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.

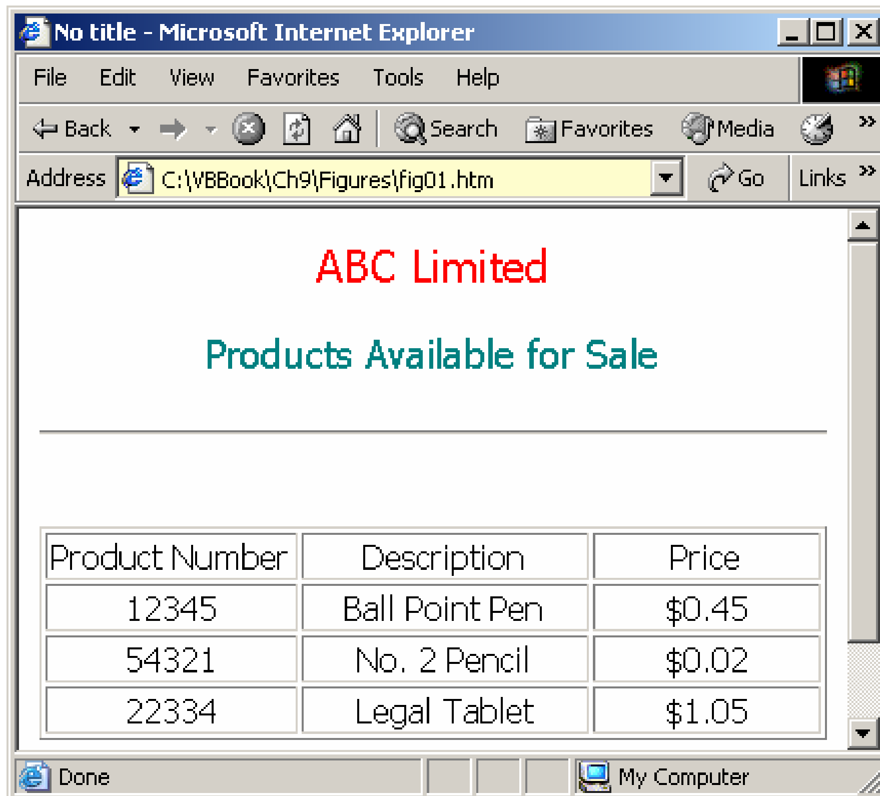


Figure 9.1 Typical web page formatted using HTML

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

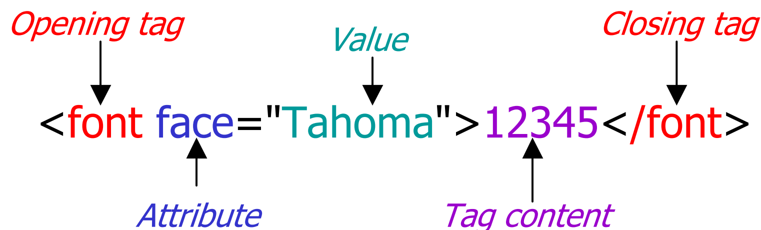
We will focus on the third line of Figure 9.3 that defines one table column using the following HTML:

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

What does this mean? The <p> tag defines a new paragraph and </p> 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.

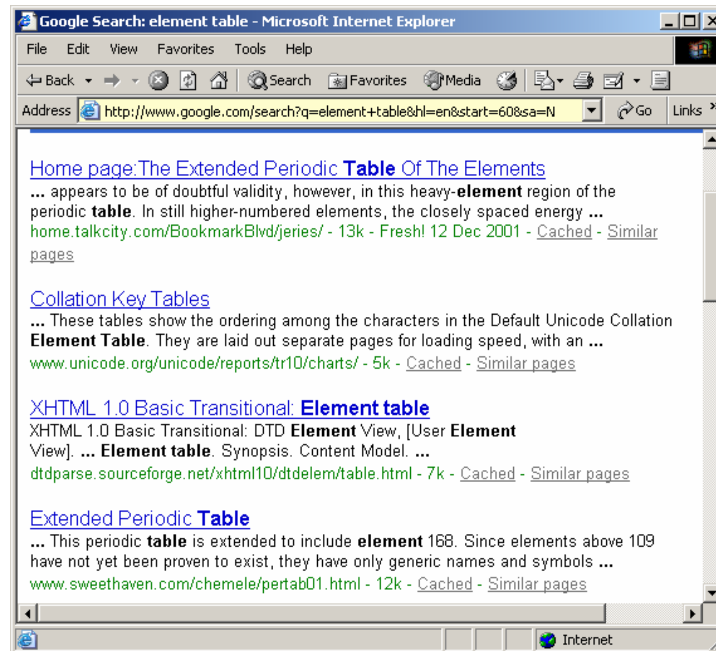


Figure 9.5 The results of a search engine searching for information on "element table"

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**.

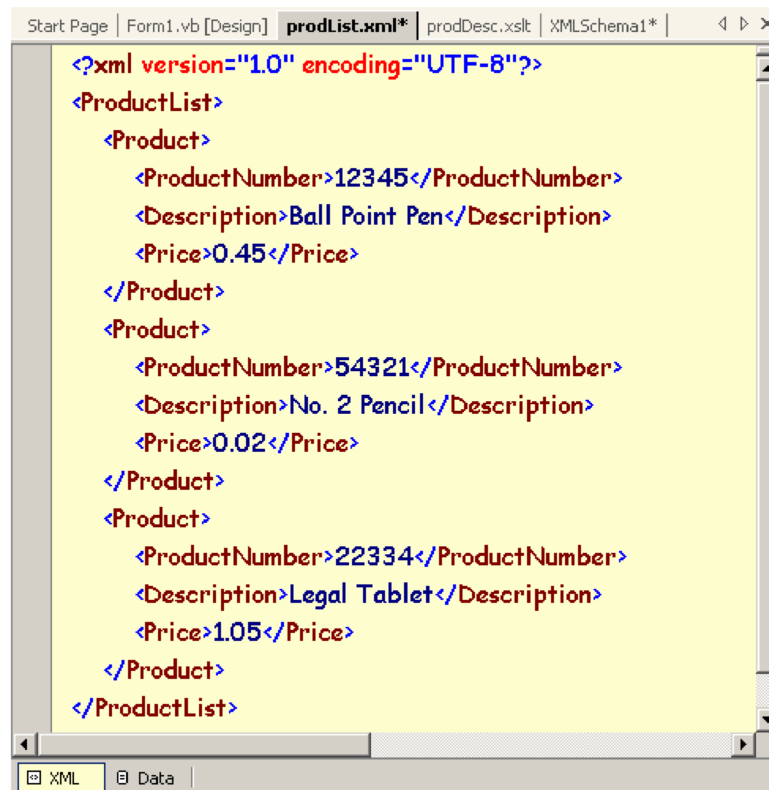


Figure 9.6 XML equivalent to the HTML content in Figure 9.2

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>

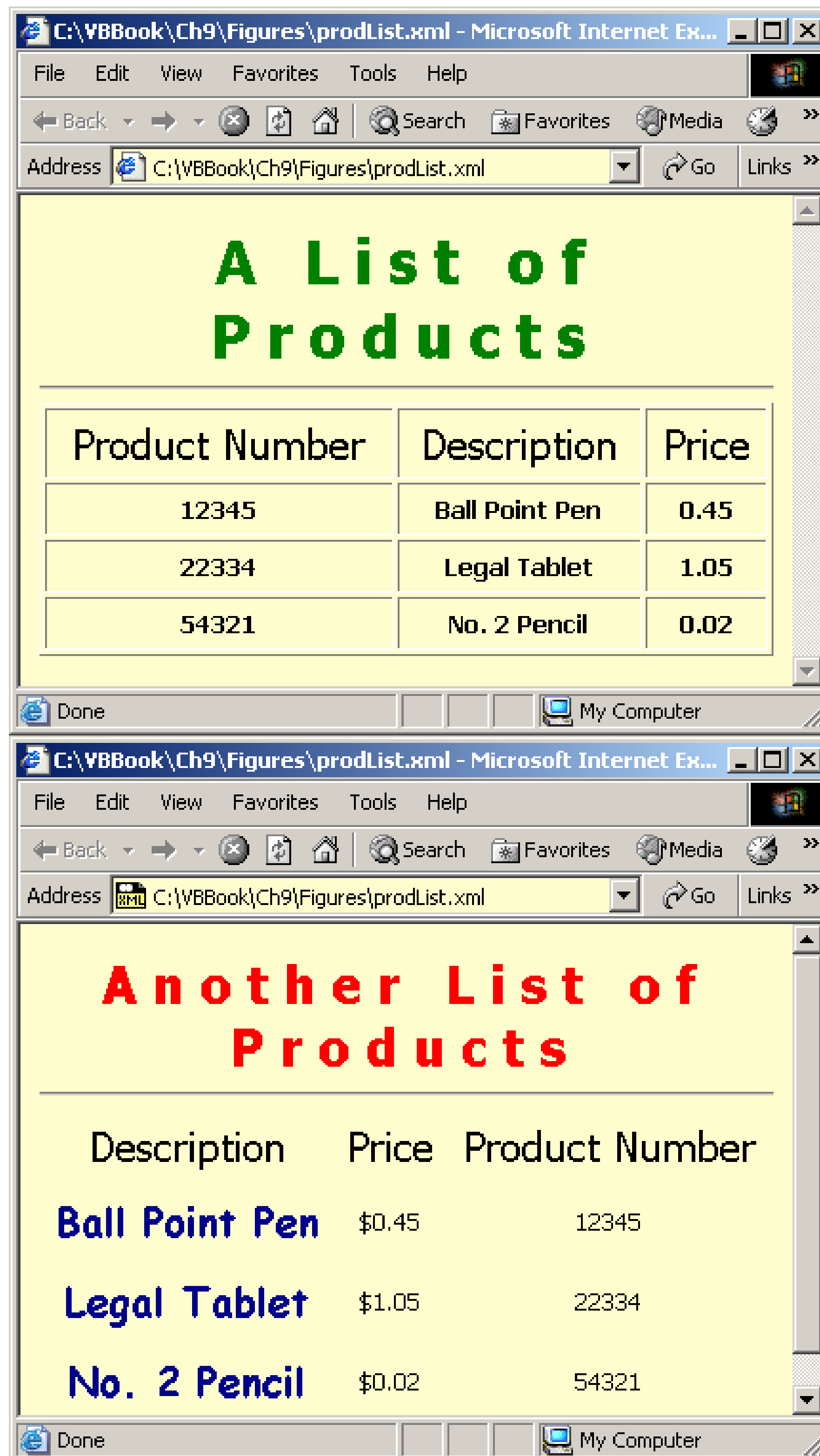


Figure 9.7 The same XML document rendered (displayed) in two different ways

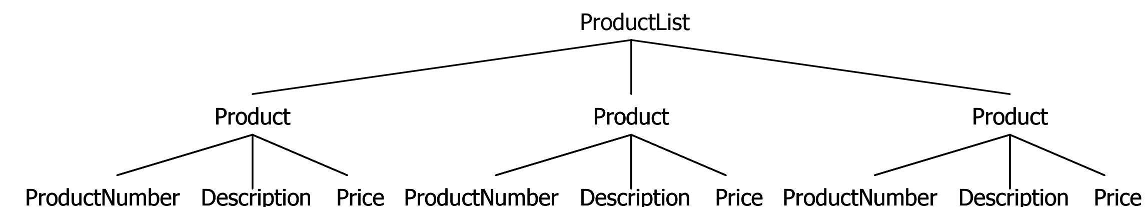
Both renderings in Figure 9.7 use the same XML file. Here you can see the power associated with separating data content from presentation – the same data can be rendered in any way that is useful. For example, one rendering might be HTML destined for a desktop browser (like shown in Figure 9.7), another 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.).

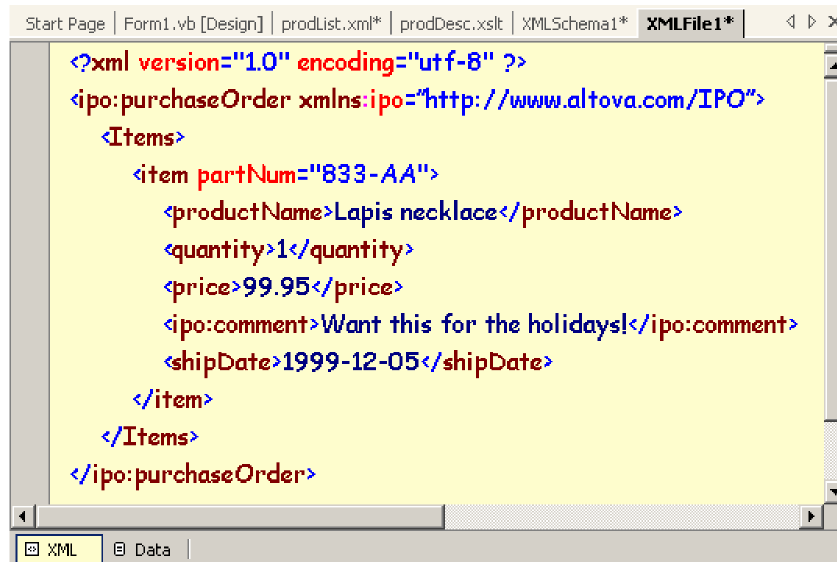


Figure 9.9 XML with a namespace definition

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).

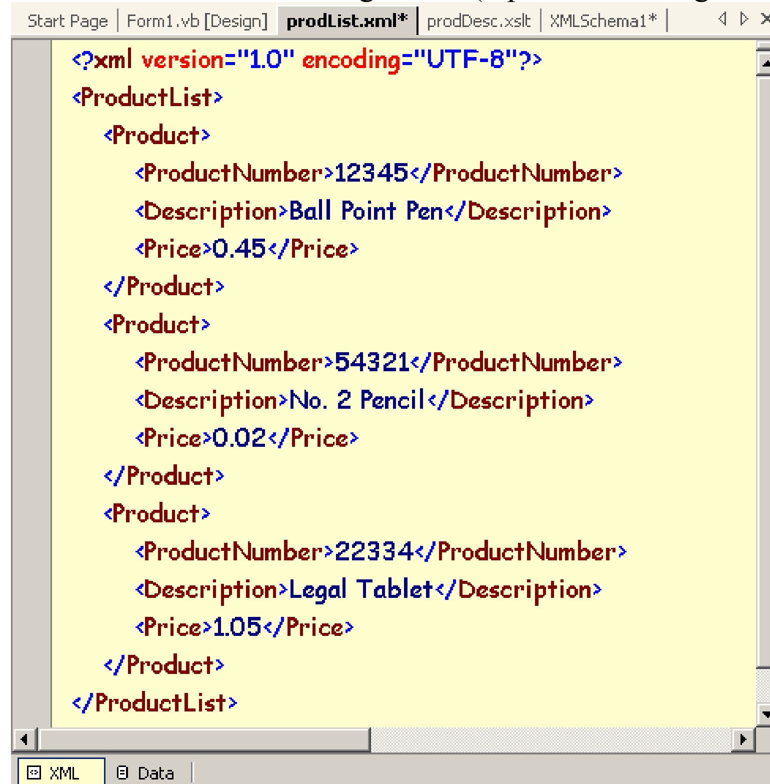


Figure 9.10 The XML from Figure 9.6

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.

