CHAPTER 9
ACCESSING DATA
USING XML

The ***eXtensible Markup Language* (*XML*) is a language used to represent data in a form that does not rely on any particular proprietary technology**. The technology-neutral language is defined by standards set up by the World Wide Web Consortium (w3c.org) and has rapidly become the standard for data exchange using the Internet. Not only is XML technology neutral, but also it is based on simple character formatting (is human-readable like HTML) and thus is compatible with the network protocols (TCP/IP) that manage data transmission on the Internet.

In addition to XML, a number of associated technologies have evolved that makes XML even more useful. One of these is the XML Schema[1] Language. This language provides a way of defining how an XML document should be formed. Since XML can be easily customized for a specific application, an ***XML Schema* provides a way of defining and validating XML documents to be sure that they follow the rules of the schema**. Business partners can agree upon a specific XML Schemas for XML documents that they exchange. In this way they are sure that the documents they exchange include all the data they expect that is in a format they can understand.

Another useful technology associated with XML is the ***eXtensible Stylesheet Language* (*XSL*) and *eXtensible Stylesheet Language Transforms* (*XSLT*). XSLT provides a means of transforming one XML document into another XML document**. This capability is useful because it provides a way to take a single XML document and transform it into a variety of different versions (include HTML) that are specialized for specific uses.

In this chapter we first take a closer look at XML and its related technologies. We then see how Visual Basic .NET works with XML, XML Schemas, and XSLT. A number of classes within Visual Basic .NET provide methods that work directly with XML. In fact, Microsoft's .NET initiative uses XML as its foundation data technology. Specifically we will see how Visual Basic .NET can work directly with XML, how it can take data from a relational database and convert it into XML, and how an XML document can be transformed into other XML documents.

Objectives

After studying this chapter you should be able to

- Understand what XML is and how it is used within business-to-business transactions.
- Understand XML Schemas and XSL Transforms and how they are used within the context of XML.
- Read and process data that is stored in an XML document.
- Read data from a relational database and transform it into its equivalent XML.
- Transform one XML document into a new XML document using an XSLT document.

9.1 AN XML PRIMER

[1] Generically, a schema, pronounced "skeema", is the definition of an entire database. It defines the structure and the type of contents that each data element within the database. (Source: TechWeb.com). An XML Schema defines of the content used in an XML document.

What is XML

As the Internet became more popular and businesses starting using it to support a variety of commercial applications, both Business-to-Consumer (B2C) and Business-to-Business (B2B), it became clear that the Hypertext Markup Language (HTML) had a number of shortcomings. ***HTML* is a language that focuses on the presentation of information for human consumption**. That is, it is designed to transform data into a form that makes it easy for humans to understand. Consider the web page shown in Figure 9.1.

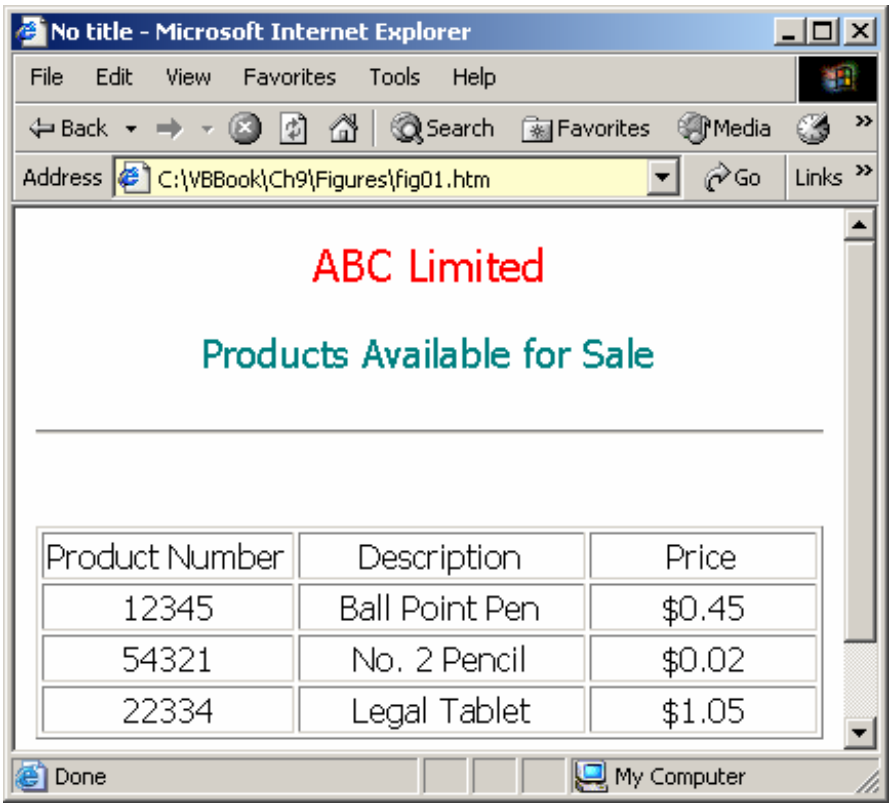


Figure 9.1 Typical web page formatted using HTML

It should be fairly clear what the meaning (semantics) of the information displayed on the web site is. We see information on three products that includes the Product Number, Description and Price. If you were asked to determine the price of the product with a Product Number equal to 54321, you would have no difficulty coming up with a price of $0.02. In fact, this task is so easy that you could do it without any thought. This is because you are intelligent and are able to attach a meaning to the visual information you process.[2]

What happens if another machine were to process the same data. First of all, another machine would likely not process the image as shown in the browser. Instead, the machine would most likely see the data in its original form, that is, the HTML that was used by the browser to create the display in Figure 9.1. What does this HTML look like? Figure 9.2 shows the HTML that was rendered by the browser in the previous figure.

