5

Introduction to XML Development

The Extensible Markup Language (XML) has been around long enough now that the majority of users are probably familiar with it. It is a platform-independent language that can be used to transmit both information and data. Rational Application Developer provides a complete set of visual tools to help you develop XML applications. XML is so versatile that it can be used anywhere—you can use it in a Web project, with a relational database, with an EJB, etc. In this chapter, you learn how to create XML and use it in your applications.

Rational Application Developer provides a number of XML authoring tools. Figure 5-1 shows the list of available XML wizards.

To create an XML file, you use the XML File Creation wizard and the XML editor. When you use the XML File Creation wizard, you will be asked whether you want to create the XML file from a *Document Type Definition* (*DTD*) file, from an *XML Schema* (*XSD*) file, or from scratch, as shown in Figure 5-2. Both the DTD and XSD files define the vocabularies and describe the proper syntax of building an XML document.

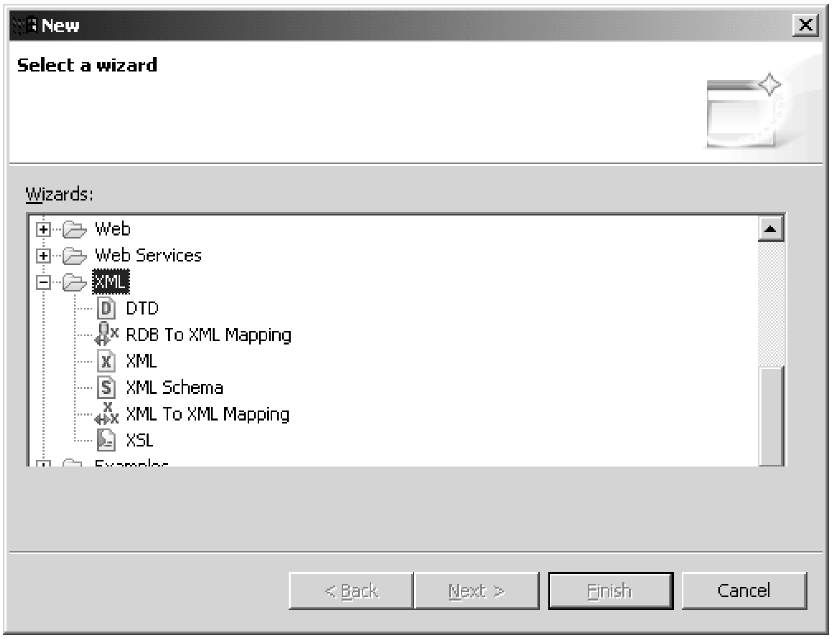
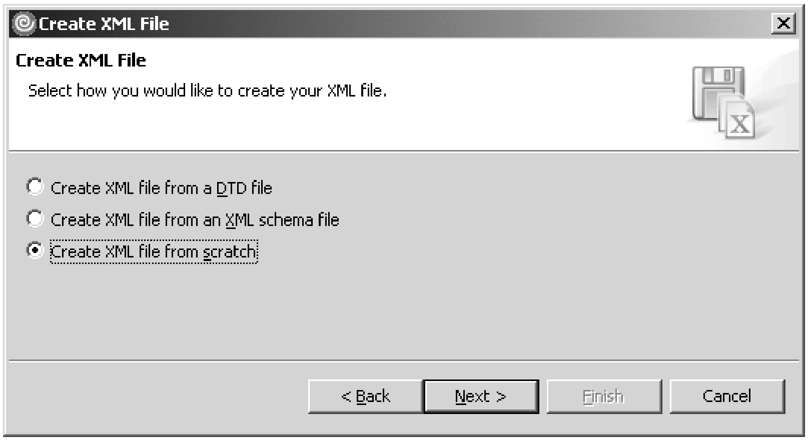


Figure 5-1: List of XML wizards.


Create XML File

Create XML File

Select how you would like to create your XML file.

- ( ) Create XML file from a DTD file
- ( ) Create XML file from an XML schema file
- (●) Create XML file from scratch

Figure 5-2: The XML File Creation wizard.

If you have a DTD or an XSD defined for your XML document, the XML parser can parse and validate the XML document against the definition. Furthermore, when different parties exchange XML data, definitions like these enable each party to validate the incoming and outgoing XML files. This helps ensure data integrity and minimizes any miscommunications.

DTD has a longer history than XSD. The following is a sample of a DTD document defined internally within an XML document, where the DTD document starts with !DOCTYPE and each element is defined with !ELEMENT:

```
<?xml version="1.0"?>
<!DOCTYPE address [
  <!ELEMENT address (street, city, country)>
  <!ELEMENT street  (#PCDATA)>
  <!ELEMENT city    (#PCDATA)>
  <!ELEMENT country (#PCDATA)>
]>
<address>
  <street>123 Ave</street>
  <city>Toronto</city>
  <country>Canada</country>
</address>
```

There are some disadvantages of using DTD documents. A DTD document is written in a different syntax than the XML document itself. In addition, DTD has a limited capability for specifying data types. As a result, the XML Schema was created to be an improvement over DTD.

The first improvement is that XSD is written in XML format. It provides numerous enhanced data types and also allows user-defined data types. XSD is object-oriented, so it supports data-type inheritance, restrictions, and patterns. Rational Application Developer provides the support to create XSD files graphically or textually. Figure 5-3 is an XSD file in the Graph view.

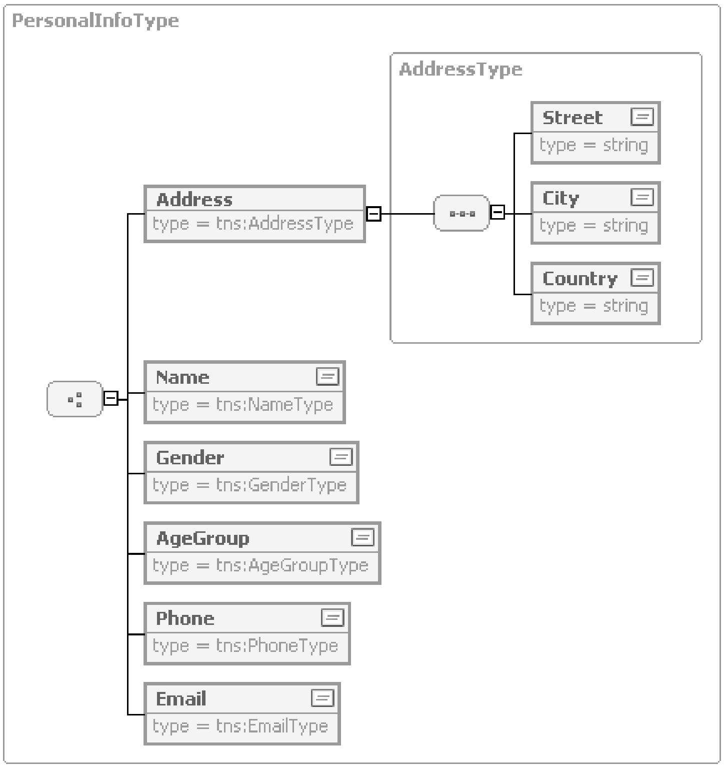


Figure 5-3: XML Schema in the Graph view.

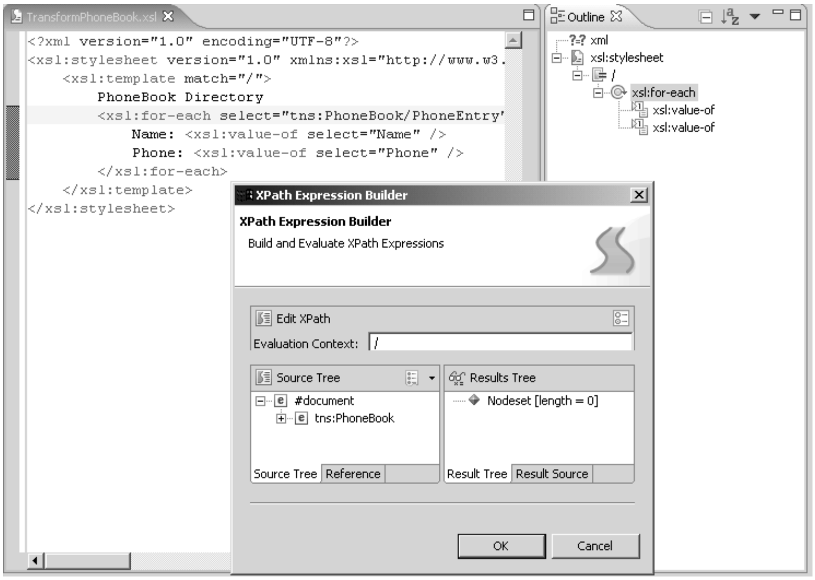
The following is a sample of an XML Schema document:

```
<?xml version=“1.0” encoding=“UTF-8”?>
<schema xmlns=“http://www.w3.org/2001/XMLSchema”
        targetNamespace=“http://www.ibm.com”
        xmlns:Address=“http://www.ibm.com”>

        <element name=“Address”>
                <complexType>
                       <sequence>
                       <element name=“Street” type=“string” />
                       <element name=“City” type=“string” />
                       <element name=“Country” type=“string” />
                       </sequence>
                </complexType>
        </element>
</schema>
```

This document has an element named Address and a sequence of elements named Street, City, and Country. As you can see, an XSD is in XML format. Tutorial 1 describes the syntax of XSD in more detail. The editor will produce compilation errors if an XML document does not conform to the DTD or XSD specified in the creation of the XML document.

After you have created XML documents, you will need to access them programmatically. One common way to read and manipulate XML documents is to use the DOM and SAX parsers, which you will see in tutorial 2. Another way is to use Service Data Objects (SDO). As mentioned in chapter 4, SDO is a data programming architecture that unifies the data program across any datasources. It provides robust APIs for querying and updating data, and consists of two major components: *data graphs* and *data objects*. Data objects are the generic representations of the data. Data graphs are envelops for data objects. Tutorial 5 demonstrates how SDO data objects can be created to manipulate XML documents.

In addition to creating DTD, XSD, and XML documents, Rational Application Developer also provides wizards to create *Extensible Stylesheet Language Transformations* (*XSLT*). In the XSLT editor, you can easily create an XPath expression using the XPath Expression Builder, as shown in Figure 5-4.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.
    <xsl:template match="/">
        PhoneBook Directory
        <xsl:for-each select="tns:PhoneBook/PhoneEntry
            Name: <xsl:value-of select="Name" />
            Phone: <xsl:value-of select="Phone" />
        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

XPath Expression Builder

Build and Evaluate XPath Expressions

Evaluation Context: /

```
#document
    tns:PhoneBook
```

```
Nodeset [length = 0]
```

Figure 5-4: The XSLT editor and XPath Expression Builder.

In tutorial 3, you will create an XSL style sheet using the XSL editor. Later, you will use the Java API for XML Processing (JAXP) to dynamically transform an XML document into another format using the XSL style sheet.

Rational Application Developer has support for translating database data into XML directly and vice versa. The support is called SQLToXML and XMLToSQL. You can generate an XST template from a database connection, which enables you to directly translate database data into XML. In addition, you can export an XML document directly into a database as a way to insert, update, or delete data. Tutorial 4 demonstrates this feature.

To summarize, this chapter includes the following tutorials to explore some of the XML tools in Rational Application Developer:

- Tutorial 1: Creating XSD and XML Files
- Tutorial 2: Using JAXP
- Tutorial 3: Using Extensible Stylesheet Language Transformations (XSLT)
- Tutorial 4: Using XML with SQL
- Tutorial 5: Using SDO with XML

Tutorial 1: Creating XSD and XML Files

This tutorial demonstrates how to create XML Schema (XSD) and XML documents using Rational Application Developer. You will create an XSD that describes the syntax of an address book. Later, you will create an XML document that conforms to the schema you created. Through the creations of these documents, you will examine the different constructs available in the XSD and XML language.

Before we begin, let’s take a look at the following simple XSD example:

```
<?xml version=“1.0” encoding=“UTF-8”?>
<schema xmlns=“http://www.w3.org/2001/XMLSchema”
      targetNamespace=“http://www.ibm.com”
      xmlns:Address=“http://www.ibm.com”>

      <element name=“Address”>
             <complexType>
```

```
                       <sequence>
                       <element name="Street" type="string" />
                       <element name="City" type="string" />
                       <element name="Country" type="string" />
                       </sequence>
               </complexType>
        </element>
</schema>
```

This XSD example describes an Address element that contains a sequence of String elements: Street, City, and Country.

There are two kinds of *type* definitions in XSD: *simpleType* and *complexType*. These type definitions could be named or anonymous. In this example, the *complexType* definition is anonymous. The Address element is a global element definition associated with a regular XSD built-in type, either a simpleType or a complexType.

Some examples of the regular XSD built-in data types are *string, decimal, boolean, integer, date,* and *time*. A simpleType allows you to control the values of an element by placing a restriction on a built-in data type. This restriction is called a *facet*.

A complexType element definition allows child elements or attributes to be defined. It can contain many other elements. The <Address> element in the example is a complexType.

