Beyond the Wizards: Understanding Your Code

.NET *and* XML

O'REILLY®

Niel M. Bornstein

CHAPTER 7
Transforming XML with XSLT

You now know how to read XML from a variety of data sources and formats, write XML documents in different formats from arbitrary data structures, create and manipulate XML documents in memory using the DOM, and navigate through any XML tree using XPath. Each of these functions builds on those that came before to open up a new series of possibilities.

The next logical step is to transform the presentation of XML data from one format to another. Extensible Stylesheet Language Transformations (XSLT) is designed to do just that.

Extensible Stylesheet Language (XSL) is a language designed to provide presentation for the content of XML documents. It is composed of three parts: XSLT, XPath (which you're already familiar with from Chapter 6), and XSL Formatting Objects (XSL-FO).

In this chapter, I'll show you XSLT and the .NET assembly that deals with it, `System.Xml.Xsl`. But first, some background.

The Standards

The terms XSL and XSLT, while similar, do not refer to the same W3C specification. XSL, the Extensible Stylesheet Language itself, is simply a language for expressing stylesheets. A stylesheet is a document that controls the presentation of an XML document of a given type. XSL can be thought of as analogous to Cascading Style Sheets (CSS); in fact, XSL shares most of its properties with CSS2, although they have different syntaxes.

XSLT, the XSL Transformation language, is a subset of XSL that was originally designed to perform transformations of XML elements into complex styles, such as nested tables and indexes. XSLT is designed to be usable independent of XSL; however, its use is constrained by its design as a transformation language for the sorts of tasks required by XSL.

Despite the differences, you'll often see the acronyms XSL and XSLT used interchangeably. Unless the speaker is describing complete formatting systems, odds are good that XSL is probably actually a mis-cited reference to XSLT. To add to the confusion, Microsoft has chosen to call the .NET XSLT namespace `System.Xml.Xsl`.

XSL-FO, or XSL Formatting Objects, provides additional, more complex formatting for XML content. However, Microsoft does not implement any specific XSL-FO functionality in .NET, so it is outside the scope of this chapter. If you're interested in learning about XSL-FO, you should look into one of the available books about it, such as *XSL-FO* (O'Reilly) or *Definitive XSL-FO* (Prentice Hall).

Introducing XSLT

A document written in XSLT, referred to as a *stylesheet*, describes the transformation of a particular type of XML document into another format. These other formats can include not only XML languages, such as HTML and Scalable Vector Graphics (SVG), but languages using other syntaxes, such as plain text, Comma Separated Values (CSV), and any number of others—including your choice of proprietary formats. In fact, the range of output formats is limited only by the amount of work you want to do to create the appropriate stylesheet—or to locate an appropriate stylesheet created by a third party.

XSLT can be thought of as a *little language*, providing complete functionality for a limited set of tasks. A little language is defined as a specialized, concise notation, designed for a specific family of problems. Much simpler than a general-purpose programming language, a little program does a limited number of things very efficiently.

Although it was designed simply to provide for the transformation of XML documents, XSLT is often used to process XML documents in other ways. XSLT can be used to generate summary statistics about XML documents, store information from an XML file in a database, or communicate data from an XML file to a mobile device. Again, the applications are limited only by your imagination.

A Brief Introduction to the XSLT Specification

The XSLT specification was designed with several goals in mind. First, the XSLT stylesheet itself is an XML document. This allows you to manipulate the stylesheet like any other XML document, up to and including transforming the stylesheet itself into another format via XSLT.

Next, the XSLT language is based on pattern matching. In fact, much of the pattern matching power of XSLT comes from the XPath specification, which is discussed in Chapter 6.

Third, like any good functional programming language, each XSLT function is free of side effects. The benefit this design goal creates is that the same function will have

the same effect on any source node on which it is invoked, no matter how many times it has already been invoked on that or any other node.

Finally, flow control in XSLT is managed through *iteration* and *recursion*. The concept of iteration will be familiar if you've used C#'s `foreach` statement. The idea is that, given a collection of nodes, the same set of functions will be applied to each one in order. Recursion should also be familiar to developers experienced with modern programming languages; a recursive function is one that calls itself during its execution.

XSLT processing consists of loading an XML source document into a source tree, applying a series of templates to the nodes in the source tree, and sending the resulting data to a result tree. Where the source document comes from, and where the result document is written to, are left up to the XSLT implementation.

As I've already mentioned, an XSLT stylesheet is an XML document. Any XML document can be considered an XSLT stylesheet if it contains the following namespace declaration, traditionally mapped to the `xsl` prefix:

```
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
```

The stylesheet's document element is one of `xsl:stylesheet` or `xsl:transform`, which are synonymous, according to the XSLT specification. The remainder of the stylesheet consists of a series of templates, in the form of `xsl:template` elements. The `xsl:template` element has a `match` attribute, the value of which is an XPath expression to be applied to the source tree. When a node in the source tree matches a template, further matching may be done. When all matching is complete, the matching node and other information from the template is written to the result tree. At the end of processing, the result tree is serialized to a document whose form is specified in the stylesheet's `xsl:output` element.

I'm going to construct a simple XSLT stylesheet, which transforms the *inventory.xml* document from Chapter 5 into an HTML representation of a catalog. Remember that, as with any programming language, there's more than one way to do it. This stylesheet represents just one way to transform the inventory document into HTML.