[2] Note that some people with certain learning disabilities might have difficulty with the question posed because their ability to process a visual display is not developed as expected.

```
<html>
<head>
<title>No title</title>
</head>
<body bgcolor="white" text="black" link="blue" vlink="purple" alink="red">

...
<table border="1">
   <tr>
      <td width="299">
         <p align="center"><font face="Tahoma">Product Number</font></p>
      </td>
      <td width="299">
         <p align="center"><font face="Tahoma">Description</font></p>
      </td>
      <td width="299">
         <p align="center"><font face="Tahoma">Price</font></p>
      </td>
   </tr>
   <tr>
      <td width="299">
         <p align="center"><font face="Tahoma">12345</font></p>
      </td>
      <td width="299">
         <p align="center"><font face="Tahoma">Ball Point Pen</font></p>
      </td>
      <td width="299">
         <p align="center"><font face="Tahoma">$0.45</font></p>
      </td>
   </tr>

...
</table>
</body>
</html>
```

Figure 9.2 The HTML used to create the browser rendering in Figure 9.1

You may not understand HTML, but the symbol <tr>, known as a tag, means the start of a table row while the tag </tr> means the end of a table row. Within a table row, each new column is defined within the pair of tags <td> </td>. Figure 9.3 shows just one table row definition.

```
   <tr>
      <td width="299">
         <p align="center"><font face="Tahoma">12345</font></p>
      </td>
      <td width="299">
```

```
        <p align="center"><font face="Tahoma">Ball Point Pen</font></p>
      </td>
      <td width="299">
        <p align="center"><font face="Tahoma">$0.45</font></p>
      </td>
    </tr>
```

Figure 9.3 The HTML definition for one row in a table

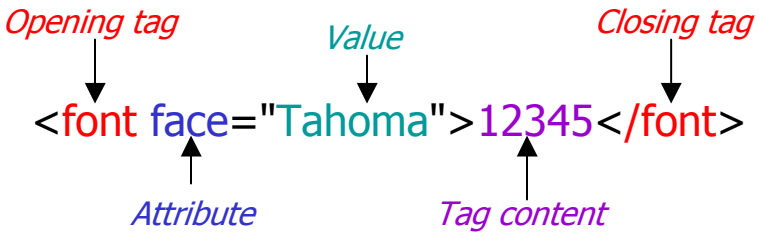
We will focus on the third line of Figure 9.3 that defines one table column using the following HTML:

```
<p align="center"><font face="Tahoma">12345</font></p>
```

What does this mean? The tag defines a new paragraph and defines the end of the paragraph. Within the paragraph tag we see an attribute named align that is equal to the string "center". This means that the contents of the paragraph will be centered. The tag defines the font to be used as indicated with the face attribute (Tahoma in this case). Finally, within the paragraph tag, between the and tags, we see 12345. This is defined as the content, which is in a centered paragraph using Tahoma font. It should be clear that HTML deals with the display of information. Figure 9.4 summarizes the terminology we just used. We should also add that the terms "tag", "node", and "element" might be used interchangeably.

Figure 9.4 Terminology used to describe HTML

We now ask, what does tag content 12345 mean? From the HTML, we know how it should look but we have no clue what it means. Again, referring to Figure 9.1, we know



that its meaning is a product number because we see it under the column heading "Product Number". But strictly from the HTML, it would be hard to draw that conclusion.

This demonstrates the major shortcoming of HTML: it does an excellent job describing how to display content but it does a very poor job communicating the meaning of its content. How does this shortcoming impact our applications? First of all, applications like search engines end up giving you some useless information because they generally search content without any application of semantics. Assume you want to search for the table of elements used in chemistry and you enter the search criteria "element table". Figure 9.5 shows the results of such a search.

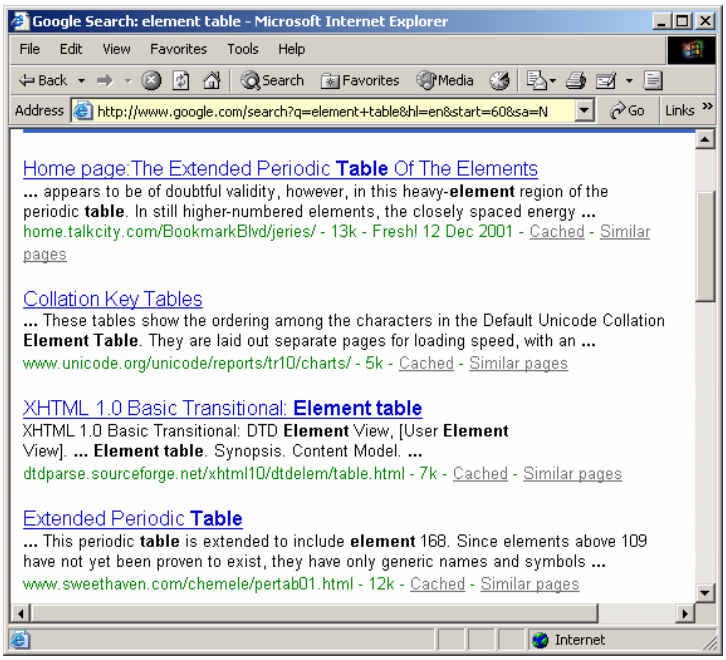


Figure 9.5 The results of a search engine searching for information on "element table"

You can see some "hits" and some "misses" because the semantics of "element table" have a number of different interpretations. In an application like this we again rely on the human to decide which "hits" are valuable and which ones are junk. Most humans are able to make this decision but having a computer make the decision is difficult at best.