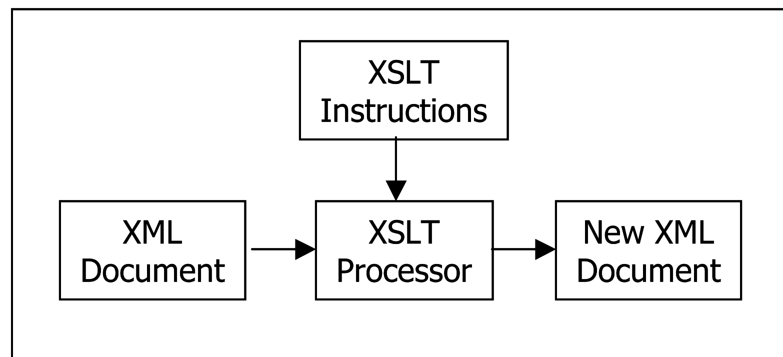


Figure 9.12 Overview of the XSLT Transformation process

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>
```

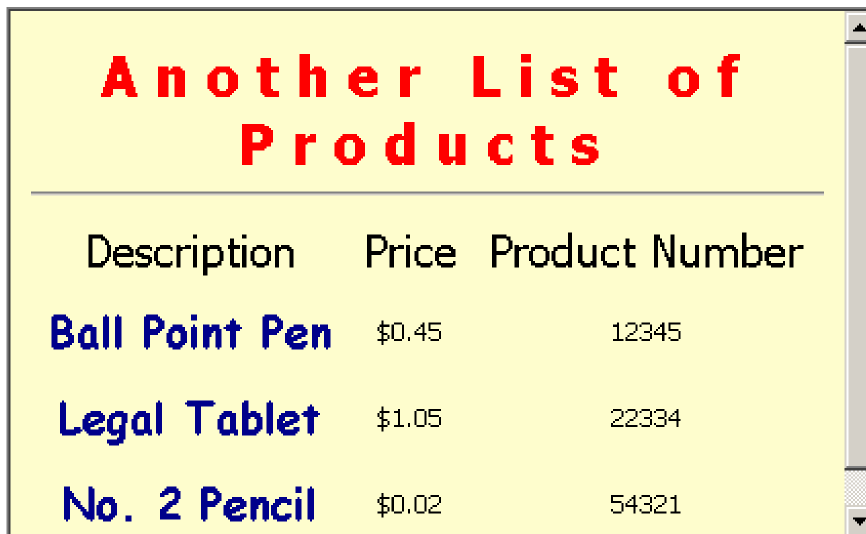


Figure 9.13 An XML document and its rendered transformation show in Internet Explorer

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.

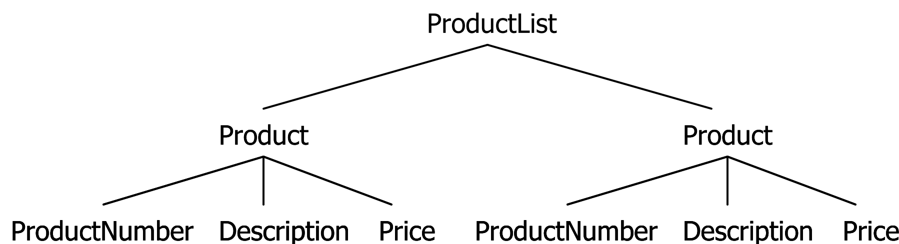


Figure 9.15 The tree representation of the original XML document

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.

9.2 CREATING XML DOCUMENTS AND XML SCHEMAS USING VISUAL BASIC .NET

Now that we have seen the basic ideas behind XML and XML Schemas, we turn our attention to Visual Basic .NET and the tools it provides that allow us to design schemas and then use them to build XML documents. Visual Basic .NET includes an XML Schema design tool that allows us to create XML Schemas using a graphical tool while it creates the actual schema code in the background. As with most graphical tools, the schema code generated in the background may not be as good as code written by a person who is an expert in XML Schema design. However, since our intention is not to teach you XML Schema design in depth, the graphical design tool is more than adequate. Since you can see the schema code generated by the tool, this can be effective in helping you understand the schema code.

Creating An XML Schema

We will first create a very simple schema that defines the information we might store regarding a student. The information relating to our student will include a student number, name, address, and age. To keep things simple, we will start with a definition for a single student (not a list of students).

We begin by creating a new Visual Basic .NET project and choosing a new Windows Application. After creating the new project, we need to add an XML Schema to the project by selecting Add New Item… from the File menu. The ensuing dialog box, shown in Figure 9.18, shows that the XML Schema icon has been chosen. Give this schema a new name if you like (we will leave the name as XMLSchema1.xsd).

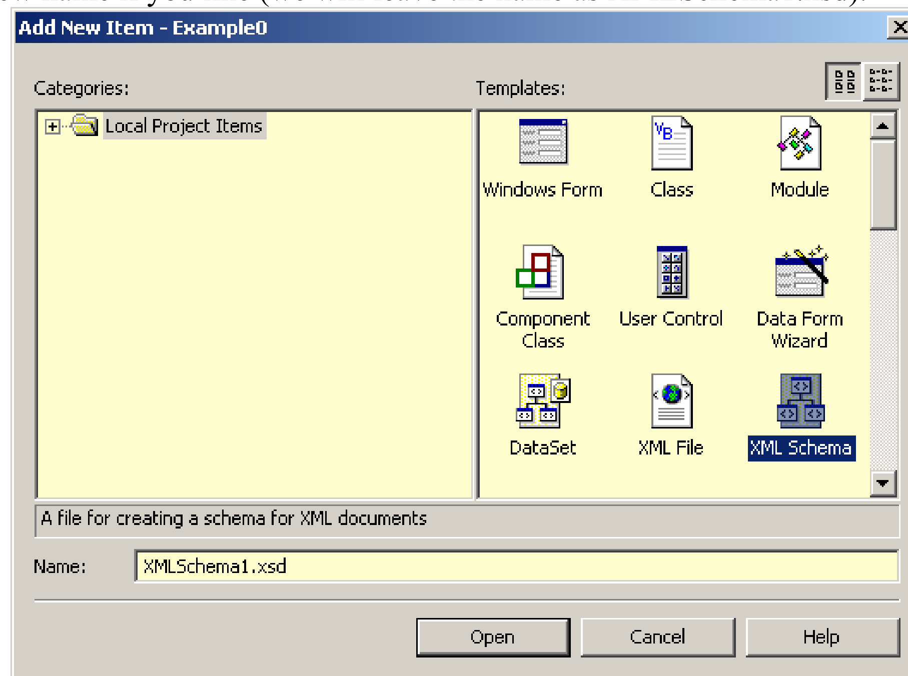


Figure 9.18 Add a new XML Schema to the project

After clicking on Open in the Add New Item dialog box, Visual Basic .NET will display the schema in the designer window. Figure 9.19 shows what you should see. Note that you can now drag items from either the Server Explorer of the Toolbox to create your schema design.

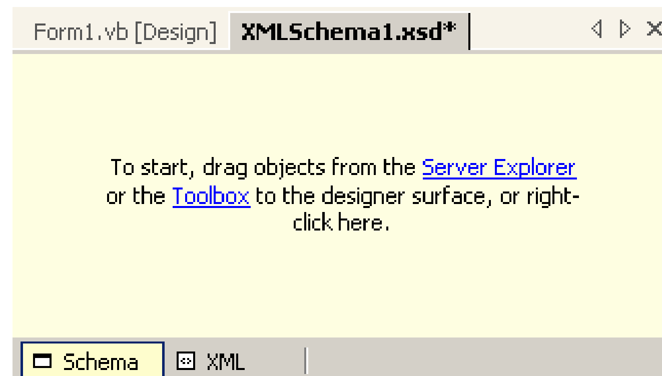


Figure 9.19 The designer window for a new XML Schema

We will create our schema using the tools in the toolbox. If you open the toolbox, you will see some new components designed exclusively for XML Schemas. Figure 9.20 shows what this toolbox should look like.

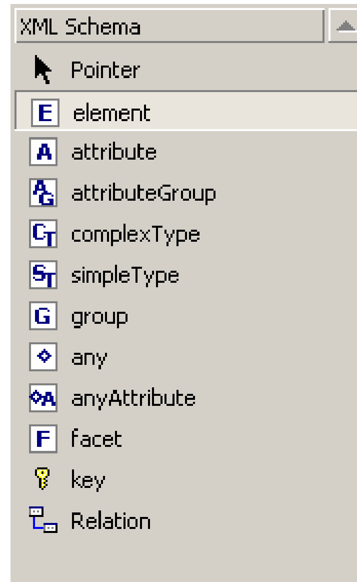


Figure 9.20 The XML Schema toolbox with the “element” tool selected

For our student example we need to create an element. Our element will store four facts (student number, name, address, and age) for the student. Drag an “element” component to the designer window. A new element (element1) will appear in the designer as shown in Figure 9.21.

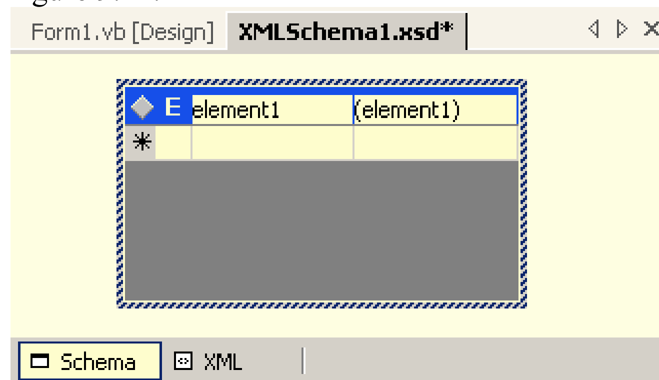


Figure 9.21 The element component in the designer window

The box in the top row next to the (E) icon should include the name of the element. In our case we should change this to Student. Starting in row 2, you need to define each child node of the Student tree. The first column is used to indicate the category of the each node. In our case these are also elements. The second column is used to hold the name of the node, and the third column defines the type of the node. Figure 9.22 show the drop-down lists for the first and third column.

<combine into one figure>

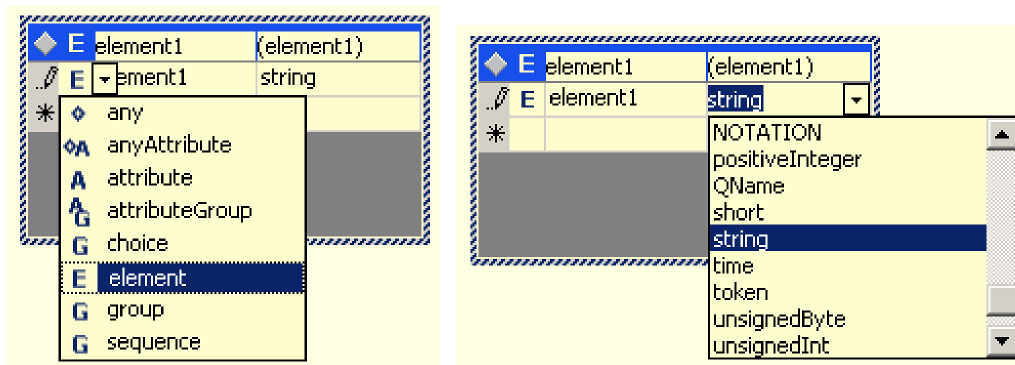


Figure 9.22 Choosing a category and type of a node

When you finish you should see the Schema definition of the element named Student as shown in Figure 9.23.

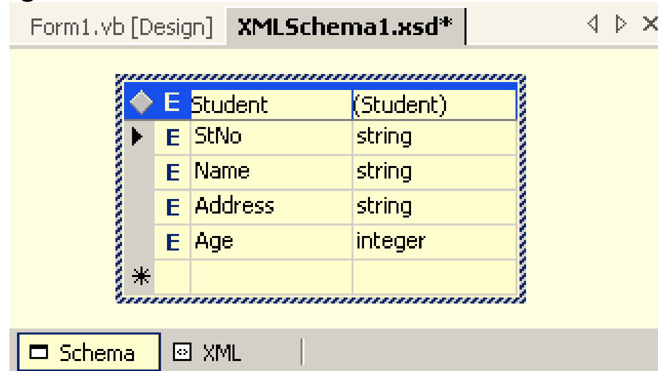


Figure 9.23 The final Schema design

Before we proceed, it is ***important to save the Schema definition***. Before we can work with the Schema definition, it must exist as a file on disk, not just as a document within the Visual Basic .NET development environment.

We can see the actual schema code by clicking on the "XML" tab at the bottom of the designer. Figure 9.24 shows this code. Even though we have not covered the XML Schema language, the XML code in the figure should be fairly clear. After defining the XML prolog and the schema element, we see an element named Student that includes a complexType that consists of a sequence of elements named StNo, Name, Address, and Age that are all string type except for Age, which is an integer type. The complexType element was added so that the student element could contain nodes, not just content. That is, the Student element is the root node and includes four additional elements (leaf nodes) named StNo, Name, Address, and Age.

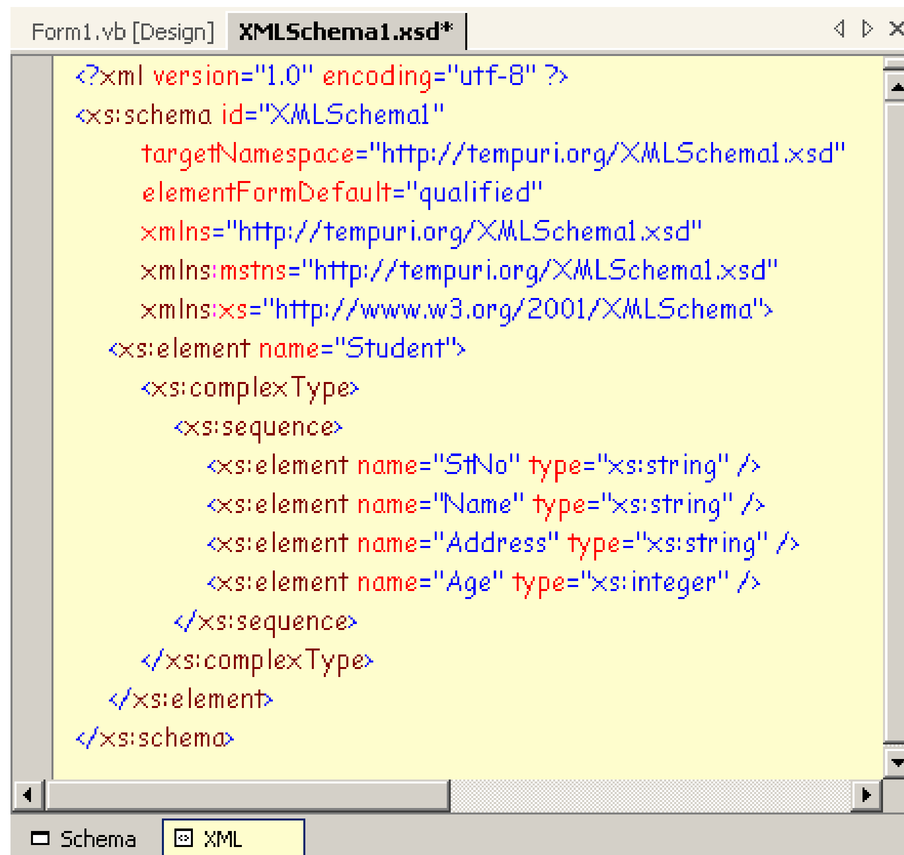


Figure 9.24 The XML Schema code for Student

Creating an XML Document

Now that we have a schema and it has been saved on disk, what can we do with it? One thing is to use it to create an XML document. Go to the File menu and select Add New Item…. In the Add New Item dialog box, click on the XML File icon; name it Student.xml, and then click on Open (see Figure 9.25).

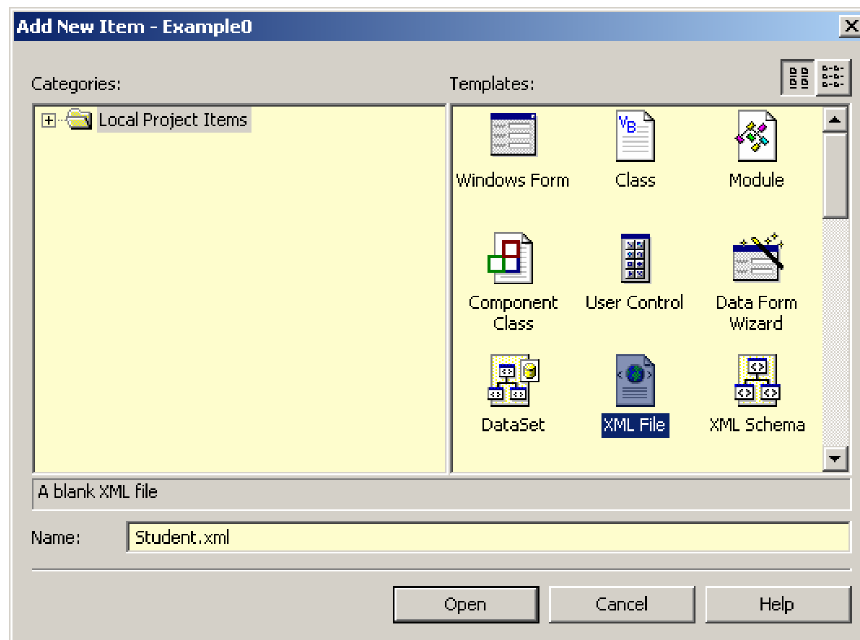


Figure 9.25 Opening a new XML File

After opening the new XML file you should see a blank XML document. The only XML code in this file is the initial prolog statement that makes this an XML file. Figure 9.26 shows this new XML file.

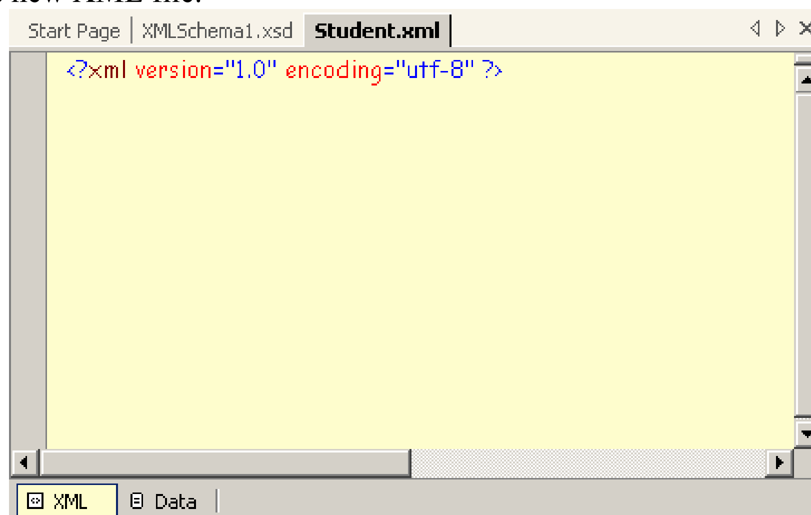


Figure 9.26 The new XML file

We next want to associate the schema we just defined with this new XML file. To do this, be sure the XML file is active in the designer and click on the targetSchema property in the Properties window. Then select the correct XML Schema file from the dropdown list as shown in Figure 9.27.

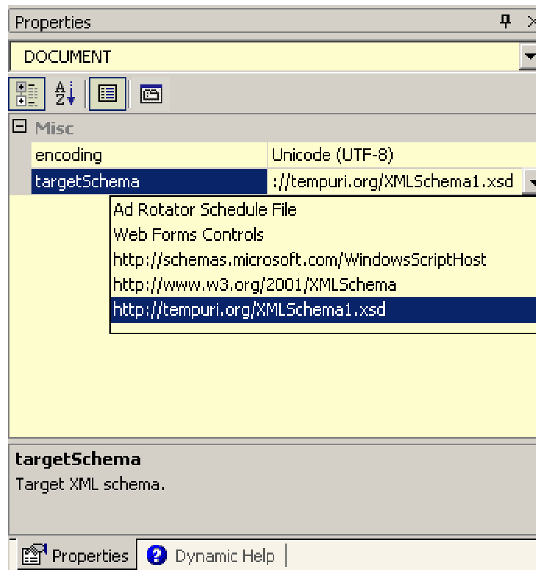


Figure 9.27 Selecting the target schema for an XML file

At this point you should see the XML code modified to include the root node (Student) as well as a reference to the XML Schema. If you click on the Data tab at the bottom of the designer, you should see the XML displayed as a table. Figure 9.28 shows these two views.

<combine into one table>

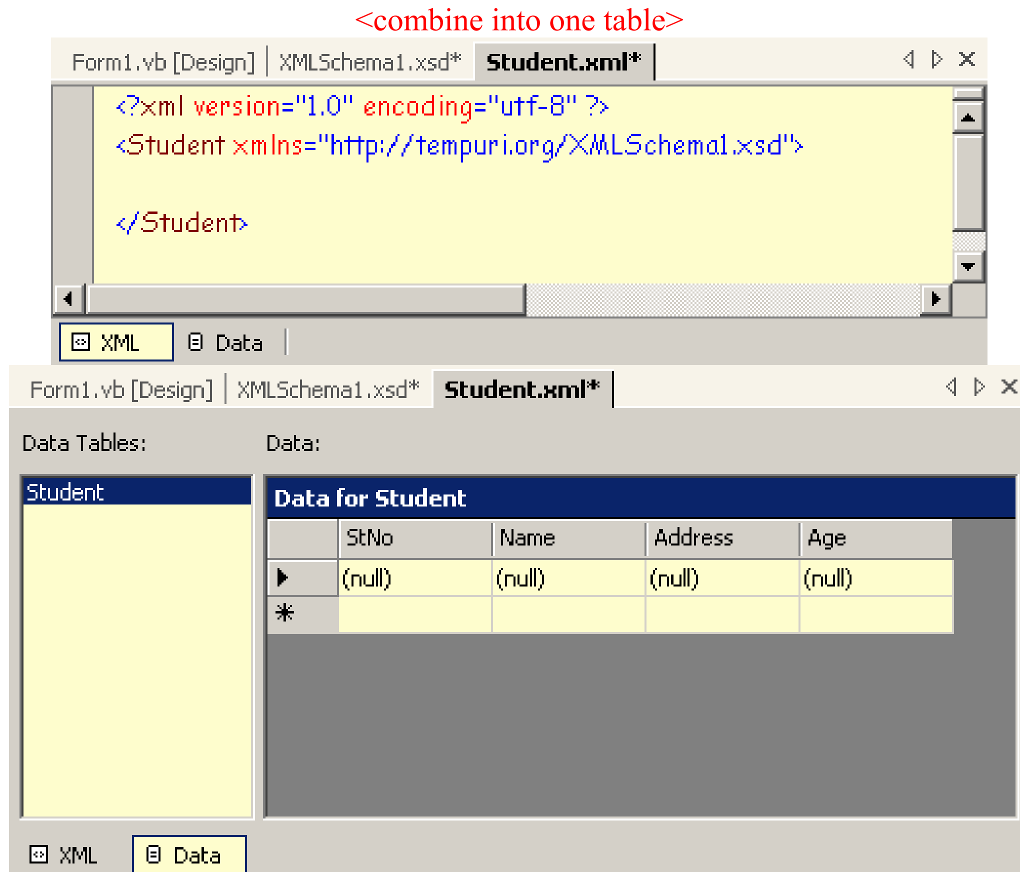


Figure 9.28 The XML and Data views of the XML document

Using the Data view, add several students to the table. After doing so, check the XML view and you should see that their data has been formatted and entered into the XML file surrounded by the appropriate tags. Figure 9.29 shows both the XML and Data views after entering some data.

<combine into one figure>

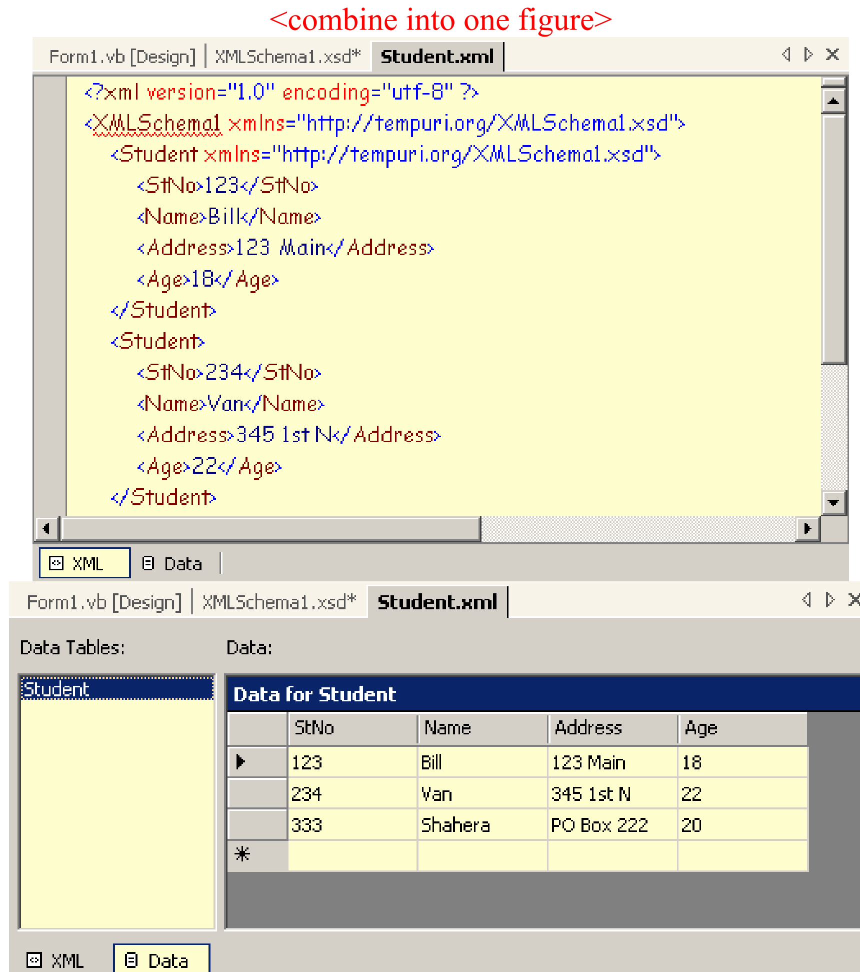


Figure 9.29 The XML file after entering some data

We have seen a simple example of creating an XML Schema and a related XML document. We now focus on some more complex examples. We assume that you understand the mechanics of creating the XML Schema and XML file so in the example we focus on the schemas and the type of relationships they support. The context for these examples is twofold. The first relates to creating XML Schemas to validate an XML document. Microsoft includes a validation command (Validate XML Data under the XML menu item) as well as methods to perform this validation at runtime. However, validating and XML documents against an XML Schema at runtime is too complex for this text but you are encouraged to investigate the capability. The second is using XML Schemas to facilitate the conversion of a relational database query to and from an XML document. In this latter context, Microsoft has added some additional elements (in their own namespace) to facilitate this translation between ADO.NET and XML. We will see some more on this in Example 9.3.

Exercise 9.6. Notice that in Figure 9.29, the second line of the XML document shows the element named XMLSchema1 with a squiggly line under it. First, what function does this element fulfill? Second, why has the XML designer placed the squiggly line under it?

Example 9.1 Using Simple and Complex Data Types in XML Schemas

In this example we create a schema for an Employee element. The schema will include a number of simple types as well as one explicit (named) complex type. A ***simple type* is one that has no elements or attributes**. We can associate facets with a simple type. A ***facet* allows you to define limits and boundaries on data values**. For example, we will use a facet to limit the number of characters in a state code to a maximum of 2.

Our schema will also include a named complex type. A ***complex type* is one that may include attributes and other elements**. We use the term "named" complex type because there are also unnamed complex types. A named complex type can be defined once and used many times while an unnamed complex type can only be one time in the definition of an element. The "Student" element we just looked at is an example of an unnamed complex type. The XML Schema Toolbox includes both the simpleType and complexType components and these support both named and unnamed types