If you created an XML document conforming to this XML Schema, it might look similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<Address:Address xmlns:Address="http://www.ibm.com"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ibm.com Address.xsd ">

        <Street>8200 Warden Ave.</Street>
        <City>Markham</City>
        <Country>Canada</Country>
</Address:Address>
```

Namespace

When defining an XSD, you need to assign it to a *namespace*. In the above example, the XSD is in the namespace named http://www.ibm.com. That is called the *targetNamespace*. Furthermore, an XML schema can reference other schemas, so elements defined in those schemas can be used or referenced.

One of the namespaces that all the schemas need to refer to is the w3.org XML Schema. This defines the basic vocabularies for XSD and is sometimes referred to as “the XML schema for Schemas.” Keywords such as <schema>, <element>, and <complexType> are from the w3.org XML Schema. The namespace for the w3.org XML Schema is http://www.w3.org/2001/XMLSchema.

Since an XSD file can have references to multiple namespaces, it is possible to have two schemas that have elements with the same name. To distinguish between these elements, you can define a prefix for each namespace. For example, xmlns:Address=“http://www.ibm.com” has a prefix of *Address* that refers to the namespace http://www.ibm.com. You can also define a default namespace for your XSD. For example, the default namespace for the above example is the w3.org XML Schema. When referring to the elements defined in the default namespace, you can directly use the element name without any prefix.

Now, let’s begin the tutorial. In this tutorial, you create an XSD file and an XML file in a Java project. Later, you will extend the XSD using some of the more advanced XML Schema constructs.

Step 1: Create a New XML Project

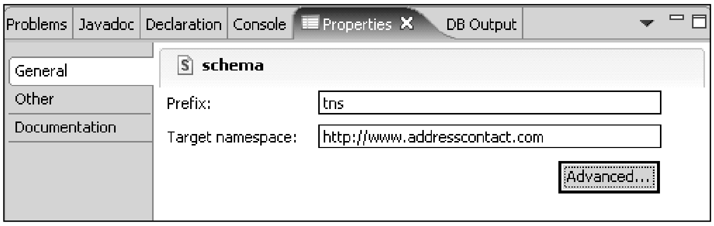
Create a new Java Project named *Ch5XMLProject* and a folder named *xml*. XML and XSD files can be stored in any type of project.

1. From the workbench, select **File => New => Project**. Click **Next**.
2. Select **Java => Java Project**. Click **Next**.
3. Enter **Ch5XMLProject** as the project name and click **Finish**.
4. Right-click **Ch5XMLProject** and select **New => Folder**.
5. Enter **xml** as the folder name. Click **Finish**.

Step 2: Create an XML Schema (XSD)

Create an XSD file named *ContactInfo.xsd*:

1. Right-click the **Ch5XMLProject/xml** and click **New => Other**. Check the **Show All Wizards** check box.
2. Select **XML => XML Schema**. Click **Next**.
3. Enter **ContactInfo.xsd** as the file name. Click **Finish**.
4. To modify the pre-filled namespaces, go to the Properties view. In the General tab, change the target namespace to **http://www.addresscontact.com**, as shown in Figure 5-5.



Prefix: tns
Target namespace: http://www.addresscontact.com

Figure 5-5: The Properties view of ContactInfo.xsd.

5. In the Outline view, right-click **ContactInfo.xsd** and select **Add Complex Type**. This will automatically create a <complexType> element, as shown in Figure 5-6. Click **NewComplexType** and you will be at the zoomed-in Graph view of the complexType node.
6. In the Graph view, modify the name to **AddressType**.
7. Right-click **AddressType** and select **Add Sequence**.
8. Right-click the **Sequence** node and select **Add Element**. Change the name of the element to **Street**.
9. Repeat step 8 to add the other two elements, **City** and **Country**. The result is shown in Figure 5-7.

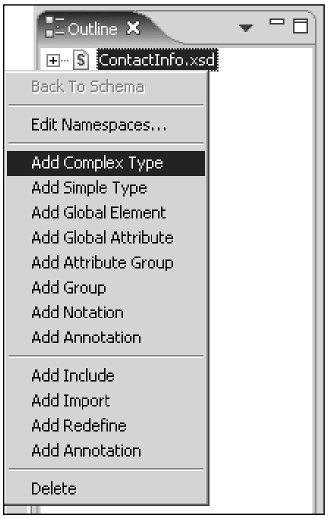


Figure 5-6: The Add Complex Type menu.

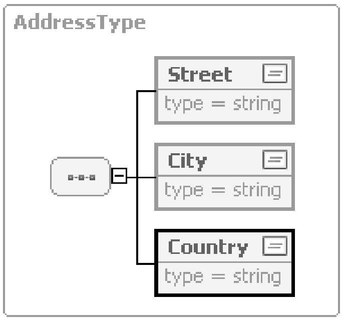


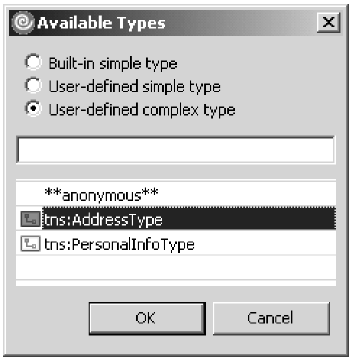
Figure 5-7: The AddressType element.

If you look at the code in the Source view, you should see the following:

```
<complexType name="AddressType">
   <sequence>
      <element name="Street" type="string" />
      <element name="City" type="string" />
      <element name="Country" type="string" />
   </sequence>
</complexType>
```

Create another complexType named *PersonalInfoType*:

1. Right-click **ContactInfo.xsd** in the Outline view and select **Add ComplexType**.
2. Right-click **PersonalInfoType** and select **Add All**.
3. Right-click the **All** node and select **Add Element**. Modify the name to **Address**.
4. In the Properties view, change the type by clicking the **Browse** button. Select the **User-defined complexType** radio box and click **tns:AddressType**, as shown in Figure 5-8. Click **OK**.



Available Types

( ) Built-in simple type
( ) User-defined simple type
(•) User-defined complex type

anonymous
tns:AddressType
tns:PersonalInfoType

Figure 5-8: Selecting tns:AddressType.

Figure 5-9 shows the resulting PersonalInfoType complexType. If you expand on the Address element, you should see the definition of the AddressType.

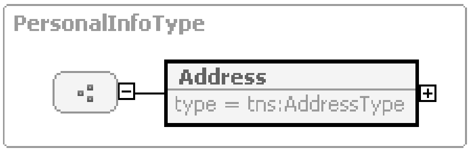
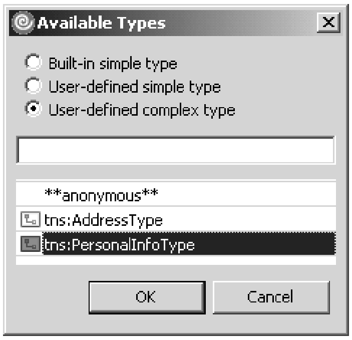


Figure 5-9: PersonalInfoType in the Graph view.

You should see the following in the Source view:

```
<complexType name="PersonalInfoType">
<all><element name="Address" type="tns:AddressType" />
</all>
</complexType>
```

1. In the Outline view, right-click **ContactInfo.xsd** and click **Add Global Element**. Change its name to **PersonalInfo**.
2. In the Properties view, browse the type to select **tns:PersonalInfoType**. As you can see in Figure 5-10, it is a user-defined complex type.



Built-in simple type
User-defined simple type
[✓ User-defined complex type]

anonymous
tns:AddressType
tns:PersonalInfoType

Figure 5-10: Selecting tns:PersonalInfoType.

3. To format the document, right-click in the editor (in the Source view) and click **Format => Document**.
4. Save the file.

The final XSD, shown in Figure 5-11, should look like this in the Source view:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.addresscontact.com"
xmlns:tns="http://www.addresscontact.com">
   <complexType name="AddressType">
       <sequence>
          <element name="Street" type="string"></element>
          <element name="City" type="string"></element>
```

```
            <element name="Country" type="string"></element>
        </sequence>
    </complexType>

    <complexType name="PersonalInfoType">
        <all>
            <element name="Address" type="tns:AddressType">
</element>
        </all>
    </complexType>

    <element name="PersonalInfo"
type="tns:PersonalInfoType"></element>
</schema>
```

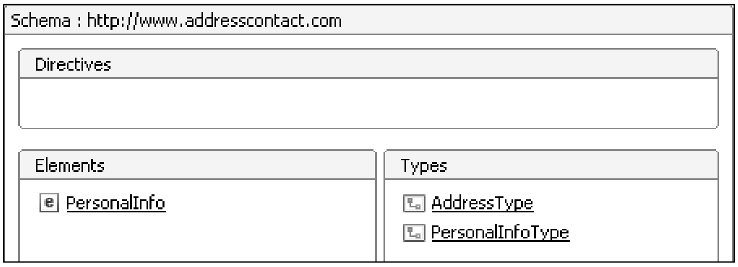


Figure 5-11: The final XSD in the Graph view.

Step 3: Generate an XML File

The XML tools in Rational Application Developer allow you to create an XML from an XSD, and vice-versa. In this step, you generate an XML file named *ContactInfo.xml* that uses the XSD file created in the previous step.

1. In the Navigator view, right-click **ContactInfo.xsd** and click **Generate => XML File**. Click **Next**.
2. Enter **ContactInfo.xml** as the file name. Click **Next**.
3. Click **Finish**. Notice that the generated XML file has all the required elements filled out:

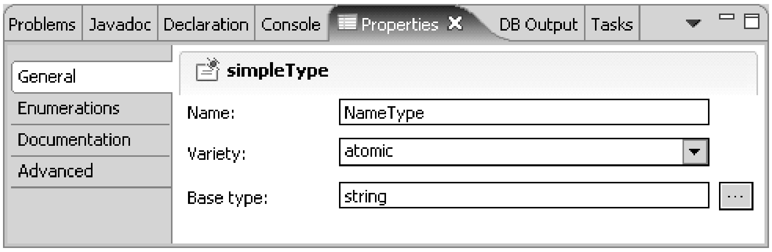
```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PersonalInfo xmlns:tns="http://www.addresscontact.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.addresscontact.com ontactInfo.xsd ">
  <Address>
    <Street>Street</Street>
    <City>City</City>
    <Country>Country</Country>
  </Address>
</tns:PersonalInfo>
```

Step 4: Expand the XML Schema

You created a simple XSD file in step 2 that contains one global element named PersonalInfo, which is of PersonalInfoType. In this step, you further expand the XSD to include other XSD types.

Add a <simpleType> definition named *NameType*:

1. In the ContactInfo.xsd editor (Outline view), right-click **ContactInfo.xsd** and click **Add Simple Type**.
2. In the Properties view, change the name to **NameType**.
3. Select **atomic** from the Variety drop-down box, as shown in Figure 5-12.



Name: NameType
Variety: atomic
Base type: string

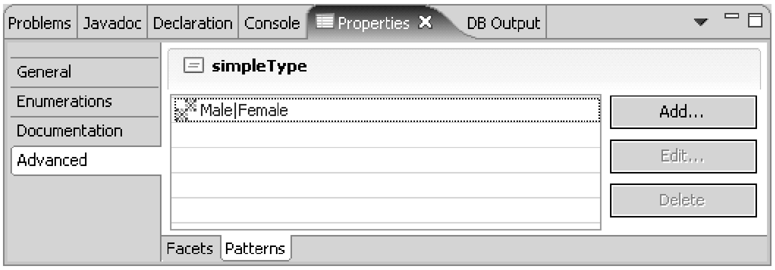
Figure 5-12: A simpleType General page.

The following code is generated:

```
    <simpleType name="NameType">
            <restriction base="string"></restriction>
    </simpleType>>
```

The next simpleType definition you need to create, named *GenderType*, has a pattern restriction that only allows a value of either *Male* or *Female*.

1. Right-click **ContactInfo.xsd** and click **Add Simple Type**. In the Properties view, change the name to **GenderType**. Select **atomic** from the Variety drop-down box.
2. Switch to the Advanced page and select the **Patterns** tab.
3. Click **Add** and enter **Male|Female** in the Current regular expression text field, as shown in Figure 5-13. Click **Next** to test the expression. Click **Finish**.



Male|Female

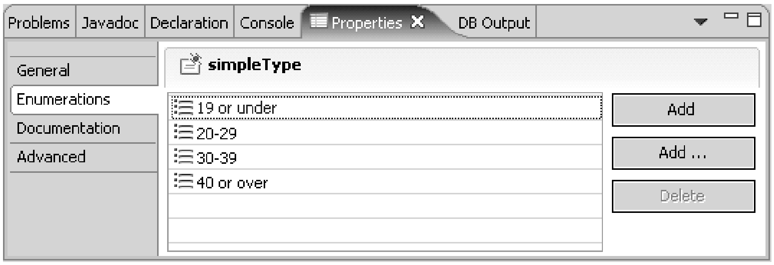
Figure 5-13: The GenderType simpleType Advanced page.

The code for GenderType is as follows:

```
<simpleType name="GenderType">
   <restriction base="string">
          <pattern value="Male|Female" />
   </restriction>
</simpleType>
```

Create a simpleType named *AgeGroupType*, for which the acceptable values are *19 or under*, *20-29*, *30-39*, and *40 or over*:

1. Right-click **ContactInfo.xsd** and click **Add Simple Type**. In the Properties view, change the name to **AgeGroupType**. Select **Atomic** as the variety.
2. Switch to the **Enumerations** page and click **Add**. Enter **19 or under**, **20-29**, **30-39**, and **40 or over** as shown in Figure 5-14. Click **OK**.



19 or under
20-29
30-39
40 or over

Figure 5-14: AgeGroupType enumerations.

The code for the AgeGroupType enumerations is as follows:

```
<simpleType name="AgeGroupType">
   <restriction base="string">
          <enumeration value="19 or under" />
          <enumeration value="20-29" />
          <enumeration value="30-39" />
          <enumeration value="40 or over" />
   </restriction>
</simpleType>
```

Create a simpleType named *PhoneType* whose only valid value is *ddd-ddd-dddd* where *d* is a digit. Each digit must be between zero and nine.

1. Right-click **ContactInfo.xsd** and click **Add Simple Type**. Enter **PhoneType** as the name. Select **Atomic** as the variety.
2. In the Advanced page's **Pattern** tab, add a new pattern and enter **[0-9]{3}-[0-9]{3}-[0-9]{4}** as the regular expression. Click **Next** to test it.
3. Enter **111** in the Sample Text field. You will see an error message saying that the sample text does not match the expression.
4. Enter **905-123-4567** in the Sample Text field. You should see no error message. Click **Finish**.

The code for this simpleType is shown here:

```
<simpleType name="PhoneType">
   <restriction base="string">
          <pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}" />
   </restriction>
</simpleType>
```

Create a simpleType named *EmailType*:

1. Right-click **ContactInfo.xsd** and click **Add Simple Type**. EmailType has a pattern restriction. The only valid value is an email address, which can contain any number of characters or digits followed by the "at" sign (@). To specify this kind of pattern, you can use the plus sign to indicate one or more characters. For example, if you add a plus sign after *[a-zA-Z0-9]*, it means the value must contain one or more uppercase, lowercase, or numeric characters. The pattern *[a-zA-Z]{3}* means that the value must contain three characters, which can be either uppercase or lowercase. Use the same procedure as the previous step to create the simple type, and add a pattern restriction for an email address, like this:

```
<simpleType name="EmailType">
   <restriction base="string">
          <pattern value="[a-zA-Z0-9]+@[a-zA-Z0-9]+.[a-zA-Z]{3}" />
   </restriction>
</simpleType>
```

2. Modify the PersonalInfoType in the Source view to include the newly created types, as follows:

```
<complexType name="PersonalInfoType">
   <all>
          <element name="Address"
type="tns:AddressType"></element>
          <element name="Name" type="tns:NameType" />
          <element name="Gender" type="tns:GenderType" />
          <element name="AgeGroup" type="tns:AgeGroupType" />
          <element name="Phone" type="tns:PhoneType" />
          <element name="Email" type="tns:EmailType" />
   </all>
</complexType>
```

   The <all> tag means that any XML element of PersonalInfoType must contain all of these elements—Address, Name, Gender, AgeGroup, Phone, and Email. Unlike the <sequence> tag, the <all> tag does not require the elements to be in sequence. Alternatively, you could add these elements graphically in the Graph view.

