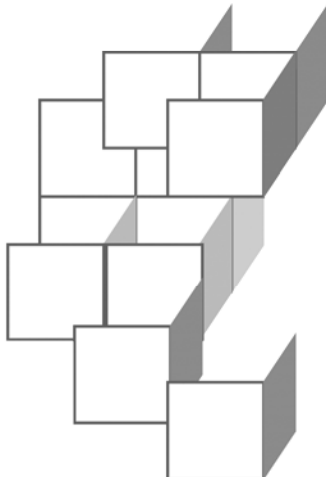




CHAPTER 1
USING JAXP FOR XML
TRANSFORMATIONS



OBJECTIVES

After completing “Using JAXP for XML Transformations,” you will be able to:

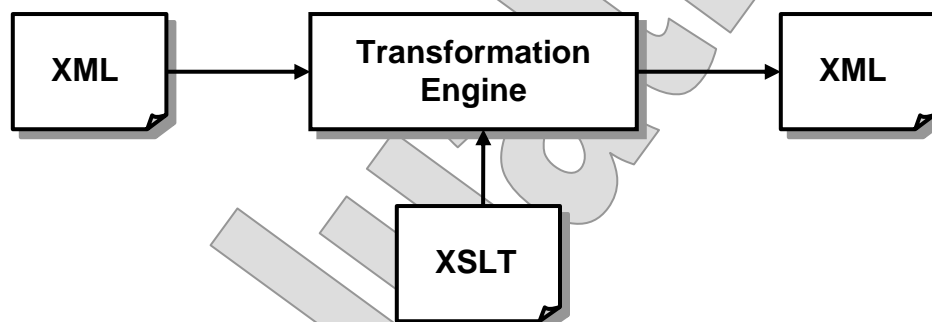
- Describe the role of JAXP in XML transformations.
- Instantiate a **Transformer** object to perform transformations based on an XSLT stylesheet.
- Use a **Transformer** to pipe information between SAX and DOM parsers and trees, and to write an XML document to a stream from a DOM tree.
- Create an XSLT stylesheet at runtime from static and dynamic information, and apply it to transform existing XML documents.
- Use the **Transformer** interface to control a transformation’s output properties, such as document encoding and document type declaration.

XML Transformations

- A powerful feature of XML is the relative ease with which an XML document can be **transformed** into new content.
 - This is a common technique for presenting **XML in a web browser**: the HTML used by the browser is a transformation from the source XML document.
 - **XML-to-XML transformations** are also possible, and quite useful in middleware and server-side applications.
 - Other output types, including plain text, have their applications as well.
- The standard language for XML transformation is actually known as **XSL Transformations**, or **XSLT**.
 - Actually, the original XSL has been conceptually split up into XSLT and XSL (or **XSLFO**), the latter focusing entirely on formatting XML for presentation using **Formatting Objects**.
 - XSLT – at recommendation version 1.0 at the time of this writing
 - is dedicated to transformation of XML, rather than the formatting of XML for visual or audio presentation.
- **XSLT can be used to define rules by which elements can be extracted from one document and written out into another.**

XSLT

- The idea behind XSLT is that a **stylesheet** (or, synonymously, a **transform**) can be defined that describes a transformation from a **source** to a **result**.
- An **XSLT Processor** is a component that can take two inputs – a source document and a stylesheet – and produce the result document.



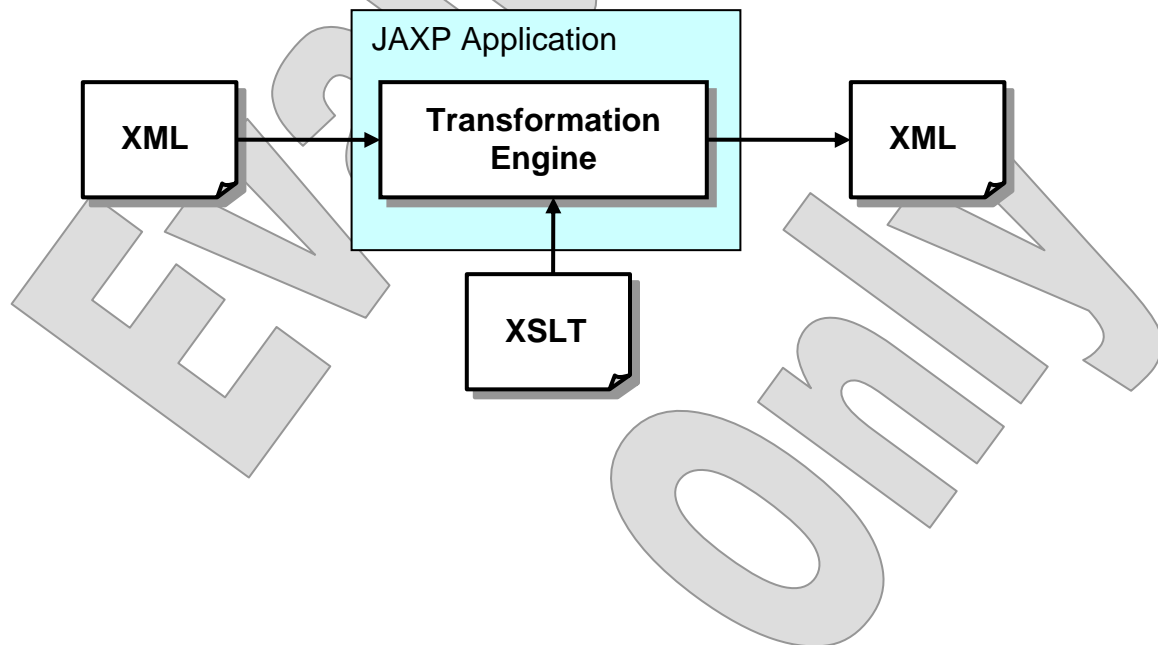
- As a stylesheet language, XSLT offers some attractive features:
 - It is an **XML vocabulary**, so stylesheets are XML elements with a known schema.
 - Thus stylesheet documents can themselves be parsed, transformed, and otherwise processed as XML documents.
 - It is not a full-blown programming language; it is much simpler and allows an author to define fairly complex transformations in a single text document. (XSLT does have some features which seem deceptively like those of a procedural programming language, as we'll see.)
 - The separation of stylesheet and processor allows the great majority of transformation logic to be written just once.

