The table above was a straight cut and paste construct from the comments in the original file. Let's convert these into DTD elements and attlists.

- ___ 8. Before we continue, **close** the `exam.xml` file.

B. Approaches

I. Just Do It!

If you wish to do this on your own, skip directly to **Section 2** (below).

II. Create a Paper and Pencil Solution and then type in the Results!

Remember the coding sheets from years passed? Preformatted sheets that we filled in from our flow charts or pseudo-code and sent off to be punched or, later, typed in by ourselves. It will simplify later steps enormously if you will complete the tables below at this

point. The preformatted items are already there. We will need two: one for elements; one for attributes.

- ___ 1. The template for an element tag is **<!ELEMENT elementName (content model)>**, if the element may include a text value remember to insert **#PCDATA** as a child in the content model. Complete this table in these notes to capture all the elements identified above.

<Element tag	Element Name	(Content) >
<!ELEMENT	test	(description, testQuestions) >
<!ELEMENT	description	(#PCDATA) >
<!ELEMENT	testQuestions	(question+) >
<!ELEMENT	question	() >
<!ELEMENT	questionText	() >
<!ELEMENT	choices	() >
<!ELEMENT	choice	((choiceText,correct) (correct,choiceText)) >
<!ELEMENT	choiceText	() >
<!ELEMENT	correct	(#PCDATA) >

Note: There are no provisions in DTD to enforce the yes/no limitation on the correct element.

- ___ 2. A template for an attribute tag is **<!ATTLIST elementName attName attType defDecl >**.

The next step is to define the attribute rules to implement the constraints on the attributes identified above. Refer to Figures in the lecture to refresh your memory on the implementation of attributes. Complete this table to capture the attribute definitions:

<Attribute tag	Element name	Attribute name	Attribute type	Default declaration	>
<!ATTLIST	test	id	CDATA	#REQUIRED	>
<!ATTLIST	question	id			>
<!ATTLIST	choices	allowMultiple			>
<!ATTLIST	choice	id			>

Note: The only interesting attribute is allowMultiple.

Section 2 - Embed the implied DTD requirements in an XML file

DOCTYPE The DOCTYPE statement tells the parser where to look for the DTD grammar to apply to determine if your (well-formed - a prerequisite) document is *valid*. It may point to internal or external or public DTD statements:


```
<!ELEMENT questionText ( #PCDATA )>
<!ELEMENT choices ( choice+ )>
<!ELEMENT choice ((choiceText,correct) | (correct,choiceText))>
<!ELEMENT choiceText ( #PCDATA )>
<!ELEMENT correct ( #PCDATA )>
<!ATTLIST test id CDATA #REQUIRED>
<!ATTLIST question id CDATA #REQUIRED>
<!ATTLIST choices allowMultiple (Yes|No) "No">
<!ATTLIST choice id CDATA #REQUIRED>
]>
<!-- End of DTD here -->

<test id="XML30-DTD">
  <description>Document Type Definition</description>
  <testQuestions>
    <question id="Q1">
      <questionText>
        Which DTD entry correctly depicts a phone number, with
        optional areas code
      </questionText>
      <choices allowMultiple="No">
        <choice id="A">
          <choiceText>
            <![CDATA[<!ELEMENT phone ( (areaCode)*, prefix, body )>]]>
          </choiceText>
          <correct>No</correct>
        </choice>
        <choice id="B">
          <choiceText>
            <![CDATA[<!ELEMENT phone ( areaCode?, prefix, body )>]]>
          </choiceText>
          . . .
        </choice>
      </choices>
    </question>
  </testQuestions>
```

Section 3 - Process the DTD statements as a separate file

What other XML files might we want to apply these DTD statements to in order to provide **valid** (in addition to **well-formed**) XML files? Let's pursue this thought by making our DTD file external to the XML file.

___ 1. Create another copy of the **exam.xml** file and rename it to **exam_ext.xml**.

- ___ 2. Highlight/select **Lab 3 - DTD**. Select the **Create a new DTD file** icon from the collection of icons below the menu bar. (It's the square box with a **D** in it and the magician's **wand** on top.) Select **Create a new file from scratch**, change the file name to **test.dtd**, and **Finish** to place the result in the same Lab 3 - DTD folder.
- ___ 3. Open the *test.dtd* file and select **Source** if not already selected. Note that Application Developer automatically adds the appropriate declaration statement best practices requires.
- ___ 4. Copy the DTD statements (only the **!ELEMENT** and **!ATTLIST** statements) from *exam_int.xml* into the new *test.dtd* file. That completes the *test.dtd* file.
- ___ 5. In the *exam_ext.xml* file immediately below the `xml version="1.0"`, use `<ctrl><space bar>` and select `<!DOCTYPE`.
- ___ 6. Modify the `<!DOCTYPE test PUBLIC. . .>` statement by replacing the **PUBLIC. . .>** with **SYSTEM "test.dtd" >**.
- ___ 7. There are several ways you can test your results: 1) click the *exam_ext.xml* file in the editor; click **XML->Validate** the current file; 2) open the *exam_ext.xml* file using the browser editor (**Note:** *You must close any editors that have the file already open for this to work*); 3) find the file on your system and use Internet Explorer to open the file via the **File => open** menu (*caution: you might have to adjust the DOCTYPE declaration path to test.dtd for this to work*).

Section 4 - Use Application Developer to Generate a DTD from XML

Application Developer has the capability to analyze an XML instance and generate a DTD, which describes it. In this section you'll generate a DTD and compare it to your hand written version.

- ___ 1. Select the original **exam.xml** file.
- ___ 2. Right-click and select **Generate' DTD...** file¹.
- ___ 3. In the **Create DTD** dialog, change the **File name:** to **generated.dtd**. Click **Finish**.
- ___ 4. A new file called **generated.dtd** now exists in the **Lab 3 - DTD** folder of your project. Open it and examine the contents.
- ___ 5. The generated file does not contain all the constraints that you were told to include in your DTD, for example, **allowMultiple** is of type **CDATA** not (**Yes|No**).

Note: to associate *generated.dtd* with *exam.xml* select **exam.xml**; right-click and select **Assign->DTD...** from the context-sensitive, drop-down menu; since the DTD file is on the same path as the XML file, browse for *generated.dtd* in the **System ID:** field and select **OK**. Open the **exam.xml** file and observe that there is now a **DOCTYPE** statement making the association between these two files. Test the validation; *exam.xml* is now also a valid file.

¹ You could also again use the icon described in Section 3., Step 2. In one case the suggested name is *exam.dtd*, in the other it is *NewDTD.dtd*; you will type your own name choice so it does not matter.

END OF LAB

Exercise 4. XML Namespaces

What This Exercise Is About

In this exercise, you are going to look at how namespaces can be used in your own applications. You are also going to be sure that you understand the rules for namespace prefixing and defaulting by identifying which portions of an XML document are in which namespace. Finally, you are going to create an XML document that contains elements from two different namespaces.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Describe three situations from your own business or applications where namespaces should be used
- Correctly tell which elements in a document belong to which namespace
- Create a document that uses elements and attributes from two different namespaces

Required Materials

- Websphere Studio Application Developer version 4/5
- Namespace lab resource files

Applicability

- XML301
- XML341

NOTICE: If you are having difficulty identifying icons and such (because of their size) drop the screen resolution down a notch to make them appear larger.

WARNING: The purpose of this exercise is to give you an opportunity to experiment with the various ways of using the namespace concept. This could easily become an all-day adventure. Try to limit yourself to about 20 minutes on Section 1 and about one hour on Section 2.

Exercise Instructions

Section 1 - Associate items with Namespaces.

___ 1. Here are the names of three Namespaces. For each Namespace, can you suggest additional elements that would belong in it and not in the other two?

___ a. `http://www.somecompany.com/farming`

Possible Answer: `farmingType(rancher | farmer) equipmentCategory, crops, livestock, silo, subsidy, farmhouse`

___ b. `http://www.somecompany.com/plants`

Possible Answer: `commonName, species, roots, habitat (indoor/outdoor), lifecycle (annual, perennial)`

___ c. `http://www.somecompany.com/gardening`

Possible Answer: `gardenKind (flower | vegetable | combination), statuary, bed shape, snail bait, planter box`

___ 2. In which namespaces would you find 'soil_type'? (You can create an additional namespace if you feel it is necessary.) You should be prepared to defend your decision in discussion with the class.

Answer: [There is no right answer for Question 2. It is intended to get you thinking about balancing the idea of maintaining namespace distinctness against the difficulty of doing so.]

___ 3. Have you encountered namespace issues inside your own organization? Can you think of three?

Section 2 - Namespace scope

In this section you will experiment with namespace scope by inserting and removing Namespace declarations in a grammar-constrained document. Application Developer will let you know (with error messages) if you are using namespaces incorrectly.

Note: The document used in this lab references an XML Schema for its grammar. Feel free to look at the XML Schema (found in the file **test.xsd**), but do not change it in any way.

___ 1. Open the document **ns.xml** located in the **Lab 4 - Namespaces** folder of your **XML Labs** project and look at the **Source** view.

___ 2. Examine the structure of the file. It should be familiar to you by now. Note that each element name is qualified by a namespace prefix. The *grammar restriction*¹ requires that each of the elements present belong to the **`http://www.ibm.com/ILS/WDO3`**

¹ "grammar restriction" refers to the .xsd file we shall be using to validate the .xml file. In Exercise 4 you will construct an XSD (Xml Schema Definition) file to replace the less rigorous, non- well-formed DTD (Document Type Definition) file we've used previously.

namespace. The namespace declaration at the top of the document associates this namespace with the prefix '**ns1**'.

Warning: WDO3 is three letters, w..., d..., and o..., capitalized, + the number 3.

- ___ 3. Remove the **ns1** prefix from the **description** and **testQuestions** elements then save the file (remember to remove the prefix from both the **start** and **end** tags for the elements you change). Your Tasks list should now indicate that there are errors in your XML document.
- ___ 4. Create a default namespace declaration for the required namespace on the **description** element and save your document (refer to your course notes and to the namespace declaration at the top of the **ns.xml** file for the information you need to do this). You should no longer see an error message referring to the **description** element.

The beginning of our solution looks like this:

```
<?xml version="1.0" encoding="US-ASCII"?>
<ns1:test id="XM30_Namespaces" xmlns:ns1="http://www.ibm.com/ILS/WDO3"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/ILS/WDO3 test.xsd ">

    <!-- Please do not change the xsi namespace declaration
         or the schemaLocation attribute -->

    <description xmlns="http://www.ibm.com/ILS/WDO3">XML Namespaces
Test</description>
    <testQuestions>
        <ns1:question id="Q1">
            <ns1:questionText>Which are true about XML
Namespaces</ns1:questionText>
            <ns1:choices allowMultiple="Yes">
                <ns1:choice id="A">
                    . . .
```

- ___ 5. **There are still errors involving testQuestions.** You can eliminate all remaining error messages by moving the default namespace declaration **xmlns="http://www.../WDO3"**. Move it now and save your file to confirm that the errors have been corrected. Remember that a default namespace applies to all unqualified, nested elements. This should help you determine the new location for the default declaration.

Here is how we solved it:

```
<?xml version="1.0" encoding="US-ASCII"?>
<ns1:test id="XM30_Namespaces" xmlns:ns1="http://www.ibm.com/ILS/WDO3"
    xmlns="http://www.ibm.com/ILS/WDO3"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

```
xsi:schemaLocation="http://www.ibm.com/ILS/WDO3 test.xsd ">
```

```
<!-- Please do not change the xsi namespace declaration  
or the schemaLocation attribute -->
```

```
<description >XML Namespaces Test</description>
```

It was necessary to move it where it would apply to both the *description* element and the *testQuestions* element. One (not recommended) approach would be to add the declaration to both *description* and *testQuestions* but this is not very elegant; it did, however, work:

```
<description xmlns="http://www.ibm.com/ILS/WDO3">XML Namespaces  
Test</description>  
  <testQuestions xmlns="http://www.ibm.com/ILS/WDO3">
```

It can not be on a line by itself. The solution immediately above is not really appropriate, so the only remaining choice is to insert it into the *ns1:test* declaration...which is what we did!

- ___ 6. Namespace prefixes can be redefined at any time. The new definition remains in scope for the element where the declaration occurs and all of its children. Edit the **choice** element with **id='A'**, redefine the **ns1** prefix to represent the namespace: **http://www.ibm.com/ILS/general** and save the file. You'll see validation errors appear in your Tasks list.
- ___ 7. The **choice** element and its children no longer belong to the Namespace required by the document grammar. Notice that the redefinition has not had an effect on the rest of the document. The change to **ns1** only applies inside the element you changed.
- ___ 8. Edit the **choice** element again to fix the problems. Add a declaration for a second namespace prefix, **ns2** that will represent the namespace required for the document grammar.
- ___ 9. Change the QNames inside the element to use the correct prefix. Save your file.
- ___ 10. At this point, your document should be valid again. *Here is our solution:*

```
<?xml version="1.0" encoding="US-ASCII"?>  
<ns1:test id="XM30_Namespaces"  
  xmlns:ns1="http://www.ibm.com/ILS/WDO3"  
  xmlns="http://www.ibm.com/ILS/WDO3"  
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
  xsi:schemaLocation="http://www.ibm.com/ILS/WDO3 test.xsd ">
```

```
<!-- Please do not change the xsi namespace declaration  
or the schemaLocation attribute -->
```

```
<description>XML Namespaces Test</description>
```

```
<ns1:testQuestions>
  <ns1:question id="Q1">
    <ns1:questionText>
      Which are true about XML Namespaces
    </ns1:questionText>
    <ns1:choices allowMultiple="Yes">
      <ns2:choice id="A"
        xmlns:ns1="http://www.ibm.com/ILS/general"
        xmlns:ns2="http://www.ibm.com/ILS/WDO3">
        <ns2:choiceText>
          XML Namespaces are associated with an XML Schema
        </ns2:choiceText>
        <ns2:correct>false</ns2:correct>
      </ns2:choice>
      <ns1:choice id="B">
        . . .
```

END OF LAB

Exercise 5. XML Schema

What This Exercise Is About

In this exercise, you are going to practice using Application Developer to capitalize on its abilities to create, associate, manipulate, and translate DTD, XML Schema, and XML documents.

In Part 1 you'll experiment with the capabilities of Studio to create an XML Schema from an existing XML document. This is similar to what we demonstrated in the associated lecture. You will also have a chance to create an XML Schema from an existing DTD file based on earlier work.

In Part 2 you'll create an empty XML schema file and, using information from earlier exercises, populate the schema incrementally, and check your work periodically using Application Developer to validate your XML Schema document.

In Part 3 you'll experiment with associating XML Schema and XML instances.

What You Should Be Able to Do

At the end of the lab, you should be able to use the power of Studio 5.x to assist your efforts to:

- Generate an XML Schema from an existing DTD
- Generate a DTD from an existing XML Schema
- Create an XML instance from an existing DTD
- Create an XML instance from an existing XML Schema
- Create an XML Schema to validate XML instances using an existing XML instance, a DTD file, and known business rules.
- Associate XML instances with XML schemas.

Required Materials

- **test.dtd**, the Document Type Definition (DTD) for tests
- An XML document, **test.xml**, that conforms to test.dtd
- Websphere Studio Application Developer version 5.x

Applicability

- XML301
- XML341

Have you discovered <Ctrl><s> yet? It's the fastest way to save your work!

Exercise Instructions

Section 0 - The Ingredients we shall Deploy

Here are the **test.dtd** items we shall be implementing in an XML schema.

```
<!ELEMENT test ( description, testQuestions ) >
<!ELEMENT description ( #PCDATA )>
<!ELEMENT testQuestions ( question+ )>
<!ELEMENT question ( questionText, choices )>
<!ELEMENT questionText ( #PCDATA )>
<!ELEMENT choices ( choice+ )>
<!ELEMENT choice ( (choiceText,correct) | (correct,choiceText) )>
<!ELEMENT choiceText ( #PCDATA )>
<!ELEMENT correct ( #PCDATA )>

<!ATTLIST test      id CDATA #REQUIRED>
<!ATTLIST question id CDATA #REQUIRED>
<!ATTLIST choices   allowMultiple (Yes|No) "No" >
<!ATTLIST choice    id CDATA #REQUIRED>
```

We shall use additional information which we could not implement because of constraints inherent to the DTD model. These were captured in the comment that preceded the items above in the test.dtd file; there could also be "business rules" such as a specification of the number of choices that could be allowed, the number of questions per test, *etc*:

```
<!--
The rules for documents that match this DTD are:

test [ id ]
description
testQuestions
question [ id ]: One or more
questionText: Required
choices [ allowMultiple ]: [ allowMultiple must be "Yes" or "No" ]
choice [ id ]: One or more
choiceText
correct: Must be "Yes" or "No"

Note: The ordering of "choiceText" and "correct" may vary
-->
```

Part 1. Using Studio to "Move" between .xml, .dtd, and .xsd Files

In this part you will use Application Developer to generate a Schema that contains all the semantics captured in either the DTD with which we began this exercise or the XML instance.

You will also see how to use Application Developer to generate an XML instance from either a DTD file or an XML Schema.

We saw in the lecture notes for this topic that we could generate an XSD corresponding to a well-formed XML instance because XSD is a language. Well, so is DTD. Arguably, less complex and therefore less expressive, but a language nonetheless.

Section 1 - Generate an XML Schema from a DTD

In this section we ask that you use Studio to generate an XSD file corresponding to the DTD file, test.dtd.

Here's how:

- ___ 1. Right-Click the **test.dtd** file in the **Lab 5 - Schemas** folder in your project and select **Generate > XML Schema...** from the context menu.
- ___ 2. In the Generate XML Schema dialog box, ensure that **Lab 5 - Schemas** is selected in the folder list and set the filename to generated.xsd. Click Finish. *Studio* will generate a schema from the information contained in your XML and place the new Schema file into the **Lab 5 - Schemas** folder.
- ___ 3. Open the **generated.xsd** file and select the Source view.
- ___ 4. Review the content of the generated schema. You'll see that the generated file is rather simplistic. On the other hand, notice how it handles "documentation."
- ___ 5. Look at the declaration that was generated for <correct>. The generator used **xsd:string** for the type because the Yes or No constraint isn't modeled in the DTD, it is only implied by the documentation in the DTD. The generator can quickly reproduce any DTD related semantics in a schema allowing you to avoid the task of manual translation that you just completed. This is a great way to convert existing DTDs to schemas where you can add constraints that a DTD can't express.
- ___ 6. If you are curious, you can alter the schema reference in **test.xml** to point to the generated schema and revalidate the document. You'll find that your document is invalid. The namespace information is not part of a DTD and is not present in the generated schema. If you correct the **xsd:schema** element of the generated file to match the **xsd:schema** element in your own schema your document will validate properly.

Section 2 - Generate an XML Schema from an XML instance

Starting with the test.xml file, use *Studio* to generate a 2nd .xsd file from test.xml.

Here's how:

- ___ 1. Right-Click the **test.xml** file in the **Lab 5 - Schemas** folder in your project and select **Generate > XML Schema...** from the context menu.
- ___ 2. In the Generate XML Schema dialog box, ensure that **Lab 5 - Schemas** is selected in the folder list and set the filename to **XMLgenerated.xsd**. Click Finish.
Application Developer will generate a schema from the information contained in your XML and place the new Schema file into the **Lab 5 - Schemas** folder.
- ___ 3. Open the **XMLgenerated.xsd** file and select the Source view.
- ___ 4. Compare the contents of the two generated schema:
 - ___ a. Select either the XMLgenerated.xsd or the generated.xsd file;
 - ___ b. Using the Ctrl key, select the other of the two files;
 - ___ c. Right-click and select **Compare with > Each other** to see these two files laid out side by side.

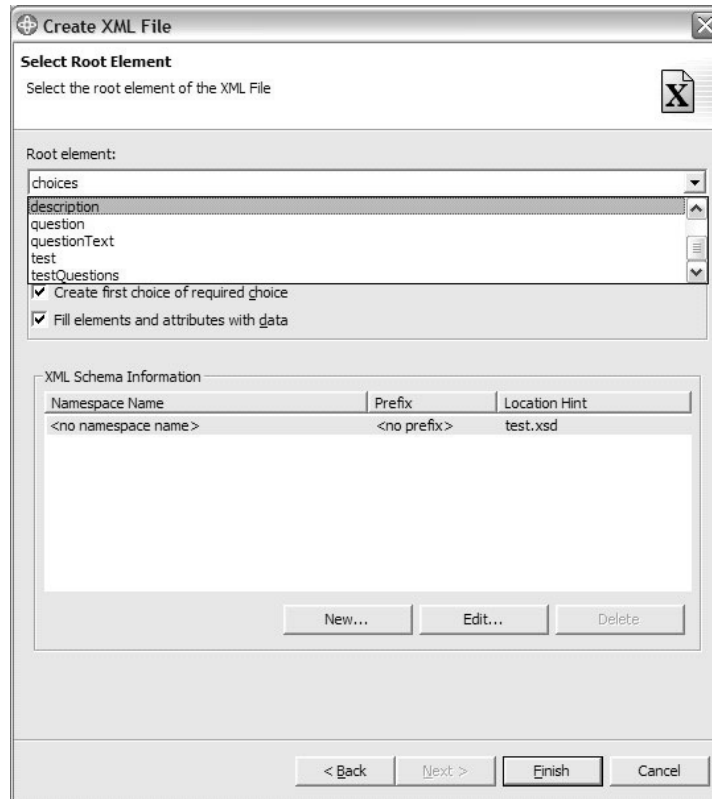
Reflection: Might this automated use of Studio (or any tool with similar capabilities) help your organization with the transition to XML?

Section 3 - Generate an XML instance from an XML Schema

Here's how:

- ___ 1. Right-Click the **test.xsd** file in the **Solutions/Lab 5 - Schemas** folder in your project and select **Generate > XML file...** from the context menu. Set the filename to **xsdGeneratedXML.xml**, highlight **Lab 5 - Schemas** to place the result in the

correct folder, and select **Next** to open a screen that lets you select the root element as well as some options:



Go ahead and select all the options shown; be sure to change the root element from **choice** to **test** from the drop down menu and use Edit. . . to remove the prefix **p** from the Prefix column.

- ___ 2. Select **Finish** to generate an xml instance that complies with the XML Schema, test.xsd. Check its validity by clicking anywhere in the **xsdGeneratedXML.xml** file and select **validate the current state of the XML file** from the XML entry on the menu bar.

Section 4 - Generate an XML instance from a DTD

Here's how:

- ___ 1. Right-Click the **test.dtd** file in the **Lab 5 - Schemas** folder in your project and select **Generate > XML file...** from the context menu.
- ___ 2. Change File Name to **dtdGeneratedXML.xml** and click **Next**.
- ___ 3. A menu similar to that in Section 3, above should appear. Notice that where the .xsd based file pointed to the test.xsd file for grammar, Studio now points to the test.dtd file for grammar. Select **Finish** to generate an xml instance that complies with the DTD, **test.dtd**. Check its validity by clicking anywhere in the **dtdGeneratedXML.xml** file and select **validate the current state of the XML file** from the XML entry on the menu bar.

Part 2. Create and Populate an XML Schema from Basic Docs

The steps are:

1. Create a schema and define the simple elements.
2. Add definitions for the complex elements.
3. Define a new simple type using an existing simple type and restricting its values.
4. Add the attribute definitions to your schema.
5. Place your schema in a namespace.
6. Use your schema to validate an earlier XML file.

You'll perform most of these operations in a manual fashion as you would with a common text editor. At the end, you'll get a chance to use some of the XMLSchema features of Application Developer that are designed to do some of this work for you.

Section 1 - Create your Schema and Define the Simple Content Elements

Use Application Developer to create an empty XML Schema definition file (test.xsd) and insert element definitions for those elements in test.dtd that don't have attributes and that don't have children, that is, contain #PCDATA only.

Include <correct> in this group. The DTD declares it as #PCDATA but also says that the value should be constrained to the strings Yes and No. We will constrain the value of <correct> later in the lab.

As you complete this task, you will use the validation capability of Application Developer to ensure that your XMLSchema syntax is correct. If you have questions on XML Schema syntax, you can refer back to the examples in the student notes.

Here's how:

- ___ 1. Open or select the XML perspective. Right-click the **Lab 5 - Schemas** folder of your XML Labs project in the Navigator view of Application Developer. Choose **New > XML Schema**.
- ___ 2. In the **Create XML Schema** dialog box that opens, ensure that Lab 5 - Schemas is selected in the folder list (*Studio, version 5 includes the project folder name, XML Labs*) and set the filename to **test.xsd**. Click **Finish**. Application Developer will open your new Schema in the XML Schema editor.
- ___ 3. The XML Schema Editor in Version 5.0.x has *three* view **tabs**¹ below the editor pane: *Design*, *Source*, and *Graph*; Version 5.1 has *Source* and *Graph* view tabs. Select the **Source** view tab. Observe that Application Developer has created the <schema> element for you with some default information. For readability in the following step, use <Enter> to break up the schema declaration line; our result looks like this:

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

¹ The design view now appears as an unlabeled (in Version 5.1.0) pane to the right of the editor pane

```
targetNamespace="http://www.ibm.com"
xmlns:test="http://www.ibm.com">
</schema>
```

___ 4. Make these changes:

- ___ a. The default Namespace has been set to <http://www.w3.org/2001/XMLSchema>. It is common practice to declare xsd as a prefix for this namespace, requiring all XMLSchema elements to be properly prefixed. You will use that practice in this lab.
- ___ b. A targetNamespace has been defined with the default value of <http://www.ibm.com>. Please remove this declaration.
- ___ c. A namespace prefix has been established using the name of the file and the URI of the targetNamespace. Remove this declaration also, you'll be placing the schema in a namespace later. Save your file; it should be error free.

Our solution now looks like this:

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

</xsd:schema>
```

If you received an error, 1) be sure you added "xsd" before "schema" for both the start and end tags! And 2) be sure you didn't accidentally erase the > in the 2nd line!

- ___ 5. Open the **test.dtd** file located in the **Lab 5 - Schemas** folder of your project. It contains the DTD information you will use to define your Schema. It has been repeated in the previous section (Section 0.) for your convenience because we will either copy, modify, and incorporate it into our test.xsd file or we will refer to it in its current form to guide us as we populate our test.xsd file to ensure we have not missed any of the constraints a DTD can capture. **We are referring to the DTD but we are entering our results in the .xsd file.**

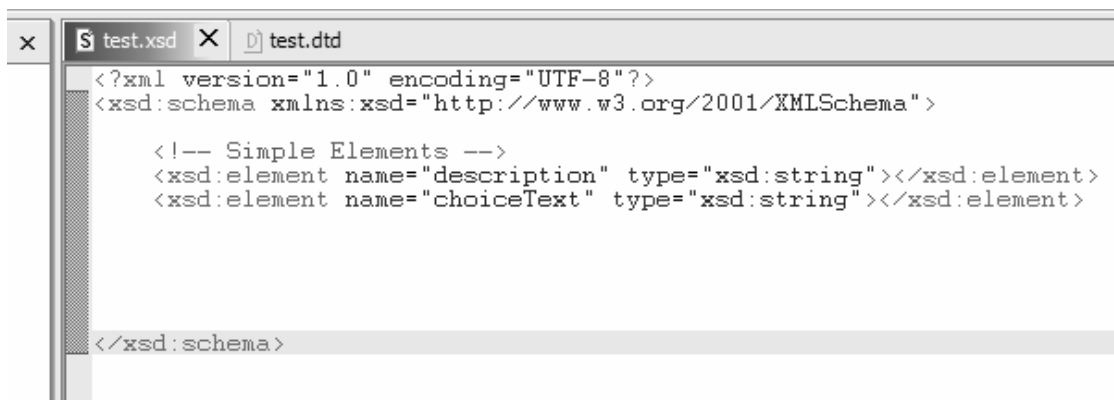
In this section we will focus only on *simple* elements:

- ___ a. Let us start with the most basic of types; these are elements that can be expressed as **<xsd:element name = "someName" type = "xsd:string" minOccurs= "1" maxOccurs= "1"/>** where the use of minOccurs/maxOccurs is optional if the values truly are "1." These will correspond to DTD "string" declarations of the form **<!ELEMENT name (#PCDATA)>**
- ___ b. Use Application developer to help you get started: begin entering data. Note the popup menu containing statement choices.² Select (double click) "xsd.element" and note that Application Developer automatically uses the generic form of **<xsd:{some choice}></xsd:{same choice}>**. The content assist feature will assist

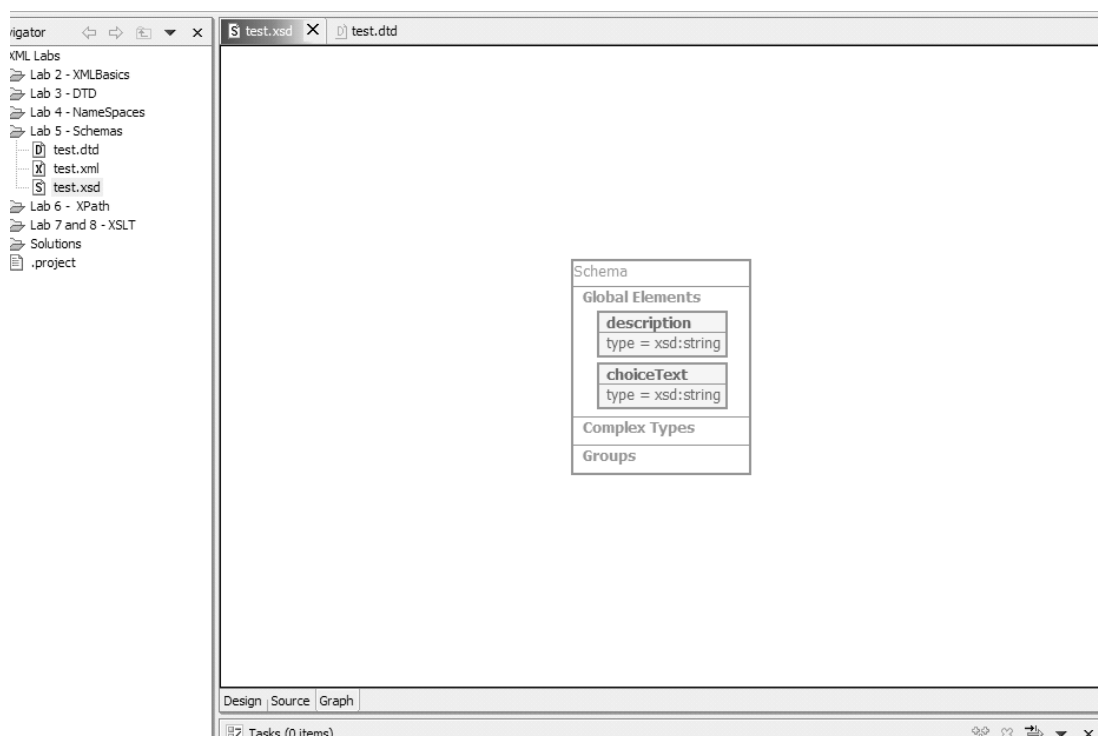
² If no menu pops up, it is because you are not between the **schema** *start* and *end* tags. If you get the popup but it does not say **xsd:some_choice** it means you did not prepend **xsd:** to **schema**. You may have to **Save** to get it to work. If you get a popup with **<DOCTYPE ...>** or comment **<!-- -->** you are below the schema end tag. Note: **<Ctrl><spacebar>** also provides a menu of choices.

you with entering all necessary information except for the attribute value for name.