3. Save the file.

Here is the complete ContactInfo.xsd file, shown graphically in Figure 5-15:

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

<complexType name="AddressType">
        <sequence>
              <element name="Street" type="string"></element>
              <element name="City" type="string"></element>
              <element name="Country" type="string"></element>
        </sequence>
</complexType>

<complexType name="PersonalInfoType">
   <all>
        <element name="Address" type="tns:AddressType"></element>
        <element name="Name" type="tns:NameType" />
        <element name="Gender" type="tns:GenderType" />
        <element name="AgeGroup" type="tns:AgeGroupType" />
        <element name="Phone" type="tns:PhoneType" />
        <element name="Email" type="tns:EmailType" />
   </all>
</complexType>

   <element name="PersonalInfo"
     type="tns:PersonalInfoType"></element>

   <simpleType name="NameType">
        <restriction base="string"></restriction>
   </simpleType>

   <simpleType name="GenderType">
        <restriction base="string">
              <pattern value="Male|Female"></pattern>
        </restriction>
   </simpleType>

   <simpleType name="AgeGroupType">
        <restriction base="string">
              <enumeration value="19 or under"></enumeration>
              <enumeration value="20-29"></enumeration>
              <enumeration value="30-39"></enumeration>
              <enumeration value="40 or over"></enumeration>
        </restriction>
   </simpleType>

   <simpleType name="PhoneType">
        <restriction base="string">
              <pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"></pattern>
        </restriction>
    </simpleType>

    <simpleType name="EmailType">
        <restriction base="string">
            <pattern value="[a-zA-Z0-9]+@[a-zA-Z0-9]+.[a-zA-Z]{3}" />
        </restriction>
    </simpleType>

</schema>
```

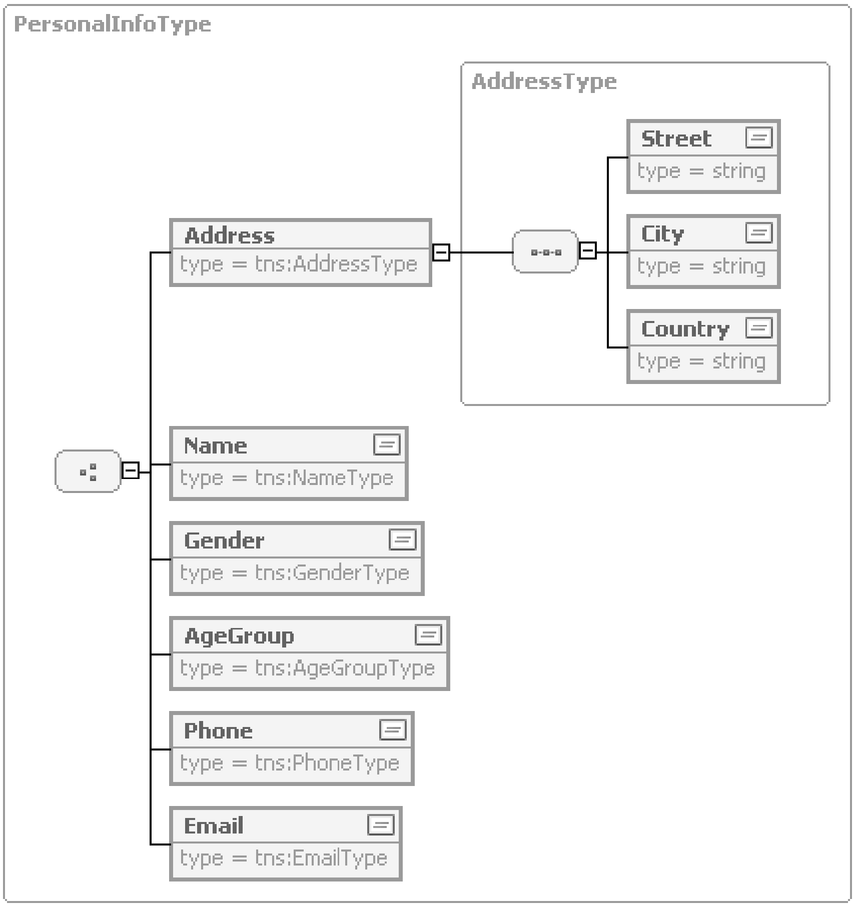


Figure 5-15: The final Contactinfo.xsd in the Graph view.

Step 5: Generate the XML File

1. In the Navigator view, delete the ContactInfo.xml file you previously generated.
2. Right-click **ContactInfo.xsd** and click **Generate => XML File**. Click **Next**.
3. Enter **ContactInfo.xml** as the file name. Click **Next**.
4. Click **Finish**. Right-click the XML file and select **Run Validation**. You might notice that there are validation errors on the Gender, Phone, and Email elements. This is because the values do not obey the restriction facet.
5. Modify the value in the Gender element to **Female**.
6. Modify the value in the Phone element to **333-333-3333**.
7. Modify the value in the Email element to **abc@ibm.com**.
8. Save the file. It should have no more validation errors, and should look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PersonalInfo xmlns:tns="http://www.addresscontact.com"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.addresscontact.com
ContactInfo.xsd">
  <Address>
    <Street>Street</Street>
    <City>City</City>
    <Country>Country</Country>
  </Address>
  <Name>Name</Name>
  <Gender>Female</Gender>
  <AgeGroup>19 or under</AgeGroup>
  <Phone>333-333-3333</Phone>
  <Email>abc@ibm.com</Email>
</tns:PersonalInfo>
```

The simpleType facets in XML Schema are extremely powerful, especially when used with patterns and restrictions. They enable you to virtually create any kind of data types. Rational Application Developer provides additional support to validate the XML document against the XML Schema. Table 5-1 summarizes the different types of XSD elements and pattern facets.

Table 5-1: Common SimpleType Facets and Pattern Facets

XSD Example	XML Example and Explanation
Enumeration `<element name="Food"> <simpleType> <restriction base="string"> <enumeration value="Pizza" /> <enumeration value="Pasta" /> <enumeration value="Wings" /> </restriction> </simpleType> </element>`	The valid values for the “food” element are Pizza, Pasta, and Wings. `<Food>Wings</Food>`
FractionDigits `<element name= "Price"> <simpleType> <restriction base="decimal"> <fractionDigits value="2"/> </restriction> </simpleType> </element>`	This specifies the maximum number of decimal places that are allowed. For example, 23.1234 would not be a valid value. `<Price>2.34</Price>`
TotalDigits `<element name="Price"> <simpleType> <restriction base="decimal"> <totalDigits value="2"/> </restriction> </simpleType> </element>`	This specifies the maximum number of total digits to be two. For example, 2.3 or 23 would both be valid. `<Price>2.3</Price>`
Length	This specifies the exact length of the element.
maxLength and minLength	maxLength specifies the maximum length of the value. minLength specifies the minimum length.
whiteSpace `<simpleType> <restriction base="string"> <whiteSpace value="collapse"/> </restriction> </simpleType>`	The whiteSpace restriction can have three values: 1) “preserve”—Preserve all the white space. 2) “replace”—Replace all the white space with spaces. 3) “collapse”—Replace all white space with spaces, remove all leading and trailing spaces, and collapse multiple spaces.

Table 5-1: Common SimpleType Facets and Pattern Facets (continued)

Pattern Example	Explanation
[a-zA-Z0-9]	A valid value is one lowercase letter, uppercase letter, or any digit.
[0-9][a-z]	A valid value requires a digit to be followed by a lowercase letter.
[xyz]	A valid value is *x, y*, or z.
([a-z])*	A valid value can contain any length of lowercase letters.
([a-z][A-Z])+	A valid value requires at least one occurrence of one lowercase letter, followed by an uppercase letter.
[a-z]{8}	A valid value contains exactly eight lowercase letters.

Tutorial 2: Using the Java API for XML Processing (JAXP)

In the previous tutorial, we examined the XML Schema editor in Rational Application Developer v6 and used the XSD file created to generate an XML document. To read and manipulate XML documents, you need to use *XML parsers*. Rational Application Developer includes a parser that is based on the Xerces open-source project on Apache. There are over a dozen XML parser implementations available from various vendors, and Xerces is one of the common ones.

Two standards of Application Programming Interfaces (APIs) can process an XML document in Java: *Simple API for XML* (*SAX*) and *Document Object Model* (*DOM*). The SAX API is event-driven. It provides a DefaultHandler handler class that your XML application can implement. Once implemented, the XML application will act as a listener, listening to events that are sent by the SAX parser. Examples of these events are startElement() indicating the beginning of an element, endElement() indicating the end of an element, and startDocument() indicating the beginning of an XML document.

Unlike the SAX API, the DOM API can create and read XML documents. It creates an in-memory tree representation of the XML document. You can manipulate the Document object by invoking methods on it. The DOM API loads the XML document into memory, so it is more memory-intensive than the SAX API. However, it allows applications to have access to the complete XML document and provides tree-based APIs for easy manipulation.

As mentioned previously, many different XML parser implementations are provided by various vendors. Each of these implementations can be a bit different to work with. Therefore, code written for a particular parser might not be portable to another parser. The solution to that is *Java API for XML Processing* (*JAXP*). JAXP provides vendor–implementation-independent interfaces for using XML parsers. It contains a set of high-level interfaces that can work with any parser implementation. Using it, you can easily swap the underlying vendor implementation by changing the setting of certain properties.

Rational Application Developer v6 supports JAXP 1.2.4, which has Xerces as the default implementation. You can configure JAXP to use different parsers by performing one of the three actions listed below:

- Modify the javax.xml.parsers.DocumentBuilderFactory system property.
- Modify the lib/jaxp.properties file.
- Modify the METAINF/services/javax.xml.parsers. DocumentBuilderFactory file.

JAXP 1.2.4 supports DOM Level 2. In DOM 2, you cannot directly save a DOM tree into an output stream or a file. However, a workaround is available, as you will see later in this tutorial. This problem should be solved in DOM Level 3 with the Load and Save functionality.

In this tutorial, you will create a Java application that uses the SAX parser, and modify an existing XML document using the DOM parser. Before following the steps below, if you have not already created the project from the first tutorial, create a new Java project named Ch5XMLProject.

Step 1: Create a New Package

Create a Java package named *xmlparsers* in *Ch5XMLProject*.

1. In the Java perspective, right-click **Ch5XMLProject** and click **New => Package**.
2. Enter **xmlparsers** as the package name. Click **Finish**.

Step 2: Create a New XML and XSD File

Create a new XSD and XML file in the xmlparsers folder. You will parse this XML file using JAXP in the next step.

1. Right-click the **xmlparsers** folder and click **New => Other**.
2. Expand **XML => XML Schema**. Click **Next**.
3. Enter **PhoneBook.xsd** as the file name. Click **Finish**.
4. Modify the XSD file (using the Source view) with the following code:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.ibm.com/PhoneBook"
      xmlns:tns="http://www.ibm.com/PhoneBook">
  <element name="PhoneBook">
    <complexType>
       <sequence>
          <element name="PhoneEntry" maxOccurs="unbounded">
            <complexType>
              <sequence>
                 <element name="Name" type="string" />
                 <element name="Phone" type="string" />
              </sequence>
            </complexType>
          </element>
       </sequence>
    </complexType>
  </element>

</schema>
```

5. Save and close the XML Schema editor. This XSD describes an element named PhoneBook and can contain an unbounded number of the PhoneEntry element. Alternatively, you can create the XSD using the Graph view, as shown in Figure 5-16.

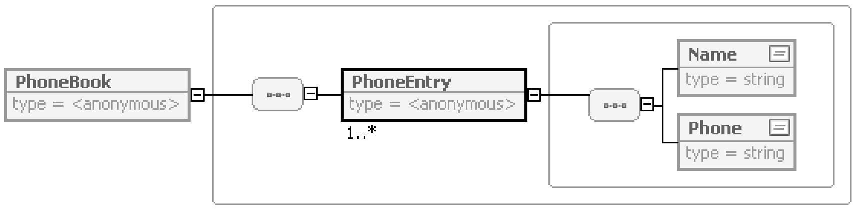


Figure 5-16: PhoneBook.xsd in the Graph view.

6. In the Package Explorer view, right-click **PhoneBook.xsd** and click **Generate => XML File**. Click **Next**, and then click **Finish**.
7. Modify the XML file with the following text to add more entries. Save the file and close the editor.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PhoneBook xmlns:tns="http://www.ibm.com/PhoneBook"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd">
  <PhoneEntry>
    <Name>Mary</Name>
    <Phone>111-1111</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>Jane</Name>
    <Phone>555-5555</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>John</Name>
    <Phone>777-7777</Phone>
  </PhoneEntry>
</tns:PhoneBook>
```

Step 3: Create a SAX Parser

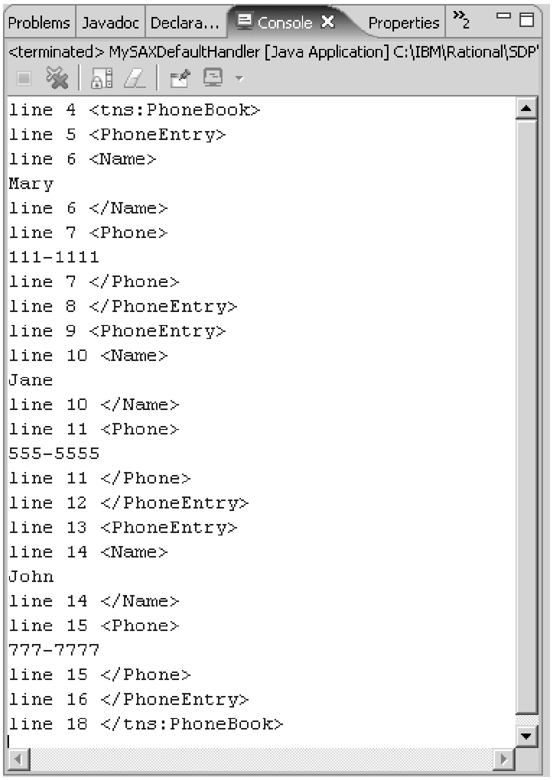
1. In the Package Explorer view, right-click the **xmlparsers** folder and click **New => Class**.
2. Enter **MySAXDefaultHandler** as the class name.
3. Browse for **org.xml.sax.helpers.DefaultHandler** as the superclass. Click **OK**.
4. Select the check box **public static void main (String[] args)**. Click **Finish**.
5. Modify the class file with the following code, and save the file:

```
package xmlparsers;

import java.io.File;

import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class MySAXDefaultHandler extends DefaultHandler {

   static private Locator documentLocator;

   public static void main(String[] args) {
        run();
   }
   public static void run () {
        DefaultHandler handler = new MySAXDefaultHandler();
        SAXParserFactory factory = SAXParserFactory.newInstance();
        factory.setValidating(true);
        try {
        // Parse the input
        SAXParser saxParser = factory.newSAXParser();
        saxParser.parse(new File("xmlparsers/PhoneBook.xml"), handler);
        } catch (Throwable t) {
        t.printStackTrace();
        }
   }
```

```
    public void setDocumentLocator(Locator locator) {
        documentLocator = locator;
    }

    public void startDocument() throws SAXException {}
    public void endDocument() throws SAXException {}

    public void startElement(String namespaceURI, String lName,
        String qName, Attributes attrs) throws SAXException {
        int line = documentLocator.getLineNumber();
        System.out.println("line " + line + " <" + qName +">");
    }

    public void endElement(String namespaceURI, String sName,
        String qName) throws SAXException {
        int line = documentLocator.getLineNumber();
        System.out.println("line " + line + " </" + qName +">");

    }

    public void characters(char[] chars, int arg0, int arg1)
        throws SAXException {
        String result = new String(chars, arg0, arg1);
        result = result.trim();
        if (!result.equals("")) {
              System.out.println( result);
        }
    }
}
```

This Java application echoes the elements in the PhoneBook.xml file. The Java application acting as a SAX handler would process the XML element event. The Locator allows you to get the line position of the XML file. You can also obtain the column position using the Locator. You can use Organize Imports to fix any compilation errors. If you are given a choice of different SAX parsers when doing Organize Imports, select the one with javax.xml.parsers as the package name. For the others, refer to the import statements for the correct packages.