Java and XSLT

- **Although a great deal can be accomplished in XML transformations using only XSLT, the combination of XSLT and Java application code is a natural one.**
 - XSLT is, after all, not a complete programming language, and XSLT stylesheets are limited in the complexity of transforms they can effect.
 - XSLT is a great 80/20 solution to the problem of XML transformations: it solves 80% of the problem with 20% of the effort.
 - Java application code is a natural complement to XSLT, to solve that nasty other 20%.
 - Even if a stylesheet expresses everything that is needed for a desired transformation, it will usually be necessary to tie the transformation into the broader XML application.
- **Thus Java control of transformations defined using XSLT is an excellent application model.**

JAXP

- The **Java API for XML Processing**, or **JAXP**, is the dominant standard for Java/XML applications.
- Java SE 6 includes JAXP 1.4.
- JAXP 1.4 includes packages of code to support SAX2, DOM3, XPath 1.0, XSLT 1.0, and the STaX XML streaming API.
- JAXP provides a standard interface to XSLT processing, based in its **Transformer** class.
 - It also defines an important standard for a transformer factory architecture, which allows transformer implementations to be plugged into the JAXP installation at runtime.



Sun and Apache

- **Sun Microsystems and the Apache Software Foundation have forged an increasingly close partnership in their work on Java standards and tools.**
 - Sun Microsystems, of course, introduced the Java language, and continues to sponsor the rapid maturation and growth of the language and its surrounding technology, including the Java Enterprise platform, from which effort JAXP emerged.
 - Apache has focused on implementing industrial-strength tools through numerous open-source development projects. Apache is responsible for the Apache and Tomcat web servers, the Ant build tool, and several XML processors.
- **A pattern has emerged by which Sun will release a specification, including print-ready documents and often Java code packages, and Apache will provide the reference implementation of the more generic side of the specified contract, such as a web server.**
 - In the case of XML processing, Sun is refining or defining JAXP, a messaging API called JAXM, a service-registry API called JAXR, and a number of other specifications.
 - Apache has built or is refining a number of XML parsers, XSLT processors – we'll work with **Xalan** – SOAP messaging toolkits, and XML-aware web servers and libraries.

Pluggable Transformers

- Probably the largest step forward represented by the JAXP specification was the identification of a standard means of creating and using parsers and XSLT processors themselves.
 - XSLT 1.0 does not specify how a transformation processor is to be found, instantiated, or asked to perform transformations.
 - Applications have had to rely on proprietary interfaces to XSLT processors, making portable coding impossible.
- JAXP defines two abstract classes for XSLT processors – a **transformer** and a **transformer factory**.
- This leads to a completely generic, three-step process by which an application can kick off document transformation:
 - Create a transformer factory using the factory's **newInstance** method.
 - Create a transformer using some overload of the factory's **newTransformer** method.
 - Call the transformer's **transform** method to start transformation, passing it a **source** object and a **result** object.
- We'll look at each of these steps on the next few pages.

The TransformerFactory Class

- Here is an abbreviated listing of **TransformerFactory**:

```
package javax.xml.transform;
public abstract class TransformerFactory
{
    public static TransformerFactory newInstance ();
    public abstract Transformer newTransformer ();
    public abstract Transformer
        newTransformer (Source);
    public abstract Templates newTemplates (Source);
    public abstract Source getAssociatedStylesheet
        (Source, String, String, String);
    public abstract void
        setURIResolver (URIResolver);
    public abstract URIResolver getURIResolver ();
    public abstract void
        setErrorListener (ErrorListener);
    public abstract ErrorListener
        getErrorListener ();
}
```

- The **newInstance** method derives a factory instance.
 - If stylesheets in use will refer to resources that the standard JAXP resolver cannot find, an application-specific **URIResolver** can be provided.
 - Transformation errors can be handled by registering an **ErrorListener**.
- A new transformer can then be created with one of the **newTransformer** methods.

Finding a Transformer Factory

- JAXP is packaged with Apache's Xalan XSLT processor as its default implementation.
- It allows other XSLT processors to be plugged in to the system.
- Any **TransformerFactory** implementation will implement the static **newInstance** method to search for an application's designated implementation as follows:
 - Check for the system property **javax.xml.parsers.TransformerFactory**, and if defined read it as the name of the preferred parser's factory class.
 - This can be set on the command line or in a properties file specific to JAXP.
 - Attempt to load the indicated factory, and return an instance of that class. In this case the default factory class will be idle from this point forward.
 - If the property is undefined, the default implementation will use its own factory class.
 - Once the factory is defined, it is understood that it will only create transformers of the designated type.

The Transformer Class

- Individual transformations can be triggered using the **Transformer** class:

```
public abstract class Transformer
{
    public abstract void transform (Source, Result);
    public abstract void
        setParameter (String, Object);
    public abstract Object getParameter (String);
    public abstract void clearParameters ();
    public abstract void
        setURIResolver (URIResolver);
    public abstract URIResolver getURIResolver ();
    public abstract void
        setOutputProperties (java.util.Properties);
    public abstract java.util.Properties
        getOutputProperties ();
    public abstract void
        setOutputProperty (String, String);
    public abstract String
        getOutputProperty (String);
    public abstract void
        setErrorListener (ErrorListener);
    public abstract ErrorListener
        getErrorListener ();
}
```

- The key method, certainly, is **transform**.
 - **URIResolver** and **ErrorListener** can be set here, perhaps differently for different transformations.
 - The transformation's **output properties** can also be managed through this class' methods. We'll revisit this at the end of this chapter.