What if the HTML shown in Figure 9.2 were replaced with the data shown in Figure 9.6? Note that the tags are now using terminology that is directly relevant to the data that is being stored. If the question "What does the content 'Legal Tablet' mean?" we can easily see that it is a product description (it's in the Description tag that is inside a Product tag so it's a product's description). In Figure 9.6 you are looking at the definition of a product list using the Extensible Markup Language or XML (www.w3c.org). Where did the tags such as <Product> and <Price> come from? The authors made them up. That's the meaning of **"*extensible*" – one is free to "extend" any XML freely as long as a few simple rules that we will cover later are followed**.

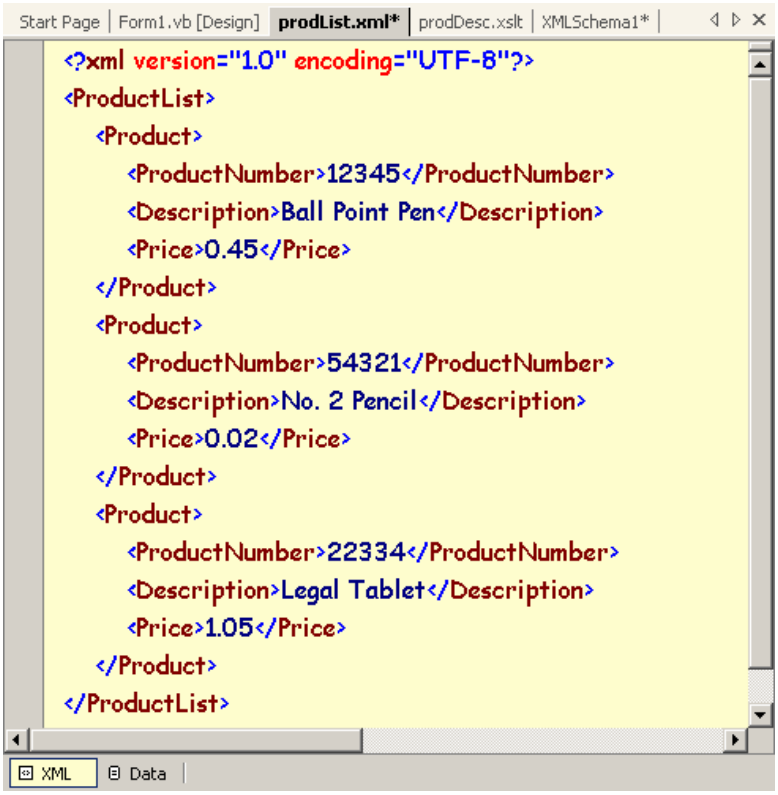


Figure 9.6 XML equivalent to the HTML content in Figure 9.2

However, you may be observing that the data doesn't look as good as the HTML rendering. You are correct but you need to understand that with XML, we separate the data content from the data presentation. Using several different techniques, we can present the same XML in a number of different ways. Figure 9.7 shows two different renderings of the XML from Figure 9.6