6. Run the code as a Java application. Right-click **MySAXDefaultHandler** and click **Run => Java Application**. The output looks like Figure 5-17. The startElement() method is called when an element start tag is encountered. The characters() method is invoked when there are characters embedded between the start and end element tags.



```
<terminated> MySAXDefaultHandler [Java Application] C:\IBM\Rational\SDP'
line 4 <tns:PhoneBook>
line 5 <PhoneEntry>
line 6 <Name>
Mary
line 6 </Name>
line 7 <Phone>
111-1111
line 7 </Phone>
line 8 </PhoneEntry>
line 9 <PhoneEntry>
line 10 <Name>
Jane
line 10 </Name>
line 11 <Phone>
555-5555
line 11 </Phone>
line 12 </PhoneEntry>
line 13 <PhoneEntry>
line 14 <Name>
John
line 14 </Name>
line 15 <Phone>
777-7777
line 15 </Phone>
line 16 </PhoneEntry>
line 18 </tns:PhoneBook>
```

Figure 5-17: Running MySAXDefaultHandler output.

Step 4: Create a DOM Parser

In this step, you parse the XML document using the DOM tree. Create a new class named *MyDOMParser*:

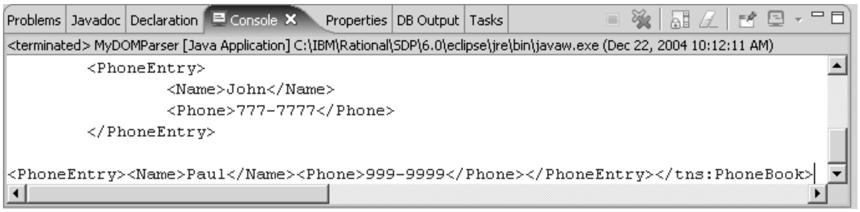
1. Right-click the **xmlparsers** folder and click **New => Class**.
2. Enter **MyDOMParser** as the class name.
3. Select the **public static void main (String[] args)** check box. Click **Finish**.
4. Modify the class file with the following code, and then save the file:

```
package xmlparsers;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.Source;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.Document;
import org.w3c.dom.Element;
import org.w3c.dom.Node;
import org.w3c.dom.Text;

public class MyDOMParser {

  public static void main(String[] args) {
     try {
     String filename = “xmlparsers/PhoneBook.xml”;
     DocumentBuilderFactory dbf =
     DocumentBuilderFactory.newInstance();
     dbf.setValidating(false);

     DocumentBuilder db = dbf.newDocumentBuilder();
     Document doc = db.parse(filename);
     Node root = doc.getDocumentElement();
     Element phoneentry = doc.createElement(“PhoneEntry”);
     Element name = doc.createElement(“Name”);
     Text paul = doc.createTextNode(“Paul”);
     Element phone = doc.createElement(“Phone”);
     Text number = doc.createTextNode(“999-9999”);

     root.appendChild(phoneentry);
     phoneentry.appendChild(name);
     name.appendChild(paul);
     phoneentry.appendChild(phone);
     phone.appendChild(number);

     // Workaround for DOM 2 - write to a file or output stream
     TransformerFactory factory =
     TransformerFactory.newInstance();
     Transformer transformer = factory.newTransformer();
     Source source = new DOMSource (doc);
     DOMResult result = new DOMResult ();
     StreamResult out_result = new StreamResult(System.out);
     transformer.transform(source, out_result);
```

```
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

The application parses PhoneBook.xml into a Document object. It adds a new element to the Document, and then prints the Document object to System.out. As mentioned previously, DOM Level 2 does not support directly serialization of the Document object. Therefore, a workaround has been used to print the XML document you see in the last part of the *main()* method.

5. Run the code as a Java application. Right-click **MyDOMParser** and click **Run => Java Application**. The output of the program is shown in Figure 5-18. Note that a new <PhoneEntry> has been added to the XML document.



```
<terminated> MyDOMParser [Java Application] C:\IBM\Rational\SDP\6.0\eclipse\jre\bin\javaw.exe (Dec 22, 2004 10:12:11 AM)
        <PhoneEntry>
                <Name>John</Name>
                <Phone>777-7777</Phone>
        </PhoneEntry>

<PhoneEntry><Name>Paul</Name><Phone>999-9999</Phone></PhoneEntry></tns:PhoneBook>
```

Figure 5-18: Running MyDOMParser.

Both the SAX and DOM parsers are easy to use. Depending on your application, you may choose to use one over another. A basic rule of thumb is that if you have a large amount of information to parse and only need to store a small amount of data, consider the SAX parser. However, if you need to manipulate the XML document, the DOM parser might be more convenient.

In addition to using the DOM and SAX parsers, there is a new way to use SDO to access XML document, which IBM is embracing. You will see more about SDO and how to use it with XML in tutorial 5.

Tutorial 3: Using Extensible Stylesheet Language Transformations (XSLT)

This tutorial introduces you to *Extensible Stylesheet Language Transformations* (*XSLT*) and *XPath expression*. You will create an XSL style sheet with the XSLT editor and use it to transform an XML document into a text document. This tutorial also shows you how to dynamically transform an XML document into HTML using a Java servlet, for display in a Web browser.

XSLT is a transformation language that uses a style sheet to describe how a document can be transformed into a different format. XSLT uses XPath expressions to select values from an input XML source, and performs processing to produce a desired output.

XPath is a query language for locating nodes, values, and attributes in XML documents. Using the following XML document as an example, the XPath expression /tns:PhoneBook will select the ROOT element PhoneBook. The XPath expression /tns:PhoneBook/PhoneEntry/Name will select all the Name elements in all the PhoneEntry elements, returning three Name elements.

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PhoneBook>
  <PhoneEntry>
    <Name>Mary</Name>
    <Phone>111-1111</Phone>
    <Age>30</Age>
  </PhoneEntry>
  <PhoneEntry>
    <Name>Jane</Name>
    <Phone>555-5555</Phone>
    <Age>25</Age>
  </PhoneEntry>
  <PhoneEntry>
    <Name>John</Name>
    <Address> 123 John St.</Address>
  </PhoneEntry>
</tns:PhoneBook>
```

XPath is extremely powerful. It can select virtually any elements. Table 5-2 lists some more advanced XPath expression examples.

Table 5-2: Advanced XPath Expression Examples

XPath Expression Example	Explanation
/tns:PhoneBook/PhoneEntry	This XPath will return all PhoneEntry elements.
/tns:PhoneBook/PhoneEntry[Age>28]	This will return all the PhoneEntry elements where Age is greater than 28.
/tns:PhoneBook/PhoneEntry[Address]	This will return all the PhoneEntry elements that have an Address element.
/tns:PhoneBook[1]	This will select the first child of the PhoneBook element, which is the first PhoneEntry element.
//Name	This will select any occurrences of the Name element, regardless of the tree hierarchy. A single slash is an absolute path. A double slash means to select any elements that fulfill the criteria (i.e., Name), regardless of the tree level.
/tns:PhoneBook/PhoneEntry/*	An asterisk (*) is a wildcard. This example will select all the elements under the PhoneEntry: Name, Phone, Age, and Address.

Xpath expressions are used in XSL style sheets to select particular elements or attributes to perform some actions. For example, in the code in Figure 5-19, the line <xsl:value-of select="Name" /> would select the value of the Name element. The word "Name" inside the quotation marks is an XPath expression. As another example, consider the line <xsl:for-each select="tns:PhoneBook/PhoneEntry"> in Figure 5-19. It means the style sheet will do something for each PhoneEntry element inside the PhoneBook element.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="http://xml.apache.org/xslt"
xmlns:tns="http://www.ibm.com/PhoneBook"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd">
        <xsl:template match="/">
               PhoneBook Directory
               <xsl:for-each select="tns:PhoneBook/PhoneEntry">
                       Name: <xsl:value-of select="Name" />
                       Phone: <xsl:value-of select="Phone" />
               </xsl:for-each>
        </xsl:template>
</xsl:stylesheet>
```

Figure 5-19: A simple XSL style sheet.

Figure 5-19 is an *inlined* XSL style sheet, meaning all the operations are embedded in one *template*. A template is a set of rules that make up an XSL style sheet. The corresponding tag is <xsl:template>:

```
<xsl:template match="Some XPath Expression">
        [actions]
</xsl:template>
```

The *match* attribute defines an XPath expression that will be used as a pattern, to match for nodes in XML documents. If a match is found, the template will apply the actions listed inside it. For example, in Figure 5-19, the match attribute is "/," which will match the root element. When the root element is found, the template will apply the actions <xsl:for-each> and <xsl:value-of>. Since Figure 5-19 is an inlined template, all the actions are performed in the root template. Alternatively, the same XSL can be modeled using separate templates, as shown in the following code, where PhoneEntry, Name, and Phone are in separate templates:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="http://xml.apache.org/xslt"
xmlns:tns="http://www.ibm.com/PhoneBook"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd ">
        <xsl:template match="/">
              PhoneBook Directory
              <xsl:apply-templates></xsl:apply-templates>
        </xsl:template>

        <xsl:template match="tns:PhoneBook/PhoneEntry">
              <xsl:apply-templates></xsl:apply-templates>
        </xsl:template>

        <xsl:template match="tns:PhoneBook/PhoneEntry/Name">
              Name: <xsl:value-of select="Name" />
        </xsl:template>

        <xsl:template match="tns:PhoneBook/PhoneEntry/Phone">
              Phone: <xsl:value-of select="Phone" />
        </xsl:template>
</xsl:stylesheet>
```

As the XSL parser parses through the XML document, when it encounters the root element, it will use the root template. In the root template, the parser is instructed to visit each child under root and apply template rules to them. When the parser gets to the PhoneEntry node, the same <xsl:apply-templates/> instruction tells the parser to visit each child under the PhoneEntry node and apply template rules to them. Finally, the parser will visit the Name and Phone node. Each has its own template that would print out its element value.

In Rational Application Developer, XSL style sheet can be created and edited in the XSL editor. Since XPath expressions are used in style sheets, the XSL editor provides an *XPath Expression Builder* that has support for building XPath expressions visually. In addition, you can preview the result of your XPath expression, given an actual XML document. Rational Application Developer also includes an XSL debugger that you can use to debug XSL style sheets.

After an XSL style sheet is created, you can test it quickly by running it as an XSL transformation. Simply right-click an XML and the XSL style sheet, and select **Run => XSL Transformation**. An XML document will be created as the result; it usually has a name that starts with an underscore.

You can use JAXP to transform an XSL style sheet dynamically in Java applications. APIs are available in JAXP to perform the transform. Figure 5-20 shows that XSL can be used to transform an XML document to produce another document format, such as HTML, XML, or text.

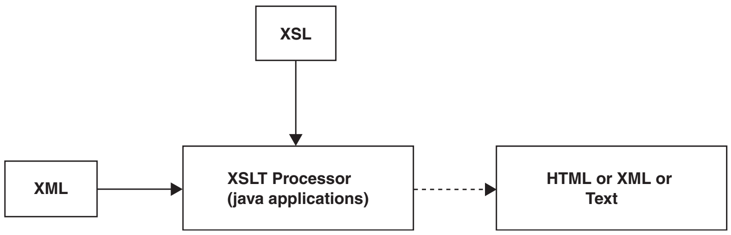


Figure 5-20: XSLT transformations in Java applications.

In this tutorial, you perform the following tasks:

- Create and test your XSL style sheet using the XSL transformation tools.
- Create a Java application using JAXP that can dynamically transform XSL style sheets.

- Create a servlet that can dynamically transform an XML document into HTML.

If you have not already created the project Ch5XMLProject, do so before going on to the first step of this tutorial.

Step 1: Create a New Package

Create a package named *xsl* in Ch5XMLProject:

1. In the Java perspective, right-click **Ch5XMLProject** and click **New => Package**.
2. Enter **xsl** as the package name. Click **Finish**

Step 2: Copy or Create XSD and XML Files

If you have created the PhoneBook.xsd and PhoneBook.xml files in tutorial 2, copy them from the xmlparser folder to the xsl folder and proceed to step 3. If not, create the two files in the xsl folder, using the following procedure:

1. Right-click the **xsl** folder and click **New => Other**. Expand **XML => XML Schema**. Click **Next**. Enter **PhoneBook.xsd** as the file name. Click **Finish**.
2. Modify the file with the following code, and save the file.:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.ibm.com/PhoneBook"
        xmlns:tns="http://www.ibm.com/PhoneBook">
  <element name="PhoneBook">
    <complexType>
        <sequence>
          <element name="PhoneEntry" maxOccurs="unbounded">
            <complexType>
              <sequence>
                <element name="Name" type="string" />
                <element name="Phone" type="string" />
              </sequence>
            </complexType>
          </element>
        </sequence>
    </complexType>
  </element>
</schema>
```

3. In the Package Explorer view, right-click **PhoneBook.xsd** and click **Generate => XML File**. Click **Next**, and then **Finish**. Modify the XML file with the following text to add more entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PhoneBook xmlns:tns="http://www.ibm.com/PhoneBook"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd">
  <PhoneEntry>
    <Name>Mary</Name>
    <Phone>111-1111</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>Jane</Name>
    <Phone>555-5555</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>John</Name>
    <Phone>777-7777</Phone>
  </PhoneEntry>
</tns:PhoneBook>
```

4. Save the file and close the editor.

Step 3: Create an XSL File

1. Right-click the **xsl** folder and click **New => Other**.
2. Expand **XML => XSL**. Click **Next**.
3. Enter **TransformPhoneBook.xsl** as the file name. Click **Next**.
4. Select **Ch5XMLProject/xsl/PhoneBook.xml** and click **Finish**.
5. Edit the namespace of the XSL editor. In the Outline view, right-click **xsl:stylesheet** and select **Edit Namespaces** as shown in Figure 5-21.