Sources and Results

- The **Source** class and its three main subtypes define the behavior of a transformation source. The **Transformer.transform** method accepts any of these, each defined in a subpackage of **javax.xml.transform**:
 - **StreamSource** provides information on an input stream.
 - **SAXSource** interprets the SAX event sequence and passes the information to the transformer.
 - **DOMSource** interprets a DOM tree.
- Similarly, the **Result** class captures the behavior of a transformation product:
 - **StreamResult** provides an output stream to which the literal XML, HTML or text can be written.
 - **SAXResult** creates a SAX event stream based on the produced XML information; this object then mimics a proper SAX parser.
 - **DOMResult** translates the XML output into a new DOM tree, and in this respect it too mimics a parser.
- This symmetrical structure allows applications to connect streams, SAX parsing and DOM trees in any combination.
- For this module, we'll use stream-to-stream transformation to enable a hands-on study of XPath and XSLT.

Parsing Tools and Setup

- For exercises in this module, we will use the following tools:
 - The **JDK 6**, from Sun – which includes **JAXP 1.4** and therefore **Xalan**
 - The **Tomcat web server**, version 6.0, from Apache – only in our final chapter, to support an XSLT-powered web application
- Several environment variables must be set so that Ant build scripts know where to find these resources:
 - **JAVA_HOME** should already be set – confirm this now
 - **CC_MODULE** should be set to **c:\Capstone\JAXPTransform**.
- The executable path must include:
 - **%JAVA_HOME%\bin**
 - **c:\Capstone\Tools\Ant1.6\bin**

Identity Transformation

EXAMPLE

- In **Examples\Identity**, the file **src\cc\jaxp\Identity.java** is the source for a simple application which performs an **identity transformation** on an XML file:

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

            Transformer transformer =
                factory.newTransformer ();

            StreamResult result = args.length > 1
                ? new StreamResult (args[1])
                : new StreamResult (System.out);

            transformer.transform
                (new StreamSource (args[0]), result);
        }
        catch (Exception ex)
        {
            ex.printStackTrace ();
        }
    }
}
```

Identity Transformation

EXAMPLE

- Running the application on the provided **TicTacToe.xml** produces an XML file substantially identical to the source.

TicTacToe.xml:

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

<Game>
  <Move Player="X" Row="3" Column="1" />
  <Move Player="O" Row="1" Column="3" />
  <Move Player="X" Row="1" Column="1" />
  <Move Player="O" Row="2" Column="1" />
  <Move Player="X" Row="3" Column="3" />
  <Move Player="O" Row="2" Column="2" />
  <Move Player="X" Row="3" Column="2" />
</Game>
```

ant

run TicTacToe.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<Game>
  <Move Player="X" Row="3" Column="1" />
  <Move Player="O" Row="1" Column="3" />
  <Move Player="X" Row="1" Column="1" />
  <Move Player="O" Row="2" Column="1" />
  <Move Player="X" Row="3" Column="3" />
  <Move Player="O" Row="2" Column="2" />
  <Move Player="X" Row="3" Column="2" />
</Game>
```

- The only differences are in whitespace.

A Scriptable Transformer

EXAMPLE

- We can expand this simple application to be significantly more useful!
- In and under **Examples\Transformer**, we'll find this same application, renamed to **XMLTransformer**.
 - A bit of code has been added to enable this application to read not only the source XML, but an XSLT stylesheet, from a provided resource.
 - The application also outputs to a parameterized destination, which can be a file or the console output stream.
 - The resulting tool is a full-fledged XSLT processor with a command-line interface.

A Scriptable Transformer

EXAMPLE

- Major code changes are summarized here:

```
...
if (args.length < 3)
{
    System.out.println
        ("Usage:  java cc.tools.xml.XMLTransformer " +
         "<source XML> <stylesheet> [<destination XML>]");
    System.exit (-1);
}
...
try
{
    TransformerFactory factory =
        TransformerFactory.newInstance ();
    Transformer transformer = factory.newTransformer
        (new StreamSource (args[1]));

    StreamResult result = args.length > 2
        ? new StreamResult (args[2])
        : new StreamResult (System.out);

    transformer.transform
        (new StreamSource (args[0]), result);
}
...
```

- **Build the application and test using the provided source document and stylesheet.**
 - Don't worry about the details of the stylesheet if they are unfamiliar: we'll study XPath and XSLT in the remaining chapters, and this file will become clear.
 - Results are shown on the next page.

A Scriptable Transformer

EXAMPLE

Source.xml:

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

<Root>
  <Detritus/>
  <Useful>This is the result I wanted!</Useful>
  <Flotsam>
    <Jetsam size="5" />
  </Flotsam>
</Root>
```

Transform.xsl:

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

<xsl:transform version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
>
  <xsl:output method="text" />