We begin with a new Windows Application project and first add a new element selecting an XML Schema we named EmployeesSchema.xsd. Using the new XML Schema, we add two simpleType components. The first one is named StateCode of type string with a length facet (F) set to 2. We second is named ZipCode of type positive integer with a maxInclusive facet set to 99999. The XML Schema design should look like the image in Figure 9.30.

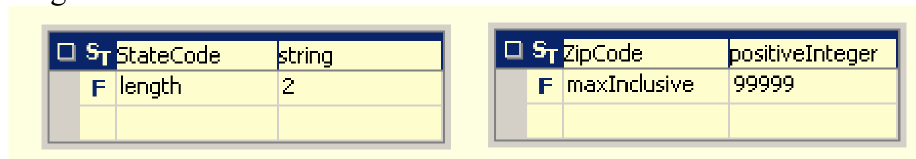


Figure 9.30 Two simpleType components in the XML Schema

Next we add a named complex type that will store the first and last name of an employee. A complexType component is dragged to the designer window and named "Name". Two string elements, LastName and FirstName are added to the component. When this is done, the XML Schema designer should look like that shown in Figure 9.31.

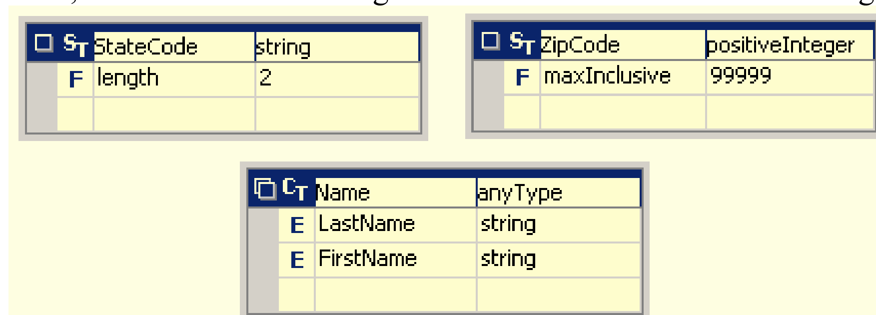


Figure 9.31 XML Schema design with the complexType Name added

The final step involves creating an unnamed complex type for the Employee element. To do this an element is dragged to the designer window and named Employee. To this we add a string element named EmpNo and then an element named EmpName. When we click on the type cell for EmpName, we choose the "Name" complex type

defined earlier. We continue by adding two string elements named Street and City, and then finish by adding a State and Zip element. For these final two elements, we again choose the simpleTypes StateCode and ZipCode we defined earlier. The final schema design is shown in Figure 9.32. Notice that since the EmpName is a complex type, the designer has added a specific instance of the Name type (named EmpName) and linked it to the Employee element. Be sure that the schema is saved before you start working on documents that will reference it.

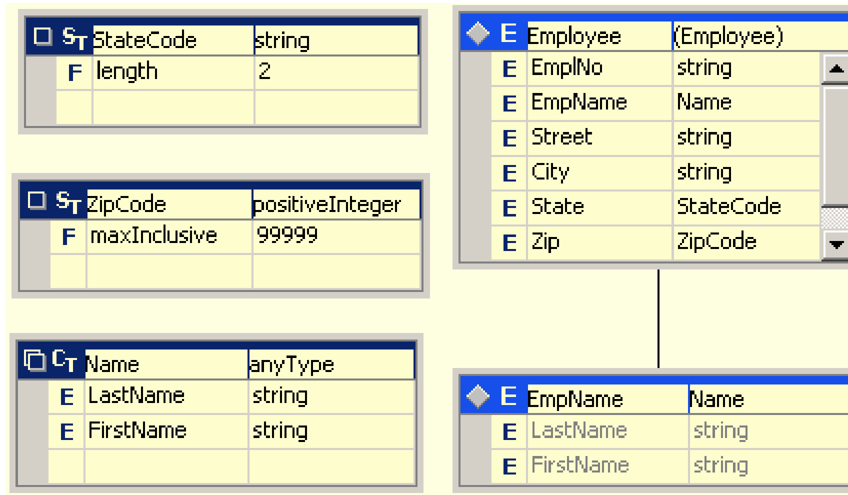


Figure 9.32 The final XML Schema design for the Employee element

Finally we need to add a new XML File item to our project. When you select Add New Item… from the File menu and select the XML File, name it something meaningful – we named ours EmployeeXML.xml. After adding the XML file, set its targetSchema property value equal to the schema just created. If you then click on the Data tab, you should see a table similar to the one in Figure 9.33. Here we have already added employee 1234 to the table.

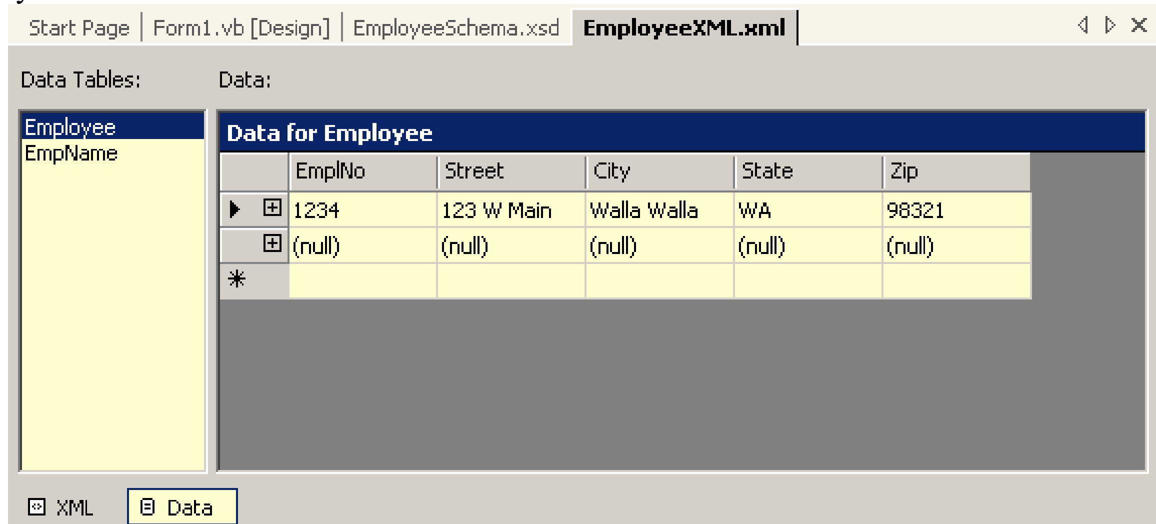


Figure 9.33 The XML "Data" view with one employee added

Clicking on the expand detail symbol (+) on the left of the row opens up an area to enter the name. This is similar to the master/detail record display we saw in Chapter 8 with the DataGrid control bound to a master/detail DataSet. Figure 9.34 shows this.

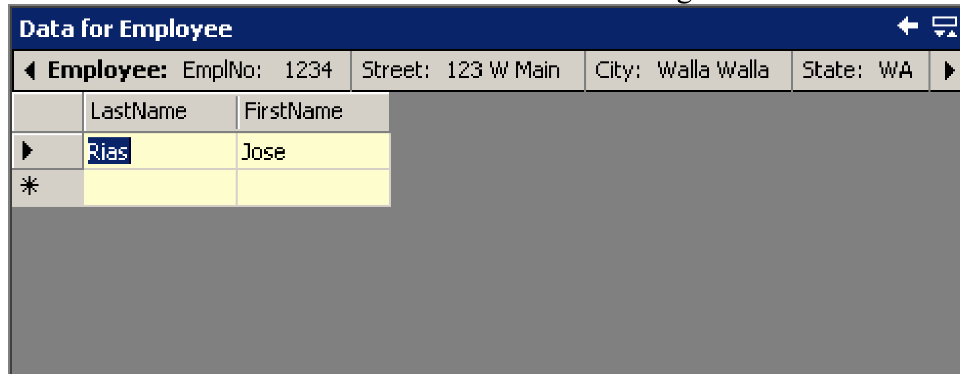


Figure 9.34 Expanding the EmpName table to add a name

In this example you have seen how to create an XML Schema by first creating simpleType and complexType components and then adding them to another element definition. This gives you the ability to create types of your own using the primary data types available within the XML Schema specification.

Exercise 9.7 Modify Example 9.1 so that it includes a named complexType named Address that consists of the City, State, and Zip. Use the StateCode and ZipCode simpleTypes in this new type. Add a new XML file and fill it some data according to the new definition in the XML Schema.

Example 9.2 Creating Nested Relationships in an XML Schema

In this example we create a Customer element and a Product element and establish a nested relationship with the Customer as the parent element and Product as the child element. This data design allows us to keep track of the products purchased by customers.

We begin with a new Visual Basic .NET windows application project and add an XML Schema. We define two elements, Customer and Product, as shown in Figure 9.35.

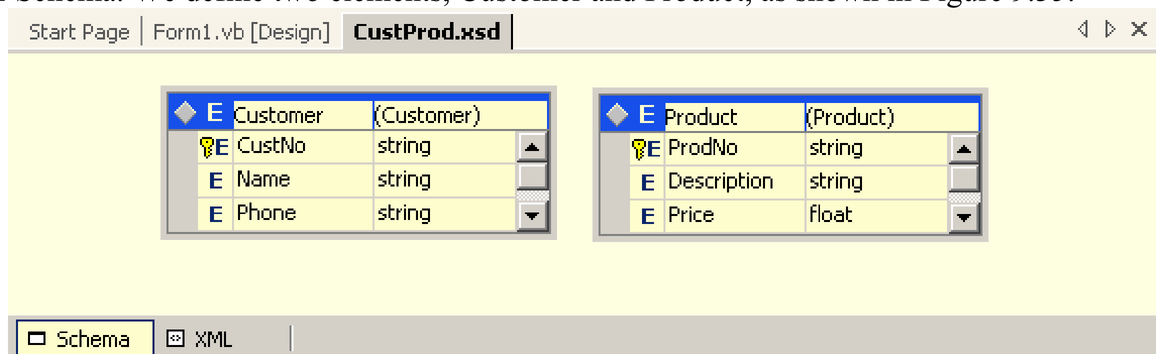


Figure 9.35 The initial two elements in the XML Schema design

We have defined key fields for each table (note the little key glyph). To do this, drag a "key" component from the XML Schema Toolbox and drop it on an element where you want to define the key. An Edit Key dialog box like the one in Figure 9.36 will be displayed. In this example we dragged the key component onto the Product element. We then named the key ProductKey (you may name it anything you want), verified that it

was being applied to the Product element and was using the ProdNo field. Finally we checked the "Dataset primary key" checkbox. This means that the values for the key must be unique. We performed similar tasks for the Customer element.

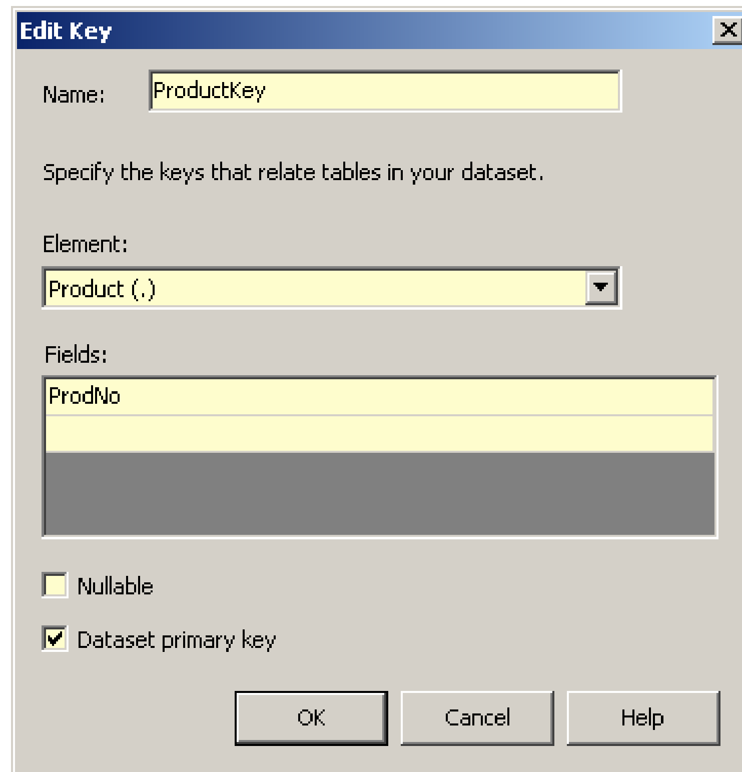


Figure 9.36 Creating a new primary key field for an element

Following this we needed to establish the parent/child relationship between the two elements. To do this we drag the child element (Product in our case) and drop it on the parent element (Customer in this example). This establishes a relationship such that for each parent (master) element, there can be many children elements (many Products). The XML Schema designer should now reflect this relationship with a line between the two elements as shown in Figure 9.37.

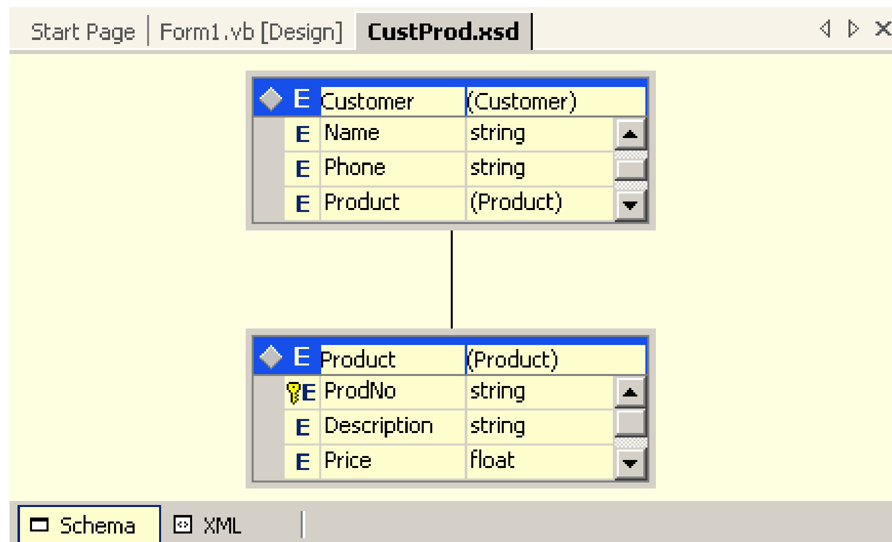


Figure 9.37 XML Schema Designer showing parent/child relationship between Customer (top element) and Product (bottom element)

After saving the schema, we can now add an XML File to our project and use its Data tab to fill the XML file with some data. This is show in Figure 9.38 in both the unexpanded and expanded views.

<combine into one figure>

Start Page | Form1.vb [Design] | CustProd.xsd | **CustProdXML.xml**

Data Tables: Customer, Product

Data:

Data for Customer

CustNo	Name	Phone
100	Peshtaz,Shah	555-1212
200	Tam,Allen	555-4323
300	Nedkov,Tiho	555-0000
400	Yam,Man	555-9898

XML | Data

Start Page | Form1.vb [Design] | CustProd.xsd | **CustProdXML.xml**

Data Tables: Customer, Product

Data:

Data for Customer

Customer: CustNo: 100 | Name: Peshtaz,Shahera | Phone: 555-1212

ProdNo	Description	Price
1000	17" flat panel	450
2000	512 mb ram	165
3000	Laser printer	450

XML | Data

Figure 9.38 Data entered into an XML file using the nested relationship between Customer and Product

---

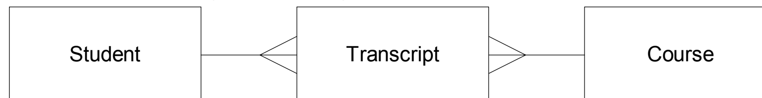
Exercise 9.8. Create a new project similar to Example 9.2 but create an XML Schema that has a Product as the parent element and Parts as the child element. Make up the elements that used to describe Product and Part. Include at least one element for Product and Part that use a facet that is appropriate given your design.

Example 9.3 Creating Tables with Relationships in an XML Schema

In this example we show how to create an XML Schema that includes ***DataRelation objects* that are used by the database management system to navigate between tables in the database**. We will build the XML Schema for the database that is described in the entity-relationship diagram shown in Figure 9.39

Figure 9.39 Entity-relationship diagram used as the basis for the XML Schema

We begin by creating a new Visual Basic .Net windows application and then add an XML Schema item to the project. Using the XML Schema Toolbox we create three



elements as shown in Figure 9.40.

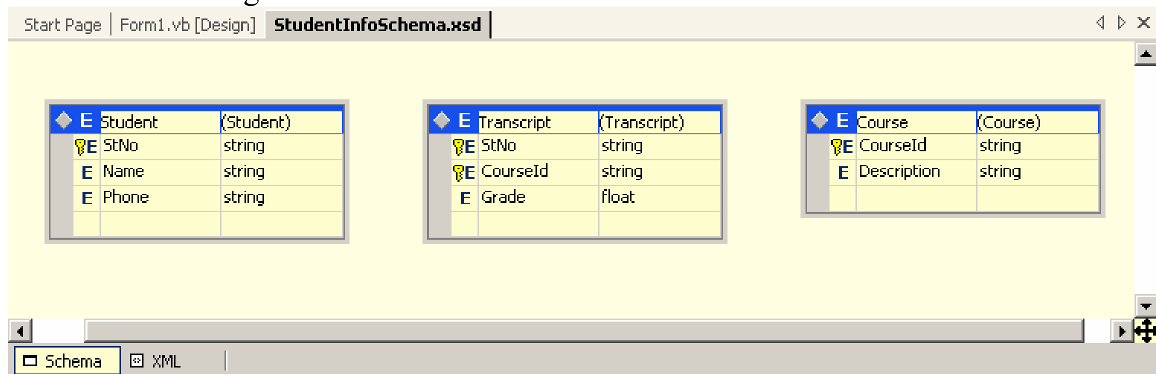


Figure 9.40 The three elements in the XML Schema

The only thing done in this schema that we have not seen before is establishing the compound key in the Transcript table. We do this by selecting two fields when we drag the key component to the Transcript table. Looking at Figure 9.41 you can see that we have added both StNo and CourseId fields to the key. When you click on a row in the Fields grid you will be able to use a dropdown arrow to select the field you want to include as part of the key.

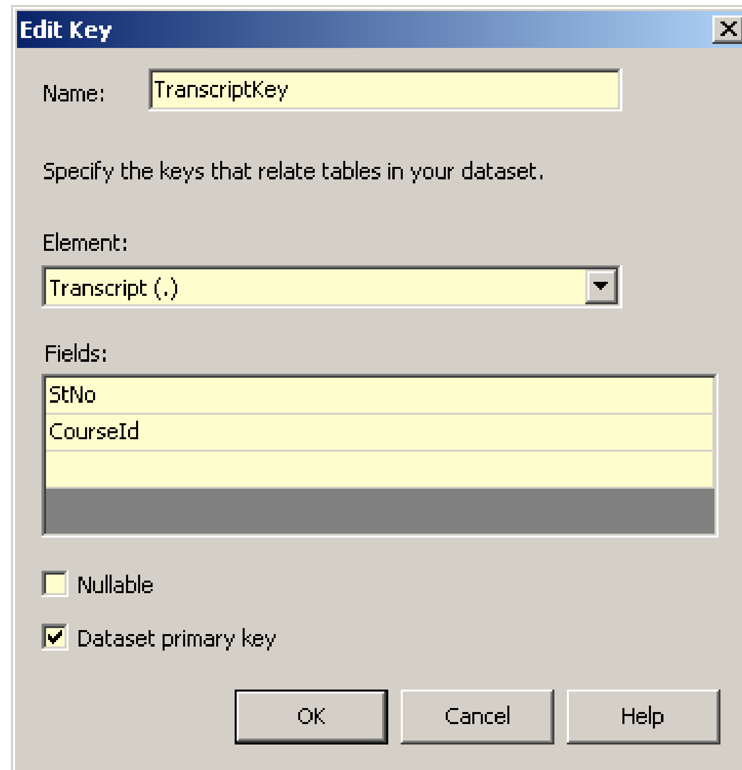


Figure 9.41 Selecting several fields for the key

Now that we have the elements defined we need to define the relationships between the elements. Dragging the "Relation" component from the XML Schema Toolbox and dropping it on the ***child*** element does this. In our case we need to first drag a Relation component to the Transcript element (the child element for both Student and Course). You should see an Edit Relation dialog box like that in Figure 9.42. Be sure that the correct parent and child elements are selected as well as the correct key field. We have also chosen the "Cascade" option for the Update and Delete rules. These mean that if you delete a Student, that student's transcript records will also be deleted under the assumption that a transcript for a student not in the database is not desirable. Of course, one might argue against this assumption but that is what we have chosen here.

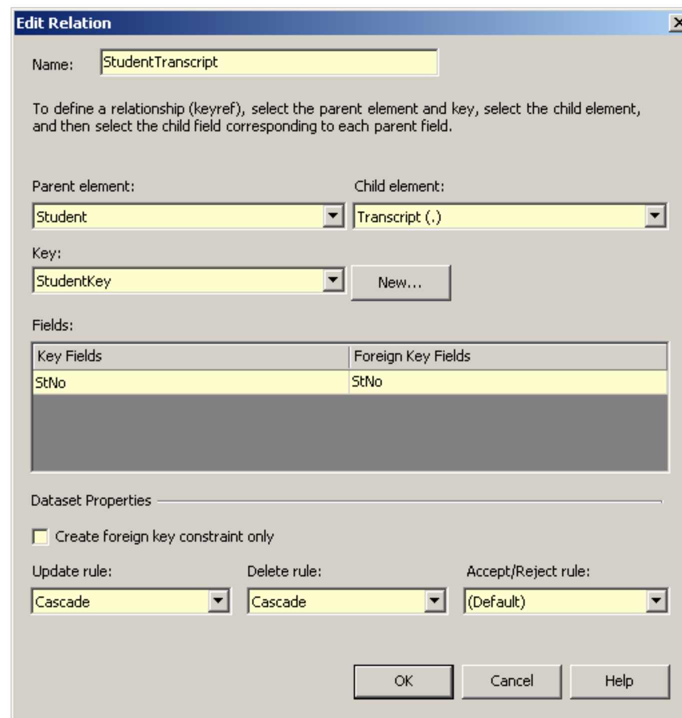


Figure 9.42 Creating the relation between the Student and Transcript elements

We also need to add another relation with the Course element (parent) and Transcript element (child). You will likely have to change some of the default values for this relation because the defaults may still be set up for the Student-Transcript relation. When you finish you should see a design like that shown in Figure 9.43.

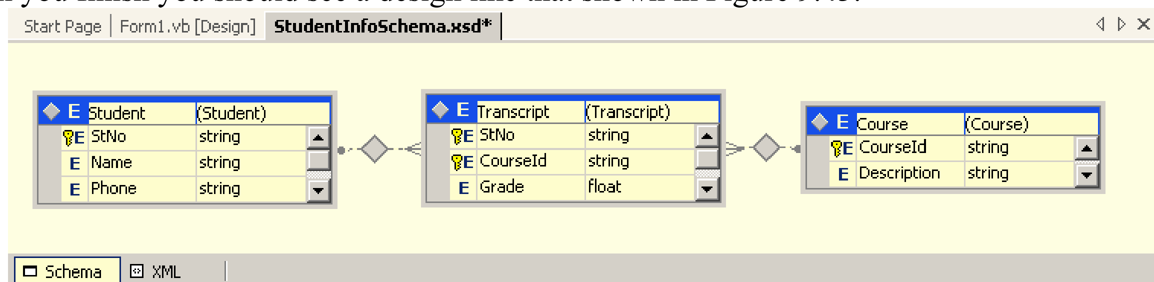


Figure 9.43 The final XML Schema design for Example 9.3

As a final note let's look at a little of the actual XML that was created by Visual Basic .NET and relate this to the namespace concept discussed earlier. The XML statement:

```
<xs:key name="StudentKey" msdata:PrimaryKey="true">
```

shows two namespace prefixes, `xs` and `msdata`. If we look at the namespace declarations in the code we see:

```
xmlns:xs=http://www.w3.org/2001/XMLSchema
xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
```

The first namespace, `xmlns:xs=http://www.w3.org/2001/XMLSchema`, includes the schema definitions based on the W3C specification, thus any processor that complies with this specification can validate any XML document against any giving schema created using this namespace. The second namespace, `xmlns:msdata="urn:schemas-`

microsoft-com:xml-msdata", is defined by Microsoft and is used to add attributes and elements (such as "PrimaryKey" attribute in the example above) that provide ADO.NET the ability to bridge between relational database and XML data. Understand, however, that these extensions are not part of the w3c standard so one might find that non-Microsoft XML parsers ignore them.

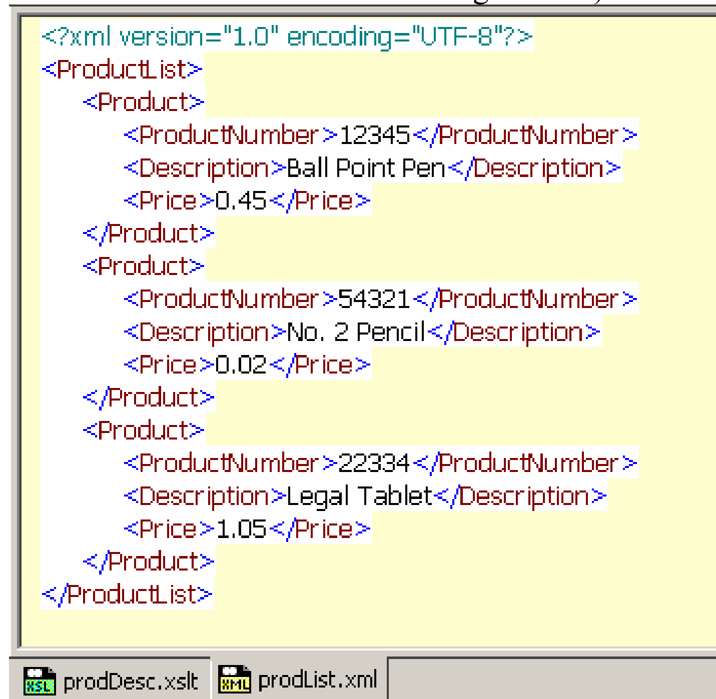
Exercise 9.9. Create an XML Schema for the following Entity-Relationship Diagram:

<insert IG0901>

Example 9.4 Transforming XML using XSLT

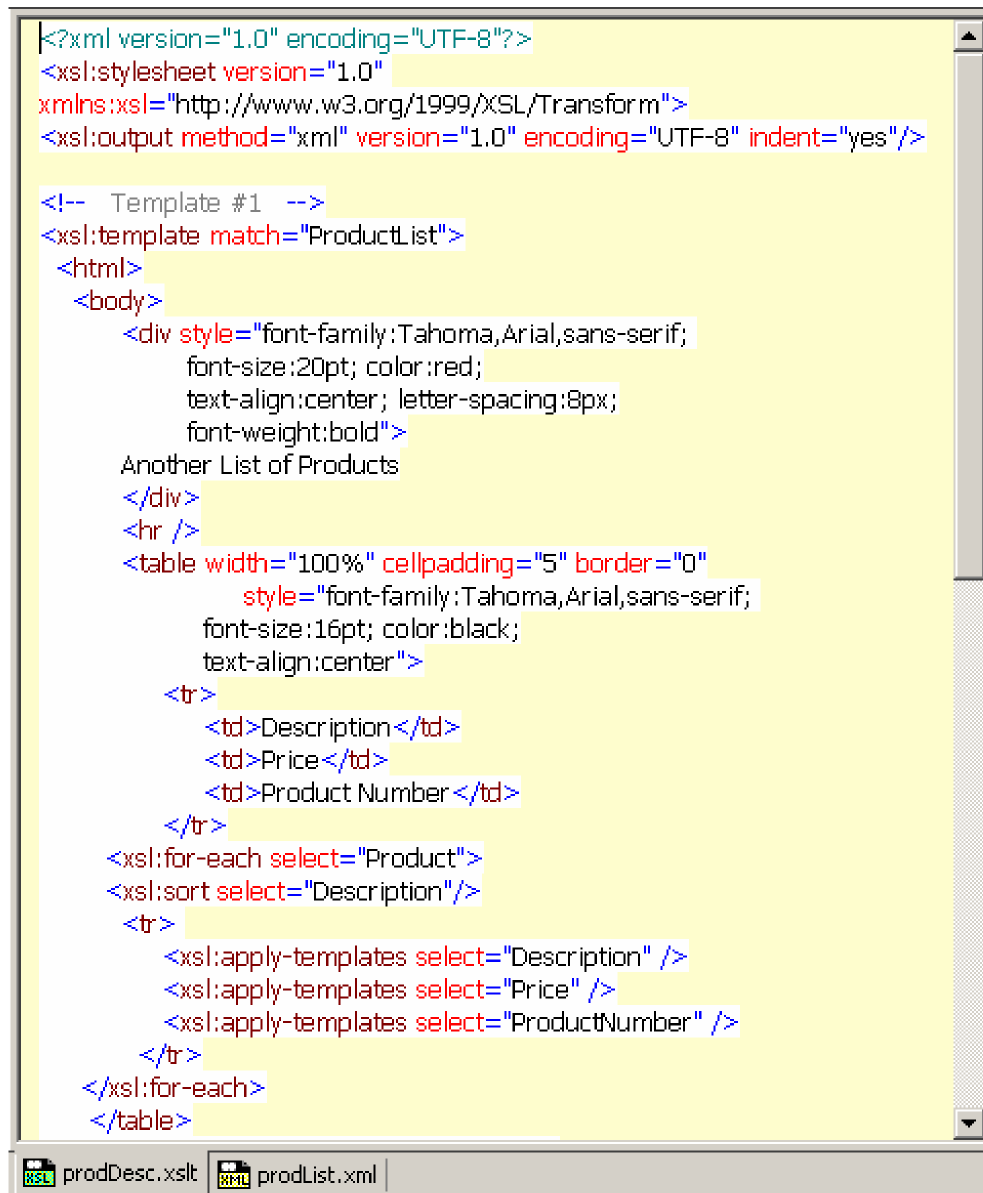
This example demonstrates how you can transform an XML file into a new XML file using an XSLT specification. We will use the XML and XSLT files we saw earlier in this section and generate a new file in HTML format. You can then view the new file using a browser to confirm that the transformation worked.

Figure 9.44 shows the original XML file and Figure 9.45 shows the first part of the XSLT file (the entire XSLT file was shown earlier in Figure 9.14).



```
<?xml version="1.0" encoding="UTF-8"?>
<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 9.44 The original XML 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>
```


Figure 9.45 The first part of the XSTL file

A Visual Basic .NET windows application has been created with a single form. The form has only one Button control that holds the code to perform the transform. To make things easier, we have placed the two files prodList.xml and prodDesc.xslt in the “bin” folder of the project. This allows us to write code to access the files without having to include the entire file path (by default, Visual Basic .NET looks in its “bin” folder for any files referenced in the executing code.)

The code for this example is shown in Figure 9.46. Notice that we have added an Imports statement to the start of the code. This statement is necessary to get access to the namespace (set of classes) that support the XSL behavior. All Imports statements must be at the very beginning of the code. If you left this Imports statement out, the Dim statement in the click event would not be valid (the reference to the XslTransform class would not be valid).

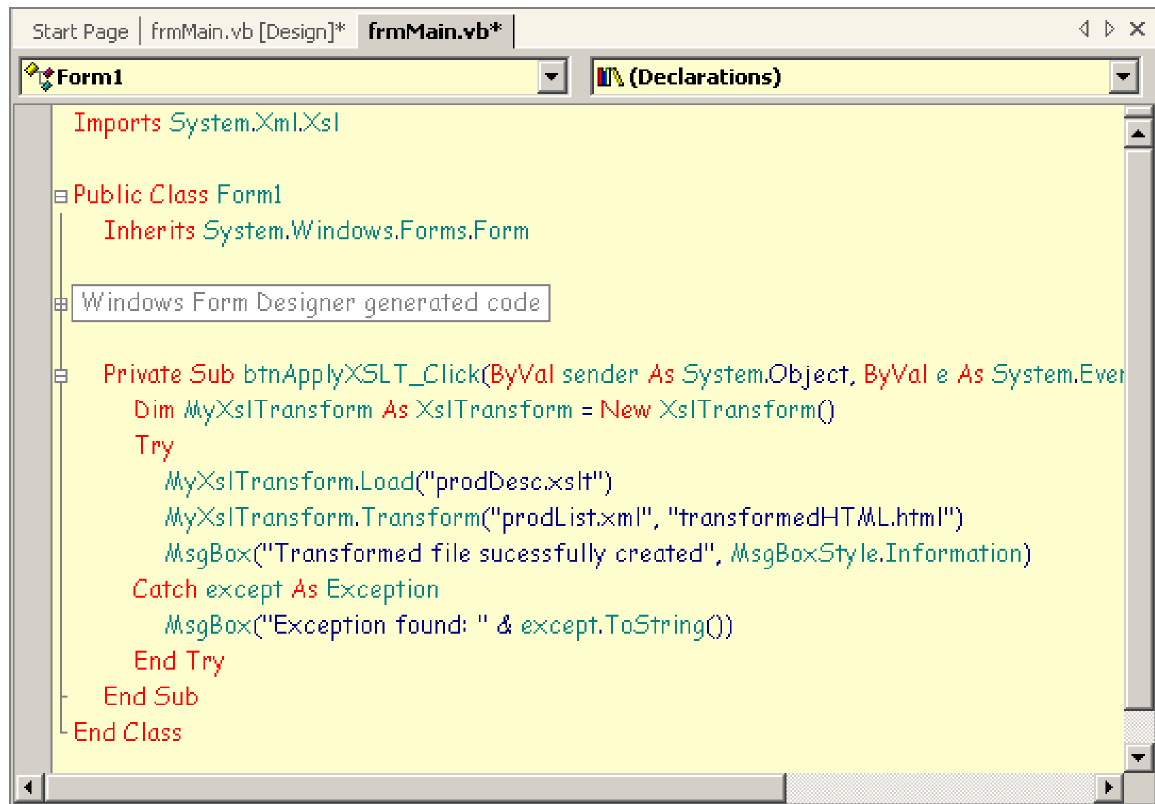


Figure 9.46 Code for Example 9.4

The actual transformation process is really very straightforward due to the power of the XSL methods available in Visual Basic .NET. We first declare an object reference MyXslTransform:

```
Dim MyXslTransform As XslTransform = New XslTransform()
```

We then use the Load and Transform methods available to XslTransform objects to perform the actual transformation.