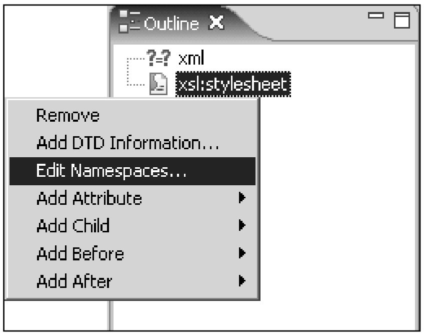
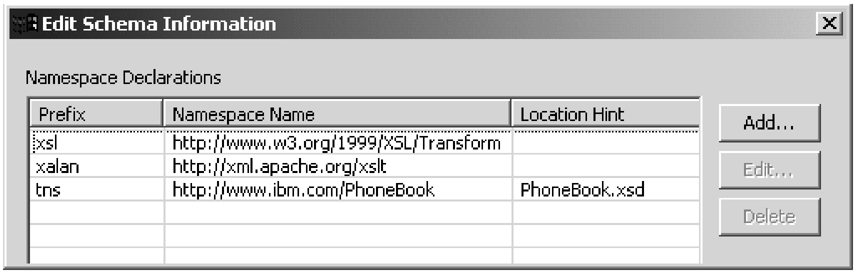


Figure 5-21: The Edit Namespace pop-up menu.

6. Click **Add** to add a new namespace. Select the **Specify New Namespace** radio box.
7. Enter **tns** as the prefix and **http://www.ibm.com/PhoneBook**, as the namespace name. Click **Browse** to select the file Ch5XMLProject/xsl/PhoneBook.xsd. This will enable you to access the elements in PhoneBook.xsd where the namespace is http://www.ibm.com/PhoneBook, as shown in Figure 5-22.
8. Click **OK** to finish the dialog box.
9. Click **OK** again to close the Edit Schema Information dialog box.



Prefix	Namespace Name	Location Hint
xsl	http://www.w3.org/1999/XSL/Transform	
xalan	http://xml.apache.org/xslt	
tns	http://www.ibm.com/PhoneBook	PhoneBook.xsd

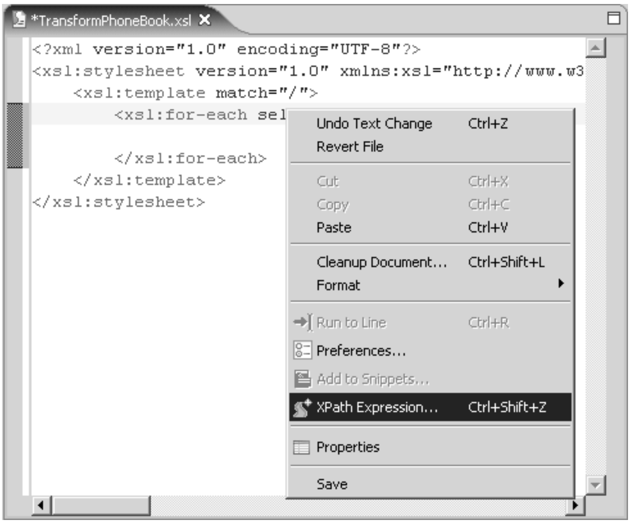
Figure 5-22: Namespaces for the style sheet.

Step 4: Edit the XSL

1. In the Outline view, right-click **xsl:stylesheet** and select **Add Child => xsl:template**. This will add the <xsl:template> tag to the editor.
2. In the editor, remove the line <xsl:apply-templates></xsl:apply-templates>, since you will create all the actions inline.
3. Place the cursor in the xsl:template tag. In the Properties view, enter / as the match:

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

4. Right-click the slash (/) node in the Outline view and select **Add Child => xsl:for-each**. Once again, remove the line <xsl:apply-templates></xsl:apply-templates> in the editor.
5. Right-click the **<xsl:for-each>** tag in the editor and select **XPath Expression**, as shown in Figure 5-23.



```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3
    <xsl:template match="/">
        <xsl:for-each sel

        </xsl:for-each>
    </xsl:template>
</xsl:stylesheet>
```

Figure 5-23: XPath Expression Builder from the pop-up menu.

6. Expand **tns:PhoneBook** in the Source Tree view of the XPath Expression Builder and select the first PhoneEntry node. Drag PhoneEntry node to the text area on top of the dialog box. Click **OK** to close the XPath Expression Builder. The <xsl:for-each> tag should look like the following:

```
<xsl:for-each select=“tns:PhoneBook/PhoneEntry”>
```

7. In the Outline view, right-click **xsl:for-each** and select **Add Child => xsl:value-of**.
8. Right-click the **<xsl:value-of>** tag in the editor and select **XPath Expression**.
9. Enter **/tns:PhoneBook/PhoneEntry** in the Evaluation Context field. The <xsl:value-of> tag is inside a selection of PhoneEntry; therefore, the evaluation context needs to be adjusted.
10. Expand **tns:PhoneBook** in the Source Tree view, and expand any one of the PhoneEntry nodes. Select **Name [Mary]** and drag it to the text area on top, as shown in Figure 5-24. Click **OK** to close the builder. The <xsl:value-of> tag should look like this:

```
Name: <xsl:value-of select=“Name”/>
```

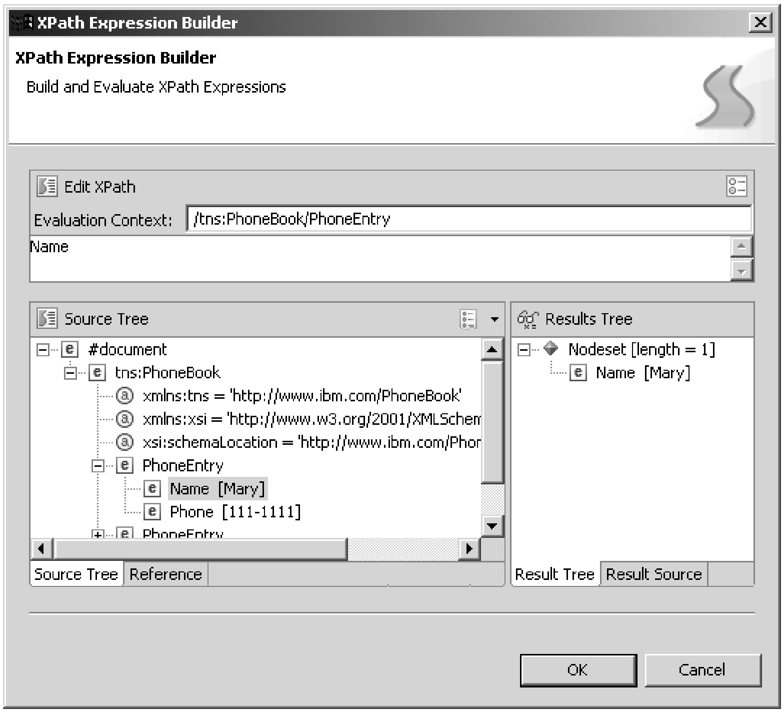


Figure 5-24: XPath Expression Builder for selecting the value of Name.

11. Repeat steps 7 and 8 to create another <xsl:value-of> under <xsl:for-each> for selecting Phone.
12. In the XPath Expression Builder, enter **/tns:PhoneBook/PhoneEntry** in the Evaluation Context field. Expand **tns:PhoneBook**, and select **PhoneEntry.** Select **Phone [111-1111]** and drag it to the text area on top. Click **OK**. The <value-of> tag should look like this:

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

13. Add the bolded labels shown below in the editor, and save the file. The final XSL file should looks like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
xmlns:xalan="http://xml.apache.org/xslt"
xmlns:tns="http://www.ibm.com/PhoneBook"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd ">
       <xsl:template match="/">
              PhoneBook Directory
              <xsl:for-each select="tns:PhoneBook/PhoneEntry">
                     Name: <xsl:value-of select="Name" />
                     Phone: <xsl:value-of select="Phone" />
              </xsl:for-each>
       </xsl:template>
</xsl:stylesheet>
```

Step 5: Test the XSL Transformation

1. In the Navigator view, select both the **PhoneBook.xml** and **TransformPhoneBook.xsl** files, right-click, and click **Run => XSL Transformation**.
2. The XSL Transformation tool will apply the TransformPhoneBook.xsl file to PhoneBook.xml, and put the results into a file named *_PhoneBook_transform.xml*.
3. Open _PhoneBook_transform.xml with the XML editor. As you can see in the following results, the XSLT will select the Name and Phone values for each PhoneEntry element in the XML file:

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

       Name: Mary
       Phone: 111-1111
       Name: Jane
       Phone: 555-5555
       Name: John
       Phone: 777-7777
```

Step 6: Write an XSL Transformation Script Using JAXP

In the previous step, you tested the XSL transformation process using the built-in tooling. This step demonstrates how to transform files dynamically.

1. In the Navigator view, right-click the **xsl** folder and click **New => Class**.
2. Enter **MyXSLTransformer** as the class name.
3. Select the **public static void main (String[] args)** check box. Click **Finish**.
4. Modify the class file with the following code, and then save the file and close the editor:

```
package xsl;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class MyXSLTransformer {

   public static void main(String[] args) {
        try {
          TransformerFactory factory =
          TransformerFactory.newInstance();
          StreamSource stylesheet =
          new StreamSource ("xsl/TransformPhoneBook.xsl");
          Transformer transformer =
          factory.newTransformer(stylesheet);
          StreamSource source = new StreamSource
          ("xsl/PhoneBook.xml");
          DOMResult result = new DOMResult ();
          StreamResult out_result = new
          StreamResult(System.out);
          transformer.transform(source, out_result);
        } catch (Exception e) {
              e.printStackTrace();
        }
   }
}
```

5. Run the file as a Java application by right-clicking **MyXSLTranform** and clicking **Run => Java Application**. The output shown below should be in the Console view:

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

        Name: Mary
        Phone: 111-1111
        Name: Jane
        Phone: 555-5555
        Name: John
        Phone: 777-7777
```

Step 7: Transform XML Documents into HTML

As seen from the previous step, it is very simple to transform an XML document with XSLT in Java applications. Similarly, it is also quite easy to transform an XML document into HTML in a Java servlet.

1. Create an EAR and a Web project named *Ch5XMLEAR* and *Ch5XMLEARWeb*, respectively.
2. Create a folder in the WebContent folder. In the Web perspective's Package Explorer view, right-click the **Ch5XMLEARWeb/WebContent** folder and select **New => Folder**. Enter **xsl** as the name. Click **Finish**.
3. Copy these three files from Ch5XMLProject/xsl to this /WebContent/xsl folder: PhoneBook.xml, PhoneBook.xsd, and TransformPhoneBook.xsl.
4. Open TransformPhoneBook.xsl in the editor. Modify the code as follows, to produce an HTML document instead of just a text document:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns:xalan="http://xml.apache.org/xslt"
        xmlns:tns="http://www.ibm.com/PhoneBook"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ibm.com/PhoneBook
PhoneBook.xsd">
        <xsl:template match="/">
        <html>
               <h1>PhoneBook Directory</h1>
```

```
                <table cellpadding="3" cellspacing="1" width="50%">
                 <xsl:for-each select="tns:PhoneBook/PhoneEntry">
                 <tr>
                        <td bgcolor="cccccc"><b>Name:</b></td>
                        <td bgcolor="ff9999">
                              <xsl:value-of select="Name" />
                        </td>
                 </tr>
                 <tr>
                        <td bgcolor="cccccc"><b>Phone:</b></td>
                        <td bgcolor="ff9999">
                              <xsl:value-of select="Phone" />
                        </td>
                 </tr>
                 <tr><td colspan="2"><br/></td></tr>

                 </xsl:for-each>
                 </table>
          </html>
          </xsl:template>
</xsl:stylesheet>
```

5. Right-click **Ch5XMLEARWeb/Java Resources/JavaSource** folder and select **New => Other**. Select **Web => Servlet**. Click **Next**.

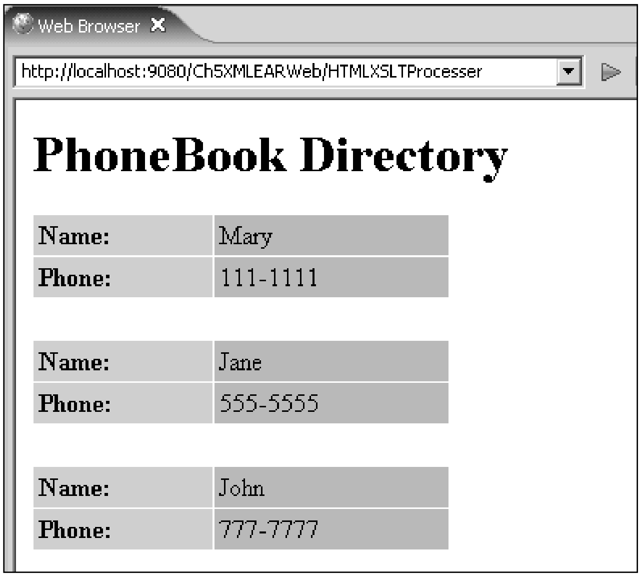
6. Enter **HTMLXSLTProcessor** as the name and click **Finish**.