<combine into one figure>

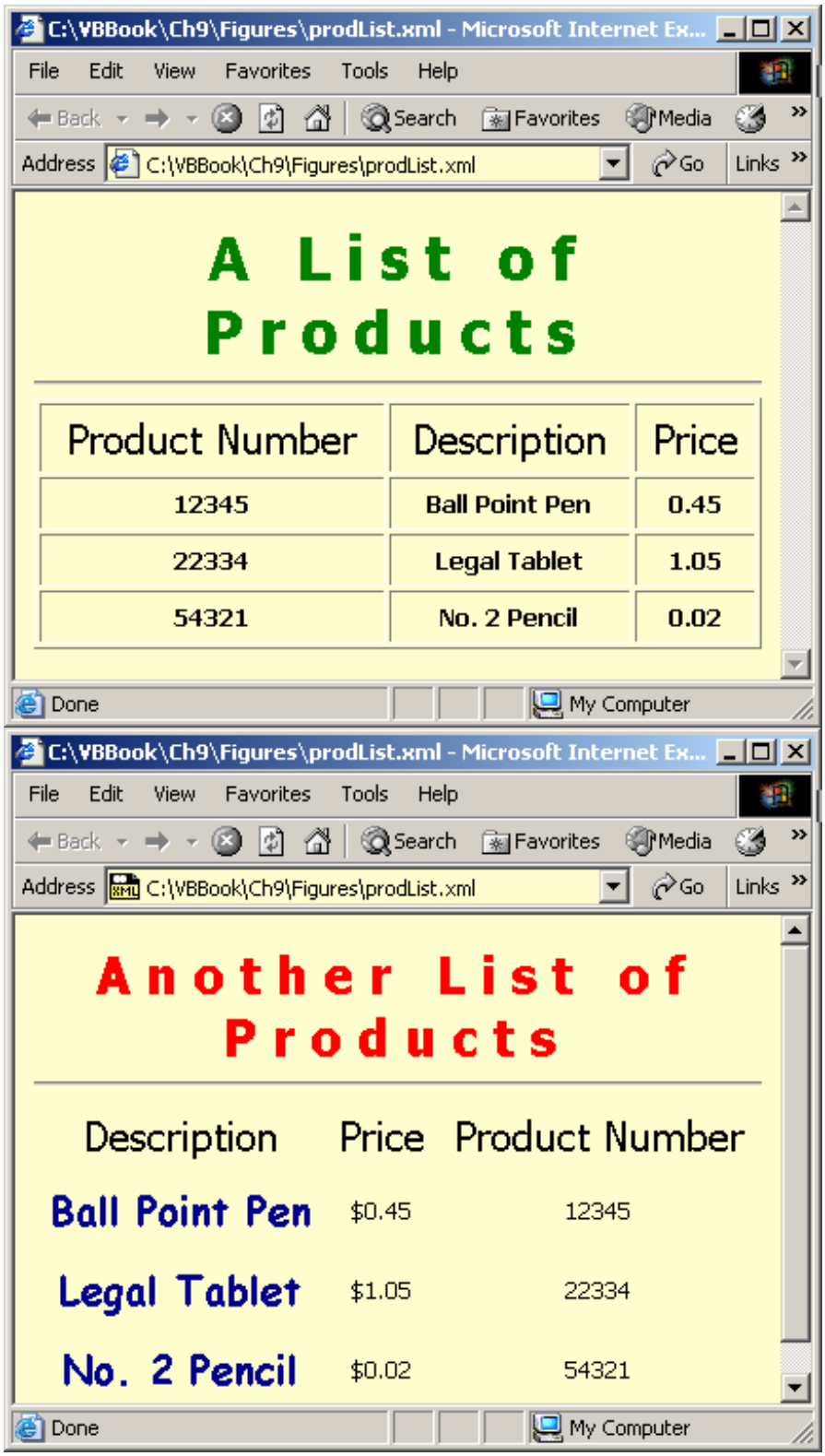


Figure 9.7 The same XML document rendered (displayed) in two different ways

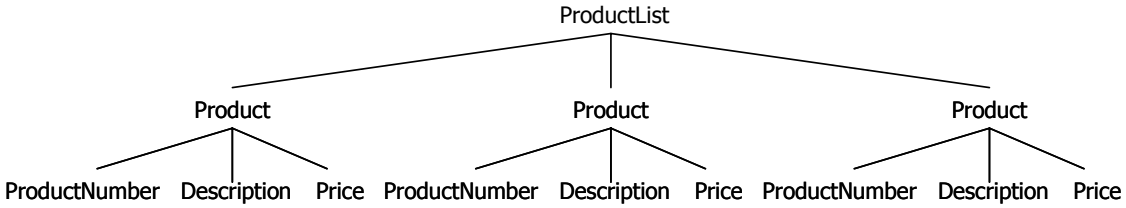
Both renderings in Figure 9.7 use the same XML file. Here you can see the power associated with separating data content from presentation – the same data can be rendered in any way that is useful. For example, one rendering might be HTML destined for a desktop browser (like shown in Figure 9.7), another rendering might be WML (Wireless Markup Language) destined for a wireless device, and a third rendering might be the original XML destined for a wholesale's electronic catalog. There is no practical limit to the number of different renderings possible for one XML file. One of the best examples is the use of XML in Newspaper industry. Many major newspapers have both a printed and a web version of their paper. By storing their news articles in XML, they can use the same information as the basis of both versions of their paper (since both are electronically rendered). This makes the publication of multiple formats very efficient.

As you might guess, XML is rapidly becoming the data format "standard" for the exchange of data on the Internet. Computer-to computer data transfers are possible because the computers can be programmed to find particular tags (such as <Price>) and use their content as appropriate. Standards, such as ebXML[3] (electronic business XML) for business-to-business transaction processing are already being used. In addition, companies are finding that the ability to transform XML data into a form that can be displayed in a browser, that is, transformed into HTML, makes it possible to create dynamic and current content.

XML data represent what is called a tree. A ***tree*** **is a data structure that is characterized by a single "root" with branches and leaves**. For example, that XML we saw previously in Figure 9.7 can be drawn as a tree as shown in Figure 9.8.

Figure 9.8 The "tree" view of XML

In Figure 9.8 "ProductList" is the root node, "Product" represents a branch node,



and "ProductNumber", "Description", and "Price" are leaf nodes. The textual content of elements (string or numeric) is usually associated with leaf nodes. It is also possible to assign some content values to non-leaf nodes using "attributes" that will be discussed later.

In addition to having just one root node, a non-root node (branch and leaf) can only be associated with a single node above it (a child node can have only one parent). Thus it would not be legal for a specific ProductNumber leaf node to be a child of more that one product.

To summarize, XML provides a way of storing structured data that is self-describing, i.e., the content meaning is more apparent due the use of "tags" that are meaningful in the context of the application. It is useful for exchanging data on the Internet either through computer-to-computer communications or as a source to be

[3] For more information on ebMXL see www.ebxml.org/.

transformed into a display for use by people. This ability to transform the "view" of the data for the particular viewer is a very important capability.

XML Syntax

XML syntax involves understanding and following a few straightforward rules. These include:

- The language is case sensitive. This means that the tags <price> and <Price> do not refer to the same thing.
- There is one and only one root node.
- All elements must have both a start tag and an end tag. This means that if you have <price> as a starting tag, you must have a corresponding closing tag. The closing tag could be </price>, that is, the same as the opening tag except a slash (/) must precede the tag name. There is a shortcut that can be used for leaf nodes if they have empty content. For example, if the <price> element had no content, you could say <price/> and this would be considered both the starting and ending tag for the "price" tag. Thus, <price></price> and <price/> are identical.
- Tags must be nested correctly. That is, one tag may be inside another tag ("nested") but its starting and ending tags must be within the starting and ending tags of the surrounding tag. The following is legal:

  ```
  <product>
      <price> 1.25 </price>
  </product>
  ```

  In this example, "price" is nested in "product". Illegal nesting is shown in the next example:

  ```
  <product>
      <price> 1.25 </product>
  </price>
  ```

  This is not legal because the price element's starting and ending tags are not both within the product element's starting and ending tags.
- If an element has an attribute, the attribute value must be quoted. ***Attributes* are values associated with a node**. For example, you might see the following XML fragment:

  ```
  <product hazardous="true">
      <name> nitroglycerin </name>
  </product>
  ```

  In this example, the product element has an attribute named hazardous. Each element can have zero or more attributes defined for it. The value of attributes must be enclosed in either single- or double-quotes.

You are free to make up element and attribute names. They must start with a letter or an underscore and can contain any number of letters, numbers, hyphens, periods, or underscores. However, keep the element names short and descriptive just like you have been doing with variable names.