  <xsl:template match="/">
<xsl:value-of select="//Useful" />
  </xsl:template>

</xsl:transform>
```

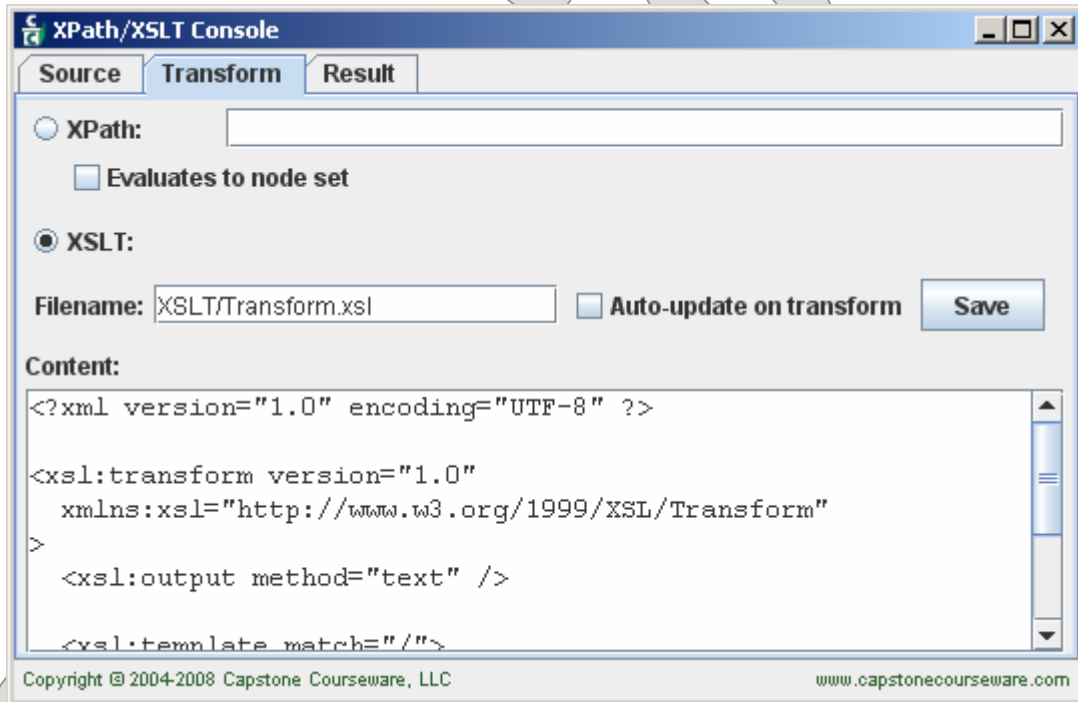
ant

```
run Source.xml Transform.xsl
```

```
This is the result I wanted!
```

A Transformations Console

- In the upcoming lab, you'll build a still better version of the transformer application:
 - It will provide a graphical interface that allows the user to load source and transform documents interactively.



- The user will be able to make minor modifications from the GUI, as well, and save changes if desired.
- A final version will incorporate support for XPath, showing the results of evaluating a user-provided XPath expression against a source document.
- This will provide some better exercise in managing transformations with JAXP.
- The resulting application will also be our tool of choice for studying XPath and XSLT in more detail.

A Transformations Console

LAB 1A

Suggested time: 30 minutes.

In this lab you will complete the implementation of a transformations console. This is a simple JFC application that gives the user a graphical interface for testing XPath expressions and XSLT transformations on source XML documents.

Detailed instructions are found at the end of the chapter.

Evaluation
Only

Template Parameters

- It should come as no surprise that a Java application using JAXP to perform XSLT transformations is not limited to simply loading a stylesheet and applying it.
- There are a few useful techniques by which the application can influence the transformation process:
 - JAXP provides an interface to manipulate the values of **stylesheet parameters**, which are referenced as variables in the stylesheet itself.
 - We could re-invent the XPath evaluator using this feature: the static stylesheet could be prepared with just a parameter called **\$XPathExpression**, and the Java code would simply set this based on command-line arguments before making the requested transformation.
 - Also, XSLT stylesheets can be manipulated at a lower level, as XML documents, using JAXP's DOM interfaces.
 - We won't go that deep into these JAXP features in this module, as we want to focus first on learning XPath and XSLT.

Output Properties

- The **Transformer** class also provides a means for dynamically controlling the transformation's **output properties**.
 - These are aspects of the transformation outside of the scope of any given template; as such they are defined in XSLT using a separate element **xsl:output**.
 - The **output method** maps to the type of result document desired: XSLT 1.0 defines options “xml”, “html”, and “text”.
 - The content of the XML declaration in the result can be controlled at an attribute level: XML version, document encoding, and the **standalone** attribute.
 - The document type definition can be shaped here, as well.
 - We'll study this element in detail in Chapter 3.
- **Output properties can also be managed from within Java code.**
 - **Transformer** has accessors and mutators for the indexed **outputProperty** property (!): for instance **setOutputProperty** takes a name-value pair, while **setOutputProperties** can be used to pass a set of property values as a **java.util.Properties** instance.
 - These calls can either create new property entries or overwrite existing ones based on the key.
 - The **OutputKeys** class utility provides public constants for the correct key values for all recognized properties.

Showing Transform Output

LAB 1B

Suggested time: 15 minutes.

In this lab you will enhance the transformations console you built in Lab 1A by adapting result presentation in the case of HTML transformation output.

Detailed instructions are found at the end of the chapter.

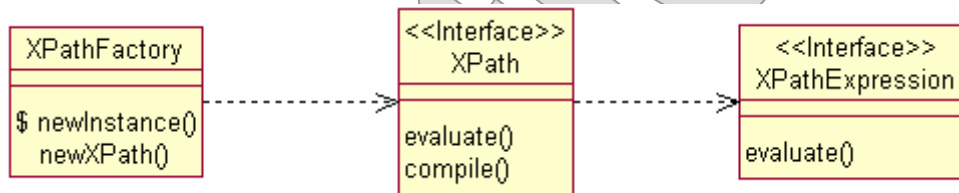
Evaluation Only

XPath

- XSLT defines an XML vocabulary of **templates** that **match** source document content and **produce** output based on that content.
- Underlying XSLT is **XPath**, which is a simple expression-based grammar for selecting information nodes from XML documents.
 - XSLT uses XPath to define match criteria for templates, sets of nodes for looping, to extract and to produce source information in the result stream, and for many other purposes.
 - Other W3C-recommended technology also uses XPath: the DOM and XML Schema, among others.
- We'll study XPath proper in the following chapter.
- But let's look at the XPath support in the JAXP ...

The XPath Class

- JAXP offers elegant support for use of XPath expressions, via the package **javax.xml.xpath**.
- The center of this system is the **XPath** interface, which represents an engine for evaluating XPath expressions against given context objects.