7. Modify the servlet as follows, and save it:

```
package xsl;

import java.io.IOException;
import java.io.InputStream;
import java.io.PrintWriter;

import javax.servlet.Servlet;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMResult;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;
```

```
public class HTMLXSLTProcessor extends HttpServlet implements Servlet {

       public HTMLXSLTProcessor() {
              super();
       }
       protected void doGet(HttpServletRequest arg0,
       HttpServletResponse arg1) throws ServletException, IOException {
              performTask (arg0, arg1);
       }
       protected void doPost(HttpServletRequest arg0,
       HttpServletResponse arg1) throws ServletException, IOException {
              performTask (arg0, arg1);
       }
  protected void performTask(HttpServletRequest req,
       HttpServletResponse resp){
    try {
       PrintWriter out = resp.getWriter();

       InputStream xslStream =
       getServletContext().getResourceAsStream
        ("/xsl/TransformPhoneBook.xsl");

       InputStream xmlStream =
       getServletContext().getResourceAsStream
        ("/xsl/PhoneBook.xml");

       TransformerFactory factory = TransformerFactory.newInstance();
       StreamSource stylesheet = new StreamSource (xslStream);
       Transformer transformer = factory.newTransformer(stylesheet);
       StreamSource source = new StreamSource (xmlStream);
       DOMResult result = new DOMResult ();
       StreamResult out_result = new StreamResult(out);
       transformer.transform(source, out_result);
    } catch (Exception e) {
       e.printStackTrace();
    }
  }

}
```

The servlet takes the PhoneBook.xml file as the XML source, and applies the TransformPhoneBook.xsl to it. To run the servlet, right-click **HTMLXSLProcessor** and select **Run => Run on Server**. You should see Figure 5-25 loaded in the browser, where the XML document has been transform to HTML contents.



Web Browser

http://localhost:9080/Ch5XMLEARWeb/HTMLXSLTProcesser

PhoneBook Directory

Name:	Mary
Phone:	111-1111

Name:	Jane
Phone:	555-5555

Name:	John
Phone:	777-7777

Figure 5-25: Transforming an XML document into HTML in a servlet.

Tutorial 4: Using XML with SQL

In this tutorial, we will look at a Rational Application Developer tool that provides the functionality to directly connect database data with XML. This tool, *SQLToXML*, is able to translate XML into database data and vice versa.

The SQLToXML function works for all the databases supported by Rational Application Developer. In this tutorial, you will generate an XST file from a database schema. The XST file stores all the information required to use the SQLToXML runtime and contains database configuration information. Then, you will write a Java application that uses the SQLToXML runtime to dynamically obtain database data in an XML format, using the generated template.

You will perform the following tasks in the tutorial:

- Generate a DDL script from the XML Schema document to create tables in the database.
- Insert data from XML directly into a database.

- Create an XST template from a SELECT statement.
- Use a Java program to call the XST template to get the database data dynamically.

If you have not already created the Java project Ch5XMLProject, create it before proceeding.

Step 1: Create a New Package

Create a package named *xst* in Ch5XMLProject:

1. In the Java perspective, right-click **Ch5XMLProject** and click **New => Package**.
2. Enter **xst** as the package name. Click **Finish**.

Step 2: Copy or Create XSD Files

If you have created the PhoneBook.xsd file in tutorial 2, copy it from the xmlparser folder to the xst folder. If not, create the PhoneBook.xsd file in the xst folder:

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://www.ibm.com/PhoneBook"
        xmlns:tns="http://www.ibm.com/PhoneBook">
  <element name="PhoneBook">
    <complexType>
        <sequence>
          <element name="PhoneEntry" maxOccurs="unbounded">
            <complexType>
              <sequence>
                  <element name="Name" type="string" />
                  <element name="Phone" type="string" />
              </sequence>
            </complexType>
          </element>
        </sequence>
    </complexType>
  </element>
</schema>
```

Step 3: Create a Database Connection

Create a connection to the database of your choice. You can create a new connection in the Database Explorer view in the Data perspective and follow the wizard to connect to your database. Refer to tutorial 1 in chapter 4 for more details. You will use this database connection later to create the table and insert the data.

Step 4: Generate a DDL from the XSD and Create a Table

A DDL script is one that can be run on a database to perform database actions. In this step, you generate a DDL from PhoneBook.xsd and run it against your database to create the table.

1. In the Java perspective's Package Explorer view, right-click **xst/PhoneBook.xsd** and select **Generate => DDL**.
2. Select **Ch5XMLProject/xst** as the parent folder. Enter **PhoneBook.sql** as the file name. Click **Finish**.
3. Open PhoneBook.sql in an editor. The DDL file created is only for your reference. Notice that it has an INTEGER field for an ID, which did not exist in the XSD originally.
4. Right-click **PhoneBook.sql** and select **Deploy**. Click **Next** twice, until you get to the Database Connection page.
5. Select the **Use Existing Connection** check box, and select **your connection** from the Existing Connection drop-down box. Click **Finish**. This creates a table in the database.
6. Switch to the Data perspective, right-click your connection, and select **Refresh**. You should see the PhoneBook table in the database connection, as shown in Figure 5-26.

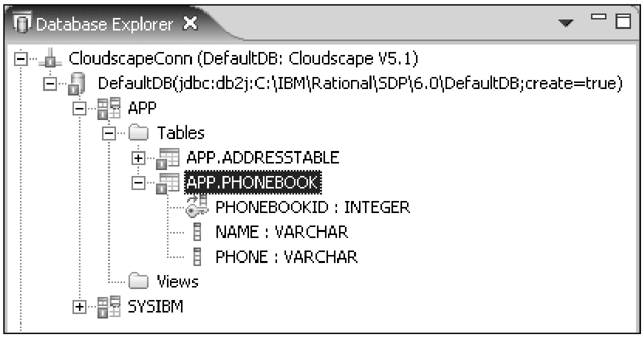


Figure 5-26: The PhoneBook table in the database connection.

Step 5: Create Insert and Select Statements

Copy the table schema to a Ch5XMLProject project:

1. In the Database Explorer view, expand your connection and database until you see the tables. Right-click the **APP.PHONEBOOK** table and select **Copy to Project**.
2. Click **Browse** and select **Ch5XMLProject**. Click **OK**.
3. Click **Finish**. Click **Yes** when asked to create the folder and schema.

Create an Insert statement named *Insert1*:

1. In the Data Definition view, right-click the **Statement** folder and select **New => Insert Statement**, as shown in Figure 5-27.

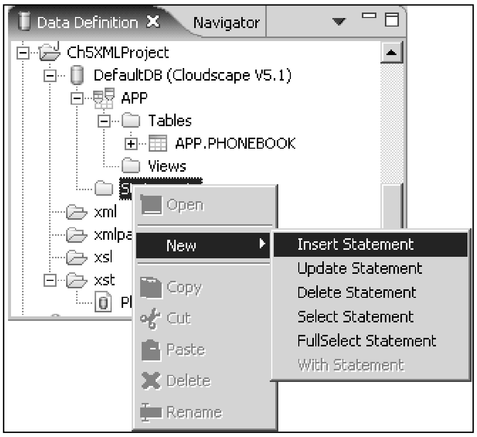
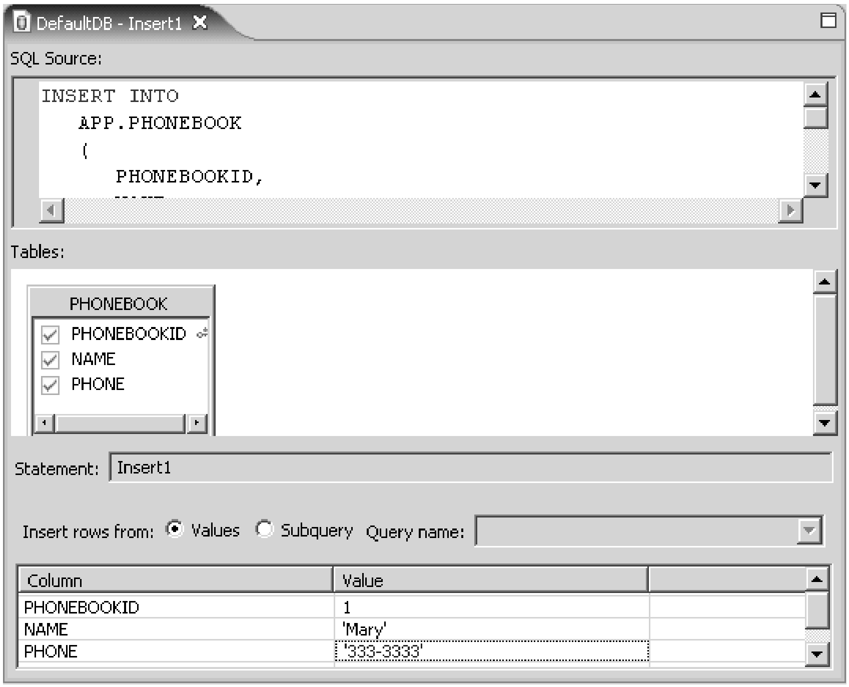


Figure 5-27: The PhoneBook table in Ch5XMLProject.

2. Enter **Insert1** as the name. Click **OK**.
3. Right-click in the Table pane in the editor and select **Add Table**. Select **APP.PHONEBOOK** as the table name and click **OK**.
4. Select all the check boxes inside the table.
5. Enter **1** as the value of the PHONEBOOKID column.
6. Enter **Mary** as the value of the NAME column.
7. Enter **333-3333** as the value of the PHONE column.
8. Save the file.
9. Right-click **Insert1** and select **Execute**. This inserts one row into the table, as shown in Figure 5-28.



```
INSERT INTO
   APP.PHONEBOOK
   (
      PHONEBOOKID,
```

PHONEBOOK: ✓ PHONEBOOKID, ✓ NAME, ✓ PHONE

Statement: Insert1

Insert rows from: [✓ Values] Subquery

Column	Value	
PHONEBOOKID	1	
NAME	'Mary'	
PHONE	'333-3333'	

Figure 5-28: The Insert1 statement.

Create a Select statement named *Select1*:

1. In the Data Definition view, right-click the **Statement** folder and select **New => Select Statement**.
2. Enter **Select1** as the name. Click **OK**.
3. Right-click in the Table pane in the editor and select **Add Table**. Select **APP.PHONEBOOK** as the table name and click **OK**.
4. Select all the check boxes inside the table.
5. Save the file.

Step 6: Generate the XST Template

1. In the Data Definition view, right-click **Select1** statement and click **Generate New XML**.
2. Browse to **/Ch5XMLProject/xst** as the output folder. Click **Finish**.

The wizard will generate XST, HTML, XSD, XML, and XSL files, as shown in Figure 5-29. The Select1.xst template file contains database configuration information that is required to use the SQLToXML runtime. Since the XST file is generated using the Select1 statement, that statement is also embedded in the XST file. The XSD file describes the database schema in XML Schema terms. The XML file is a sample of the database content. The XSL file is a sample XSL transformation file that can transform XML data into HTML. The HTML file is the sample output that was created when the sample XSL file transformed the XML file. The database would use the XST file as a data mapping to convert to an XML file, which you can then use with the XSL to transform into HTML.

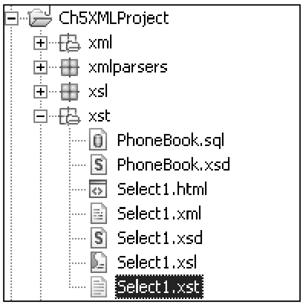


Figure 5-29: The generated XST template.

Step 7: Generate Database Data Directly from XML

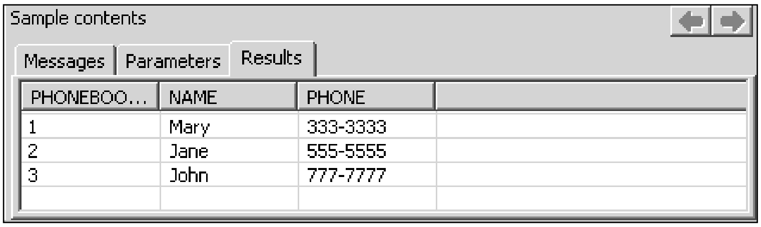
This step exports XML data directly into the PhoneBook table in the database.

1. Open Select1.xml in the editor and modify it as follows, to add more entries:

```
<?xml version="1.0" encoding="UTF-8"?>
<SQLResult xmlns="http://www.ibm.com/PHONEBOOK"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
Xsi:schemaLocation="http://www.ibm.com/PHONEBOOK&#xa;&#x9;Select1.xsd">
    <PHONEBOOK>
        <PHONEBOOKID>2</PHONEBOOKID>
        <NAME>Jane</NAME>
        <PHONE>555-5555</PHONE>
    </PHONEBOOK>
    <PHONEBOOK>
        <PHONEBOOKID>3</PHONEBOOKID>
        <NAME>John</NAME>
        <PHONE>777-7777</PHONE>
    </PHONEBOOK>
</SQLResult>
```

2. Right-click **Select1.xml** and select **Generate => Database data**.
3. Select the **Use Existing Connection** check box, and select **your connection** from the Existing Connection drop-down box. Click **Next**.
4. Click **Finish**. The XML data is written to the database.

5. If you want to see the data, you can *sample contents* via the database connection. Right-click the **APP.PHONEBOOK** table in the Database Explorer view and select **Sample contents**. You should see three rows in the result, as shown in Figure 5-30.



PHONEBOO...	NAME	PHONE	
1	Mary	333-3333	
2	Jane	555-5555	
3	John	777-7777	

Figure 5-30: Sample contents of APP.PHONEBOOK table.

Step 8: Create an SQLToXML Java Application

In the previous step, you exported XML data directory to the database. In this step, you do the opposite: get XML data from database. This will be done dynamically in a Java application, without writing any JDBC code.

1. Right-click the **xst** folder and click **New => Class**.
2. Enter **RunXST** as the class name.
3. Select the **public static void main (String[] args)** check box. Click **Finish**.
4. Modify the class file with the following code:

```
package xst;

import com.ibm.etools.sqltoxml.*;
import java.io.*;

public class RunXST {

   public static void main(String[] args) {
       try {
             com.ibm.etools.sqltoxml.QueryProperties qp
           = new QueryProperties();
             qp.load("xst/Select1.xst");

             SQLToXML sql2xml = new SQLToXML(qp);
             sql2xml.setXSDFile("xst/Select1.xsd");
             PrintWriter outWriter = new PrintWriter(System.out);
             sql2xml.setXMLWriter(outWriter);
             sql2xml.execute();
```

```
        } catch (Exception e) {
              e.printStackTrace();
        }
    }
}
```

5. Save the file and close the editor. This application will use the SQLToXML runtime to obtain the data from the database and display it in System.out as XML. It uses Select1.xsd as the XSL Schema.

Modify the Java build path:

1. Right-click **Ch5XMLProject** and select **Properties**. Select **Java Build Path**.
2. Click **Add External Jars** and browse to the following file:

```
<SDP_install>\6.0\rwd\eclipse\plugins\com.ibm.etools.
sqltoxml_6.0.0\sqlxml.jar
```

3. Also add the JDBC driver of your database to the Java build path. Use Table 5-3 to select the JDBC JAR files for your database.

Table 5-3: JDBC Drivers Summary

DB2	C:\Program Files\IBM\SQLLIB\java\db2jcc.jar C:\Program Files\IBM\SQLLIB\java\db2jcc_license_cisuz.jar
Cloudscape	C:\IBM\Rational\SDP\6.0\runtimes\base_v6\-cloudscape\lib\db2j.jar
Oracle	C:\oracle\ora91\jdbc\lib\classes12.jar
SQL Server	C:\Program Files\Microsoft SQL Server 2000 Driver for JDBC\lib\msbase.jar C:\Program Files\Microsoft SQL Server 2000 Driver for JDBC\lib\mssqlserver.jar C:\Program Files\Microsoft SQL Server 2000 Driver for JDBC\lib\msutil.jar You can download the JDBC drivers for SQL Server from Microsoft ‘s Web site.
Sybase	C:\jConnect-5_5\devclasses\jconn2d.jar

4. Right-click RunXST.java and click **Run => Java Application**. The output looks like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<SQLResult xmlns="http://www.ibm.com/PHONEBOOK"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.ibm.com/PHONEBOOK&#xa;&#x9;
Select1.xsd">
   <PHONEBOOK>
        <PHONEBOOKID>1</PHONEBOOKID>
        <NAME>Mary</NAME>
        <PHONE>333-3333</PHONE>
    </PHONEBOOK>
    <PHONEBOOK>
        <PHONEBOOKID>2</PHONEBOOKID>
        <NAME>Jane</NAME>
        <PHONE>555-5555</PHONE>
    </PHONEBOOK>
    <PHONEBOOK>
        <PHONEBOOKID>3</PHONEBOOKID>
        <NAME>John</NAME>
        <PHONE>777-7777</PHONE>
    </PHONEBOOK>
</SQLResult>
```

This program gets the database content as XML dynamically, without writing any JDBC or XML code. It writes the XML output to the output stream using the setXMLWriter() method. Alternatively, you can write the output to an XML file by setting the file through the setXMLFile() method. However, these two methods are mutually exclusive; you can either write to an output stream or to a file, but not both.