If you follow theses rules your XML document is characterized as being **"*well formed*". This simply means that there are no syntax violations with the document**. There is another way to characterized XML documents. This is knows as "valid". We

will discuss valid documents later but for the time being understand that a document may be well formed but not valid.

Namespaces. What if you receive an XML document from a furniture supplier that has a <table> tag and you also get a different XML document from a tax advisor that also has a <table> tag? In the first document, <table> referred to a piece of furniture (like a table and chairs) while in the second document, <table> referred to a row/column oriented display of information (like the itemize deductions table). How can you resolve the duel meaning for the same element tag? You might be able to resolve the meaning by looking at the content of the two tags but this could be difficult or perhaps not possible. XML has a solution for this type of situation. The solution is to use something called a namespace (www.w3.org/TR/REC-xml-names/). A ***namespace*** **simply defines a point of reference**. In the furniture supplier's namespace we know what <table> means and in the tax advisor namespace we also know what <table> means and we know the two do not mean the same thing.

How do we use namespaces within an XML document? We simply change the element tag by adding a prefix and a colon to the tag. Using the examples from above, we might say <furn:table> and <tax:table> to differentiate the two types of elements. Be aware, however, if the two tags did not appear in the same XML document then there may not be a problem because the two documents themselves might be sufficient to establish the context.

Namespaces serve another function in addition to differentiated two tags within the same document. This second function allows XML parsers (a parser is a program that processes the XML to determine its content) to understand the context of a particular tag even if the tag itself is unambiguous in the specific document. For example, we will see later the XML statement:

```
<xsl:sort select="ProductNumber"/>
```

Here the namespace "xsl" stands for Extensible Stylesheet Language. When an XML parser sees this namespace, it knows that it must perform the "sort" function as defined within XSL.

How do we establish namespaces for our documents? To define a namespace, you add an xmlns (**xml n**ame**s**pace) declaration as an attribute within any element. All descendents of the element may then use the namespace. If you want the namespace available within the entire XML tree, you could place the xmlns declaration as an attribute of the root node. The declaration has the following syntax:

xmlns:name = "uri"

where you make up the name. The "***uri*****" (*Uniform Resource Identifier*) is a unique identifier and is often, but not necessarily, a url (Uniform Resource Locator**). We will say more about the uri later.

Figure 9.9[4] provides an example of creating a namespace definition and then using it later as part of a tag in the definition of a node.

[4] This is taken from an example by Alexander Falk that comes with XML Spy v4.0 (Altova, Inc.).

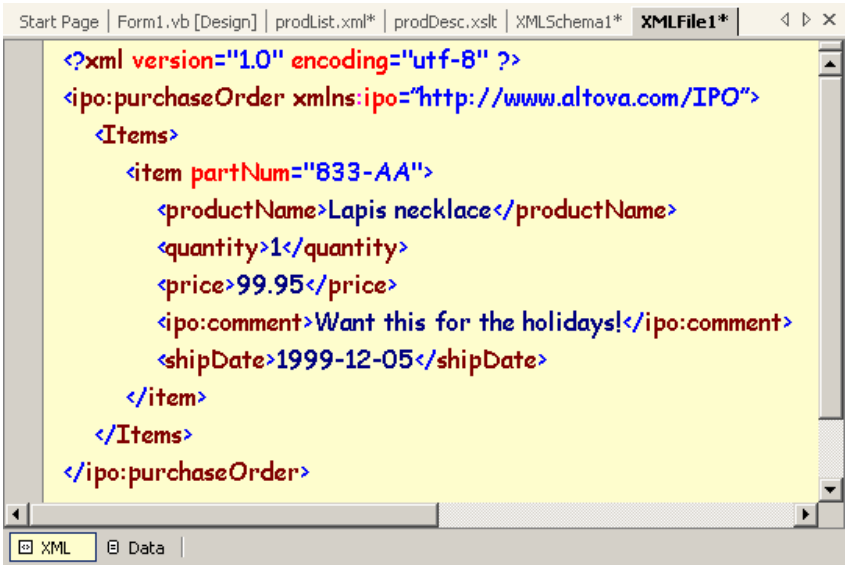


Figure 9.9 XML with a namespace definition

In this example the root node is the <purchaseOrder>. An attribute has been added to the definition of this node (xmlns:ipo="http://www.altova.com/IPO"). The name "ipo" could be any name. Note that the name is added to both the root node's starting and ending tags (ipo:purchaseOrder) as well as the comment node's starting and ending tags (ipo:comment).

The uri in the example (that is actually a URL) is "http://www.altova.com/IPO". If you were to go to this site with a browser you would get an error because there is no HTML content there. The sole purpose of this uri is to be sure it is unique. Since the company Altova, Inc. owns the URL www.altova.com, it has full control of the URL and no other firm can use this. Thus, as a mater of convenience, the uri, which must be unique, is often created using a URL.

Sometimes the uri is not constructed using a URL but in this case there is usually an international standards organization that guarantees the uniqueness of the uri. In addition, when a URL is used as the uri, some organizations put content at the URL that documents the namespace. We should add that "uniqueness" is only necessary within a document, that is, two different and unrelated entities may by chance choose the same uri. As long as the two uris do not appear in the same document there will be no problem.

Document Prolog. The ***document prolog*** **indicates that the document is XML as well as other things such as document type, entity definitions and other processing instructions**. Here we cover the just the XML declaration and not entity definitions or other processing instructions.

The first line in any XML document indicates that the document is XML and declares the version of XML being used. An example is:

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

There are additional attributes available besides the version attribute. These include how the document is encoded (the character set used) as well as an indication of other files that this document needs to be loaded for it to operate correctly. Note the special starting and ending tags for this entry (<? and ?>). These tags are used to define a

processing instruction (sometimes referred to as PI), that is, **information used by the XML parser and not part of the actual data content**.