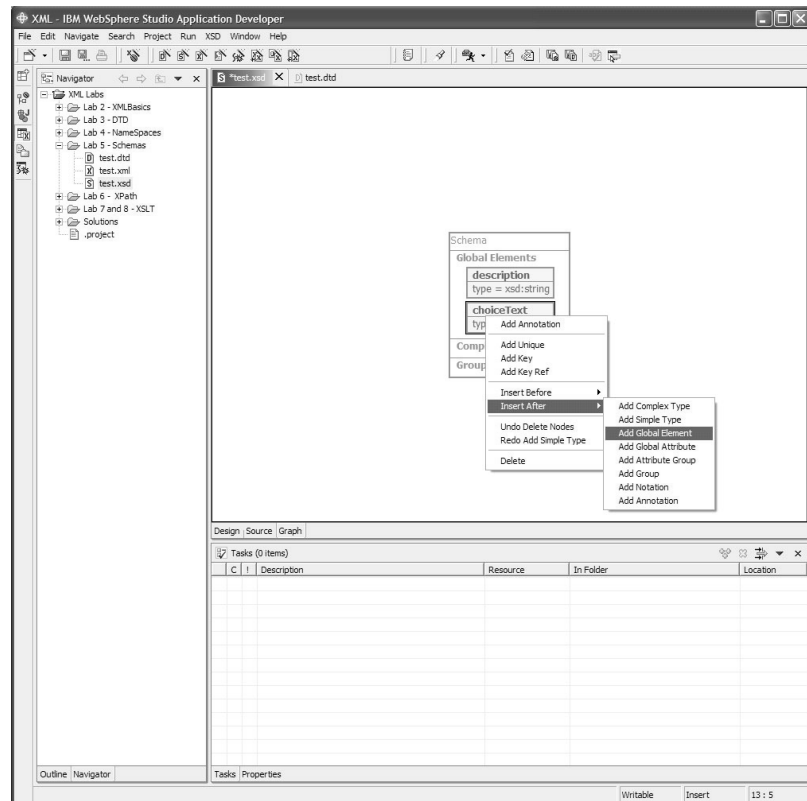
- ___ c. When you have entered two statements, experiment with the **Format** options available inside the Editor by right-clicking in the editor area. (Application Developer can be of great assistance if you become familiar with its capabilities.) Here is what our Source view looks like with two elements defined:



- ___ d. Here is another way to add elements using the new power of Studio 5.x: Switch to the Graph view:



Select the choiceText box; right-click (context-click) to bring up the menu shown below:



Select **Insert After** to open a selection menu. Add **Global element** produces the desired format, *plus* it has the advantage of automatically inserting the correct type. Select it and shift back to Source view. You should now see this:

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

    <!-- Simple Elements -->
    <xsd:element name="description" type="xsd:string"/>
    <xsd:element name="choiceText" type="xsd:string"/>
    <xsd:element name="NewGlobalElement" type="xsd:string"/>

</xsd:schema>
```

All you need do is change *NewGlobalElement* to *questionText* and you have another element defined.

Add the correct element anyway you wish³ and we will proceed.

Caution: Design (and its unnamed analog in 5.1), Source, and Graph are interconnected much the way Outline and a .xml file is: if you don't get the result you expect, try placing

³ "i" indicates an *end note*. It will be found as one of the last pages of this lab.

your cursor in another part of the source or graph. Design is a follower: you cannot introduce a new element if an existing element is selected already.

Our solution at this point adds these lines between the *schema* tags in the **test.xsd** file:

```
<xsd:element name="description" type="xsd:string"></xsd:element>
<xsd:element name="questionText" type="xsd:string"></xsd:element>
<xsd:element name="choiceText" type="xsd:string"></xsd:element>
<xsd:element name="correct" type="xsd:string"></xsd:element>
```

Don't be misled by the slightly different format from that in Step a.

- ___ e. Clearly the datatype *at this point* is going to reflect the #PCDATA called for by the DTD. We will later create the capability to set "correct" to be yes or no; for now leave it as xsd:string.
- ___ 6. Validate your schema document using one of three ways Studio provides:
 - i. save your schema file; or
 - ii. select "validate your schema" from the "XSD" entry in the menu bar; or
 - iii. use the "document-with-a-check-mark" icon (at the right of the flashlight).

Section 2 - Add Definitions for the Complex Content Elements

Now that the *Simple* elements have been defined, you can express the *Complex* elements that are composed from them and any elements that have attributes.

Here is what we have left to do:

```
<!ELEMENT test ( description, testQuestions ) >
<!ELEMENT description ( #PCDATA )>
<!ELEMENT testQuestions ( question+ )>
<!ELEMENT question ( questionText, choices )>
<!ELEMENT questionText ( #PCDATA )>
<!ELEMENT choices ( choice+ )>
<!ELEMENT choice ( (choiceText,correct) | (correct,choiceText) )>
<!ELEMENT choiceText ( #PCDATA )>
<!ELEMENT correct ( #PCDATA )>

<!ATTLIST test      id CDATA #REQUIRED>
<!ATTLIST question id CDATA #REQUIRED>
<!ATTLIST choices   allowMultiple (Yes|No) "No" >
<!ATTLIST choice    id CDATA #REQUIRED>
```

- ___ 1. Create an element declaration in the schema for each remaining element in the DTD (above) that has not yet been declared in the test.xsd file. These are considered to be Complex elements in XMLSchema so you'll have to use the appropriate syntax to define them (see below).

Figures 7-5 (which, along with 7-15 and 7-31, demonstrates the use of *ref=*) and 7-9 through 7-10 provide general guidance.

Remember that the number of occurrences of a child is constrained in some cases, you may have to specify **minOccurs** and/or **maxOccurs** values to express these constraints. Refer to Figures 7-6 through 7-8 for additional help.

Also, you'll find it easier to keep track of what you are doing if you define the elements that have Simple-content children before you define the elements that have complex-content children. Save your work often, this will avoid loss of work and it will also run the validator and let you know if any errors exist.

Finally, until you are sure you have everything defined correctly it is advisable to use inline definitions first and then replace them with named complexTypes (Figure 7-31) once everything works.

Some examples follow [you may want to refer to the solutions folder but remember, the solution folder contains the situation at the end of the exercise - some of the constructs may not obviously follow from where we are in the progression of the exercise].

___ 2. Sample inline complexType: [I copied this from the *Solutions* file.]

```
<xsd:element name="test">4
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="description"/>
      <xsd:element ref="testQuestions"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

The DTD indicates that **test** has two *children* "description" and "testQuestions" and they must occur in sequence as listed. There is only one occurrence of each, so no explicit occurrence facet need be stated.

- ___ a. Insert the above fragment into the valid test.xsd you built after the declarations of the simple types and **Save** it. You receive one error.
- ___ b. The error is obvious: the testQuestions element has not yet been defined.
- ___ 3. Create the specification for the **testQuestions** element. According to the DTD this element consists of many **question** elements.
 - ___ a. The basic structure is the same as that for test.
 - ___ b. The question elements will occur in sequence.
 - ___ c. There will always be at least one, but we do not know what the maximum number of questions will be. We do not need to include the **minOccurs** facet since it is "1".

```
<xsd:element name="testQuestions">
  <xsd:complexType>
    <xsd:sequence>
```

⁴ Elements with content can not have both a type attribute and an anonymous type child.

```

        <xsd:element ref="question" maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

```

- ___ d. Add this fragment and evaluate the result. The good news is the error involving **testQuestions** has been fixed; the bad news is we have a new error involving the **question** elements.
- ___ 4. Go ahead and create the specification for the **question** element. Based on the material we have implemented thus far, a solution could be:

```

<xsd:element name="question">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="questionText"/>
      <xsd:element ref="choices"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

Save your file and note we have moved the errors to the **choices** element.

- ___ 5. The **choices** element will look just like that for the **question** element except it has only one **child**, choice which can have many occurrences.

```

<xsd:element name="choices">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="choice" maxOccurs="unbounded">
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

```

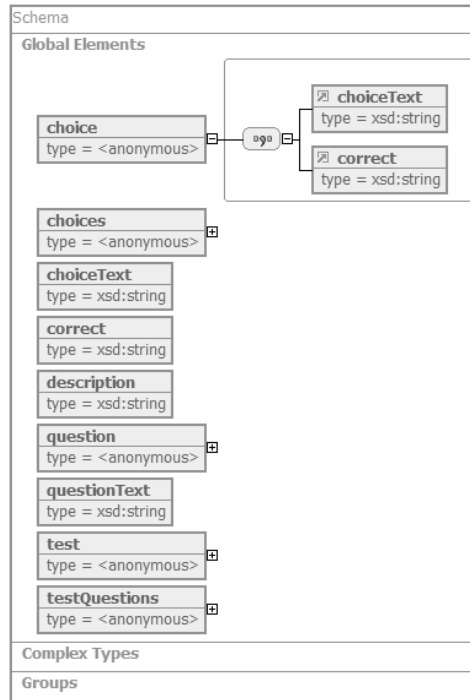
- ___ 6. The **choice** is slightly different: it has two children **choiceText** and **correct**; they may appear in either order but they both must appear. For this declaration we will, therefore, use the **xsd:all** compositor. Our solution looks like this:

```

<xsd:element name="choice">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="choiceText"/>
      <xsd:element ref="correct"/>
    </xsd:all>
  </xsd:complexType>
</xsd:element>

```

When you save your work, there should be no more errors. Here is what the Graph view looks like at this point. Choice was expanded so you could have a(n) idea of what one would look like:



It was necessary to restart Studio in order to display all the elements in Source; you may only need to close and re-open the test.xsd file.

Note: We earlier urged you to use inline definitions (that is, avoid the use of named complexTypes) to keep things simple. If you wanted to use a named type you would simply add an identifier in the declaration of the complexType. For example, for **choice** above, we might declare its complexType as follows:

```
<xsd:complexType name= "ct1">
  <xsd:all>
    <xsd:element ref="choiceText"/>
    <xsd:element ref="correct"/>
  </xsd:all>
</xsd:complexType>
```

This would then allow us to implement **choice** as follows:

```
<xsd:element name= "choice" type= "ct1"></xsd:element>
```

...or if you want the absolute minimalist approach,

```
<xsd:element name= "choice" type= "ct1" maxOccurs="unbounded" />
```

We pointed out it will be easier if, when planning to use named types, you use inline definitions first and then replace them with named complexTypes once everything is defined correctly, that is, no error messages appear.

Other than the use of a **yesOrNo** type covered below, there is no benefit to using named types in this schema as there are no element patterns that are common so reuse is not required.

CAUTION: We have not yet captured all the constraints that the power of an XML schema allows.

Section 3 - Restricting Simple Content Types

We have addressed basic simple and complex types. Here is what we have left from the DTD:

```
<!ELEMENT test ( description, testQuestions ) >
<!ELEMENT description ( #PCDATA )>
<!ELEMENT testQuestions ( question+ )>
<!ELEMENT question ( questionText, choices )>
<!ELEMENT questionText ( #PCDATA )>
<!ELEMENT choices ( choice+ )>
<!ELEMENT choice ( (choiceText,correct) | (correct,choiceText) )>
<!ELEMENT choiceText ( #PCDATA )>
<!ELEMENT correct ( #PCDATA )>

<!ATTLIST test      id CDATA #REQUIRED>
<!ATTLIST question  id CDATA #REQUIRED>
```

```
<!ATTLIST choices allowMultiple (Yes|No) "No" >
<!ATTLIST choice id CDATA #REQUIRED>
```

We will get to attributes in Section 4; right now let us address the requirement that both the **correct** element and the **allowMultiple** attribute of **choices** is constrained to the values of Yes or No.

- ___ 1. Review your notes for syntax and usage. This is a more legitimate use of a *named type* because it is used in more than one place.
- ___ 2. Define a **simpleType** called "yesOrNo" as a restriction of **xsd:string**.
 - ___ a. Create an **xsd:simpleType** element with name="yesOrNo" as a child of the schema element; named types must be declared as children of the schema.
 - ___ b. Place an **xsd:restriction** element as a child of the **xsd:simpleType** element and indicate that the base type for the restriction is **xsd:string**.
 - ___ c. Create two **xsd:enumeration** elements as children of the **xsd:restriction** element. One should have a value of Yes and the other should have a value of No.
 - ___ d. Return to the declaration of <correct> that you created earlier in the lab and change it's type to "yesOrNo".
 - ___ e. Use this type for the declaration of the allowMultiple attribute in the next section.

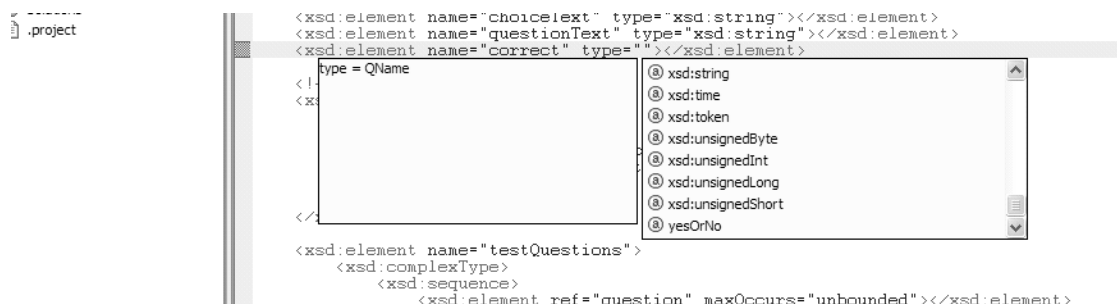
Our solution looks like this:

```
<xsd:simpleType name="yesOrNo">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="Yes"/>
    <xsd:enumeration value="No"/>
  </xsd:restriction>
</xsd:simpleType>
```

- ___ 3. This block of code may appear anywhere, but to make it easier to use and maintain, insert your simpleType definition immediately after the **<xsd:schema>** tag.
- ___ 4. To use this you need to modify our earlier work.
 - ___ a. Earlier we had:

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

- ___ b. Highlight **xsd:string**, delete it, and then use **<Ctrl><spacebar>** to bring up a menu which now includes **yesOrNo** as a choice. Highlight the **yesOrNo** and press **<Enter>**



to produce:

```
<xsd:element name="correct" type="yesOrNo"></xsd:element>
```

Section 4 - Add the attribute definitions to the Schema

You now have a schema that describes your document in terms of its elements, their nested relationships and content. Its time to add definitions for the attributes that are permitted on each element. Remember that, in XML Schema, an element's attributes are defined as children of its definition.

- ___ 1. For each attribute declared in the DTD, create an attribute declaration in the Schema. Here is what we have left from the DTD:

```
<!ELEMENT test ( description, testQuestions ) >
<!ELEMENT description ( #PCDATA )>
<!ELEMENT testQuestions ( question+ )>
<!ELEMENT question ( questionText, choices )>
<!ELEMENT questionText ( #PCDATA )>
<!ELEMENT choices ( choice+ )>
<!ELEMENT choice ( (choiceText,correct) | (correct,choiceText) )>
<!ELEMENT choiceText ( #PCDATA )>
<!ELEMENT correct ( #PCDATA )>

<!ATTLIST test      id CDATA #REQUIRED>
<!ATTLIST question id CDATA #REQUIRED>
<!ATTLIST choices   allowMultiple (Yes|No) "No" >
<!ATTLIST choice    id CDATA #REQUIRED>
```

Refer to your course notes, hover help, or the *Studio Help* facility, if you need guidance.

- ___ 2. Remember to run the validator often, saving your work is an easy way to do this.
- ___ a. Let's start by implementing the id attribute in **test**; fortunately, all the id attributes are implemented similarly.

```
<xsd:attribute name="id" type="xsd:ID"/>
```

ID is one of the "built-in simple types" listed in Figure 7- 17. Our solution for **test** with the attribute incorporated looks like this:⁵

Don't forget to add the use attribute to the specification.

```
<xsd:element name="test">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="description"/>
      <xsd:element ref="testQuestions"/>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"6 use="required"/>
  </xsd:complexType>
</xsd:element>
```

___ b. Do the same as above for **question** and **choice**; use **Save** to check your work.

```
<xsd:element name="question">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="questionText"></xsd:element>
      <xsd:element ref="choices"></xsd:element>
    </xsd:sequence>
    <xsd:attribute name="id" type="xsd:ID"7 use="required"/>
  </xsd:complexType>
</xsd:element>
<xsd:element name="choice">
  <xsd:complexType>
    <xsd:all>
      <xsd:element ref="choiceText" />
      <xsd:element ref="correct" />
    </xsd:all>
    <xsd:attribute name="id" type="xsd:string"8
use="required"></xsd:attribute>
  </xsd:complexType>
</xsd:element>
```

___ c. That leaves **choices**: we already have a **yesOrNo** simpleType defined; all that remains is to capture the default to "no" and test our result. Refer to the figures just before step 3 (on the previous page) for guidance, if necessary.

```
<xsd:element name="choices">
  <xsd:complexType>
```

⁵ You can use control assist to augment your memory. Here we are using it to find the correct type.

⁶ If there is to be only one test per XML instance, then typing this as ID is pointless; STRING would do.

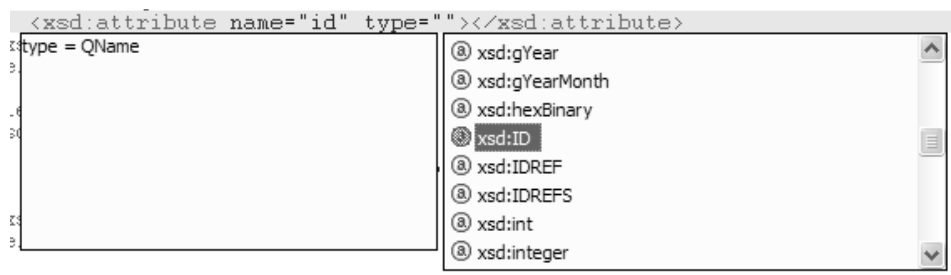
⁷ The question number should be unique within this instance.

⁸ Every question could have an A, B, C, and so forth; they won't be unique.

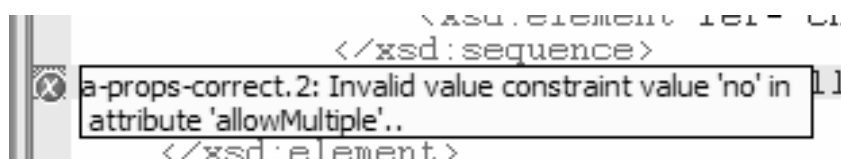

```

<xsd:sequence>
  <xsd:element ref="choice" maxOccurs="unbounded"/>
</xsd:sequence>
<xsd:attribute name="allowMultiple" type="yesOrNo" default="No"/>
</xsd:complexType>
</xsd:element>

```



Note: If you receive this error:



You spelled "No" wrong. (The 'n' is *capitalized*.)

At this point we have completed our implementation of the knowledge we currently have about the exam schema. We need to do some housekeeping and then apply our xsd schema to validate a document. Use the context menu in the editor window to **Format->Document**.

Part 3. Joining XML Schema and XML Instances

Section 1 - Associate Your "handbuilt" Schema with a Namespace

The last step in creating a schema is to make sure that all your definitions are associated with a Namespace so that they don't conflict with names in some other XML language. In the following, Studio content assist can help you avoid silly blunders. Once you've loaded the URI below into your cut and paste buffer you should be ready to go.

- ___ 1. Select a URI for the Namespace you want your definitions to be in. We chose **<http://www.mycorp.com/Schema/Test>**.
- ___ 2. Add a targetNamespace attribute with the value of your Namespace URI to the <xsd:schema> element.

Literally, targetNamespace="http://www.mycorp.com/Schema/Test".

- ___ 3. Add a namespace declaration to your schema that identifies the URI used for the targetNamespace as the default namespace in the schema. Given our choice of URI, this has to be **xmlns="http://www.mycorp.com/Schema/Test"**
- ___ 4. Add an **elementFormDefault="qualified"** and an **attributeFormDefault="unqualified"** attribute to the <schema> element.

By adding these declarations to the schema, you are asserting that XML documents constrained by this schema must qualify references to its elements with the proper Namespace but attribute references need not be qualified. Don't forget to **Save!** The 1st few lines of our **test.xsd** now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.mycorp.com/Schema/Test"
  targetNamespace="http://www.mycorp.com/Schema/Test"
  attributeFormDefault="unqualified"
  elementFormDefault="qualified">
  <!-- these are the simple elements, they have no children and no
attributes -->
  . . .
</xsd:schema>
```

EOF

Section 2 - Validate a document using this Schema

In this section, you will use your new schema to validate a document that currently references the DTD that you began with. After you have done this, you may want to introduce invalid content into the document to see what kind of error messages are produced.

- ___ 1. Open the file **test.xml**, located in **Lab 5 - Schemas** folder of your project.
- ___ 2. Select the Source view of this document.
- ___ 3. Remove the <!DOCTYPE declaration from this document.
- ___ 4. Add a namespace declaration for the XML Schema Instance namespace to the root element of this document, that is,
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance".
- ___ 5. Add an **xsi:schemaLocation** attribute to the root element of this document. Remember that each value of an xsi:SchemaLocation is a pair consisting of a namespace URI and the location of the schema. In this case it should be the name of your file. **test.xsd** Our solution is

```
xsi:schemaLocation="http://www.mycorp.com/Schema/Test test.xsd"
```

- ___ 6. Add a namespace declaration making the target namespace for your schema the default namespace for the document. xmlns="the URI you used in the XSD" For us, that is

```
xmlns="http://www.mycorp.com/Schema/Test"
```

- ___ 7. Save test.xml to invoke the validator. This document is valid according to the DTD and should also be valid according to your schema. If it fails validation against your schema, the error messages should help you to determine the area of your schema that improperly expresses the language it contains.

The 1st several lines of test.xml should now look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<test
  xmlns="http://www.mycorp.com/Schema/Test"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.mycorp.com/Schema/Test test.xsd"
  id="t1">
  <description/>
  .
  .
  .
</test>
EOF
```

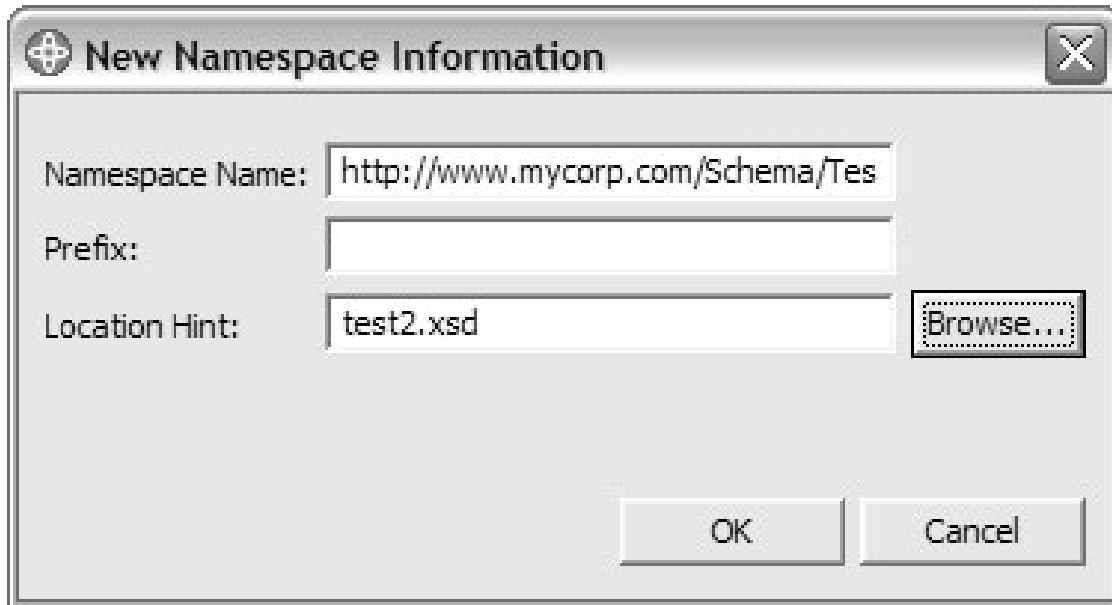
Section 3 - Using Studio to Associate Schema and Instances

This is a repeat of material covered in lecture but it is included to allow you to practice. We described a process in Section 1 of the lecture notes for associating our schema definitions with a namespace and then associating the schema and xml instance. We can use the power of Studio to help us without using the technique described above.

Here's how:

- ___ 1. Begin by copying your "handbuilt" test.xsd file into a new file named test2.xsd.
- ___ 2. Copy the test.xml file into a new file named test2.xml.
- ___ 3. **If** you skipped Section 2 (above), remove the line `<!DOCTYPE test SYSTEM "test.dtd" >` from test2.xml. (You may wish to remove the opening comment as well to increase readability.)
- ___ a. **Otherwise**, remove the three lines (shown in **bold** in Section 2., above) you just added.

- ___ 4. Select test2.xml and right-click to open a menu from which **Assign > XML Schema...** may be selected to open the New Namespace Information window from which we can (via **Browse...**) select **test2.xsd** which auto-fills the entries as follows:



and click **OK**.

- ___ 5. As a result, the beginning of the test2.xml file is modified so that the beginning of the file looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<test id="t1"
xmlns="http://www.mycorp.com/Schema/Test"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.mycorp.com/Schema/Test test2.xsd ">
    <description/>
    .
    .
    .
</test>
EOF
```

Notice how *Application Developer* rearranged the order of the attributes so that id now appears first.

- ___ 6. Click anywhere in the test2.xml view; the file should again be valid.

ⁱYou can also use the Design tab in Version 5.0: begin the definition of the element in Source, then select the Design tab to add the rest:

Element

Element name:

Type information

☒ Built-in simple type
☐ User-defined simple type
☐ User-defined complex type

xsd:string
xsd:short
xsd:string
xsd:time
xsd:token
xsd:unsignedByte

Value Information

☒ Fixed
☐ Default

Advanced

Abstract:
Nillable:
Block:
Final:
Substitution group:
Form qualification:

Here is how Version 5.1 presents similar information (after correct has been converted to a *defined* type):

XML - test.xml - C:\XML4.0 - IBM WebSphere Studio Application Developer

File Edit Navigate Search Project Run XSD ClearCase Window Help

test.xml

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

  <!-- Defined Simple Type -->
  <xsd:simpleType name="yesOrNo">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="Yes" />
      <xsd:enumeration value="No" />
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Built-in Simple Types -->
  <xsd:element name="description" type="xsd:string" />
  <xsd:element name="questionText" type="xsd:string" />
  <xsd:element name="choiceText" type="xsd:string" />

  <!-- User Defined Simple Type -->
  <xsd:element name="correct" type="yesOrNo" />

  <!-- Complex Type -->
  <xsd:element name="test">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="description"/>
        <xsd:element ref="testQuestions"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="testQuestions">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="question" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

```

Element

Name:

Type information

☐ Built-in simple type
☒ User-defined simple type
☐ User-defined complex type

yesOrNo

Value

☒ Fixed
☐ Default

Other Attributes	Value
abstract	
nillable	
block	
final	
substitutionGroup	

Source Graph

Writable Insert 18 : 19

END OF LAB

Exercise 6. XPath

What This Exercise Is About

In this exercise, you extract portions of an XML document using XPath expressions. The exercise consists of several XPath challenges for you to solve. You will use the XPath Expression Wizard feature of Application Developer 5.x to test your expressions. A simple Online Exam XML document is provided as an input source. The italicized material in this exercise is from the version 5.0 Help 'Help Contents' Search ("XPath Wizard") Go search (which contains additional material beyond the scope of this introductory exercise).

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Analyze an XML document and extract portions using XPath expressions
- Use the XPath Expression Wizard feature of Application Developer to execute XPath expressions against a given XML source document

Required Materials

- Application Developer 5.x
- Lab resource files for Lab 6 - XPath

Applicability

- XML301
- XML341

Timing

The material in this unit is open-ended: one could spend all day exploring possibilities. Limit yourself to about 1 hour in class.

Much of the material is explanatory: once you grasp the concept of how to effectively use *Studio* to test your path, skip to Section 3.

Are you using <Ctrl>s to save? It's really quick!

Exercise Instructions

Section 1.0 - Introduction

The XML Path Language (**XPath**) is an XSL sublanguage designed to uniquely identify or address parts of a source XML document. An **XPath expression** can be used to search through an XML document, and extract information from the **nodes** (any part of the document, such as an element or attribute) in it. There are four different kinds of XPath expressions:

Boolean

An expression type with two possible values.

Node set

A collection of nodes that match an expression's criteria, usually derived with a location path.

Number

A numeric value, useful for counting nodes and for performing simple arithmetic.

String

A fragment of text that may be from the input tree, processed or augmented with general text.

To accomplish this search and extract function in WebSphere version 4 someone had to supply two files one of which, filter.xsl, had to be manually changed to each trial XPath expression and then reexecuted to extract information from an XML instance. The other file, process.xsl, provided a stylesheet, which used the now deprecated term, transform, to create an HTML file to display the results of applying the filter.xsl file.

WebSphere 5 not only provides a direct capability to execute XPath expressions, it provides tabs which, in turn, can provide additional guidance and help.

Section 1.1 - WebSphere 4.0

To understand the files that we used with four requires material we will cover in the lectures on XSL transforms. However, the constructs are sufficiently intuitive and the accompanying comments sufficiently helpful that we include them here for your consideration. (Note the use of a relative path to locate the process.xsl file.)

Filter.xsl

```
<?xml version="1.0" encoding="UTF-8" ?>
  <xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0" xmlns:xalan="http://xml.apache.org/xslt">
  <xsl:variable name="expr" select="/" />
  <xsl:include href="./process.xsl" />
</xsl:transform>
```

Your test
expression
goes here

Process.xsl

```
<?xml version="1.0" encoding="UTF-8"?>
  <!-- This stylesheet should be included in another stylesheet that
        declares a variable called expr; the value of which is an XSLT
        compatible XPath expression. The match '/' template contained
        herein will generate an HTML document containing the result of
        the expression when applied against the source document.
        Do not reformat this file, pretty printing may mess up the output.
  -->
```

```
<xsl:transform xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0"
xmlns:xalan="http://xml.apache.org/xslt">
```

```
<xsl:output method="html" indent="no"/>
```

This forces the
output to be HTML

```
<xsl:template match="/">
  <html>
    <head>
      <title>XPath Expression Tester</title>
    </head>
    <body bgcolor="lightblue">
      <h1>XPath Expression Tester</h1><br/>
      <h2>Expression Result:</h2><hr/>
      <xsl:apply-templates select="$expr" mode="parse"/><hr/>
    </body>
  </html>
</xsl:template>
```

```
<!-- This template gets fired when the student expression matches an
Element node. -->
<xsl:template match="*" mode="parse">
  <p/><xsl:call-template name="renderStartTag"/>
  <xsl:call-template name="renderAttributes"/><![CDATA[&gt;]]>
  <xsl:apply-templates mode="parse"/>
  <xsl:call-template name="renderEndTag"/><p/>
</xsl:template>

<!-- This template gets fired when the student expression matches
Attributes. It just renders each attribute followed by <p/>.
-->
<xsl:template match="@*" mode="parse">
  <xsl:call-template name="renderAttribute"/><p/>
</xsl:template>

<!-- rendering stuff -->

<xsl:template name="renderStartTag">
  <![CDATA[&lt;]]><font color="blue"><xsl:value-of
select="name(.)"/></font>
</xsl:template>

<xsl:template name="renderEndTag">
  <![CDATA[&lt;]]></font><font color="blue"><xsl:value-of select="name(.)"/>
</font><![CDATA[&gt;]]>
</xsl:template>

<xsl:template name="renderAttributes">
  <xsl:for-each select="@*">
    <![CDATA[&#x0020;]]><xsl:call-template name="renderAttribute"/>
  </xsl:for-each>
</xsl:template>

<xsl:template name="renderAttribute">
  <font color="purple"><xsl:value-of
select="name(.)"/></font>=&quot;<font color="Fuchsia"><xsl:value-of
select="."/></font>&quot;
</xsl:template>