The SQLToXML runtime can generate an XSL file that transforms the XML output into HTML tables. Use setXSLFile() to set the output location of the XSL file. However, the XSL must be applied externally. The SQLToXML runtime does not perform the transform implicitly. You can also use the setParameter() method to supply an input parameter for those queries that require it.

Tutorial 5: Using SDO with XML

The Service Data Objects (SDO) architecture is a framework that unifies data programming. The current SDO specification level is at version 1.0.

As mentioned earlier, the two main concepts in SDO are the data graph and the data object. A data object is a generic representation of the business data provided by the Data Mediator Service (DMS). A DMS can load data from different datastores, including EJB files, XML files, relational databases, Web services, Java Connector Architecture (JCA), and JMS messages. A data graph contains a data object, along with a change summary that keeps track of the changes made to the data object. A data graph also contains the schema that describes the data object.

A DMS populates a data graph and data object from a datastore. The DMS loads a data graph from the datastore, and saves it back to the datastore. For example, a JDBC DMS can load a data graph from a relational database, and save the data graph back to the database. An XML mediator can load a data graph from an XML file, and save it back to the XML file. In this tutorial, you will look at the latter example, loading a data graph from an XML file and saving it back.

The APIs in the SDO architecture include those for the data graph, data object, and change summary. A data graph is represented by commonj.sdo.DataGraph, while a data object is represented by commonj.sdo.DataObject.

SDOs are commonly used in a disconnected fashion, as shown in Figure 5-31. A client submits a request to the DMS for a data graph. The DMS constructs a data graph from the datastore, and it is responsible for loading and saving the data graph. Upon receiving the data graph from the DMS, the client can modify the data object inside the data graph, and the changes are tracked in the change summary. In addition, the client can choose to serialize or deserialize the data graph. SDO provides the API for accessing data within a data object and serializing data graphs. The DMS is only connected to the datastore while loading and saving data graphs. The rest of the time it can be disconnected from the datastore.

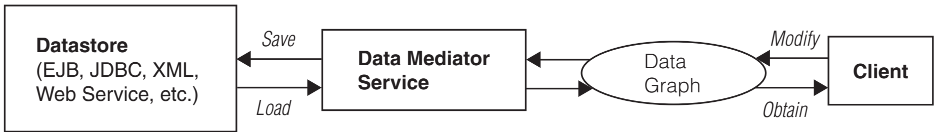


Figure 5-31: The disconnected architecture of SDO.

One of the major advantages of using SDO is that it is technology-independent. Since SDO unifies data programming and the abstraction of data, all you need to know is the SDO API to access any data from any datastores. Furthermore, SDO incorporates a number of J2EE patterns and best practices, which makes it easy to incorporate SDO into your applications. For more information, see *http://www-106.ibm.com/developerworks/java/library/j-sdo/,* “Introduction to Service Data Objects.”

SDO APIs

An SDO data object can consist of primitive types, sequence, or another data object. A data graph class has these APIs:

```
abstract public DataObject getRootObject();
abstract public DataObject createRootObject (String arg, String arg);
abstract public DataObject createRootOjbect (Type arg);
abstract public ChangeSummary getChangeSummary();
abstract public Type getType (String arg, String arg);
```

After you obtain a data graph from the DMS, you can invoke the getRootObject() method to obtain the root data object, or invoke the getChangeSummary() method to obtain the change summary. Once you have the data object, you can navigate it using either getters or setters. Here are some of the APIs provided for navigating data objects:

```
boolean getBoolean(String xpath);
byte getByte(String xpath);
char getChar(String xpath);
double getDouble(String xpath);
float getFloat(String xpath);
int getInt(String xpath);
long getLong(String xpath);
short getShort(String xpath);
byte[] getBytes(String xpath);
BigDecimal getBigDecimal(String xpath);
BigInteger getBigInteger(String xpath);
DataObject getDataObject(String xpath);
Date getDate(String xpath);
String getString(String xpath);
List getList(String xpath);
Sequence getSequence(String xpath);
```

The contents inside a data object can be accessed using a getter with an XPath expression as a parameter. The rest of this tutorial assumes you are familiar with XPath expressions, which were introduced in tutorial 3 of this chapter. Let's say you obtained a data graph from some DMS on the following XML file:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PhoneBook xmlns:tns="http://www.ibm.com/PhoneBook"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.ibm.com/PhoneBook
PhoneBook.xsd">
        <PhoneEntry>
              <Name>Mary</Name>
              <Phone>111-1111</Phone>
        </PhoneEntry>
        <PhoneEntry>
              <Name>Jane</Name>
              <Phone>555-5555</Phone>
        </PhoneEntry>
        <PhoneEntry>
              <Name>John</Name>
              <Phone>777-7777</Phone>
        </PhoneEntry>
</tns:PhoneBook>
```

The root data object can be obtained from the data graph by calling getRootObject(). The root object is a PhoneBook type. To get the first PhoneEntry data object, you can use the XPath expression *PhoneEntry.0*, as shown here:

```
phonebook.getDataObject("PhoneEntry.0");
```

Since the SDO supports the XPath expression 1.0 specification, more complex XPath expressions can be used. For example, let's say you want to get the PhoneEntry data object, given that the name is John. You can use the XPath expression *PhoneEntry[Name='John']*, as shown in this code snippet:

```
phonebook.getDataObject("PhoneEntry[Name='John']");
```

Besides accessing existing data objects, new data objects can also be added to the root object. For example, the following code would add a new PhoneEntry to the PhoneBook.

```
DataObject newPhoneEntry =
phonebook.createDataObject("PhoneEntry");
newPhoneEntry.setString("Name", "Kitty");
newPhoneEntry.setString("Phone", "999-9999");
```

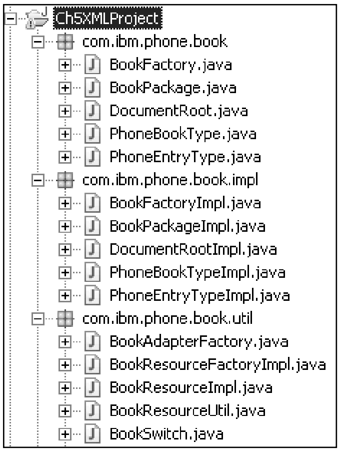
SDO and the Eclipse Modeling Framework (EMF)

The SDO implementation used in this tutorial is actually implemented using the Eclipse Modeling Framework (EMF). The SDO/EMF implementation is an Eclipse open-source project. EMF is a Java-based framework that can be used to define models. It supports automatic model generation from Java interfaces, XML schemas, and UML diagrams. You can get more information about EMF from the Eclipse.org Web site, which includes the reference "The Eclipse Modeling Framework (EMF) Overview" at *http://download.eclipse.org/tools/emf/scripts/docs.php?doc=references/overview/EMF.html*.

When a DMS creates a data graph from a datastore, it needs to organize the data in a common format, where both the DMS and the client can interpret it. The common format is implemented as an EMF model, and it is different depending on the datastore. The DMS is responsible for creating the model that is used to create the data object and data graph. The model is used when the data objects are being transversed and created. For example, a DMS for an XML datastore would create a model from an XML schema and use it later to generate the data object and data graph. Rational Application Developer provides a tool that can generate models from an XML schema. It can be used to create and manipulate data objects and data graphs.

To create an EMF model from an XML schema, simply right-click the schema file and click **Generate => Java => SDO Generator**. This will create an EMF model, as well as utility classes that can help you load and save data objects to and from XML files. For the following XML schema, several EMF model classes will be generated, as shown in Figure 5-32.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
targetNamespace="http://www.ibm.com/PhoneBook"
mlns:tns="http://www.ibm.com/PhoneBook">
  <element name="PhoneBook">
    <complexType>
       <sequence>
         <element name="PhoneEntry" maxOccurs="unbounded">
           <complexType>
             <sequence>
               <element name="Name" type="string" />
               <element name="Phone" type="string" />
             </sequence>
           </complexType>
         </element>
       </sequence>
    </complexType>
  </element>
</schema
```



```
Ch5XMLProject
    com.ibm.phone.book
        BookFactory.java
        BookPackage.java
        DocumentRoot.java
        PhoneBookType.java
        PhoneEntryType.java
    com.ibm.phone.book.impl
        BookFactoryImpl.java
        BookPackageImpl.java
        DocumentRootImpl.java
        PhoneBookTypeImpl.java
        PhoneEntryTypeImpl.java
    com.ibm.phone.book.util
        BookAdapterFactory.java
        BookResourceFactoryImpl.java
        BookResourceImpl.java
        BookResourceUtil.java
        BookSwitch.java
```

Figure 5-32: Generated EMF model for PhoneBook.xsd.

Three packages are created when you create SDO from the XSD file. The com.ibm.phone.book and com.ibm.phone.book.impl packages store the EMF model that represents the PhoneBook.xsd schema. The com.ibm.phone.book.util package provides a sample utility class to load and save data objects. The sample utility class provided does not deal with the data graph. However, you can further modify the utility class to load and save the data graph.

When you load an XML document with the BookResourceUtil class, you get a DocumentRoot object, which contains a PhoneBookTypeImp object. The PhoneBookTypeImpl object is inherited indirectly from the DataObject class.

SDO is very versatile. You can use it not only with XML, but with any other datastore, provided there is a DMS for the datastore. The rest of this tutorial demonstrates how to use SDO with XML. Before going on, if you do not already have a Java project named Ch5XMLProject, create one.

Step 1: Create a New Package

Create a package named *sdo* in Ch5XMLProject:

1. In the Java perspective, right-click **Ch5XMLProject** and click **New => Package**.
2. Enter **sdo** as the package name. Click **Finish**.

Step 2: Copy XSD and XML Files

If you created the PhoneBook.xsd and PhoneBook.xml files in tutorial 2, copy them from the xmlparser folder to the sdo folder. If not, follow step 2 in tutorial 2 to create the two files in the sdo folder.

Step 3: Use the SDO Generator

1. Right-click **sdo/PhoneBook.xsd** and click **Generate => Java**. The Generate Java dialog box opens.
2. Select **SDO Generator** from the Generator list, and make sure **Ch5XMLProject** is selected as the container. Click **Finish**.

After the generator has finished, you should see three new packages under the Ch5XMLProject: com.ibm.phone.book, com.ibm.phone.book.impl, and com.ibm.phone.book.util.

Step 4: Load SDO from XML

In this step, you try to load the XML into a SDO, using the created utility class.

1. Right-click **Ch5XMLProject** and click **New => Class**.
2. Enter **sdo** as the package name and **SDOTest** as the class name. Make sure the **public static void main (String[] args)** check box is selected. Click **Finish**.
3. Modify the SDOTest class as follows and save it. This code loads the XML file using the BookResourceUtil class and obtains a PhoneBookTypeImpl data object from the DocumentRoot:

```
package sdo;

import com.ibm.phone.book.DocumentRoot;
import com.ibm.phone.book.impl.PhoneBookTypeImpl;
import com.ibm.phone.book.util.BookResourceUtil;

public class SDOTest {

  public static void main(String[] args) {
    try {
        DocumentRoot root =
        BookResourceUtil.getInstance().load("sdo/PhoneBook.xml");
        System.out.println ("root = " + root);

        if (root.getPhoneBook() instanceof PhoneBookTypeImpl) {
          PhoneBookTypeImpl phonebook =
          (PhoneBookTypeImpl) root.getPhoneBook();
          System.out.println ("phonebook = " + phonebook);
        }

     }catch (Exception e) {
        e.printStackTrace();
     }
   }
}
```

4. Run the application by right-clicking **SDOTest** in the Package Explorer view and clicking **Run => Java Application**. You should see the following result in the console:

```
root = com.ibm.phone.book.impl.DocumentRootImpl@58a1ce1c (mixed:
[book:phoneBook=com.ibm.phone.book.impl.PhoneBookTypeImpl@524a4e1c])
phonebook = com.ibm.phone.book.impl.PhoneBookTypeImpl@524a4e1c
```

Step 5: Navigate the SDO

The PhoneBookTypeImpl data object was loaded in the previous step. In this step, you navigate the SDO.

1. Add the bolded code in the try/catch block below to the SDOTest class, and save it:

```
DocumentRoot root =
BookResourceUtil.getInstance().load("sdo/PhoneBook.xml");
System.out.println("root = " + root);
if (root.getPhoneBook() instanceof PhoneBookTypeImpl) {
  PhoneBookTypeImpl phonebook =
  (PhoneBookTypeImpl) root.getPhoneBook();
  System.out.println("phonebook = " + phonebook);
```

```
    DataObject obj = phonebook.getDataObject("PhoneEntry.0");
    Type type = obj.getType();
    String typename = type.getName();
    String name = obj.getString("Name");
    String phone = obj.getString("Phone");

    System.out.println("obj = " + obj + "\ntypename = " +
typename  + "\nname = " + name + "\nphone = " + phone);
}
```

This code looks up the first PhoneEntry data object using the XPath expression *PhoneEntry.0*. After obtaining a data object, you can get its type, which is PhoneEntryImpl in this case. In addition, you can get the content by using the XPath expression *Name and Phone* to get the values of the name and phone elements.