I'll call this file *catalog.xsl*. Let's examine it one element at a time. To begin with, since the XSLT stylesheet is an everyday XML document, it never hurts to have an XML declaration:

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

The root element of the stylesheet is `xsl:stylesheet`. Either `xsl:stylesheet` or `xsl:transform` *must* be present in an XSLT stylesheet, and the namespace URI and version must be included *exactly* as shown. Different XSLT processors may behave differently, but many will throw a warning or an error if the namespace or version is missing or different:

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

The `xsl:output` element indicates the output method for the transformation. The XSLT specification defines three: `html`, `xml`, and `text`. Specific XSLT processor implementations are free to define others. Some output methods allow method-specific attributes; for example, the `html` and `xml` output methods allow an `indent` attribute, to control whether the output is to be indented:

```
<xsl:output method="html"/>
```

The `xsl:template` element defines a template. The `match` attribute contains an XPath expression indicating which nodes in the document the template is to be executed for. In this case, the expression is `/`, which matches the document root. This template will be the first one executed when transforming an XML document:

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

Anything within the `xsl:template` element that does not have the `xsl` prefix will be copied to output verbatim. In this case, upon reading the beginning of the source tree, this stylesheet will cause the HTML header information to be written to the result tree:

```
<html>
  <head>
    <title>Angus Hardware | Online Catalog</title>
  </head>
```

The `xsl:apply-templates` element indicates that any further templates are to be processed at this point. I'll define a number of other templates later in the stylesheet, and any one of them that match any elements in the source tree would now be executed:

```
<xsl:apply-templates/>
  </html>
```

The stylesheet is an XML document, remember? You have to close every element you open in order for the stylesheet to be valid:

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

This template matches the `inventory` element. Since this is the document element, the template's output is the HTML `body` element, followed by the output of any other matched templates:

```
<xsl:template match="inventory">
  <body bgcolor="#FFFFFF">
    <h1>Angus Hardware</h1>
    <h2>Online Catalog</h2>
    <xsl:apply-templates/>
  </body>
</xsl:template>
```

Upon matching the `date` element, this template will cause the element's attributes to be output, formatted as `month/day/year`. Here you can see again that anything within

the `xsl:template` element that does not have the `xsl` prefix is sent to the output tree verbatim, including character data:

```
<xsl:template match="date">
  <p>Current as of
    <xsl:value-of select="@month" />/<xsl:value-of select="@day" />/<xsl:value-of
select="@year" />
  </p>
</xsl:template>
```

This template outputs a `table` element and the table header, and applies any other templates for nodes that are found within the `items` context:

```
<xsl:template match="items">
  <p>Currently available items:</p>
  <table border="1">
    <tr>
      <th>Product Code</th>
      <th>Description</th>
      <th>Unit Price</th>
      <th>Quantity in Stock</th>
    </tr>
    <xsl:apply-templates />
  </table>
</xsl:template>
```

This template is applied to each item element, sending a table row to the output context:

```
<xsl:template match="item">
  <tr>
    <td><xsl:value-of select="@productCode" /></td>
    <td><xsl:value-of select="@description" /></td>
    <td><xsl:value-of select="@unitCost" /></td>
    <td><xsl:value-of select="@quantity" /></td>
  </tr>
</xsl:template>
```

And finally, the stylesheet's document element must be closed:

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

Example 7-1 shows the complete stylesheet.

Example 7-1. An XSLT stylesheet for inventory.xml

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

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
```

Example 7-1. An XSLT stylesheet for inventory.xml (continued)

```
        <title>Angus Hardware | Online Catalog</title>
      </head>
      <xsl:apply-templates/>
    </html>
  </xsl:template>

  <xsl:template match="inventory">
    <body bgcolor="#FFFFFF">
      <h1>Angus Hardware</h1>
      <h2>Online Catalog</h2>
      <xsl:apply-templates/>
    </body>
  </xsl:template>

  <xsl:template match="date">
    <p>Current as of
      <xsl:value-of select="@month" />/<xsl:value-of select="@day" />/<xsl:value-of
select="@year" />
</p>
  </xsl:template>

  <xsl:template match="items">
    <p>Currently available items:</p>
    <table border="1">
      <tr>
        <th>Product Code</th>
        <th>Description</th>
        <th>Unit Price</th>
        <th>Quantity in Stock</th>
      </tr>
      <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="item">
    <tr>
      <td><xsl:value-of select="@productCode" /></td>
      <td><xsl:value-of select="@description" /></td>
      <td><xsl:value-of select="@unitCost" /></td>
      <td><xsl:value-of select="@quantity" /></td>
    </tr>
</xsl:template>

</xsl:stylesheet>
```

Example 7-2 shows the HTML output resulting from processing *inventory.xml* with *catalog.xsl*, and Figure 7-1 shows a screenshot of the HTML in a web browser.

Example 7-2. HTML output from the catalog.xsl stylesheet

```
<html>
  <head>
    <title>Angus Hardware | Online Catalog</title>
```

Example 7-2. HTML output from the catalog.xsl stylesheet (continued)

```
  </head>
  <body bgcolor="#FFFFFF">
    <h1>Angus Hardware</h1>
    <h2>Online Catalog</h2>
    <p>Current as of 6/22/2002</p>
    <p>Currently available items:</p>
    <table border="1">
      <tr>
        <th>Product Code</th>
        <th>Description</th>
        <th>Unit Price</th>
        <th>Number in Stock</th>
      </tr>
      <tr>
        <td>R-273</td>
        <td>14.4 Volt Cordless Drill</td>
        <td>189.95</td>
        <td>15</td>
      </tr>
      <tr>
        <td>1632S</td>
        <td>12 Piece Drill Bit Set</td>
        <td>14.95</td>
        <td>23</td>
      </tr>
    </table>
  </body>
</html>
```

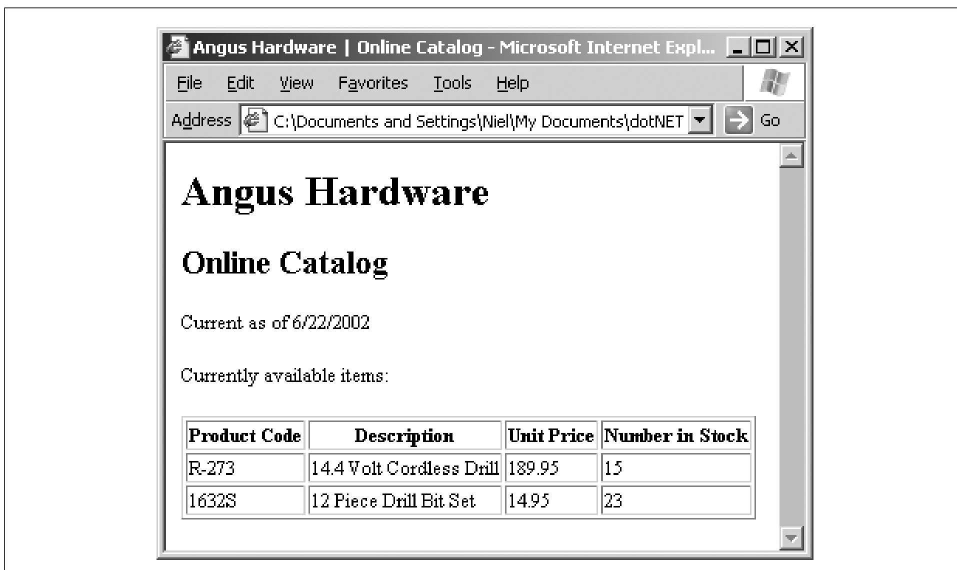


Figure 7-1. Output of the catalog.xsl stylesheet

This sort of transformation is done with a *push model*, in which the source document controls the structure of the result document while the stylesheet controls the appearance of the result document. The other way to use XSLT is a *pull model*, wherein the stylesheet controls both the structure and appearance of the result document, pulling content out of the source document as needed.

I'll show you how to construct a pull model stylesheet to transform the same hardware catalog XML file into a summary text file below. First, the XML declaration and stylesheet element remain the same as with the push model stylesheet:

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

For this stylesheet, however, I want the output to go to a plain text file. The `xsl:output` element takes care of this:

```
  <xsl:output method="text" />
```

Finally, the stylesheet has only one template. Because the output method is text, there's no need to put any HTML tags in the stylesheet. The text will be copied out to the result tree verbatim, except for the `xsl:value-of` element, which uses the `sum()` function to add up the total values of the `quanity` attributes of all the `item` elements:

```
  <xsl:template match="/">
Angus Hardware
Inventory Summary
========= =======

There are <xsl:value-of select="sum(/inventory/items/item/@quantity)" /> units in
stock.
  </xsl:template>
```

Example 7-3 shows the complete stylesheet.

Example 7-3. Inventory summary stylesheet

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

  <xsl:output method="text" />

  <xsl:template match="/">
Angus Hardware
Inventory Summary
========= =======

There are <xsl:value-of select="sum(/inventory/items/item/@quantity)" /> units in stock.
  </xsl:template>
</xsl:stylesheet>
```

Example 7-4 shows the output resulting from this stylesheet.

Example 7-4. Inventory summary stylesheet output

```
Angus Hardware
Inventory Summary
========= =======

There are 38 units in stock.
```

Like XPath, XSLT itself has much more functionality than I can possibly describe here. Entire books have been written about it; if you are interested in learning more about XSLT, take a look at *XSLT* (O'Reilly) or *Learning XSLT* (O'Reilly).

When to Use XSLT

Using XSLT is entirely appropriate when you need to present XML data in a different format. For example, you may be providing a web site that needs to communicate with a variety of devices. Some devices may speak HTML, some may speak WAP, and some may understand some totally unrelated language, such as PDF, EDIFACT, or Minitel. XSLT can transform your XML source documents into the different formats required for diverse clients.

Another appropriate use for XSLT is when you need a common intermediate format for disparate XML data formats. If you can write XML, it can be transformed into any standard or proprietary XML schema for use in another computing environment. For example, you may wish to convert a proprietary XML format into another company's published XML format.

Pull templates make up another category of good use for XSLT. For example, you can use XSLT with a pull template to create summary documents.

In short, you should use XSLT whenever you need to place the content of an XML document into a different structure.

Using XSLT

`System.Xml.Xsl` is an extremely small namespace. Although it consists of just two exceptions, three types, and two interfaces, it still manages to provide full support for version 1.0 of the W3C XSLT specification, as well as providing additional support for embedded scripting in C#, Visual Basic .NET, and JScript.

Transforming an XML Document

Example 7-5 shows one of the simplest possible XSLT-related programs in C#. Given a source filename, a stylesheet filename, and a destination filename, it transforms the source into the destination using the stylesheet. It will work with any XML source file and any XSLT stylesheet.

Example 7-5. One of the simplest possible XSLT programs

```
using System.Xml.Xsl;

public class Transform {
  public static void Main(string [] args) {
    string source = args[0];
    string stylesheet = args[1];
    string destination = args[2];
    XslTransform transform = new XslTransform( );
    transform.Load(stylesheet);
    transform.Transform(source, destination);
  }
}
```

I won't explain in excruciating detail how this program works, but I will point out a few important facts about the `XslTransform` type. It only has one property, `XmlResolver`, and two methods, `Load( )` and `Transform( )`, but it is still one of the most versatile pitchers in the .NET XML bullpen.

First, the `Load( )` method has eight overloads. All of them load the specified XSLT stylesheet, but each of them takes the stylesheet in a different parameter type. The stylesheet may be specified as an `IXPathNavigable` (such as `XmlDocument`), URL, `XmlReader`, or `XPathNavigator`; together, these types cover every possible way to read an XML document. For each `Load( )` method that takes one of these parameters, there is another one taking an `XmlResolver` as the second parameter. This `XmlResolver` is used to resolve any stylesheets referenced in `xsl:import` and `xsl:include` elements.

`xsl:import` and `xsl:include` perform similar, but not identical, functions, Both allow you to place common templates in a separate stylesheet for easy reuse. However, `xsl:import` can only appear at the beginning of a stylesheet, and any imported templates have a lower priority than the template in the importing stylesheet. In contrast, `xsl:include` can appear anywhere in a stylesheet, and any included templates have the same priority as those in the including stylesheet.

If an overload of `Load( )` with no resolver parameter is used, the default `XmlResolver`, `XmlUrlResolver`, is used; if a null instance is passed in, external namespaces are not resolved. The `XmlResolver` passed in is used only for purposes of loading the specified stylesheet, and is not cached for use in processing the XSL transform.

In version 1.1 of the .NET Framework, an additional parameter of type `System.Security.Policy.Evidence` is added to several overloads of the `Load( )` method. The `Evidence` type provides information used to authorize assembly access and code generation for any scripts included in the stylesheet. I'll discuss scripting towards the end of this chapter.

Second, the `Transform( )` method has a total of nine overloads. One takes two `strings`, an input URI and an output URI. The others take various combinations of `IXPathNavigable` or `XPathNavigator` as the first parameter, an `XsltArgumentList` as the

second parameter, and nothing, a `Stream`, an `XmlWriter`, or a `TextWriter` as the third parameter. Two other overloads take only two parameters and return the result tree as an `XmlReader`, which can be navigated as you've already seen in Chapter 2.

Finally, in version 1.0 of the .NET Framework, the `XmlResolver` property contains the `XmlResolver` used when invoking the XSLT `document( )` function. The `document( )` function lets you process multiple source documents in a single stylesheet. This `XmlResolver` may be the same as the one passed in to the `Load( )` method, but does not need to be. In version 1.1 of the .NET Framework, this `XmlResolver` instance is passed as a parameter to the Transform( ) method.

You may never need to do any more than this with XSLT but if you do, you should be aware that there is a lot more you can do with .NET and XSLT.

Associating a Stylesheet with an XML Document

Although it's not part of the actual XSLT specification, there is a way to associate a stylesheet with an XML document. The XML Stylesheet recommendation, Version 1.0, suggests that the `xml-stylesheet` processing instruction be used to link an XML document to its stylesheets:

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

Although you're free to use the `xml-stylesheet` processing instruction in your source XML document, there is no guarantee that any given XSLT processor will do anything special with it; .NET's `XslTransform`, for example, does not.

You can make use of the `xml-stylesheet` processing instruction even though `XslTransform` does not use it automatically. Let's construct a program that examines the source document for an `xml-stylesheet` processing instruction, and transforms it according to the `href` contained within it. I'll just call it *Transform.cs*.

These familiar lines create an instance of `XmlDocument` and load it from the first command-line argument:

```
public class Transform {
  public static void Main(string [] args) {

    string source = args[0];
    string destination = args[1];

    XmlDocument document = new XmlDocument( );
    document.Load(source);
```

The next line will search the loaded `XmlDocument` for the XPath expression `//processing-instruction('xml-stylesheet')`. As you will recall, this expression will match any processing instruction with the target `xml-stylesheet`. If none is found, `SelectSingleNode( )` will return a null instance:

```
XmlProcessingInstruction stylesheetPI =
  (XmlProcessingInstruction)document.SelectSingleNode(
    "//processing-instruction('xml-stylesheet')");
```

After determining that the xml-stylesheet processing instruction is present, these lines declare and initialize a couple of variables that will be used in a moment:

```
if (stylesheetPI != null) {
  char [] splitChars = new char [] { ' ', '=' };
  char [] quoteChars = new char [] { '"', '\'' };
  string stylesheet = null;
```

The XmlProcessingInstruction has a Data property, the contents of which are everything after the target. In this case, the data is href="stylesheet.xsl" type="text/xsl". Here, the string.Split( ) method splits the string on every space and equals sign, returning an array containing four elements: href, "stylesheet.xsl", type, and "text/xsl":

```
String [] stylesheetParts = stylesheetPI.Data.Split(splitChars);
```

The content of the xml-stylesheet processing instruction may look like two attributes, but it's not. It just happens to look that way for this particular processing instruction. In general, a processing instruction's data format is entirely dependent on the expectations of the processor that is consuming it. In .NET, you have to use XmlProcessingInstruction.Data to retrieve the pseudoattributes, and then do the work of parsing them into name/value pairs yourself.

Before proceeding, you need to ensure that the xml-stylesheet processing instruction you've located is one that links the document to an XSLT stylesheet. Since the xml-stylesheet processing instruction can also be used for CSS stylesheets, for example, this line checks the processing instruction's Data property to ensure that it contains the string text/xsl:

```
if (stylesheetPI.Data.IndexOf("text/xsl") != -1) {
```

Once you have verified that the processing instruction does specify an XSLT stylesheet, you then need to find the name of the stylesheet referenced. You know that the array should contain four items, and the item following href should be the name of the stylesheet. These lines of code pluck that item from the array, trimming off any quote characters at the start and end:

```
int indexOfHref = Array.IndexOf(stylesheetParts,"href");
if (indexOfHref != -1) {
  stylesheet = stylesheetParts[indexOfHref+1].Trim(quoteChar);
}
```

If, at the end of all this processing, the stylesheet is still not an empty string, you're ready to use it to transform the source document much as it was done in the first example:

```
if (stylesheet != null) {
  XslTransform transform = new XslTransform( );
  transform.Load(stylesheet);
  transform.Transform(source, destination);
}
```

Example 7-6 shows the complete source code listing for *Transform.cs*.

Example 7-6. Using the xml-stylesheet processing instruction

```
using System.Xml;
using System.Xml.Xsl;

public class Transform {
  public static void Main(string [] args) {

    string source = args[0];
    string destination = args[1];

    XmlDocument document = new XmlDocument( );
    document.Load(source);

    XmlProcessingInstruction stylesheetPI =
     (XmlProcessingInstruction)document.SelectSingleNode(
        "//processing-instruction('xml-stylesheet')");

      if (stylesheetPI != null) {
      char [] splitChars = new char [] { ' ', '=' };
      char [] quoteChars = new char [] { '"', '\'' };
      string stylesheet = null;

      string [] stylesheetParts = stylesheetPI.Data.Split(splitChars);

      if (stylesheetPI.Data.IndexOf("text/xsl") != -1) {
        int indexOfHref = Array.IndexOf(stylesheetParts,"href");

        if (indexOfHref != -1) {
          stylesheet = stylesheetParts[indexOfHref+1].Trim(quoteChar);
        }
      }

      if (stylesheet != null) {
        XslTransform transform = new XslTransform( );
        transform.Load(stylesheet);
        transform.Transform(source, destination);
      }
    }
  }
}
```

The `xml-stylesheet` processing instruction can also specify the medium to which it applies with the `media` pseudoattribute; this allows you to transform the source document differently, depending on who's asking. Given what you've seen in Example 7-5, it should be fairly easy to figure out the additional steps to do that. Remember that more than one node may match the `//processing-instruction('xml-stylesheet')` XPath expression.

I've rewritten the `Main( )` method in Example 7-7, with the changes highlighted. Simply put, it executes the transformation for every `xsl-stylesheet` processing instruction with media pseudoattribute equal to "printer" or no media pseudoattribute. Note that if there is more than one matching processing instruction, it will overwrite the destination file each time.

Example 7-7. Using the xml-stylesheet processing instruction with different media types

```
public static void Main(string [] args) {

  string source = args[0];
  string destination = args[1];

  XmlDocument document = new XmlDocument( );
  document.Load(source);

  XmlNodeList nodeList =
    document.SelectNodes("//processing-instruction('xml-stylesheet')");

  foreach (XmlProcessingInstruction stylesheetPI in nodeList) {
    char [] splitChars = new char [] { ' ', '=' };
    char [] quoteChars = new char [] { '"', '\'' };
    string stylesheet = null;

    string[] stylesheetParts = stylesheetPI.Data.Split(splitChars);

    if (stylesheetPI.Data.IndexOf("text/xsl") != -1) {
      int indexOfHref = Array.IndexOf(stylesheetParts,"href");
      if (indexOfHref != -1) {
        int indexOfMedia = Array.IndexOf(stylesheetParts,"media");
        string media = null;
        if (indexOfMedia != -1) {
          media = stylesheetParts[indexOfMedia+1].Trim(quoteChars);
        }
        if (media == null || media == "printer") {
          stylesheet = stylesheetParts[indexOfHref+1].Trim(quoteChars);
        }
      }

      if (stylesheet != null) {
        XslTransform transform = new XslTransform( );
        transform.Load(stylesheet);
        transform.Transform(source, destination);
      }
    }
  }
}
```

Working with a Stylesheet Programmatically

The real magic of XSLT in .NET comes from the fact that an XSLT stylesheet is just an XML document; therefore, you can create and manipulate the stylesheet just like any other XML document, using the tools you're already familiar with.

Creating a stylesheet

Example 7-8 shows a program that creates the stylesheet in Example 7-1 using `XmlWriter`. It's fairly straightforward, so I'll let the code speak for itself.

Example 7-8. Creating a stylesheet programmatically

```
using System;
using System.IO;
using System.Xml;
using System.Xml.Xsl;

public class CreateStylesheet {
  private const string ns = "http://www.w3.org/1999/XSL/Transform";

  public static void Main(string [] args) {
    XmlTextWriter writer = new XmlTextWriter(Console.Out);
    writer.Formatting = Formatting.Indented;

    writer.WriteStartDocument( );

    writer.WriteStartElement("xsl","stylesheet",ns);
    writer.WriteAttributeString("version","1.0");

    writer.WriteStartElement("xsl:output");
    writer.WriteAttributeString("method","html");
    writer.WriteEndElement( );

    CreateRootTemplate(writer);
    CreateInventoryTemplate(writer);
    CreateDateTemplate(writer);
    CreateItemsTemplate(writer);
    CreateItemTemplate(writer);

    writer.WriteEndElement( ); // xsl:stylesheet
    writer.WriteEndDocument( );
  }

  private static void CreateRootTemplate(XmlWriter writer) {

    writer.WriteStartElement("xsl:template");
    writer.WriteAttributeString("match","/");

    writer.WriteStartElement("html");

    writer.WriteStartElement("head");
```

Example 7-8. Creating a stylesheet programmatically (continued)

```
    writer.WriteStartElement("title");
    writer.WriteString("Angus Hardware | Online Catalog");
    writer.WriteEndElement( ); // title

    writer.WriteEndElement( ); // head

    writer.WriteStartElement("xsl:apply-templates");

    writer.WriteEndElement( ); // xsl:apply-templates
    writer.WriteEndElement( ); // html
    writer.WriteEndElement( ); // xsl:template
  }

  private static void CreateInventoryTemplate(XmlWriter writer) {
    writer.WriteStartElement("xsl:template");
    writer.WriteAttributeString("match","inventory");

    writer.WriteStartElement("body");
    writer.WriteAttributeString("bgcolor","#FFFFFF");

    writer.WriteStartElement("h1");
    writer.WriteString("Angus Hardware");
    writer.WriteEndElement( ); // h1

    writer.WriteStartElement("h2");
    writer.WriteString("Online Catalog");
    writer.WriteEndElement( ); // h2

    writer.WriteStartElement("xsl:apply-templates");
    writer.WriteEndElement( );

    writer.WriteEndElement( ); // body
    writer.WriteEndElement( ); // xsl:template
  }

  private static void CreateDateTemplate(XmlWriter writer) {
    writer.WriteStartElement("xsl:template");
    writer.WriteAttributeString("match","date");

    writer.WriteStartElement("p");

    writer.WriteString("Current as of ");

    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select", "@month");
    writer.WriteEndElement( ); // xsl:value-of

    writer.WriteString("/");

    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select", "@day");
    writer.WriteEndElement( ); // xsl:value-of
```

Example 7-8. Creating a stylesheet programmatically (continued)

```
    writer.WriteString("/");

    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select", "@year");
    writer.WriteEndElement( ); // xsl:value-of

    writer.WriteEndElement( ); // p
    writer.WriteEndElement( ); // xsl-template
  }

  private static void CreateItemsTemplate(XmlWriter writer) {
    writer.WriteStartElement("xsl:template");
    writer.WriteAttributeString("match","items");

    writer.WriteStartElement("p");
    writer.WriteString("Currently available items:");
    writer.WriteEndElement( ); // p

    writer.WriteStartElement("table");
    writer.WriteAttributeString("border","1");

    writer.WriteStartElement("tr");

    writer.WriteStartElement("th");
    writer.WriteString("Product Code");
    writer.WriteEndElement( ); // th

    writer.WriteStartElement("th");
    writer.WriteString("Description");
    writer.WriteEndElement( ); // th

    writer.WriteStartElement("th");
    writer.WriteString("Unit Price");
    writer.WriteEndElement( ); // th

    writer.WriteStartElement("th");
    writer.WriteString("Number in Stock");
    writer.WriteEndElement( ); // th

    writer.WriteStartElement("xsl:apply-templates");
    writer.WriteEndElement( ); // xsl:apply-templates

    writer.WriteEndElement( ); // tr
    writer.WriteEndElement( ); // table
    writer.WriteEndElement( ); // xsl:template
  }

  private static void CreateItemTemplate(XmlWriter writer) {
    writer.WriteStartElement("xsl:template");
    writer.WriteAttributeString("match","item");
```

Example 7-8. Creating a stylesheet programmatically (continued)

```
    writer.WriteStartElement("tr");

    writer.WriteStartElement("td");
    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select","@productCode");
    writer.WriteEndElement( ); // xsl:value-of
    writer.WriteEndElement( ); // td

    writer.WriteStartElement("td");
    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select","@description");
    writer.WriteEndElement( ); // xsl:value-of
    writer.WriteEndElement( ); // td

    writer.WriteStartElement("td");
    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select","@unitCost");
    writer.WriteEndElement( ); // xsl:value-of
    writer.WriteEndElement( ); // td

    writer.WriteStartElement("td");
    writer.WriteStartElement("xsl:value-of");
    writer.WriteAttributeString("select","@quantity");
    writer.WriteEndElement( ); // xsl:value-of
    writer.WriteEndElement( ); // td

    writer.WriteEndElement( ); // xsl:template

    writer.WriteEndElement( ); // xsl:
  }
}
```

This example is just one of many ways to create a stylesheet programmatically. In this case, I've used `XmlTextWriter`, which I introduced in Chapter 4, to create the document and write it to a file. I also could have created an `XmlTextWriter` from any other sort of `Stream` or `TextWriter`; for example, I could have written the XML to a `MemoryStream`, and then turned around and used that `MemoryStream` in the constructor of an `XmlTextReader`, which can be passed to `XslTransform`'s `Load( )` method.

Assuming you had something like the code in Example 7-8 in a private method called `WriteStylesheet( )`, the following snippet would write the stylesheet to a memory buffer, then cause it to be loaded and used to transform an XML document from an input tree to an output tree:

```
MemoryStream stream = new MemoryStream( );
XmlTextWriter writer = new XmlTextWriter(stream, Encoding.UTF8);
this.WriteStylesheet(writer);

stream.Seek(SeekOrigin.begin, 0);

XslTransform transform = new XslTransform( );
```

```
    transform.Load(new XmlTextReader(stream));

    transform.Transform(input, output);
```

Manipulating an existing stylesheet

Just like any XML document, an XSLT stylesheet can be loaded into an `XmlDocument` and manipulated via the DOM, or navigated with XPath. For example, if you have a stylesheet on disk but wish to change the order of some nodes, you could load it into an `XmlDocument`, navigate to the desired node with `SelectSingleNode()`, and then detach that node from its current location and place it into its new location.

Scripting with XslTransform

Creating and manipulating stylesheets is all well and good if all you need to do is use XSLT's built-in abilities. But .NET's XSLT tools can do a lot more. They allow you to execute arbitrary C#, Visual Basic, or JScript code embedded in the stylesheet, pass arguments to the XSLT processor, and call back to methods in your own C# code from expressions in the XSLT code.

Embedded scripts

.NET has the ability to interpret script code embedded within the XSLT stylesheet. Example 7-9 shows the same `catalog.xsl` stylesheet shown previously, but with a few minor changes. I'll explain the highlighted changes.

Example 7-9. Catalog stylesheet with embedded scripting

```
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:msxsl="urn:schemas-microsoft-com:xslt"
  xmlns:angus="http://angushardware.com/"
  version="1.0">

  <msxsl:script implements-prefix="angus" language="C#">
    <![CDATA[
      decimal CalculateInventoryValue(int number, decimal unitCost) {
          return number * unitCost;
      }
    ]]>
  </msxsl:script>

  <xsl:output method="html"/>

  <xsl:template match="/">
    <html>
      <head>
        <title>Angus Hardware | Online Catalog</title>
      </head>
```

Example 7-9. Catalog stylesheet with embedded scripting (continued)

```
      <xsl:apply-templates/>
    </html>
  </xsl:template>

  <xsl:template match="inventory">
    <body bgcolor="#FFFFFF">
      <h1>Angus Hardware</h1>
      <h2>Online Catalog</h2>
      <xsl:apply-templates/>
    </body>
  </xsl:template>

  <xsl:template match="date">
    <p>Current as of <xsl:value-of select="@month" />/<xsl:value-of select="@day" />/<xsl:
value-of select="@year" />
    </p>
  </xsl:template>

  <xsl:template match="items">
    <p>Currently available items:</p>
    <table border="1">
      <tr>
        <th>Product Code</th>
        <th>Description</th>
        <th>Unit Price</th>
        <th>Number in Stock</th>
        <th>Value of Inventory</th>
      </tr>
      <xsl:apply-templates />
    </table>
  </xsl:template>

  <xsl:template match="item">
    <tr>
      <td><xsl:value-of select="@productCode" /></td>
      <td><xsl:value-of select="@description" /></td>
      <td><xsl:value-of select="@unitCost" /></td>
      <td><xsl:value-of select="@quantity" /></td>
      <td><xsl:value-of select="angus:CalculateInventoryValue(@quantity,@unitCost)"/></td>
    </tr>
  </xsl:template>

</xsl:stylesheet>
```

Each new section of code is explained below:

```
xmlns:msxsl="urn:schemas-microsoft-com:xslt"
```

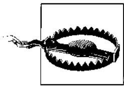
To use the scripting capabilities of `XslTransform`, the stylesheet must include the namespace `urn:schemas-microsoft-com:xslt`. By convention, this namespace is mapped to the `msxsl` prefix:

```
xmlns:angus="http://angushardware.com/"
```

Each piece of script code in the stylesheet must belong to a namespace as well. This can be thought of as equivalent to a .NET assembly namespace. In this case, I've defined the namespace `http://angushardware.com/` with the prefix `angus`:

```
<msxsl:script implements-prefix="angus" language="C#">
  <![CDATA[
    decimal CalculateInventoryValue(int number, decimal unitCost) {
        return number * unitCost;
    }
  ]]>
</msxsl:script>
```

Now the `CalculateInventoryValue( )` method is defined, to return the total value of the stock on hand for each product type. The `msxsl:script` element indicates that this is a block of code. The `implements-prefix` attribute indicates that the code is associated with the `angus` prefix, and the `language` attribute indicates that this block of code is written in C#.



Although the permissible values of the `msxsl:script` element's `language` attribute are `C#`, `CSharp`, `VisualBasic`, `VB`, `JScript`, and `JavaScript`, all code with the same `implements-prefix` attribute value must use the same language. The default language is `JScript`. The `language` attribute is case insensitive, so `VisualBasic` and `visualbasic` are equivalent; and `C#` is an alias for `CSharp`, just as `VisualBasic` is for `VB` and `JavaScript` is for `JScript`.

The code within this `msxsl:script` element is enclosed within a `CDATA` section; while not strictly necessary, this is strongly recommended to avoid XML parsing problems.

The `CalculateInventoryValue( )` method itself should require no further explanation:

```
<td><xsl:value-of select="angus:CalculateInventoryValue(@quantity,@unitCost)"/></td>
```

In this portion of the stylesheet, the `CalculateInventoryValue( )` method defined previously is actually invoked. The `xsl:value-of` element converts the expression in the `select` attribute into a text node; in this case, the expression is the call to `CalculateInventoryValue( )`, with the `angus` prefix. The parameters passed into the method are themselves XPath expressions, and they are coerced into the appropriate types (`int` and `decimal`, respectively) by XSLT.

The result of transforming the *inventory.xml* document with this stylesheet appears in Example 7-10, with changes highlighted.

Example 7-10. New HTML catalog output

```
<html xmlns:msxsl="urn:schemas-microsoft-com:xslt" xmlns:angus="http://angushardware.com/
">
  <head>
    <META http-equiv="Content-Type" content="text/html; charset=utf-8">
    <title>Angus Hardware | Online Catalog</title>
  </head>
```

Example 7-10. New HTML catalog output (continued)

```
  <body bgcolor="#FFFFFF">
    <h1>Angus Hardware</h1>
    <h2>Online Catalog</h2>
    <p>Current as of 6/22/2002</p>
    <p>Currently available items:</p>
    <table border="1">
      <tr>
        <th>Product Code</th>
        <th>Description</th>
        <th>Unit Price</th>
        <th>Number in Stock</th>
        <th>Value of Inventory</th>
      </tr>
      <tr>
        <td>R-273</td>
        <td>14.4 Volt Cordless Drill</td>
        <td>189.95</td>
        <td>15</td>
        <td>2849.25</td>
      </tr>
      <tr>
        <td>1632S</td>
        <td>12 Piece Drill Bit Set</td>
        <td>14.95</td>
        <td>23</td>
        <td>343.85</td>
      </tr>
    </table>
  </body>
</html>
```

The obvious difference is the new fifth column on the HTML table in the output. In addition, the `msxsl` and `angus` namespaces are included in the `html` element, even though they are not used in the output document.

This technique of embedding code in the stylesheet can be quite convenient, because changes to the embedded code do not require you to recompile any C# code. However, embedded code is not portable to XSLT processors that do not support the languages in question or the `urn:schemas-microsoft-com:xslt` namespace. For that reason, passing additional arguments to the XSLT processor becomes an interesting solution.

Adding parameters with XsltArgumentList

`XsltArgumentList` is a type which may be passed as a parameter to most of the `XslTransform.Transform( )` overloads. It allows additional parameters and extensions to be passed to the XSLT processor.

Compared to embedded scripting, `XsltArgumentList` provides for better encapsulation and code reuse, enables you to keep stylesheets smaller and thus more easily

maintainable, supports use of classes in other namespaces besides those supported by XslTransform, and allows node fragments to be passed to the stylesheet using XPathNavigator.

There are two ways of using XsltArgumentList. The first allows you to pass a parameter into the stylesheet, to be replaced in the output tree whenever the xsl:value-of element with that name is evaluated.

The following code would set a parameter named greeting to a value of Hello!:

```
XslTransform transform = new XslTransform( );
transform.Load(stylesheet);

XsltArgumentList argList = new XsltArgumentList( );
argList.AddParam("greeting","","Hello!");

transform.Transform(new XPathDocument(source), argList, new
  StreamWriter(destination));
```

In the stylesheet, you would first need to declare the parameter with the following element:

```
<xsl:param name="greeting" />
```

The xsl:param element can appear either at the top-level element or within an xsl:template element. The parameter thus declared will then be scoped at the level of xsl:param's parent. The name attribute is required, and is used to map the name passed in to its value.

xsl:param has an optional attribute, select, which can be used to define the parameter's value, although this defeats the purpose of XsltArgumentList.

To refer to the parameter within that scope, you can think of it as another XPath expression. The stylesheet uses the name of the parameter prefixed with a dollar sign ($) in the select attribute of an xsl:value-of element:

```
<xsl:value-of select="$greeting" />
```

Adding extensions with XsltArgumentList

In addition to direct parameter replacement, XsltArgumentList enables you to associate your own C# code with an xsl:value-of element. For example, you may wish to create a type that has a property to output today's date:

```
public class Utilities {
  public string Today {
    get {
      return DateTime.Now.ToString( );
    }
  }
}
```

The `Utilities` type simply has one property, `Today`, which returns a `string` representation of the current date and time. You could add other properties and methods, including methods that take parameters.

XSLT extensions only work with the *methods* of extension types, not their *properties*. However, if you know a simple trick, you can access properties just like methods. The trick to remember is that the C# compiler creates a method named `get_XXX( )` for each property named `XXX`. So you can access a property named `Today` by using `util:get_Today( )` in the select attribute of `xsl:value-of`.

Now, it's simply a matter of instantiating a `Utilities` object, and adding it to the `XsltArgumentList`. The changed lines of the source code are highlighted:

```
XslTransform transform = new XslTransform( );
transform.Load(stylesheet);

XsltArgumentList argList = new XsltArgumentList( );
argList.AddParam("greeting","","Hello!");
argList.AddExtensionObject("urn:Utilities", new Utilities( ));

transform.Transform(new XPathDocument(source), argList, new
  StreamWriter(destination));
```

The `XsltArgumentList.AddExtensionObject( )` method allows you to associate a type with a qualified name in the stylesheet. In this example, `urn:Utilities` is the qualified name.

In order to use `Utilities.Today( )`, you must make some small changes in *catalog.xsl*:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:util="urn:Utilities"
version="1.0">
...
  <xsl:template match="inventory">
    <body bgcolor="#FFFFFF">
      <h1>Angus Hardware</h1>
      <h2>Online Catalog</h2>
      <p><xsl:value-of select="$greeting" /></p>
      <p><xsl:value-of select="util:get_Today( )" /></p>
      <xsl:apply-templates/>
    </body>
</xsl:template>
```

In these lines, the `util` prefix is associated with the `urn:Utilities` namespace, and `util:get_Today( )` is invoked in the select expression of an `xsl:value-of` element.

The relevant section of the HTML output is shown below:

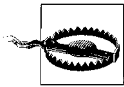
```
<h1>Angus Hardware</h1>
<h2>Online Catalog</h2>
<p>Hello!</p>
<p>7/27/2002 7:14:57 PM</p>
<p>Current as of 6/22/2002</p>
```

An extension need not be a custom type. For example, to insert the XSLT process name and process ID into the output tree, you might add a `System.Diagnostics.Process` instance to the `XsltArgumentList`:

```
argList.AddExtensionObject("urn:Process",Process.GetCurrentProcess( ));
```

And then you could call the `urn:Process` extension from your XSLT stylesheet:

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:util="urn:Utilities"
  xmlns:proc="urn:Process"
  version="1.0">
...
<p>Generated by <xsl:value-of select="proc:get_ProcessName( )" />,
  process ID <xsl:value-of select="proc:get_Id( )" /></p>
```



Be careful when making a non-custom extension available to XSLT. Instead of `proc:get_ProcessName( )`, you could easily have called `proc:Kill( )`, which would terminate the XSLT processor—and any other threads in the same process, such as a web server—in mid-stream. When you make a type available as an extension, you make *all* its public methods available.

Moving On

In addition to all the simple XML reading, writing, manipulation, and navigation you started with, you have now seen how to create and use one particular type of XML document, the XSLT stylesheet. You should have learned a little bit about the XSLT language, and how to pass parameters and extension objects to it in order to customize your output.

Continuing along this theme, I'll show you another very specific type of XML document next, the W3C XML Schema definition document. Though you've already seen some use of XML Schema earlier in Chapter 2, in Chapter 8 I'll present a more in-depth look at W3C XML Schema and what it can do for you.