</xsl:transform>
```

Those of you familiar with HTML will recognize html, head, body, title, font, h1, h2, hr, and p as HTML tags. What process.xsl does, as you will learn in the next lectures, is to transform selected pieces of an XML file into an HTML file.

The way WSAD 4 worked was to have us jointly select our XML file and the filter.xsl file with our test expression in place of the '/', and apply the XPath API package in Xalan to the combination using the process.xsl file referenced in filter.xsl as a stylesheet.

The above is obviously an oversimplification but, here's hoping, you get the idea.

Let us continue and introduce you to how we test expressions using WSAD 5.x.

Section 1. 2 - WebSphere 5.x

___ 1. To launch XPath:

You can launch the XPath Expression wizard in any of the following ways:

- ___ a. The pop-up menu of an XML file. (Right-click the file and select **Generate > XPath...**)
- ___ b. The pop-up menu of an XML schema file, (Right-click the file and select **Generate > XPath...**)
- ___ c. The XSL editor. (Select **XSL > XPath Expression.**)
- ___ d. Select **File > New > Other > XML > XPath.**

If you launch the XPath Expression wizard via the XSL editor or New wizard, you are prompted to select an XML file or XML schema....

___ 2. To create an XPath expression using the XPath Expression wizard, following these steps:

- 1) Switch to the XML perspective.
- 2) Launch the XPath Expression wizard using one of the methods described above.
- 3) Select a context node for the XPath expression. The context node is the starting point for your XPath expression; it can be the document root node or any node contained the root node. [This is key. . .this is where you identify the current node. See the examples, later.]
- 4) The **History list** lists any XPath expressions you have previously created for your XML document. If you want to base your new XPath expression on an existing one, select it from the list and click **Next**. [This list will disappear when you click **Finish**, which closes the Wizard.] Otherwise just click **Next**.
- 5) The **XPath** field contains your XPath expression. You can create or modify it by selecting the appropriate options from the nodes tree and various pages in the XPath Expression wizard.

- 6) The nodes tree lists all the nodes (such as attributes and elements) in your XML document. Select the node(s) you want to search for.
- 7) Click the **Location Paths** tab. You can specify the following in the Location Paths page:
 - **Axis specifiers** - Determines the direction you move within the XML document.
 - **Node tests** - Specifies what kinds of nodes to search for.
 - **Predicate** - Enables you to specify specific values to search for.
 - **Abbreviations** - Short cuts that enable you to shorten the syntax of the XPath expression.
- 8) Click the **Operators** tab. You can specify the following in the Operators page:
 - **Node-sets** - Operators that combine or define paths.
 - **Booleans** - Operators that compare string or numeric expressions, or Boolean values.
 - **Numbers** - Operators that can be used to manipulate numeric values.
- 9) Click the **Functions** tab. You can specify the following in the Functions page:
 - **Node set** - Functions that apply to node sets.
 - **String** - Functions used for dealing with strings.
 - **Boolean** - Functions used for Boolean mathematics.
 - **Number** - Functions used to manipulate numeric values.
 - **XSLT** - XSLT functions used in XPath expressions.
- 10) Click the **Match Conditions** tab. This page contains a list of possible conditions for your XPath expression based on the node you have selected from the nodes tree. To add a condition to your XPath expression, select it from the list.
- 11) Click **Clear** to remove your XPath expression from the **XPath** field, or click **Execute** to see the results of your XPath expression. The execution is based on the XPath API package in Xalan.
- 12) *Edit your expression as necessary and click **Finish** when you are finished [with the Wizard]. [This action clears the History List.]*

Section 2 - Examine a Sample XML file

In this section, you examine the sample Online Exam XML file to be used in this exercise.

- ___ 1. Open the document **exam.xml** located in the **Lab 6 - XPath** folder of your **XML Labs** project and look at the Source view.
- ___ 2. Analyze the tree structure and elements within the document.

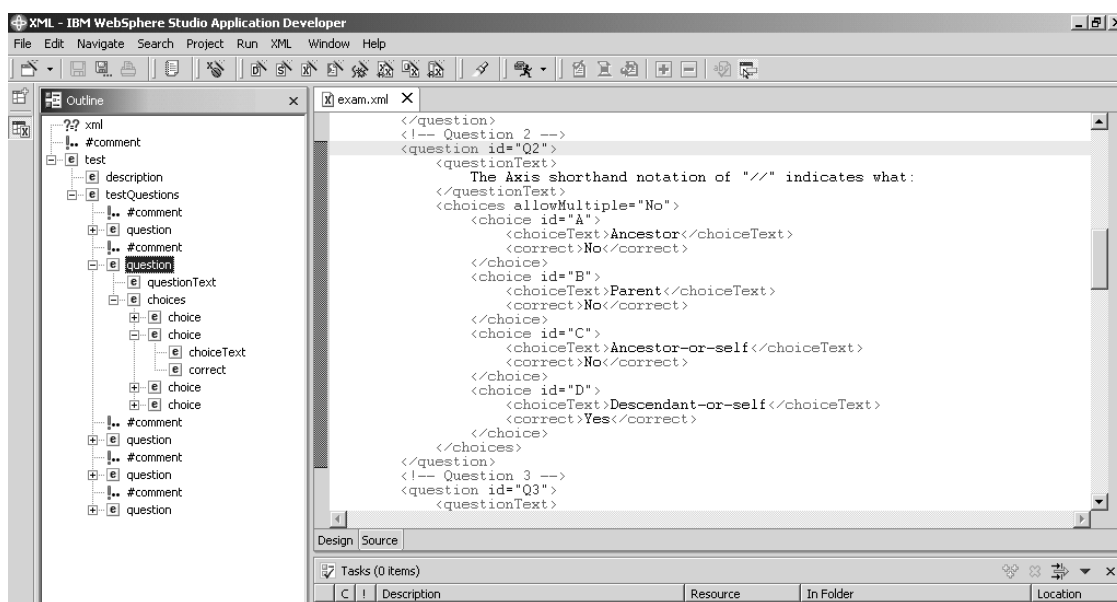
- ___ 3. Keep the document open as a reference for the rest of this exercise. You may wish to either copy it to a text file in Notepad since the XPath wizard and output windows will obscure your view of the XML editor view; having it in a separate (text) file lets you switch between Studio and the text file, or view the XML file in the Outline view and use method, or to open the wizard.

Section 3 - Testing XPath Expressions in Application Developer

In the next section, you are presented with a series of XPath challenges. For each challenge, you will construct an XPath expression and test it using the XPath Expression Wizard in Application Developer to select information matching your expression from the **exam.xml** instance.

You will use the XPath Expression Wizard in Application Developer to execute an expression that you believe selects the document portion requested in each challenge.

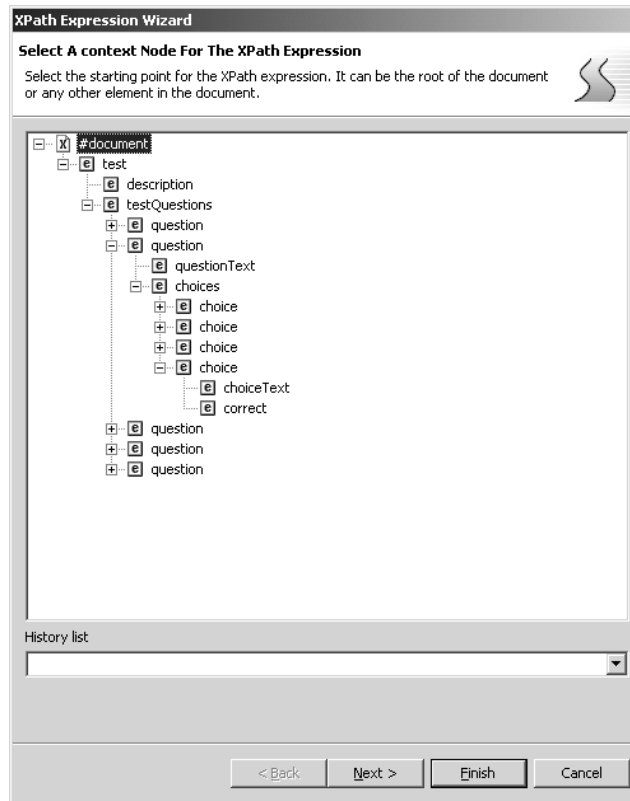
- ___ 1. Let's go through an example so that you get the sense of how this works. Let us use the wizard to extract the text of the correct choice of the question whose id = "Q2".
- ___ a. To quickly orient ourselves using the knowledge we have gained thus far with the exam example, click the **Outline** tab and select the second question.
- ___ b. The editor immediately realigns itself to display that block of the XML file as shown here:



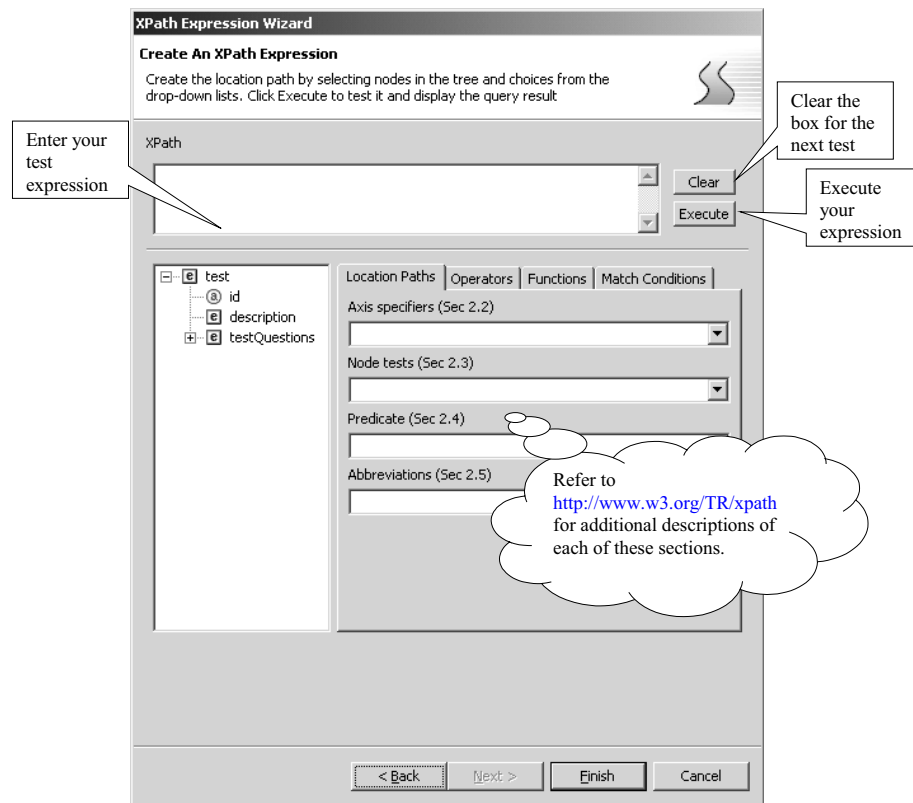
We see the answer we seek is "Descendant-or-self" in the choice whose id = "D":

```
<choice id="D">
  <choiceText>Descendant-or-self</choiceText>
  <correct>Yes</correct>
</choice>
```

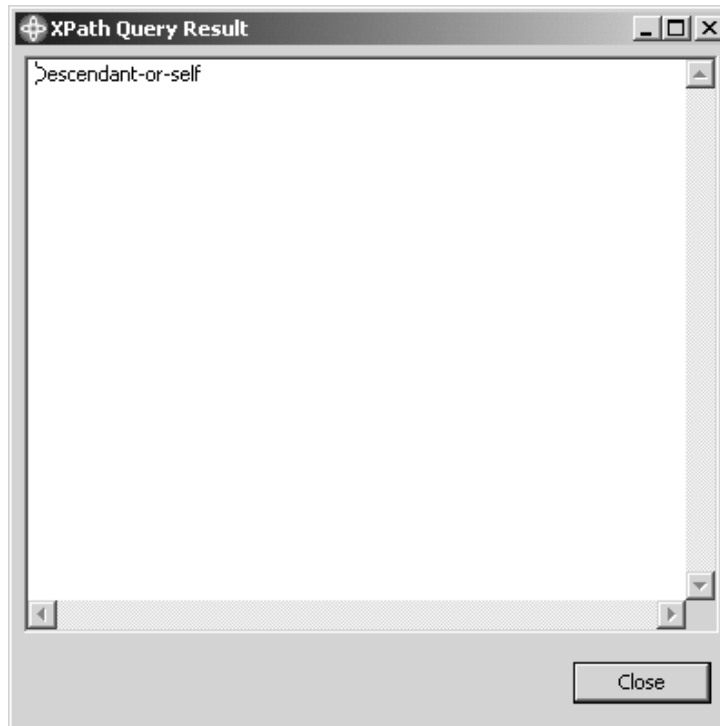
- ___ c. Switch to the **Navigator** view, right-click **exam.xml** and select **Generate> Xpath...**, (near the bottom of the context (right mouse-click) menu). This will launch the XPath Expression Wizard. (In the screen capture below, we have expanded **test** and **testQuestions** and the second occurrence of **question** and **choices** and the fourth **choice** just to show you what it would look like. Please note we then reselected **#document** for the next step.)



- ___ 2. **Select a Context Node for the XPath Expression:** If you want the wizard to present a subset of a larger document for you to examine, you can select the parent node of the subset here. For this instance, leave the default selection (**#document**) and click **Next>** to open the wizard:

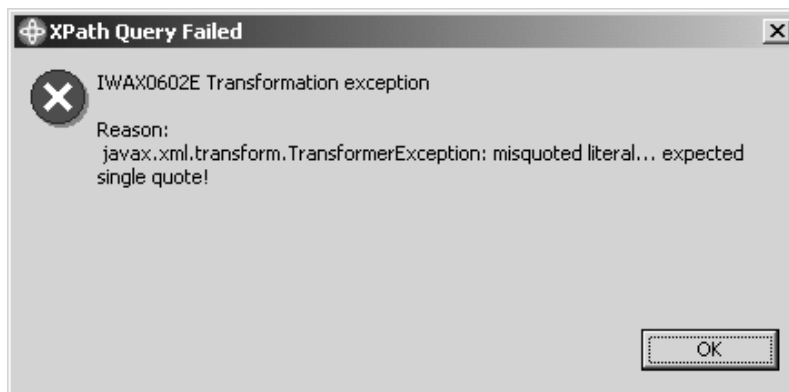


- ___ 3. **Enter an XPath Expression:** XPath expressions are entered in the XPath text area on this page of the wizard. Once provided, the expression can be tested by clicking the **Execute** button.
- ___ a. Based on what we have learned, an expression to solve this is
`"//question[@id='Q2']//choice[correct='Yes']/choiceText/text()"`.
- ___ b. Enter the expression (selecting the entire document) and execute it. The results of the expression will appear in a query result window.



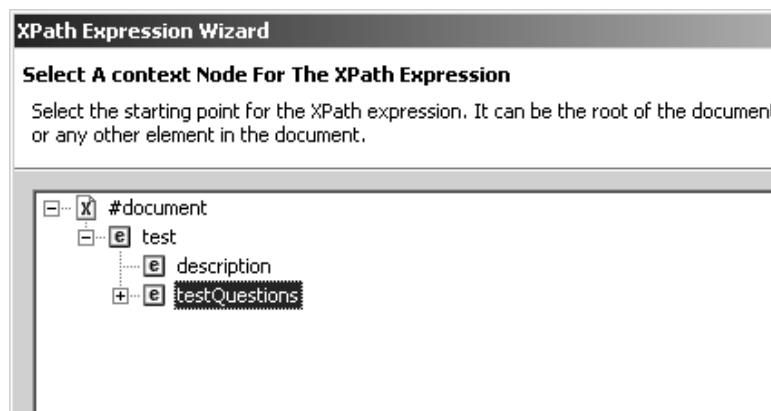
Showing that our expression returns the correct answer. If this expression returned null, check to insure that you typed Yes rather than yes.

- ___ 4. If the XPath Expression Wizard encounters a syntax error in your expression, an error message will be displayed with a brief hint as to the cause. Try this now, enter the following expression and execute it: '/' (a single quote followed by a forward slash). The following error will be displayed.

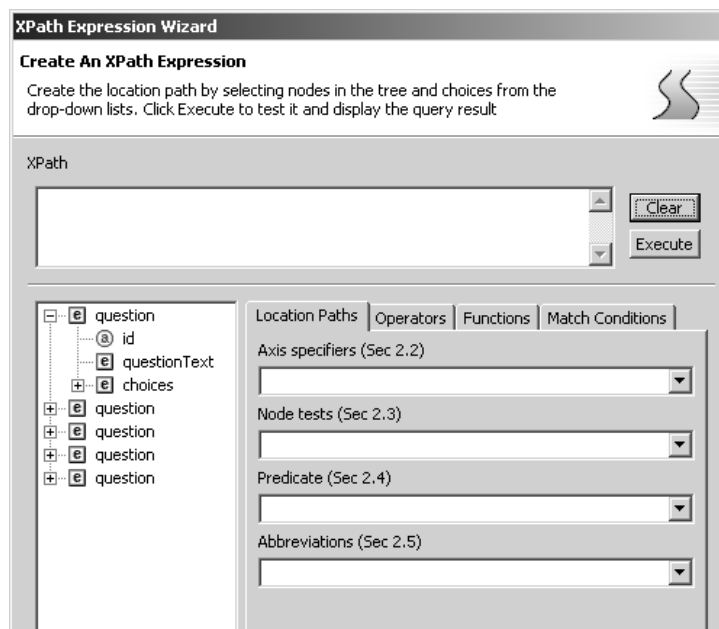


- ___ 5. **Explore the XPath Wizard:** XPath expressions can be constructed on the fly using a combination of reference-node selection and the smaller menu windows. Recall, the challenge is to extract the text of the correct choice of the question whose id = "Q2".

- ___ a. Let's set the context node to be **testQuestions**: first open the Wizard via method 1 above, and select **Next>**; you should see the same picture we have above [with the callouts]. Click **testQuestions**; note that **./testQuestions/** appears in the Xpath window in the Wizard. **Execute** produces null. Change **./** to **./.**, which stands for element descendants of the context node. Try it again! [Note: **./.** instead of **./** also produces all nodes at and below test question.] To set the context node to be **testQuestions** use the **<Back** button, expand **test** and select **testQuestions** as shown here:



Notice we are doing exactly what the description in the information window suggests: we are selecting the starting point for our XPath expressions, which is an element other than the root of the document. Click **Next>** to open the Wizard:



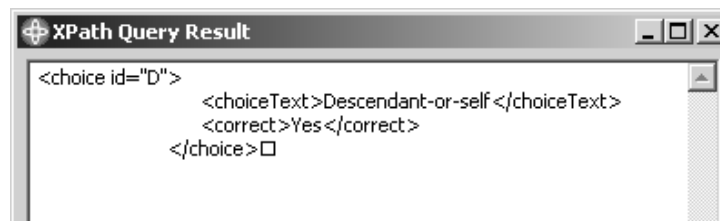
Notice our starting point is now the first **question** element. Select the Axis specifier for descendant-or-self, and it will be added into the expression window.

Execute at this point produces nothing: we need a node test. Try the various Node tests to see what results each produces. Note the text in the gray area below the window that describes what the Wizard is attempting.

- ___ b. We want the **question** whose attribute ID, is 'Q2'. Click **question** to produce `./question/`. Move your cursor to the left of the `/` and select `[]` from the Predicate menu. Place your cursor inside the `[]` and select `@id` from the tree; the `=` can be obtained from **Booleans (Sec 3.4)** under the **Operators** tab but it is necessary to manually add 'Q2' after `@id=` to form `./question[@id='Q2']`. **Execute** produces everything for Q2, most of which is shown below.



- ___ c. We want the text of the correct choice for this question. The correct choice is the one of **choices** whose **correct** element has text content = 'Yes'. Click **choices** to append `choices/`; click **choice** to append `choice/` and insert another predicate to the left of the `/`; in the predicate, click **correct** and change the `/]` to `'Yes']` to produce



We have one more level to go since we are one level above the text we want: select **choiceText** to bring us to the node we seek (try it and see). All we want is the text for this node so select **text()** from the **Node tests (Sec 2.3)** and **Execute** to produce our final result. The expression we have constructed starting from a node levels down from the root is

`./question[@id='Q2']/choices/choice[correct='Yes']/choiceText/text()`

- ___ 6. Try some other starting points and approaches or use the challenges below to practice with the power afforded by WebSphere Studio Version 5. Remember, though, that a tool of this complexity may require many hours of study and practice to truly master.

Now you're ready to test your own XPath expressions. For each challenge presented below, use the XPath Expression Wizard to test an expression that you believe selects the elements requested. Try using the features of the subwindows to help you. Use **<Back** to change your starting point if you wish.

Section 4 - Test your XPath Knowledge

Easy - as pi...

- ___ 1. Extract the ID of this test. Answer: _____

- ___ 2. Extract all questionText elements.

Answer: _____

- ___ 3. Extract all the ID attributes from this document.

Answer: _____

Medium - You may need to think about these

- ___ 1. Extract the second to the last question element on this test.

Answer: _____

- ___ 2. Extract the questionText element of the question with an ID of "Q4".

Answer: _____

- ___ 3. Extract the first choice descendant of the last question element in the document.

Answer: _____

- ___ 4. Extract all the text in the document.

Answer: _____

- ___ 5. Extract the ID of the last choice for each question.

Answer: _____

Hard - So you really think you know XPath now?

- ___ 1. Extract the text of the correct choice of question ID "Q2".

Answer: _____

- ___ 2. Extract the ID of all questions that allow multiple selected answers.

Answer: _____

- ___ 3. Extract the ID of all questions with more than four choices.

Answer: _____

___ 4. Extract any questionText element containing the word "XPath" in its text.

Answer: _____

___ 5. Extract the text of any choices that contain the string "Ancestor".

Answer: _____

___ 6. Extract the text of all correct choices in the test.

Answer: _____

Section 4 - XPath - Solutions

Easy - as pie...

- ___ 1. Extract the **id** of this test.
`/test/@id`
- ___ 2. Extract all **questionText** elements.
`/test/testQuestions/question/questionText`
Or `//questionText`
- ___ 3. Extract all the **id** attributes from this document.
`//@id`

Medium - You may need to think about these

- ___ 1. Extract the second to the last **question** element on this test.
`//question[position()=last()-1]`
- ___ 2. Extract the **questionText** element of question number (id) "Q4".
`//question[@id='Q4']/questionText`
- ___ 3. Extract the first **choice** descendant of the last **question** element in the document.
`//question[last()]//choice[1]`
- ___ 4. Extract all the text in the document.
`//text()`
- ___ 5. Extract the **id** of the last **choice** for each **question**.
`//question//choice[last()]/@id` Or `//choice[last()]/@id`

Hard - So you really think you know XPath now?

- ___ 1. Extract the text of the correct choice of question ID Q2.
`//question[@id='Q2']//choice[correct='Yes']/choiceText/text()`
- ___ 2. Extract the id of all questions that allow multiple selected answers.
`//choices[@allowMultiple='Yes']/../@id`
- ___ 3. Extract the ID of all questions with more than four choices.
`//choices[count(choice)>4]/../@id`
- ___ 4. Extract any **questionText** element containing the word "XPath" in its text.
`//questionText[contains(text(),'XPath')]`
- ___ 5. Extract the text of any choices that contain the string Ancestor.
`//choice[contains(choiceText,'Ancestor')]/choiceText/text()`
- ___ 6. Extract the text of all correct choices in the test.

`//choice/correct[text()='Yes']/../choiceText/text()`

END OF LAB

Exercise 7. XSLT Part 1 - Simple XSL Transforms

What This Exercise Is About

XSLT is a language for defining transforms (also called stylesheets) that ingest XML documents and generate other structures from them. XSLT implements XPath expressions to extract information from the source document(s). WebSphere Studio Application Developer contains Xalan, the XSLT processor implementation developed by the Apache Project.

This Lab is the first in a series of two labs in which you will define a transform that ingests an XML document containing the content of an Exam and generates a web-based user interface through which the exam questions may be answered. You will use WebSphere Studio Application Developer to create the transform and apply it to the XML source document. This lab builds on the knowledge you acquired using the XPath Wizard.

The second lab in this series contains instructions for completing the project.

What You Should Be Able to Do

At the end of the lab, you should be able to:

- Identify and understand the means by which an XML document is transformed
- Use WebSphere Application Developer version 5 to create a simple XSL stylesheet
- Use WebSphere Application Developer version 5 to transform XML documents into HTML and XML using the stylesheet you develop.
- Use WebSphere Application Developer version 5 Debug Perspective to trace execution of a transformation.

Required Materials

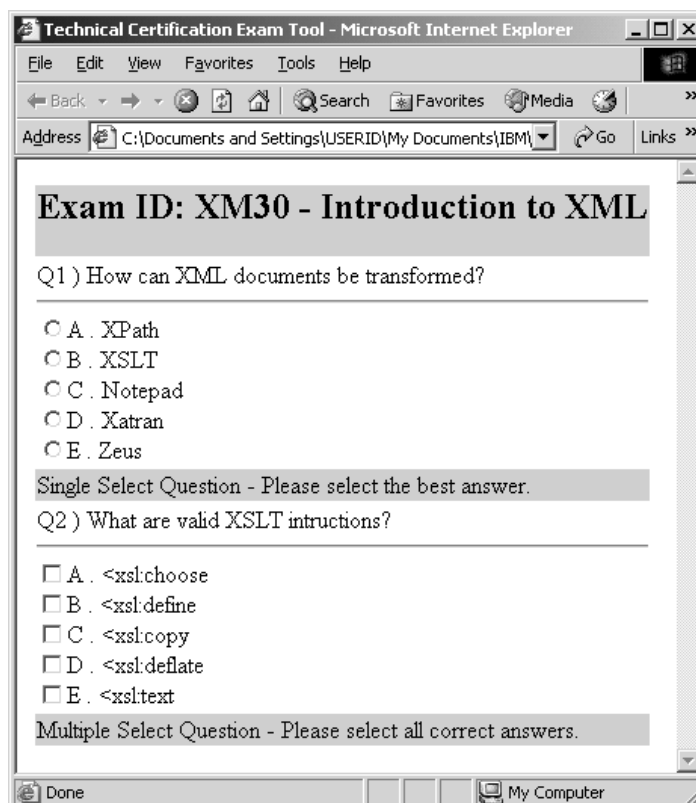
- exam.xml, an XML document containing the content of an exam.
- WebSphere Studio Application Developer Version 5.x
- The folder C:\xml\labs\Startup Files

Applicability

- XML301
- XML341

Exercise Instructions

The goal of this lab, is to transform the **exam.xml** document into the web page shown below. Note that this screen capture is the result of using *Microsoft Internet Explorer*. The browser in Studio will not produce this *exact* rendering (the title *Technical Certification Exam Tool* being but one example).



The "<" character in the choices for Q2 is appropriate for an opening tag; its presence is part of the choice; more about displaying them as shown rather than as < in Exercise 8.

Exercise Overview

This exercise is designed to ensure that you can create a simple XSL stylesheet in order to do XSL transformations. In the course of doing this, you will become familiar with the XSL Trace Editor and XSL Editor *content assist* features in Application Developer.

With regard to XSL in Application Developer, there is one frustrating short-coming you may run into. The Trace Editor does not like non-XML documents, which are not well-formed XML. This is only a problem when the transformation is generating a non-XML output structure.

If, when applying a transform, you encounter the error message "Hierarchy request error" it's likely that this is due to a well-formedness problem in the output.

Section 1 - Create a transform to turn XML into XHTML

In this part of the lab you will write a stylesheet to transform the exam.xml document into XHTML output. Refer to the screenshot above to get an idea of what the finished transformation should yield. It contains some elements that you won't insert until the second lab but you should use this screenshot as a guide for structuring output as you work through this exercise.

- ___ 1. Create the stylesheet file.
 - ___ a. In the Navigator view of the XML Perspective in Application Developer, Right-click the folder called **Lab 7 and 8 - XSLT**, and select **New > XSL** file.

Note: If you do not see a menu with XSL available, you need to reset your perspective to XML (you are probably in the Resource Perspective). Note: in most cases you can find a way to force V5 to do what you want: if you are in the Resource Perspective you could follow this path

New'Other...'XML' XSL'Next

In any event, name the new file as in the next step.

- ___ b. In the Create XSL File dialog box, ensure that **Lab 7 and 8 - XSLT** is selected in the folder list and change the filename to **exam_html.xsl**.
- ___ c. Click Finish. Application Developer will open your new XSL file in the XSL editor.
- ___ d. Application Developer knows to add the XML declaration and the `xsl:stylesheet` element with the *namespace prefix definition* appropriate to 1.0 to the file. It should look like this (after inserting line-feeds to make it more readable and adding space for our "code"):

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
  xmlns:xalan="http://xml.apache.org/xslt">

</xsl:stylesheet>
```

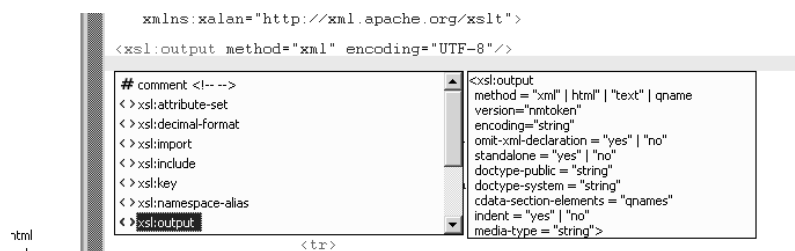
Diagram illustrating the structure of the XSL stylesheet with callouts:

- Best practices** points to the XML declaration: `<?xml version="1.0" encoding="UTF-8"?>`
- xsl: prefix needs to be assigned to a namespace*** points to the `xsl:stylesheet` element.
- Not used; delete if you wish**** points to the `xmlns:xalan="http://xml.apache.org/xslt"` attribute.

___ 2. Choose the output format you desire.*

Note: The default output method used by the XSL processor in Application Developer is HTML. If you wish to view the results of changes to the transform you are building in a browser, do nothing. For demonstration purposes, we will set the output to be xml. When you have seen how its presence affects the outputs - and the process you must follow -- you may remove the statement or convert it to a comment. Here are specific instructions for using the content assist capabilities of V5 to create the correct output tag.

- ___ a. Place your cursor above the stylesheet end tag and use Studio's *content assist* feature (<Ctrl><spacebar> to open Studio's menu of allowable choices.
- ___ b. Select < > xsl:output to open the box on the right, below.



The choices are xml, html, text, or you can define your own choice as long as your choice satisfies the XML specification (qname). The details of usage can be found in Studio Help'Help Contents by searching for xsl:output.

- ___ c. Double-click to add it to your file:
- ```
<xsl:output/>
```
- \_\_\_ d. Insert some spaces before the /> and use content-assist to insert
- ```
method="html"
```
- ___ e. Add some space between html" and /> and use content assist to insert **encoding=" "**; insert **UTF-8** to complete the statement:
- ```
<xsl:output method="html" encoding="UTF-8" />
```

\*Refer to <http://www.w3.org/TR/xslt#xslt-namespace> for additional information. The URI specified for the xsl prefix is required for any XSLT. The 1999 indicates the year in which the URI was allocated by the W3C; it does not indicate the version of the XSLT being used; the version attribute determines the version (see the referenced web-site for values other than 1.0). XSLT stylesheets are free to use any prefix not just xsl but the namespace declaration must bind that prefix to the URI of the XSLT namespace above.

- \_\_\_ 3. We begin by defining a template:<sup>1</sup>
  - \_\_\_ a. Place your cursor to the left of the < for </xsl:stylesheet> and use the **enter** key to insert a blank line.
  - \_\_\_ b. Place your cursor in the blank line and enter <Ctrl><spacebar>
  - \_\_\_ c. From the pop-up menu scroll down and double-click "< > **xsl:template**" to add <xsl:template></xsl:template> to your file.
  - \_\_\_ d. We will be using the "match=/" template:<sup>2</sup> add a space between the 'e' and the '>' and use <Ctrl><spacebar> to bring up a menu containing match and double-click to select it; inside the "" add the / character to produce the result <xsl:template match="/">.
  - \_\_\_ e. Insert some empty lines between the start and end tag and you are ready to proceed!
- \_\_\_ 4. Inside the xsl:template you just defined, insert the following HTML markup.<sup>3</sup> This will form the skeleton of the final transform. To minimize your aggravation (that is, typing errors) you might want to copy lines from the **Startup Files** folder. The code below is in the **Step 1\_4** file. [Note: the grayed-out lines are those already in your new exam.xsl file from 3., above.]

\*Refer to <http://xml.apache.org/xalan-j/extensions.html> for more information. Xalan-Java supports the introduction and use of extension elements and functions including calls to a procedural language. The *current* xalan namespace is **xm:ns:xalan="http://xml.apache.org/xalan "** the **.../xslt** is still supported for backward compatibility.

<sup>1</sup> Conceptually the thing that marks a file as an XSLT file is that it consists of a set of template rules, each of which describes how a particular element type or other construct should be processed. The rules are not arranged in any particular order; they don't have to match the order of the input or the order of the output, and in fact, looking at someone's XSLT script there are few clues as to what ordering or nesting of elements the stylesheet author expects to encounter in the source document. It is this that makes XSLT a declarative language: you say what output should be produced when particular patterns occur in the input, as distinct from a procedural program where you have to say what tasks to perform in what order. It also implies we need to include comments to explain our choices.

<sup>2</sup> / is the widest possible match for a document: it indicates the root of the tree is to be matched.