2. Run this application again. You will see the name and phone number of the first PhoneEntry element:

```
root = com.ibm.phone.book.impl.DocumentRootImpl@5a2ac823
(mixed:
[book:phoneBook=com.ibm.phone.book.impl.PhoneBookTypeImpl@543f8
823])
phonebook = com.ibm.phone.book.impl.PhoneBookTypeImpl@543f8823
obj = com.ibm.phone.book.impl.PhoneEntryTypeImpl@5532c823
(name: Mary, phone: 111-1111)
typename = PhoneEntryType
name = Mary
phone = 111-1111
```

Step 6: Look Up Another SDO Using XPath

As mentioned previously, you can use XPath expressions to look up SDOs. This step looks up the PhoneEntry data object where the name is equal to *John*.

1. Add the following code at the end of the *if* block in the SDOTest class, and save it. This will look up a PhoneEntry data object where the name is *John*.

```
// Get a data object with a condition
DataObject johnObj =
phonebook.getDataObject("PhoneEntry[Name='John']");
phone = johnObj.getString("Phone");
System.out.println ("John's phone = " + phone);
```

2. Run the application to see that John's phone number is 777-7777.

Step 7: Update the SDO

In this step, you update the SDO and save it back to an XML file. Add the following code at the end of the *if* block in the SDOTest class, and save it:

```
// Update the Name of the PhoneEntry object
obj.setString("Name", "Smith");

// Add a node
DataObject newPhoneEntry =
phonebook.createDataObject("PhoneEntry");
newPhoneEntry.setString("Name", "Bob");
newPhoneEntry.setString("Phone", "222-2222");

//Save back to a file
BookResourceUtil.getInstance().save(root, "sdo/test.xml");
```

This code updates the name of the first PhoneEntry data object and updates the name to *Smith*. In addition, it adds a new PhoneEntry data object to the PhoneBook data object with, *Bob* as the name and *222-2222* as the phone. Furthermore, the code saves the updated SDO back to an XML file named *text.xml*.

After you run the application, you should refresh the sdo package:

1. Right-click the **sdo** package and click **Refresh**.
2. You should see the text.xml file inside the sdo package. Open it in the XML editor. It should look like the file below:

```
<?xml version="1.0" encoding="UTF-8"?>
<tns:PhoneBook xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tns="http://www.ibm.com/PhoneBook"
xsi:schemaLocation="http://www.ibm.com/PhoneBook PhoneBook.xsd">
  <PhoneEntry>
    <Name>Smith</Name>
    <Phone>111-1111</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>Jane</Name>
    <Phone>555-5555</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>John</Name>
    <Phone>777-7777</Phone>
  </PhoneEntry>
  <PhoneEntry>
    <Name>Bob</Name>
    <Phone>222-2222</Phone>
  </PhoneEntry>
</tns:PhoneBook>
```

Notice that the name of the first PhoneEntry has been updated to Smith, and also that a new entry was added, with Bob and 222-2222 as the name and phone number, respectively. The complete SDOTest class should look like the following:

```
package sdo;

import com.ibm.phone.book.DocumentRoot;
import com.ibm.phone.book.impl.PhoneBookTypeImpl;
import com.ibm.phone.book.util.BookResourceUtil;
import commonj.sdo.DataObject;
import commonj.sdo.Type;

public class SDOTest {

  public static void main(String[] args) {
    try {
       DocumentRoot root = BookResourceUtil.getInstance().load(
                                "sdo/PhoneBook.xml");
       System.out.println("root = " + root);
       if (root.getPhoneBook() instanceof PhoneBookTypeImpl) {
         PhoneBookTypeImpl phonebook = (PhoneBookTypeImpl) root
                                      .getPhoneBook();
```

```
        System.out.println("phonebook = " + phonebook);

        DataObject obj = phonebook.getDataObject("PhoneEntry.0");
        Type type = obj.getType();
        String typename = type.getName();
        String name = obj.getString("Name");
        String phone = obj.getString("Phone");

        System.out.println("obj = " + obj + "\ntypename = "
        + typename     + "\nname = " + name + "\nphone = " + phone);

        // Get a data object with a condition
        DataObject johnObj =
        phonebook.getDataObject("PhoneEntry[Name='John']");
        phone = johnObj.getString("Phone");
        System.out.println ("John's phone = " + phone);

        // Update the Name of the PhoneEntry object
        obj.setString("Name", "Smith");

         // Add a node
         DataObject newPhoneEntry =
         phonebook.createDataObject("PhoneEntry");
         newPhoneEntry.setString("Name", "Bob");
         newPhoneEntry.setString("Phone", "222-2222");

         //Save back to a file
         BookResourceUtil.getInstance().save(root, "sdo/test.xml");
         }
     } catch (Exception e) {
         e.printStackTrace();
     }
   }
}
```

You might have noticed that we have not dealt with any data graphs up to this point. The generated BookResourceUtil class acts as a simple DMS for XML, which is sufficient for simple SDO manipulations. However, if you want to use the data graph feature, you can extend the BookResourcUtil to return a data graph instead of a data object when loading the XML file. The main feature of the data graph is the change summary. As changes are made to the data objects in the data graph, the change summary is updated, and you can access the change summary to perform incremental updates to the back-end datastore. In the case of XML, the back-end datastore is actually the XML file.

The save() method in the BookResourceUtil is normally sufficient for updating the XML file, and there is no need to use the change summary for incremental updates. However, in the case of other datastore types, you might need to update the file incrementally using the change summary. The next section of this tutorial shows you how to create a data graph from the XML datastore and navigate the change summary.

Step 8: Extend the BookResourceUtil

Create a Java class named *MyBookResourceUtil* extending the BookResourceUtil class:

1. Right-click the **com.ibm.phone.book.util** package in the Package Explorer view, and click **New => Class**.
2. Enter **MyBookResourceUtil** as the name and browse to **BookResourceUtil** as the superclass.
3. Make sure the **static void main (String[] args)** check box is cleared. Click **Finish**.
4. Modify the file as follows, and save it:

```
package com.ibm.phone.book.util;

import java.io.IOException;
import java.util.Iterator;
import java.util.List;

import org.eclipse.emf.common.util.URI;
import org.eclipse.emf.ecore.EObject;
import org.eclipse.emf.ecore.sdo.EDataGraph;
import org.eclipse.emf.ecore.sdo.SDOFactory;

import com.ibm.phone.book.DocumentRoot;
import commonj.sdo.ChangeSummary;
import commonj.sdo.DataGraph;
import commonj.sdo.DataObject;
import commonj.sdo.Property;

 public class MyBookResourceUtil extends BookResourceUtil {
        private static MyBookResourceUtil instance;
        public static MyBookResourceUtil getMyInstance() {
              if (instance == null) {
                     instance = new MyBookResourceUtil();
              }
              return instance;
        }

        public DataGraph loadGraph(String filename) throws IOException {
              DocumentRoot documentRoot = load(filename);
              EDataGraph graph =
               SDOFactory.eINSTANCE.createEDataGraph();
              graph.setERootObject((EObject) documentRoot);
              return (DataGraph) graph;
        }

        public void saveGraph(DataGraph graph, String filename)
        throws IOException {
              DataObject obj = graph.getRootObject();
              if (obj instanceof DocumentRoot) {
                      DocumentRoot documentRoot = (DocumentRoot) obj;
                      save(documentRoot,filename);
              }
        }
   public void update(DataGraph dataGraph) {
     ChangeSummary changeSummary = dataGraph.getChangeSummary();

     // Use SDO ChangeSummary’s getChangedDataObjects() method.
     List changes = changeSummary.getChangedDataObjects();
     Iterator it = changes.iterator();

     while (it.hasNext()) {
        DataObject changedObject = (DataObject) it.next();
        if (changedObject instanceof PhoneEntryType){
          System.out.println(“Update for ”
          + changedObject.getString(“name”));
        }else if (changedObject instanceof PhoneBookType){
          System.out.println(“Update for ” + changedObject);
        }

        Iterator settingIt =
          changeSummary.getOldValues(changedObject).iterator();
        while (settingIt.hasNext()) {
          ChangeSummary.Setting changeSetting =
           (ChangeSummary.Setting) settingIt.next();
          Property changedProperty = changeSetting.getProperty();
          Object oldValue = changeSetting.getValue();
          Object newValue = changedObject.get(changedProperty);
          System.out.println(“ (changed “ + changedProperty.getName()
          + “ from \”” + oldValue + “\“ to \”” + newValue + “\”)”);
        }

     }
   }
```

```
    private BookResourceImpl getBookResourceImpl(DocumentRoot documentRoot)
    {
    BookResourceImpl resource =
    (BookResourceImpl) ((EObject) documentRoot).eResource();
      if (resource == null)
         resource = (BookResourceImpl) (new BookResourceFactoryImpl())
                        .createResource(URI.createURI("*.xml"));
         return resource;
    }
  }
```

The code has loadGraph(), saveGraph(), and update() methods. The loadGraph() method returns a data graph, given an XML file. The saveGraph() method saves a data graph to an XML file. The update() method accesses the change summary and prints out the changes that were made to the data objects.

The loadGraph() and saveGraph() methods use the load() and save() methods from the BookResourceUtil class. The loadGraph() method, after calling the load() method from BookResrouceUtil, uses the following lines to create a data graph and then set the root object as the DocumentRoot, before returning:

```
    EDataGraph graph = SDOFactory.eINSTANCE.createEDataGraph();
       graph.setERootObject((EObject) documentRoot);
```

An EDataGraph is the EMF implementation of SDO, which is inherited from the commonj.sdo.DataGraph. Similarly, a data graph is passed into the saveGraph() method as a parameter. The method gets the DocumentRoot object using the getRootObject() method, and invokes the parent save() method to save to an XML file.

The update() method gets the change summary and the list of changes using these lines:

```
ChangeSummary changeSummary = dataGraph.getChangeSummary();

// Use SDO ChangeSummary's getChangedDataObjects() method.
List changes = changeSummary.getChangedDataObjects();
```

The method then navigates the list of changes and prints it out in System.out.

Step 9: Use the DataGraph

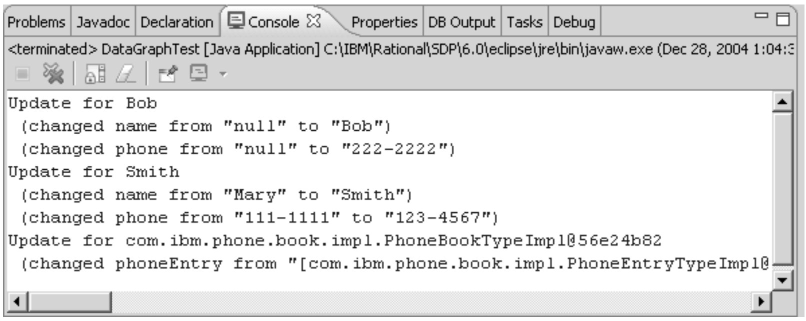
Create a Java class named *DataGraphTest*:

1. Right-click the **sdo** package in the Package Explorer view and click **New => Class**.
2. Enter **DataGraphTest** as the name and make sure the **static void main (String[] args)** check box is selected. Click **Finish**.
3. Modify the file as follows and save it:

```
package sdo;
import com.ibm.phone.book.DocumentRoot;
import com.ibm.phone.book.impl.PhoneBookTypeImpl;
import com.ibm.phone.book.util.MyBookResourceUtil;
import commonj.sdo.ChangeSummary;
import commonj.sdo.DataGraph;
import commonj.sdo.DataObject;

public class DataGraphTest {

  public static void main(String[] args) {
    try {
       DataGraph graph = MyBookResourceUtil.getMyInstance()
                          .loadGraph("sdo/PhoneBook.xml");

       DataObject rootobj = graph.getRootObject();
       if (rootobj instanceof DocumentRoot) {
          DocumentRoot root = (DocumentRoot) rootobj;
          PhoneBookTypeImpl phonebook =
          (PhoneBookTypeImpl) root.getPhoneBook();
          DataObject obj =
          phonebook.getDataObject("PhoneEntry.0");

          // Get the ChangeSummary
          ChangeSummary summary = graph.getChangeSummary();
          // Begin recording what has been changed
          summary.beginLogging();

          // Update the first phone entry
          obj.setString("Name", "Smith");
          obj.setString("Phone", "123-4567");

          // Add a new phone entry
          DataObject newPhoneEntry =
          phonebook.createDataObject("PhoneEntry");
          newPhoneEntry.setString("Name", "Bob");
          newPhoneEntry.setString("Phone", "222-2222");

          // End the recording
          summary.endLogging();
```

```
            MyBookResourceUtil.getMyInstance().update(graph);
            MyBookResourceUtil.getMyInstance().saveGraph(graph,
            "sdo/datagraphtest.xml");

        }

     } catch(Exception e) {
         e.printStackTrace();
     }
   }
}
```

Run the DataGraphTest application, and you will see the change summary in Figure 5-33 in the console. In addition, if you refresh the sdo package, you should see the datagraphtest.xml file with the updated changes. This change summary captures any changes to the data objects between the two lines *summary.beginLoggin()* and *summary.endLogging()*. Any changes made outside of these two lines are not captured.



```
<terminated> DataGraphTest [Java Application] C:\IBM\Rational\SDP\6.0\eclipse\jre\bin\javaw.exe (Dec 28, 2004 1:04:3
Update for Bob
 (changed name from "null" to "Bob")
 (changed phone from "null" to "222-2222")
Update for Smith
 (changed name from "Mary" to "Smith")
 (changed phone from "111-1111" to "123-4567")
Update for com.ibm.phone.book.impl.PhoneBookTypeImpl@56e24b82
 (changed phoneEntry from "[com.ibm.phone.book.impl.PhoneEntryTypeImpl@
```

Figure 5-33: Running the DataGraphTest application.

SDO is simple, convenient, and versatile. It can be used with any back-end datastores and can be executed in a disconnected mode.

Summary

This chapter shows how XML, XSD, and XSLT can be created and used in Rational Application Developer. In addition, SQL-to-XML support greatly simplifies the process of getting XML data directly from a database. Furthermore, Rational Application Developer provides an SDO Generator that generates the model and utility classes. This makes using SDO with XML even easier.