```
MyXslTransform.Load("prodDesc.xslt")
MyXslTransform.Transform("prodList.xml", "transformedHTML.html")
```

The Load() method loads the XSLT code (see Figure 9.45) from the file named prodDesc.xslt located in the project's bin directory. The Transform() method then uses the XSLT code to transform the XML file (prodList.xml) into a new file (transformedHTML.html). If you look at the transformed file (it will be stored in the project's bin directory too) using a browser you should see what is shown in Figure 9.47.

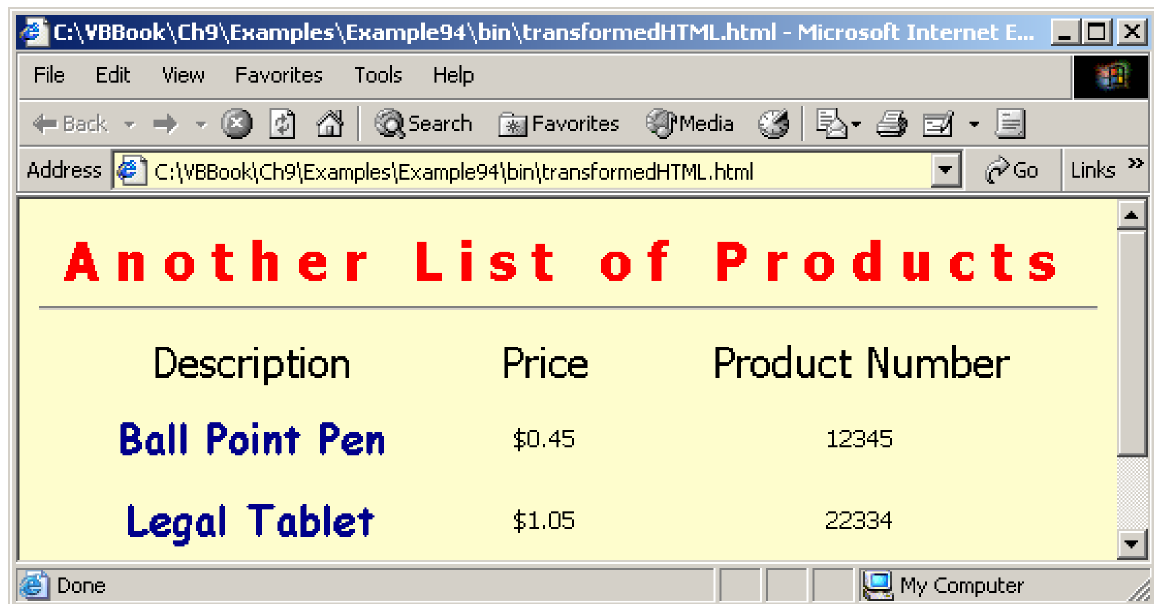


Figure 9.47 The HTML file transformed from the XML file using XSLT

Exercise 9.10. Modify Example 9.4 so that the HTML document shows details for each product with the product number first, followed by the description and price.
Exercise 9.11. Create an XML document for a Student element that consists of a Student Number, Name, and Phone Number. Then modify the XSLT in Figure 9.45 so that is processes your XML document and shows, for each Student, the three elements in the order defined (Student Number, Name, and Phone Number).

9.3 USING ADO.NET WITH XML

ADO.NET provides a number of useful classes and methods to work directly with XML documents. This should not be a surprise since a DataSet component uses XML as its internal data representation. Thus, data are already in XML form and all we are doing with the various XML-related ADO.NET classes is getting access to the XML that has already been created.

In this section we provide a number of examples that demonstrate some of the XML capabilities of the ADO.NET framework. We begin with an example that reads XML data directly instead of reading data from a relational database. This would be useful if a company kept its data in raw XML form or if they needed to process some XML that was sent to them by one of their business partners. We then look at an example where we read data from a relational database and convert it into equivalent XML. We finish with an example that that reads data from a relational database, converts it into XML, and then uses an XSLT document to reformat the data.

For these examples we assume that you have read Chapter 8 and are familiar with ADO.NET and its associated components such as a connection, data adapter and dataset.

Example 9.5 Processing an XML Document

In this example we see how you can create a DataSet from an XML document. You can then use the DataSet to display its contents in a DataGrid control. We also see how you can generate an XML Schema from within a running application once the XML

document is processed. Figure 9.48 shows the example as it is running and Figure 9.49 shows the XML Schema.

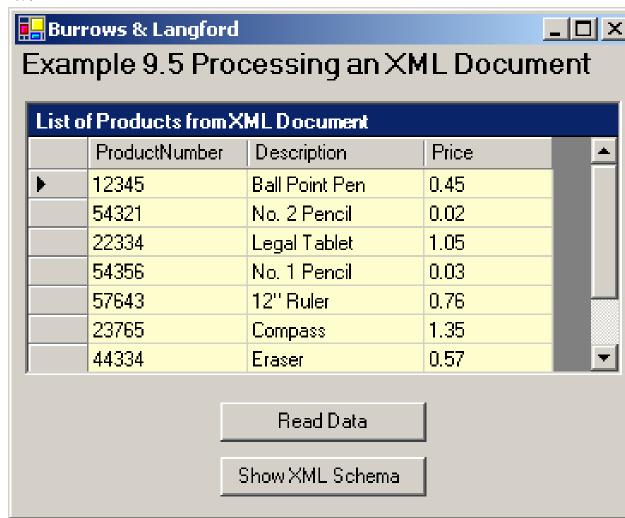


Figure 9.48 Example 9.5 at runtime

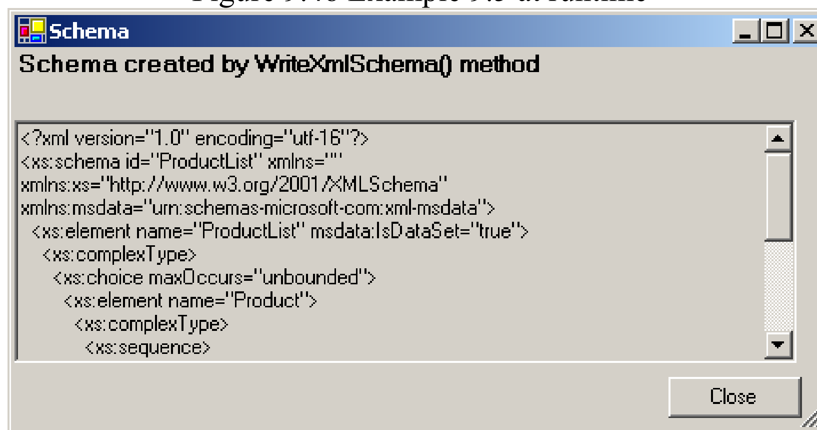


Figure 9.49 Example 9.5 with the XML Schema displayed

The XML document used in this example is an expanded version of the product list XML document we have seen earlier. Figure 9.50 shows a portion of this document.

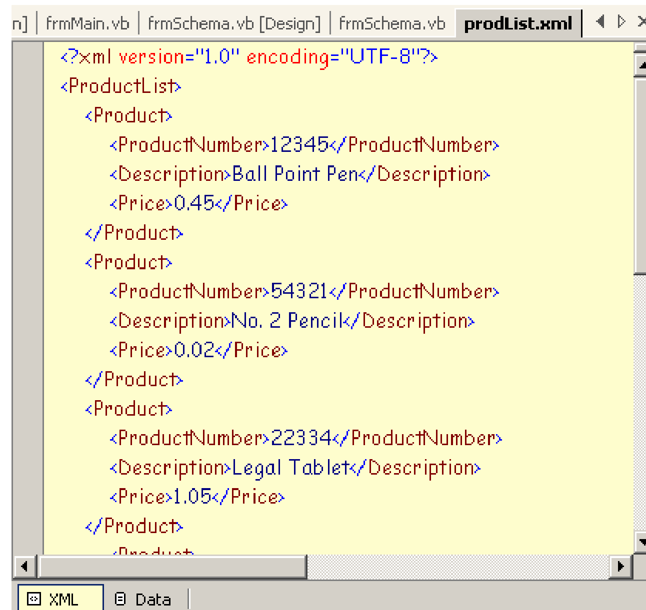


Figure 9.50 A portion of the XML document used in Example 9.5

In this example we create a DataSet object directly in the code. This is because we are not using a database connection or data adapter like we did in Chapter 8. Objects instantiated from the DataSet class have a method called ReadXML(). This method reads an XML document directly from a file and uses the contents of the document to fill the DataSet. Once the DataSet is created and filled with data, it is used to populate a DataGrid component. Code to perform these tasks is shown in Figure 9.51.

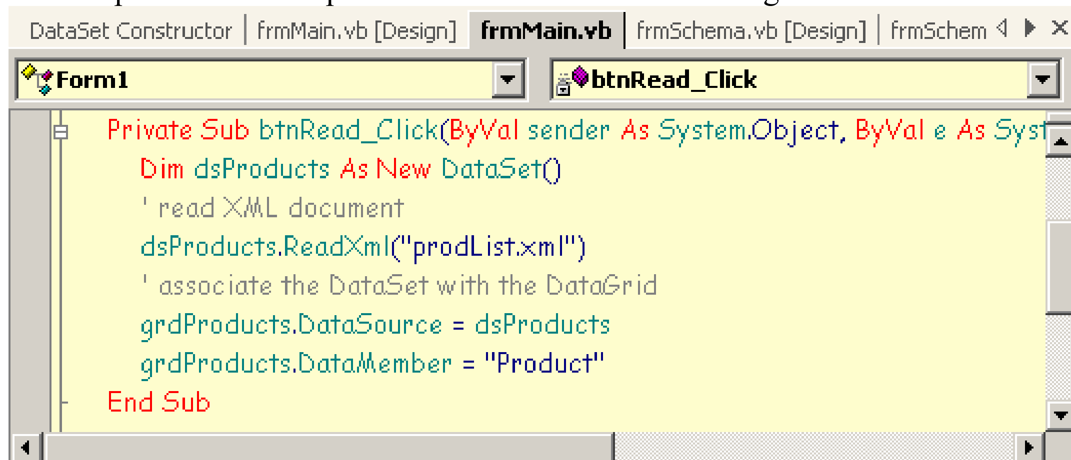


Figure 9.51 Code to process the XML document and then fill a DataGrid

The Dim statement creates the DataSet object reference named `dsProducts`. This object reference then uses its ReadXml() method to read the XML document from a file named prodList.xml. This XML document must be in the project's "bin" directory for it to be found. Following this, DataSource property value is set equal to the DataSet `dsProduct`. Finally, the grid's DataMember property value is set equal to "Product" because the XML document uses <Product> … </Product> as the main tag to identify each row of the DataSet.

To show the XML Schema, we create a second form and display it with the code shown in Figure 9.52.

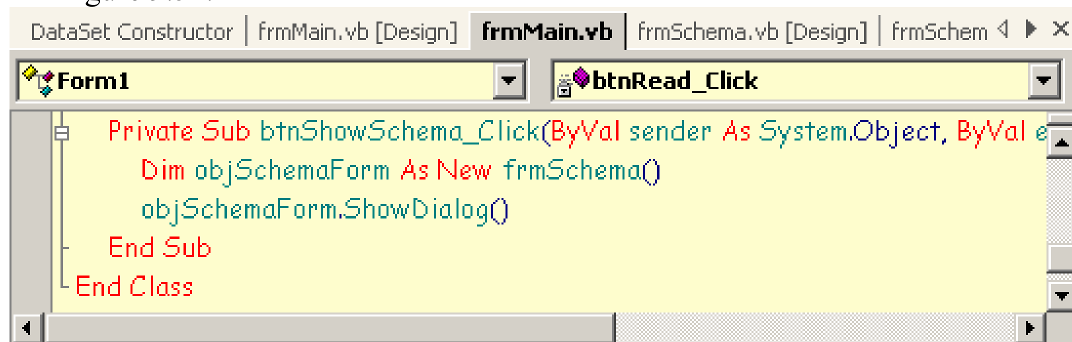


Figure 9.52 Form and code to display the form that will show the XML Schema

This form includes one TextBox component with its Multiline and ReadOnly property values set to True. It creates and uses a DataSet object to read the XML document just like we saw before. It also uses the DataSet's WriteXmlSchema() method to develop and show an XML Schema that describes the XML document. The code also uses a StringWriter object that is initially used to store the XML Schema generated by the DataSet's WriteXmlSchema() method. The StringWriter object is then turned into a String using its ToString() method to populate the TextBox component. The code for this part of the example is shown in Figure 9.53.

```
Private Sub btnClose_Click(ByVal sender As System.Object, ByVal e As Sys
    Me.Hide()
End Sub