<sup>3</sup> Forget your HTML? Try [http://hotwired.lycos.com/webmonkey/reference/html\\_cheatsheet/](http://hotwired.lycos.com/webmonkey/reference/html_cheatsheet/) for help.

```

<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:output method="xml" encoding="UTF-8" />
<xsl:template match="/">

<html>
 <head>
 <title>Technical Certification Exam Tool</title>
 </head>
 <body>
 <form action="/ILSCS01/ExamForm" method="post">4
 <table>
 <tbody>

 <tr>
 <td align="center" bgcolor="lightblue">
 <h2>
 Exam ID:
 <!-- Location 1 -->

 </h2>
 </td>
 </tr>
 <!-- Location 2 -->

 </tbody>
 </table>
 </form>
 </body>
</html>

</xsl:template>
</xsl:stylesheet>

```

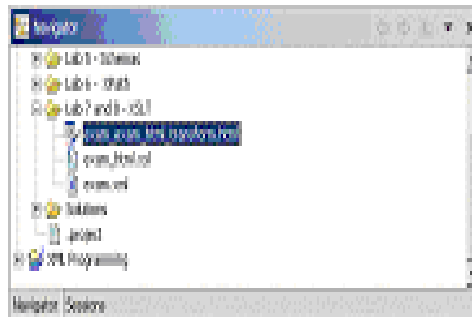
\_\_\_ 5. Save your changes to **exam\_html.xsl**. The file should be error free.

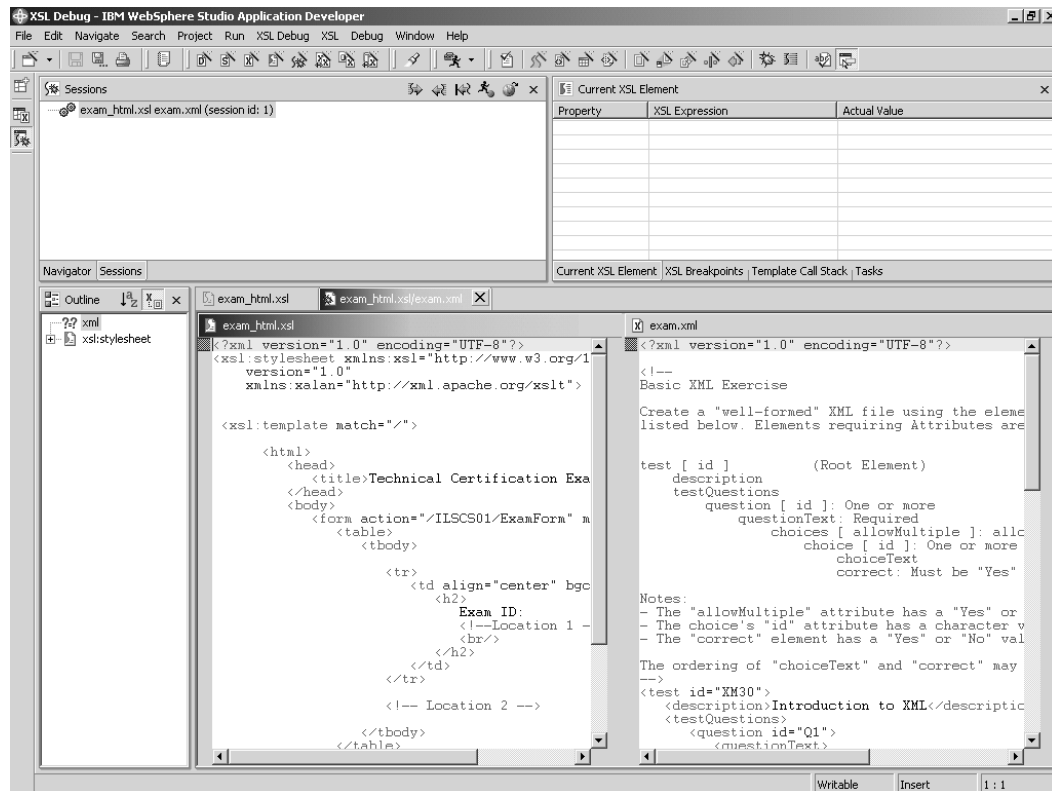
<sup>4</sup> Form has one required attribute, action, which specifies the URI of a CGI script that processes the form and returns feedback. There are two methods to send form data to a server. GET, the default, will send the form input in a URI, whereas POST sends it in the body of the submission. The latter method lets you send larger amounts of data, and that the URI of the form results doesn't show the encoded form.

## Section 2 - Test your XSLT and Trace the Transformation

In this section, you will use the XSL Trace Editor in Application Developer to apply your XSL transform to the **exam.xml** file and view the results as HTML.

- \_\_\_ 1. Select both **exam\_html.xsl** and **exam.xml**, then apply the XSL as HTML. (It's OK if you apply as XML but the output will not be renderable in a browser; it will be appropriately indented and will resemble any other XML instance.) Here's how:
  - \_\_\_ a. In the Navigator view, select both the **exam.xml** and **exam\_html.xsl** files. (Hold down the <Ctrl> key and click each file.)
  - \_\_\_ b. Right-click the selected files and select **Transform>Debug** from near the bottom of the context menu.
- \_\_\_ 2. The XSL Debug Perspective will open (as shown on the next page), containing your two input files each in an editor pane.
  - \_\_\_ a. The top left pane is labeled **Sessions**; it shares its location with **Navigator**.
  - \_\_\_ b. Click **Navigator**: observe a new file, a transformation file representing the concatenation of exam.xml and exam\_html.xsl, it is of type .html because that is what you selected!





- \_\_\_ 3. Click the **Sessions** tab to return it to the foreground.
  - \_\_\_ a. Observe the "world" symbol to the right of the "running man" in the Sessions tool bar.
  - \_\_\_ b. Click it and a browser will open showing any renderable parts of your newly created .html file:

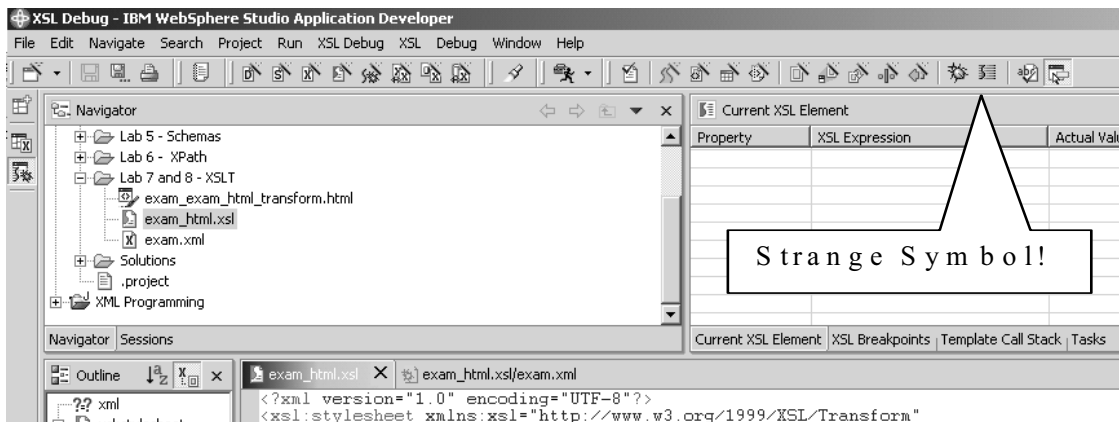


- \_\_\_ c. Close the **Web Browser** to return to the exam\_html.xml file.

- \_\_\_ 4. Analyze the current state.
- \_\_\_ a. Observe the tool bar that becomes available when the exam\_html.xsl editor is active. Hover over the strange symbol to the right of the "bug" icon (3rd icon from the right)

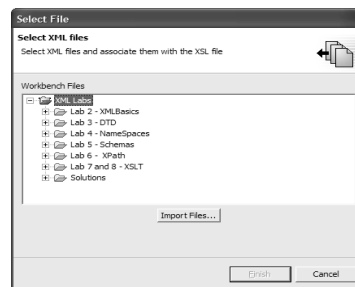


it should say **Run transformation on the XSL**. See the snippet below for the context to follow this step:



**Notes:** 1) If your picture matches ours but you do not have the icons shown on the previous page, close WebSphere and reopen it; repeat steps 1, 2, and 3; you should now see the full tool bar. 2) If you have the icons but they are not active (grayed-out), it is because the editor pane is not active: click either file name **exam\_html.xsl** or **exam\_html.xsl/exam.xml** to activate these icons. 3) Icon order is **NOT** guaranteed.

- \_\_\_ b. Change the content of <h2> from Exam ID: to anything else, say, Exam 21; save the modified file and select the **Run transformation on the XSL** icon. The first time you do this a **Select File** window pops up in which you will select the XML file to associate with this .xsl file.






- \_\_\_ c. Select **exam.xml** from the **Lab 7 and 8 - XSLT** folder and press **Finish**. You should now see "Exam ID:" replaced by whatever you entered and displayed in a Web Browser that opened automatically. If you chose to use the 'output method = "xml"' you will see an XML file consisting of HTML tags. The only way you can



tell it's an HTML file is the declaration and the file extension in the URI. (If you do not see the new value, did you Save?) Switch to **Navigator** view and you will see a new file **exam\_exam\_html\_transform.xml** which has the .xml extension called for by the **xsl.output** statement we added above.

- \_\_\_ d. Notice, however, that selecting the globe in the Sessions toolbar produces a rendering with the original Exam ID: value! This is because we have not updated the session. If we want to display the revised file as HTML it is necessary to context-click on the session and select **Relaunch** from the menu. The session id will increase by one.

**Note:** If you want to keep a record of how your changes affect the output it is necessary to return to the Navigator window, reselect the .xml / .xsl pair and re-apply the **Transform>Debug** described in Step 1. This will create an additional session using the current, saved version of the .xsl file.

- \_\_\_ 5. **Tracing a transform.** In version 4, it was necessary to re-initiate the entire transform process choosing **Apply XSL As XML** instead of **As HTML**. Now we simply select the **Sessions** tab which activates the debug trace controls in the **Sessions** toolbar. (See previous page for a snapshot.)
  - \_\_\_ a. Hover over each of the buttons to identify their purpose; note the blue-green (?) ball associated with the **Run to breakpoint** icon (to the left of the Globe icon).
  - \_\_\_ b. Click the **Restart from the beginning** button  on the toolbar to reset the transform processor. Notice that an editor with the combination of the .xsl and .xml files is automatically brought into focus. The top line in each editor sub-pane will become highlighted and the **Current XSL Element** pane (to the right of the Navigator/Sessions pane) will display the properties associated with the current step at the upper left.
  - \_\_\_ c. Now you can use the forward  and back  trace buttons to watch as the instructions in your XSL file execute against the highlighted line in the Input XML to generate the highlighted line in the Output XML. Set a breakpoint at any line in the .xsl subpane by double-clicking in the vertical slider bar at the left of the panel. Click the **Run to breakpoint** icon and the transform will execute up to that point and pause. You can set many breakpoints; you can review what you have set by selecting the **XSL Breakpoints** tab instead of the **Current XSL Element** tab.
  - \_\_\_ d. You can restart the transform at anytime you wish.

However, restarting the transform does NOT regenerate a new HTML file. You will need to either Relaunch the Session or re-apply the Transform by selecting the appropriate XML/XSL file pair.

### Section 3 - Add the Exam Id to the output

- \_\_\_ 1. Locate the **Location 1** comment. Replace this comment with XSLT code to insert the value of the test's id, the value of the test's description, and place a dash between the two values. Here's how:
  - \_\_\_ a. Immediately after the "Location 1" comment line, insert **<xsl:value-of** content assist cannot help you with this quite yet so manually type it.
  - \_\_\_ b. Now use content assist to insert **select=""** and complete the statement using content assist to add the close tag. You will receive an error because an attribute value is required for **select**.
  - \_\_\_ c. Place your cursor inside the "" and from the menu just below the title bar select **XSL->XPath Expression** to open the XPath Expression wizard. Select **#document** and **Next** to get to a picture that should be familiar to you from the last exercise. Select **test** followed by **id** to produce **./test/@id** and **Execute** to produce **XM30** in the **XPath Query Result** pane. This, of course, is what we want to appear in the final rendering. Close the Result pane and select **Finish** in the Wizard to *automatically* insert **./test/@id** in our XSLT file. [Note: **/** and **./** work equally well; since the Wizard likes the latter better, who are we to argue?]
  - \_\_\_ d. Let's do something similar to insert the value of the test's description. Steps a. and b. are the same. This time we wish the XPath Expression Wizard to return the text associated with the test, itself. We again start by selecting **#document** and **Next**. We need **test** followed by **description** but we only want the text so we can select **text()** from the **Node Test** subpane. **Execute** the expression to obtain **Introduction to XML** and use finish to insert the expression into our XSLT file to provide the attribute value for select. Note: the school solution uses **/test/description**, which in the XPath Wizard output produces **<description>Introduction to XML</description>**. This, too, renders as **Introduction to XML**. Why?
  - \_\_\_ e. If we were to stop here, we would render **XM30 Introduction to XML**, which is OK except it needs something between the 0 and the I to make it readable. Let's insert a dash between the two selection statements to make it pretty.

Our solution looked like this:

<!-- Location 1 -->

This single comment line will be replaced (or followed, if you want to try this again at home) by these three lines:

```
<xsl:value-of select="./test/@id"/>
-
<xsl:value-of select="./test/description/text()"/>
```

3<sup>rd</sup> line!

\_\_\_ 2. Save your changes and test your XSL.

- \_\_\_ a. By clicking the **Run transformation on the XSL** icon in **XSL Trace Editor** in the **XSL Debug** perspective. The result should be a Web Browser pane showing raw HTML with the extracted values;
- \_\_\_ b. By **Relaunching** the Session and clicking the Globe in the Sessions tool bar to produce a blue box with the text **Exam ID: XM30 - Introduction to XML** (see below).

Exam ID: XM30 - Introduction to XML

**Note:** if you close Studio for any reason, your Sessions will be lost. You will need to again select both the XML file and the XSLT file and Apply XSL 'As HTML' to generate the picture above.

## Section 4 - Add the Exam Questions to the Output

Let's pause in order to take a step back to gain perspective. Look at the HTML code we have and what it has done:

```
<table>
 <tbody>

 <tr>
 <td align="center" bgcolor="lightblue">
 <h2>
 Exam ID:
 <!-- Location 1 -->
 <xsl:value-of select="./test/@id"/>
 -
 <xsl:value-of select="./test/description/text()"/>

 </h2>
 </td>
 </tr>
```

We are at "<!--Location 2 -->". At this point we have constructed the first row in a table, which when completed in the next lab, will render what appears to be an exam. (Although a two question exam is rather thin!) The approach we choose is to create a template that will populate each row (consisting of a single cell) for a single question such as this one:

Q2 ) What are valid XSLT intructions?

- ☐ A . <xsl:choose
- ☐ B . <xsl:define
- ☐ C . <xsl:copy
- ☐ D . <xsl:deflate
- ☐ E . <xsl:text

Multiple Select Question - Please select all correct answers.

Referring back to the original XML file we want to repetitively apply our template to each occurrence of the **question** tag. So we will substitute an apply-templates statement that collects the question nodes, in natural order of occurrence. We will then begin construction of the template to be applied to each occurrence of a **question** element. Here's how:

- \_\_\_ 1. Add the statement to define the content of the rest of the table.
  - \_\_\_ a. After the Location 2 comment begin an **xsl:apply-templates** tag.
  - \_\_\_ b. We want to collect all the question elements with their content and children. Use content assist to continue the <xsl:apply-templates by adding select = "node-set". Remember to add an end tag.
  - \_\_\_ c. Highlight node( ) and use Xpath Expression (menu bar: **XSL->Xpath Expression**) to find the expression that returns the "node set" question elements (and their baggage).
  - \_\_\_ d. An answer is <xsl:apply-templates select="./test/testQuestions/node( )"/>. **Note:** While this may satisfy our craving for *rigor*, it does *not* help us make the connection to Section 5 (below); let us use <xsl:apply-templates select="//question"/> instead, which produces the same result but is more intuitive in making the connection between a **select** and its corresponding **match** template. Alternatively, insert a comment before or after the ...node( ) statement indicating that the node-set consists of **question** elements and, perhaps, pointing to the corresponding **question** template.
  - \_\_\_ e. At this point we can complete the HTML table by adding these end tags.
 

```
</tbody>
</table>
```
- \_\_\_ 2. Save your changes and test your XSL as you did in Section 2. The result should still be a blue box with the text Exam ID: XM30 - Introduction to XML but now you have question-related text atop the blue box. This simply reflects the necessity to define what the **question template** should do.

## Section 5 - Define the "question" Template

Refer back to Section 1. We can complete the statements that define the 1st template we created. They are already present:

```

 </form>
 </body>
 </html>
 </xsl:template>

```

- \_\_\_ 1. Open some space between the existing template end tag and the stylesheet end tag.
- \_\_\_ 2. Define a new xsl:template that matches a **question** element. Your template should look like:

```

<xsl:template match="question">
</xsl:template>

```

The last nine lines of the .xsl file should now look like this (with the pre-existing lines grayed-out):

```

 </html>

 </xsl:template>

 <xsl:template match="question">

 </xsl:template>

 </xsl:stylesheet>

```

- \_\_\_ 3. Save your changes and test your XSL as you did in Section 2. The result should still be a blue box with the text Exam ID: XM30 - Introduction to XML and now the random text atop the blue box is gone. We have ingested the information, but we have not defined our output.
- \_\_\_ 4. Let us begin to add output from this template. Add the following code to the xsl:template element you created in step 2 (the template element is shown grayed-out) or copy it from the **Step 5\_4** file in the **Startup Files** folder:

```

<xsl:template match="question">

<tr>
 <td>
 <!-- Location 3 -->
 <hr/>
 <!-- Location 4 -->
 <tr>

```

```

 <td bgcolor="lightblue">

 <!-- Location 5 -->

 </td>
</tr>
</td>
</tr>

</xsl:template>

```

- \_\_\_ 5. Rerun your file (**Relaunch** your Session). You will now see a row (tr) consisting of a cell (td) with a "hard rule" (hr) (which defaults to black) followed by an embedded row (tr) with an embedded cell (td) with a light blue line (which will become a blue background as soon as it becomes an area):

**Exam ID: XM30 - Introduction to XML**

---



---



---



---

- \_\_\_ 6. Locate the **Location 3** comment. Replace this comment with XSL code to print the **id** of the question, followed by a ), followed by the content of the **questionText** element.

Your xsl should look like:

```

<xsl:value-of select="@id"/>
)
<xsl:value-of select="questionText/text()" />

```

**Note:** The XPath expression wizard will lead you astray: in the **id** case it prefers `./question/@id` and in the text case it prefers `./question/questionText/text()`. Since we have established **question** as the current node via the `match =` statement the `./question/` misleads the engine.

- \_\_\_ 7. Test your XSL. Now you should see the blue box and under it should be another box containing Q1 and Q2 and the corresponding text:

**Exam ID: XM30 - Introduction to XML**

Q1 ) How can XML documents be transformed?

---

Q2 ) What are valid XSLT intructions?

---



---

This brings you to the end of the first XSLT Lab. You'll finish this work in the next Lab.

## Additional reading

XSL transforms (XSLTs) provide a powerful toolset. This lab has been a most humble introduction.

The Internet is loaded with additional information. One example is a tutorial at [http://www.topxml.com/xsl/articles/xslt\\_what\\_about/31290106.asp](http://www.topxml.com/xsl/articles/xslt_what_about/31290106.asp)

Many books have been written at all levels of detail and subject area; one quick, overview is *Sams Teach Yourself XML in 24 Hours*, 2nd edition, © 2002 by Sams Publishing, ISBN 0-672-32213-7.

And, of course, there are the IBM Web sites, Redbooks, and so forth.

## **END OF LAB**





## Exercise 8. XSLT Part 2 - Conditional XSL Transforms

### What This Exercise Is About

XSLT is a language for defining transforms (also called stylesheets) that ingest XML documents and generate other markup structures from them, including XHTML. XSLT uses XPath expressions to extract information from the source documents. WebSphere Studio Application Developer contains Xalan, the XSLT processor implementation developed by the Apache Project.

This Lab is the second in a series of two labs in which you define a transform that consumes an XML document containing the content of an Exam. The transform generates a Web-based user interface through which the exam questions may be answered. You will use WebSphere Studio Application Developer to create the transform and apply it to the XML source document.

This lab contains instructions for completing the project begun in Exercise 7.

### What You Should Be Able to Do

At the end of the lab, you should be able to:

- Identify and understand the means by which an XML document is transformed
- Complete a simple XSL stylesheet
- Use WebSphere Application Developer version 5 to transform XML documents into HTML and XML
- Use WebSphere Application Developer version 5 to trace execution of a transformation

### Required Materials

- **exam.xml**, an XML document containing the content of an exam.
- **exam\_html.xsl**, the XSL transform file that was created in the 1st part of this lab.
- WebSphere Studio Application Developer 5.x.

## Applicability

- XML301
- XML341

## Exercise Instructions

The goal of this project, is to transform the exam.xml document into the Web page shown below.

**Exam ID: XM30 - Introduction to XML**

Q1) How can XML documents be transformed?

☐ A. XPath

☐ B. XSLT

☐ C. Notepad

☐ D. Xalan

☐ E. Zeus

Single Select Question - Please select the best answer.

Q2) What are valid XSLT instructions?

☐ A. <xsl:choose

☐ B. <xsl:define

☐ C. <xsl:copy

☐ D. <xsl:date

☐ E. <xsl:text

Multiple Select Question - Please select all correct answers.

In Exercise 7 we started the template that populates each question. We did it by inserting an apply templates "command" which is intended to be applied to each question element. We positioned this in an HTML table following the exam title.

We then defined a template to populate the XHTML file we are creating. It defined additional rows and cells for the HTML table. We began to flesh out the **question** template by capturing the question number attribute and appending a hard-coded ')' followed by the text of the question.

We want to continue the definition of the **question** template partly to demonstrate how we can nest templates and their specifications. We have yet to capture the choice information (**B** and **C**), the possibility of multiple correct answers (**D**), and the proper answer-selection-mechanism, radio button or checkbox (**A**).

Try not to become bogged down in the *minutiae* of the statements; focus, instead, on how we create building blocks and employ them. We have copied and expanded on the **question** template to show what we mean:

```
<xsl:template match="question">
<tr>
 <td>
 <!-- Location 3 This is Q#) + the text of the Question -->
 <xsl:value-of select="@id"/>
)
</td>
</tr>
</template>
```

```
<xsl:value-of select="questionText/text()" />
<hr/>
<!-- Location 4 Where we will put the choice stuff B and C -->
<tr>
 <td bgcolor="lightblue">
 <!-- Location 5 Blue box, D, where we will put the
single/multiple choice text, the value of which controls A -->
 </td>
 </tr>
</td>
</tr>
</xsl:template>
```

If you are not already there, open WebSphere Studio to the XSL Debug perspective using either the perspective wizard in the left-hand vertical bar or using Window 'Open Perspective' XSL Debug.

## Section 1 - In-line Conditional XSLT Processing

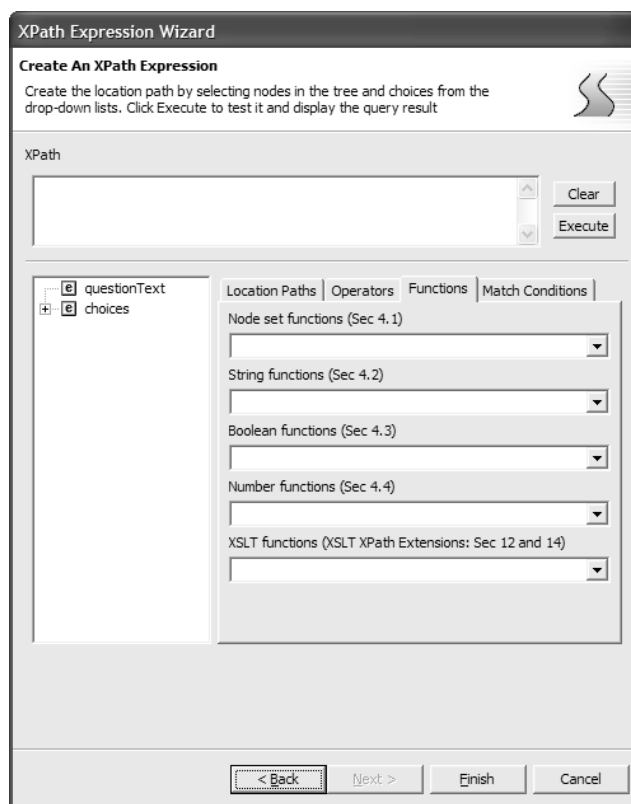
Since order is *not* important, let us start with **Location 5**, which represents an ideal opportunity (because of the complexity it involves) to create and employ a nested template. However, we are not forced to use a template: we can put our script *inline*. The decision should generally be based on *understandability* and *maintainability*. We will do it inline to demonstrate. We will do this inside the existing template for question.

- \_\_\_ 1. If you have not already done so, access the exam.xml file associated with Lab 7 and 8 - XSLT.
  - \_\_\_ a. Double-click exam.xml to open it in the editor pane.
  - \_\_\_ b. Use the Outline view, which is synchronized with the editor, to step through the tree to find where the possible number of correct outcomes is established.
  - \_\_\_ c. You see that the **choices** element, which we identified as a grouping element in the first exercise, contains the "allowMultiple" attribute; and the attribute values are "Yes" and "No".
- \_\_\_ 2. Open or return to your .xsl transform file. Pick the XSLT statement you wish to employ at Location 5; we will use **if**, placing the proper choice as content of the tag:
  - \_\_\_ a. Before we can use content assist we have to supply enough information for content assist to deploy. Begin typing `<xsl:if + <blank>` at which point content assist may be used to provide the `test=""`. Which prompts us for a Boolean expression.

```
<xsl:if test=""
```
  - \_\_\_ b. We can return to content assist to add the remainder of the statement so that we have

```
<xsl:if test="" ></xsl:if>
```

- \_\_\_ c. We can again employ XSL' Xpath Expression to open that wizard and select the path and conditions. We want to start at question in the opening window; *that is*, set the context to question.
- \_\_\_ d. Click Next to display a tree of **questionText** and **choices** in the next window:



Select **choices** and expand it to allow selection of **allowMultiple**.

- \_\_\_ e. Use Operators and Booleans (Sec 3.4) to insert an = sign.
- \_\_\_ f. Finish by adding "Yes" so we have  
`./choices/@allowMultiple="Yes"`
- \_\_\_ g. Execute your expression; you should receive "False". Click Close. Click **Finish** to insert the expression into the test=" ". If you save you will receive an error because of the confusion caused by the consecutive quotes; change one of the pairs to an apostrophe ' and try again.
- \_\_\_ h. Add the content Multiple Select Question - Please select all correct answers.
- \_\_\_ i. Copy and paste the same pattern changing "Yes" to "No" and Multiple Select ... to Single Select and process the result. Your code should look similar to this:

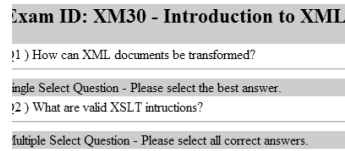
```
<!-- Location 5 Blue box, D, where we will put the single/multiple
choice text, the value of which controls A -->
<xsl:if test='./choices/@allowMultiple="Yes"'>Multiple Select
Question - Please select all correct answers.</xsl:if>
```

```

 <xsl:if test='./choices/@allowMultiple="No"'>Single Select Question
- Please select the best answer.</xsl:if>
 <!--Location 5 End -->

```

and your output, like this:



## Section 2 - Nesting Templates - Dealing with Choice

We will embed a second apply-templates at Location 4 to deal with callouts A, B, and C (above). This template will apply to each choice. Since we have decided to also address the pre-pending of a radio button or checkbox but the decision criteria is above the current node, we will have to adjust our path expression to deal with it.

\_\_\_ 1. Open exam.xml and locate where we need to be. . .given that we are already at the **question** node by virtue of the template we are already in. Again, we recommend using XSL'XPath Expression starting at **question** but we may (as we did for **question**) want to tinker with our answer because that "node( )" is the **choice** set is not obvious and may make it more difficult to read our file.

\_\_\_ a. A correct answer is

```

<!-- Location 4 Where we will put the choice stuff B and C -->
<xsl:apply-templates select="./choices/node()" />
<!-- Location 4 End The node set node() holds consists of choice
elements -->

```

\_\_\_ b. But, this answer also works and is easier to relate to the template to which it refers:

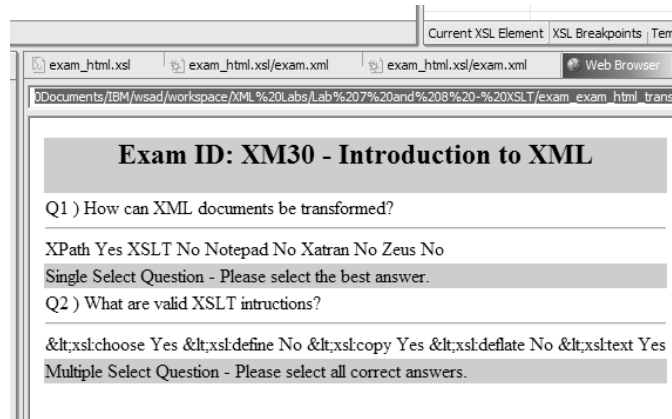
```

<!-- Location 4 Where we will put the choice stuff B and C -->
<xsl:apply-templates select="choices/choice"/>1
<!-- Location 4 End The node set node() holds consists of choice
elements -->

```

\_\_\_ c. Relaunch the session to produce this result:

<sup>1</sup> ./choices/choice also works. Why?



### Section 3 - Create the choice Template

In this lab you will create the template to insert the remaining elements to transform the **exam.xml** document into XHTML output. Refer to the screenshot in the Instructions to view the finished transformation. We have three tasks to accomplish, two are easy, one is a little more complicated and will be dealt with in Section 4. Here's how:

- \_\_\_ 1. Begin by creating a template for the **choice** element. We can put templates anywhere; let's put this one just above the stylesheet end tag. Here is our solution:

```
<xsl:template match="choice"></xsl:template>
```

Remember, match in the template has placed us at the **choice** node; our context is the **choice** element.

- \_\_\_ 2. Next let us do the easy tasks: **B** and **C** in the callouts. These involve selecting elements and embedding them. Peruse the exam.xml file to see where to expect the two items: **choice ID** and **choice** text. Using the synchronization feature of Studio it is easy to see that the **choice** id is the **ID** attribute of **choice**; similarly, the **choiceText** element contains the **choice** text. What construct could we use to extract the value of these two elements? How about some form of `<xsl:value-of` ? We can use Studio to suggest possibilities. Here's how:
  - \_\_\_ a. Although order does not matter, let us embed the question ID first.
  - \_\_\_ b. Begin by placing your cursor in the content of the match="choice" element we added in Step 1 and insert some blank lines to provide some space for subsequent entries.
  - \_\_\_ c. Use *content/code* assist and pick the **<xsl:value-of** template; while you're there, insert the **select=""** option.
  - \_\_\_ d. We are already at choice so the attribute value of select is **@id**.

The complete statement is **<xsl:value-of select="@id"/>**, which, if you were to test it would produce **ABCDE**:

**Exam ID: XM30 - Introduction to XML**

Q1 ) How can XML documents be transformed?

ABCDE

Single Select Question - Please select the best answer.

Q2 ) What are valid XSLT intructions?

ABCDE

Multiple Select Question - Please select all correct answers.

- \_\_\_ e. Which is nice but we want these to be one **ID** per line. The solution is to insert a break tag **<br/>**. . . and while we're at it, we may as well insert the . after the choice letter. Our more complete solution (with a comment to indicate what we are doing) is

```
<xsl:template match="choice">
 <!-- Embed the choice id -->
 <xsl:value-of select="@id"/>
 .2

</xsl:template>
```

You can Relaunch the Session but by now you should know what to expect.

- \_\_\_ f. Let us complete this task by adding a statement to append the choice text to the choice id. Once again we shall use value-of select. We place it before the **break** tag because it is part of the line. Our solution is this: `<xsl:value-of select="./choiceText/text()" />`<sup>3</sup>

Relaunching the Session produces this picture:

<sup>2</sup> We changed the font size to make it easier for you to see it: it's still just a **period**.