```

public class XPath
{
    String evaluate (String expr, InputSource source);
    Object evaluate (String expr,
        InputSource source, QName expectedType);
    String evaluate (String expr, Object context);
    Object evaluate (String expr,
        Object context, QName expectedType);
    XPathExpression compile (String expr);
    ...
}
  
```

The XPath Class

- **XPath** offers features including:
 - **Precompilation** of expressions for faster repeated use
 - Control over **namespace context**, available **functions**, and resolution of **variables**
- The API is also agnostic about the means of representing the XML information set.
 - DOM is the default representation, and that's what most of us will want to use.
- Most applications can get what they need from a simple, three-step process:
 - Create an **XPathFactory**:

```
XPathFactory factory = XPathFactory.newInstance ();
```

- Create the **XPath** engine itself:

```
XPath xpath = factory.newXPath ();
```

- Evaluate expressions against parsed DOM trees as needed:

```
NodeList list = (NodeList) xpath.evaluate  
("/Game/Move/@Player", myDocument,  
XPathConstants.NODESET);
```

```
String value = xpath.evaluate  
("/Game/Move[3]/@Player", myDocument);
```

Evaluating XPath Expressions

LAB 1C

Suggested time: 30 minutes.

In this lab you will enhance the transformations console you built in Lab 1B, by implementing immediate evaluation of XPath expressions against the source document.

Note: if you do not complete this lab, be certain to build the answer version of the application. This will be used in later demos, examples, and labs.

Detailed instructions are found at the end of the chapter.

Evaluation Only

SUMMARY

- We've seen the most basic uses of the **JAXP Transformer** class.
- This module is intended first to develop the ability to control the execution of XML transformations from Java code, which we can now do.
- The rest of the module will develop facility with the XPath and XSLT languages.
 - We will use our Java **TransformConsole** to test out XPath syntax during hands-on exercises.
 - This application will be our XSLT processor as well, as we study the structure of XSLT stylesheets in later chapters.
- JAXP also offers some programmatic control over transformations, as discussed at the end of this chapter.
 - Deeper study of these features is beyond the scope of this module.

A Transformations Console

LAB 1A

Introduction

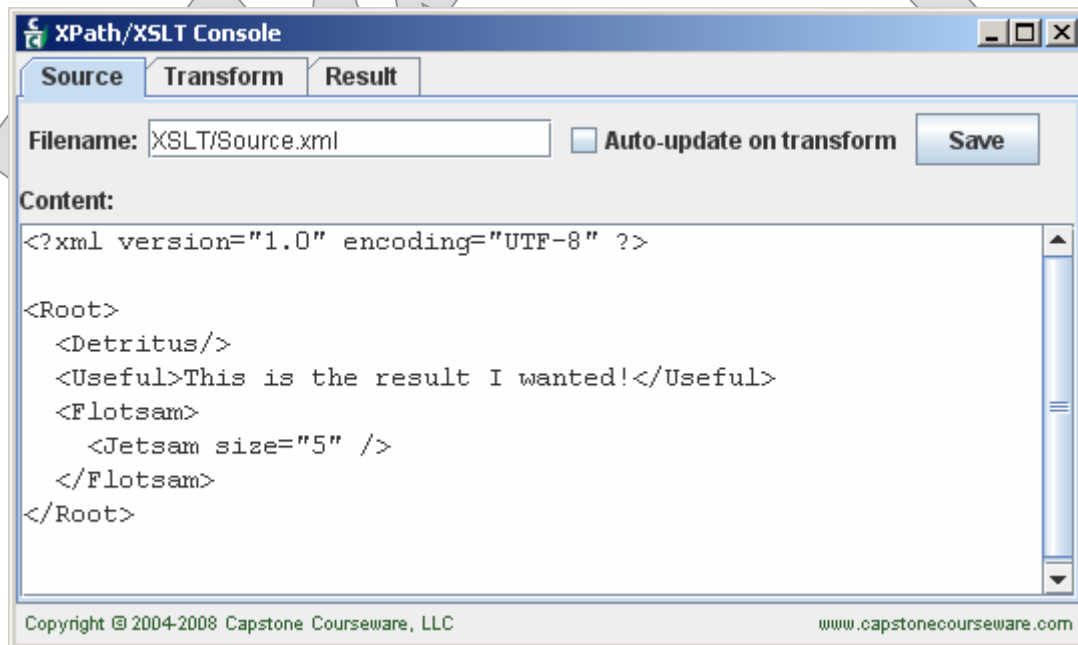
In this lab you will complete the implementation of a transformations console. This is a simple JFC application that gives the user a graphical interface for testing XPath expressions and XSLT transformations on source XML documents.

Directories: **Labs\Lab1A** (do your work here)
 Examples\Console\Step1 (backup copy of starter files)
 Examples\Console\Step2 (answer)

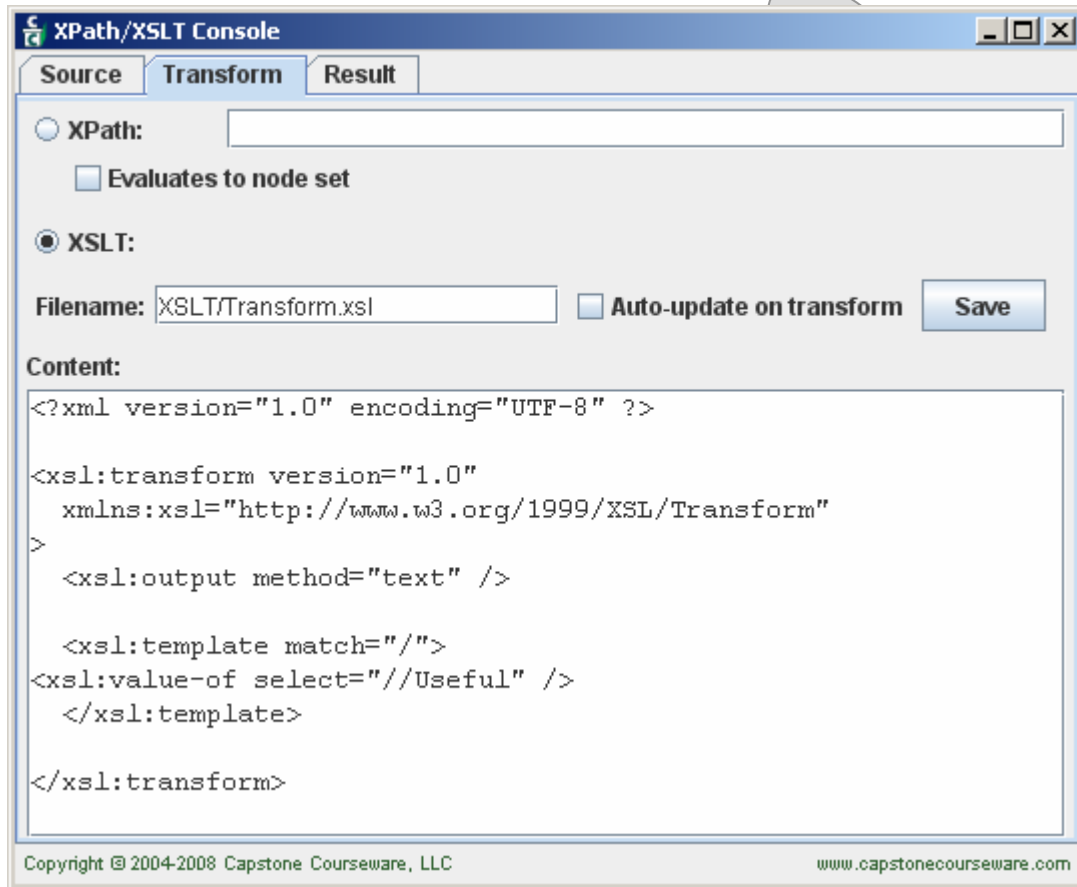
Files: **src\cc\jaxp\TransformConsole.java**
 XSLT\Source.xml
 XSLT\Transform.xsl

Instructions

1. Build the starter application by typing **ant**. Run with **run**, and try out the existing features of the JFC application. Note that you can specify a file for source or transform in the text field labeled, "Filename:" and hit Enter to load the file into the text area below. Try typing in "XSLT/Source.xml" for the source file, and see that you can load it.