Private Sub frmSchema_Load(ByVal sender As System.Object, ByVal e As S
    Dim dsProducts As New DataSet()
    Dim XmlStringWriter As New System.IO.StringWriter()
    dsProducts.ReadXml("prodList.xml")
    dsProducts.WriteXmlSchema(XmlStringWriter)
    txtSchema.Text = XmlStringWriter.ToString
    ' move the selection so no text will not be selected
    txtSchema.SelectionStart = 0
    txtSchema.SelectionLength = 0
End Sub
```

Figure 9.53 Code to display XML Schema in TextBox component

Note that after the Text property of the TextBox is set, the TextBox's SelectionStart and SelectionLength property values are set. If this were not done, then all the text in the TextBox would be selected and we want to avoid this behavior because it would look strange to the end user.

Exercise 9.12. Modify Example 9.5 so that it uses the XML document defined in Exercise 9.11.

Example 9.6 Converting Relational DB Data into XML

The objective of this example is to demonstrate how you can read data from a conventional relation database and after you get the results, you can transform the results into an XML document. For example, this might be useful if you store your data in a conventional database but need to send some information to a business partner using XML. Another example might be the need to place the data on a remote web server that does not have the database software installed on it.

In this example we take data from the "pubs" database and recreate Example 8.5. In addition, we add the code to generate and display the XML document for the results of the query used to create the DataSet displayed in a DataGrid. We also add code to generate and display the XML Schema for the data in the DataSet like we did in Example 9.5. Figure 9.54 shows the application at runtime. Note that Example 8.5 demonstrated how to create Master/Detail DataSet. The "+" expander symbols can be used to see the employees of a specific publisher.

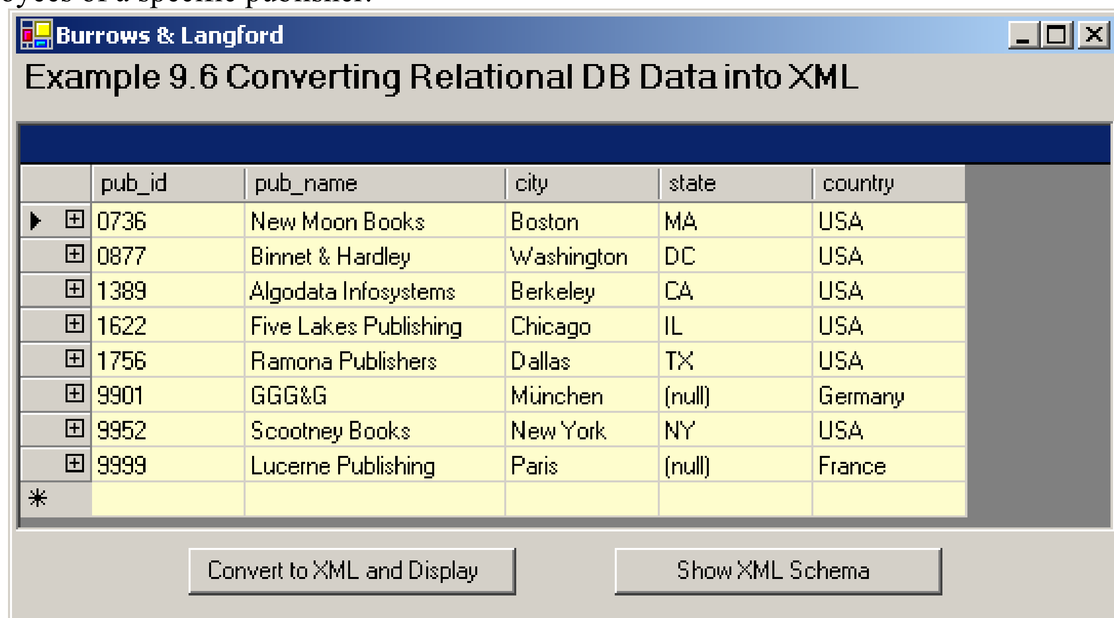


Figure 9.54 Example 9.6 at runtime

Clicking on the "Convert to XML and Display" button shows a form like that in Figure 9.55 and clicking on the "Show XML Schema" button shows another form like that in Figure 9.56.

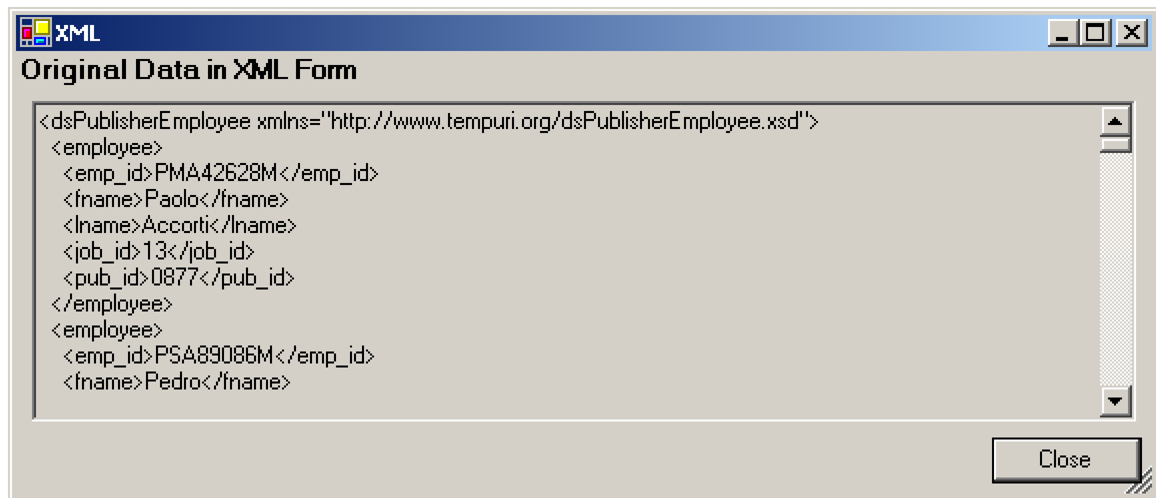


Figure 9.55 The XML document for the DataSet shown in Figure 9.54

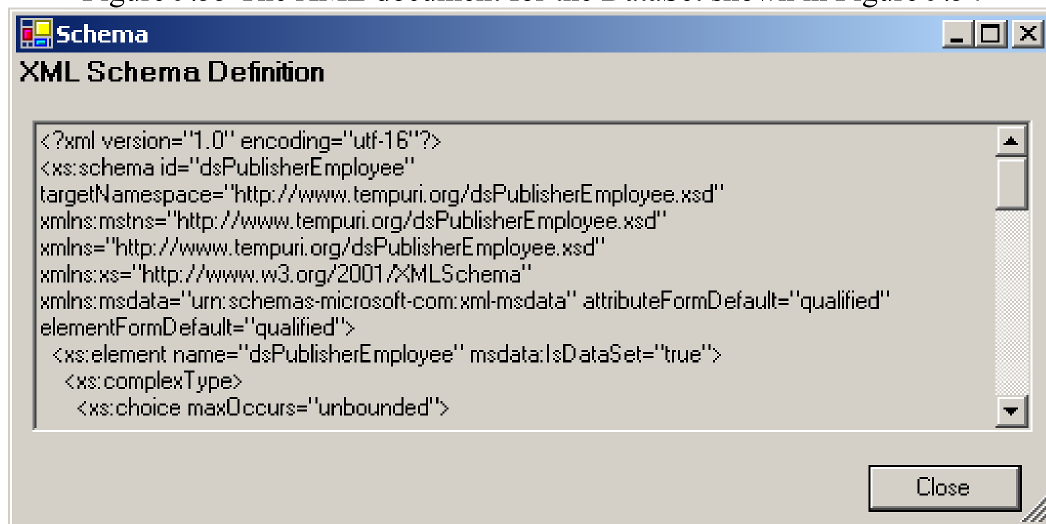


Figure 9.56 The XML Schema for the XML shown in Figure 9.55

We start by creating the project described in Example 8.5. Using that as the starting point, we add two new buttons to the main form, one to display the XML document in a new form and another to display the XML Schema in another new form. The code for displaying the XML is shown in Figure 9.57.

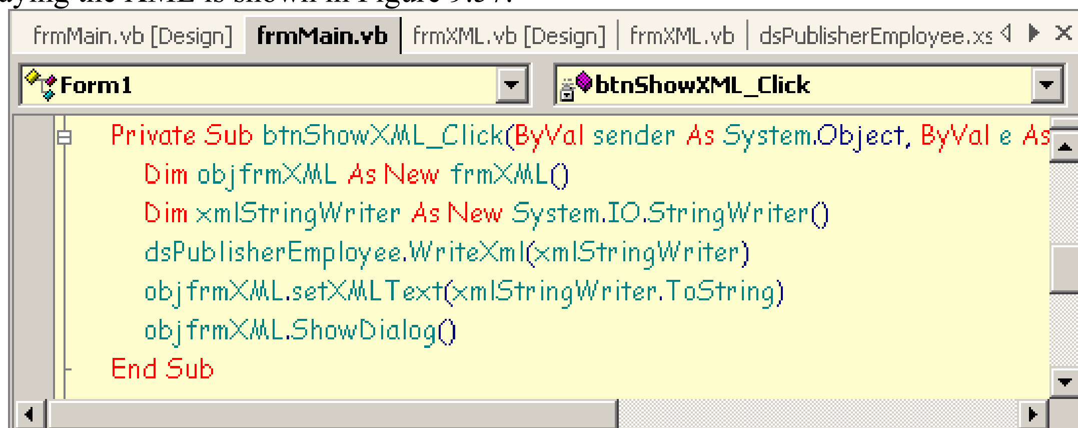


Figure 9.57 Code to display the XML document in a new form

In the code in Figure 9.57, the DataSet, dsPublisherEmployee, was created using the instructions given in Example 8.5 (Figures 8.58 and 8.62). The DataSet represents the data as shown in Figure 9.58.

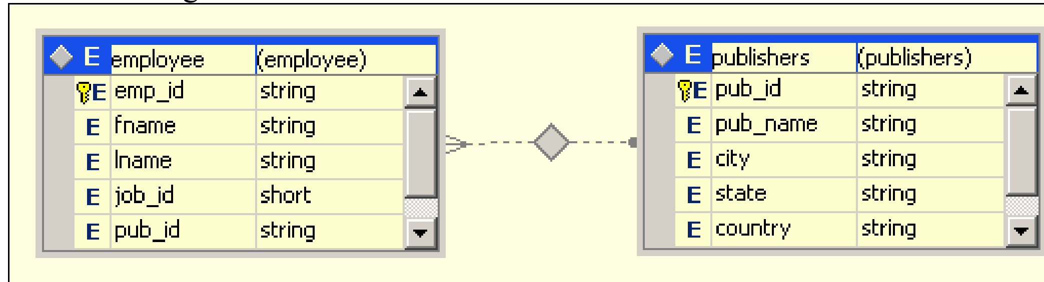


Figure 9.58 The data design used to create the data stored in dsPublisherEmployee

In addition to having a WriteXmlSchema() method, the DataSet class also has a WriteXml() method that is used in the code in Figure 9.57. This method writes the XML document to a StringWriter like was done in Example 9.5. The new form, an instance of frmXML, contains a TextBox and Close button. We call the new form's setXMLText() method and pass the String equivalent of the StringWriter object. Figure 9.59 shows the code for frmXML's setXMLText() method.

```
sign] | frmMain.vb | frmXML.vb [Design] | frmXML.vb | dsPublisherEmployee.xsd | frmSchema.vt
frmXML                                   setXMLText

Public Sub setXMLText(ByVal XmlText As String)
    txtXML.Text = XmlText
End Sub
```

Figure 9.59 Code for the method setXMLText() on the form frmXML

The code used to create the XML Schema and display it on another form is shown in Figure 9.60. We create the schema like we did in Example 9.5 and we fill the new form's TextBox like we just filled the XML form's TextBox.

```
sign]* | frmMain.vb* | frmXML.vb [Design] | frmXML.vb | dsPublisherEmployee.xsd | frmScheme
Form1                                    btnShowSchema_Click

Private Sub btnShowSchema_Click(ByVal sender As System.Object, ByVal e
    Dim objfrmSchema As New frmSchema()
    Dim xmlStringWriter As New System.IO.StringWriter()
    dsPublisherEmployee.WriteXmlSchema(xmlStringWriter)
    objfrmSchema.setSchemaText(xmlStringWriter.ToString)
    objfrmSchema.ShowDialog()
End Sub
```

Figure 9.60 Code to create the XMLSchema and display it on a new form

Exercise 9.13. Using Microsoft Access, create the database described in Exercise 9.9. After population this database with a few sample records, modify Example 9.6 so that it works with the new database.

Exercise 9.14. Modify Example 9.6 so that it uses the authors, titles, and titleauthor tables in the pubs database.

Example 9.7 Using XML and XSLT to Transform Data from a Relational Database[5]

This example brings everything we have seen in this chapter together and also adds some additional capabilities we have not seen in the past. We start with a hypothetical database that contains email messages. We read the records from this database and convert them into an XML document. We then allow the user to specify whether to see just the email "headers" or the entire email document. We use two different XSLT documents to convert the XML into HTML depending on their choice of display format. Finally we use a component on the Windows Form that renders and displays the HTML for the user. Runtime images of the example are shown in Figure 9.61 (email headers only) and Figure 9.62 (full view).

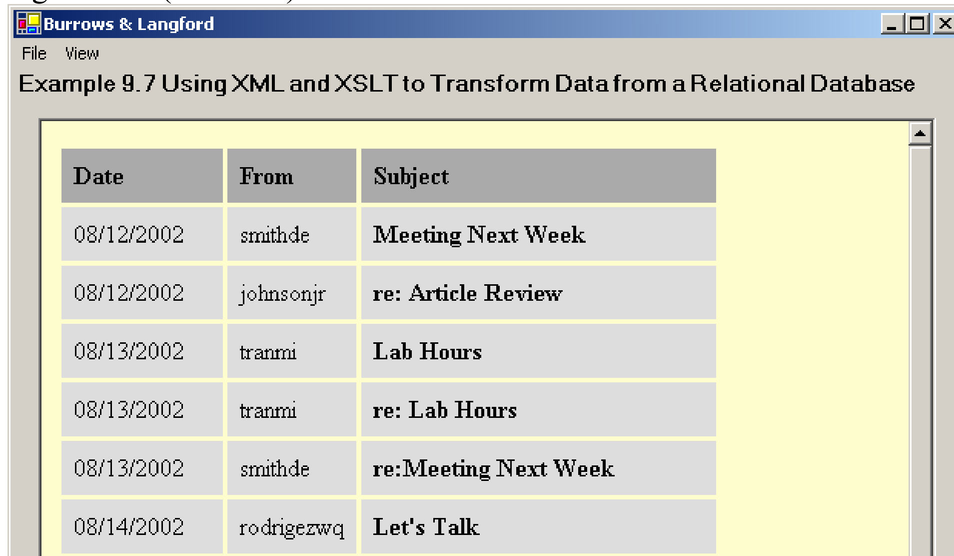


Figure 9.61 Example 9.7 showing the email headers

[5] The example is based on a "Walkthrough" titled "Displaying an XML Document in a Web Forms Page using Transformations" that is provided with Microsoft Visual Studio .NET.

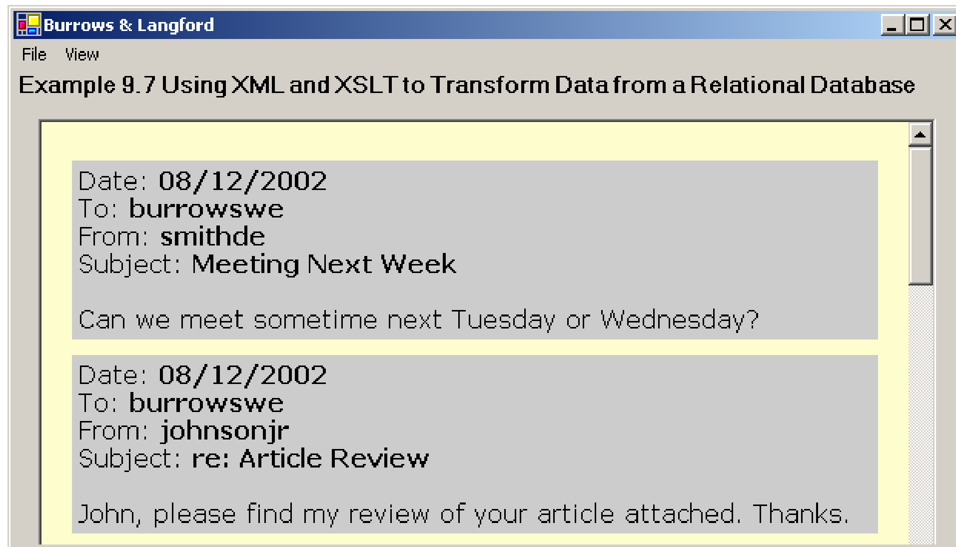


Figure 9.62 Example 9.7 showing the complete email message

As you can see in Figure 9.62, there is a View menu item that includes two subitems “Headers Only” and “Full View”. Recall that Section 6.8 covered the details of using the MainMenu control.

When we prepare examples we try to choose situations that are realistic but do not present any unexpected “problems” as far as coding the solution is concerned. While creating this example, we found a little “problem”. This involved the XML created by the DataSet’s WriteXML method. The problem meant that the XSLT that we were using to render the XML document into HTML would not work. Instead of throwing the example out or changing it to eliminate the problem we chose to program a “fix”[6]. Our rationale was very simple – in real life programming it is not unusual to find little problems so we think it is important to show an example of this and how we got around it.

The database is a Microsoft Access database named email.mdb consisting of a single table. Figure 9.63 shows the table and its field definitions.

emailList : Table

Field Name	Data Type
MessageId	Text
Date	Text
To	Text
From	Text
Subject	Text
Body	Text

Figure 9.63 The emailList table in the database

[6] We actually had two choices for our “fix”. One would involve modifying the XSLT to take into account the namespace defined in the XML document. While this would be a relatively easy fix, we chose to use the programming approach because it allows the introduction of some file-processing code and a short example of some useful string manipulation.

This database is added to the Server Explorer (see Example 8.6) and a connection is established. We then create an OleDbDataAdapter named odaEmail and a DataSet named dsEmail. This gets us ready to start the project code.

The coding begins with a form load event. This event's primary function is to process the database and convert it into an XML document. However, this is where we encountered our little "problem". The WriteXml() method of the DataSet class adds a namespace specification to the opening of the XML document (see the first line in the XML document shown in Example 9.6's Figure 9.55). There is nothing wrong with doing this except for the fact that when we later transform the XML into HTML, the XSLT code fails because of this namespace specification. So we need to write some code to remove it.

Instead of writing the XML to a StringWriter like we have done in the past, we need to write it to a file on the disk. To do this we'll see some new code that deals with reading and writing files on the disk without the aid of a database management system. Given this background, the overall logic of the form load event is:

1. Fill the dataset using the data adapter.
2. Create a temporary disk file to store the XML.
3. Write the XML from the dataset to the temporary disk file.
4. Copy each record from the temporary disk file to another disk file to be used later. This new disk file will contain the XML from the temporary file but will not include the namespace specification that caused the problem.
5. Close all the files and delete the temporary disk file.

Step 4 is the step that "solves" our problem. The code for the form load event that covers steps 1-3 is shown in Figure 9.64.

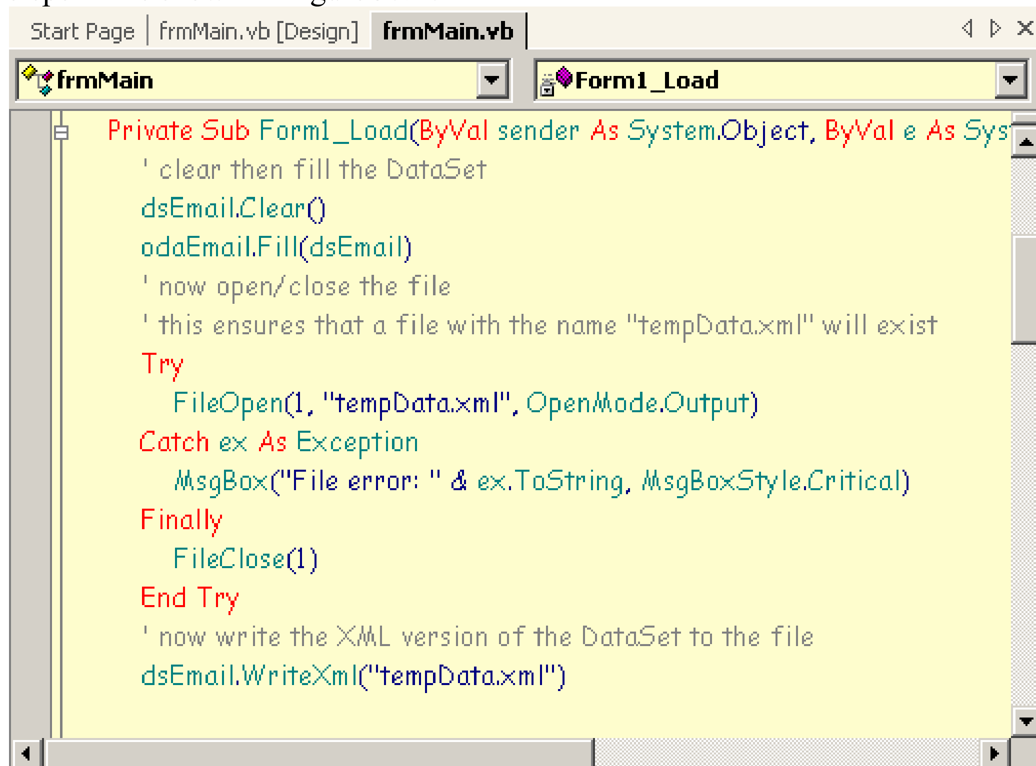


Figure 9.64 Code for step 1-3 of the form load event for Example 9.7

The only part of the code in Figure 9.64 that we have not seen before relates to the temporary file – the FileOpen() and FileClose() methods. The FileOpen with the OpenMode.Output parameter creates a new file if one does not exist. If one exists, it just "opens" it meaning that we can write data into the file if we want to. We are ignoring the situation where the file already exists; we are only taking advantage of the fact that a new file will be created if it does not already exist. The file will be created in the project's bin directory by default. Files are identified by number inside the code so we are calling this file #1 in the open and using "1" in the close. When the code in Figure 9.64 is done, the XML document (with the problem namespace specification) is in the file named tempData.xml stored in the project's bin directory.

Now we need to fix the problem. The code in Figure 9.65 (a continuation of the form load event) shows how we performed the fix.

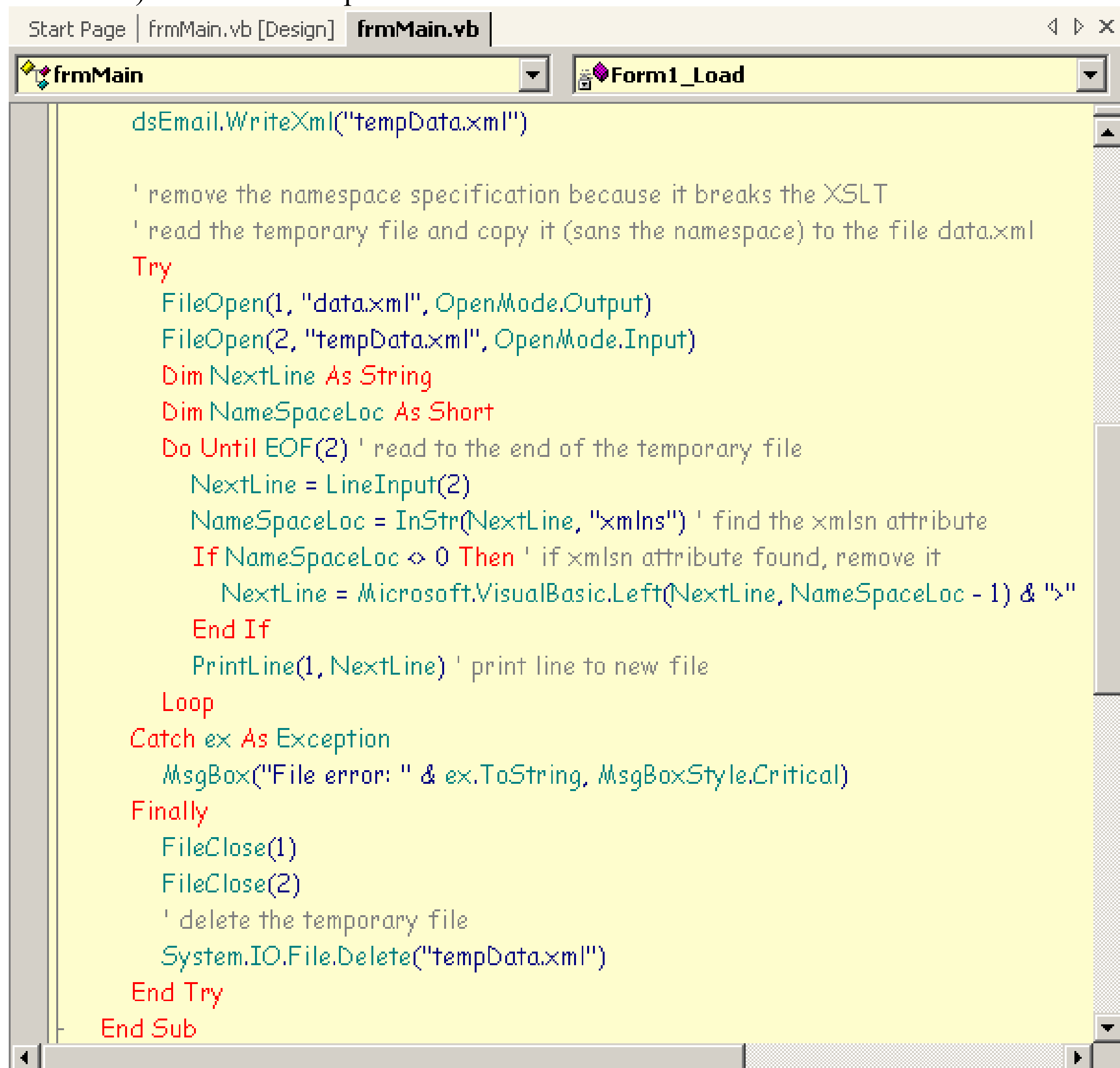


Figure 9.65 Continuation of the code for Example 9.7's form load event

This code begins by opening two files – file #1 is a new file named data.xml that will store the corrected XML and file #2 is our temporary file that contains the original problematic XML. It then enters a loop until the function EOF(2) becomes True. EOF stands for End of File so we are going to process the records in file #2 until we come to the end of that file. Inside the loop we first get the next line (record) from the temporary file (#2). We then search this line for a namespace specification that looks like:

```
<dsEmail xmlns="http://www.tempuri.org/dsEmail.xsd">
```

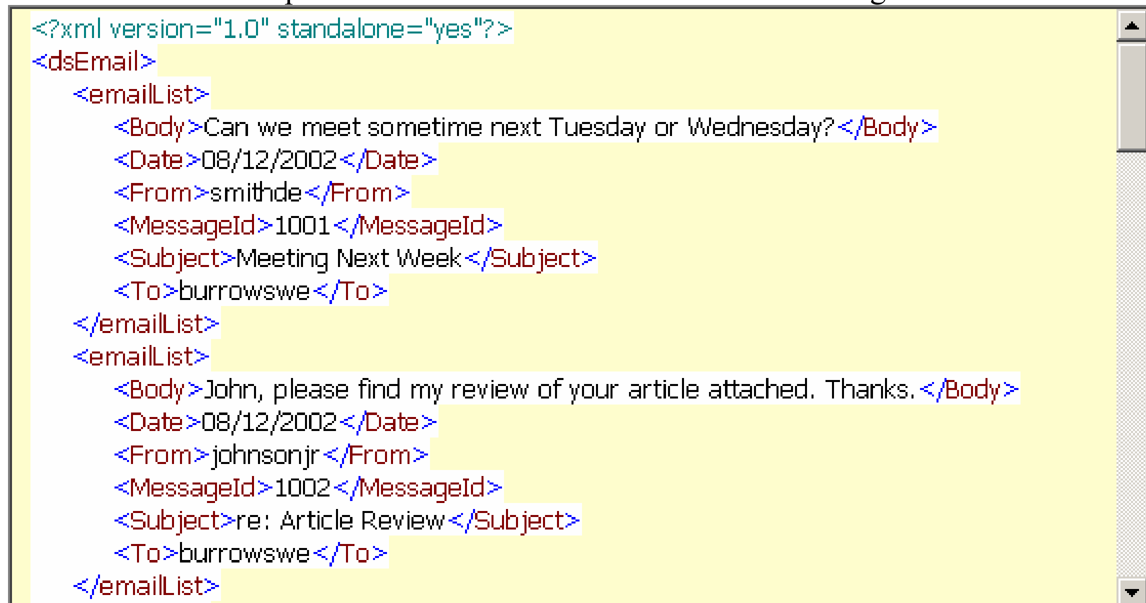
If we find the String "xmlns", we need to remove everything up to the closing ">" symbol. In the code above, "xmlns" is found starting in position 9 of the string. So we replace the NextLine string with the original 8 characters on the left followed by a ">" character. The result looks like:

```
<dsEmail>
```

Regardless if we found the bad line or not, we then use the PrintLine statement to write the string NextLine to file #1 (our correct XML document file).

When the loop terminates, all the lines of the temporary file have been transferred to the new file (including the corrected line). We then close both files and delete the temporary file (tempData.xml) using the Delete() function in the System.IO.File namespace.

We now have the data from the database successfully corrected and transferred to the file named data.xml. A portion of that XML document is shown in Figure 9.66.



```
<?xml version="1.0" standalone="yes"?>
<dsEmail>
    <emailList>
        <Body>Can we meet sometime next Tuesday or Wednesday?</Body>
        <Date>08/12/2002</Date>
        <From>smithde</From>
        <MessageId>1001</MessageId>
        <Subject>Meeting Next Week</Subject>
        <To>burrowswe</To>
    </emailList>
    <emailList>
        <Body>John, please find my review of your article attached. Thanks.</Body>
        <Date>08/12/2002</Date>
        <From>johnsonjr</From>
        <MessageId>1002</MessageId>
        <Subject>re: Article Review</Subject>
        <To>burrowswe</To>
    </emailList>
```

Figure 9.66 A portion of the XML document

Before we go any further, we need to point out two lines of code added to the form that are outside any events. These are shown in Figure 9.67. There are two Imports statements that are needed so we can perform the XSL transforms and find out the name of the directory where our application is currently running. We know that our application will be running in the Project's "bin" directory but we need to know the complete path and there is a method called getCurrentDirectory() in the System.IO namespace that does this.

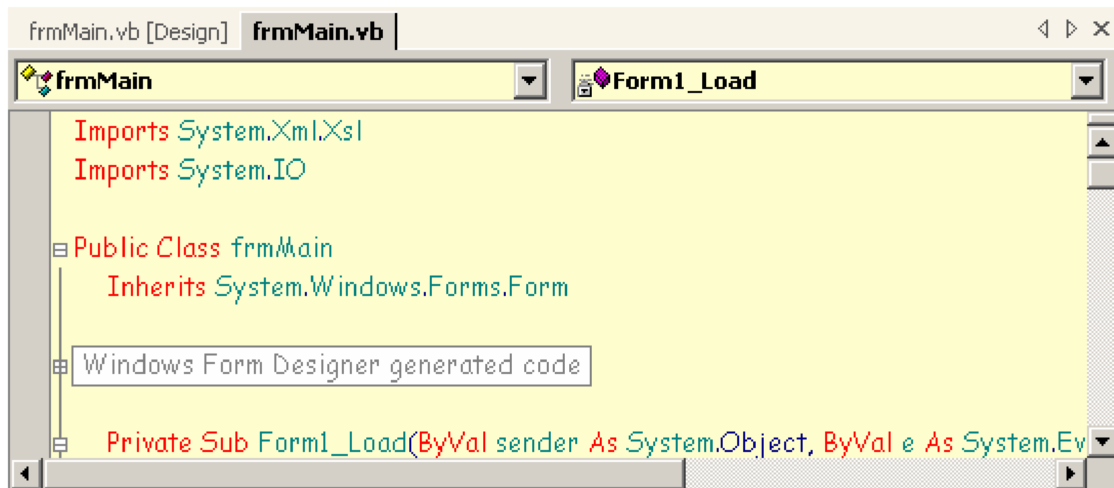


Figure 9.67 The Imports statements needed in the project

Now we turn our attention to translating the XML into HTML and displaying it on the Windows Form. If we were displaying it on a web form using a browser, there would be no difficulties. However, we want to display the HTML on a Windows Form and there is not a Visual Basic .NET component to do this. We can, however, add an ActiveX component to our Toolbox and use it as long as Internet Explorer is installed on the computer. To add a new component to the Toolbox, right-click on "Windows Forms" and select "Customize Toolbox…" from the popup menu as shown in Figure 9.68.

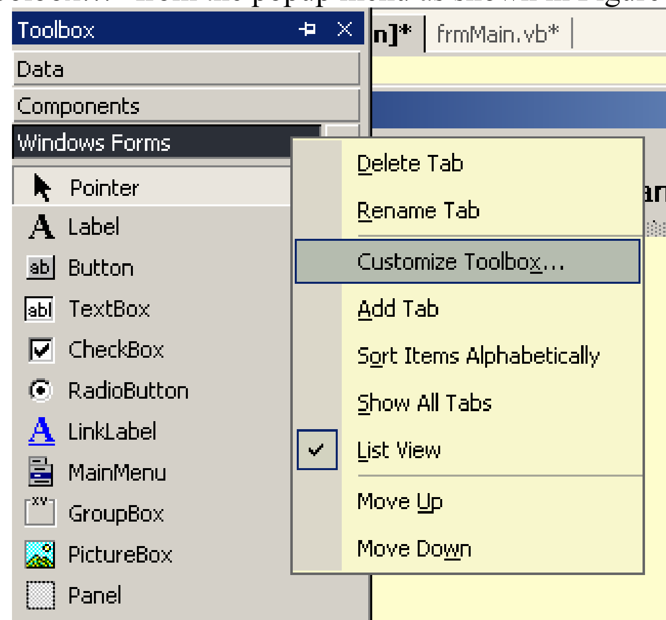


Figure 9.68 Popup menu used to a new component to the Toolbox

In the Customize Toolbox dialog box, scroll down until you can see the "Microsoft Web Browser" component as shown in Figure 9.69. Select this by selecting the check box and then click on OK.

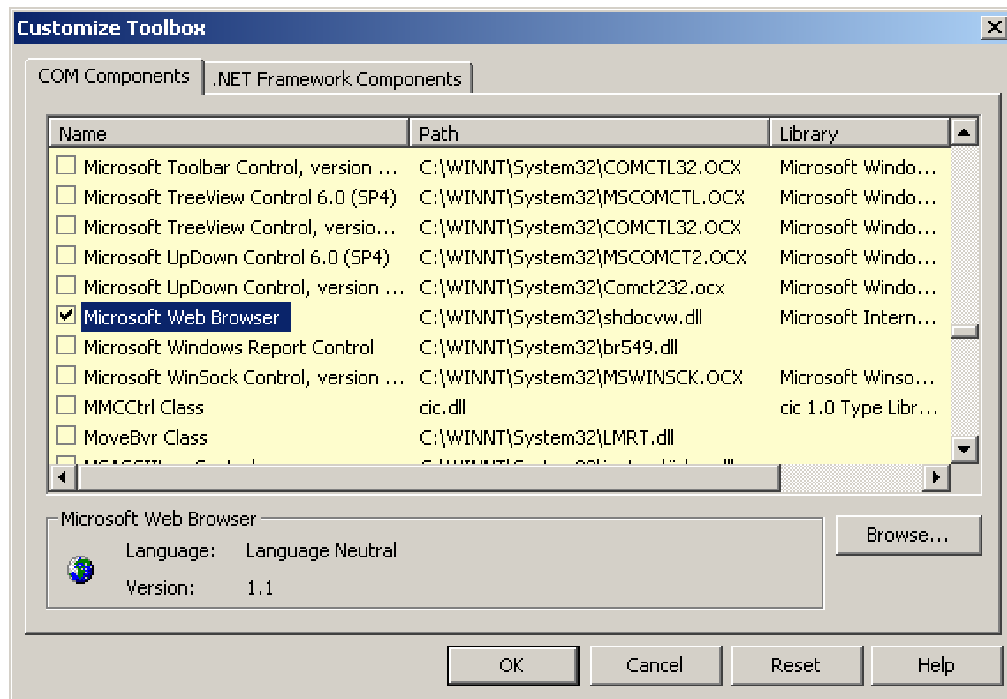


Figure 9.69 Dialog box for adding a new component to the Toolbox

When the dialog box closes, you should see the new component in the Windows Forms toolset. The new component will be called “Explorer” and will likely be at the bottom of the toolset. Figure 9.70 shows the new Explorer component. You use the ▼ icon on the lower right of the figure to scroll through the components in the toolbox. We added this component to our form and named it “brwMail”.

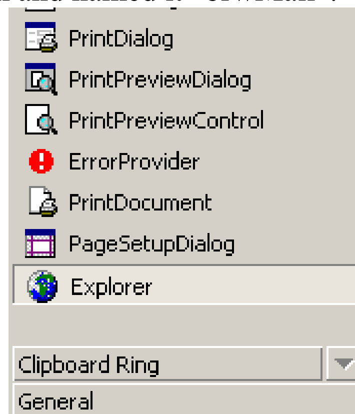


Figure 9.70 The Explorer component in the Windows Forms Toolbox

We now turn our attention to the code needed to convert the XML into HTML and display it in the Explorer component. The code that shows the email header information is shown in Figure 9.71.

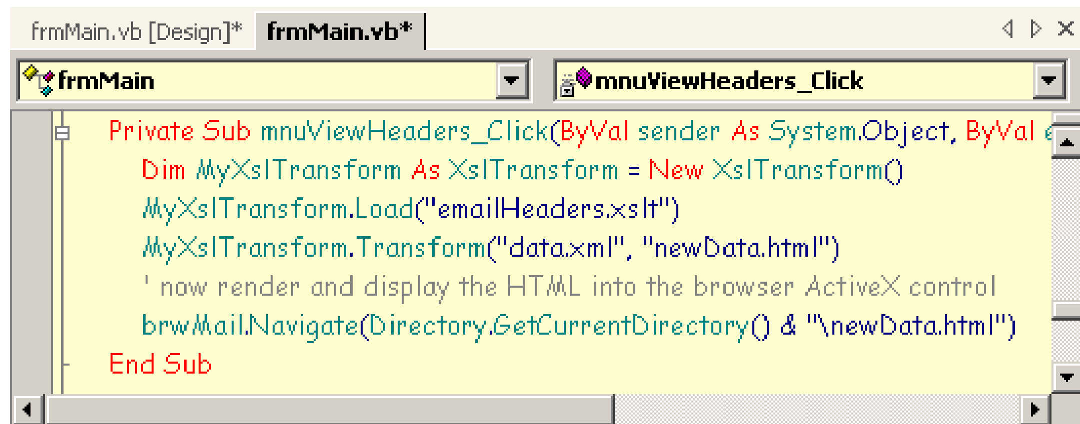


Figure 9.71 Code that converts XML into HTML and displays it in the Explorer component

We have worked with the XslTransform class in previous examples. We create a new XslTransform object and then use the Load() and Transform() methods to load an XSLT document and transform the XML document, using the XSLT, into an HTML document. The XSLT for this process is shown in Figure 9.72.

```
emailHeaders.xslt
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="dsEmail">
<HTML>
<BODY>
<TABLE cellspacing="3" cellpadding="8">
  <TR bgcolor="#AAAAAA">
    <TD class="heading"><B>Date</B></TD>
    <TD class="heading"><B>From</B></TD>
    <TD class="heading"><B>Subject</B></TD>
  </TR>
  <xsl:for-each select="emailList" >
  <TR bgcolor="#DDDDDD">
     <TD width="25%" valign="top">
       <xsl:value-of select="Date"/>
     </TD>
    <TD width="20%" valign="top">
       <xsl:value-of select="From"/>
    </TD>
     <TD width="55%" valign="top">
       <B><xsl:value-of select="Subject"/></B>
    </TD>
  </TR>
  </xsl:for-each>
</TABLE>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Figure 9.72 The XSLT used to show the email headers

The last statement in Figure 9.71 uses the Explorer component (brwMail) and its Navigate() method to load the HTML file and display it. The code:

```
Directory.GetCurrentDirectory() & "\newData.html"
```

finds the path of the project's bin folder and concatenates the name of the HTML file using a "\" to separate the directory path and the file name. This is then used by the Navigate() method to find the HTML.

The code for showing all the email content is almost identical to the code used for showing the headers only. The difference is the XSLT filename and more important, the XSLT contents. Figure 9.73 shows the code and Figure 9.74 shows the XSLT.

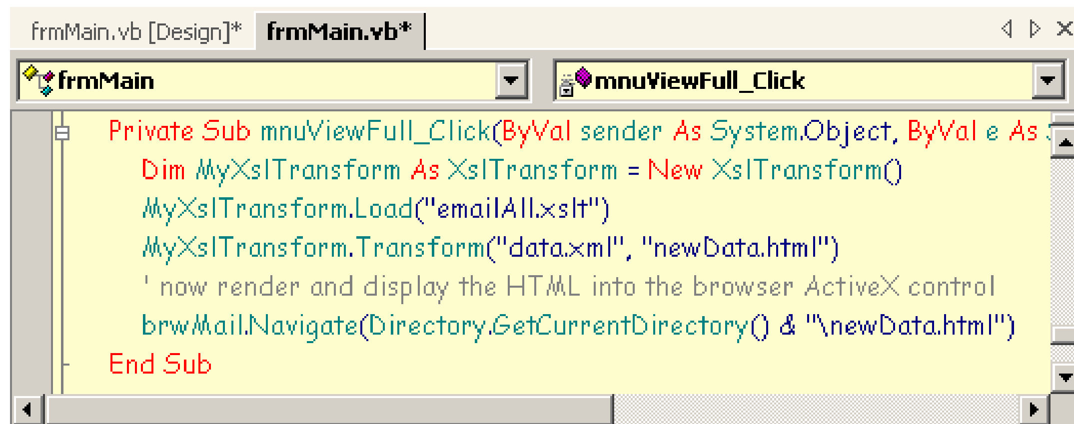


Figure 9.73 Code used to display all the email content

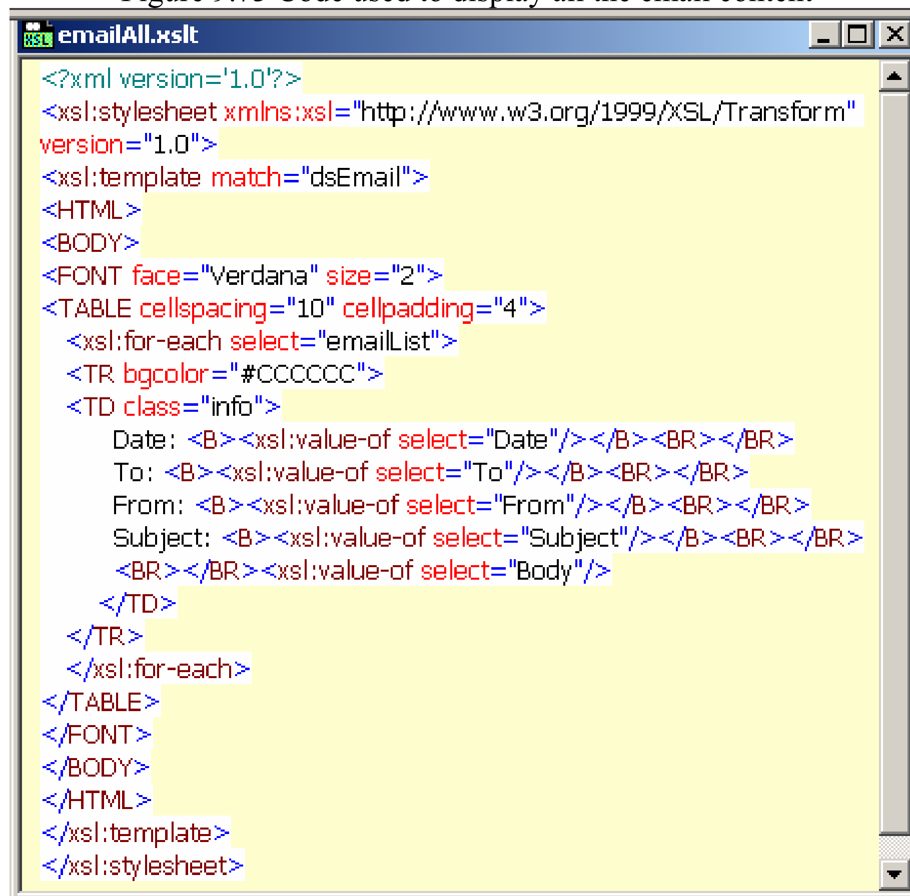


Figure 9.74 The XSLT used to show the entire email content

Exercise 9.15. Modify Example 9.7 so that the "complete" email view shows the elements as follows:

Date:<Date>	To:<To>
Subject:<Subject>	From:<From>

Message:<Body>

Exercise 9.16. Modify Example 9.7 so that the “header” view shows the email headers in order by the <From> element.