<sup>3</sup> We constructed the attribute using XSL/XPath Expression starting at the choice node. We could have simply used choiceText and relied on the browser to ignore extraneous information but a later browser version might render it in some *unfortunate* way.



**Exam ID: XM30 - Introduction to XML**

Q1 ) How can XML documents be transformed?

- A . XPath
- B . XSLT
- C . Notepad
- D . Xatran
- E . Zeus

Single Select Question - Please select the best answer.

Q2 ) What are valid XSLT intructions?

- A . <lt;xsl:choose
- B . <lt;xsl:define
- C . <lt;xsl:copy
- D . <lt;xsl:deflate
- E . <lt;xsl:text

Multiple Select Question - Please select all correct answers.

- \_\_\_ 3. Except for the radio buttons / check boxes and the **&lt;** we are *essentially* finished. We have added comments to the **choice template** to explain what we are doing. Our solution at this point looks like this (with a comment showing where we want to select between a check box and a radio button):

```
<xsl:template match="choice">

 <!-- This is where the radio button or checkbox should be -->

 <!-- Embed the choice id -->
 <xsl:value-of select="@id"/>
 <!-- Insert a . after the letter ID -->
 .
 <!-- Add the text for this choice -->
 <xsl:value-of select="./choiceText/text()" />
 <!-- Add a break to separate this line from the next -->
 <!-- so the lines appear in vertical order -->

</xsl:template>
```

As you can surmise, this is an important block in terms of constructing the picture we wish to display.

We defer to Section 6 for guidance on processing entities of the form **&xx;**.

## Section 4 - Using the Variable Template

- \_\_\_ 1. The radio buttons/checkboxes (callout A in the figure much earlier) need to precede each choice. We will use a simple form of the **variable**<sup>4</sup> template to do this:
- \_\_\_ a. Change the "This is..." comment to be more meaningful. Perhaps,

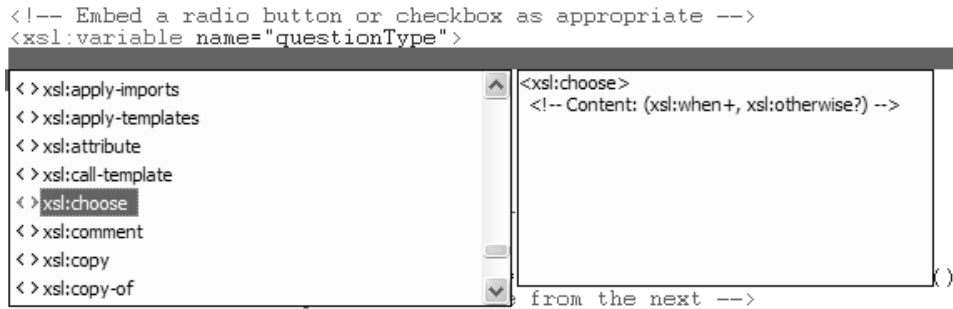
<sup>4</sup> Refer to your notes (if you are not already doing so) to find descriptions/additional information of/for these items

<!-- Embed a radio button or checkbox as appropriate -->

- \_\_\_ b. Using content assist (if you wish), insert a variable template after the comment; for a name we chose "questionType" to attempt to capture the radio button/checkbox possibilities:

```
<xsl:variable name="questionType"></xsl:variable>
```

- \_\_\_ 2. **questionType** has the value **checkbox** if the question is a multiple select, or the value **radio** if the question is a *single* select. We will use the **choose** element; content assist tells us this:



The notation is consistent with DTD notation (for example) where + means one or more, ? means optional, and order is as stated. See your notes for additional information.

- \_\_\_ a. Using content assist, our solution (after using **Format->Document** from the context menu in the Editor pane) looks like this:

```
<xsl:variable name="questionType">

 <xsl:choose>
 <xsl:when test=""></xsl:when>
 <xsl:otherwise></xsl:otherwise>
 </xsl:choose>

</xsl:variable>
```

- \_\_\_ b. We need to define the **test** attribute and supply content for these elements.

```
<xsl:variable name="questionType">
 <xsl:choose>
 <xsl:when test='../@allowMultiple="No"'>radio</xsl:when>
 <xsl:otherwise>checkbox</xsl:otherwise>
 </xsl:choose>
</xsl:variable>
```

## Section 5 - Create the Data Entry field for Form

Maybe this is where we should have started! This is the mechanism by which we input the test-taker's choices into the HTML **form** element. We shall use the **input** HTML tag. A complete description of the 30 or so fields possible with this tag is beyond the scope of this exercise.<sup>5</sup> We only need the following fields, which were generated using code assist:

```
<input type="text" name="" value=""/>
```

You can use the hover feature of Studio to see the defaults for each of these fields.

- \_\_\_ 1. The type attribute value is determined by the outcome of the **variable** template; since the value must be one of the default choices or a member of an ENUM, the syntax is:

```
{ $questionType }
```

where "questionType" is set by **choose** in the **variable** template.

- \_\_\_ 2. The **name** attribute value is the question number the test taker is answering; looking at the exam.xml file, since we are at choice we need to back up the tree two levels to get to the **question** element, which has the **ID** attribute. You can use XSL'XPath Expression provided you select the **choice** element for your context node before typing in the expression:

```
../../@id
```

- \_\_\_ 3. The **value** attribute determines the value to be passed. With the same node position caveat as in b., above, here is one solution:

@id returns the letter of the choice, which will require processing by the CGI script that will have to somehow access an answer key; perhaps a simpler solution might be to return the yes/no flag that identifies the correctness of this answer:

./correct/ returns Yes or No depending on whether this choice is correct or not. The following snippet is to help you locate where this completed line belongs:

```
<xsl:template match="choice">
 <xsl:variable name="questionType">
 <xsl:choose>
 <xsl:when test='../../@allowMultiple="No"'>radio</xsl:when>
 <xsl:otherwise>checkbox</xsl:otherwise>
 </xsl:choose>
 </xsl:variable>

 <input type="{ $questionType }" name="{ ../../@id } " value="{ @id } " />

 <xsl:value-of select='@id' />
 .
<!--
 <xsl:value-of select='choiceText' />
```

<sup>5</sup> See, for example, [http://www.htmlcodetutorial.com/forms/\\_INPUT.html](http://www.htmlcodetutorial.com/forms/_INPUT.html).

-->

```


</xsl:template>
```

\_\_\_ 4. Save the .xsl file and rerun the result:

#### Exam ID: XM30 - Introduction to XML

Q1 ) How can XML documents be transformed?

- ☐ A . XPath
- ☐ B . XSLT
- ☐ C . Notepad
- ☐ D . Xatran
- ☐ E . Zeus

Single Select Question - Please select the best answer.

Q2 ) What are valid XSLT intructions?

- ☐ A . &lt;xsl:choose
- ☐ B . &lt;xsl:define
- ☐ C . &lt;xsl:copy
- ☐ D . &lt;xsl:deflate
- ☐ E . &lt;xsl:text

Multiple Select Question - Please select all correct answers.

## Section 6 - Output Escaping

Do you see strange characters in your output? Try displaying the raw HTML using Internet Explorer. (**Hint:** begin a second instance of Studio; the path appears in the path window.) Are they still there? If so, locate the `<xsl:value-of select=` statement that writes the question choice text (about four lines from the end of the .xsl file) and use content assist to add `disable-output-escaping="no"` changing the "no" to "yes":

`<xsl:value-of disable-output-escaping="yes" select="choiceText" />` or our alternate form

`<xsl:value-of disable-output-escaping="yes" select="./choiceText/text()" />`

Rerun and re-launch to produce this "final" output:

#### Exam ID: XM30 - Introduction to XML

Q1 ) How can XML documents be transformed?

- ☐ A . XPath
- ☐ B . XSLT
- ☐ C . Notepad
- ☐ D . Xatran
- ☐ E . Zeus

Single Select Question - Please select the best answer.

Q2 ) What are valid XSLT intructions?

- ☐ A . <xsl:choose
- ☐ B . <xsl:define
- ☐ C . <xsl:copy
- ☐ D . <xsl:deflate
- ☐ E . <xsl:text

Multiple Select Question - Please select all correct answers.

Examine, if you wish, the exam\_exam\_html\_transform.xml/html files in Internet Explorer; try opening the source view. The rendered output translates the escape characters but the

underlying files maintain their XML well-formedness: the output structure truly mimics the input structure.

We have left one stone unturned: we have not addressed how you would "submit" the completed exam. See the reference in footnote (2), for example, for some cool ideas on how you might implement the submission process.

***END OF LAB***

## Addenda

### Addendum 1: Notes about Approaches:

There is more than one way to write the XSLT to generate the desired HTML. The two design choices you are most likely to consider are 1) use of **templates** versus **for-each** and 2) the alternate approach for constructing the INPUT element using `xsl:element` and `xsl:attribute`.

Our best practices recommendation is to use **template** whenever it makes sense; in our example it is very natural to have a **template** for each question and each choice output. We could replace the `apply-templates` calls with `for-each` loops which would enclose the contents of the template being called.

We choose to use the `xsl:variable` to hold the result of the `xsl:choose` for radio or checkbox and declare the INPUT element instead of using `xsl:element` and `xsl:attribute` primarily for simplicity. We felt it would be more understandable for the students. It also has them use `xsl:variable` which is an important construct. The use of `xsl:element` and `xsl:attribute` is not as common when converting to HTML as when converting to XML, but is an acceptable solution for the problem presented.

Below is the code for the template using `xsl:element` and `xsl:attribute`.

```
<xsl:element name="INPUT">
 <xsl:choose>
 <xsl:when test='parent::choices/@allowMultiple="No"'>
 <xsl:attribute name="type">radio</xsl:attribute>
 </xsl:when>
 <xsl:otherwise>
 <xsl:attribute name="type">checkbox</xsl:attribute>
 </xsl:otherwise>
 </xsl:choose>
 <xsl:attribute name="name">
 <xsl:value-of select="../../@id" />
 </xsl:attribute>
 <xsl:attribute name="value">
 <xsl:value-of select="@id" />
 </xsl:attribute>
</xsl:element>
```

## Addendum 2: Linking Your XSL to Your XML

### *Using WSAD*

We placed both the XML file to be processed and the .XSL file to be applied in the same directory.

We used the <Ctrl> key to select both and we used the **XSL Trace Editor** to create the result.

### *Outside WSAD*

What if you do not have a tool such as Application Developer readily available? Then what?

Simple! You will include a line like this

```
<?xml-stylesheet type="text/xsl" href="XSLT_filename.xsl"?>
```

at the beginning of your XML file to link your stylesheet to your .xml file; for this exercise the XSL Transform file is exam\_html.xsl and the source file is exam.xml. Here is our solution.

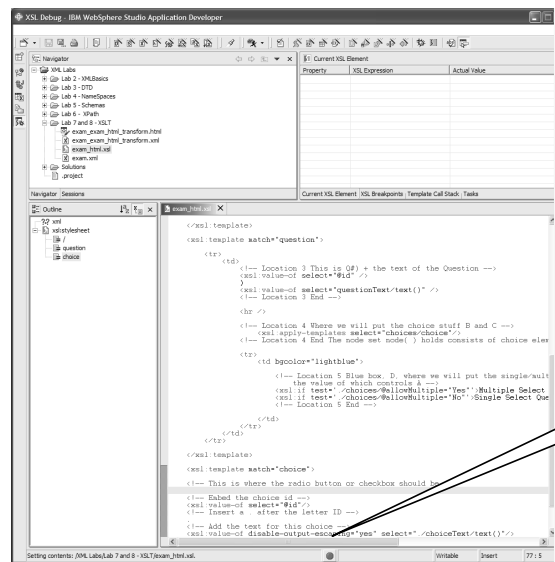
**exam.xml:**

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="exam_html.xsl"?>
<!-- big long comment -->
<test id="XM30">
.
.
.
</test>
<!-- End of File -->
```

For additional information refer to [http://www.w3schools.com/xsl/xsl\\_transformation.asp](http://www.w3schools.com/xsl/xsl_transformation.asp).

## Addendum 3: Understanding Studio 5.x

1. The very first time you try to use Help it will require several minutes to index its information.
2. When you open Studio and open a file in a new editor or save a change to an existing file you may experience a long pause during which nothing appears to be happening. In fact, Studio is performing internal processing. Instead of displaying an hourglass or similar icon, it puts a red ball at the bottom:



I'm working...  
please wait!

***END OF LAB***



## Exercise 9. SAX Parser Programming

### What This Exercise is About

In this exercise, you will put your knowledge of SAX programming to use by developing a real-world application.

You have been assigned the task of writing a program that will read in an XML document containing exam data and create an object graph of Javabeans based on the content. The Javabeans are provided. Your task focuses on parsing the XML data and setting the properties of the Javabeans based on the information found. Once the object graph has been created you will test your results by calling the "toString()" method of the Exam bean. The resulting String should contain all the XML data presented in a printable format.

The Javabeans, a sample application class, a simple base class, and a basic skeleton class have been provided to give you a starting point.

There are a total of three beans -- Exam, Question, and Choice. Exam holds information related to the entire exam and a Vector of Questions. The Question bean holds information for each Question, along with a Vector of Choices. The Choice bean contains information related to a single choice for a question.

### What You Should Be Able to Do

At the end of the lab, you should be able to:

- Analyze an XML document and map its content to Javabean attributes
- Develop a working SAX application to parse an XML document
- Load XML data into Java beans
- Verify the data was loaded correctly

### Required Materials

- IBM's WebSphere Studio Application Developer 5.0+
- XML Parser (*Xerces is included with Application Developer*)
- XML document and DTD of an online exam
- Prebuilt Java beans to hold exam data
- Prebuilt Application program, and base support class

## Applicability

- XM321 Exercise 2
- XM341 Exercise 9

## Exercise Instructions

### Section 1 - Analyze Source Documents and Target Java beans

In this section of the lab, you will examine the source Exam XML document, its associated vocabulary, and the Java bean components that will hold the final information. Knowing what you have will aid you greatly in developing the application.

The files that are specific to this lab are:

**data/sax\_exam.xml**,

**sax.lab.PrintExam.java**,

**sax.lab.SAXExamReader.java** and

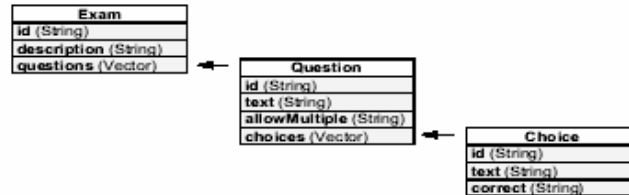
**sax.lab.SAXExamReaderBase.java**.

Depending on the order in which the labs are completed, you may have reviewed the others in a previous lab. If that is the case there is no need to review them again.

\_\_\_ 1. Open **Application Developer** and locate the following resources:

Resource	Location / File	Description
Project	XML Programming	Project location for all course resources
Source XML	<i>Folder</i> data/sax_exam.xml	XML containing Exam information to parse. Do <b>not</b> edit.
Source DTD	<i>Folder</i> data/test.dtd	DTD vocabulary for XML document. Do <b>not</b> edit.
	<i>Package Class</i>	
Target Java bean Classes	cs01 Exam.java cs01 Question.java cs01 Choice.java	Simple Java beans used to store exam data. Do <b>not</b> edit.
Application	sax.lab PrintExam.java	Simple runnable application to test your new SAX program.
SAX Base Class	sax.lab SAXExamReaderBase.java	Base class that implements DefaultHandler and provides simple error handler methods. Do <b>not</b> edit.
SAX Skeleton Class	sax.lab SAXExamReader.java	Empty program class that extends Base class. You will need to implement several methods in this class.

- \_\_\_ 2. Familiarize yourself with the XML source file and its associated DTD. These are the files that define the structure you will be processing.
- \_\_\_ 3. **Java bean classes:** The beans contain a simple hierarchical structure. Examine the three target java beans: you will find them in the **cs01** package under XML Programming:



\*Note: Do not modify the bean files

- \_\_\_ a. cs01.**Question.java** class
- \_\_\_ b. cs01.**Choice.java** class
- \_\_\_ c. cs01.**Question.java**
- Do not modify these files!**<sup>1</sup>
- \_\_\_ 4. **PrintExam.java** Configure and examine this *application* class: It contains a simple main() method to instantiate the SAXExamParser with a supplied file name to the Command Arguments of this class, which for this exercise will be **sax\_exam.xml**; in Section 2 we show you how to configure Studio 5.x to execute this class.
- \_\_\_ 5. **SAXExamReaderBase.java**
- \_\_\_ a. See how this file extends DefaultHandler, and provides simple implementations of the three error methods.
- \_\_\_ b. Notice how it contains an Exam instance variable and getter method.
- \_\_\_ c. This class is extended by the SAXExamParser.

Do **not** modify this file.

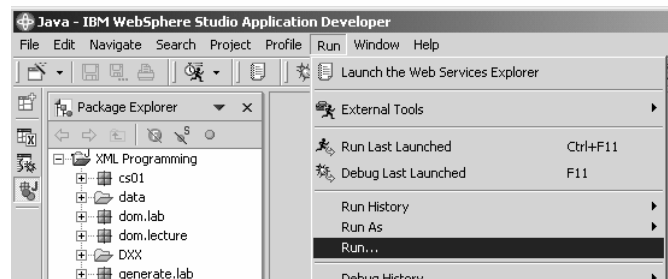
## Section 2 - Setup the Launch Configuration for PrintExam.java

We will use the **PrintExam.java** class in the **sax.lab** package in the **XML Programming** project in the **Java Perspective** to check our progress. Studio 5.x introduced a new approach to executing java programs by creating a *Launch Configurations* window.

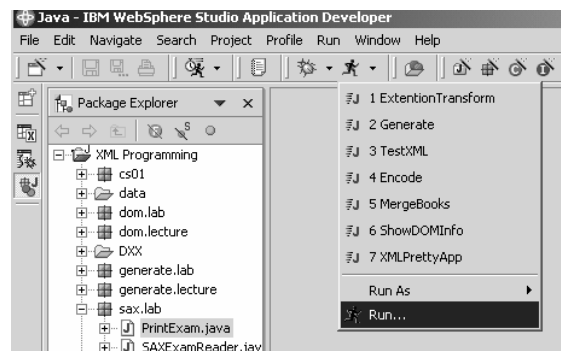
- \_\_\_ 1. Open the **Launch Configurations** window. There are two easy ways to open this window; both require you to be in the Java perspective.

<sup>1</sup> Reformatting is not modifying. [In *Studio 5.0*, **Format** is on its own line in the context-sensitive menu; in *5.1*, it is a sub-line under **Source**.

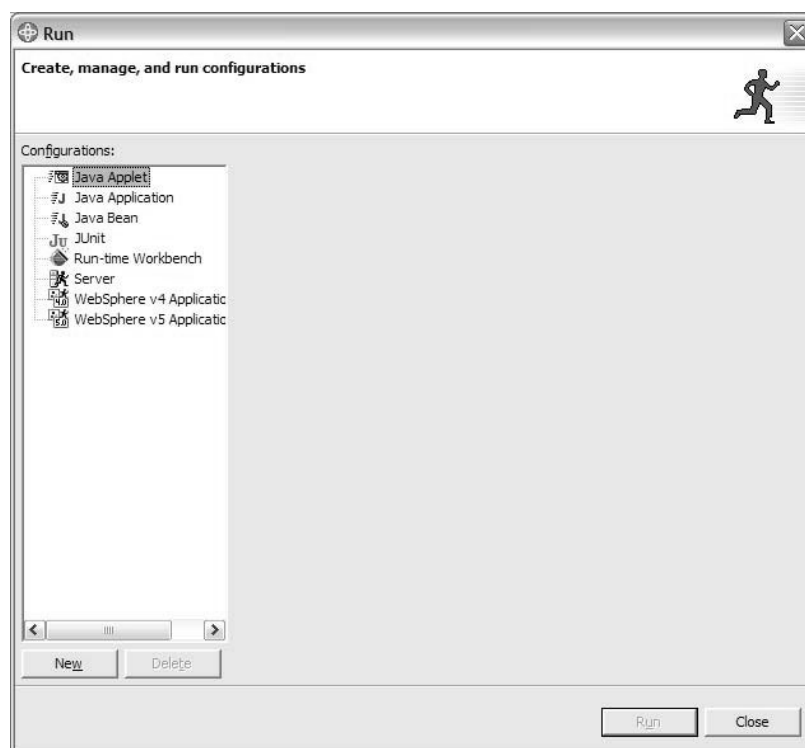
\_\_\_ a. Using Run on the menu bar:



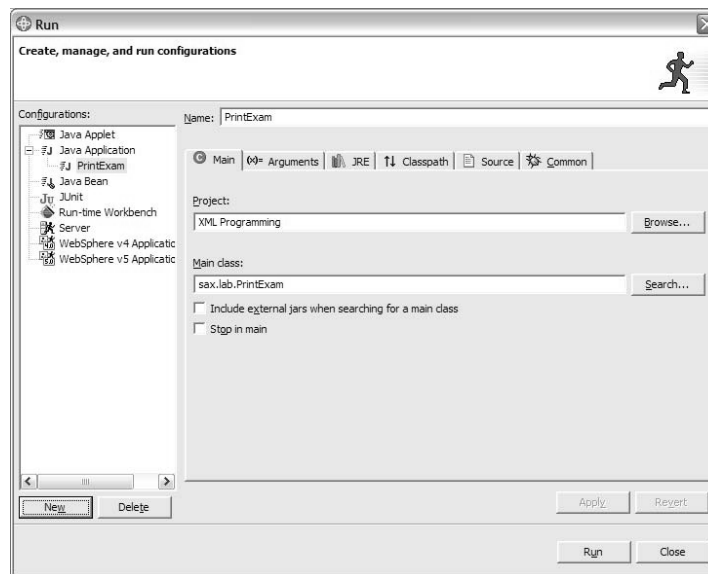
\_\_\_ b. Using the running man icon and adjacent arrowhead:



\_\_\_ c. Either way this **Launch Configurations** window will open:



- \_\_\_ 2. Configure an execution environment. As you can see, the LCW above has not been used. We want to add the executable class **PrintExam** to the Configurations list; we want PrintExam to use the **sax\_exam.xml** (as identified in the table in Section 1) as input. Here's how:<sup>2</sup>
- \_\_\_ a. Arguably, the most efficient way to proceed is to select the *executable* class **PrintExam** and then use method b., above, selecting **Run**. Then select **Java** Application and click **New** to create a new configuration that looks like this:



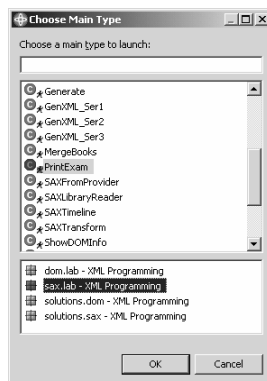
If you followed a different path, change your screen to look like ours.<sup>3</sup> Since **PrintExam** will occur in other labs you might want to change the **Name**: to something more memorable like **SAX PrintExam**. [The **Name**: is just a label.]

- \_\_\_ b. **PrintExam.java** needs input to process. Click the Arguments tab and enter **data/sax\_exam.xml**.

<sup>2</sup> The phrase here's how is an alert that what follows are either *hints*, *detailed* instructions, or *partial* solutions. If you wish, you may work out the solution on your own.

<sup>3</sup> 1.) Browse for the **XML Programming** project.

2.) Search for **PrintExam**; because there is more than one such file, you are asked to select the appropriate package, **sax.lab**.



Select OK to make your choice and return to the Launch Configurations window.

3.) Select **Apply** to move your Name into the Launch Configurations list.

- \_\_\_ c. Select **Run** to execute the PrintExam.java class. A console window opens at the lower right-hand corner of the Studio screen. We have yet to do anything so the result should be

Exam Reader results:  
Null

Which is what you would expect based on the code for PrintExam.java.

### **Notes on the LCW:**

- \_\_\_ a. Do I have to go through this process for every change? No. You can re-run the same configuration either by re-opening the Configurations window or selecting any of a number of choices in the drop-down menu associated with either Run in the toolbar or the dropdown menu from the running man icon a the right of the debug icon or just by selecting the icon, itself. Look for the **Name:** label you assigned above.
- \_\_\_ b. What happens if I forget an argument? You will need to include code (as we did in PrintExam.java with the first executable statement in main) to alert users. We should have (but didn't) include a file not found exception handler; in our example, any argument satisfies our *if* test.

## **Section 3 - Implement Your Solution into SAXExamReader.java**

In this section of the lab, you will add all the core logic to the **SAXExamReader.java** class. Items that must be added include:

A **constructor** that contains SAX parser creation logic;

All **ContentHandler event methods**; and

Any **instance variables** that need to be used to process the XML document.

It may help you to relate what follows in *our solution*<sup>4</sup> to the charts in the SAX Parser lecture.

This is what the SAXExamReader.java class looks like before we modify it; we've added line numbers<sup>5</sup> to help us orient you in the sections that follow.

```

1 package sax.lab;
2 import org.xml.sax.Attributes;
3 import org.xml.sax.SAXException;
4 /** SAXExamReader: Read in Exam XML document and create Exam bean */
5 public class SAXExamReader extends SAXExamReaderBase {
6
7 //-- Possible Instance variables
8 // Question Holder
9 // Choice Holder
10 // LIFO holder for characters input

```

<sup>4</sup> In addition to *here's how*: or *here's how we did it*: this is another sentinel that what follows is an outline — or the exact steps — we followed to achieve a solution: if you want to work on your own, please do so. You can always refer back to our solutions if you have problems. Henceforth, solutions are found in the *package* corresponding to the lab you are in.

<sup>5</sup> The line numbers were added via **Window> Preferences> Editor> {{checkbox} Show line numbers}**.

```
11
12 /** Constructor */
13 SAXExamReader(String input) {
14 // Create the parser factory (JAXP)
15 // Set Validation
16 // Create a Parser from factory
17 // Create an XMLReader
18 // Set Content Handler
19 // Set Error Handler
20 // Parser the input file
21 }
22 /** startDocument() */
23 public void startDocument() {
24 // Initialize any instance variables
25 }
26 /** endDocument() */
27 public void endDocument() {
28 // Clean up any variables
29 }
30 /** startElement() */
31 public void startElement(
32 String uri,
33 String lName,
34 String qName,
35 Attributes attrs)
36 throws SAXException {
37
38 // Create & configure any needed instances for certain elements
39 // (e.g.: Exam, Question, Choice)
40
41 }
42 /** endElement() */
43 public void endElement(String uri, String lName, String qName)
44 throws SAXException {
45
46 // Store any important data into the correct bean
47 }
48 /** characters */
49 public void characters(char[] ch, int start, int length) {
50 // Save the text for the current element
51
52 }
53
54 }
```

### **Section 3.1 - Constructor Method**

Include all Parser generation and configuration steps in this method.

\_\_\_ 1. Our first cut at a solution<sup>6</sup> looks like the following, it replaces lines 12 – 21 above.



Our new code is in **bold**; line 12 is now line 18 because of the import statements WSAD automatically added (if you expect a "light bulb" prompt but do not get one, the build you are using has probably added an import statement - you need to validate it) or we added, as described below, to accommodate the new classes/methods.

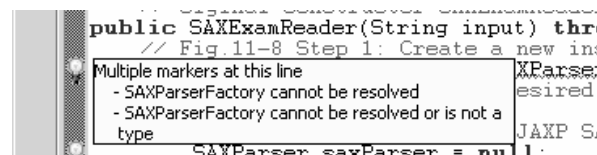
```

18 /** Constructor */
19 public SAXExamReader(String input) throws SAXException {
20 // Create the parser factory (JAXP)
21 SAXParserFactory spf = SAXParserFactory.newInstance();
22 spf.setValidating(true); // Set Validation
23 SAXParser saxParser = null; // Create a Parser from factory
24 try {
25 saxParser = spf.newSAXParser();
26 } catch (ParserConfigurationException pce) {
27 System.err.println("Failed to configure parser");
28 pce.printStackTrace(System.err);
29 }
30 XMLReader xr = saxParser.getXMLReader(); // Create an XMLReader
31 xr.setContentHandler(this); // Set Content Handler
32 xr.setErrorHandler(this); // Set Error Handler
33 try {
34 xr.parse(input); //Parse the input file
35 } catch (IOException ioe) {
36 System.err.println("Failed to read input"+input);
37 ioe.printStackTrace(System.err);
38 }
39
40 }

```

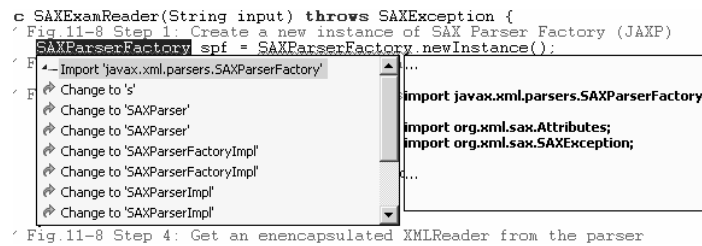
This is a *Studio 5.1* screen capture; 5.0 shows only the light bulbs until you **Save**. The next two screen captures are the same for either version.

- \_\_\_ a. Hover your cursor over the first bulb to see this box:



<sup>6</sup> CAUTION: we are not showing any packages you might have to import to compile our solutions. Application Developer can automatically import the appropriate packages for you. If right-clicking on the idea icon suggests you import a package, usually the top offering is the correct one.

- \_\_\_ b. With your cursor still on the 1<sup>st</sup> bulb, left-click your mouse to open this action window:



- \_\_\_ c. From the lecture notes you know that you need to import a SAX parser factory so double-click the highlighted choice (or select it and press <Enter>).
- \_\_\_ d. The bulb is replaced by the red circle with an X inside. Save your file to recompile and remove it. [If you look carefully in version 5.0 you'll see each bulb overlays a red error circle. Version 5.1 makes these circles more obvious and it is not necessary to **Save** (recompile) to see them.]
- \_\_\_ e. Resolve the remaining bulbs. When you have finished observe that Studio has not only added the missing import statements, it has done so in alphabetical order:

```

1 package sax.lab;
2 import java.io.IOException;
3
4 import javax.xml.parsers.ParserConfigurationException;
5 import javax.xml.parsers.SAXParser;
6 import javax.xml.parsers.SAXParserFactory;
7
8 import org.xml.sax.Attributes;
9 import org.xml.sax.SAXException;
10 import org.xml.sax.XMLReader;
11 /** SAXExamReader: Read in Exam XML document and create Exam bean */
12 public class SAXExamReader extends SAXExamReaderBase {
13
14 //-- Possible Instance variables

```

Lines 1, 8, and 9 can be seen to be present in the screen capture at the beginning of this section.

We are now positioned at new line number 42 where we deal with **startDocument**.

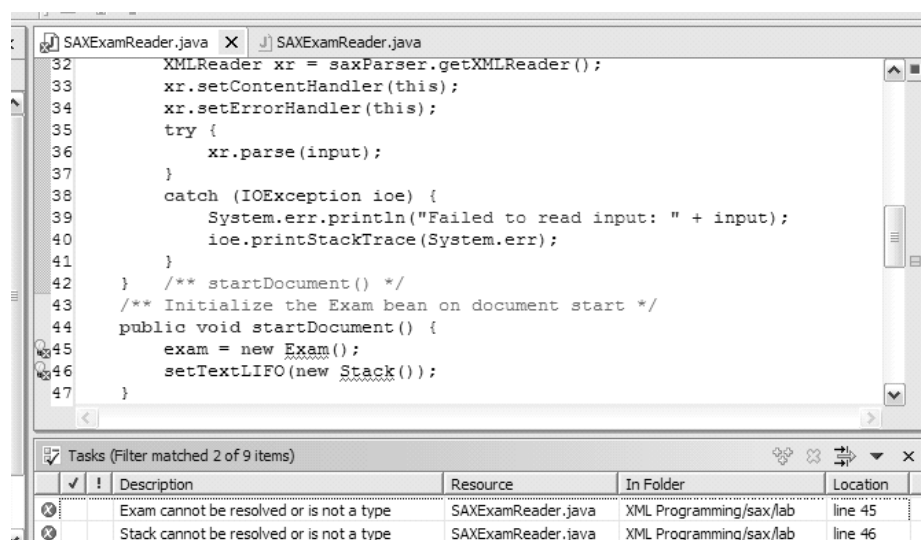
## Section 3.2 - Content Handler event methods

This section addresses our overriding of the *Overridable Methods in DefaultHandler* in Fig. -12 in the lecture that we need here. We have arranged the subsections to follow the order in that chart.

### Section 3.2.1 - startDocument() –

Add any required variable initializations to this method.

\_\_\_ 1. Here is our partial solution:

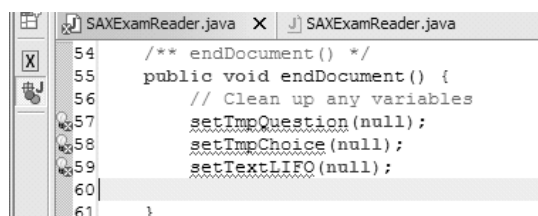


- \_\_\_ 2. Exam.java is one of the beans in the cs01 package. Accept the suggestion to import cs01.Exam to make the Exam.java class available. Remember to save your file to remove the error from the Task list.
- \_\_\_ 3. The second glitch represents two problems. First, Stack is an external class. Accept Studio's offer to import java.util.Stack and Save to solve the first problem. Second, when you type setTextLIFO() you will get an offer from *Studio* to create the method for you; decline its offer for now.<sup>7</sup> It is clear that textLIFO is an attribute of type Stack. textLIFO is a Last In First Out holder (stack) for character input. Let's go on and add the instance variable textLIFO to the attribute list at the top of this file immediately before the comment on line 20: *// LIFO holder for characters input.*

### Section 3.2.2 - endDocument() –

Add any cleanup of variables here.

\_\_\_ 1. Here is our solution, which is at our line number 55:



<sup>7</sup> We want to show you a technique *Studio* has for generating getters and setters: it's more impressive if you have more than one instance variable to operate on.

**Note:** This block is not really important in this exercise except that it identifies two more instance variables:

- \_\_\_ 2. The set method is strong evidence that TmpQuestion and TmpChoice must also be attributes (instance variables) to be defined at the head of this class. Go ahead and add them before their corresponding comments at lines 18 and 19. Based on the errors, we recognize that Question and Choice are classes in package cs01; go ahead and import them.<sup>8</sup> Then please peruse Section 3.3 at the end of this section to see how to *efficiently* use *Studio* to help.

### Section 3.2.3 - startElement()

- \_\_\_ 1. Add any special logic for the beginning of any elements.
  - \_\_\_ a. Use a **StringBuffer** instance to reserve space on a **Stack** for the text encountered in each element. Review your course notes for an example of how to do this.

Here is our solution:

```
/** Set the application state based on start element tag name */
public void startElement(
 String uri,
 String lName,
 String qName,
 Attributes attrs)
 throws SAXException {

 String currElement;
 // Set our element based on namespace or not
 if (qName.length() > 0)
 currElement = qName;
 else
 currElement = lName;

 // Whatever we have, push a new String Buffer onto text stack
 getTextLIFO().push(new StringBuffer());
```

- \_\_\_ b. When a **<test>** is detected, you could read the test attributes and assign them to the properties of the Exam objects. Here is our solution:

```
if (currElement.equals("test")) {
 // Get ID attribute
 exam.setId(attrs.getValue("id"));
}
```

- \_\_\_ c. When a **<question>** is detected, you could create a new Question instance, assign it to an instance variable so it can be accessed elsewhere in your class,

<sup>8</sup> Be careful, Choice is also a class in java.awt; we want the one in cs01.

read the question attributes, and assign them to the newly created Question object properties. Here is our solution:

```
else if (currElement.equals("question")) {
 // Create new Question bean, and get ID attribute
 setTmpQuestion(new Question());
 getTmpQuestion().setId(attrs.getValue("id"));
}
```

- \_\_\_ d. When a **<choice>** is detected, you could create a new Choice instance, assign it to an instance variable so it can be accessed elsewhere in you class, read the choice attributes, and assign them to the newly created Choice object properties. Here is our solution:

```
else if (currElement.equals("choice")) {
 // Create new Choice bean, and get ID attribute
 setTmpChoice(new Choice());
 getTmpChoice().setId(attrs.getValue("id"));
}
```

- \_\_\_ e. When a **<choices>** is detected, you could read the allowMultiple attribute and assign the value to the allowMultiple property of the current Question. Here is our solution:

```
else if (currElement.equals("choices")) {
 // Set allow mult. variable of Question bean
 getTmpQuestion().setAllowMultiple(attrs.getValue("allowMultiple"));
}
```

### Section 3.2.4 - endElement() –

This method tells you that you are finished with an element and are ready to call the appropriate Javabeans setter (to assign a property value or establish the relationships depicted in the diagram above). The processing is handled in this method.

- \_\_\_ 1. Add logic to determine the element name of the element that is ending. Review your course notes for an example of how to do this. Remember that the element may be namespace qualified. Here is our solution:

```
/** Set the application state based on end element tag name */
public void endElement(String uri, String lName, String qName)
 throws SAXException {

 // Set our element based on namespace or not
 String currElement;

 if (qName.length() > 0)
```

```
 currElement = qName;
 else
 currElement = lName;
```

- \_\_\_ 2. Pop the **StringBuffer** at the top of the stack and store a reference in a local variable, you'll need the StringBuffer for some of the following steps. Here is our solution:

```
// Pop the current StringBuffer and see if we want it
StringBuffer sb = (StringBuffer) getTextLIFO().pop();

// Take action based on the element we are closing.
```

### Section 3.2.5 - Schema Elements –

- \_\_\_ 1. If the element is a **question**, add the current Question instance to the exam. Here is our solution:

```
if (currElement.equals("question")) {
 // Create a populated temp question, and add it to exam
 exam.getQuestions().add(getTmpQuestion());
}
```

- \_\_\_ 2. If the element is a **choice**, add the current Choice instance to the current Question instance. Here is our solution:

```
else if (currElement.equals("choice")) {
 // Create a populated temp choice, and add it to question
 getTmpQuestion().getChoices().add(getTmpChoice());
}
```

- \_\_\_ 3. If the element is a **correct**, assign the text value of the popped StringBuffer to the correct property of the current Question object. Don't forget to eliminate the leading and trailing whitespace. Here is our solution:

```
else if (currElement.equals("correct")) {
 getTmpChoice().setCorrect(sb.toString().trim());
}
```

- \_\_\_ 4. If the element is a **description**, assign the text value of the popped StringBuffer to the description property of the Exam object. Don't forget to eliminate the leading and trailing whitespace. Here is our solution:

```
else if (currElement.equals("description")) {
 exam.setDescription(sb.toString().trim());
}
```

- \_\_\_ 5. If the element is a **questionText**, assign the text value of the popped StringBuffer to the text property of the current Question object. Don't forget to eliminate the leading and trailing whitespace. Here is our solution:

```
else if (currElement.equals("questionText")) {
```

```

 getTmpQuestion().setText(sb.toString().trim());
 }

```

- \_\_\_ 6. If the element is a **choiceText**, assign the text value of the popped StringBuffer to the text property of the current Choice object. Don't forget to eliminate the leading and trailing whitespace. Here is our solution:

```

 else if (currElement.equals("choiceText")) {
 getTmpChoice().setText(sb.toString().trim());
 }

```

### Section 3.2.6 - characters() —

Store any text content for the current element.

### Section 3.3 - Instance Variables/Attributes

- \_\_\_ 1. Of what type are question, choice and LIFO? Answer: Question, Choice, and Stack. Our solution is:

```

// Possible Instance variables

private Question tmpQuestion; // Question Holder

private Choice tmpChoice; // Choice Holder

private Stack textLIFO; // LIFO for char input

```

- \_\_\_ 2. Now you have a new problem: Question and Choice (and maybe even Stack) are not defined. Refer to the Resource Table at the beginning of this exercise. Question and Choice are classes in package cs01. Click the light bulb for one or the other and double-click the cs01.xxx to import it (if you have not already done so). Do the same for any others: accept the topmost (preferred) entry.
- \_\_\_ 3. You are still in the woods: you need setters and/or getters for one or more of these variables. If you accepted Studio's offer to generate the setter for textLIFO earlier you got some private void . . . junk. Instead select any one of the attributes; right-click and select Source->Generate Getter and Setter. . . , then select all and Studio will put exactly what you need, neatly, alphabetically by method name, at the bottom of the page.
- \_\_\_ 4. **Save** your work to remove the associated errors. If you do this as part of the completion of Section 3.3.2 your work should now be error free.

## Section 4 - Execute the Result

- \_\_\_ 1. Once your program is complete, run the **PrintExam.java** driver program to test it. Refer to Section 2 for help. Compare your results with those on the last page of this exercise.

Here is our solution with the missing statements filled in:

```
1 package solutions.sax;
2 import java.io.IOException;
3 import java.util.Stack;
4
5 import javax.xml.parsers.ParserConfigurationException;
6 import javax.xml.parsers.SAXParser;
7 import javax.xml.parsers.SAXParserFactory;
8
9 import org.xml.sax.Attributes;
10 import org.xml.sax.SAXException;
11 //import org.xml.sax.SAXParseException;
12 import org.xml.sax.XMLReader;
13
14 import cs01.Choice;
15 import cs01.Exam;
16 import cs01.Question;
17 /** SAXExamReader: Read in Exam XML document and create Exam bean */
18 public class SAXExamReader extends SAXExamReaderBase {
19
20 // -- Possible Instance variables
21 private Question tmpQuestion; // Holder Question
22 private Choice tmpChoice; // Holder Choice
23 private Stack textLIFO; // LIFO holder for characters input
24
25 /** Constructor */
26 public SAXExamReader(String input) throws SAXException {
27 // Create the parser (JAXP)
28 SAXParserFactory spf = SAXParserFactory.newInstance();
29 spf.setValidating(true);
30 SAXParser saxParser = null;
31 try {
32 saxParser = spf.newSAXParser();
33 } catch (ParserConfigurationException pce) {
34 System.err.println("Failed to configure parser");
35 pce.printStackTrace(System.err);
36 }
37 XMLReader xr = saxParser.getXMLReader();
38 xr.setContentHandler(this);
39 xr.setErrorHandler(this);
40 try {
41 xr.parse(input);
42 } catch (IOException ioe) {
```



```

43 System.err.println("Failed to read input: " + input);
44 ioe.printStackTrace(System.err);
45 }
46 }
47
48 /** Initialize the Exam bean on document start */
49 public void startDocument() {
50 exam = new Exam();
51 setTextLIFO(new Stack());
52 }
53
54 /** Add final counts to Exam bean on document end */
55 public void endDocument() {
56 // Clean up our variables
57 setTmpQuestion(null);
58 setTmpChoice(null);
59 setTextLIFO(null);
60 }
61
62 /** Set the application state based on start element tag name */
63 public void startElement(
64 String uri,
65 String lName,
66 String qName,
67 Attributes attrs)
68 throws SAXException {
69
70 String currElement;
71 // Set our element based on namespace or not
72 if (qName.length() > 0)
73 currElement = qName;
74 else
75 currElement = lName;
76
77 // Whatever we have, push a new String Buffer onto text stack
78 getTextLIFO().push(new StringBuffer());
79
80 if (currElement.equals("test")) {
81 // Get ID attribute
82 exam.setId(attrs.getValue("id"));
83 } else if (currElement.equals("question")) {
84 // Create new Question bean, and get ID attribute
85 setTmpQuestion(new Question());
86 getTmpQuestion().setId(attrs.getValue("id"));
87 } else if (currElement.equals("choices")) {
88 // Set allow mult. variable of Question bean
89
90 getTmpQuestion().setAllowMultiple(attrs.getValue("allowMultiple"));
91 } else if (currElement.equals("choice")) {

```

```
91 // Create new Choice bean, and get ID attribute
92 setTmpChoice(new Choice());
93 getTmpChoice().setId(attrs.getValue("id"));
94 }
95 }
96
97 /** Set the application state based on end element tag name */
98 public void endElement(String uri, String lName, String qName)
99 throws SAXException {
100
101 // Set our element based on namespace or not
102 String currElement;
103
104 if (qName.length() > 0)
105 currElement = qName;
106 else
107 currElement = lName;
108
109 // Pop the current StringBuffer and see if we want it
110 StringBuffer sb = (StringBuffer) getTextLIFO().pop();
111
112 // Take action based on the element we are closing.
113 if (currElement.equals("question")) {
114 // Create a populated temp question, and add it to exam
115 exam.getQuestions().add(getTmpQuestion());
116 } else if (currElement.equals("choice")) {
117 // Create a populated temp choice, and add it to question
118 getTmpQuestion().getChoices().add(getTmpChoice());
119 } else if (currElement.equals("correct")) {
120 getTmpChoice().setCorrect(sb.toString().trim());
121 } else if (currElement.equals("description")) {
122 exam.setDescription(sb.toString().trim());
123 } else if (currElement.equals("questionText")) {
124 getTmpQuestion().setText(sb.toString().trim());
125 } else if (currElement.equals("choiceText")) {
126 getTmpChoice().setText(sb.toString().trim());
127 }
128 }
129
130 /** append any characters that are detected to the buffer for the current
element */
131 public void characters(char[] ch, int start, int length) {
132 ((StringBuffer) getTextLIFO().peek()).append(ch, start, length);
133 }
134
135 /**
136 * Gets the tmpQuestion
137 * @return Returns a Question
138 */
```

```
139 private Question getTmpQuestion() {
140 return tmpQuestion;
141 }
142 /**
143 * Sets the tmpQuestion
144 * @param tmpQuestion The tmpQuestion to set
145 */
146 private void setTmpQuestion(Question tmpQuestion) {
147 this.tmpQuestion = tmpQuestion;
148 }
149
150 /**
151 * Gets the tmpChoice
152 * @return Returns a Choice
153 */
154 private Choice getTmpChoice() {
155 return tmpChoice;
156 }
157 /**
158 * Sets the tmpChoice
159 * @param tmpChoice The tmpChoice to set
160 */
161 private void setTmpChoice(Choice tmpChoice) {
162 this.tmpChoice = tmpChoice;
163 }
164
165 /**
166 * Gets the textLIFO
167 * @return Returns a Stack
168 */
169 private Stack getTextLIFO() {
170 return textLIFO;
171 }
172 /**
173 * Sets the textLIFO
174 * @param textLIFO The textLIFO to set
175 */
176 private void setTextLIFO(Stack textLIFO) {
177 this.textLIFO = textLIFO;
178 }
179
180 }
```

## Sample Output of the PrintExam Program

Exam Reader results:

Exam: XML32-SAX - Simple API for XML (SAX) Parser  
Number of Questions: 4

=====

1. What are the three main sections of a SAX application:

- < > A. SAXParserFactory, SAXParser, XMLReader
- < > B. Document, Elements, Characters
- < > C. Parser Creation, Content Handler, Error Handler \*
- < > D. JAXP, SAX, DefaultHandler
- < > E. StartDocument, EndDocument, Parse

=====

2. Which of the following statements accurately describe SAX application programming

- [ ] A. SAX is simple, because all of the work is done for you
- [ ] B. SAX provides no state management facilities \*
- [ ] C. SAX events provide hierarchical data about the XML
- [ ] D. SAX provides Default Handler implementations that do what you would typically want

(Multiple answer question - Select all that apply)

=====

3. Which SAX history statements are correct

- [ ] A. Created by David Megginson and XML-Dev Mailing list members \*
- [ ] B. SAX 2 is now a W3C standard recommendation
- [ ] C. SAX is considered a "defacto" standard \*
- [ ] D. SAX was written for Java, no other "official" bindings \*
- [ ] E. SAX is only available for Java, no other languages

(Multiple answer question - Select all that apply)

=====

4. Which of the following DefaultHandler statements is true:

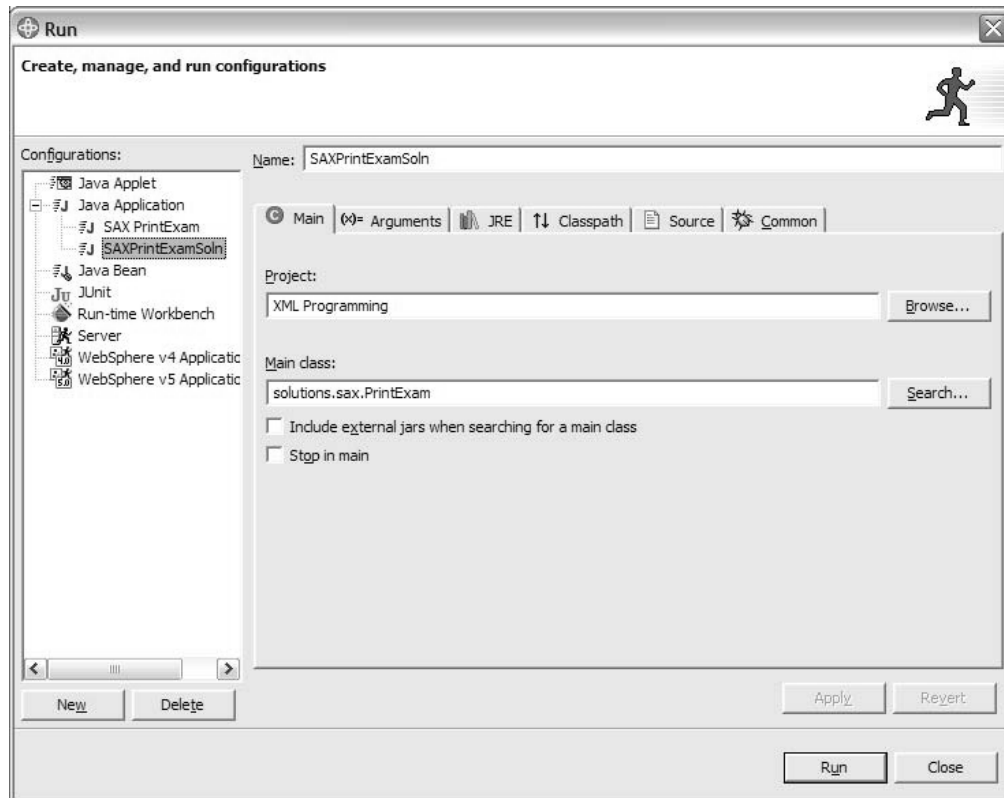
- < > A. Provides empty implementation methods for several SAX interfaces \*
- < > B. Contains methods for SAXFactory creation
- < > C. Provides methods for full error handling capabilities
- < > D. Abstracts the vendors actual parser implementation

=====

End of Exam

## Running Multiple Programs

In testing this lab we ran both the SAXExamReader.java file we created in the sax.lab package and the identically-named files in the solutions.sax package. Here's what our run configuration window looks like after the steps we provide below:



Here's how:

1. Select PrintExam.java in the solutions.sax package in Package Explorer in the Java perspective.
2. Select the **V** next to the Running Man icon; from the list select **Run** next to the icon.
3. Select **Java Application** (shown in the window above).
4. Select **New** (shown in the window above).
5. Provide a **Name**: that uniquely identifies it.
6. Check that the **Project**: and **Main class**: are what you intend.
7. Copy (or type in) the proper arguments.
8. **Run** it: the results will appear in the Console window.

## END OF LAB



## Exercise 10. DOM Parser Programming

### What This Exercise is About

In this exercise, you will put your knowledge of DOM programming to use by developing a real-world application.

You have been assigned a project that will read in an XML document that contains the familiar exam data. The goal of this assignment is to parse the XML data using DOM and store the test data into preexisting Java beans. Once everything is loaded, test your results by calling the "toString()" method of the Exam bean. You should be presented with all the XML data presented in a printable format.

The Java beans, a sample application class, and a basic skeleton class have all been provided that offers a starting point.

There are a total of three beans -- Exam, Question, and Choice. Exam holds information related to the entire exam and a Vector of Questions. The Question bean holds information for each Question, along with a Vector of Choices. The Choice bean contains information related to a single choice for a question.

### What You Should Be Able to Do

At the end of the lab, you should be able to:

- Analyze an XML document and map its content to Java bean attributes
- Develop a working DOM application to parse an XML document
- Load XML data into Java beans
- Verify the data was loaded correctly

### Required Materials

- IBM's WebSphere Studio Application Developer 5.x
- XML Parser (Xerces is included with Application Developer)
- XML document and DTD of an online exam
- Prebuilt javabeans to hold data
- Prebuilt Application program, and base support class

## Applicability

- XML321 Exercise 3
- XML341 Exercise 10



## Exercise Instructions

### Section 1 - Analyze Source Documents and Target Java beans

If you haven't done so already, examine the source Exam XML document, its associated vocabulary, and the Java bean components that will hold the final information. Knowing what you have will aid you greatly in developing the application.

The files that are specific to this lab are:

**data/dom\_exam.xml**,  
**dom.lab.PrintExam.java**,  
**dom.lab.DOMExamReader.java** and

Depending on the order in which the labs are completed, you may have reviewed the others in a previous lab, if that is the case there is no need to review them again.

\_\_\_ 1. Open the Application Developer and locate the following resources:

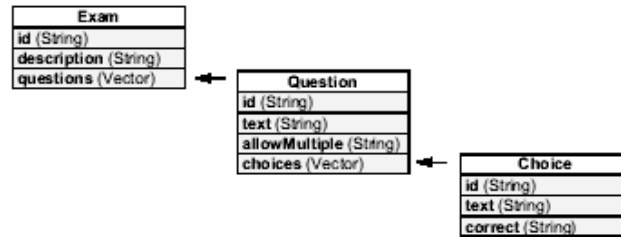
Resource	Location	Description
Project	XML Programming	Project location for all course resources
Source XML	<i>Folder</i> data/dom_exam.xml	XML containing Exam information to parse. Do <b>not</b> edit.
Source DTD	<i>Folder</i> data/test.dtd	DTD vocabulary for XML document. Do <b>not</b> edit.
	Package Class	
Target Java bean Classes	cs01 Exam.java cs01 Question.java cs01 Choice.java	Simple Java beans used to store exam data. Do <b>not</b> edit.
Application	dom.lab PrintExam.java	Simple runnable application to test your new DOM program. <sup>a</sup>
DOM Skeleton Class	dom.lab DOMExamReader.java	Empty program class that extends Base class. You will need to implement several methods in this class.

a. You may edit this file if you wish but there is no requirement to do so in this exercise

\_\_\_ 2. Familiarize yourself with the XML source file and its associated DTD. These are the files that define the structure you will be processing. Do **not** modify these files. [Exit without saving changes.]

\_\_\_ 3. **Javabeen Classes:** Examine the three target Java bean files:

- Compare how each of the XML elements and attributes compare to a matching Java bean attribute.
- The beans relate to each other in a simple hierarchical structure.



\_\_\_ 4. **PrintExam.java** Examine this *application* class: It contains a simple `main()` method to instantiate the `DOMExamParser` with a supplied file name via the Command Arguments of this class. For this exercise that file name is **dom\_exam.xml**. Here's how to configure it for a run:

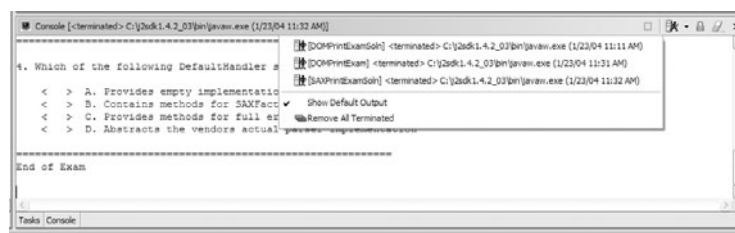
- \_\_\_ a. Select the **PrintExam.java** in the Packages view.
- \_\_\_ b. Select **Run** on the menu bar and **Run** from the drop-down list that appears.
- \_\_\_ c. Double-click **Java Application** at the top of the left-most window and select **PrintExam** [Warning: if you completed the previous lab, a *PrintExam* will already be there; in this case select `PrintExam(1)` and verify that the Main Class is **dom.lab.PrintExam** rather than **sax.lab.PrintExam**] You might want to modify the name to something like **DOMPrintExam** just to make it easier on yourself.
- \_\_\_ d. Click the Program Arguments tab, add:

**data/dom\_exam.xml**

- \_\_\_ e. Click Run. The results will appear in the Console window associated with the Debugger perspective that is created by the run process. The Console output should be Exam Reader results: null [Not on the same line: we are saving space.]

- To check your results at any time, run this program by clicking the running man icon when **PrintExam (or PrintExam(1) or DOMPrintExam)** is displayed. The results will appear in the Console window that appears in the pane formerly occupied by the **Tasks** view.<sup>1</sup> associated with the Debugger perspective that is created by the run process.

<sup>1</sup> In *Studio 5.0* you need to open the Debug perspective to review previous runs; in *Studio 5.1* you can accomplish a similar result from a drop-down list in the context window:



## Section 2 - Implement a Solution into DOMExamReader.java

In this section of the lab, you will add all the core logic to the `DOMExamReader.java` class. Items that must be added include:

- a Constructor that contains DOM parser creation logic and chooses which approach to employ;
- logic to transform the DOM document into the bean class; and
- any instance variables used to process the XML document.

The challenge can be solved many different ways. Two possible solution styles 1.) using recursion; and 2.) by direct element access are shown below. Use whatever style you are comfortable with.

### Section 2.0 - Constructor Method

Include all Parser instantiation and configuration steps in `DomExamReader` constructor. The way our solution is constructed it is a matter of enabling one set of statements and disabling another in the constructor. This is what the start of the **DOMExamReader.java** class looks like up to, and including, the end of the constructor method before we modify it; the line number option in **Editor** in **Preferences** in **Window** is still checked:

```

1 package dom.lab;
2
3 import java.util.Vector;
4
5 import cs01.Exam;
6
7 import org.w3c.dom.Document;
8 import org.w3c.dom.Element;
9 import org.w3c.dom.Node;
10
11 /** DOMExamReader: Read in Exam XML document and create Exam bean */
12 public class DOMExamReader {
13
14 private Exam exam;
15
16 /** Constructor */
17 DOMExamReader(String input) throws Exception {
18 // try {
19 // // Create the parser factory (JAXP)
20 // // Set Validation
21 // // Create a Parser from factory
22 // // Parser the input file
23
24 // -----
25 // Two distinct solutions are possible. Choose the one
26 // you feel most comfortable with. If time permits, try
27 // to implement both solutions and decide which is easier.

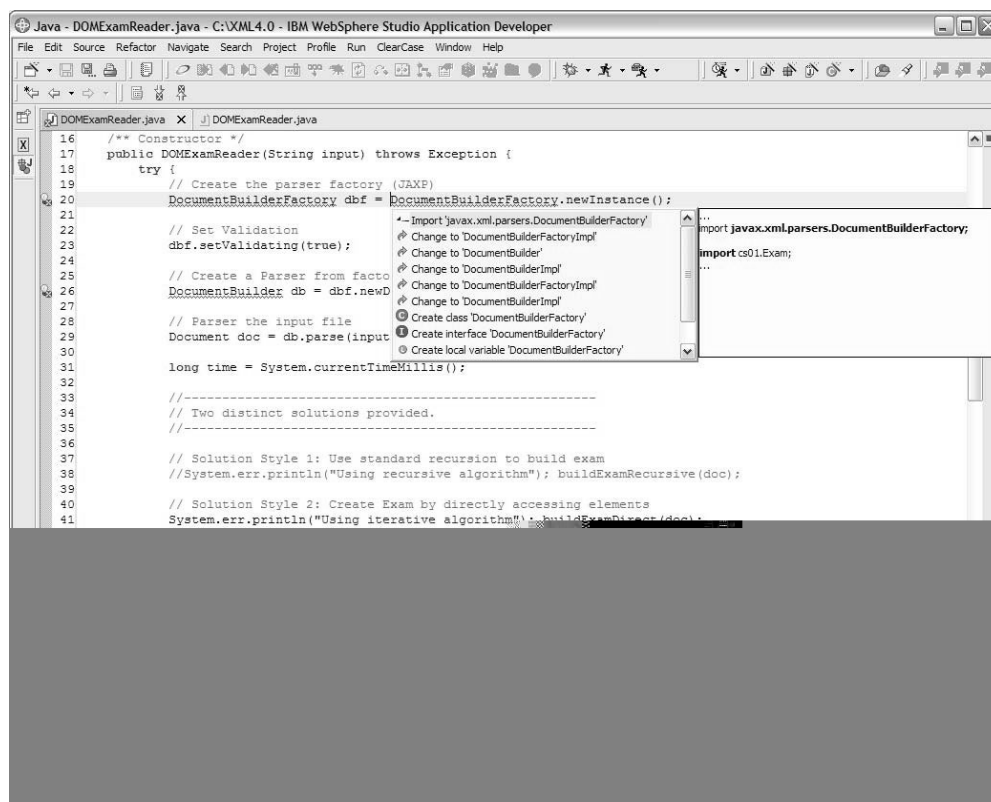
```

```
28 //-----
29 // Solution Style 1: Use standard recursion to build exam
30 // buildExamRecursive(doc);
31 // Solution Style 2: Create Exam by directly accessing elements
32 // buildExamDirect(doc);
33
34
35 // } catch (javax.xml.parsers.ParserConfigurationException pce) {
36 // throw new Exception(
37 // "The parser was not configured correctly -" + pce.getMessage());
38 // } catch (java.io.IOException ie) {
39 // throw new Exception("Cannot read input file -" + ie.getMessage());
40 // } catch (org.xml.sax.SAXException se) {
41 // throw new Exception("Problem parsing the file -" +
42 // se.getMessage());
43 // }
44 }
```

- \_\_\_ 1. If you want to solve this exercise on your own, refer to the Lecture Notes chart DOM Application Structure and subsequent charts for guidance on how to proceed.

Here's how we did it:<sup>2</sup>

- \_\_\_ a. We began by supplying the code implied by the comments. Here are the same lines:



- \_\_\_ b. We received two light bulbs prompting us to import the appropriate packages. Accept *Studio's* suggestions in both cases; don't forget to **Save** to recompile and remove the red error circles and clear the **Tasks** list.
- \_\_\_ c. Notice lines 31 and 43 -- we've inserted these to track the time spent. We use that for Section 3, question 5.

## Section 2.1 - Direct Access (Iterative) Solution

- \_\_\_ 1. **Logic to load exam bean from XML data:** We chose to use the 2nd approach in the code by modifying the comment at line 32 in the original into the line at line 41 above. Note that we have two statements on that line: one to print a message and the second to build the document.

### Section 2.1.1 - buildExamDirect()

A method to assemble an exam: extracts exam level information to create the Exam bean and calls a method to acquire a Vector of Questions that are then assigned to the Exam bean.

<sup>2</sup> Here's how and similar phrases indicate our solution, typically step-by-step, follows. Some of our exercises can take many hours depending on your skill level and confidence level. Do not hesitate to refer to our step-by-step solution for hints.

\_\_\_ 1. At the outset our file looks like this:

```
J\DOMExamReader.java X J\DOMExamReader.java
98 private void buildExamDirect(Document doc) {
99 // Obtain the "test" element from the document (this is our Exam!)
100 // Next get the "id" attribute, and "description" element and set bean
101 // Next, get the questions Vector, and set them in the exam
102 exam.setQuestions(buildQuestions(test));
103 }
```

\_\_\_ 2. Here is our solution:

```
157 private void buildExamDirect(Document doc) {
158 Element test = doc.getDocumentElement();//Obtain the "test" element from
the document (this is our Exam!)
159 if (test != null) // Next get the "id" attribute, and "description"
element and set bean
160 setExam(new Exam());
161 getExam().setId(test.getAttribute("id"));
162 Element desc =(Element)test.getElementsByTagName("description").item(0);
// Next, get the questions Vector, and set them in the exam
163 getExam().setDescription(desc.getFirstChild().getNodeValue());
164
165 getExam().setQuestions(buildQuestions(test));//exam.setQuestions(
buildQuestions(test));
166 }
```

- \_\_\_ a. *test* is the root element. Refer to the figure *Traversing the DOM Tree* to see that **doc.getDocumentElement()** returns an instance of the root element. *Studio 5.1* provides similar information via its *hover* utility; it differentiates between *Element*, *test*, and *doc* on one hand, and *getDocumentElement()* on the other.
- \_\_\_ b. *test.getAttribute(id)* returns a string. Refer to the figure *Dealing with Element Attributes* several pages after the figure referred to above. Again, take advantage of the power of *Studio* to help.
- \_\_\_ c. Similarly *test.getElementsByTagName(description)* is a specific use of `NodeList org.w3c.dom.Element.getElementsByTagName(String name)` which Returns a `NodeList` of all descendant `Elements` [of `Element`] with a given tag name, in the order in which they are encountered in a preorder traversal of this `Element` tree.

*Parameters:*

*name* The name of the tag to match on. The special value "" matches all tags.

*Returns:*

*A list of matching Element nodes.*

[The above from hovering over the actual code, using <F2> to make the *help* available and then cutting and pasting the italicized material here.]

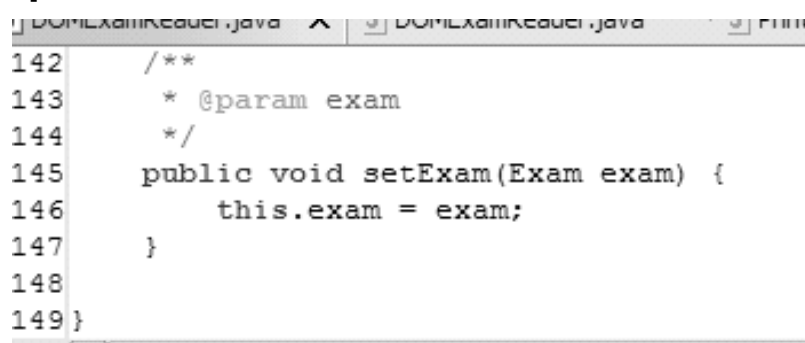
- \_\_\_ d. **desc.getFirstChild()** is on the same chart as a.

- \_\_\_ e. **getNodeValue()** is on a later figure *Working with Nodes*.
- \_\_\_ 3. When you **Save**, it produces a light bulb since the **setExam()** method is not defined. Go ahead and let *Studio* create it. However, *Studio 5.1* will not generate a complete method: it will, instead add a *TODO* to your **Task** list.