XML Schemas

Suppose two business partners decide to exchange data using XML documents. How do they communicate what is legal in the documents and how can they verify that a document they receive from the other follows the rules? Imagine if there were no rules, then each partner would be free make up tags that the other partner would not expect or not even understand. In addition, the two partners would also be free to use different tags for the same content. One might use <po> for purchase order and the other might use <purchOrder> for the same thing. You can imagine how difficult communication would be without an agreed upon set of rules for valid XML documents.

XML has two ways of defining the rules for valid XML documents: Document Type Definitions (DTD) and XML Schemas (www.w3c.org/XML/Schema). The DTD was the original tool used for this purpose. However, it is rapidly being replaced with the newer, and by most accounts much better, XML Schema. We will not cover the DTD and focus instead on the XML Schema because it is considered better and because it is the standard used by Microsoft in Visual Basic .NET.

Consider the XML we saw earlier in Figure 9.6 (reproduced in Figure 9.10).

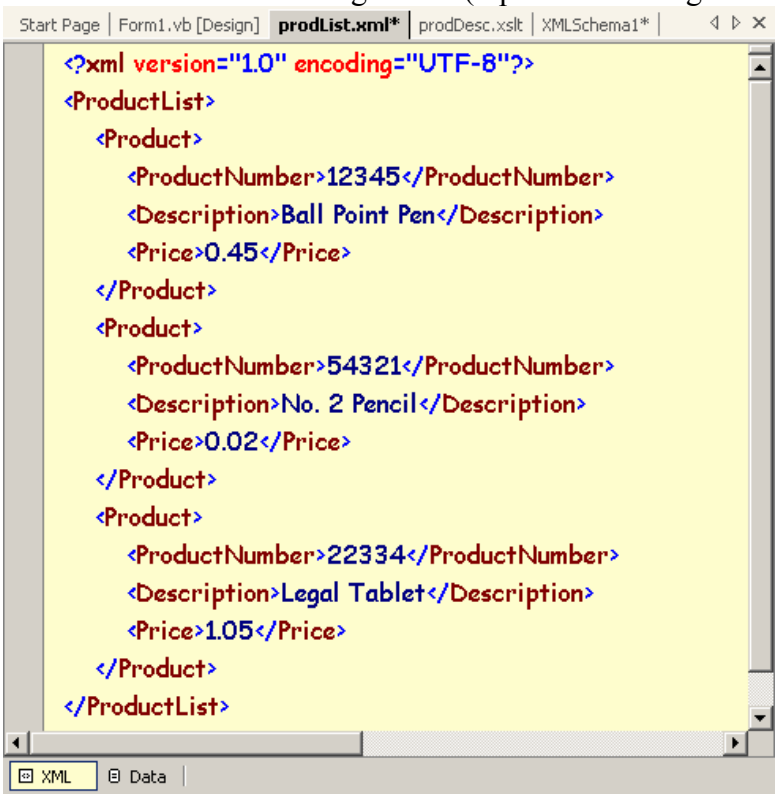


Figure 9.10 The XML from Figure 9.6

An XML Schema would define what was legal as far as elements and tags are concerned as well as what the content would consist of. Figure 9.11 shows an XML Schema that would support the XML document in Figure 9.10.

```
<?xml version="1.0" ?>
<xs:schema id="ProductList"
    targetNamespace="http://tempuri.org/ProductList.xsd"
    xmlns:mstns="http://tempuri.org/ProductList.xsd"
    xmlns="http://tempuri.org/ProductList.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified" elementFormDefault="qualified">
  <xs:element name="ProductList" msdata:IsDataSet="true" msdata:EnforceConstraints="False">
    <xs:complexType>
      <xs:choice maxOccurs="unbounded">
        <xs:element name="Product">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="ProductNumber" type="xs:string" />
              <xs:element name="Description" type="xs:string" />
              <xs:element name="Price" type="xs:float" />
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:choice>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Figure 9.11 XML Schema definition

This discussion is not intended to provide the degree of detail so that you can become an expert at creating XML Schemas. In the next Section 9.2 we discuss some tools available within Visual Basic .NET that help you build a Schema without having to know all the syntax rules. In fact, the schema code shown in Figure 9.11 was created with one such tool. However, it would be helpful here to talk about the schema code in Figure 9.11 to enhance your understanding.

First note that an XML Schema is in fact just an XML document (as you can see this in the document prolog in the first line of the schema). This means that you already know the basic syntax rules associated with writing the schema. The second through eighth line is the root element that defines the schema <xs:schema …>. In addition to defining the root node, this statement also defines the id, targetNamespace, attributeFormDefault and elementFormDefault attributes. Finally, it defines four namespaces including the namespace "xs" that qualifies the remaining tags in the definition. Do not worry about all this detail at this time. In fact, this element definition, as generated by Visual Basic .NET, is actually more complex than it need be. Often, when a tool creates code, it produces a very general version that is more verbose than well-designed custom code would be.

The schema then defines an element named ProductList as a complex type. A ***complex type* is one that consists of additional elements**. The ProductList element is made up of an unbounded number of elements that are chosen from the list of Product elements. Again this "choice" element is really not necessary when you are selecting

from a choice of one element, but for a general solution, the choice element is provided. The Product element is defined as a complex type that consists of a sequence of elements named ProductNumber, Description, and Price. These final three elements store content of type string, string, and float respectively.

This is not the only way to write the schema. That is, there are generally several ways to write a schema for the same XML document. There are additional tags that can be used within the schema definition. We will see some additional tags in Section 9.2 when we develop some XML and XML Schemas.

Once an XML Schema has been developed for an XML document, that document can be validated against the schema. That is, the XML document is compared to the rules defined in the schema and if any rule is broken, then the XML document will be considered as invalid. Note that it can still be well-formed, that is, it does not violate any of the syntax rules outlined earlier in this section, but still be invalid. Well-formed refers to basic syntax rules while ***valid*** **refers to a well-formed document following some set of additional rules controlling how the elements are arranged within the document**.