This concludes our discussion of XML and its associated technologies. As mentioned earlier, XML has very quickly become a very important part of many applications. It’s basic understanding is essential for anyone entering the information technology field and advanced knowledge will likely be very valuable. We encourage the reader to try and learn as much as possible about this important data technology.

9.4 PROJECT 10: PRODUCT CATALOG APPLICATION

Description of the Application

Most companies that sell products have a document called their “product catalog” that shows information on the products they sell. More than one audience might use this catalog. These could include their customers, as well as internal users like the sale and order department as well as the warehouse. Of course the different audiences would likely be interested in different information about a product. This project involves accessing the product information from a relational database and then, using XML and XSLT, transforming the information for two audiences – customer and inventory views.

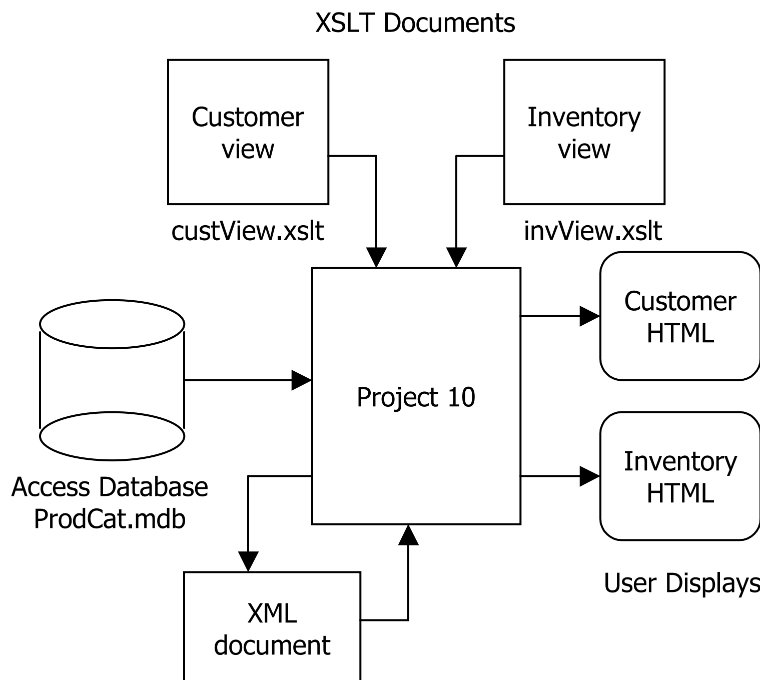


Figure 9.75 shows the overall transformation process performed by this project.

Figure 9.75 The transformations that Project 10 perform

The two views seen by the users are shown in Figure 9.76. Note that the customer view shows the products ordered by their description and the inventory view show the products ordered by product number.

<combine into one figure>

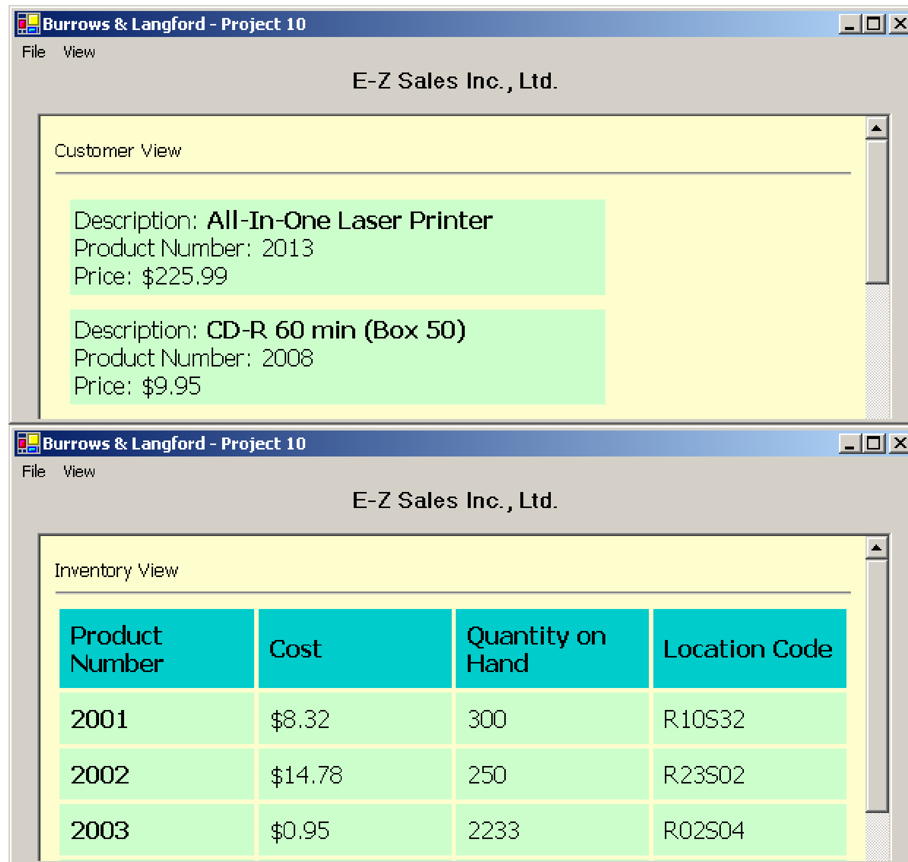


Figure 9.76 The two views generated by Project 10

Design of the Application

The Access database, named ProdCat.mdb, includes one table. This table includes six fields as shown in Figure 9.77.

ProdList : Table

Field Name	Data Type
ProdNo	Text
Description	Text
Cost	Number
Qty	Number
Price	Number
LocCode	Text

Figure 9.77 The ProdList table definition

You will need to write code that converts the records in this database table into an XML document and write this document to a file for later processing. Be aware of the potential problem with the namespace attribute as discussed in Example 9.7.

Once the XML document is available for processing, you need to apply the appropriate XSLT to transform in into HTML. The XSLT for producing the inventory view is shown in Figure 9.78. We leave the design of the XSLT for the customer view as a part of the project. What you have seen so far regarding XSLT examples and the XSLT for the inventory view should provide you with enough information to build the XSLT for the customer view.

```
invView.xslt
<?xml version='1.0'?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
version="1.0">
<xsl:template match="dsProdCat">
<HTML>
<BODY>
<FONT face="Tahoma" size="2">
Inventory View<hr></hr>
<TABLE cellspacing="3" cellpadding="8">
  <TR bgcolor="#00CCCC">
    <TD class="heading"><B>Product Number</B></TD>
    <TD class="heading"><B>Cost</B></TD>
    <TD class="heading"><B>Quantity on Hand</B></TD>
    <TD class="heading"><B>Location Code</B></TD>
  </TR>
  <xsl:for-each select="ProdList" >
  <TR bgcolor="#CCFFCC">
     <TD width="25%" valign="top">
       <b><xsl:value-of select="ProdNo"/></b>
     </TD>
    <TD width="25%" valign="top">
      $<xsl:value-of select="Cost"/>
    </TD>
     <TD width="25%" valign="top">
       <xsl:value-of select="Qty"/>
    </TD>
    <TD width="25%" valign="top">
       <xsl:value-of select="LocCode"/>
    </TD>
  </TR>
  </xsl:for-each>
</TABLE>
</FONT>
</BODY>
</HTML>
</xsl:template>
</xsl:stylesheet>
```

Figure 9.78 XSLT used to produce the HTML for the inventory view

Construction of the Application

Construction of the application is straightforward. You will need to add the Explorer ActiveX component to your Windows Forms Toolbox if you have not done so already (see Example 9.7). Your code uses the Navigate method of this control to link to the HTML you generated to render and display it.

Add a MainMenu component at set its menu options as shown in Figure 9.79. Use the click events for the menu to generate the appropriate views from the XML using the appropriate XSLT document.

<combine into one figure>

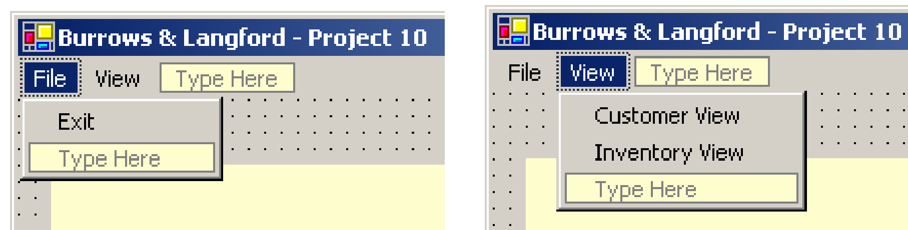


Figure 9.79 Menu options for Project 10

Chapter Summary

1. XML, the e**X**tensible **M**arkup **L**anguage, provides a technology-neutral way of representing data that has rapidly become the data representation standard of the Internet. XML stores data in a "tree" relationship with a single root node and where the non-root nodes are related to one and only one parent. Leaf nodes, nodes that have no children, are usually used to store content. Non-leaf nodes can also store content by using attributes. The data are stored in character format and are thus compatible with the data transmission protocols of the Internet.

   The "tags" of an XML document can be anything the designer of the XML document wants them to be. This allows documents to be designed that are described as "self-documenting" in that the tags relate directly to the problem-domain underlying the data. When you see a <Student> tag in the context of a school it is obvious what is being described.
2. It is possible to define a special set of rules, called an XML Schema, that control what is valid or not valid as far as the corresponding XML document is concerned. For example, if the XML Schema states that a particular tag, e.g., a <Course> tag, can only occur one time, then any document that conforms to that XML Schema must follow that rule.

   An XML Schema is itself an XML document so it must follow the rules associated with all XML documents.
3. An XML document can be well formed and also valid. A well-formed document follows the basic rules of all XML documents. There must be a single root node, a closing tag must exist for every opening tag, tags must be correctly nested, and all attribute values must be enclosed in quotes (either single or double quotes). The tags within an XML document are also case sensitive.

   A document is valid if it is well formed and it conforms to the XML Schema definition it is associated with.
4. Because tags are made up by the designer, it is possible to have two or more identical tags that have different meanings. To solve this problem the XML standard allows for the use of a "namespace". The namespace is associated with a unique Uniform Resource Identifier (URI) and tags associated with the namespace are prefixed with a unique character string associated with the namespace.

   Namespaces are also used to identify some predefined tags used by software such as XML parsers. This category of software analyzes the XML tree and determines its content. Some namespaces, such as those defined by the extensible stylesheet language (XSL), are used by the processor to perform the processing task identified by the XSL instruction.

5. EXtensible Stylesheet Language Transforms (XSLT) provide a way of transforming one XML document into a new XML document. This capability is extremely useful in the context of transforming XML into HTML so that the data can be formatted and displayed within a browser.
6. Visual Basic .NET includes classes that define methods to work with XML and its related technologies. For example, the DataSet class includes the WriteXml() and ReadXml() methods that process XML data directly and the XslTransform class includes the Load() and Transform() methods to load XSLT and transform XML into another form using the XSLT document. Visual Basic .NET also includes a number of methods that work with XML Schemas.

Key Terms

attributes	facet	valid
complex type	HTML	well formed
DataRelation objects	namespace	XBRL
document prolog	processing instruction	XML
ebXML	simple type	XML Schema
extensible	template	XSL
eXtensible Markup Language	tree	XSLT
eXtensible Stylesheet Language	uniform resource identifier	
eXtensible Stylesheet Language Transforms	uri	

End-of-Chapter Problems

1. What function does the use of namespaces serve in the context of an XML document? Provide an example of namespace definition.
2. Explain the difference between a well-formed and a valid XML document. A document can be well-formed but not be valid. Can a document be valid but not well-formed?
3. A university uses qualifications to determine a student's quarterly tuition. These qualifications include class standing (undergraduate, graduate, or professional) as well as residency status (in-state or out-of-state). Draw a tree representation of an XML document that would store this information uniquely for each student. Label the root, branches, and leaves of the tree.
4. Explain the difference between an XML Schema document and an XSLT document. What is the function of an XSLT processor?
5. In the definition of an XML Schema, when would you use a simple type and when would you use a complex type? Give an example of using each. For the complex type, describe when you would use a named or unnamed complex type.
6. Why are nested relationships needed in XML? Are key fields required to create a nested relationship?
7. Are the relationships created for XML Schemas used in the same manner as the relations that link databases in the last chapter?
8. How are XSLT transformations performed with Visual Basic code? Write the code that would perform a transformation of an XML file called studentList.xml using a transformation file named attendanceForm.xslt.
9. What would be the purpose of using the FileOpen() method (with the OpenMode.Output parameter) immediately followed by the FileClose() method? Write a statement line to stores in a string variable the proper path to the file "studentList.xml" if it was located in the current directory.
10. Name and describe the three primary methods that can be executed on a DataSet object to perform XML operations.

Programming Problems

1. Use Visual Basic.Net to create an XML Schema that can hold information about your CDA (CD Audio) collection. Create an element in the schema and name it CDA. Columns for the element should include CD id, title, artist, number of tracks,

and total running time. Define the cd_id field as an integer and make it the primary key. After you've created the schema, create an XML file from it, add five records, and save the data.

2. XML can be used to support hierarchical data in a nested data set. In chapter one you learned about the hierarchical nature of classes and how inheritance could be used effectively to handle employee data. Create a Visual Basic.NET application that contains the schema for a Store database that has hourly and salaried employees. You will need to define a Store element, a HourlyEmployee element, and a SalariedEmployee element. The HourlyEmployee element should contain fields for an employee number, a name, an hourly rate, and hours worked. The SalariedEmployee element should contain fields for an employee number, a name, and a salary. The Store element should contain fields for the store ID and location. When you parent the Store element to the other two elements, it will also contain fields for the two employee types.

   Save the schema and create an XML file with some employee data. When you examine the XML of the data file, the beginning should appear to be something like this:

```
<?xml version="1.0" encoding="utf-8"?>
<EmployeeSchema xmlns="http://tempuri.org/EmployeeSchema.xsd">
  <Store xmlns="http://tempuri.org/EmployeeSchema.xsd">
    <storeid>0</storeid>
    <location>San Diego</location>
    <HourlyEmployee>
      <employeenum>0</employeenum>
      <hourlyrate>8.5</hourlyrate>
      <hoursworked>42</hoursworked>
      <name>John Doe</name>
    </HourlyEmployee>
    <SalariedEmployee>
      <employeenum>0</employeenum>
      <salary>42000</salary>
      <name>Jane Doe</name>
    </SalariedEmployee>
  </Store>
```

3. Contact databases are often created which hold only two telephone numbers: business and personal. The increase in communications has increased the needs to an unknown number of related phone numbers from pagers to cell phone. Create an XML Schema for a contact database that connects two elements (Contact and Telephone) with a one-to-many relation. Define the Contact element with the fields: contactid, name, address, city, state, and zip. Set the contactid as the Primary DataSet key. Define the Telephone element with fields: contactid, type, phonenum, and notes. Set the contactid field as a key field. Add a relation between the two elements that allows many telephone number records for each contact. Create an XML data file from the schema.

4. An XML complex type can be very helpful when you need to re-use a number of element fields in a schema. Create an XML Schema with a complex type named FullAddress that contains fields for address, aptnum, city, state, and zip. In the schema, add a new element named billto and set the type to FullAddress. As soon as you set the type, you should see all of the fields of the complex type appear as dimmed text in the body of the element. Create another element named shipto using the same method as the billto element. Create an Order element that contains shipto and billto elements. Save the schema and create an XML data file from it.
5. Create an application that can read an XML file into a DataSet and display it in a DataGrid control. Use a form that has a TextBox control, two Button controls, and a DataGrid control. The text box should be used to allow the user to enter the path and filename of the desired XML file. For one of the buttons, set the Text property to '...' and place it directly to the right of the filename text box. This button will activate the OpenFile dialog box to allow the user to browse for the appropriate file.

   In the toolbox, you will find a component named OpenFileDialog. Double-click it to add it to the current form where it will be automatically named OpenFileDialog1. You will need to add code to your browse button to use this dialog. The code should first set the Filter property to only display XML files (OpenFileDialog1.Filter = "XML files | *.xml"). Then use the ShowDialog() method you learned in chapter 6 to display the window. Finally, the filename and path that the user selected can be obtained from the FileName property of the dialog. Store this value in the text box on the form.

   To complete the application, use the second button you created on the form to read the XML file (specified in the TextBox) into a DataSet object and display it in the DataGrid as shown in the chapter. It would be a good idea to add some error trapping in case the user selected an invalid XML file. To test the application, use the XML file of your CDAList that you created in Programming Problem #1.
6. As mentioned in the chapter, a DataSet object can not only read an XML file, but it can write it from a current DataSet object as well. The WriteXML() method can be used in the same manner as the ReadXML() method except the supplied filename is the output destination (WriteXML("c:\stores.xml")). Since Internet Explorer 6 can display any XML file complete with color highlighting and a collapsible hierarchy, the Microsoft Web Browser control can be used to show an XML file that has been written to file storage.

   Create an application that reads the stores table from the pubs databases, writes an XML file with all of the data in it, and displays this XML file on a form with the Microsoft Web Browser component. Don't forget to fill the data adapter in your form load event or the XML file that is written will be empty of data.