```
private void setExam(Exam newExam) {
 exam = newExam;
};
```

or

- \_\_\_ a. Erase the block, click the *instance variable exam* [line 14 in the screen capture, page 4], and let *Studio* create this block at the bottom of the file:



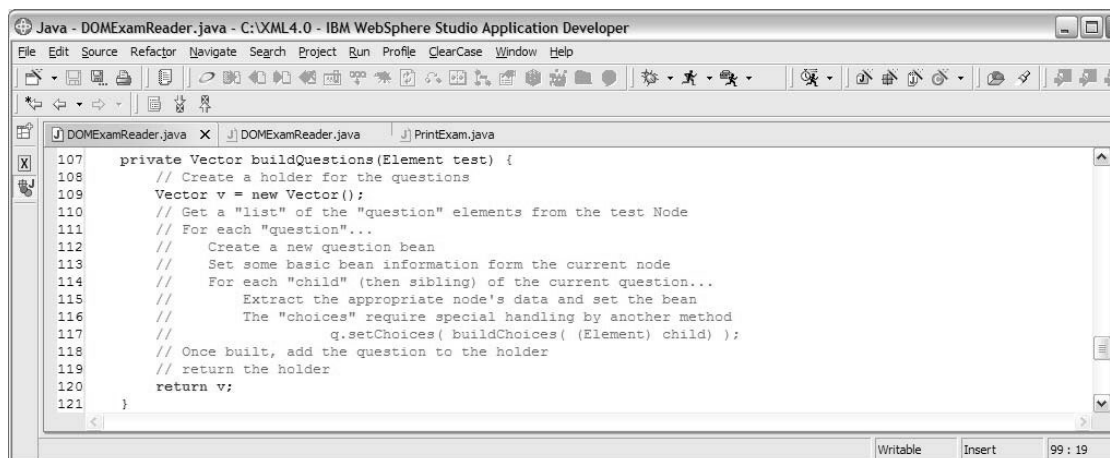
```
142 /**
143 * @param exam
144 */
145 public void setExam(Exam exam) {
146 this.exam = exam;
147 }
148
149 }
```

Note the key difference is *Studio* models it as a public method. Since we do not want objects of other classes calling this method, change its visibility to private.

### Section 2.1.2 - buildQuestions()

A method to build a Vector of Questions: creates a vector of Question beans, one for each Question in a given Exam and calls a method to acquire a Vector of Choices to be loaded into each Question.

- \_\_\_ 1. At the outset our file looks like this:





\_\_\_ 2. Here is our solution after we let *Studio* add two more imports, **org.w3c.dom.NodeList** and **cs01.Question**:

```

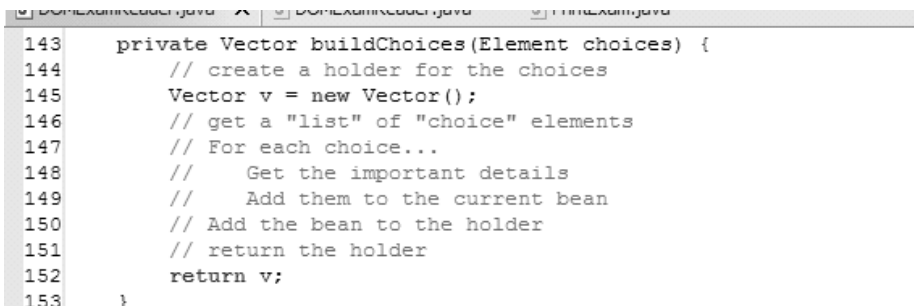
166 private Vector buildQuestions(Element test) {
167 // Create a holder for the questions
168 Vector v = new Vector();
169 // Get a "list" of the "question" elements from the test Node
170 NodeList questions = test.getElementsByTagName("question");
171 // For each "question"...
172 for (int i = 0; i < questions.getLength(); i++) {
173 Element qe = (Element) questions.item(i);
174 Question q = new Question(); //Create a new question bean
175 q.setId(qe.getAttribute("id"));
176 //Set some basic bean information from the current node
177 for (Node child = qe.getFirstChild(); child != null; child
=child.getNextSibling()) {
178 //Extract the appropriate node's data and set the bean
179 if ("questionText".equals(child.getNodeName())) {
180 q.setText(child.getFirstChild().getNodeValue());
181 } else if ("choices".equals(child.getNodeName())) {
182 //The "choices" require special handling by another method
183 q.setAllowMultiple(
184 ((Element) child).getAttribute("allowMultiple"));
185 q.setChoices(buildChoices((Element) child));
186 }
187 }
188 v.add(q); // Once built, add the question to the holder
189 }
190 return v; // return the holder
191 }

```

### Section 2.1.3 - buildChoices()

A method to build a Vector of Choices: creates a Vector of Choice beans, one for each choice in a given question.

\_\_\_ 1. At the outset our file looks like this:



```

143 private Vector buildChoices(Element choices) {
144 // create a holder for the choices
145 Vector v = new Vector();
146 // get a "list" of "choice" elements
147 // For each choice...
148 // Get the important details
149 // Add them to the current bean
150 // Add the bean to the holder
151 // return the holder
152 return v;
153 }

```

\_\_\_ 2. Here is our solution after we let *Studio* add **cs01.Choice**:

```

private Vector buildChoices(Element choices) {
 // create a holder for the choices

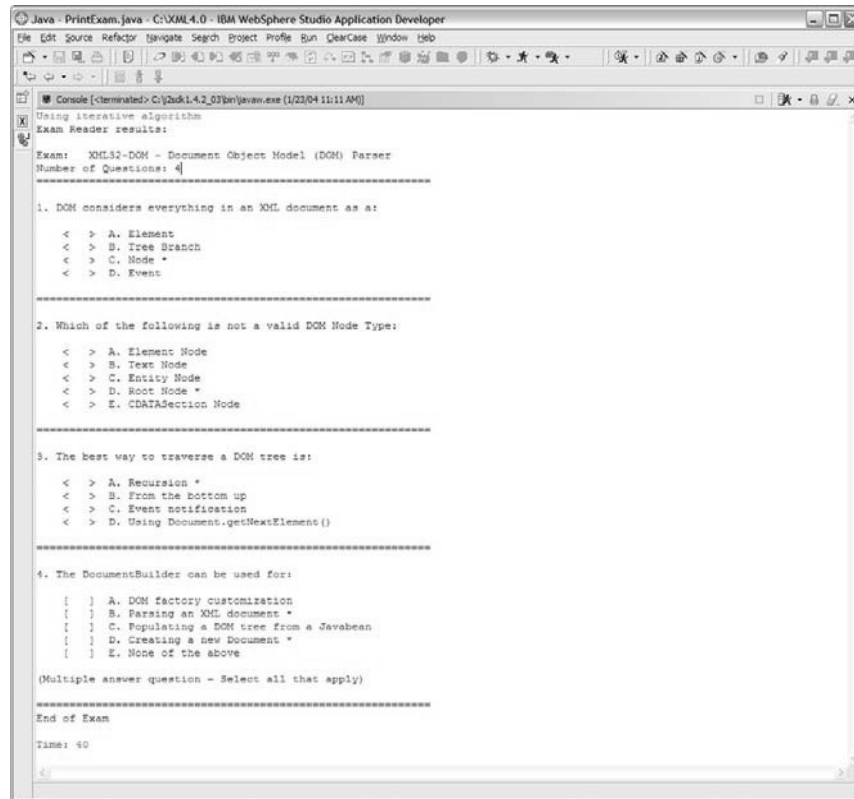
```



```
Vector v = new Vector();
// get a "list" of "choice" elements
NodeList choiceList = choices.getElementsByTagName("choice");
// For each choice...
// Get the important details
// Add them to the current bean
for (int i = 0; i < choiceList.getLength(); i++) {
 Element ce = (Element) choiceList.item(i);
 Choice c = new Choice();
 c.setId(ce.getAttribute("id"));
 for (Node child = ce.getFirstChild();
 child != null;
 child = child.getNextSibling()) {
 if ("choiceText".equals(child.getNodeName())) {
 c.setText(child.getFirstChild().getNodeValue());
 } else if ("correct".equals(child.getNodeName())) {
 c.setCorrect(child.getFirstChild().getNodeValue());
 }
 }
 v.add(c); // Add the bean to the holder
}
return v; // return the holder
}
```

## Section 2.1.4 - Test your program:

Once your program is complete, run the **PrintDomExam.java** driver program (or your equivalent) to test it. Here is our result:



```

Java - PrintExam.java - C:\XML4.0 - IBM WebSphere Studio Application Developer
File Edit Source Refactor Navigate Search Project Profile Run ClearCase Window Help
Console [terminated] C:\jdk1.4.2_03\bin\java.exe [1/23/04 11:11 AM]
Using iterative algorithm
Exam Reader results:
Exam: XML32-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====
1. DOM considers everything in an XML document as a:
< > A. Element
< > B. Tree Branch
< > C. Node *
< > D. Event
=====
2. Which of the following is not a valid DOM Node Type:
< > A. Element Node
< > B. Text Node
< > C. Entity Node
< > D. Root Node *
< > E. CDATASection Node
=====
3. The best way to traverse a DOM tree is:
< > A. Recursion *
< > B. From the bottom up
< > C. Event notification
< > D. Using Document.getNextElement()
=====
4. The DocumentBuilder can be used for:
[] A. DOM factory customization
[] B. Parsing an XML document *
[] C. Populating a DOM tree from a JavaBean
[] D. Creating a new Document *
[] E. None of the above
(Multiple answer question - Select all that apply)
=====
End of Exam
Time: 40

```

Note that a run required 40 milliseconds to complete; other trials produced a range from 50 - 30.

## Section 2.2 - Recursive Solution:

The use of a straight recursive walkthrough of the document, extracting the desired information as it is encountered.

Begin by changing the commenting so that `System.err.println("Using recursive algorithm");` `buildExamRecursive(doc);` is active and `System.err.println("Using iterative algorithm");` `buildExamDirect(doc);` is commented out of the stream.

### Section 2.2.1 - *buildExamRecursive()*

A method to build the document. This method shifts the load to `processElementNode()` where the actual work will occur.

\_\_\_ 1. At the outset our file looks like this:

```

75 private void buildExamRecursive(Node node) {
76 // Only bother with "Element" nodes, so test for ELEMENT_NODE
77 // Check the node name and take action on certain elements
78
79 // Recurse through tree
80 for (Node child = node.getFirstChild();
81 child != null;
82 child = child.getNextSibling()) {
83 buildExamRecursive(child);
84 }
85 }

```

\_\_\_ 2. Here is our solution after we let *Studio* add the **processElementNode()** method:

```

private void buildExamRecursive(Node node) {
 // Only bother with "Element" nodes, so test for ELEMENT_NODE
 if (node.getNodeType() == Node.ELEMENT_NODE) {
 processElementNode(node);
 }
 // Check the node name and take action on certain elements
 // Recurse through tree
 for (Node child = node.getFirstChild();
 child != null;
 child = child.getNextSibling()) {
 buildExamRecursive(child);
 }
}

```

### Section 2.2.2 - processElementNode()

\_\_\_ 1. At the outset our file looks like the above (lines 89-95).

\_\_\_ 2. Our solution follows. Notice that the line numbers have shifted: it was necessary to define two instance variables **currQuestion** and **currChoice** to keep track of where we are.

```

20 private Exam exam;
21 private int currQuestion = -1; // Pointer to current Question
22 private int currChoice = -1; // Pointer to current Choice

```

This, in turn, required the addition of getters and setters, which was accomplished by placing the cursor on any of these attributes and using *Studio* to create the appropriate code at the end of the file.

```

private void processElementNode(Node node)
 throws org.w3c.dom.DOMException {
 Question q = null;
 Choice c = null;
 // Pre-load current Question and choice if applicable
 if (getCurrQuestion() >= 0)
 q =
 (Question) getExam().getQuestions().elementAt(
 getCurrQuestion());
}

```

```
if (getCurrChoice() >= 0)
 c = (Choice) q.getChoices().elementAt(getCurrChoice());
// Check the node name and take action on certain elements
String name = node.getNodeName();
if (name.equals("test")) {
 setExam(new Exam());
 getExam().setId(((Element) node).getAttribute("id"));
} else if (name.equals("description")) {
 getExam().setDescription(node.getFirstChild().getNodeValue());
} else if (name.equals("question")) {
 // create new question, set currQuestion, reset currChoice
 getExam().getQuestions().add(new Question());
 setCurrQuestion(getCurrQuestion() + 1);
 setCurrChoice(-1);
 q =
 (Question) getExam().getQuestions().elementAt(
 getCurrQuestion());
 q.setId(((Element) node).getAttribute("id"));
} else if (name.equals("questionText")) {
 q.setText(node.getFirstChild().getNodeValue());
} else if (name.equals("choices")) {
 q.setAllowMultiple(((Element) node).getAttribute("allowMultiple"));
} else if (name.equals("choice")) {
 // Create new choice, set currChoice
 q.getChoices().add(new Choice());
 setCurrChoice(getCurrChoice() + 1);
 c = (Choice) q.getChoices().elementAt(getCurrChoice());
 c.setId(((Element) node).getAttribute("id"));
} else if (name.equals("choiceText")) {
 c.setText(node.getFirstChild().getNodeValue());
} else if (name.equals("correct")) {
 c.setCorrect(node.getFirstChild().getNodeValue());
}
}
```

## Section 2.2.3 - Test your program:

Once your program is complete,<sup>3</sup> run the **PrintDomExam.java** driver program (or your equivalent) to test it. Here is our result:

```
Using recursive algorithm
Exam Reader results:

Exam: XML32-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====

1. DOM considers everything in an XML document as a:

< > A. Element
< > B. Tree Branch
< > C. Node *
< > D. Event
=====

2. Which of the following is not a valid DOM Node Type:

< > A. Element Node
< > B. Text Node
< > C. Entity Node
< > D. Root Node *
< > E. CDATASection Node
=====

3. The best way to traverse a DOM tree is:

< > A. Recursion *
< > B. From the bottom up
< > C. Event notification
< > D. Using Document.getNextElement()
=====

4. The DocumentBuilder can be used for:

[] A. DOM factory customization
[] B. Parsing an XML document *
[] C. Populating a DOM tree from a Javabeen
[] D. Creating a new Document *
[] E. None of the above

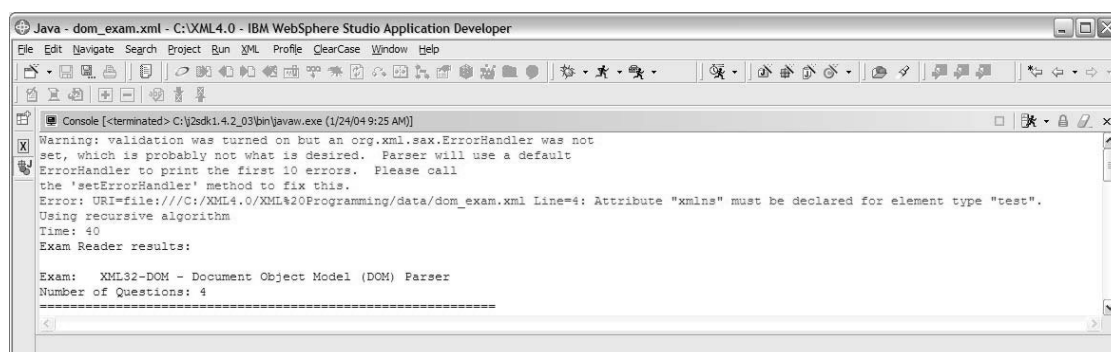
(Multiple answer question - Select all that apply)
=====

End of Exam

Time: 51
```

Note that a run required 51 milliseconds to complete; other trials produced a range from 40 - 30.

<sup>3</sup> We may not really be finished: introduce a namespace attribute in the root element that caused the XML instance to become invalid: `<test id="XML32-DOM" xmlns="http://www.ibm.com">`



Your notes contain additional information. The issue is where should we do validity testing? The right answer lies inside your group, inside your overall organization. There may not be a *single* answer. Maybe, it depends.

### ***Section 3 - Compare using DOM and SAX parsers to solve this business problem***

**Note:** This section assumes that you have previously performed the SAX Parser exercise. In this section, review how each parser tackled the same business problem. Some questions to think about are:

- \_\_\_ 1. Which parser type required the most effort on the programmer?  
\_\_\_\_\_
- \_\_\_ 2. Which parser type was easier to use in solving this problem?  
\_\_\_\_\_
- \_\_\_ 3. Which parser style would offer easier maintenance when the XML structure changes?  
\_\_\_\_\_
- \_\_\_ 4. Which parser style would be easier to use if the complexity of the document was to increase dramatically?  
\_\_\_\_\_
- \_\_\_ 5. Which DOM parser style (select or recursive) is faster?\_\_\_\_\_

***END OF LAB***

# Exercise 11. Generating XML from Java Objects

## What This Exercise Is About

You have been assigned the task of writing a program to convert exam data into an XML document. The goal of this assignment is to take a Java object graph containing exam data and write a program which converts the exam data into an XML document.

You will use two different techniques to accomplish this goal:

1. The first uses the XML persistence features of the J2SE, first introduced in version 1.4.
2. The second uses the Apache Xerces-J serializer classes and requires you to modify a set of Java objects so that they can be represented as XML.

In each case, you can use the supplied application class to test your success.

The exam data Java beans, a class that can parse exam data, and a sample application class have been provided as a starting point.

There are a total of three beans -- Exam, Question, and Choice. Exam holds information related to the entire exam and a Vector of Questions; the Question bean holds information for each Question, along with a Vector of Choices. The Choice bean contains information related to a single choice for a question.

## What You Should Be Able to Do

At the end of the lab, you should be able to:

- Use `java.beans.XMLEncoder` and `java.beans.XMLDecoder` instances to read and write JavaBeans
- Analyze a set of Java Beans and map their attributes into an XML document
- Convert Java Beans into XML by adding a `toXML` method to the Java Beans
- Convert Java Beans into a DOM tree
- Use the Apache Xerces-J serializer classes to convert a DOM Tree into an XML document

## Required Materials

- IBM's WebSphere Studio Application Developer IDE
- XML Parser (Xerces is included with Application Developer)
- XML document and DTD of an online exam
- Prebuilt Javabeans to hold exam data
- Prebuilt Application program
- J2SE v1.4 or higher

## Applicability

- XM321 Exercise 4
- XM341 Exercise 11



## Exercise Instructions

### Section 1 - Analyze Source Documents and Target Java beans

In this section of the lab, you will examine the source Exam XML document, its associated vocabulary, and the Java bean components that will hold the final information. Knowing what you have will aid you greatly in developing the application.

\_\_\_ 1. Open the Application Developer and locate the following resources:

Resource	Location / File	Description
Project	XML Programming	Project location for programming course resources
Source XML	<i>Folder data/generate_exam.xml</i>	XML containing Exam information to parse. Do <b>not</b> edit.
Source DTD	<i>Folder data/test.dtd</i>	DTD vocabulary for XML document. Do <b>not</b> edit.
	Package                  Class	
Target Java bean Classes	generate.lab   XMLExam.java generate.lab   XMLQuestion.java generate.lab   XMLChoice.java	Simple Java beans used to store exam data. <b>Incomplete.</b>
Application 1	generate.lab          Generate.java	Simple executable application to test your new program. <b>Incomplete.</b>
Application 2	generate.lab          Encode.java	Program you complete to read and write JavaBeans using XMLEncoders and XMLDecoders. <b>Incomplete.</b>
Exam Parser Class	generate.lab DOMExamReader.java	Helper class. <b>Do not edit.</b>

\_\_\_ 2. **Source XML and DTD:** Examine these files.

- \_\_\_ a. Familiarize yourself with the XML structure based on the DTD.
- \_\_\_ b. View both document's Source and Design tabs.

**Note:** Do not modify these files.

\_\_\_ 3. **Java bean Classes:** Examine the three target Java bean files (**XMLChoice.java**, **XMLExam.java**, **XMLQuestion.java**). Observe that these beans encapsulate the XML-related behavior only and *rely* on their base classes **cs01.Choice**, **Exam**, and **Question**, respectively for the domain related behavior.

- \_\_\_ 4. **Generate.java**: This class contains a simple **main()** method that parses an XML Data file to create an exam object. It then calls code that you supply to turn that exam object back into an XML file, and rereads the file to ensure that everything is correct. The name of the XML file must be provided as a command line argument.
- Run this program to check your progress as you work through the lab. Review section 2 of Lab 9 for instructions on how to setup command line arguments in Application Developer using a **Launch Configuration**. The name of the input file for this exercise is data/generate\_exam.xml.
- \_\_\_ 5. **Encode.java**: This class contains a simple **main()** method that parses an XML Data file to create an exam object. It then calls code that you supply to encode and decode the exam object using the new XML persistence features in J2SE v1.4. The name of the XML file must be provided as a command line argument.
- Run this program to check your progress as you work through the lab. Review section 2 of Lab 9 for instructions on how to set up command line arguments in Application Developer using a **Launch Configuration**. The name of the input file for this exercise is data/generate\_exam.xml.
- \_\_\_ 6. **DOMExamReader.java**: Examine this helper class. See how this file uses one of two possible methods to turn the DOM tree into a Java object. Do **not** modify this file.

**Notes:**

1. There are many possible solutions to this exercise. The end result should be a nicely formatted exam printed to stdout.
2. The intermediate XML files generated during this lab are stored in the temp directory associated with your user account, typically **c:\Documents and Settings\<user>\Local Settings\Temp**.<sup>1</sup> It is likely you will be interested in seeing the file contents.
3. Source for a working solution is provided at:  

```
XML Programming/solutions/generate/.java
```
4. The DOMExamReader.java file is the same as the one found in the solutions.dom package. It is duplicated here to avoid having to change the original.
5. *Studio 5.1* notes that where the node is used to refer the Node interface, in DOMExamReader.java at line 85, for example, `[if (node.getNodeType() == Node.ELEMENT_NODE)]` it should be a static reference, Node, not an instance reference, node, this does not prevent the application from running successfully but it does load our task list with information icons (an 'i' inside a triangle.)]

<sup>1</sup> You may want to **Search** on generated\*.xml for these temporary files.

## Section 2 - Serialize JavaBeans to XML using *java.beans.XMLEncoder* and *java.beans.XMLDecoder*

In this section of the lab you will modify the `main()` method in `Encode.java` to:

1. Write an object graph to a file as XML,
2. Re-create the object graph by reading the XML, and
3. Verify the structure of the recreated object graph by displaying the Exam on stdout.

### Section 2.1 - Encode the object graph

Locate the comment encode the object graph here (line 32, if you have enabled line numbers in Preferences) and replace it with Java instructions to use an `XMLEncoder` instance to write an Exam object graph (read by the `DOMExamReader` earlier in the method) to a file as XML.

\_\_\_ 1. Here's how we did it:<sup>2</sup>

```

25
26 File encodedFile = File.createTempFile("encode", ".xml");
27
28 System.out.println("Encoding Exam to XML...");
29 FileOutputStream fos = new FileOutputStream(encodedFile);
30
31 //encode the object graph here
32 XMLEncoder encoder = new XMLEncoder(fos);
33 encoder.writeObject(der.getExam());
34 encoder.close(); //done!
35
36 System.out.println("Done\n");
37
38 System.out.println("Reading encoded Exam...");
39 FileInputStream fis = new FileInputStream(encodedFile);
40 Exam anExam = null;
41

```

- \_\_\_ a. The new code is lines 32-35; we chose to display a little more to give you the context.

<sup>2</sup> Here's how means detailed instructions will follow. Look to these blocks for hints on a way to proceed.

- \_\_\_ b. This executes because we already set up the necessary imports:

```

1 package generate.lab;
2
3 import java.beans.XMLDecoder;
4 import java.beans.XMLEncoder;
5 import java.io.File;
6 import java.io.FileInputStream;
7 import java.io.FileOutputStream;
8
9 import cs01.Exam;
10

```

Notice we are using the already completed Exam bean, cs01.Exam.

- \_\_\_ c. You can create an entry in the Run configuration window. We created one called **Encode**. Part of the output is a file: encode29861.xml which contains formatting information so you'll want to use **Wordpad** or **Internet Explorer** to view this file. Here is a piece:

```

<?xml version="1.0" encoding="UTF-8" ?>
- <java version="1.4.2_03" class="java.beans.XMLDecoder">
- <object class="cs01.Exam">
 - <void property="description">
 <string>Document Object Model (DOM) Parser</string>
 </void>
 - <void property="id">
 <string>XML32-DOM</string>
 </void>
 - <void property="questions">
 - <void method="add">
 - <object class="cs01.Question">
 - <void property="choices">

```

This is what the console output looks like at this stage:

```

Console [<terminated> C:\j2sdk1.4.2_03\bin\javaw.exe (1/24/04 11:39 AM)]
Reading Exam from xml file
Done

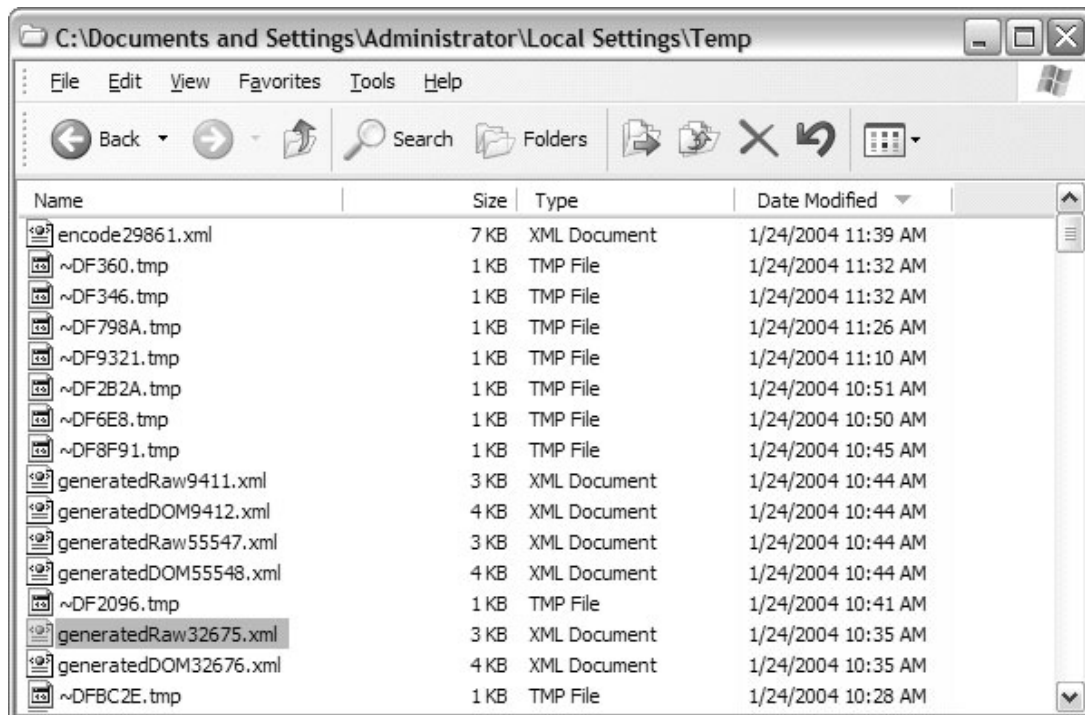
Encoding Exam to XML...
Done

Reading encoded Exam...
Done

Displaying Exam...
null
|

```

- \_\_\_ d. Here is a sample list of the files associated with this lab — so far!



## Section 2.2 - Decode the object graph

- \_\_\_ 1. Locate the comment decode the object graph here [now lines 42-44] and replace it with Java instructions to:
  - \_\_\_ a. Use an XMLDecoder instance to parse the source information and reconstruct the object graph, and
  - \_\_\_ b. Display the reconstructed Exam instance on System.out to verify that it still represents the Exam you started with.
- \_\_\_ 2. Here's how we did it:
  - \_\_\_ a. As we begin the modification we no longer need line 40, Exam anExam=null; which was there to permit a clean compile. We will comment it out of the stream.

```

38 System.out.println("Reading encoded Exam...");
39 FileInputStream fis = new FileInputStream(encodedFile);
40// Exam anExam = null;
41
42 //decode the object graph here
43 XMLDecoder decoder = new XMLDecoder(fis);
44 Exam anExam = (Exam)decoder.readObject();
45 decoder.close();
46 //done!
47 System.out.println("Done\n");
48
49 System.out.println("Displaying Exam...");

```

50      `System.out.println(anExam);`

**Note:** if you choose to copy from the solutions folder, you will have to change the 'e' in line 44 to 'anExam' or change 'anExam' in line 50 to 'e'. We don't recommend the latter because on line 52 we have **Exception e** and even though it is in a different scope it could be confusing.

## Section 2.3 - Test your Work.

Run the **Encode.java** program again to check your success. The output should look like this.

```

Java - Encode.java - C:\XML4.0 - IBM WebSphere Studio Application Developer
File Edit Source Refactor Navigate Search Project Profile ClearCase Run Window Help
Console [terminated] C:\2ask1.4.2_03\bin\java.exe (1/24/04 12:30 PM)

Done
Encoding Exam to XML...
Done
Reading encoded Exam...
Done
Displaying Exam...
Exam: XML42-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====
1. DOM considers everything in an XML document as a:
 < > A. Element
 < > B. Tree Branch
 < > C. Node *
 < > D. Event
=====
2. Which of the following is not a valid DOM Node Type:
 < > A. Element Node
 < > B. Text Node
 < > C. Entity Node
 < > D. Root Node *
 < > E. CDATASection Node
=====
3. The best way to traverse a DOM tree is:
 < > A. Recursion *
 < > B. From the bottom up
 < > C. Event notification
 < > D. Using Document.getNextElement()
=====
4. The DocumentBuilder can be used for:
 [] A. DOM factory customization
 [] B. Parsing an XML document *
 [] C. Populating a DOM tree from a JavaBean
 [] D. Creating a new Document *
 [] E. None of the above
(Multiple answer question - Select all that apply)
=====
End of Exam

```

You have de-constructed and re-constructed the exam.

## Section 3 - Generate XML by adding a toXML() method

In this section of the lab, you will modify the toXML() methods of the XMLChoice.java, XMLQuestion.java and XMLExam.java classes to emit XML that represents the object in its current state. Remember that the instances of these classes are related to each other, that is, an XMLQuestion has a collection of XMLChoices, and so forth. This means that, in addition to object specific XML, an Exam must also emit Questions and similarly, Questions must emit Choices.

### Section 3.1 - XMLChoice.java

- \_\_\_ 1. Fill in the body of String toXML() at line 17.
- \_\_\_ 2. Here's how we did it; don't forget to replace the return null line necessary for a clean compile:

```

16 public String toXML() {
17 /* Fill in for Section 3.1 */
18 StringBuffer sb = new StringBuffer();
19 sb.append("<choice id=\"" + getId() + "\">\n");
20 sb.append(" <choiceText>");
21 sb.append(getText());
22 sb.append(" </choiceText>\n");
23 sb.append(" <correct>");
24 sb.append(getCorrect());
25 sb.append(" </correct>\n");
26 sb.append("</choice>\n");
27 return sb.toString(); //return null;
28 }

```

### Section 3.2 - XMLQuestion.java

- \_\_\_ 1. Fill in the body of String toXML() at Line 18. Remember to handle the embedded Choices.
- \_\_\_ 2. Here's how we did it; don't forget to replace the return null line necessary for a clean compile with the actual string:

```

18 public String toXML() {
19 /* Fill in for Section 3.2 */
20 StringBuffer sb = new StringBuffer();
21 sb.append("<question id=\"" + getId() + "\">\n");
22 sb.append(" <questionText>");
23 sb.append(getText());
24 sb.append(" </questionText>\n");
25 sb.append(" <choices allowMultiple=\"" + getAllowMultiple() + "\">");
26 Vector choiceVector = getChoices();
27 for (int i = 0; i < choiceVector.size(); i++) {
28 sb.append(new XMLChoice((Choice) choiceVector.get(i)).toXML());
29 }
30 sb.append(" </choices>\n");
31 sb.append("</question>\n");
32 return sb.toString(); //return null;
33 }

```

### Section 3.3 - XMLExam.java

- \_\_\_ 1. Fill in the body of String toXML() at Line 18. Again, remember to handle the embedded Questions.
- \_\_\_ 2. Here's how we did it; don't forget to replace the return null line necessary for a clean compile with the actual string:

```

18 public String toXML() {
19 /* Fill in for Section 3.3 */
20 StringBuffer sb = new StringBuffer();
21 sb.append("<test id=\"" + getId() + "\">\n");

```

```

22 sb.append(" <description>");
23 sb.append(getDescription());
24 sb.append(" </description>\n");
25 sb.append(" <testQuestions>");
26 Vector questionVector = getQuestions();
27 for (int i = 0; i < questionVector.size(); i++) {
28 sb.append(new XMLQuestion((Question)
questionVector.get(i)).toXML());
29 }
30 sb.append(" </testQuestions>\n");
31 sb.append("</test>\n");
32 return sb.toString();//return null;
33 }

```

### Section 3.4 - Run Generate.java

Check it over and then run it. Here's what ours looks like:

```

Console [terminated] C:\jdk1.4.2_03\bin\javaw.exe (1/24/04 1:48 PM)
Reading input xml file
Generating xml from raw code
Verifying raw generation...
Done.
Displaying Exam...
Exam: XML32-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====
1. DOM considers everything in an XML document as a:

< > A. Element
< > B. Tree Branch
< > C. Node *
< > D. Event
=====
2. Which of the following is not a valid DOM Node Type:

< > A. Element Node
< > B. Text Node
< > C. Entity Node
< > D. Root Node *
< > E. CDATASection Node
=====
3. The best way to traverse a DOM tree is:

< > A. Recursion *
< > B. From the bottom up
< > C. Event notification
< > D. Using Document.getNextElement()
=====
4. The DocumentBuilder can be used for:

[] A. DOM factory customization
[] B. Parsing an XML document *
[] C. Populating a DOM tree from a Javabeen
[] D. Creating a new Document *
[] E. None of the above

(Multiple answer question - Select all that apply)
=====
End of Exam

```



## Section 4 - Generate XML by adding a toDOM method and calling the Apache Xerces-J serializers.

In this section of the lab, you will modify the **Element toDOM(Document)** methods of the **XMLChoice.java**, **XMLQuestion.java** and **XMLExam.java** classes. These methods return a DOM tree fragment representing the XML for that object. Again, remember that these classes are related to each other, that is, an XMLQuestion has a collection of XMLChoices, and so forth. You'll need to use the **appendChild** method provided by the **org.w3c.dom.Node** interface when adding the siblings of each <question> in the DOM tree (and any other hierarchically nested elements).

### Section 4.1 - XMLChoice.java

\_\_\_ 1. Fill in the body of Element toDOM() at Line 30.

\_\_\_ 2. Here's our solution:

```

30 public Element toDOM(Document doc) {
31 /* Fill in for Section 4.1 */
32 Element correctElt = doc.createElement("correct");
33 correctElt.appendChild(doc.createTextNode(getCorrect()));
34 Element choiceTextElt = doc.createElement("choiceText");
35 choiceTextElt.appendChild(doc.createTextNode(getText()));
36 Element choiceElt = doc.createElement("choice");
37 choiceElt.setAttribute("id", getId());
38 choiceElt.appendChild(choiceTextElt);
39 choiceElt.appendChild(correctElt);
40 return choiceElt; //return null;
41 }

```

### Section 4.2 - XMLQuestion.java

\_\_\_ 1. Fill in the body of Element toDOM() at Line 35.

\_\_\_ 2. Here's our solution:

```

34
35 public Element toDOM(Document doc) {
36 /* Fill in for Section 4.2 */
37 Element choicesElt = doc.createElement("choices");
38 Vector choiceVector = getChoices();
39 for (int i = 0; i < choiceVector.size(); i++) {
40 choicesElt.appendChild(new XMLChoice((Choice)
choiceVector.get(i)).toDOM(doc));
41 }
42 choicesElt.setAttribute("allowMultiple", getAllowMultiple());
43 Element questionTextElt = doc.createElement("questionText");
44 questionTextElt.appendChild(doc.createTextNode(getText()));
45 Element questionElt = doc.createElement("question");
46 questionElt.setAttribute("id", getId());
47 questionElt.appendChild(questionTextElt);

```

```
48 questionElt.appendChild(choicesElt);
49 return questionElt; // return null;
50 }
```

### Section 4.3 - XMLExam.java

\_\_\_ 1. Fill in the body of Element toDOM() at Line 36.

\_\_\_ 2. Here's our solution:

```
34 public Element toDOM(Document doc) {
35 /* Fill in for Section 4.3 */
36 Element questionsElt = doc.createElement("testQuestions");
37 Vector questionVector = getQuestions();
38 for (int i = 0; i < questionVector.size(); i++) {
39 questionsElt.appendChild(
40 new XMLQuestion((Question) questionVector.get(i)).toDOM(doc));
41 }
42 Element descriptionElt = doc.createElement("description");
43 descriptionElt.appendChild(doc.createTextNode(getDescription()));
44 Element testElt = doc.createElement("test");
45 testElt.setAttribute("id", getId());
46 testElt.appendChild(descriptionElt);
47 testElt.appendChild(questionsElt);
48 return testElt; // return null;
49 }
```

### Section 4.4 - Generate.java

\_\_\_ 1. Use the Xerces-J serializer classes to serialize the DOM tree as XML, using UTF-8 encoding. Use the output stream provided as the destination.

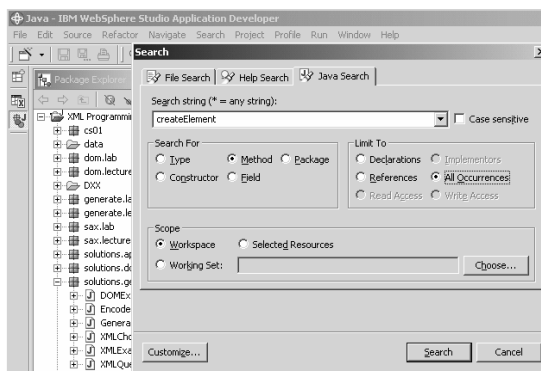
- \_\_\_ a. Uncomment the block of code at about Line 40. [See the next page, first if you want our hints.]
- \_\_\_ b. Insert your code in the area directed by the comments.
- \_\_\_ c. Run your program when you are finished.

#### General Hints:

- You'll need the following methods on the **org.w3c.dom.Document** interface in order to create new DOM nodes:<sup>3</sup>
  - createElement(String elementName)
  - createTextNode(String textContent)

<sup>3</sup> org.w3c.dom can be found in the XERCES\_API\_JAR we set up in Day 1; you can use the Search feature of Studio starting with **Search** in the menu bar at the top:

- You'll need the following methods on the `org.w3c.dom.Element` interface:
  - `setAttribute(String attributeName, String attributeValue)`



- You'll need the following methods on the `org.w3c.dom.Node` interface (Document and element are subtypes of node)
  - `appendChild(Node)`**

2. Here is our solution; we duplicated our instructions instead of simply removing the comment tags to make it easier for you to follow:

```

29
30 System.out.println("Generating xml from raw code");
31 XMLExam e = new XMLExam (der.getExam()); //ERC4.01 5/13/04
32 File generated = File.createTempFile("generatedRaw", ".xml"); //ERC4.01
5/13/04
33 FileOutputStream ofs = new FileOutputStream(generated);
34 PrintStream ps = new PrintStream(ofs);
35 ps.print(e.toXML());
36 System.out.println("Verifying raw generation...");
37 DOMExamReader newDoc = new DOMExamReader(generated.getAbsolutePath());
38 System.out.println("Done.");
39
40 /* Uncomment out for Section 4.4
41
42 System.out.println("Generating DOM from Java objects");
43 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
44 DocumentBuilder db = dbf.newDocumentBuilder();
45 Document doc = db.newDocument();
46 doc.appendChild(e.toDOM(doc));
47
48 generated = File.createTempFile("generatedDOM", ".xml");
49 ofs = new FileOutputStream(generated);
50 //
51 // Insert Java code here to use an XMLSerializer to serialize
52 // the Document created above
53 //
54 OutputFormat format = new OutputFormat("xml", "UTF-8", true);
55 BaseMarkupSerializer serializer = new XMLSerializer(ofs, format);

```

```
56 serializer.serialize(doc);
57 System.out.println("Verifying DOM generation...");
58 newDoc = new DOMExamReader(generated.getAbsolutePath());
59 System.out.println("Done\n");
60
61 */
62 System.out.println("Generating DOM from Java objects");
63 DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
64 DocumentBuilder db = dbf.newDocumentBuilder();
65 Document doc = db.newDocument();
66 doc.appendChild(e.toDOM(doc));
67
68 generated = File.createTempFile("generatedDOM", ".xml");
69 ofs = new FileOutputStream(generated);
70 //
71 OutputFormat format = new OutputFormat("xml", "UTF-8", true);
72 BaseMarkupSerializer serializer = new XMLSerializer(ofs, format);
73 serializer.serialize(doc);
74 //
75 System.out.println("Verifying DOM generation...");
76 newDoc = new DOMExamReader(generated.getAbsolutePath());
77 System.out.println("Done\n");
78
```

- \_\_\_ a. As you can see, the serializer work is being done in lines 71-73.
- \_\_\_ b. Run Generate.java, your output should resemble ours:

```

Java - C:\XML4.0 - IBM WebSphere Studio Application Developer
File Edit Navigate Search Project Profile ClearCase Run Window Help
[Icons]
Console [terminated] C:\jdk1.4.2_03\bin\javaw.exe (1/24/04 2:32 PM)
Reading input xml file
Generating xml from raw code
Verifying raw generation...
Done.
Generating DOM from Java objects
Verifying DOM generation...
Done
Displaying Exam...
Exam: XML32-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====
1. DOM considers everything in an XML document as a:
< > A. Element
< > B. Tree Branch
< > C. Node *
< > D. Event
=====
2. Which of the following is not a valid DOM Node Type:
< > A. Element Node
< > B. Text Node
< > C. Entity Node
< > D. Root Node *
< > E. CDATASection Node
=====
3. The best way to traverse a DOM tree is:
< > A. Recursion *
< > B. From the bottom up
< > C. Event notification
< > D. Using Document.getNextElement()
=====
4. The DocumentBuilder can be used for:
[] A. DOM factory customization
[] B. Parsing an XML document *
[] C. Populating a DOM tree from a JavaBean
[] D. Creating a new Document *
[] E. None of the above
(Multiple answer question - Select all that apply)
=====
End of Exam

```

**END OF LAB**



## Exercise 12. XSLT and Java

### What This Exercise is About

In this exercise, you will use XSLT to transform exam information from a vendor-specific XML vocabulary into an XML format that the system can understand.

You have been assigned the task of writing a program that exchanges data with an external provider of exam test questions. This external provider supplies their questions in XML, but they use their own vocabulary, which of course, is different from the vocabulary used by our system. Your approach to solving the problem will be to use XSLT to transform incoming questions into the format used by the exam system, and to use XSLT to transform outgoing questions into the format used by the external provider.

The exam data Java beans, a class that can parse exam data, and a sample application class have all been provided as a starting point.

Sample exam data, DTDs for the exam, and the external provider's format are also provided.

There are a total of three beans -- Exam, Question, and Choice. Exam holds information related to the entire exam and a Vector of Questions.

The Question bean holds information for each Question, along with a Vector of Choices. The Choice bean contains information related to a single choice for a question.

### What You Should Be Able to Do

At the end of the lab, you should be able to:

- Interpose an XSLT processor at the input and output stages of an application
- Use a DOM-based interface to a JAXP compliant XSLT processor
- Use a SAX-based interface to a JAXP XSLT processor

### Required Materials

- IBM's WebSphere Studio Application Developer 5.0 or above
- XML Parser (Xerces is included with Application Developer)
- J2SE 1.4 or later (provides the required Xalan implementation)

- XML document and DTD of an online exam
- Prebuilt Java beans to hold exam data
- Prebuilt Application program

## **Applicability**

- XM321 Exercise 5
- XM341 Exercise 12



## Exercise Instructions

### Section 1 - Analyze Source documents and Target Java beans

In this section of the lab, you will examine the source Exam XML document, its associated vocabulary, and the Java bean components that will hold the final information. Knowing what you have will aid you greatly in developing the application.

\_\_\_ 1. Open the Application Developer and locate the following resources:

Resource	Location / File	Description
Project	XML Programming	Project location for all course resources.
Internal XML	data/internal.xml	XML containing Exam information to parse in internal format. Do not edit.
Internal DTD	data/test.dtd	DTD defining the vocabulary for the internal exam format. Do not edit.
Provider XML	data/provider.xml	XML containing Exam information in provider.s vocabulary. Do not edit.
Provider DTD	data/exam.dtd	DTD defining the vocabulary for the provider.s exam format. Do not edit.
Internal to Provider XSL	data/internal2provider.xsl	XSLT stylesheet to convert exam information from the internal format to the provider.s format. Do not edit.
Provider to Internal XSL	data/provider2internal.xsl	XSLT stylesheet to convert exam information from the provider.s format to internal format. Do not edit.
Target Java-bean Classes	xslt.lab.XMLExam.java xslt.lab.XMLQuestion.java xslt.lab.XMLChoice.java	Simple Java beans used to store exam data. Do not edit.
Application	xslt.lab.DOMToProvider.java xslt.lab.SAXFromProvider.java	Simple application skeletons for you to build on. Include indicators where code should be filled in.
Exam Parser Class	xslt.lab.DOMExamReader.java xslt.lab.SAXExamReader.java xslt.lab.SAXExam-Reader-Base.java	Base class that implements Default Handler and provides simple error handler methods. Do not edit.

- \_\_\_ 2. provider.xml, exam.dtd, internal.xml, test.dtd, internal2provider.xsl and provider2internal.xsl: Examine these files, view both the Source and Design tabs for all documents. Note: Do not modify these files.
- \_\_\_ 3. Java bean Classes: Examine the three target Java bean files: XMLChoice.java, XMLExam.java, XMLQuestion.java. Examine the toXML() and toDOM() methods.

- \_\_\_ 4. **DOMToProvider.java:** This *Application* class contains a simple `main()` method that parses an XML Data file to create an exam object. It then calls code that you supply to transform that exam object into XML conforming to the customer's grammar. The name of the XML file to parse must be provided as a command line argument.

Run this program to check your progress as you work through the lab. Review section 2 of Lab 9 for instructions on how to setup command line arguments in Application Developer using the **Launch Configuration Window**. The name of the input file for this exercise is **data/internal.xml**.

- \_\_\_ 5. **DOMExamReader.java:** Examine this helper class; look at the **getExam()** method. Do not modify this file.

- \_\_\_ 6. **SAXFromProvider.java:** This *Application* class contains a **main()** method that assumes the existence of an Exam object. It prints the contents of that Exam object out at the end of the run. Add the provider.xml file name to the Command Arguments of this class.

Run this program to check your progress as you work through the lab. Review section 2 of Lab 9 for instructions on how to setup command line arguments in Application Developer using the **Launch Configuration Window**. The name of the input file for this exercise is **data/provider.xml**.

- \_\_\_ 7. **SAXExamReader.java and SAXExamReaderBase.java:** Examine these helper classes:

- \_\_\_ c. See how `SAXExamReader.java` uses SAX to convert XML into an Exam instance.
- \_\_\_ d. Review the `getExam()` method.

Do not modify these files.

As you can see, all our efforts will be focused on the two applications:  
`xslt.lab.DOMToProvider.java` and `xslt.lab.SAXFromProvider.java`.

## ***Section 2 - Convert existing Exam objects to Customer's Vocabulary: DOMToProvider.java***

In this section of the lab, you will modify **DOMToProvider.java**. You should use the TRaX API to get an instance of an XSLT processor and use that processor to convert an Exam object into the provider's XML format.

- \_\_\_ 1. Your changes should be restricted to the body of the **toProvider(Exam exam)** method, a Java comment block has been provided at Lines 52-54 to identify where your changes should go. Use the **internal2provider.xsl** stylesheet to do the conversion.

Use the following strategy:

- \_\_\_ a. Invoke the **XMLEexam.toDOM()** method to convert an Exam to a DOM tree

- \_\_\_ b. Provide the DOM tree to the XSLT processor as input.
- \_\_\_ c. Return the resulting DOM tree from the toProvider method.
- \_\_\_ 2. Here's how we did it:<sup>1</sup>

```

48 if (tFactory.getFeature(DOMSource.FEATURE)
49 && tFactory.getFeature(DOMResult.FEATURE)) {
50 /*
51 Your code goes here
52 */
53 //Instantiate a DocumentBuilderFactory.
54 DocumentBuilderFactory dFactory = DocumentBuilderFactory.newInstance();
55
56 // And setNamespaceAware, which is required when parsing xsl files
57 dFactory.setNamespaceAware(true);
58
59 DocumentBuilder db = dFactory.newDocumentBuilder();
60
61 // create a DOM from the xsl
62 Document xslDoc = db.parse(DOC_XSL);
63
64 // create the xslDOMSource for the Transformer and set the System ID
65 DOMSource xslDomSource = new DOMSource(xslDoc);
66 xslDomSource.setSystemId(DOC_XSL);
67
68 // Create a Transformer.
69 Transformer transformer = tFactory.newTransformer(xslDomSource);
70
71 // create a document to hold the source DOM
72 Document doc = db.newDocument();
73
74 // build the source DOM from the Exam and set the doc root
75 doc.appendChild(new XMLExam(exam).toDOM(doc));
76
77 // create the DOMSource
78 DOMSource xmlDomSource = new DOMSource(doc);
79
80 // Create an empty DOMResult for the Result.
81 DOMResult domResult = new DOMResult();
82
83 // Perform the transformation, placing the output in the DOMResult.
84 transformer.transform(xmlDomSource, domResult);
85
86 // return the result of the root node
87 return domResult.getNode();// return null;
88 }

```

<sup>1</sup> "Here's how" or words to that effect, signal that what follows are detailed instructions should you want them.

- \_\_\_ 3. Create a new entry in the Run Configuration Window for **DOM To Provider** (our name) and run it with the file **data/internal.xml** as an Argument. Here is our result:

If you run it as is:



You can see why by examining the code in XMLQuestion.java: everything is *appended*. Go ahead and make the necessary changes if you wish.

## Sample output of DOMToProvider.java:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE exam SYSTEM "exam.dtd">
<exam id="XML32-DOM" title="Document Object Model (DOM) Parser">
 <question id="1" multiple="No">
 <text>DOM considers everything in an XML document as a:</text>
 <option correct="No" id="A">Element</option>
 <option correct="No" id="B">Tree Branch</option>
 <option correct="Yes" id="C">Node</option>
 <option correct="No" id="D">Event</option>
 </question>
 <question id="2" multiple="No">
 <text>Which of the following is not a valid DOM Node Type:</text>
 <option correct="No" id="A">Element Node</option>
 <option correct="No" id="B">Text Node</option>
 <option correct="No" id="C">Entity Node</option>
 <option correct="Yes" id="D">Root Node</option>
 <option correct="No" id="E">CDATASection Node</option>
 </question>
 <question id="3" multiple="No">
 . . .
```

## Section 3 - Convert the Customer's XML into Exam objects: SAXFromProvider.java

In this section of the lab, you will modify **SAXFromProvider.java**. You should use the TRaX API to get an instance of the XSLT processor, and then use that processor to convert an Exam object in the provider's XML format into the format understood by SAXExamReader.

Once you have done this, you can feed the input into SAXExamReader to create an Exam object.

- \_\_\_ 1. **SAXFromProvider.java** - Your changes should be restricted to the body of the **main(String args[])** method, a Java comment block has been provided at Lines 16-18 to identify where your changes should go. Use the **provider2internal.xsl** stylesheet to do the conversion.  
 Use the following strategy:
  - \_\_\_ a. Create a TransformerFactory instance and use it as a SAXTransformerFactory. Don't forget to confirm that the platform supports this capability.
  - \_\_\_ b. Create a ContentHandler to handle the parsing of the stylesheet.
  - \_\_\_ c. Create an XMLReader and set its ContentHandler to the handler just created.
  - \_\_\_ d. Parse the stylesheet.
  - \_\_\_ e. Get the Templates instance from the ContentHandler.
  - \_\_\_ f. Using the Templates instance, create a TransformerHandler to handle parsing of the XML source.
  - \_\_\_ g. Set the XMLReaders' ContentHandler to the newly created TransformerHandler.
  - \_\_\_ h. Configure the XMLReaders' ContentHandler to function also as a LexicalHandler, which includes lexical events (for example, comments and CDATA).
  - \_\_\_ i. Set the result handling of the TransformerHandler to use a SaxExamReader instance
  - \_\_\_ j. Parse the XML input document.
  - \_\_\_ k. Write the examReaders' exam to stdout.
- \_\_\_ 2. Here's how we did it; we left the comment lines in for orientation; don't forget to add the import packages:

```

1 package xslt.lab;
2 // import cs01.Exam;
3 import javax.xml.transform.Result;
4 import javax.xml.transform.Templates;
5 import javax.xml.transform.TransformerFactory;
6 import javax.xml.transform.sax.SAXResult;
7 import javax.xml.transform.sax.SAXSource;
8 import javax.xml.transform.sax.SAXTransformerFactory;
9 import javax.xml.transform.sax.TemplatesHandler;
10 import javax.xml.transform.sax.TransformerHandler;
11
12 import org.xml.sax.XMLReader;
13 import org.xml.sax.helpers.XMLReaderFactory;
14 public class SAXFromProvider {
15 public static void main(String[] args) {

```

```
16 try {
17 if (args.length != 1) {
18 System.err.println("Usage: SAXFromProvider provider.xml");
19 return;
20
21
22 SAXExamReader examReader = null;
23
24
25 /*
26 Your code goes here
27 */
28
29 // Instantiate a TransformerFactory.
30 TransformerFactory tFactory = TransformerFactory.newInstance();
31
32 // Determine whether the TransformerFactory supports The use of
33 SAXSource
34 // and SAXResult
35 if (tFactory.getFeature(SAXSource.FEATURE)
36 && tFactory.getFeature(SAXResult.FEATURE)) {
37
38 // Cast the TransformerFactory.
39 SAXTransformerFactory saxTFactory =
40 ((SAXTransformerFactory) tFactory);
41
42 // Create a ContentHandler to handle the parsing of the stylesheet.
43 TemplatesHandler templatesHandler =
44 saxTFactory.newTemplatesHandler();
45
46 // Create an XMLReader and set its ContentHandler to the handler just
47 created.
48 XMLReader reader = XMLReaderFactory.createXMLReader();
49 reader.setContentHandler(templatesHandler);
50
51 // Parse the stylesheet.
52 reader.parse(DOC_XSL);
53
54 //Get the Templates instance from the ContentHandler.
55 Templates templates = templatesHandler.getTemplates();
56
57 // Using the Templates instance, create a TransformerHandler to handle
58 parsing of the XML source.
59 TransformerHandler handler =
60 saxTFactory.newTransformerHandler(templates);
61
62 // Set the XMLReaders' ContentHandler to the newly created
63 TransformerHandler.
64 reader.setContentHandler(handler);
```

```

61
62 // Configure the XMLReaders' ContentHandler to function also as a
LexicalHandler, which
63 // includes "lexical" events (e.g., comments and CDATA).
64 reader.setProperty(
65 "http://xml.org/sax/properties/lexical-handler",
66 handler);
67
68 // Set the result handling of the TransformerHandler to use a
SaxExamReader instance
69 examReader = new SAXExamReader();
70 Result result = new SAXResult(examReader);
71 handler.setResult(result);
72
73 // Parse the XML input document.
74 reader.parse(args[0]);
75
76 // Write the examReader's exam to stdout.
77
78
79 System.out.println(examReader.getExam());
80 } else
81 System.out.println(
82 "The TransformerFactory does not support SAX input and SAX
output");
83
84 }catch (Exception e) {
85 e.printStackTrace();
86 }
87 }
88 }

```

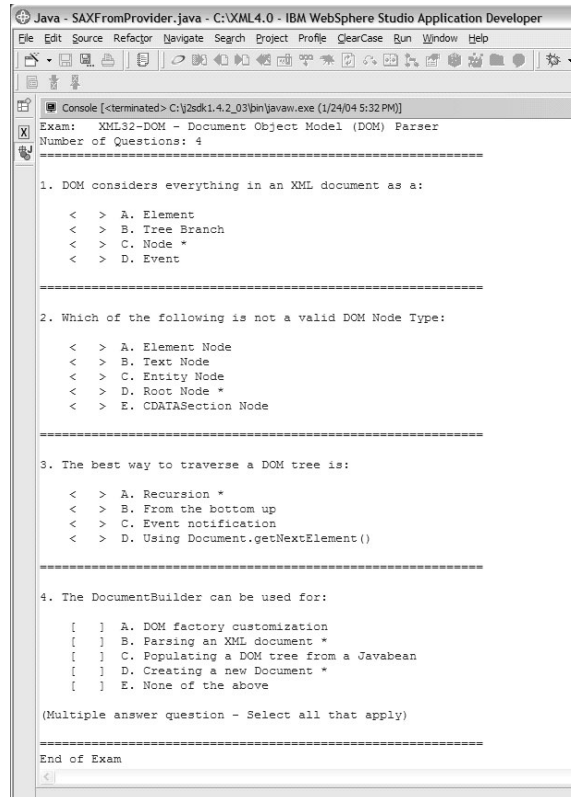
...and don't forget to add the stylesheet:

```

11
12 import org.xml.sax.XMLReader;
13 import org.xml.sax.helpers.XMLReaderFactory;
14 public class SAXFromProvider {
15 private static final String DOC_XSL = "data/provider2internal.xsl";
16 // provider to internal transform
17 public static void main(String[] args) {

```

\_\_\_ 3. Run the result using **data/provider.xml** for the argument:



The screenshot shows a Java IDE window titled "Java - SAXFromProvider.java - C:\XML4.0 - IBM WebSphere Studio Application Developer". The console output displays an exam titled "Exam: XML32-DOM - Document Object Model (DOM) Parser" with 4 questions. The questions are multiple-choice and cover DOM concepts. The output is as follows:

```
Console [terminated> C:\jdk1.4.2_03\bin\javaw.exe (1/24/04 5:32 PM)]
Exam: XML32-DOM - Document Object Model (DOM) Parser
Number of Questions: 4
=====
1. DOM considers everything in an XML document as a:
< > A. Element
< > B. Tree Branch
< > C. Node *
< > D. Event
=====
2. Which of the following is not a valid DOM Node Type:
< > A. Element Node
< > B. Text Node
< > C. Entity Node
< > D. Root Node *
< > E. CDATASection Node
=====
3. The best way to traverse a DOM tree is:
< > A. Recursion *
< > B. From the bottom up
< > C. Event notification
< > D. Using Document.getNextElement()
=====
4. The DocumentBuilder can be used for:
[] A. DOM factory customization
[] B. Parsing an XML document *
[] C. Populating a DOM tree from a Javabeen
[] D. Creating a new Document *
[] E. None of the above
(Multiple answer question - Select all that apply)
=====
End of Exam
<|
```

**END OF LAB**



# Appendix A. Exercise Solutions

## Exercise 3

6. Remove the "`![CDATA[. . .]]`" from any of the choices in Q1 in the exam.xml you opened in Step 2, above (leave the `<!ELEMENT...>` declaration). What happens?

Answer: The parser tries to use the result as part of the schema rather than a (dumb) value, so you are told that "the content of elements must consist of well-formed character data or markup." This, of course, is what the CDATA does: it tells the parser to ignore the characters it doesn't like.

## Exercise 4

### Section 1 - Associate items with Namespaces.

- \_\_\_ 1. Here are the names of three Namespaces. For each Namespace, can you suggest additional elements that would belong in it and not in the other two?
- \_\_\_ a. `http://www.somecompany.com/farming`  
Possible Answer: `farmingType`(`rancher` | `farmer`) `equipmentCategory`, `crops`, `livestock`, `silo`, `subsidy`, `farmhouse`
- \_\_\_ b. `http://www.somecompany.com/plants`  
Possible Answer: `commonName`, `species`, `roots`, `habitat` (`indoor/outdoor`), `lifecycle` (`annual`, `perennial`)
- \_\_\_ c. `http://www.somecompany.com/gardening`  
Possible Answer: `gardenKind` (`flower` | `vegetable` | `combination`), `statuary`, `bed shape`, `snail bait`, `planter box`
- \_\_\_ 2. In which namespaces would you find 'soil\_type'? (You can create an additional namespace if you feel it is necessary.) You should be prepared to defend your decision in discussion with the class.

---

Answer: [There is no right answer for Question 2. It is intended to get you thinking about balancing the idea of maintaining namespace distinctness against the difficulty of doing so.]

## Exercise 6

### Section 4 - XPath - Solutions

#### Easy - as pie...

- \_\_\_ 1. Extract the **ID** of this test.
- `/test/@id`

- \_\_\_ 2. Extract all **questionText** elements.  
    **/test/testQuestions/question/questionText**  
    Or **//questionText**
- \_\_\_ 3. Extract all the **ID** attributes from this document.  
    **//@id**

**Medium - You may need to think about these**

- \_\_\_ 1. Extract the second to the last **question** element on this test.  
    **//question[position()=last()-1]**
- \_\_\_ 2. Extract the **questionText** element of question number (ID) "Q4".  
    **//question[@id='Q4']/questionText**
- \_\_\_ 3. Extract the first **choice** descendant of the last **question** element in the document.  
    **//question[last()]/choice[1]**
- \_\_\_ 4. Extract all the text in the document.  
    **//text()**
- \_\_\_ 5. Extract the **ID** of the last **choice** for each **question**.  
    **//question//choice[last()]/@id Or //choice[last()]/@id**

**Hard - So you really think you know XPath now?**

- \_\_\_ 1. Extract the text of the correct choice of question ID Q2.  
    **//question[@id='Q2']/choice[correct='Yes']/choiceText/text()**
- \_\_\_ 2. Extract the ID of all questions that allow multiple selected answers.  
    **//choices[@allowMultiple='Yes']/../@id**
- \_\_\_ 3. Extract the ID of all questions with more than four choices.  
    **//choices[count(choice)>4]/../@id**
- \_\_\_ 4. Extract any **questionText** element containing the word "XPath" in its text.  
    **//questionText[contains(text(),'XPath')]**
- \_\_\_ 5. Extract the text of any choices that contain the string Ancestor.  
    **//choice[contains(choiceText,'Ancestor')]/choiceText/text()**
- \_\_\_ 6. Extract the text of all correct choices in the test.  
    **//choice/correct[text()='Yes']/../choiceText/text()**