Finally, a number of industry and standards groups are developing XML schemas that define documents specific to their industry or specialized needs. For example, ***ebXML*** **(electronic business XML – www.ebxml.org)** provides a set of schemas for various transactions and other common documents that are used in electronic business. Another example is ***XBRL*** **(Extensible Business Reporting Language – www.xbrl.org).** Through the adoption of these open standards, business partners can exchange documents via XML with the assurance that others will be able to understand and process them.

Styling XML

In Figure 9.7 we saw two different renderings of the same XML document. This was done with the help of Extensible Stylesheet Language Transforms (XSLT – www.w3.org/Style/XSL). XSLT provides a means of transforming one XML document into a second XML document. Figure 9.12 shows how this transformation process works.

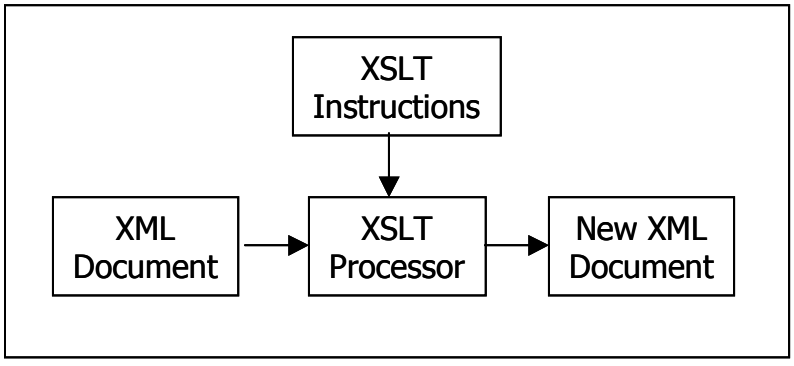


Figure 9.12 Overview of the XSLT Transformation process

As you can see in Figure 9.12, an XML document and XSLT instructions are processed by an XSLT processor to create a new XML document. In our case, the new

XML document was also an HTML document. Note that HTML is legal XML as long as it follows the rules we defined earlier. This is a common transformation process by which we can transform an XML document into any valid XML document. Other documents that are valid XML include the Wireless Markup Language (WML) and HTML Basic that support wireless devices. XSLT processors are available in a variety of software products. For example, Internet Explorer includes an XSLT processor that can transform and render XML documents into HTML for display. Visual Basic .NET includes classes and methods that also have the ability to process XML using an XSLT processor. There are also a number of XSLT processors available for free as both open source and freeware products. We will be using Internet Explorer and Visual Basic .NET for our processing of XML and XSLT.

We must repeat again the caution that XSLT, like the XML Schema, is very complex and our coverage here is just an overview to show you some examples and provide you with a high-level awareness of the technology and its applications. XSLT provides the means to take the original XML document (remember it is a tree structure), and select the entire tree or any sub trees (called pruning ☺) and format or rearrange the nodes of the new tree into a new XML document.

Figure 9.13 shows our original XML and the Internet Explorer rendered version of the transformed XML (transformed into HTML using an XSLT document).

<combine into one figure>

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="prodDesc.xslt"?>
<ProductList>
  <Product>
    <ProductNumber>12345</ProductNumber>
    <Description>Ball Point Pen</Description>
    <Price>0.45</Price>
  </Product>
  <Product>
    <ProductNumber>54321</ProductNumber>
    <Description>No. 2 Pencil</Description>
    <Price>0.02</Price>
  </Product>
  <Product>
    <ProductNumber>22334</ProductNumber>
    <Description>Legal Tablet</Description>
    <Price>1.05</Price>
  </Product>
</ProductList>
```

Figure 9.13 An XML document and its rendered transformation show in Internet Explorer

Note that in the second line of the XML document you see the statement:

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

This statement is a directive telling the browser that a stylesheet has been defined for it and that stylesheet is found in a file named prodDesc.xslt. Figures 9.14a and 9.14b show the contents of the XSLT document stored in the prodDesc.xslt file.

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

<!--  Template #1  -->
<xsl:template match="ProductList">
 <html>
  <body>
   <div style="font-family:Tahoma,Arial,sans-serif;
            font-size:20pt; color:red;
            text-align:center; letter-spacing:8px;
            font-weight:bold">
       Another List of Products
   </div>
   <hr />
   <table width="100%" cellpadding="5" border="0"
                style="font-family:Tahoma,Arial,sans-serif;
            font-size:16pt; color:black;
            text-align:center">
      <tr>
           <td>Description</td>
           <td>Price</td>
           <td>Product Number</td>
```

```
        </tr>
      <xsl:for-each select="Product">
      <xsl:sort select="Description"/>
        <tr>
            <xsl:apply-templates select="Description" />
            <xsl:apply-templates select="Price" />
            <xsl:apply-templates select="ProductNumber" />
        </tr>
    </xsl:for-each>
    </table>
    <div style="font-family:Arial,sans-serif;
            font-size:8pt; margin-left:10px">
    </div>
   </body>
 </html>
</xsl:template>

<!--  Template #2  -->
<xsl:template match="Description">
 <td style="font-family:Comic Sans MS, Arial,sans-serif;
         color:darkblue; font-size:16pt;
         font-weight:bold">
  <b><xsl:value-of select="."/></b>
 </td>
</xsl:template>

<!--  Template #3  -->
<xsl:template match="ProductNumber">
 <td style="font-family:Tahoma,Arial,sans-serif;
         font-size:10pt">
  <xsl:value-of select="."/>
 </td>
</xsl:template>

<!--  Template #4  -->
<xsl:template match="Price">
 <td style="font-family:Tahoma,Arial,sans-serif;
         font-size:10pt">
  $<xsl:value-of select="."/>
 </td>
</xsl:template>

</xsl:stylesheet>
```

Figure 9.14a First part of the XSLT document used to transform the XML into HTML

Figure 9.14b Remainder of the XSLT document used to transform the XML into HTML

Let's look at the XSLT to try and see what is going one. First note that the prolog (first line of code) defines this as an XML document so just like XML Schemas, XSLT documents are also XML. The second line defines the stylesheet element. This includes the definition of the appropriate namespace (xsl) plus setting the version attribute value. Line 3 defines the output element that is used by the XSLT processor to determine what type of output to generate. In this case, the method attribute indicates that XML will be the output type.

The sixth line, <xsl:template match="ProductList">, starts the definition of a template element. Templates are the primary method used by XSLT to define how the transformation should look. You may have used "templates" for drawing. Drawing templates define the basic shape of an object and you use the template with a pen or pencil to draw the object on paper. **XSLT *templates* work in a similar way – they define the basic shape of the resulting document and what "parts" of the original XML tree to include in this new document**.

To understand the XSLT, we need to review the tree structure of the original XML document. Figure 9.15 shows this tree.

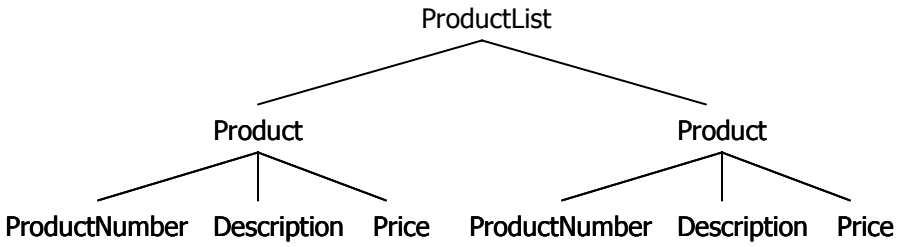


Figure 9.15 The tree representation of the original XML document

Now look at the XSLT in Figure 9.14 and you will see four templates (xsl:template). Each of these templates has a "match" attribute. The first template matches "ProductList", the second matches "Description", the third "ProductNumber", and the fourth "Price". The first template, the one that matches "ProductList", is a template that applies to the entire tree (because it matches the root). The other three templates each apply to one specific leaf node.

The template that applies to the entire tree should be considered the "master" template in that it controls the overall structure of the new XML document (tree). You may recognize that the first few lines within the first template contain HTML. This HTML will be part of the new tree (remember that HTML that follows our XML rules is also XML). The first XSLT code within the first template is shown in Figure 9.16.

```
<xsl:for-each select="Product">
<xsl:sort select="Description"/>
    <tr>
```

```
            <xsl:apply-templates select="Description" />
            <xsl:apply-templates select="Price" />
            <xsl:apply-templates select="ProductNumber" />
        </tr>
    </xsl:for-each>
```

Figure 9.16 The XSLT code within the first template.

The first statement (xsl:for-each) is very much like a Visual Basic .NET "For Each…Next Structure" loop. In the case of XSLT, we are building a loop that will repeated for each Product node in the XML document. Our document has three product nodes (Product numbers 12345, 54321, and 22334) so the loop will iterate 3 times, once for each product node. The next statement (xsl:sort) does exactly what it says, it sorts the nodes in the resulting tree in order by their Description field. You then see the HTML definition of a table row "<tr> … </tr>. This means that the three xsl:apply-templates instructions will generate content that will be within an HTML table row. The three xsl:apply-templates statements, each with different select attribute values, will search for a corresponding template and apply it to generate content.

The first xsl:apply-templates statement selects the Description node. The template for that node is shown in Figure 9.17.

```
<!--  Template #2  -->
<xsl:template match="Description">
 <td style="font-family:Comic Sans MS, Arial,sans-serif;
        color:darkblue; font-size:16pt;
        font-weight:bold">
  <b><xsl:value-of select="."/></b>
 </td>
</xsl:template>
```

Figure 9.17 Template for the Description node in the XML tree

Here we see that an HTML table column definition is being created <td> … </td>. Within this table definition, the xsl statement:

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

is found (enclosed in the HTML bold tag (…)). The select="." attribute means to select the content at the current node. Since the current node is the Description node, the content is whatever is stored at this leaf node, e.g., "Ball Point Pen".

The order of the xsl:apply-templates in the loop determines the order in which the columns within each row of the table will be filled (Description first, then Price, and finally the ProductNumber).

Of course, XML trees can be much more complex than our example, so XSLT has many additional capabilities than we see here. We encourage you to research this important topic in more depth but hope that at this point the general ideas associated with transforming one XML document into another XML document are clear.

Exercise 9.1. Can an XML document be valid but not be well formed? Explain.

Exercise 9.2. Write an XML document that describes the books in a library. When describing a book, include just three or four elements. For your example, put information for three books.

Exercise 9.3. Given the following XML document, construct a tree that represents the underlying structure.

```
<?xml version="1.0"?>
<StudentList>
  <Student>
     <StNo>1111</StNo>
     <Name>Skippy</Name>
     <Class>1</Class>
  </Student>
  <Student>
     <StNo>2222</StNo>
     <Name>Karen</Name>
     <Class>2</Class>
  </Student>
  <Student>
     <StNo>3333</StNo>
     <Name>Joe</Name>
     <Class>3</Class>
  </Student>
</StudentList>
```

Exercise 9.4. Why would you use an XML Schema and why would you use an XSLT document?

Exercise 9.5. Assume you have information on a student that includes a student number, name, address, and courses taken. The address is made up of street address, city, state, and zip. Information on a course includes course number, description, credit hours and grade received.

Given this information, create a tree that correctly models this data and write a short XML document that is consistent with the tree. Populate the XML document with information on two students who each have taken at least 3 courses.