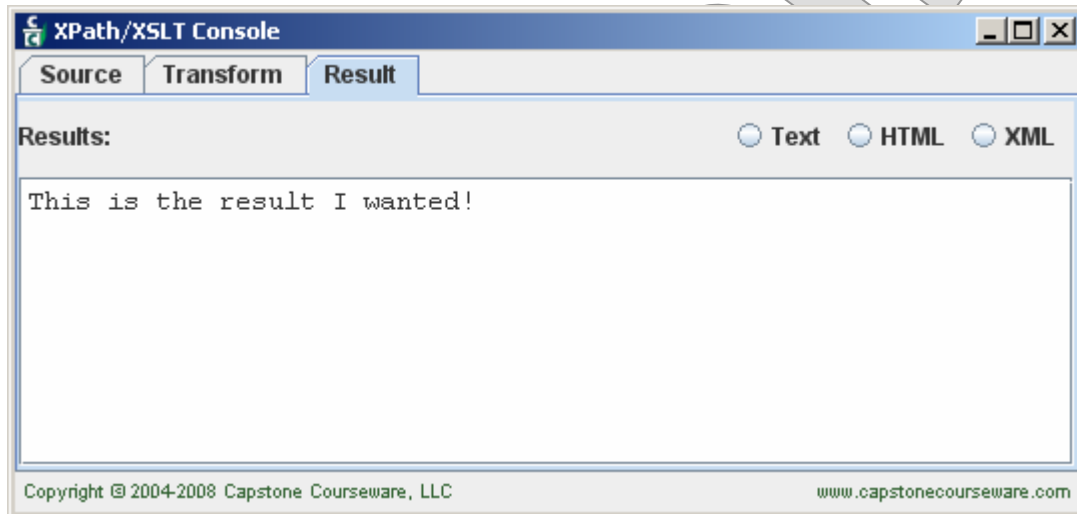


2. Move to the Transform tab (One handy tip about JFC tabbed panes: use the Ctrl-Page-Up and Ctrl-Page-Down keys to easily navigate between cards, as you use the Tab key to move between controls on a card.) Check the XSLT radio button, and try loading “XSLT/Transform.xml” for the Transform file.



3. When you navigate to the results tab, however, nothing happens. This is the part of the implementation you will complete now. Close the application.
4. Open the source file **TransformConsole.java** and find the inner-class event handler **TransformHandler**. The **showXSLTResult** method has been partially implemented to derive **source** and **transform** content as simple strings. Open a **try-catch** system after this code, checking all exceptions.
5. In the **try** block, create a new **StringWriter** called **out**.
6. Create a new **TransformerFactory** called **factory**.
7. Use this to create a new **Transformer** called **transformer** – to the **newTransformer** method pass a new **StreamSource** built on a new **StringReader** which in turn is built on the string **transform**.
8. Call **transform**, passing a **StreamSource** based on a **StringReader** built on **source**, and a new **StreamResult** based directly on **out**.
9. Set the string **result**, which is already declared as a class member, to **out.toString**.

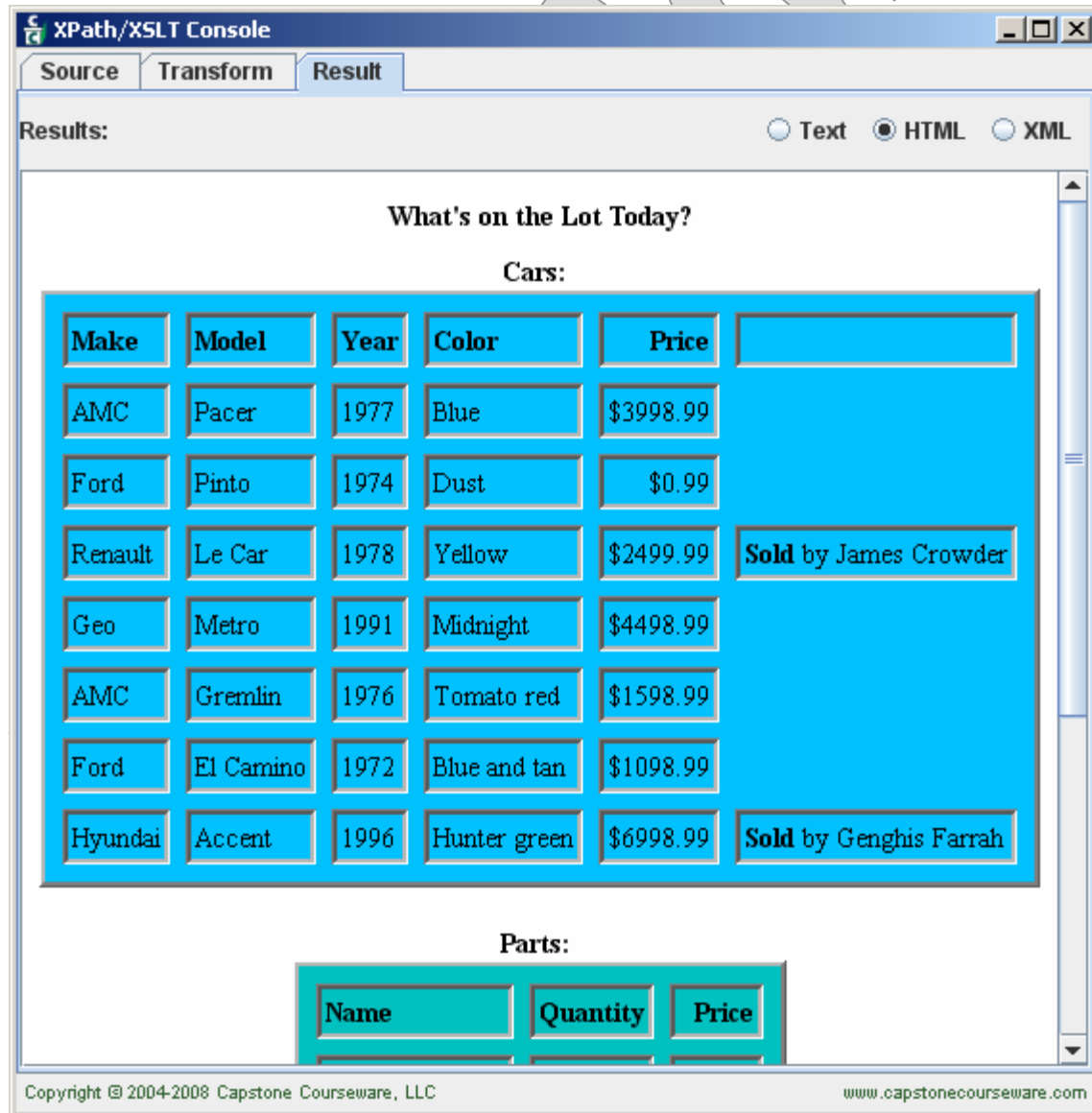
10. In the catch block, set **result** to an error string – you may want to include the exception class, message, or stack trace.
11. After the **try-catch** block, call **txResult.setText**, passing **result**.
12. Build (**ant**) and test (**run**) the application. You should now find that you can load **Source.xml** and **Transform.xsl** and see transform results when you navigate to the Results tab.



13. If you don't get output as above, double-check your code. Try printing out the value of **transformation** before using the string, and see if anything is wrong there. Consult your instructor at this point.

Instructions

1. Run the starter application – which is just the completion of Lab 1A – and test it on another pair of source files: **Cars/OnTheLot.xml** and **Cars/Display.xsl**. The transformation should work fine, but notice that the output is HTML, shown as raw text. JFC text controls are certainly able to render HTML, and the application already has a set of radio buttons that allow the user to force the rendering from plain text to HTML. Try clicking the HTML button, and you should see a graphical rendition of the contents of **OnTheLot.xml**:



2. It would be nice, wouldn't it, if the application showed this automatically, as appropriate? All you need to do is find the output method of the transform in use, and act on that by changing the content type of the text editor. We'll work through this over the next few steps.

3. Notice the existing method **changeContentType**, which changes the content type of the **JEditorPane txResults**, and sets the control's text:

```
private void changeContentType (String method)
{
    txResult.setContentType (method.equals ("html")
        ? "text/html"
        : "text/plain");
    txResult.setText (result);
}
```

4. At the beginning of **TransformHandler.showXSLTResult**, after **source** and **transform** are derived, declare a string **method**, and set it to "text" by default.
5. At the end of the **try** block, set **method** to **transformer.getOutputProperty**, passing **OutputKeys.METHOD**.
6. At the bottom of the method code, remove the call to **txResult.setText** and replace it with a call to **changeContentType**, passing **method**.
7. Before this new code, check the value of **method** and call **setSelected** on the appropriate radio button: **rdText** for "text", **rdHTML** for "html", or **rdXML** for "xml".
8. Save the file and rebuild the application. Test again, and you should find that the results are shown as parsed HTML for the **Display.html** transform. You can try the old **Source.xml** and **Transform.xsl** to confirm that these still render as plain text.

Evaluating XPath Expressions

LAB 1C

Introduction

In this lab you will enhance the transformations console you built in Lab 1B, by implementing immediate evaluation of XPath expressions against the source document.

Directories: **Labs\Lab1C** (do your work here)
 Examples\Console\Step3 (backup copy of starter files)
 Examples\Console\Step4 (answer)

Files: **src\cc\jaxp\TransformConsole.java**
 XPath\TicTacToe.xml

Instructions

1. You've noticed that there's an **XPath** radio button on the **Transform** tab of the console application – but so far it doesn't do anything. Open **TrnasformConsole.java** and find the **stateChanged** method of the inner class **TransformHandler**. See that, when the XPath option is selected, this method calls the helper function **showXPathResult**. This method currently does nothing.
2. Start off an implementation of **showXPathResult** by creating a **try/catch** system that catches all **Exceptions**. In the **catch** block, print a stack trace and set the **result** field to an error message.
3. Now, to start the **try** block, derive an **XPath** engine. We'll collapse two steps into one here, and also make the engine reference **final** since there's no advantage in creating a new one every time the method is invoked:

```
final XPath xpath = XPathFactory.newInstance ().newXPath ();
```

4. Set the **result** field to an empty string, so that we can compile values (or error messages) into it.

- Now a not-particularly-fun part of the process: we need to parse the source document into a DOM tree. You either know DOM processing with JAXP or you don't; either way, the incantation is as follows:

```

DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance ();
DocumentBuilder parser = factory.newDocumentBuilder ();
parser.setErrorHandler (new ErrorHandler ());
Document doc = parser.parse (new ByteArrayInputStream
    (pnSource.getContent ().getBytes ());

```

The **ErrorHandler** is an inner class that's all ready to catch any parsing problems and report them – here's one of the handler methods:

```

public void error (org.xml.sax.SAXParseException ex)
    throws org.xml.sax.SAXException
{
    System.out.println ("Error: " + ex.getMessage ());
    result += "Error: " + ex.getMessage () + "\n";
}

```

- Now, after your DOM-parsing code, set **result** by evaluating the user's XPath expression:

```
result = xpath.evaluate (txXPath.getText (), doc);
```

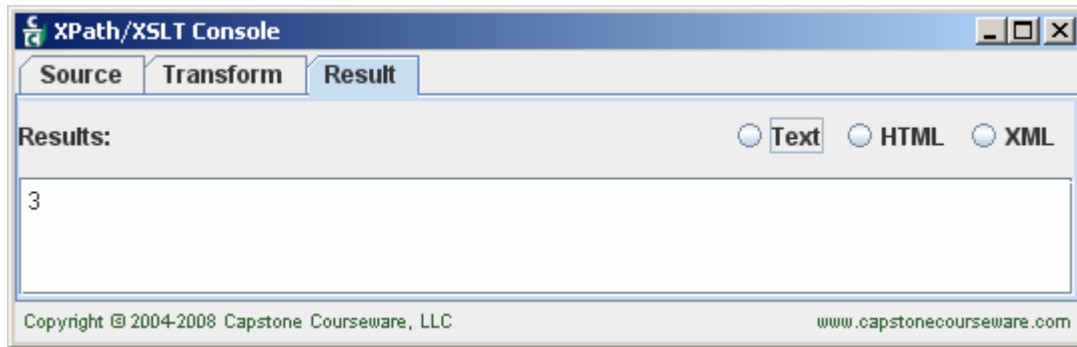
- After the **try/catch** block, put the **result** into the text area for the user to see. (This way, either the successful evaluation or the error message from the **catch** block – or any errors reported by **ErrorHandler** – will show to the user.)

```
changeContentType ("text");
```

- Build with **ant**, and **run** to test your code. For a source document, load the document "XPath/TicTacToe.xml". Then select the **XPath** radio button and enter the expression "count (/Game/Move[@Player='X'])":



- Navigate to the **Results** tab, and you should see the results of evaluating your expression – a count of the moves made by player O:



- Now, go back to the **Transform** tab and try a different expression: `"/Game/Move/@Row"`. Go to the **Results** tab and see the value `"3"`.

Now, we haven't yet had a chance to study XPath in any depth, but this is an expression that produces a set of results. That is, the prior expression was a **count** of nodes that met a certain criterion, and that's a single value that can be converted to a string and shown to the user. But this expression is meant to derive all the Row indices of all the moves in the tic-tac-toe game. If you look at the source document, you can see that this would be a list of numbers: 3, 1, 1, 2, 3, 2, 3.

Your code so far just evaluates the XPath expression to a string. As we'll learn in the next chapter, this means that you're converting what should be a node set into a single value, and the evaluator responds by giving you the first node in the set.

There's a checkbox on the UI with which the user can advise your code that he or she expects to see a list of values, rather than just one. But checking it won't help! because we have yet to do anything differently in `showXPathResult`.

- Instead of just setting `result` to the evaluation of the expression, first call `ckNodeSet.isSelected`, and if it is `false`, go ahead with your current code. Otherwise, open a code block and add the following code to it.
- Call `xpath.evaluate`, as you do in the single-value case, but with a third argument, `XPathConstants.NODESET`. Instead of capturing the return value as `result`, downcast it to `NodeList` and store it in a variable `list` of that type. (Odd bit of naming here: DOM calls this a node list, and XPath calls it a node set. They're both right! The object does preserve ordering, like a list, and guarantees uniqueness (of the XML nodes in it, not their values), like a set.)

13. Create a new **StringBuilder** called **builder**.
14. Now loop over the node list, and for each value, **evaluate** it again, using the expression “.”, which just means the current node. (This is a cheap way to get a quick conversion to a string representation of a DOM node. It would take more DOM code than XPath code, and we’re studying XPath anyway, so ...)

```
for (int i = 0; i < list.getLength (); ++i)
    builder.append (xpath.evaluate (".", list.item (i))).append ('\n');
```

15. Now, set **result** to the contents of the **builder** – just call **toString**.
16. Build and test again. When you use the expression “/Game/Move/@Row” – and, remember to check the **Evaluates to node set** box – you should now see:

