The Art of XSD
SQL Server XML Schema Collections

Jacob Sebastian

The Art of XSD

SQL Server XML Schema Collections

By Jacob Sebastian

Table of Contents

ABOUT THE AUTHOR

Jacob Sebastian is a SQL Server Consultant based in Ahmedabad, India. He's a Microsoft SQL Server MVP, a Moderator at the MSDN and Technet Forums, and volunteers with the Professional Association for SQL Server as Regional Chapter Coordinator for Asia.

As if that wasn't enough, he also runs a SQL Server User Group in his home town, is a frequent columnist at SQLServerCentral.com and blogs regularly at http://blog.beyondrelational.com/.

Jacob started his database career in the early nineties with Dbase and then moved to Clipper, Foxpro and finally settled on SQL Server. He's now been working with SQL Server for over 11 years and is a regular speaker at local User Groups and SQL Server events. He is also a well-known SQL Server trainer, and teaches at various SQL Server classes across the country.

When not working, Jacob likes to watch movies and spend time with family and friends. Somehow, he manages to squeeze it all in together with offering advice in forums, learning more about SQL Server, writing presentations, articles, blogs … or even a book.

ABOUT THE TECHNICAL REVIEWER

David McKinney is a Senior Developer in the Asset Management industry, and has been working on applications using SQL Server since 2000. In recent years, he has focused increasingly on making best use of XML, XSD and XSL with SQL Server, and is the author of several articles on the subject.

Born in Ireland, he speaks 5 languages and has lived and worked in London, Paris and Luxembourg. He lives currently in North East France with his wife Henriette, and children Evan and Owen.

ACKNOWLEDGEMENTS

three-letter words in the school, she reads 'XML' and 'SQL' along with 'cat', 'rat', and 'bat'. Julie, I am so proud of you, and I love you.

I would like to thank all the readers of my articles and blogs for their continuous feedback, support and encouragement for finally making this possible, and I'm sure this book will receive the same affection and acceptance you always gave me.

INTRODUCTION

When information is exchanged in XML format, there needs to be an agreement between the sender and receiver about the structure and content of the XML document. An XSD (XML Schema Definition Language) Schema can be used to enforce this contract and validate the XML data being exchanged and, given that a lot of applications exchange information in XML format, the Art of XSD is becoming an increasingly vital technical skill.

To give you some quick background, in SQL Server 2005 Microsoft introduced the new native XML data type, which represents an XML document or fragment. This is a significant enhancement to the limited XML support available in SQL Server 2000. SQL Server 2005 supports a limited subset of XML Schema Definition Language (XSD), and stores XML schemas as 'XML Schema Collections,' which are a SQL Server object like tables, views or stored procedures.

This book is intended to help you learn and use XML Schema collections in SQL Server. Prior knowledge of XSD is not required to start reading this book, although any experience with XSD will make your learning process easier. I'll start with the basics of XML schemas and then walk you through the schema concepts, schema components, examples and labs to make sure that you're thorough with everything needed to build powerful XML schemas in SQL Server.

If you have any question on the topics discussed in this book or on XSD in general, feel free to write to me at Jacob@beyondrelational.com.

CHAPTER 1
INTRODUCTION TO XML SCHEMA

This chapter provides a basic introduction to XML Schema. It briefly explains what an XML schema is and its significance in today's programming world. Further, it gives a brief overview of DTD and XDR, the predecessors of XSD. Finally, this chapter looks into the XML capabilities of SQL Server 2000, 2005 and 2008.

What is an XML Schema?

An XML Schema is a document which describes another XML document. XML Schemas are used to validate XML documents. An XML schema itself is an XML document which contains the rules to be validated against a given XML instance document.

When do we need an XML schema?

When we write a piece of code (a class, a function, a stored procedure, etc.) which accepts data in XML format, we need to make sure that the data that we receive follows a certain XML structure and should contain values which are coherent. Let us look at an example.

Assume that you are writing a function/method for an application that manages employee data. Your function is expecting the employee information in the following XML structure:

```
<Employee>
  <Name>
    <First>Jacob</First>
    <Middle>V</Middle>
    <Last>Sebastian</Last>
  </Name>
  <!-- Deleted other information for brevity -->
</Employee>
```

Your function needs to make sure that the caller passes correct XML data. You could make use of an XML Schema to perform this validation. An XML Schema which describes and validates the above XML document is given below.

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Employee">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Name">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="First"/>
              <xs:element name="Middle"/>
              <xs:element name="Last"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

By validating the XML data against this schema, you could make sure that the XML document is structured exactly the way your function expects it to be.

To summarize, we need an XML schema when we need to make sure that the XML document that we need to work with is in the expected format. Further, a schema can help to make sure that the values of elements and attributes are within the accepted range (*age* should be between 18 and 65, *Order Date* cannot be a future date, etc.) and in the required format (*Phone Number* should be in the format of *(999) 999-9999, Zip Code* should have 5 digits, Product Code should start with an upper case letter followed by 5 digits, etc.).

Relevance of XSD

There has been a significant increase in the popularity and usage of XML in the past few years. More and more websites and applications started adopting XML for exchanging or publishing information. A few examples are given below:

- Web sites started publishing information in the form of XML feeds (example: RSS, ATOM, RDF, etc.).
- XML Web services became an integral part of enterprise applications.
- A large number of applications are being written that make use of XML web services such as Google APIs, Amazon Web Services, etc. Many small applications that work with frequently changing information (example: news headlines, stock data, weather information, etc.) rely on XML web services.

- Most of the document formats that we use today can be converted to and from XML. Microsoft Open Office XML Format (.docx) of office 2007 and WordML of Word 2003 are examples of XML support getting into word processing. XML is extensively used for documentation. An example is the XML documentation support extended by Visual Studio.
- More and more web sites are turning to AJAX (Asynchronous Java Script and XML) programming, where data is exchanged in XML format. Many of the web pages today use XSLT to generate HTML from XML data.
- An increasing number of web sites adhere to the XHTML standard.
- Many applications use xml to store session or user related data. Microsoft Dot.net applications use XML files for storing configuration data (web.config and app.config). Reporting Services stores report definitions as XML documents.

When data is managed and exchanged in XML format, there needs to be clear agreement about the structure of the XML document. Values of elements and attributes should be in the expected range as well as in the desired format. There needs to be a contract between the caller and the callee about the XML document being exchanged. Once the contract is defined, there has to be a way to enforce it and validate the XML document to make sure that it adheres to the format defined in the contract.

This is where we need an XML Schema! A Schema provides such a contract. It defines the structure of the XML document. It defines rules to validate the value of elements and attributes as well as their formats. Once a schema is defined, a Schema Validator (For example: XmlValidating Reader class of .NET xml library, SQL Server 2005, etc.) can validate an XML document against the rules defined in the Schema.

Schema Languages

As the usage of XML increased, schema languages were also developed to support the validation requirements. DTD, XDR, SOX, Schematron, DSD, DCD, DDML, RELAX NG are a few among them. We will have a quick glance into DTD and XDR in this chapter. An introduction to the other Schema languages is beyond the scope of this book.

Document Type Definition (DTD)

Document Type Definitions (DTD) is one of the commonly used methods for describing XML documents. A DTD can be used to define the basic structure of the XML instance, data type of the attributes, default and fixed values, etc. DTDs are relatively simple and have a compact syntax. On the other side, they have their own syntax. DTD does not provide ample support for common requirements like namespaces, data types, etc.

The following is an approximate representation of the DTD which describes the sample XML we saw previously.

```
<!ELEMENT Employee  (Name)>
<!ELEMENT Name      (First, Middle, Last)>
<!ELEMENT First     (#PCDATA)>
<!ELEMENT Middle    (#PCDATA)>
<!ELEMENT Last      (#PCDATA)>
```

An XML document may have a reference to an external DTD file or can have the DTD embedded as part of the XML file. The XML document given below has embedded DTD information.

```
<?xml version="1.0"?>
<!DOCTYPE Employee [
  <!ELEMENT Employee  (Name)>
  <!ELEMENT Name      (First, Middle, Last)>
  <!ELEMENT First     (#PCDATA)>
  <!ELEMENT Middle    (#PCDATA)>
  <!ELEMENT Last      (#PCDATA)>
]>
<Employee>
  <Name>
    <First>Jacob</First>
    <Middle>V</Middle>
    <Last>Sebastian</Last>
  </Name>
</Employee>
```

The example given below shows an XML document that refers to an external DTD file.

```
<?xml version="1.0"?>
<!DOCTYPE Employee SYSTEM "employee.dtd">
<Employee>
  <Name>
    <First>Jacob</First>
    <Middle>V</Middle>
    <Last>Sebastian</Last>
  </Name>
</Employee>
```

XML-Data Reduced (XDR)

XML-Data Reduced (XDR) was developed in 1998 with the joint effort of Microsoft and University of Edinburgh. The syntax of XDR is very close to that of XSD and is documented at :

http://www.ltg.ed.ac.uk/~ht/XMLData-Reduced.htm.

Microsoft implemented XDR in MSXML Parser. SQL Server 2000 supported creating XML using Annotated XDR Schemas. In SQLXML 4.0 Microsoft added support for XSD schemas and deprecated XDR schemas.

An approximate XDR representation of the sample schema (We have seen an XSD version as well as DTD version) is the following:

```xml
<Schema name="Employee"
        xmlns="urn:schemas-microsoft-com:xml-data">
  <ElementType name="First"/>
  <ElementType name="Middle"/>
  <ElementType name="Last"/>

  <ElementType name="Name" content="eltOnly"  order="seq">
    <element type="First"/>
    <element type="Middle"/>
    <element type="Last"/>
  </ElementType>

  <ElementType name="Employee">
    <element type="Name"/>
  </ElementType>
</Schema>
```

XML Support in SQL Server 2000

SQL Server 2000 was released with a basic set of XML programming capabilities, which includes generating XML data using FOR XML and reading XML data with OPENXML.

FOR XML

FOR XML helps to generate XML output from the results of a TSQL query. When used with AUTO, RAW or EXPLICIT, FOR XML provides different levels of control over the structure of the XML result being generated.

OPENXML

OPENXML() function shreds an XML document and provides a *rowset* representation of the XML data.

SQLXML

SQLXML is an add-on which added additional XML capabilities to SQL Server 2000. Before you could access any of those features, SQLXML should be configured in IIS using the MMS snap-in which is installed as part of SQLXML setup.

With the assistance of SQLXML, SQL Server 2000 offered the following additional features:

Querying Data over HTTP

Once SQLXML is configured in IIS, you can send a TSQL statement over HTTP to the server and receive the results.

XML Views

An *XML View* provides an XML representation of the relational data of one or more tables. Using an XML View, you can run XPath queries on the relational data exposed by the XML View. XML views can be used with *Updategrams* to perform updates on the database.

Web Services

Another important feature exposed by SQLXML is the capability to expose SQL Server 2000 as a web service. This will enable you to send HTTP SOAP requests to the server to execute stored procedures, functions, etc.

XML Support in SQL Server 2005

In addition to many enhancements to the existing XML features, SQL Server 2005 introduced a new data type: **XML**. Let us briefly examine the XML capabilities of SQL Server 2005.

FOR XML – To generate XML Data

SQL Server 2000 supported three different modes with FOR XML, namely: RAW, AUTO and EXPLICIT. SQL Server 2005 added a new mode, PATH. The usage of PATH is relatively simple and it helps to achieve many of the complex XML formatting requirements which were possible only with complex usage of EXPLICIT earlier.

XML Data Type

SQL Server 2005 introduced a new data type: **XML**. An instance of the XML data type represents an XML document or fragment. XML data type can be used to define columns and can also be passed as parameters to functions and stored procedures. Functions can return XML values. You can declare XML variables in TSQL.

XQuery Support

The support for XML data type raised the requirement for querying the XML document stored in an XML column or variable. SQL Server 2005 supports XQuery (XML Query Language). XQuery is a W3C specification designed to provide a flexible and standardized way of querying XML data.

Support for XSD (XML Schema Definition)

SQL Server 2005 supports XSD (XML Schema Definition) to perform validations on the structure and value of XML documents. XML columns and variables can be bound to an XSD schema and the Schema Processing Engine will perform validations on the data, based on the schema definition. Please note that the support of XSD in SQL Server 2005 is still limited.

XML Support Enhancements in SQL Server 2008

SQL Server 2008 added several enhancements to the XML capabilities of the previous version of SQL server.

Schema Validation Enhancements

SQL Server 2008 added a number of enhancements to Schema Validation. Let us quickly examine them.

Lax Validation Support

To increase the flexibility of an XSD schema, wild card components are often used. This is usually done by using elements <xsd:any> or <xsd:anyAttribute>. Wild card components allow adding content that is not known at the time of schema design.

SQL Server 2005 always had options to either "skip" the validation of such elements or to perform a "strict" validation. When validation is "skipped" no validation is applied on such elements. When validation is set to "strict" the elements are always validated.

SQL Server 2008 supports "lax" validation, which validates only elements and attributes for which schema declarations are available. If the schema declaration is not available, the validation will be skipped for those elements and attributes."lax" validation is explained in Chapter 13

Full support for *date*, *time* and *dateTime* data types

XSD specification defines time-zone information as optional with *date*, *time* and *dateTime* data types. However, the XSD implementation of SQL Server 2005 required time zone information to be present with a *date*, *time* or *dateTime* value. However, it did not preserve the time zone information. The value is normalized into UTC date/time.

SQL Server 2008 removes this limitation. You can omit time zone information when storing *date*, *time* or *dateTime* data types. If you include time-zone information, the information is preserved. We will see these enhancements in Chapter 7.

Improved support for *union* and *list* types

SQL Server 2008 adds support for *list* types that contains *union* types. It allows *union* types that contain *list* types as well. We will examine this in Chapter 7.

XQuery Enhancements

SQL Server 2008 adds support for the "let" clause in the "query()" method of the XML data type. Refer to Books Online for a detailed explanation of the "let" clause.

XML DML Enhancements

The only significant DML change is the support for inserting an XML variable (or value of XML type) into another XML variable or XML column (using the XQuery "modify()" method with "insert" operation).

TYPED and UNTYPED XML

SQL Server 2005/2008 supports two flavors of XML known as TYPED and UNTYPED. Typed XML is associated with an XML Schema that defines the structure of the XML variable or column. Any text data can be stored to an UNTYPED XML column or variable as long as it is in XML format. But a TYPED XML column or variable must strictly follow the structure defined in the XML schema (XSD).

TYPED XML has many advantages over UNTYPED XML.

- SQL Server has prior knowledge about a TYPED XML column or variable because it is bound to a schema known to it. This knowledge will help the query optimizer generate better query plans.
- When a TYPED XML is used, SQL Server knows the data types of elements and attributes and can do better query processing.
- SQL Server can perform validations when value is inserted or updated. If the XML document or fragment does not pass all the validations defined in the XML Schema, SQL Server will raise an error and will not modify/insert the data.

By using an XSD schema, you can perform all sorts of validations that need to be done before accepting the XML data. If you work with XML data often you may be familiar with the following requirements, which will make your application less prone to error.

Validate the structure of the XML

Example:
<address> should occur after *<name>*. *<phone>* is optional but there should be one or more *<item>* elements.

Validate the data types

Example:
<zip> should be numeric, *<age>* should be numeric, *<phone>* is alpha numeric, *<dateOfBirth>* should be a valid date value, *<maritalStatus>* should be Boolean.

Perform restrictions on values

Example:
<hiredate> should not be earlier than 1900. *<age>* should be between 18 and 80. *<itemnumber>* should have 3 digits, followed by a *"-"* and then 4 alpha-numeric characters.

There are many more validations that we might need to do, depending upon the nature of our application and the type of data that we receive. Performing such validations without the help of a SCHEMA will be extremely difficult most of the time. Think of reading/parsing the XML document using your favorite XML library and validating each element and attribute. Though you could do this for some of the basic validations, most of the real life validations will be impractical to perform without a SCHEMA.

By using an XSD schema you can define all the validation rules using simple XML structure, and SQL Server 2005 will perform all the validations on your behalf.

How This Book Is Organized

Note: XSD support is added with SQL Server 2005 and that is the subject of this entire book. The XSD support added by SQL Server 2005 is further enhanced by SQL Server 2008. Whenever I say *SQL Server*, I would be referring to both SQL Server 2005 and 2008. I will use the version number to refer to a specific version of SQL Server.

Chapter 1: This chapter gives an introduction to XML schema, discusses a few different schema languages, and then explains the XML support extended by SQL Server 2000, 2005 and 2008.

Chapter 2: In this chapter we will write our first XSD schema. We will then have a look at XML namespaces and become familiar with SQL Server XML Schema collections. We will see how to validate XML instances against schema collections and will have a quick overview of the SSMS XSD editor.

Chapter 3: This chapter presents a fictitious company, North Pole Corporation, which needs some web services developed for their order processing application. This chapter identifies the structure of the XML data to be exchanged and determines the validations to be performed using XSD**.**

Chapter 4: This chapter explains the basic building blocks of XSD. We will see Element Declarations, Attribute Declarations, Simple Types, Complex Types, Attribute Groups, Element Groups, Order Indicators, Occurrence Indicators, annotations, etc. This chapter also explains how to associate data types with element declarations and perform additional validations on the value of elements and attributes.

Chapter 5: This chapter explains element declarations in detail. We will examine *global* and *local* element declarations and discuss each attribute that an element declaration can take. We will examine *name, id, type, default, fixed, nillable, substitutionGroup, abstract, block, final, minOccurs, maxOccurs, ref* and *form* attributes.

Chapter 6: This chapter explains attribute declarations in detail. We will have a quick glance into the behavior of elements and attributes and will discuss global and local attribute declarations. We will then look at each parameter of an attribute declaration and will discuss attribute groups and their usages.

Chapter 7: This chapter starts with a basic discussion on the importance of data types. We will discuss the characteristics of XSD data types and XSD Primitive Data Types in much detail.

Chapter 8: This chapter focuses on Simple Types. It explains the difference between simple types and complex types and discusses local and global simple type declarations. We will examine simple type derivation as well as the enhancements added to list and union types in SQL Server 2008. We will also examine how to restrict derivation of simple types.

Chapter 9: We will discuss XSD Derived Data Types in this chapter. This chapter starts with a discussion on XSD Primitive Data types and Derived data types and then discusses the facets of the primitive data types. Finally, it explains each of the Derived Data Types in detail.

Chapter 10: We will examine Complex Types in this chapter. We will examine local and global complex types and discuss the different content models of complex types. We will also examine order indicators, occurrence indicators and element groups.

Chapter 11: This chapter focuses on derivation of complex types. It discusses the derivation of each content model by restriction as well as by extension. It also explains how to control complex type derivation.

Chapter 12: This chapter explains the Regular Expression language supported in XSD. We will discuss regular expressions, patterns, meta characters, case sensitivity, shorthand character classes, negative expressions and character class subtraction.

Chapter 13: This chapter discusses some advanced XSD concepts. We will discuss element wildcards, attribute wildcards, and extension of wildcard elements and attributes. We will see different ways a schema processor validates wildcard declarations. We will then see the different attributes of a schema declaration.

Chapter 14: This chapter looks closer into XML Schema Collections. We will discuss how to create schema collections having more than one schema definition. We will then discuss how to alter schema collections, how to retrieve the definition of a schema collection from SQL Server and will examine the various system catalog views related to schema collections.

CHAPTER 2

START WRITING THE FIRST SCHEMA

Let us get familiar with writing schemas. We will do the following in this chapter.

- Write our first XML schema
- Learn about XML Namespaces in general
- Look at XSD namespaces and the concept of default namespace
- Add an element declaration to our first schema
- Understand SQL Server SCHEMA Collections
- Validate an XML variable using a schema
- Validate an XML column using a schema
- Modify schema collections
- Look at SSMS schema editor

We have had a basic introduction to XML schemas in the previous chapter. It is time to start getting familiar with writing schemas. In this chapter we will write our first schema and see how the SQL Server validates XML instances against a schema.

We will use the terms "XSD Schema" as well as "XML Schema" to refer to an XML Schema document which describes and validates the structure and content of another XML document.

An XML schema is an XML document. The root element of an XML schema should always be the "*<schema>*" element. All definitions should appear under the root element *"<schema>."*

Writing the First XML Schema

Here is the basic declaration of an XML schema.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
</schema>
```

Listing 2.1: An empty schema declaration.

This is just an empty schema. An empty schema usually does not make any sense. However, this would help us analyze the structure of a basic schema. We will enhance this basic schema and make it more meaningful later in this chapter.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
</schema>
```

Listing 2.2: The schema element.

As mentioned earlier, *"<schema>"* is the root element of an XML Schema. The declarations of all the elements and attributes, along with the validation rules, should appear within the root element.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
</schema>
```

Listing 2.3: The schema namespace.

All elements and attributes of XML Schema are declared in the namespace "http://www.w3.org/2001/XMLSchema." Hence, every XML schema document should contain the above namespace declaration.

Let us try to understand namespaces in a bit more detail.

XML Namespaces

XML Namespaces help to resolve ambiguity. In our day-to-day programming life we do a lot of stuff to resolve ambiguities. Let me give you a few examples:

Most of the times we use aliases while writing TSQL queries. When selecting data from a single table, you may not need to prefix the column

names with table name or alias. But when you join tables, and if more than one table has columns with the same name, you always need to give an alias to resolve ambiguity.

For example:

```sql
SELECT
       d.Name AS DepartmentName,
       e.Name AS EmployeeName
FROM Employees e
INNER JOIN Departments d
       ON d.DepartmentID = e.DepartmentID
```

Listing 2.4: A TSQL example showing the usage of aliases.

Since both *Employees* and *Departments* tables have a column named *Name*, we use an alias to resolve the ambiguity. After we added the alias, SQL Server can distinguish between the two columns having the same name.

Now let us look at a VB.NET example.

```vbnet
'Get the data
Dim dbHelper As New Database.Helper()
dbHelper.GetSomeData()

'Display the data
Dim uiHelper As New UI.Helper()
uiHelper.DisplayTheData()
```

Listing 2.5: A VB.NET example showing the usage of namespaces.

In the example you could see two classes named *Helper* but prefixed with two different namespaces. Those two classes would cause an ambiguity error if you do not use the namespace prefixes.

Regardless of the tool or programming language, we almost always use some kind of naming mechanism to avoid ambiguity. Such cases of ambiguity can exist in XML, too. The following XML document stores the configuration data of an application.

```xml
<configuration>
  <connection>
    <provider>World Wide Internet Providers</provider>
    <speed>512 KBPS</speed>
  </connection>
  <connection>
    <provider>SQL Client Provider</provider>
    <protocol>TCP/IP</protocol>
    <authentication>Windows</authentication>
```

```
   </connection>
</configuration>
```

Listing 2.6: An XML document having ambiguous elements.

The first *Connection* element contains information of an Internet Connection and the second *Connection* element contains information of a Database Connection.

The application needs to have a way to identify which element stores Internet Connection information and which one stores Database Connection information. When the application needs to make a connection to the database, it needs to read the specific element containing connection information. Currently, the application does not have a way to identify the correct element due to the ambiguity those two elements create.

This is where a namespace declaration can help. Look at the following example that uses namespace declarations to resolve ambiguity.

```
<configuration
        xmlns:db="http://www.sqlserverandxml.com/db"
        xmlns:net="http://www.sqlserverandxml.com/net">
  <net:connection>
    <net:provider>World Wide Internet Providers</net:provider>
    <net:speed>512 KBPS</net:speed>
  </net:connection>
  <db:connection>
    <db:provider>SQL Client Provider</db:provider>
    <db:protocol>TCP/IP</db:protocol>
    <db:authentication>Windows</db:authentication>
  </db:connection>
</configuration>
```

Listing 2.7: Using XML namespaces to avoid ambiguity.

The above XML document has separate namespaces to qualify the elements that stores Internet Connection information and Database Connection information. The application can read the desired *Connection* element by using the specific namespace associated with the element. This removes the ambiguity we discussed with the previous example.

A namespace is declared using *"xmlns"* attribute. It then associates a prefix with the namespace URI. A namespace URI uniquely identifies a namespace. It is used only to uniquely identify a namespace and it does not need to be a valid web URL.

These namespace prefixes are used in the XML document to qualify the elements, which is very close to what we do with table aliases in TSQL queries.

Default Namespaces

When you declare namespaces in an XML document, you can specify one of those namespaces as the *default namespace*. A default namespace does not take a prefix and all the elements of the default namespace should not take a prefix, as well. Let us modify the xml document we saw in the previous example and add a default namespace to it.

```
<configuration
        xmlns="http://www.sqlserverandxml.com/db"
        xmlns:net="http://www.sqlserverandxml.com/net">
  <net:connection>
    <net:provider>World Wide Internet Providers</net:provider>
    <net:speed>512 KBPS</net:speed>
  </net:connection>
  <connection>
    <provider>SQL Client Provider</provider>
    <protocol>TCP/IP</protocol>
    <authentication>Windows</authentication>
  </connection>
</configuration>
```

Listing 2.8: Using default namespace.

Note that the namespace of *Database Connection* is declared as the default namespace; hence, the elements of that namespace are not qualified with namespace prefixes. There is no ambiguity because all un-prefixed elements belong to the default namespace.

Here is another version of the XML which makes the namespace of *Internet Connection* as the default namespace.

```
<configuration
        xmlns:db="http://www.sqlserverandxml.com/db"
        xmlns="http://www.sqlserverandxml.com/net">
  <connection>
    <provider>World Wide Internet Providers</provider>
    <speed>512 KBPS</speed>
  </connection>
  <db:connection>
    <db:provider>SQL Client Provider</db:provider>
    <db:protocol>TCP/IP</db:protocol>
    <db:authentication>Windows</db:authentication>
```

```
   </db:connection>
</configuration>
```

Listing 2.9: Another example using a default namespace

The same rules are applicable here, too. The un-prefixed *Connection* element is part of the default namespace; hence, there is no ambiguity.

XSD Namespace

I had mentioned earlier that all XML schema documents should take the namespace declaration http://www.w3.org/2001/XMLSchema. In the previous section we have learned about XML Namespaces and Default Namespaces. With the new understanding that we gathered about namespaces, let us once again look into the basic schema we created at the beginning of this chapter.

```
<schema xmlns="http://www.w3.org/2001/XMLSchema">
</schema>
```

Listing 2.10: An empty schema showing namespace declaration.

The above example declares the namespace as the default namespace. It is a common practice to use a prefix while creating schemas. *"xs"* and *"xsd"* are the most common prefixes assigned to the XML schema namespace. Most of the times you will see XML schemas written as follows:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
</xs:schema>
```

Listing 2.11: A Schema using namespace prefix "xs."

Or

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
</xsd:schema>
```

Listing 2.12: A Schema using namespace prefix "xsd."

It is recommended that you always use*"xs"* or "*xsd"* while creating your schema documents. However, it is completely legal to use a different prefix; for example:

```
<jacob:schema xmlns:jacob="http://www.w3.org/2001/XMLSchema">
</jacob:schema>
```

Listing 2.13: Example of a schema using non-standard namespace prefix.

Though you can change the namespace prefix, you cannot touch the namespace URI. The namespace URI should always be http://www.w3.org/2001/XMLSchema. Note that namespace URIs are case sensitive. Your schema will be invalid if you do not use the correct namespace name with correct casing.

Adding an element

The schema that we defined in the previous section was empty. It did not declare any element that should exist in the XML instance document.

> **We will use the term XML Instance Document to refer to an XML document that we intend to validate with a given schema.**

Let us modify our schema and add an element declaration.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Greetings"/>
</xsd:schema>
```

Listing 2.14: A schema with an element declaration.

The above schema declares a root element named *Greetings*. The XML Instance Document of this schema should contain a root element named *Greetings.* Here is a valid XML instance document which successfully validates against this schema.

```
<Greetings>Welcome to XSD Workshop!</Greetings>
```

Listing 2.15: An XML instance document which validates against the schema in Listing 2.14.

SQL Server Schema Collections

We just wrote a simple schema having a single element declaration. Our schema is not yet added to SQL Server, and so we are not yet ready for performing validations using the schema we just defined.

SQL Server 2005 introduced a new system object: *XML Schema Collection*. An *XML Schema Collection* is a SQL Server object that stores the definition of one or more XML Schemas. So the next step is to create an *XML Schema Collection* from the XSD schema we created above.

An *XML Schema Collection* is created with TSQL command *CREATE XML SCHEMA COLLECTION*. Once a *Schema Collection* is created, the definition cannot be altered. If you want to modify your schema definition, you should drop it and create it again with the new definition (which could be a pain in many cases).

> **A *Schema Collection*, as the name suggests, can store the definition of more than one *XML Schema.* In most cases, though, you'll only want to store one schema in a schema collection. This is explained in Chapter 14.**

Creating a Schema Collection

Let us create a *Schema Collection* with the schema definition we wrote earlier in this chapter. Here is the code to create an *XML Schema Collection* in SQL Server.

```
CREATE XML SCHEMA COLLECTION GreetingsSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Greetings"/>
</xsd:schema>'
GO
```

Listing 2.16: TSQL Code to create an XML Schema Collection.

In the above example, the TSQL command "CREATE XML SCHEMA COLLECTION" creates a new XML Schema Collection with the name "GreetingsSchema." The definition of the schema is given after the schema name.

You can query the system catalog view "sys.xml_schema _collections" to retrieve a list of all the schema collections. Chapter 14 has a detailed discussion on all the system catalog views related to XML Schema Collections.

Performing validations using Schema Collections

We have just created a Schema Collection named *GreetingsSchema.* Now let us perform some validations using the newly created schema collection.

In Chapter 1, we discussed a bit about TYPED and UNTYPED XML. An XML variable or column is said to be TYPED when it is bound to a schema collection. Only TYPED xml can be validated with an XML schema.

SQL Server validates a TYPED XML column or variable when a value is assigned or modified. So to validate an XML document, simply create a TYPED xml variable and try to store the value to it. If the validation succeeds, the value will be assigned to the variable. If the validation fails, SQL Server will raise an error and the assignment operation will fail.

Validating an XML variable

Having created an XML Schema Collection, we are ready to perform the validations. Let us try to validate an XML variable using the XML Schema Collection we just created.

```
-- declare an XML variable and bind it to a schema
DECLARE @x XML(GreetingsSchema)

-- validate and assign a value
SET @x = '<greetings>Hi There!</greetings>'
```

Listing 2.17: Validating an XML variable with a schema.

The statement *"DECLARE @x XML(GreetingsSchema)"* declares an XML variable bound to the Schema Collection named *"GreetingsSchema."* The next statement (skip the comments, please) tries to assign a value to the XML variable.

If you run the above code, you will get the following error.

```
Msg 6913, Level 16, State 1, Line 2
XML Validation: Declaration not found for element 'greetings'.
Location: /*:greetings[1]
```

XML is case sensitive. The first letter of the element we declared in our schema was in upper case, but the value we tried to assign to the variable has the element name in lower case. Here is the corrected version of the code.

```
-- declare an XML variable
DECLARE @x XML(GreetingsSchema)

-- validate and assign a value
SET @x = '<Greetings>Hi There!</Greetings>'
```

Listing 2.18: Validating an XML variable with a schema.

Validating an XML column

We just saw how to validate an XML variable. Now let us look at validating XML columns. Just as with XML variables, SQL Server performs validations when a value is assigned to a TYPED XML column. Validation will take place when a TYPED XML column is modified as well.

To test the validation of a TYPED XML column, let us create a table with a TYPED XML column and see the validation in action. You can associate an XML column with a Schema Collection while creating a table.

```
-- create a table with a TYPED XML column
CREATE TABLE EmployeeGreetings(
      EmployeeID INT,
      Greeting XML(GreetingsSchema)
)

-- insert a record
INSERT INTO EmployeeGreetings(EmployeeID, Greeting)
SELECT 1, '<Greetings>Hi There!</Greetings>'
```

Listing 2.19: Validating an XML column with a schema collection.

In the above example, we have created a table having a TYPED XML column. The XML column is bound to an XML Schema Collection named *"GreetingsSchema."* Then we tried to insert a new record to the table. SQL Server will perform the validations defined in the Schema Collection before assigning the value. The operation will succeed only if the validation succeeds.

You could also use ALTER TABLE syntax to add a new TYPED XML column to a table. The following code snippet demonstrates that.

```
-- add a new TYPED XML column
ALTER TABLE EmployeeGreetings
      ADD TodaysGreeting XML(GreetingsSchema)
```

Listing 2.20: Adding a new TYPED XML Column.

The ALTER COLUMN syntax can be used to change the schema binding. You can use it to change a TYPED XML column to UNTYPED, as well as to change the Schema Collection to which the column is associated.

```
-- create an UNTYPED XML column
ALTER TABLE EmployeeGreetings
      ADD YesterdaysGreetings XML

-- alter the UNTYPED XML column to TYPED
ALTER TABLE EmployeeGreetings ALTER COLUMN
      YesterdaysGreetings XML(GreetingsSchema)
```

Listing 2.21: Altering an UNTYPED XML column to TYPED.

Again, the following code shows how to turn a TYPED XML column to UNTYPED.

```
-- alter the TYPED XML column to UNTYPED
ALTER TABLE EmployeeGreetings ALTER COLUMN
      YesterdaysGreetings XML
```

Listing 2.22: Altering a TYPED XML column to UNTYPED.

It is also possible to directly change the Schema Collection to which the column is bound. The following code snippet demonstrates that.

```
-- Bind the column to "GreetingsSchema"
ALTER TABLE EmployeeGreetings ALTER COLUMN
      YesterdaysGreetings XML(GreetingsSchema)

-- Alter the schema binding to "MessageSchema"
ALTER TABLE EmployeeGreetings ALTER COLUMN
      YesterdaysGreetings XML(MessageSchema)
```

Listing 2.23: Changing the binding to a new schema.

When a column is bound to a schema collection, SQL Server will validate each row in the table to make sure that the value stored in the column (being modified) validates successfully with the new Schema Collection. If

any of the rows has a value that does not validate successfully, the ALTER operation will fail.

Modifying a Schema Collection

As I had mentioned earlier, SQL Server does not easily allow editing the definition of a Schema Collection. If you want to make changes, you should drop the XML Schema Collection and recreate it.

This will be a problem if the Schema Collection is bound to one or more columns in any of the tables in the database. SQL Server will not allow you to drop a Schema Collection if it is bound to a column of any of the tables in the database. This makes modifying a schema definition very difficult and frustrating.

To update the schema, you will have to do the following:

1. Alter all TYPED XML columns bound to the given Schema Collection to UNTYPED XML (or bind to any other Schema Collection which is compatible with the data already stored in the column).
2. Drop the schema collection.
3. Create the schema collection with the updated definition.
4. Bind the columns back to the newly created Schema Collection.

Let us try to follow the above steps and try to alter the Schema Collection we created previously. First of all, let us alter the columns to UNTYPED XML to remove the association between columns and the Schema Collection.

```
-- alter the columns to UNTYPED
ALTER TABLE EmployeeGreetings
      ALTER COLUMN Greeting XML
ALTER TABLE EmployeeGreetings
      ALTER COLUMN TodaysGreeting XML
ALTER TABLE EmployeeGreetings
      ALTER COLUMN YesterdaysGreetings XML
GO
```

Listing 2.24: Altering columns to UNTYPED before dropping Schema Collection.

Now, let us drop the schema collection.

```
-- drop the schema collection
DROP XML SCHEMA COLLECTION GreetingsSchema
GO
```

Listing 2.25: Dropping a Schema Collection.

The next step is to recreate the Schema Collection with the modified definition. Let us modify our schema definition a bit. Let us rename the root element name from *Greetings* to *GreetingsOfTheDay*. Here is the code to create the new schema.

```
-- create updated schema collection
CREATE XML SCHEMA COLLECTION GreetingsSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="GreetingsOfTheDay"/>
</xsd:schema>'
GO
```

Listing 2.26: Creating a new schema collection with updated schema definition.

After recreating the Schema Collection with the new definition, let us alter the columns and bind them back to the newly created Schema Collection.

```
-- alter the columns back to TYPED
ALTER TABLE EmployeeGreetings
    ALTER COLUMN Greeting XML(GreetingsSchema)
ALTER TABLE EmployeeGreetings
    ALTER COLUMN TodaysGreeting XML(GreetingsSchema)
ALTER TABLE EmployeeGreetings
    ALTER COLUMN YesterdaysGreetings XML(GreetingsSchema)
```

Listing 2.27: Altering columns to bind them to a Schema Collection.

If you attempt to run this code, you will see that SQL Server generates the following error.

```
Msg 6913, Level 16, State 1, Line 2
XML Validation: Declaration not found for element 'Greetings'.
Location: /*:Greetings[1]
The statement has been terminated.
```

The error occurs because some of the records in the columns that we tried to alter have data that is not valid per the Schema Collection to which we are trying to bind the column. When you bind a column to a Schema Collection, SQL Server will validate the data stored in the column with the given Schema Collection and make sure that all the rows succeed the

validation. If any of the rows fails the validation, SQL Server will generate an error and the operation will fail.

In our case, the new schema definition has a different root element. The name of the root element was *"Greetings"* in the previous version. The records already stored in the table have this root element. The new schema collection has a different root element. The name of the new root element is *"GreetingOfTheDay."* Because of this, the validation of existing values with the new schema collection will fail and SQL Server will raise an error and abort the operation.

To overcome this, either we need to delete the existing records or update the values with correct data (that validates with the new schema definition) before binding the column to the new schema collection.

SQL Server Management Studio – Schema Editor

Earlier in this chapter we learned to create XML schemas. The simplest tool you can use to create a schema is a text editor. It could be as simple as notepad or as complicated as any specialized XML/XSD editors available today. Specialized XSD editors can increase productivity by providing intellisense/auto-completion features, real-time syntax checks, etc. There are many such tools available today. Visual Studio and SQL Server Management Studio contain such tools that I think would be familiar to most of you.

SQL Server Management Studio comes with a nice XML editor. It helps you to quickly write schemas. The intellisense support and features like auto-completion will help you write schema code faster and with less of a possibility of typing errors.

Launching the Schema Editor

So, how do we launch the schema editor so that we can start creating a schema and see those nice features? Where is the menu? *File -> New* and then?

Well, there isn't a menu available to launch the XML/Schema editor. I am not kidding! SSMS does not have a menu to launch the XML/Schema editor.

So what do we do now? Well, there are a few workarounds to launch the XML/Schema editor of SSMS.

Open a dummy XML file.

I think this would be the easiest way. Go to *File* menu and select *Open*. Select an XML file from any of the folders. If you don't have an XML file available, create one.
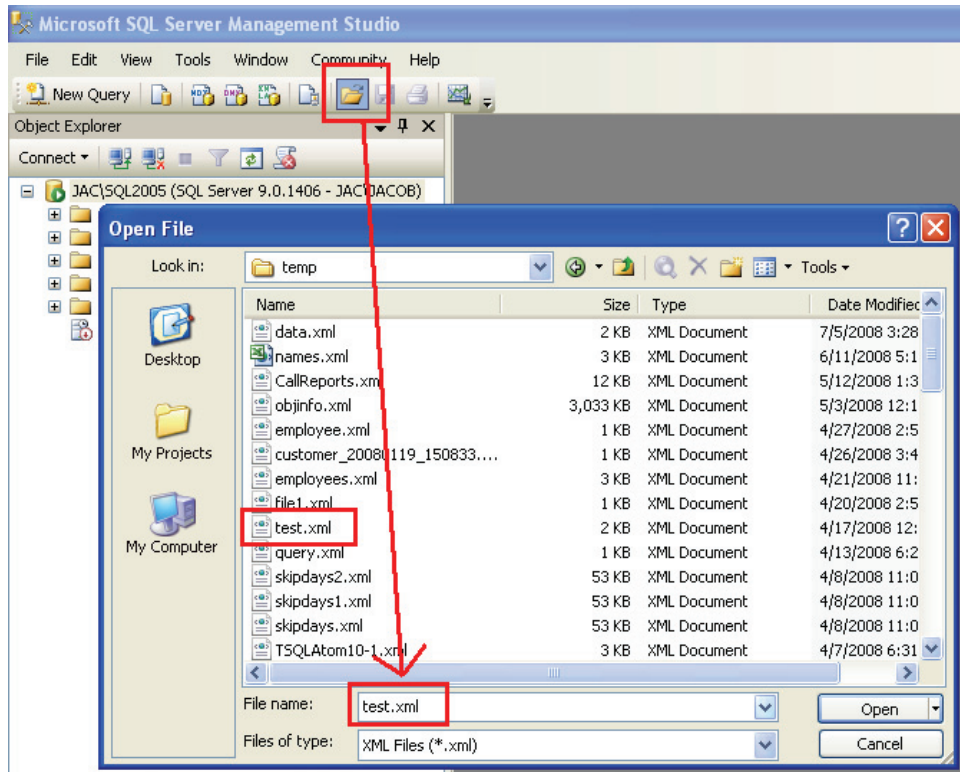
After selecting the XML file, click on the "Open" button. SSMS will open the XML in the XML/XSD editor. If It is a dummy file (a file that you don't really need), just delete the content and start writing your schema. If it is not a dummy file, then save it with a new name (using File->Save As menu) and start editing it.

Write a dummy FOR XML query

Another option is to write a fake FOR XML query. For example:

```sql
SELECT TOP 1 * FROM SYS.TABLES FOR XML AUTO
```

This query will produce an XML output if you have selected the *Output to Grid* option. You can enable *Output to Grid* by pressing CTRL + D.
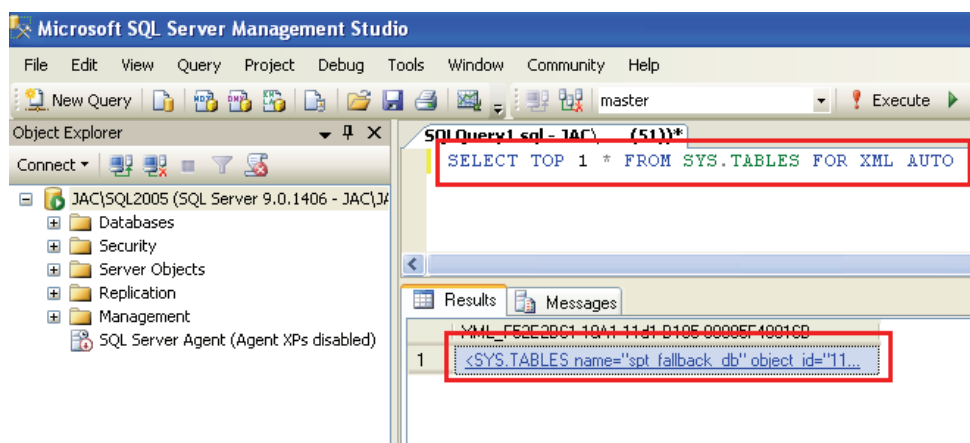
Click on the query result and it will open the XML document in a new XML editor window. Delete the existing content and then start writing your schema.

Write a dummy CAST

The approach used by the previous point is to generate an XML result and clicking on the XML result to open it in a new XML editor window. Hence, we wrote a FOR XML query that produces an XML output. So the key here is to produce an XML output.

An XML output can be easily produced by a query as simple as the following:

```sql
SELECT CAST('<A />' AS XML)
```

This will produce an XML output and you can click on the result window to open the XML document in a new XML editor window. This is what I usually do to open the XML editor when I want to write a new schema.

Open a new Analysis Services XMLA Query

Another option is to open a new Analysis Services XMLA Query. You can do it from the *File* menu.
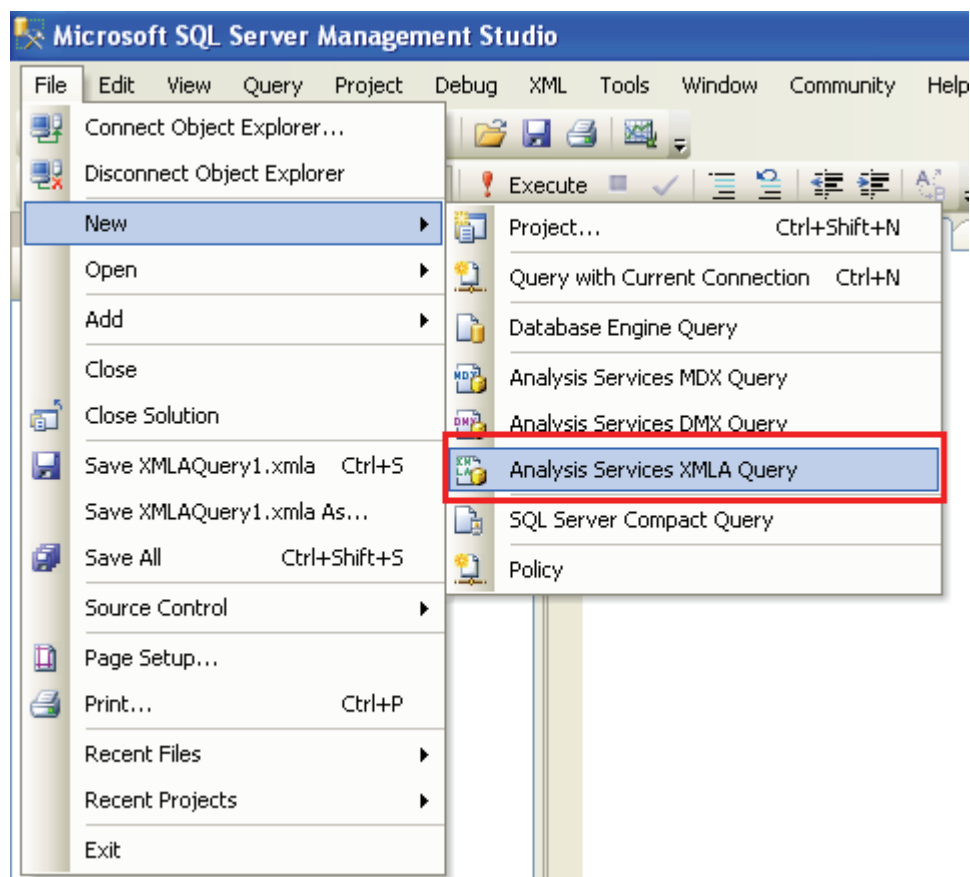
When you select *Analysis Services XMLA Query,* SSMS will ask you to provide information to connect to an Analysis Server instance. Since we don't really need to do anything with Analysis Services (our intention is to open a new XML editor window) you can press the *"Cancel"* button. This will take you to a new instance of XML editor window.

I hope the next version of SSMS will have a *File –> New –> XML File* menu which will open a new XML editor window. SQL Server 2008 RC0 does not have this yet – maybe on the next version.

Editor Features

The XML editor window of SSMS has quite a lot of features to help writing schemas. There are three basic features that the XML editor provides.

1. Intellisense

The intellisense support helps to quickly write schema elements.
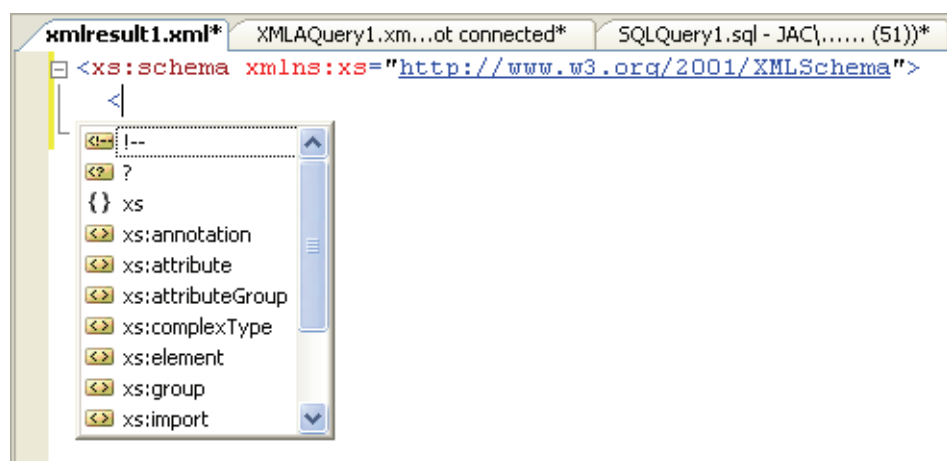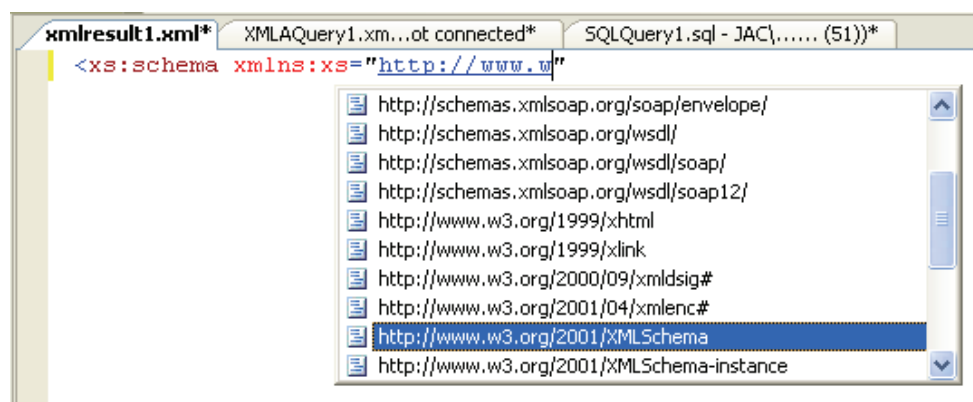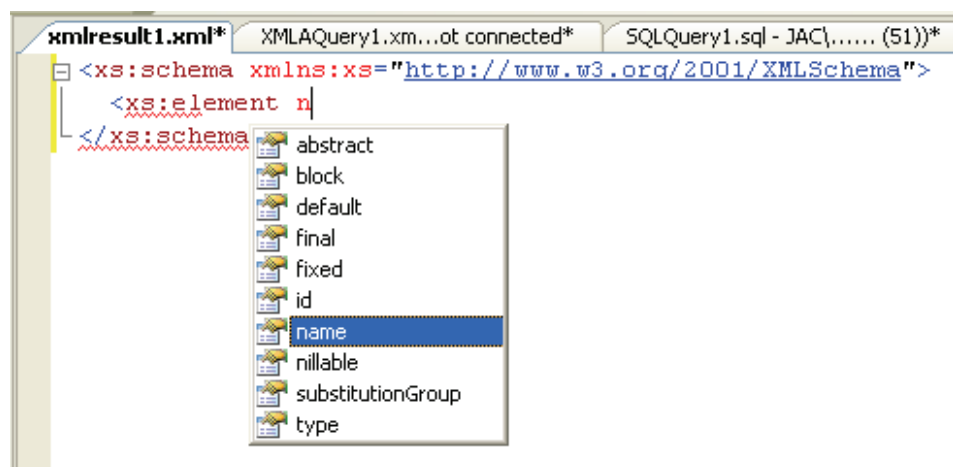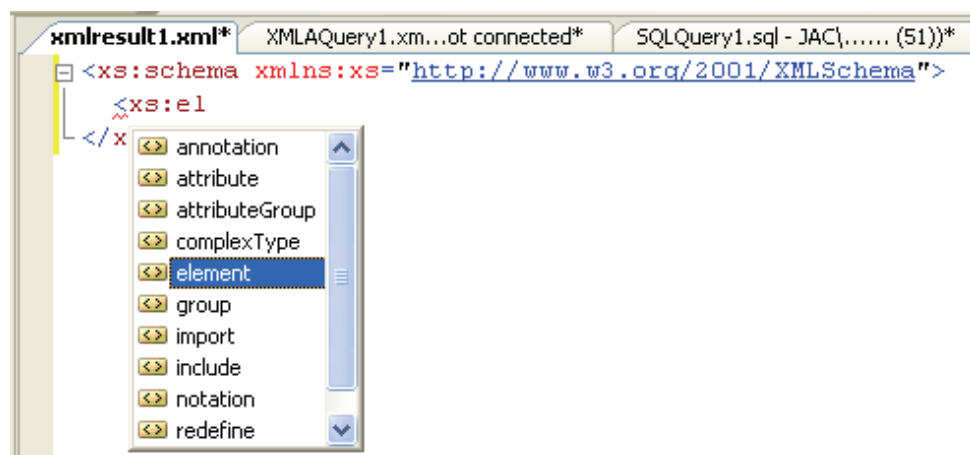
2. Auto Completion

The Auto completion feature helps writing schema elements faster and helps avoid making typing mistakes. The editor will automatically add an end tag when you create a start tag.
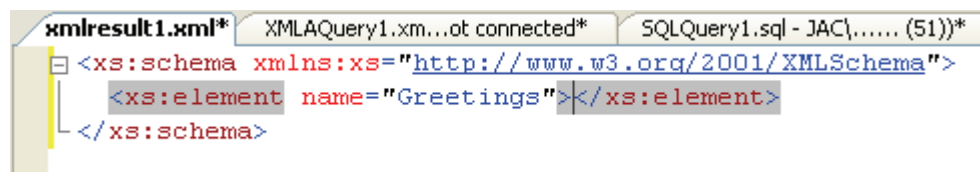
3. Real-time syntax checks

One of the most important features of the XML editor is the real-time syntax check feature. This will help you to quickly spot mistakes.

You will experience these features when we go ahead and start writing a few schemas in the next few chapters.

Chapter Summary

XSD Schema is an XML document which describes and validates other XML documents. The root element of an XML schema is <schema>. All the elements and attributes of XSD are defined in the namespace *"http://www.w3.org/2001/XMLSchema."* You can use any namespace prefix while creating an XSD schema. However, *"xs"* and *"xsd"* are the most commonly used namespace prefixes.

SQL Server stores schema definitions in *XML Schema Collections*. An *XML Schema Collection* can store the definition of more than one XML schema. When an XML column or variable is bound to a Schema Collection, it is called TYPED XML. When they are not bound to any Schema Collection they are called UNTYPED XML. An XML column or variable can be bound to only one Schema Collection. However, each schema collection can contain the definition of more than one schema.

SQL Server validates TYPED XML variables and columns when the value is assigned or modified. The assignment/modification operation will succeed only if the validation succeeds. If the validation fails, SQL Server will throw an error and will abort the operation.

SQL Server does not allow modifying the definition of a Schema Collection. If you want to modify the definition, you need to drop the Schema Collection and recreate it with the new definition.

SQL Server will not allow you to drop a Schema Collection which is bound to a column. Before dropping a Schema Collection you need to alter the columns to UNTYPED XML or to any other Schema Collection compatible with the values stored in the columns.

When you bind an existing column to a Schema Collection, SQL Server will validate the data stored in all the rows against the new Schema Collection and will throw an error if any of the rows fails the validation.

SQL Server Management Studio comes with a nice XML/XSD editor. It has a number of features that makes schema writing lot easier. Intellisense, auto-completion, real-time syntax checks, etc., are a few of those features.

CHAPTER 3

LAB: ORDER PROCESSING APPLICATION FOR NORTH POLE CORPORATION

We have started our journey to learn SQL Server XSD and have so far reached two milestones. In the coming chapters we will build significantly on what we have learned so far about XSD and will do a few hands-on labs to make sure that we have learned it well. All the hands-on labs that we will take in this book are focused around an order processing application required by North Pole Corporation, a fictitious company.

This chapter will give you an introduction to the Order Processing Application and the detailed requirements. We will use this information to develop the required schema in the labs that we will attend in the coming chapters.

North Pole Corporation

North Pole Corporation, based in New York City, is a leading distributor of consumer goods. They have a nationwide distribution network and have ten warehouses throughout the country.

The company has its own sales team that deals directly with retail outlets and manages orders and delivery of shipments. As part of its expansion plans, the company has decided to appoint franchisees (Let us call them *Partner Agencies* hereafter) in some of the cities where the company does not have its own sales team. Under this model, the Partner Agencies will collect orders from customers and will submit the orders to North Pole Corporation. North Pole Corporation will process the order and will deliver the merchandise to the customers.

The IT team of North Pole Corporation was given the responsibility to identify the additional software requirements to fully support and manage the new business model. After doing the required research and analysis, the IT team came up with the following recommendations.

- The partner agencies will use their own software to register and manage orders.
- North Pole Corporation will develop an integration module that will enable data exchange with the application of the partner agencies.
- The partner agencies will do the required software changes/additions on their side to communicate with the integration module of North Pole Corporation.
- When the partner agencies receive an order from a customer, they will register it with their software system. Once the order is registered in their software system, the order will be submitted to North Pole Corporation using the integration module.
- After an order is submitted to North Pole Corporation, the partner agencies will use the integration module to query the status of the order.

The recommendations of the IT team were approved and the software development team started designing the integration module. They decided to develop a web service which exposes the required web methods needed for the integration.

The Web Service

The Software Team identified the web methods needed for the integration. They needed two web methods: one to register the order and the other to query the status of an order previously submitted.

1. RegisterOrder

This method accepts order information from the software system of partner agencies. They will submit the order data in the form of an XML document that contains information of one or more orders.

2. QueryOrderStatus

After submitting an order, partner agencies will call this method to query the status of the order. An order cycle usually takes a few days and during that period the order will pass through different states (Waiting, Approved, Picking, Shipped, etc.). Partner agencies can track the current status of their orders using this web method.

The XML Data

The software system of partner agencies will call Web Method *RegisterOrder* and pass the order information as an XML document. The XML document needs to follow a certain structure and the values of attributes and elements need to follow a set of rules. These rules are to be explained to each partner agency to make sure that they will send data in correct format to the application. This brings up the need for the following:

- A method to describe the XML structure to be used for registering orders. The development team of vendor agencies will carefully examine the required XML structure and will make sure that they will use the correct XML structure while registering orders.
- A method to validate the XML data being passed to the web service. Before accepting the XML data it has to be validated, and if the validation fails the order should not be accepted.

Sheryl from the development team suggested that they should create a Word Document that explains the XML structure. The document should clearly outline the elements and attributes along with all the validation rules. The format and ranges of values should also be clearly documented.

This document can be sent to all the partner agencies with a request that they make sure that the data that they send will comply with the requirements specified in the document.

When the web service receives the order information, it should perform the same set of validations on the XML data and should not accept an order if the validation fails. She suggested using the .NET XML library to perform the validations. A validation function needs to be created that uses the .NET XmlTextReader class to parse the XML document. The function will traverse through the XML tree and will validate each element and attribute against the rules defined in the document.

Enter XSD

Steve, the lead developer, was not convinced by the approach suggested by Sheryl. He had read about the XML capabilities of SQL Server. He had a basic idea of XSD and the XSD support in SQL Server. He decided to investigate further and came up with a more elegant solution using XSD.

After doing some exercises with SQL Server and XSD, Steve and his team set up the plan to create an XSD schema to describe and validate the XML data accepted by the web service. It is decided that, once the schema is developed, it will be sent to all the partner agencies. The development team of the partner agencies will get a clear idea about the required XML structure from the schema. They will develop the integration module per the rules defined in the XSD schema.

At North Pole Corporation, SQL Server will validate the XML submitted by the client applications. Steve and his team need not worry about the validation. The only task they need to focus on is to write a SCHEMA that correctly describes and validates the XML accepted by the web service.

As you could figure out from the above, the decision to use an XSD schema helped Steve and his team in a number of ways:

- Communications with the development team on the other end became easier. Since a schema explains the required XML structure much better than a word document or a document of any other kind, there are no chances of misunderstanding or misinterpreting what is required.
- Validation of the XML data submitted by the client applications became easier. SQL Server will perform the validation based on the schema that Steve and his team developed; hence, no other validation process is needed in the web service.

Identifying the Data Elements

The next step that Steve and his team took was to identify the different pieces of information needed to register an order. They looked into existing order processing systems, as well as the new requirements, and came up with a final listing as follows:

Order Header Information	
Agency Code	Alpha-numeric code of the partner agency
Order Number	An order number generated by each agency. They will use this code while querying the status of an order.
Order Date	Date on which the Customer placed the order.

Order Header Information	
Delivery Date	Date on which the merchandise is to be delivered at the shipping address.
Customer Number	Code to identify the customer within an agency. They may use this code while querying the status of an order.
Customer Name	Name of the customer
Billing Address	Billing Address
Shipping Address	Shipping Address
Terms	Payment Terms
Contact Person	The name of the person to contact for any queries related to the order. Usually this will be the purchase in-charge/manager of the customer.
Title	Title of the contact person. Example: Purchase Manager, Store Supervisor, etc.
Email	Email address of the contact person.
Phone	Phone number of the contact person.
Fax	Fax number of the contact person.
Order Note	Notes to be considered while processing the order.
Invoice Note	Notes to be considered while invoicing the order.
Discount	Some times the customer is eligible for a discount. Discount may be given as a fixed amount or as a percentage of the total invoice amount.

Item Information	
Item Number	Unique code of an item as it exists in the inventory system of North Pole Corporation.
Quantity	Quantity ordered.
Price	Price per case. North Pole Corporation always sells in cases and not in units.

After identifying the data elements, they proceeded to define the structure of the order XML document.

Defining the XML Structure

After a few rounds of discussions, Steve finalized the XML structure and created a sample XML document.

```xml
<OrderInfo AgencyCode="S008">
  <Order OrderNumber="20001">
    <OrderDate>2008-01-01Z</OrderDate>
    <DeliveryDate>2008-01-20T08:00-08:00</DeliveryDate>
    <Customer CustomerNumber="GREAL">
      <CustomerName>Great Lakes Food Market</CustomerName>
      <Billing City="Eugene" State="OR" Zip="97403">
        <Address>2732 Baker Blvd.</Address>
      </Billing>
      <Shipping City="Eugene" State="OR" Zip="97403">
        <Address>2732 Baker Blvd.</Address>
      </Shipping>
      <Terms>30 Days Credit</Terms>
      <Contact Name="Howard Snyder" Title="Purchase Manager">
        <Email>hsnyder@greatlakes.com</Email>
        <Phone>(503) 555-7555</Phone>
        <Fax>(503) 555-2376</Fax>
      </Contact>
    </Customer>
    <Items>
      <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
      <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
      <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
    </Items>
    <OrderNote>Delivery needed before 8 AM</OrderNote>
    <InvoiceNote>
      Adjust the previous credit note with this invoice
    </InvoiceNote>
    <Discount>
      <Amount>300</Amount>
    </Discount>
  </Order>
  <Order OrderNumber="20002">
    <OrderDate>2008-01-01Z</OrderDate>
    <DeliveryDate>2008-01-16T09:00-08:00</DeliveryDate>
    <Customer CustomerNumber="LAZYK">
      <CustomerName>Lazy K Kountry Store</CustomerName>
      <Billing City="Walla Walla" State="WA" Zip="99362">
        <Address>12 Orchestra Terrace</Address>
      </Billing>
      <Shipping City="Walla Walla" State="WA" Zip="99362">
        <Address>12 Orchestra Terrace</Address>
      </Shipping>
      <Terms>30 Days Credit</Terms>
      <Contact Name="John Steel" Title="Store Manager">
        <Email>jsteel@lazykountry.com</Email>
        <Phone>(509) 555-7969</Phone>
        <Fax>(509) 555-6221</Fax>
      </Contact>
```

```
    </Customer>
    <Items>
      <Item ItemNumber="FB001923" Quantity="24" Price="18.25" />
      <Item ItemNumber="FB060020" Quantity="20" Price="12.75" />
      <Item ItemNumber="SG031667" Quantity="48" Price="3.50" />
    </Items>
    <OrderNote></OrderNote>
    <InvoiceNote></InvoiceNote>
    <Discount>
      <Percent>7.5</Percent>
    </Discount>
  </Order>
</OrderInfo>
```

Listing 3.1: Sample XML document with order information.

Most of the elements and attributes in the above XML structure are self explanatory. However, elements like *OrderDate* or *DeliveryDate* might look a little strange as the value contains some characters that we usually do not expect as part of a date value.

The XSD implementation of SQL Server expects time-zone information along with a *date*, *time* or *datetime* values. This is the reason why we have *z* (which stands for UTC time zone) along with the *OrderDate* value. *DeliveryDate* has date and time information that indicates the time at which the delivery is required on the given date. Some customers expect merchandise to be delivered at a predefined time and the Delivery Time is very important for them. Hence, the *DeliveryDate* element should include the delivery time as well as the time zone (GMT -08:00 hours for Eugene) of the cities specified in the *Shipping Address*.

SQL Server 2005's implementation of XSD *date, time* and *dateTime* data types requires time zone information to be present along with the value. Per XSD specification, this is optional information. However, SQL Server's implementation made it mandatory. This behavior changed in SQL Server 2008. Time zone information is optional in SQL Server 2008.

A more detailed description of date, time and dateTime data types is given in Chapter 7.

Defining the Validation Rules

The next step was to define the validation rules. As we discussed earlier, the validation rules are to be defined using an XSD Schema. Before writing the schema, Steve made a list of validations that had to be included in the XSD Schema. He organized the validation rules into a few logical sections to make it simpler to understand and easier while translating them to XSD.

The *Root* Element

The root element of the XML document should be defined as follows:

```xml
<OrderInfo AgencyCode="S008">
   <Order />
   <Order />
   <Order />
</OrderInfo>
```

Listing 3.2: Root element of Order Information XML.

The root element should be validated against the following rules:

- The name of the root element should be *OrderInfo.*
- There may be multiple *Order* elements inside the root element. Each *Order* element will hold information of a single order. If the *OrderInfo* has information of three orders, there should be three *order* elements inside the *OrderInfo* element. *OrderInfo* element should contain at least one *Order* element and there is no maximum limit.
- *OrderInfo* element should have an attribute named *AgencyCode*.
- The *AgencyCode* attribute is mandatory and should be exactly four characters long.
- The first character of the *AgencyCode* should be an alpha character and the other three should be numeric.

The Order Element

The definition of the root element says that it should contain one or more *Order* elements. Each *Order* element should contain complete information about a particular order and should look like the example given below.

```xml
<Order OrderNumber="20002">
   <OrderDate>2008-01-01Z</OrderDate>
   <DeliveryDate>2008-01-16T09:00-08:00</DeliveryDate>
```

```
   <Customer />
   <Items />
   <OrderNote>Delivery needed before 8 AM</OrderNote>
   <InvoiceNote>
     Adjust the previous credit note with this invoice
   </InvoiceNote>
   <Discount />
 </Order>
```

Listing 3.3: Example of an Order element.

The *Order* element should comply with the following rules:

- It should have a mandatory attribute named *OrderNumber.*
- The value of *OrderNumber* attribute should be non-empty.
- This value should not be longer than twenty characters.
- *OrderNumber* can hold any combination of digits as well as letters of the English alphabet (in upper, lower or mixed case).
- *Order* element can have the following child elements:

  o OrderDate
  o DeliveryDate
  o Customer
  o Items
  o OrderNote
  o InvoiceNote
  o Discount

- The child elements of *Order* should follow the same order as given above.
- *OrderDate, DeliveryDate, Customer* and *Items* are mandatory elements.
- *OrderNote, InvoiceNote* and *Discount* are optional elements.
- None of the elements can appear more than once under an *Order* element.
- *OrderDate* should be of *date* type, which should contain a valid *date* value. The *date* value should not contain *time* information.
- *DeliveryDate* should be a *datetime* value, which should contain *date* as well as *time* information.
- If present, *OrderNote* can store a text note as long as 500 characters.
- If present, *InvoiceNote* can store a text note as long as 500 characters.

The *Customer* Element

Each order should contain certain information about the customer who booked the order. The following example shows how a *Customer* element should look.

```
<Customer CustomerNumber="LAZYK">
   <CustomerName>Lazy K Kountry Store</CustomerName>
   <Billing />
   <Shipping />
   <Terms>30 Days Credit</Terms>
   <Contact />
</Customer>
```

Listing 3.4: Example of a Customer element.

The *Customer* element should follow the rules given below:

- Each *Customer* element should contain an attribute named *CustomerNumber.*
- *CustomerNumber* is mandatory, should be EXACTLY five characters long, and contain only alphabets A to Z in upper case.
- A *Customer* element can have the following child elements:
  - CustomerName
  - Billing
  - Shipping
  - Terms
  - Contact
- The child elements of *Customer* should appear exactly in the same order as given above.
- None of the elements should appear more than once under a *Customer* element.
- *CustomerName* is optional and if present should not be more than fifty characters long.
- *Billing* is mandatory.
- *Shipping* is optional. If not present, the address given in *Billing* is assumed to be the shipping location.
- *Terms* is mandatory. It should be one of the following values:
  - 30 Days Credit
  - 60 Days Credit
  - 90 Days Credit
  - Against Delivery
- *Contact* is mandatory.

Billing and *Shipping* Addresses

The *Customer* element should contain billing and shipping information. Only the billing address is mandatory. If the shipping address is not specified, the billing address is assumed to be the shipping location.

Billing and *Shipping* elements have the same structure and they follow the same validation rules. The following example shows how shipping and billing addresses should look.

```
<Billing City="Eugene" State="OR" Zip="97403">
   <Address>2732 Baker Blvd.</Address>
</Billing>
```

Listing 3.5: Example of a Billing Address.

```
<Shipping City="Eugene" State="OR" Zip="97403">
   <Address>2732 Baker Blvd.</Address>
</Shipping>
```

Listing 3.6: Example of a Shipping Address.

As mentioned earlier, *Billing* and *Shipping* elements should follow the same validation rules. Here are the rules that these elements should follow:

- Should have three mandatory attributes: *City, State* and *Zip.*
- The value of *City* should not be empty and should not be longer than thirty characters.
- The value of *State* should be exactly two characters long. Only upper case letters are permitted.
- The length of *Zip* should be exactly five characters and should contain only digits 0 to 9. Leading zeros are not allowed.
- *Address* is mandatory and should not be more than fifty characters long.

The *Contact* Element

Each *Customer* element should contain a *Contact* element. This element contains information about the contact person at the customer's organization. In the event of any queries or communication related to the order, this person should be contacted.

Here is an example of a *Contact* element.

```
<Contact Name="Howard Snyder" Title="Purchase Manager">
   <Email>hsnyder@greatlakes.com</Email>
   <Phone>(503) 555-7555</Phone>
   <Fax>(503) 555-2376</Fax>
</Contact>
```

Listing 3.7: Example of a Contact element.

The contact element should follow the rules given below:

- Should have two mandatory attributes: *Name* and *Title.*
- Value of *Name* should not be empty and should not contain more than twenty characters.
- Value of *Title* should not be empty and should not contain more than twenty characters.
- *Contact* element can have the following child elements:
  - Email
  - Phone
  - Fax
- The child elements should appear exactly in the same order as given above.
- *Email* is mandatory and is expected in the format of string1@string2.string3. The schema should contain only the following simple validations. (Steve did not want to make it too complicated.)
  - Only alpha-numeric characters are allowed in string1, string2 and string3.
  - There should be EXACTLY one "@" sign in the whole email address and it should appear between string1 and string2.
  - There should be at least one "." between string2 and string3.
- *Phone* is mandatory and should be in the following format: (503) 555-7555.
- *Fax* is optional. If present, it should be in the same format as the phone number.

The *Items* Element

Each order should have an *items* element, which contains the details of items ordered. The following example shows the structure of the *Items* element.

```
<Items>
   <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
   <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
   <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
</Items>
```

Listing 3.8: An example of Items element.

The *Items* element can contain one or more *Item* elements. There should be one *Item* element for each item the customer has ordered.

- There should be at least one *Item* element.
- There is no maximum limit.
- Each *Item* element should have three mandatory attributes:
  - ItemNumber
  - Quantity
  - Price
- *ItemNumber* should be exactly eight characters long.
- The first two characters of the *ItemNumber* should be upper case letters [A to Z] and the next six characters should be digits.
- *Quantity* should be a number and should be between 1 and 9999. Decimals are not allowed.
- *Price* should be a number between 0.01 and 999999.99. The value should have two decimal places.

The *Discount* Element

Each *Order* element may contain an optional *Discount* element. Discount may be given in terms of a fixed amount or as a certain percentage of the total invoice amount. Here are some examples of *Discount* element.

```
<Discount>
   <Amount>300</Amount>
</Discount>
```

Listing 3.9: Discount as Amount.

```
<Discount>
   <Percent>7.5</Percent>
</Discount>
```

Listing 3.10: Discount as Percentage.

The *Discount* element should follow the rules given below:

- It should contain either an *Amount* element or a *Percent* element. These two elements are mutually exclusive. Either one of them can be present in a *Discount* element.
- If *Amount* is present, the minimum value should be 0.01. There is no maximum limit. The value should always have two decimals.
- If *Percent* is present, the value should be between 0.01 and 100.00. The value should always have two decimals.

Getting Ready to Write the Schema

Steve and his team are ready to start writing the schema. They have all the input needed. We have had a good look into the information they have collected and documented so far. The next step is to create an XSD schema based on the rules we have seen above.

We will have our first hands-on lab at the end of Chapter 6. In Chapters 4 and 5 we will try to develop the skills needed to write the schema based on the information documented by Steve and his team.

We will develop the schema for the *root* element in Chapter 6, the *Order* element in Chapter 7, the *Customer* element in Chapter 8, the *Billing* and *Shipping* elements in Chapter 9, the *Contact* element in Chapter 10, the *Items* element in Chapter 11 and the *Discount* element in Chapter 12. Then we will combine all the pieces we developed from Chapters 6–12 and will assemble the final schema.

CHAPTER 4
UNDERSTANDING SCHEMA COMPONENTS

An XSD Schema is an XML document. An XML document is primarily composed of elements and attributes. Hence, we could say that the building blocks of an XSD Schema at the most granular level are elements and attributes. Then there are bigger blocks like *Simple Types, Complex Types, Attribute Groups and Modal Groups,* etc. All of these bigger blocks are built using basic components: elements and attributes.

We will try to have a closer look at the basic building blocks of XSD in this chapter. In this chapter we will:

- Examine the building blocks of XSD such as element declarations, *attribute declarations*, *simple types*, *complex types*, *attribute groups*, *modal groups*, *annotations*, etc.
- Learn declaring elements and attributes, adding child elements, defining occurrence and order of child elements.
- Have a quick look into XSD data types and see how to perform basic data validation using XSD.

Building Blocks of XSD

We will examine the building blocks of XSD in this section. I had mentioned earlier that the schema components at the most granular level are elements and attributes. At a higher granular level there are types (simple types, complex types), groups (attribute groups, modal groups), etc. We will examine each of them in detail.

Element Declarations

Writing the XSD code for describing and validating an element in an XML document is called *element declaration.* Each element in the XML instance document (the document to be validated) is represented by an *element declaration.*

In Chapter 2 we saw a basic element declaration. We created a sample schema having a single element. We could say that we wrote a schema having an *element declaration.*

The following example shows an element declaration.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Greetings"/>
</xsd:schema>
```

Listing 4.1: Example of an element declaration.

This *element declaration* describes an XML instance document that has a single element named *Greetings*. This is how the XML instance might look.

```
<Greetings>Hi</Greetings>
```

Listing 4.2: An XML instance document that validates with the schema given in Listing 4.1.

The *element declaration* we created above has an attribute named *name*. This attribute specifies the name of the element expected in the XML instance document. This is just one of the several attributes that an element declaration usually takes. Examples of other attributes are *id*, *type*, etc.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ElementDeclaration'
) BEGIN
     DROP XML SCHEMA COLLECTION ElementDeclaration
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ElementDeclaration AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Greetings"/>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ElementDeclaration)
SELECT @x = '<Greetings>Hi</Greetings>'
```

Listing 4.3: TSQL Code demonstrates an element declaration.

> **We will have a very detailed look into element declarations in Chapter 5.**

Attribute Declarations

The XSD code to describe and validate an attribute in an XML instance document is called *attribute declaration.* Attribute declarations usually comes along with *element declarations.* An attribute cannot exist without a parent element; hence, attribute declarations usually comes along with element declarations. An element declaration need not necessarily be followed with an attribute declaration.

The following example shows an attribute declaration.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="FirstName"/>
</xsd:schema>
```

Listing 4.4: Example of an attribute declaration.

The above schema declares an attribute named *FirstName*. Though this is a valid schema, this does not really make any sense. Attributes cannot exist by themselves. Attributes are always placed within XML elements and should appear along with element declarations. Thus, it is impossible to create an XML document that is valid according to the above schema.

To make this schema meaningful, we need to add an element to it. Here is a schema that describes an XML fragment storing employee information.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.5: Example showing an element declaration as well as attribute declaration.

```sql
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ElementAndAttribute'
) BEGIN
     DROP XML SCHEMA COLLECTION ElementAndAttribute
```

```
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ElementAndAttribute AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
     <xsd:complexType>
        <xsd:attribute name="Name"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ElementAndAttribute)
SELECT @x = '<Employee Name="Jacob"/>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ElementAndAttribute
```

Listing 4.6: A TSQL example showing element and attribute declarations.

Chapter 6 explains attribute declarations in great detail.

Simple Types

When you look at an XML document, you might see elements with different characteristics. Some elements might have child elements and attributes and others may have just a value. An element may have a *simple* type or *complex type* based on its structure/content. It has a *simple type* if it does not have any child element or attribute. If it has an attribute or contains other child elements, it has a *complex type.*

The following example declares an element named *Greetings* and has a *simple type*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Greetings"/>
</xsd:schema>
```

Listing 4.7: XSD example showing a simple type.

This schema describes an XML instance document that looks like the following.

```
<Greetings>Hi</Greetings>
```

Listing 4.8: XML example showing a simple type.

The *Greetings* element does not have child elements or attributes and has a *simple type*.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'SimpleType'
) BEGIN
      DROP XML SCHEMA COLLECTION SimpleType
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION SimpleType AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Greetings"/>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(SimpleType)
SELECT @x = '<Greetings>Hi</Greetings>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION SimpleType
```

Listing 4.9: TSQL Code showing a simple type.

Chapter 8 explains Simple Types in detail.

Complex Types

If an element contains child elements or attributes, it has a *complex type.* The following XML fragment shows an XML element that has an attribute. Because it has an attribute, we could say it has a *complex type.*

A complex type contains elements and/or attributes. Elements and attributes appear within *"<xsd:complexType>"* tag.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
```

```
      <xsd:attribute name="FirstName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.10: Defining a Complex Type.

Let us look at an XML element that has a complex type.

```
<Employee FirstName="Jacob"/>
```

Listing 4.11: A complex type having an attribute.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ComplexType'
) BEGIN
      DROP XML SCHEMA COLLECTION ComplexType
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ComplexType AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ComplexType)
SELECT @x = '<Employee FirstName="Jacob"/>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ComplexType
```

Listing 4.12: TSQL code showing a complex type.

The above example shows a Complex Type having an attribute. A complex type can have attributes, child elements or both. Here is another example that shows a complex type having child elements.

```
<Employee>
  <FirstName>Jacob</FirstName>
</Employee>
```

Listing 4.13: A complex type having a child element.

Employee element has a child element *FirstName* and is a complex type. Here is the schema that describes the above XML instance.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.14: Another complex type example.

```sql
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ComplexType'
) BEGIN
      DROP XML SCHEMA COLLECTION ComplexType
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ComplexType AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ComplexType)
SELECT @x =
'<Employee>
  <FirstName>Jacob</FirstName>
</Employee>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ComplexType
```

Listing 4.15: TSQL code showing a complex type having child elements.

Chapter 10 covers Complex Types in detail.

Attribute Groups

Attribute Groups provide a certain level of reusability of XSD code. If a few attributes are to be declared in more than one complex type, those attributes can be put to an *Attribute Group* and you can simply insert the *attribute group* at the locations where you need the attributes to be present.

Let us look at an example. Look at the following XML fragment.

```
<Employees>
   <Manager FirstName="Jacob" LastName="Sebastian"/>
   <Programmer FirstName="Bob" LastName="Jones"/>
</Employees>
```

Listing 4.16: Employee information XML document.

Both *Manager* and *Programmer* elements have attributes *FirstName* and *LastName*. While writing the schema, an attribute group can be defined which may then be reused in both the elements. Here is the schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:attributeGroup name="FullName">
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
   </xsd:attributeGroup>
   <xsd:element name="Employees">
      <xsd:complexType>
         <xsd:sequence>
            <xsd:element name="Programmer">
               <xsd:complexType>
                  <xsd:attributeGroup ref="FullName"/>
               </xsd:complexType>
            </xsd:element>
            <xsd:element name="Manager">
               <xsd:complexType>
                  <xsd:attributeGroup ref="FullName"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 4.17: Example of an attribute group.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'AttributeGroup'
```

```
) BEGIN
      DROP XML SCHEMA COLLECTION AttributeGroup
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION AttributeGroup AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attributeGroup name="FullName">
    <xsd:attribute name="FirstName"/>
    <xsd:attribute name="LastName"/>
  </xsd:attributeGroup>
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Programmer">
          <xsd:complexType>
            <xsd:attributeGroup ref="FullName"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Manager">
          <xsd:complexType>
            <xsd:attributeGroup ref="FullName"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(AttributeGroup)
SELECT @x =
'<Employees>
  <Programmer FirstName="Bob" LastName="Jones"/>
  <Manager FirstName="Jacob" LastName="Sebastian"/>
</Employees>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION AttributeGroup
```

Listing 4.18: TSQL code showing an attribute group.

Attribute Groups are explained in Chapter 6

Element Groups

Just like *Attribute Groups, Element Groups* also provide a good deal of reusability. You can create a named element group and then add a reference to it at other parts of the schema.

```xml
<Employees>
  <Manager>
    <FirstName>Jacob</FirstName>
    <LastName>Sebastian</LastName>
  </Manager>
  <Programmer>
    <FirstName>Bob</FirstName>
    <LastName>Jones</LastName>
  </Programmer>
</Employees>
```

Listing 4.19: Employee information XML document.

The above example shows an XML fragment containing employee information. Both Manager and Programmer have elements *FirstName* and *Last Name*. The schema of this XML document can use an element group to reuse the declarations for elements *FirstName* and *LastName.*

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:group name="FullName">
    <xsd:sequence>
      <xsd:element name="FirstName"/>
      <xsd:element name="LastName"/>
    </xsd:sequence>
  </xsd:group>
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Manager">
          <xsd:complexType>
            <xsd:group ref="FullName"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Programmer">
          <xsd:complexType>
            <xsd:group ref="FullName"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.20: Example of an element group.

```sql
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ElementGroup'
) BEGIN
    DROP XML SCHEMA COLLECTION ElementGroup
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ElementGroup AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
   <xsd:group name="FullName">
     <xsd:sequence>
       <xsd:element name="FirstName"/>
       <xsd:element name="LastName"/>
     </xsd:sequence>
   </xsd:group>
   <xsd:element name="Employees">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Manager">
           <xsd:complexType>
             <xsd:group ref="FullName"/>
           </xsd:complexType>
         </xsd:element>
         <xsd:element name="Programmer">
           <xsd:complexType>
             <xsd:group ref="FullName"/>
           </xsd:complexType>
         </xsd:element>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ElementGroup)
SELECT @x =
'<Employees>
  <Manager>
    <FirstName>Jacob</FirstName>
    <LastName>Sebastian</LastName>
  </Manager>
  <Programmer>
    <FirstName>Bob</FirstName>
    <LastName>Jones</LastName>
  </Programmer>
</Employees>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ElementGroup
```

Listing 4.21: TSQL code showing an element group.

Element Groups are discussed in Chapter 5.

Order Indicators

Order of elements matters a lot to XML. Most of the time XML parsers would expect elements to be present in a specific order. On the other hand, order of attributes is not significant at all. Attributes can appear in any order and there is no way in XSD to restrict the attributes to be in a specific order.

XSD uses *Order Indicators* to control the order of child elements and they are: *all, sequence* and *choice.*

When *all* is used, the child elements can appear in any order.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="FirstName"/>
        <xsd:element name="LastName"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.22: Example showing "all" indicator.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'OrderIndicatorAll'
) BEGIN
     DROP XML SCHEMA COLLECTION OrderIndicatorAll
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION OrderIndicatorAll AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="FirstName"/>
        <xsd:element name="LastName"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(OrderIndicatorAll)

-- LastName follows FirstName
SELECT @x =
'<Employee>
    <FirstName>Jacob</FirstName>
    <LastName>Sebastian</LastName>
</Employee>'

-- FirstName follows LastName
SELECT @x =
'<Employee>
    <LastName>Sebastian</LastName>
    <FirstName>Jacob</FirstName>
</Employee>'

-- Drop the schema collection
```

```
DROP XML SCHEMA COLLECTION OrderIndicatorAll
```

Listing 4.23: TSQL example showing "all" indicator.

When **sequence** is used, the child elements should appear in exactly the order given in the schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
        <xsd:element name="LastName"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.24: Example showing "sequence" indicator.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'OrderIndicatorSequence'
) BEGIN
    DROP XML SCHEMA COLLECTION OrderIndicatorSequence
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION OrderIndicatorSequence AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
        <xsd:element name="LastName"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(OrderIndicatorSequence)
SELECT @x =
'<Employee>
    <FirstName>Jacob</FirstName>
    <LastName>Sebastian</LastName>
</Employee>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION OrderIndicatorSequence
```

Listing 4.25: TSQL example showing "sequence" indicator.

When **`choice`** is used, only one element from a given set of elements should appear in the XML instance.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentInfo">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="CheckNumber"/>
        <xsd:element name="CreditCardNumber"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.26: Example showing "choice" indicator.

```sql
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'OrderIndicatorChoice'
) BEGIN
     DROP XML SCHEMA COLLECTION OrderIndicatorChoice
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION OrderIndicatorChoice AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentInfo">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="CheckNumber"/>
        <xsd:element name="CreditCardNumber"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(OrderIndicatorChoice)

-- XML Instance having "FirstName"
SELECT @x =
'<PaymentInfo>
    <CheckNumber>1234</CheckNumber>
</PaymentInfo>'

-- XML Instance having "LastName"
SELECT @x =
'<PaymentInfo>
    <CreditCardNumber>****1234</CreditCardNumber>
```

```
</PaymentInfo>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION OrderIndicatorChoice
```

Listing 4.27: TSQL example showing "choice" indicator

Order indicators are explained in Chapter 10.

Occurrence Indicators

The number of times in which an element can appear in an XML document might be very important in most of the cases. An Order information XML document may have only one *Order Date* or *Customer Number*. However, it may have any number of *item* elements.

XSD provides a way to control this by using *occurrence indicators*. Occurrence of an element can be controlled by setting correct values to the facets *minOccurs* and *maxOccurs.*

The following example shows an *order date* element that is mandatory and should appear EXACTLY once.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" minOccurs="1"
maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.28: An element that should appear only once.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create new schema collection
```

```
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" minOccurs="1"
maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x =
'<OrderInfo>
    <OrderDate>2008-01-01</OrderDate>
</OrderInfo>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ExampleSchema
```

Listing 4.29: TSQL code showing a mandatory element that should appear only once.

The following example shows the Item element of an order information XML document that allows one or more *Item* elements.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Item" minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.30: An element that can appear any number of times.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
```

```
           <xsd:element name="Item" minOccurs="1"
                        maxOccurs="unbounded"/>
        </xsd:sequence>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x =
'<OrderInfo>
    <Item>Item 1</Item>
    <Item>Item 2</Item>
    <Item>Item 3</Item>
</OrderInfo>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ExampleSchema
```

Listing 4.31: TSQL code showing a an element that appears more than once.

Occurrence indicators are discussed in Chapter 10.

Annotations are used to add documentation to schema components. By storing this documentation within the defined annotation element, it is readable by humans as well as by other applications (e.g. using XSLT).

Let us add some documentation to the schema that we defined earlier.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation xml:lang="en-us">
      "OrderInfo" stores details of the order.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="OrderInfo"/>
</xsd:schema>
```

Listing 4.32: Example showing an annotation.

Annotations are used to describe the schema only. Presence or absence of annotations does not affect the validation of the XML instance in any manner.

Annotations are explained in Chapter 13.

Working with XSD Data Types

XSD has a number of built-in data types. When we declare an element or attribute, we can associate it with a certain data type and the XSD processor (inside SQL Server) will perform additional validations to make sure that the value is valid for the given data type.

The following XML fragment shows the *name* and *age* of an employee.

```
<Employee Name="Jacob" Age="30" />
```

Listing 4.33: Employee information document.

Here is a schema which validates the above XML instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name" />
      <xsd:attribute name="Age" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.34: Schema to validate the employee XML instance.

The above schema has a shortcoming. It does not validate the data type or the value stored in the attributes. For example, this schema will consider the following XML instances as valid:

```
<Employee Name="Jacob" Age="thirty" />
```

Listing 4.35.a

```
<Employee Name="Jacob" Age="who knows" />
```

Listing 4.35.b

The first example shows the age of an employee as *"thirty,"* which makes sense to a human reader but may not be recognized by an application reading the XML document. For example, if the application needs to do some processing for employees older than 25 this record will cause a problem. In most cases the application may not be able to understand that the text value *"thirty"* stands for the number 30.

The value in the second example does not make sense as well. We need a way to validate this type of data and to make sure that the value stored in the *Age* attribute is a valid numeric value. This is one of those cases where the XSD Data Types will help us.

Let us rewrite our declaration of *Age* attribute using XSD *integer* data type.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name" />
      <xsd:attribute name="Age" type="xsd:integer"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.36: Declaration of an attribute using xsd:integer data type.

Note the usage of *type* attribute, which is set to *xsd:integer*. This is an instruction to the XSD processor to consider the value of this attribute as an *integer* value and to validate it.

Let us create a schema collection and see the validation in action.

```
CREATE XML SCHEMA COLLECTION DataTypeTest AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name" />
      <xsd:attribute name="Age" type="xsd:integer"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 4.37: Schema collection with data type validation.

Let us validate an XML variable given in L*isting 4.11.a* against this schema.

```
DECLARE @x XML(DataTypeTest)
SET @x = '<Employee Name="Jacob" Age="thirty" />'
```

Listing 4.38: Incorrect value for "Age" which is declared as integer.

If you run the above code, SQL Server will generate the following error:

```
(local)(JAC\JACOB): Msg 6926, Level 16, State 1, Line 2
XML Validation: Invalid simple type value: 'thirty'. Location:
/*:Employee[1]/@*:Age
```

Let us correct the data and try again.

```
DECLARE @x XML(DataTypeTest)
SET @x = '<Employee Name="Jacob" Age="30" />'
```

Listing 4.39: Example of an XML instance with correct integer value.

> XSD data types are explained in Chapter 7. *Integer* is a *Derived Data Type* and Chapter 9 covers integers and other XSD derived data types in detail.

Performing Basic Data Validations

We just saw the importance of XSD data types to validate the value stored in elements and attributes; most of the time we need a few additional levels of validation other than just the validation of data types. For example, the primary validation on the age of an employee could be that the value should be a number. But we may still need additional validations to make sure that the value is within a given range.

Each data type has a number of properties that can be restricted to perform additional validations on the value. The string data type can have *length*; numbers can have *min* and *max,* etc. These properties are called *Facets* in XSD.

So each data type has a certain number of predefined Facets. You can apply restrictions to these facets to perform additional validations on the value stored in an element or attribute. We will see a few basic validations in the following sections. More detailed validations are demonstrated in Chapter 7 – XSD Primitive Data Types.

Range Validation

We have seen how to validate the data type of an element or attribute. We have seen the example of the *"age"* attribute earlier. There may be times when we will come across cases where we need a few additional levels of validation.

For example, by declaring *Age* as *"xsd:integer,"* we applied a certain level of restriction on the value. With this validation values like *"Blah"* or *"Ten"* should be restricted. However, what if the *Age* is set to *-10* or *866?* Those

values are indeed integers but they are not acceptable as a valid value for the *Age* attribute of an employee.

This raises the need for another level of validation. We need a way to specify the minimum and maximum ranges of numeric values. We need to be able to set the number of decimals a numeric value can take. We need to be able to set the length of string values. And so on.

Such restrictions can be achieved in XSD using the *facets* of a data type. A *facet* controls a certain attribute or characteristic of a data type. For example, *integer* data type has a facet *minInclusive,* which specifies the smallest value it can store. *String* data type has a *facet* named *maxLength,* which specifies the maximum length of the value.

> **Facets of data types are explained in Chapter 7 – XSD Primitive Data Types**

Let us see how to rewrite our *DataTypeTest* schema so that it applies a restriction on the *Age* attribute. Let us restrict the age to be between 18 and 55 inclusive. (18, 55 and any other number between them are acceptable.)

The previous version of attribute definition was very simple and basic. We declared an *attribute* element and assigned values to the *name* and *type* attributes. To apply a restriction on any of the *facets* of an attribute we need to declare the attribute as a *simpleType.* So, first of all, we need to alter the definition from:

```
<xsd:attribute name="Age" type="xsd:integer"/>
```

Listing 4.40: An attribute with integer data typeTo:

```
<xsd:attribute name="Age" type="xsd:integer">
  <xsd:simpleType>
    <!-- Other definition here -->
  </xsd:simpleType>
</xsd:attribute>
```

Listing 4.41: Basic declaration of a simpleType.

After the simpleType is defined, we need to add a restriction element to it.

```
<xsd:attribute name="Age" type="xsd:integer">
   <xsd:simpleType>
     <xsd:restriction>
       <!-- restrictions here -->
     </xsd:restriction>
   </xsd:simpleType>
</xsd:attribute>
```

Listing 4.42: A simpleType with restriction element.

A restriction always works based on a particular data type. The restrictions applicable for *strings* may not be applicable for *numeric* data types. For example, *maxLength* makes sense to *strings* but is meaningless to numeric values. Similarly, *minInclusive* and *maxInclusive* are applicable to numeric data types but they do not make sense with *string* data types.

Hence, we need to specify on which data type the restrictions should be based when you declare *xsd:restriction*. The *base* attribute of *xsd:restriction* should be set to the desired data type. Here is an updated version of the schema.

```
<xsd:attribute name="Age" type="xsd:integer">
   <xsd:simpleType>
     <xsd:restriction base="xsd:integer">
       <!-- restrictions here -->
     </xsd:restriction>
   </xsd:simpleType>
</xsd:attribute>
```

Listing 4.43: Restriction with base attribute.

Note that we need to remove the *type* definition from the *attribute* declaration. A *simpleType* cannot take a *type* attribute. The *base* attribute of *xsd:restriction* is used to identify the data type of an element or attribute when it is declared as a *simpleType.*

The next step is to apply restrictions on the *facets* of our choice. We need to restrict the value of *age* between *18* and *55*. Let us use the *minInclusive* and *maxInclusive* facets to set this restriction. Here is the version of the schema that includes this restriction.

```
<xsd:attribute name="Age">
   <xsd:simpleType>
     <xsd:restriction base="xsd:integer">
       <xsd:minInclusive value="18"/>
       <xsd:maxInclusive value="55"/>
```

```
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

Listing 4.44: Restricting the attribute value between two numeric values.

> **Refer to Chapter 7 for a detailed description of all the data types and their facets.**

Let us test this schema with SQL Server. Let us drop the previous Schema Collection and create the new version.

```
-- DROP the previous version
DROP XML SCHEMA COLLECTION DataTypeTest
GO

-- CREATE the new version
CREATE XML SCHEMA COLLECTION DataTypeTest AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name" />
      <xsd:attribute name="Age">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="18"/>
            <xsd:maxInclusive value="55"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 4.45: Code to create the new version of "DataTypeTest."

Let us create a TYPED XML variable bound to this schema and see if SQL Server validates the data correctly.

```
DECLARE @x XML(DataTypeTest)
SET @x = '<Employee Name="Jacob" Age="60" />'
```

Listing 4.46: An invalid XML fragment.

If you run this code, you will notice the following error:

```
Msg 6926, Level 16, State 1, Line 2
XML Validation: Invalid simple type value: '60'. Location:
/*:Employee[1]/@*:Age
```

60 is an invalid value per the Schema Collection we just created. *Age* can accept only a value between *18* and *55*. Here is a correct XML value.

```
DECLARE @x XML(DataTypeTest)
SET @x = '<Employee Name="Jacob" Age="55" />'
```

Listing 4.47: A valid XML fragment for Schema Collection "DataTypeTest."

Length Validation

String data types have a different set of *facets* than numeric data types. We have seen two facets of numeric data types in the previous section. Now, let us see how to restrict the length of a string value.

Here is a schema that restricts employee ID to 6 characters.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="ID">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:length value="6"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.48: Example of restricting the length of a string.

Refer to Chapter 7 for a detailed description of all the data types and their facets.

Format Validation

One of the common validations applied on a string value is format validation. Certain values are expected to be in a specified format (for example, phone numbers, email addresses, etc).

Format validation is done in XSD using the *pattern* facet. The *pattern* facet takes an expression which specifies the required format of the value. XSD supports a regular expression language, which is very close to the regular

expression languages used by programming languages like Perl or Microsoft .NET.

> **A detailed explanation of the regular expression language supported by XSD is explained in Chapter 12.**

In this section we will see a simple example which validates the format of a value using *pattern* facet. We will not touch this regular expression language in detail just yet as later in the book I have dedicated a complete chapter to the regular expression language of XSD.

Let us take the XML fragment we used for the other examples in this chapter.

```xml
<Employee Name="Jacob" />
```

Listing 4.49: Sample XML to store employee information.

Let us add a restriction to the name of the employee. Let us insist that the employee name should always be in upper case. Here is the schema which declares the pattern to perform this validation.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z]+"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 4.50: Example showing pattern restriction.

The expression *"[A-Z]"* stands for any character from "A" to "Z" in upper case. The plus sign (+) stands for one or more occurrence of the preceding character. So a combination of *"[A-Z]"* and *"+"* stands for *"one or more occurrence of letters A to Z in upper case."*

> **Chapter 12 has a number of examples that explain how to create patterns for various format validation requirements.**

Chapter Summary

In this chapter we have seen the basic building blocks of XSD. An XML document is composed of elements and attributes. An XML schema describes and validates XML documents. Description and validation of the XML elements and attributes are written in XSD using *element declarations* and *attribute declarations.*

An element may have a *simple type* or *complex type*. If an element contains attributes, child elements, or both, it has a *complex type*; otherwise, it has a *simple type.*

Commonly used attributes and elements can be grouped together into *Attribute Groups* and *Element Groups*. You can then refer to such a group at multiple locations in your schema definition. *Attribute groups* and *Element groups* provide a certain level of reusability.

Most of the time the physical position or order of elements within a parent node is significant in an XML document. You can control the order of elements using *XSD Order Indicators*. If you want the elements to appear in a specific order, you could use *sequence* order indicator. Or if you don't care about the order, you could use *all* indicator. Sometimes you would need only one element out of a set of given elements. You could use *choice* indicator in such situations.

Occurrence of elements is important, too. Certain elements may be optional, certain elements mandatory, and some others may appear more than once. Attributes can appear only once (or may not appear if they are optional), whereas elements may appear even more than once. Occurrence of elements is controlled in XSD using *minOccurs* and *maxOccurs* indicators. You can set an element to *optional* by setting *minOccurs* to 0.

Documentation is important in any piece of code that we write. XSD provides a documentation mechanism using *annotations.* Annotations can be used to add documentation helpful to humans as well as to applications that process/read the schema document.

To perform validations on the value of an element or attribute, XSD supports the concept of data types. When an element or attribute is

declared with a data type, the schema processor will validate the value against the given data type.

Each XSD data type has a certain characteristics/properties known as *facets*. Length of a string or minimum or maximum values of a number are examples of such facets. These facets can be restricted to perform additional validations on the value stored in a given element or attribute. Examples of most common validations are range validations on numeric values and length or format validations on string values.

CHAPTER 5

UNDERSTANDING ELEMENT DECLARATIONS

An XSD element declaration represents an element in the XML instance document. In Chapter 4 we covered the basics of element declarations. In this chapter we will have a closer look at element declarations. We will learn the following:

- Element Declarations
- Global and local element declarations
- Different parameters that an element declaration takes
- A detailed look at the *xsi:nil* attribute and handling of null values
- Creating *variable content containers* with substitution groups
- Different ways of controlling element substitution

Let us examine these points in detail.

Element Declaration

Elements are broadly classified into *Simple Types* and *Complex Types*. If an element has attributes and/or child elements, it is said to have a *Complex Type*. An element is said to have a *Simple Type* if it does not have any attributes and does not have child elements. *Simple Types* can only store a value.

Let us look for a second at the following XML fragment:

```
<Employee EmployeeID="101">
   <Name>
      <FirstName>Jacob</FirstName>
      <LastName>Sebastian</LastName>
   </Name>
   <Age>30</Age>
   <Phone Location="home">999 888 7777</Phone>
</Employee>
```

Listing 5.1: An XML fragment which contains Simple and Complex Types.

The *"Employee"* element should be declared as *complexType* in the SCHEMA because it contains an attribute (*EmployeeID*) and child elements (*Name, Age and Phone*). The *"Name"* element is also a candidate to have a *complexType* because it contains child elements: *FirstName* and *LastName.* The *Phone* element should have a *complexType* as well, because it contains an attribute (*Location*).

On the other hand, the XSD declaration of *FirstName, LastName* and *Age* should have a *simpleType* because they contain only a value and do not have a child element or attribute.

> *Simple Types* and *Complex Types* are explained in Chapters 8 and 10 respectively.

An XSD schema which validates the above XML fragment might look like this.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="FirstName" />
              <xsd:element name="LastName" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Age" />
        <xsd:element name="Phone">
          <xsd:complexType>
            <xsd:attribute name="Location" />
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="EmployeeID" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.2: A schema showing Simple Types and Complex Types.

All the lines above that are marked in yellow are examples of element declarations. The declarations of *"FirstName"* and *"LastName"* are very simple. On the other hand, the declaration of *"Employee"* or *"Name"* is much more complex and detailed (note that they have child elements).

Global And Local Element Declaration

An XSD schema may contain *Global* and *Local* element declarations. *Global* element declarations are top level element declarations right under the *schema* element. *Local* element declarations usually appear inside a *complexType* or *element group* declaration.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:complexType>
      <xsd:attribute name="First"/>
      <xsd:attribute name="Last"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:all>
        <xsd:element ref="Name"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.4: Example of a "Global" element declaration.

Note the declaration of the global element *"Name."* It is declared right under the *"schema"* element. The advantage of using a global element declaration is that it can be reused (referred) at other locations from within the same schema. Note the usage of attribute *"ref"* to refer to a global element declaration.

Let us create a schema collection and see this in action.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:complexType>
      <xsd:attribute name="First"/>
      <xsd:attribute name="Last"/>
```

```
      </xsd:complexType>
   </xsd:element>

   <xsd:element name="Employee">
     <xsd:complexType>
       <xsd:all>
          <xsd:element ref="Name"/>
       </xsd:all>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Employee>
      <Name First="Jacob" Last="Sebastian"/>
</Employee>'

-- Drop the schema collection
DROP XML SCHEMA COLLECTION ExampleSchema
```

Listing 5.5: An example showing a global element declaration.

One of the interesting things about this schema is that it allows two different XML structures.

```
-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Employee>
      <Name First="Jacob" Last="Sebastian"/>
</Employee>'
```

Listing 5.6: This XML instance will be validated against global element "Employee."

This is a valid XML instance. The *Employee* element is declared in the schema collection as the root element of the XML instance document. It has a child element named *Name* which contains two attributes: *First* and *Last* that represent *First Name* and *Last Name* respectively.

However, the schema collection also allows the following XML instance.

```
-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Name First="Jacob" Last="Sebastian"/>'
```

Listing 5.7: This XML instance will be validated against global element "Name."

The reason is that the schema contains two global elements: *Name* and *Employee.* SQL Server accepts any XML instance that successfully validates with any of the global elements declared in the schema.

> **Often you won't want this behavior, but you will want reusability benefits of a global type. In Chapter 12 we'll show you how to achieve this by using a named complex type.**

Assume that the XML instance we intend to validate is the following.

```
<Employee>
   <Name First="Jacob" Last="Sebastian"/>
</Employee>
```

Listing 5.8: Employee information XML document.

We have already seen how to write a schema to describe this XML fragment using a global element declaration (Listing 5.4). Now let us see how to write the same schema using a *local element declaration.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
      <xsd:complexType>
         <xsd:all>
            <xsd:element name="Name">
               <xsd:complexType>
                  <xsd:attribute name="First"/>
                  <xsd:attribute name="Last"/>
               </xsd:complexType>
            </xsd:element>
         </xsd:all>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 5.9: A schema that uses a local element declaration to describe employee information.

This schema uses a local element declaration to describe the *Name* element. Note that the *Name* element is declared NOT under the *schema* element, but under the *Employee* element.

When you have a global element declaration, you can refer it in multiple locations in your schema. This gives you a certain level of reusability. Local element declarations cannot be reused. If you want the element to appear in multiple places, you need to repeat the declaration code. You cannot add a reference to a local element declaration.

Element Declaration Parameters

Declaration of an element can take a number of attributes/parameters that perform different levels of validation. The shortest declaration of an element is:

```
<xsd:element name="Name">
```

Listing 5.10: A basic element declaration.

The above element declaration shows an element with a single attribute: *"name."* It is a mandatory attribute. You cannot declare an element without the *name* attribute.

The only mandatory attribute that an element declaration should take is the *"name"* attribute. There are a few other optional attributes that an element declaration can take, depending on whether it is declared globally or locally.

All element declarations (global as well as local) can take the following attributes.

- ***name***
- ***type***
- ***id***
- ***default***
- ***fixed***
- ***nillable***
- ***block***

In addition to the above attributes, a *global* element declaration can take the following attributes. These attributes are not permitted in a *local* element declaration.

- ***final***
- ***abstract***
- ***substitutionGroup***

Local element declarations can have the following additional attributes. These attributes are not permitted in a *global* element declaration.

- ***minOccurs***
- ***maxOccurs***
- ***ref***
- ***form***

Let us look at these attributes in detail.

Attribute: *name*

This is the only mandatory attribute that an element declaration should have. The following example describes an XML element with the name ***"Age."***

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Age" />
</xsd:schema>
```

Listing 5.11: An element declaration with "name" attribute.

There are a few points to be aware of while naming your XML elements. An XML element name should follow the rules given below.

- Name of an element should start with a *letter* or an *underscore (_).*
- Name of an element can contain letters, numbers and other characters. However, the most commonly used characters are letters *"a"* to *"z"* in lower and upper case, digits 0 to 9, underscore, period and hyphen.
- Name of an element cannot start with the word *"xml"* in uppercase, lower case or mixed case.
- Name of an element cannot contain spaces.

It is a good practice to give meaningful names to the elements. It is also important that you use correct casing of letters. It is a recommended practice to use either *camel Case ("employeeName")* or *Pascal Case ("EmployeeName")*.

You could also name elements like *"_employeeName," "employee.name,"* or *"employee_name," or "employee-name."* But such naming can be confusing to applications that process the XML document. Though the element name is valid, it might cause trouble for some applications that do not expect such characters to be part of an element name. If your XML instance contains database related information, it is a good practice to follow the database naming rules by avoiding special characters and reserved keywords.

It is good to be descriptive while giving names to XML elements. *"SalesOrderNumber"* or *"SalesPersonCode"* is more informative than *"ordno"* or *"srcode."* However, when you have a hierarchy of elements, sometimes the child elements can be short without losing clarity of information.

Let us look at an example:

```
<Order>
   <SalesRepName>Jacob</SalesRepName>
   <SalesRepCode>1000</SalesRepCode>
</Order>
```

Listing 5.12: A long name (SalesRepName) is more meaningful most of the time.

```
<Order>
   <SalesRep>
      <SalesRepName>Jacob</SalesRepName>
      <SalesRepCode>1000</SalesRepCode>
   </SalesRep>
</Order>
```

Listing 5.13: A long name (SalesRepName) looks like overkill under the 'SalesRep' element.

```
<Order>
   <SalesRep>
      <Name>Jacob</Name>
      <Code>1000</Code>
   </SalesRep>
</Order>
```

Listing 5.14: A short name (name) is good enough to refer to the name of a sales rep when it is put under the 'SalesRep' element.

"SalesRepName" and *"SalesRepCode"* looks good when they are the children of *"Order"* element (*Listing 5.12*). However, they look a little odd in the second example (*Listing 5.13*) where the parent element is *"SalesRep."* Under the element *"SalesRep"* it will be more reasonable to add the elements *"Name"* and *"Code"* rather than using *"SalesRepName"* and *"SalesRepCode"* (*Listing 5.14*).

Attribute: *type*

"type" adds more restrictions on the content of the XML element. *"type"* adds data type related validations to the declaration of an element. When

"type" is not specified, the element is assumed to be *"xsd:anyType,"* which allows any text value in the column. By adding *"type"* we can restrict the value to a given data type. For example, the value of an element declared as *"xsd:integer"* can accept only a numeric value. It cannot have a decimal component, nor can it accept a non-digit character.

Let us look at another version of the employee information XML fragment that we saw earlier in this chapter.

```
<Employee>
   <Name>Jacob</Name>
   <Age>30</Age>
</Employee>
```

Listing 5.15: Employee information.

Here is a basic schema that validates the above XML structure.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Name" />
         <xsd:element name="Age" />
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 5.16: SCHEMA definition for the XML presented in Listing 5.15.

This SCHEMA will validate the XML structure but none of the values stored in the elements. For example, the following XML fragments will be accepted as valid by the above schema.

```
<Employee>
   <Name>Jacob</Name>
   <Age>forty</Age>
</Employee>
```

Listing 5.17.a: This XML makes sense to a human reader. However, it may not be meaningful to an application that processes the XML document. Though we expect the "Age" to be specified as a numeric value, the SCHEMA does not have a validation for this.

```
<Employee>
   <Name>Jacob</Name>
   <Age>who knows?</Age>
</Employee>
```

Listing 5.17.b: This XML is not meaningful. However, since the SCHEMA does not have a validation for the value, this XML fragment will be accepted as a valid piece of XML.

To fix the issues we saw in Listing 5.17.a and 5.17.b, let us add a "type" declaration to the element declaration.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Name" />
         <xsd:element name="Age" type="xsd:integer"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 5.18: The schema declares "Age" as "xsd:integer." This will validate the "Age" element so that it will contain only integer values.

The new version of the schema (*Listing 5.18*) will make sure that the *"Age"* element should contain only a numeric value. The *"integer"* data type will allow only a numeric value and, hence, the two XML fragments we saw earlier (*Listing 5.17.a and 5.17.b*) will not be accepted anymore.

> **XSD data types are explained in Chapter 7. XSD supports a few dozen different data types. We will examine them in detail in Chapters 7 and 9.**

While the new version of the schema (*Listing 5.18*) makes sure that the *"Age"* element will contain only numeric values, there can still be problems. Look at this example.

```
<Employee>
   <Name>Jacob</Name>
   <Age>-30</Age>
</Employee>
```

Listing 5.19: This XML does not make sense. "Age" cannot be negative. However, the schema we saw in listing 6.9 does not restrict negative values.

A negative value does not make the "Age" element meaningful. Let us change the data type to "xsd:positiveInteger" to make sure that a negative value is not accepted in the "Age" element.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" />
        <xsd:element name="Age" type="xsd:positiveInteger"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.20: This version of the schema makes sure that the "Age" element does not take a negative value.

> **We might still need restrictions to make the value more meaningful. For example, the value "200" or "0" does not make any sense when provided as the "Age" of an employee. Such validations can be added by using "xsd:restriction" and applying restrictions on the facets of the given data type. We will examine this when we discuss Simple Types in Chapter 8.**

The example given above uses one of the built-in data types of XSD. You could also use a *simpleType* or *complexType* that you have defined globally in your schema. Chapter 8 explains Simple Types and Chapter 10 covers Complex Types.

Attribute: *id*

"id" is an optional attribute which uniquely identifies the given element. If present, it should be unique within the SCHEMA document.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" id="EmpName" />
        <xsd:element name="Age" id="EmpAge"
                     type="xsd:nonNegativeInteger"/>
      </xsd:sequence>
```

```
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 5.21: The 'id' attribute has significance only within the schema document.

Usage of *"id"* attribute has nothing to do with the XML instance. *"id"* is used only to identify the XSD objects. In no way will it affect the XML instance, its structure, or the values stored in elements and attributes. The same XML data we used in the previous example will validate against this schema, too.

Attribute: *default*

The attribute *"default"* is used to specify the default value of an element. It is applicable only if the element is optional. When the XML data is assigned to the column/variable, if an element is empty and a default value is specified for that element in the schema, the default value is assigned to the element.

Let us look at an example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OS" type="xsd:string" default="Windows XP"/>
</xsd:schema>
```

Listing 5.22: An element declaration with a 'default' value.

The above SCHEMA specifies that the default value of **"OS"** element is "Windows XP." Let us create an XML SCHEMA CLLECTION and test a few scenarios.

```
CREATE XML SCHEMA COLLECTION DefaultElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OS" type="xsd:string" default="Windows XP"/>
</xsd:schema>'
GO
```

Listing 5.23: Creates a new XML SCHEMA COLLECTION named "DefaultElement."

```
DECLARE @x XML(DefaultElement)
SET @x = '<OS />'

-- Read the value from the XML variable
SELECT
```

```
      @x.value('(/OS)[1]', 'VARCHAR(20)') AS OS

/*
OUTPUT:

OS
--------------------
Windows XP

(1 row(s) affected)
*/
```

**Listing 5.24: Note that a blank value is assigned to the XML variable.
However, when reading the value we see that it is filled with the default
value.**

> **In the above example I have used XQuery to read values
> from the XML variable. I will show you some simple usages
> of XQuery in this book. However, a detailed explanation is
> beyond the scope of this book. I have given a detailed
> XQuery tutorial in my upcoming book XQuery For SQL
> Server Developers.**

The default value of an element is similar to the default value that we
assign to the column of a table. When no value is inserted to the column,
the default value is stored. Similarly, when you store an empty value to an
element that has a *"default"* value specified, SQL Server will store the
default value in the element.

The following code is equivalent to the previous one and will produce the
same result.

```
DECLARE @x XML(DefaultElement)
SET @x = '<OS></OS>'


-- Read the value from the XML variable
SELECT
      @x.value('(/OS)[1]', 'VARCHAR(20)') AS OS


/*
OUTPUT:
OS
--------------------
```

```
Windows XP

(1 row(s) affected)
*/
```

Listing 5.25: The schema processor assigns the default value to the element if the value is missing in the XML instance.

The *"default"* value does not have any effect if the element contains a value. The following example demonstrates it.

```
DECLARE @x XML(DefaultElement)
SET @x = '<OS>Linux</OS>'

-- Read the value from the XML variable
SELECT
      @x.value('(/OS)[1]', 'VARCHAR(20)') AS OS

/*
OUTPUT:
OS
--------------------
Linux

(1 row(s) affected)
*/
```

Listing 5.26: In the above example, the "default" value specification has no effect because the element is not empty.

The *default* value will be inserted to the XML element only if the element is present and is empty. If the element is not present, then it is not added automatically. The *default* specification has no effect when the element is absent in the XML instance.

The value assigned to the *default* attribute must be of the correct data type. For example, it is illegal to specify a date value as the default value of an integer element.

There is an exception to the rules we have defined above. The default value of an element will not be assigned if the element is declared as in the schema and the XML instance contains an attribute as is explained later in this chapter.

Attribute: *fixed*

The attribute *"fixed"* is used to restrict the value of an element to a pre-defined value. If the value is missing, then the predefined value is assigned to the element. If the value is present, it must be the same value as defined in the schema. If you try to assign a different value, SQL Server will generate an error.

"fixed" and *"default"* are mutually exclusive. Either one of them can be present in the declaration of an element.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OS" type="xsd:string" fixed="Windows"/>
</xsd:schema>
```

Listing 5.27: Example using "fixed" element.

In this example, the element **OS** should always take the predefined value: **"Windows."** If the XML fragment contains any other value, SQL Server will raise an error.

```sql
-- Create the XML SCHEMA collection
CREATE XML SCHEMA COLLECTION FixedElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OS" type="xsd:string" fixed="Windows"/>
</xsd:schema>'
GO

-- Declare a variable bound to the schema
DECLARE @x XML(FixedElement)
SET @x = '<OS>Linux</OS>'
```

Listing 5.28: 'Linux' is not an accepted value because the element is declared with a fixed value 'Windows.'

The above code will generate the following error:

```
Msg 6921, Level 16, State 1, Line 4
XML Validation: Element or attribute 'OS' was defined as fixed,
the element value has to be equal to value of 'fixed' attribute
specified in definition. Location: /*:OS[1]
```

Listing 5.29: When an element is declared as "fixed" it cannot accept a different value.

If you provide a value, it must be **Windows**. Note that XSD is case sensitive. The following will throw an error as well, because the letter **W** in **Windows** is expected in **Upper Case**.

```
-- Declare a variable bound to the schema
DECLARE @x XML(FixedElement)
SET @x = '<OS>windows</OS>'
```

Listing 5.30: values defined with 'fixed' attributes are case sensitive.

The following is the correct XML.

```
-- Declare a variable bound to the schema
DECLARE @x XML(FixedElement)
SET @x = '<OS>Windows</OS>'
```

Listing 5.31: Example showing the correct XML instance for schema defined in Listing 5.27.

Just as in the case of **"default,"** if the element is empty the value defined in the **"fixed"** attribute will be assigned to the element.

```
-- Declare a variable bound to the schema
DECLARE @x XML(FixedElement)
SET @x = '<OS></OS>'

-- Read the value from the XML variable
SELECT
      @x.value('(/OS)[1]', 'VARCHAR(20)') AS OS

/*
OUTPUT:
OS
--------------------
Windows

(1 row(s) affected)
*/
```

Listing 5.32: If an element declared with the 'fixed' attribute is empty, the schema processor assigns the 'fixed' value to the element.

Attribute: *nillable*

Several times we come across empty elements in an XML instance. Often they have different meanings. There are times when an empty element does not make any sense and can therefore be dropped from the XML instance. There are also times when an empty element is meaningful, even

though it is empty. A familiar example is the *XHTML </br>* element, which is always empty.

An empty element may be interpreted differently by different applications. Some applications might make it mandatory to have an element present in the XML instance, even though it is empty. On the other hand, some applications will generate a validation error if the element is empty. There are times when you really want to keep an empty element to indicate that the information is missing. However, this missing information should be distinguished from another having a blank value. XSD provides a way to do that using *xsi:nil.*

Blank vs. *Missing* values

In database programming, we sometimes use NULL to indicate that the value is missing. A missing value is different from an empty value. Numeric columns can have different meaning for 0 and NULL. In one of my applications we used a bit for the *gender* attribute which took values 0, 1 and NULL. 0 stood for *female*, 1 stood for *male* and *null* stood to indicate that the information is not available yet.

Many applications use NULL to indicate that the value is not initialized. If a value is not available (missing) it may be assigned with NULL. In such cases, we need a way to differentiate between a missing value from an empty value (empty string in case of text column or 0 for a numeric column).

The same is true for XML. When we process an XML document, we need a way to differentiate an element without a value from another element that has an empty string as its value. XSD does not have a data type equivalent to the NULL value in SQL Server. None of the XSD data types can store a NULL value. The effect of a NULL can be produced in an XML instance by declaring the element as *"nillable."*

Let us look at an example which explains this. Here is the SCHEMA of an employee element.

```
CREATE XML SCHEMA COLLECTION NillableElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name" type="xsd:string" />
              <xsd:element name="Remarks" type="xsd:string" />
```

```
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 5.33: Employee Schema.

Now, look at the following XML fragment which validates against the above XML SCHEMA.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employees>
  <Employee>
    <Name>Jacob</Name>
    <Remarks>  </Remarks>
  </Employee>
  <Employee>
    <Name>Steve</Name>
    <Remarks/>
  </Employee>
</Employees>'

SELECT
      e.value('Name[1]','VARCHAR(20)') AS Name,
      e.value('Remarks[1]','VARCHAR(20)') AS Remarks
FROM @x.nodes('/Employees/Employee') x(e)

/*
Name                Remarks
------------------- --------------------
Jacob
Steve
*/
```

Listing 5.34: It is difficult to differentiate an empty element from another element that is blank or having one or more spaces.

Look at the above XML fragment. The first employee element (*"Jacob"*) has a *Remarks* element which contains a space. The second employee element (*"Steve"*) has an empty *Remarks* element. Look at the query results. Both rows show empty strings. How do we distinguish between the two?

This can be done by making the element *"nillable."* Here is the version of the schema with *"Nillable"* attribute.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
```

```
            <xsd:element name="Employee" maxOccurs="unbounded">
              <xsd:complexType>
                <xsd:sequence>
                  <xsd:element name="Name" type="xsd:string" />
                  <xsd:element name="Remarks" type="xsd:string"
                               nillable="true"/>
                </xsd:sequence>
              </xsd:complexType>
            </xsd:element>
          </xsd:sequence>
        </xsd:complexType>
      </xsd:element>
</xsd:schema>
```

Listing 5.35: An example using 'xsd:nillable.'

Let us update the XML Schema Collection. As discussed previously, we cannot update a Schema Collection. We need to drop the previous Schema Collection and create a new one.

```
DROP XML SCHEMA COLLECTION NillableElement
GO
CREATE XML SCHEMA COLLECTION NillableElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Name" type="xsd:string" />
              <xsd:element name="Remarks" type="xsd:string"
                           nillable="true"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 5.36: Employee Schema with nillable element.

Here is the new XML instance. You will notice a few changes here. First of all, the element is declared with a namespace. The second difference is the presence of attribute *"xsi:nil"* in the second employee element.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employees xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  <Employee>
    <Name>Jacob</Name>
    <Remarks> </Remarks>
  </Employee>
  <Employee>
```

```
     <Name>Steve</Name>
     <Remarks xsi:nil="true"/>
   </Employee>
</Employees>'

SELECT
      e.value('Name[1]','VARCHAR(20)') AS Name,
      e.value('Remarks[1]','VARCHAR(20)') AS Remarks
FROM @x.nodes('/Employees/Employee') x(e)

/*
Name                   Remarks
-------------------    --------------------
Jacob
Steve                  NULL
*/
```

Listing 5.37: An XML instance that shows an element with 'xsi:nil.'

"xsi:nil" is declared in the namespace *"http://www.w3.org/2001/ XMLSchema-instance."* Hence, we need to declare it in the XML instance. Note that the query returns NULL for the *remark* element of the second *Employee* row.

Excluding Validation of NULL Values

Sometimes we need to apply a validation on values if they are not NULL. However, no validation may be needed if the value is null. For example, think of a validation rule that says, *"if present, the Age of an employee should be between 22 and 55."* How do we implement such a rule? Let us try to build such a schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Age">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="22"/>
              <xsd:maxInclusive value="55"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
```

```
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.38: A schema that restricts the 'Age' element to values between 22 and 55 inclusive.

The above schema restricts the *"Age"* element to be between 22 and 55 inclusive. Now let us create an XML Schema Collection with the schema definition we just saw.

```
DROP XML SCHEMA COLLECTION NillableElement
GO

CREATE XML SCHEMA COLLECTION NillableElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Age">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="22"/>
              <xsd:maxInclusive value="55"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
'
```

Listing 5.39: Schema collection that applies a range restriction on age.

Here is an XML instance which validates with the above Schema Collection.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employee>
    <Name>Jacob</Name>
    <Age>22</Age>
</Employee>'
```

Listing 5.40: "Age" should be between 22 and 55 as per the restrictions given in the Schema Collection.

Though the above Schema Collection restricts *"Age"* to be between 22 and 55 and does not allow it to be NULL, the rule that we were trying to

implement should allow NULL values. The validation should occur only if the value is not NULL.

If you try the below code, you will get a validation error.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employee>
    <Name>Jacob</Name>
    <Age></Age>
</Employee>'
```

Listing 5.41.a: This will generate a validation error because the value of Age should be between 22 and 55.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employee>
    <Name>Jacob</Name>
    <Age />
</Employee>'
```

Listing 5.41.b: This is equivalent to the previous example. Schema processor will generate an error because the Age element is empty.

Both examples given above will produce the following error.

```
JAC\SQL2005(JAC\JACOB): Msg 6926, Level 16, State 1, Line 2
XML Validation: Invalid simple type value: ''. Location:
/*:Employee[1]/*:Age[1]
```

We should make the *"Age"* element *"nillable"* to get the requested validation done. After making the element *nillable*, we could add *xsi:nil="true"* to the XML instance to instruct the Schema processor to skip validating the given element.

Here is the updated version of the Schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Age" nillable="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="22"/>
              <xsd:maxInclusive value="55"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
```

```
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 5.42: A corrected version of the schema using the 'nillable' attribute.

Let us update the Schema Collection.

```
DROP XML SCHEMA COLLECTION NillableElement
GO

CREATE XML SCHEMA COLLECTION NillableElement AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Age" nillable="true">
          <xsd:simpleType>
            <xsd:restriction base="xsd:integer">
              <xsd:minInclusive value="22"/>
              <xsd:maxInclusive value="55"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
'
```

Listing 5.43: Employee information with nillable element.

And here is the XML instance which has an *Age* element with NULL value.

```
DECLARE @x XML(NillableElement)
SET @x =
'<Employee xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <Name>Jacob</Name>
    <Age xsi:nil="true"/>
</Employee>'
```

Listing 5.44: By setting "nillable" to "true" in the schema and then adding "xsi:nil" in the XML instance, you can generate a "validate-if-exists" effect. The Schema will allow the value to be empty and will perform the validation only if it is not empty.

"nillable" and *default*

When an element is declared as *"nillable"* in the schema and if the XML instance contains *"xsi:nil,"* then the default value is not assigned to the element.

As discussed earlier in this section, different applications treat missing values in different manners. Sometimes we need to specify explicitly that a piece of information is missing. However, if the element has a default value assigned, the value of the empty element will be replaced with the default value.

Let us look at an example:

```
<Employees>
  <Employee>
    <Name>Jacob</Name>
    <Gender>Male</Gender>
  </Employee>
</Employees>
```

Listing 5.45: Employee information XML document.

The XML fragment contains a list of employees and each employee element contains the *Name* and *Gender* of the employee. Assume that we have a schema that sets the default value of *Gender* to *Male*. Here is the schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Gender" type="xsd:string"
                     default="Male" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.46: Schema that describes the employee information XML document.

Note that the SCHEMA declares *Male* as the default value of element *Gender*.

```
DROP XML SCHEMA COLLECTION NillableElement
GO
CREATE XML SCHEMA COLLECTION NillableElement AS '
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Gender" type="xsd:string"
                     default="Male" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

***Listing 5.47: TSQL code that creates the Schema Collection that describes
the employee information XML document.***

If the *Gender* element is empty in the XML instance, the value *"Male"* is
assigned to it by the schema processor. Let us see an example.

```
DECLARE @x XML(NillableElement)
SET @x = '
<Employee>
    <Name>Jacob</Name>
    <Gender></Gender>
</Employee>'

SELECT @x

/*
<Employee>
  <Name>Jacob</Name>
  <Gender>Male</Gender>
</Employee>
*/
```

***Listing 5.48: "Male" is declared as the default value of "Gender," and hence
the schema processor adds it to the empty Gender element.***

```
DECLARE @x XML(NillableElement)
SET @x = '
<Employee>
    <Name>Jacob</Name>
    <Gender />
</Employee>'

SELECT @x

/*
<Employee>
  <Name>Jacob</Name>
  <Gender>Male</Gender>
</Employee>
*/
```

***Listing 5.49: "Male" is declared as the default value of "Gender," and hence
the schema processor adds it to the empty Gender element.***

There are times we need the Schema processor to ignore the default value assignment on a given element. It can happen that the information about the gender of a given employee is not available, and we do not want the schema processor to set it to the default value.

This can be done by making the element *"nillable"* in the schema and adding an *"xsi:nil"* attribute to the given element. Here is an example:

```
DROP XML SCHEMA COLLECTION NillableElement
GO
CREATE XML SCHEMA COLLECTION NillableElement AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Gender" type="xsd:string"
                     default="Male" nillable="true"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 5.50: A variation of the schema that sets the "Gender" element "nillable."

This example works in the same way it did previously.

```
DECLARE @x XML(NillableElement)
SET @x = '
<Employee>
    <Name>Jacob</Name>
    <Gender />
</Employee>'

SELECT @x

/*
<Employee>
   <Name>Jacob</Name>
   <Gender>Male</Gender>
</Employee>
*/
```

Listing 5.51: "Male" is declared as the default value of "Gender," and the schema processor adds it to the empty Gender element.

However, when *"xsi:nil"* attribute is added, the default value is not assigned to the *Gender* element.

```
DECLARE @x XML(NillableElement)
SET @x = '
<Employee >
```

```
    <Name>Jacob</Name>
    <Gender xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:nil="true" />
</Employee>'

SELECT @x

/*
<Employee>
  <Name>Jacob</Name>
  <Gender xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:nil="true" />
</Employee>
*/
```

Listing 5.52: When "xsi:nil" is set to "true" the schema processor does not assign the default value to an empty element.

Attribute: *SubstitutionGroup*

When you have a TYPED XML column or variable bound to a Schema Collection, SQL Server will perform very strict validations on the XML value. SQL Server will make sure that the structure of the XML, as well as the values of elements and attributes, strictly follows the rules defined in the Schema Collection.

This strict validation is good most of the time. However, there are times when we will need a little flexibility. Let us take the example of a billing application. The billing application needs to work with payment information. The payment information will vary based on the payment type: Credit Card, Cash or Check. If the payment is done through check, then we will have to store check number, bank name, etc. If it is done by Credit Card, we might need to store the card number, card type, expiration date, etc. If the payment is done by cash, we might need to store the currency as well as the denomination of the bills.

These types of requirements point to a certain type of element declaration that allows different sets of XML structure and values based on a certain flag value. Such elements are generally called *"variable content containers."* As the name indicates, they are containers capable of storing different types of content. XSD provides a few different ways to declare such elements. One way to declare variable-content-containers is by using *substitutionGroups*. Using *"substitutionGroup,"* we could create an extensible chain of XSD element declarations, just as you create an inheritance chain in your favorite OOP language.

Another way of creating *variable content containers* is by using *choice groups*. *Choice Groups* are explained in Chapter 10.

The Employee Information Document

To understand the usage of *substitutionGroup*, let us take the example of an employee information document. Assume that we need to write the schema of an XML document that stores employee information. Information of each employee will be represented by an element under the employee root node.

Simple so far; now let us move to the difficult part. The information being stored will not be the same for every employee. A manager will have a different set of attributes than a developer. The attributes of a developer will be different from that of a salesperson.

At first glance, you might think this could be easily done by declaring optional attributes so that each employee row can fill the attributes relevant to it. This will not work, however. We need to perform a strict validation based on the employee type. For example, *"Area"* should be mandatory for a sales person but should not be allowed in the element representing a developer.

This is an XML instance document for which we need to write the schema.

```
<Employees>
   <Manager Name="Jacob" Dept="IT" ReportsTo="CEO" />
   <Developer Name="Richard" Project="BI Tools" />
   <SalesPerson Name="David" Area="NY" ReportsTo="Jacob" />
   <Developer Name="John" Project="BI Tools" />
</Employees>
```

Listing 5.53: Employee information XML document.

Note that the element *employee* has three different elements as its children: *Manager, SalesPerson* and *Developer.* Each child element has its own set of attributes and may appear any number of times.

Before we think about the XSD way of doing this, let us think a second in an OOPS (Object Oriented Programming System) way. If we were to do it in an Object Oriented Programming Language, we might have landed with something like the following.
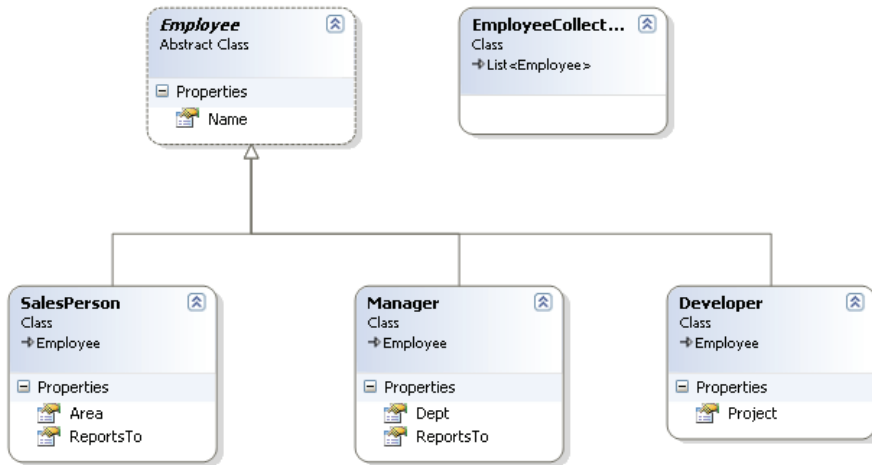
Listing 5.54: Diagram showing employee inheritance chain.

The class diagram given above is pretty simple and self explanatory. Many of us must have done this several dozen times in our programming career.

The above diagram shows a base class named *Employee* and three classes that derive from it. The base class has a property named "Name" and all the child classes inherit this property from the base class. The diagram also shows a collection class that can store *Employee* objects.

Since *SalesPerson*, *Manager* and *Developer* classes are derived from *Employee* and *EmployeeCollection* is a list of *Employee* objects, we could add an object of *Manager, Developer* and *SalesPerson* into *EmployeeCollection.*

This is exactly what we need to do in XSD. We need to make an element declaration that can store *employee, manager* or *salesperson* elements. To achieve this, let us follow the same steps as given in the OOPS class diagram given above.

Step 1: Define the Base Type

Let us first define the base type named *EmployeeBase*. The class diagram shows that the base class should have a *name* attribute which the other three classes inherit. Let us define the base class.

```
<xsd:complexType name="EmployeeBase">
   <xsd:attribute name="Name" type="xsd:string"/>
</xsd:complexType>
```

Listing 5.55: Definition of the Base types.

> We have defined a *Complex Type* named *EmployeeBase*. We have not yet discussed Complex Types. Refer to Chapter 10 for a detailed discussion of Complex Types.

Step 2: Declare Employee element

The next step is to define the *Employee* element. The *Employee* element will be of type *EmployeeBase,* which will serve as the base element for the collection that we will shortly declare.

```
<xsd:element name="Employee" type="EmployeeBase" />
```

Listing 5.56: Employee element derives from EmployeeBase and will be used for substitution.

Step 3: Declare the Manager Element

Let us now declare the *Manager* element. The *Manager* type will derive from base type *EmployeeBase* by extension. It inherits the *name* attribute from the parent and adds two new attributes named *Dept* and *ReportsTo*.

```
<xsd:element name="Manager" substitutionGroup="Employee">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="EmployeeBase">
          <xsd:attribute name="Dept" type="xsd:string" />
          <xsd:attribute name="ReportsTo" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
```

Listing 5.57: Declaration of the Manager element.

> **A new type can be derived from a complex type by *extension* or by *restriction.* This is explained in Chapter 11 – *Complex Type Derivation*.**

Step 4: Declare the Developer Element

Now let us look at the declaration of the *Developer* element. We will do it in the same manner we did for the *Manager* element. We will derive the *Developer* element from *EmployeeBase* by extension. It inherits the *name* attribute from the parent and adds a new attribute named *Project*.

```xsd
<xsd:element name="Developer" substitutionGroup="Employee">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="EmployeeBase">
        <xsd:attribute name="Project" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Listing 5.58: Declaration of the Developer element.

Step 5: Declare the SalesPerson Element

Next, let us declare the *SalesPerson* element. Just as with the other elements, it inherits the *name* attribute from the parent type and adds two instance specific attributes named *Area* and *ReportsTo.*

```xsd
<xsd:element name="SalesPerson" substitutionGroup="Employee">
  <xsd:complexType>
    <xsd:complexContent>
      <xsd:extension base="EmployeeBase">
        <xsd:attribute name="Area" type="xsd:string" />
        <xsd:attribute name="ReportsTo" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Listing 5.59: Declaration of the SalesPerson element.

Step 6: Declare the root element

We have created all the building blocks needed for the employee information schema. Now it is time to assemble them and build the final declaration.

```xml
<xsd:element name="Employees">
   <xsd:complexType>
     <xsd:sequence maxOccurs="unbounded">
       <xsd:element ref="Employee"/>
     </xsd:sequence>
   </xsd:complexType>
</xsd:element>
```

Listing 5.60: Declaration of the root element "Employees."

Step 7: Congratulations! The schema is done.

We are done with the schema. Let us put everything together so that I can show you the complete schema definition. Here is the final schema. I have added comments to explain the flow.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- root element declaration -->
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element ref="Employee"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <!-- The root element contains a sequence of "Employee"
elements.
      Here is the definition of "Employee" element
    -->
  <xsd:element name="Employee" type="EmployeeBase" />

  <!-- The "Employee" element is of Type "EmployeeBase"
      Here is the definition of EmployeeBase
    -->
  <xsd:complexType name="EmployeeBase">
    <xsd:attribute name="Name" type="xsd:string"/>
  </xsd:complexType>

  <!-- Manager Element : derived from EmployeeBase -->
  <xsd:element name="Manager" substitutionGroup="Employee">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="EmployeeBase">
          <xsd:attribute name="Dept" type="xsd:string" />
          <xsd:attribute name="ReportsTo" type="xsd:string"/>
        </xsd:extension>
```

```
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>

    <!-- Developer Element : derived from EmployeeBase -->
    <xsd:element name="Developer" substitutionGroup="Employee">
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:extension base="EmployeeBase">
            <xsd:attribute name="Project" type="xsd:string"/>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>

    <!-- SalesPerson Element : derived from EmployeeBase -->
    <xsd:element name="SalesPerson" substitutionGroup="Employee">
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:extension base="EmployeeBase">
            <xsd:attribute name="Area" type="xsd:string" />
            <xsd:attribute name="ReportsTo" type="xsd:string"/>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>

</xsd:schema>
```

Listing 5.61: Complete listing of the schema showing substitutionGroup.

Let us create a Schema Collection and see this in action:

```
CREATE XML SCHEMA COLLECTION
EmployeeCollectionSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element ref="Employee"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Employee" type="EmployeeBase" />
  <xsd:complexType name="EmployeeBase">
    <xsd:attribute name="Name" type="xsd:string"/>
  <xsd:element name="Manager" substitutionGroup="Employee">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="EmployeeBase">
          <xsd:attribute name="Dept" type="xsd:string" />
          <xsd:attribute name="ReportsTo" type="xsd:string"/>
        </xsd:extension>
      </xsd:complexContent>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Developer" substitutionGroup="Employee">
    <xsd:complexType>
      <xsd:complexContent>
        <xsd:extension base="EmployeeBase">
```

```
            <xsd:attribute name="Project" type="xsd:string"/>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>
    <xsd:element name="SalesPerson" substitutionGroup="Employee">
      <xsd:complexType>
        <xsd:complexContent>
          <xsd:extension base="EmployeeBase">
            <xsd:attribute name="Area" type="xsd:string" />
            <xsd:attribute name="ReportsTo" type="xsd:string"/>
          </xsd:extension>
        </xsd:complexContent>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>'
```

Listing 5.62: TSQL code to create a schema collection with substitutionGroup.

Now let us validate our XML instance document against the schema collection that we just created.

```
DECLARE @x XML(EmployeeCollectionSchema)
SET @x = '
<Employees>
  <Manager Name="Jacob" Dept="IT" ReportsTo="CEO" />
  <Developer Name="Richard" Project="BI Tools" />
  <SalesPerson Name="David" Area="NY" ReportsTo="Jacob" />
  <Developer Name="John" Project="BI Tools" />
</Employees>'
```

Listing 5.63: Validating the Employee information XML document against the schema having substitutionGroup.

Run the above code and you will see that we have created a correct schema that fulfills all the requirements we discussed. Note that each element, *Manager, Developer* and *SalesPerson,* is validated based on rules specific to each one.

This design is highly extensible. In the future we could easily add new types derived from *EmployeeBase* and you will not need to modify the schema other than adding the new type definition. However, I do feel that the usage of *substitutionGroup* is a little more complex than it might have been.

> **Earlier in this chapter we discussed local and global element declarations. Only global element declarations can take a *substitutionGroup* element.**

Attribute: *abstract*

We have seen the power of *Substitution Groups* in the previous section. Since *Substitution Groups* are very powerful and provide a great amount of flexibility, there needs to be a way to control their usage. The usage of *Substitution Groups* can be controlled by using *abstract, block* and *final* attributes.

When an element is declared as *abstract*, it cannot be instantiated. This is very close to the OOPS concept of abstract classes. An abstract class usually serves as a base class and cannot be instantiated. It is the same with XSD. When an element is declared as abstract, you cannot use it in an element declaration.

Let us see an example.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>

  <!-- Instaitiate the "Employee element" -->
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Employee"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<Employees>
```

```
  <Employee FirstName="Jacob" LastName="Sebastian"/>
</Employees>'

DROP XML SCHEMA COLLECTION ExampleSchema
GO
```

Listing 5.64: Schema showing a global element declaration.

The above example declares a global element named *"Employee."* Another global element *"Employees"* holds a reference to the *Employee* element.

Now let us try to make the *Employee* element *abstract* and see what happens.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee" abstract="true">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>

  <!-- Instaitiate the "Employee element" -->
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="Employee"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<Employees>
  <Employee FirstName="Jacob" LastName="Sebastian"/>
</Employees>'

DROP XML SCHEMA COLLECTION ExampleSchema
GO
```

Listing 5.65: Abstract elements cannot be used in the XML instance.

The above code will produce the following error because the element Employee is declared as abstract and cannot be used directly.

```
XML Validation: Element 'Employee' requires substitution, because
it was defined as abstract. Location:
/*:Employees[1]/*:Employee[1]
```

> **Only global element declarations can use the *abstract* attribute.**

Attribute: *block*

In the previous section we examined the *abstract* attribute which forces an element substitution. *Abstract* elements cannot be used unless they are substituted, and thus force an element substitution. The usage of *block* is the opposite. It is used to control substitution. The *block* attribute can take 4 different vales, namely: *substitution, extension, restriction* and *#all*. Let us examine each one of them.

Substitution

By setting the *block* attribute to *substitution* you can prevent an element from being substituted. For example:

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee" block="substitution">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.66: Blocking substitution of an element.

Since the *Employee* element is declared with *block* attribute set to *substitution*, it cannot be substituted.

Restriction

I had mentioned earlier that a new type can be derived from a base type by *extension* or by *restriction*. By setting *block* to *restriction* you could prevent the element being inherited by restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee" block="restriction">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.67: Blocking derivation by restriction.

Extension

By setting the value of *block* to *extension* you can prevent the element/type from being inherited by extension.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee" block="extension">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.68: Blocking derivation by extension.

#all

By setting the value of *block* attribute to *#all* you could restrict inheritance as well as type substitution. When *#all* is specified it prevents type substitution, derivation by extension, as well as derivation by restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Employee element -->
  <xsd:element name="Employee" block="#all">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.69: Blocking substitution as well as derivation.

blockDefault

If the *block* attribute is not specified, the schema processor will check the value of the *blockDefault* attribute in the declaration of the *schema* element. If this attribute is present, its value is used for all elements that do not have a *block* attribute. If the *blockDefault* attribute is missing, no restriction is applied on substitution.

The following example shows a schema declaration with *blockDefault* value set to *substitution.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            blockDefault="substitution">
  <xsd:element name="Employee" final="">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.70: A schema declaration with "blockDefault" attribute.

Only global element declarations can use the *block* attribute.

Attribute: *final*

The *final* attribute limits the declaration of substitution groups in schemas. The behavior of *"final"* attribute is close to that of *"block,"* except that *final* is used to apply restrictions at the *schema* level and *block* is used to apply restrictions at the *instance* level.

The validation of *final* attribute takes place when the schema collection is generated, whereas the validation of *block* takes place when you actually validate an XML instance with the schema collection.

The *final* attribute can take the following values: *#all, restriction* and *extension.*

extension

When *final* attribute of an element is set to *restriction* you cannot declare a substitution group with the restricted element as the head. Let us see an example in order to understand this. Let us modify the schema we created

for demonstrating substitution groups and add a *final* attribute with value *extension.*

```
<!-- The root element contains a sequence of "Employee" elements.
     Here is the definition of "Employee" element
  -->
<xsd:element name="Employee" type="EmployeeBase" final="extension"/>

<!-- The "Employee" element is of Type "EmployeeBase"
     Here is the definition of EmployeeBase
  -->
<xsd:complexType name="EmployeeBase">
  <xsd:attribute name="Name" type="xsd:string"/>
</xsd:complexType>

<!-- Manager Element : derived from EmployeeBase -->
<xsd:element name="Manager" substitutionGroup="Employee">
'Manager' cannot be a member of substitution group with head element 'Employee'.
      <xsd:complexContent>
      <xsd:extension base="EmployeeBase">
        <xsd:attribute name="Dept" type="xsd:string" />
        <xsd:attribute name="ReportsTo" type="xsd:string"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>
</xsd:element>
```

Listing 5.71: The SSMS schema editor will show an error when you try to use the "Employee" as a substitution group head.

If you still go ahead and try to create a schema collection, SQL Server will generate an error.

```
Invalid element definition, element 'Manager' is not valid
derivation of element 'Employee'
```

restriction

When the *final* attribute is set to "*restriction,"* the element cannot be used as the head of a substitution group that derives by restriction.

The SSMS XSD editor will show a warning message when you try to use a restricted element as the head of a substitution group. If you try to create a Schema Collection with such a declaration, SQL Server will generate an error.

#all

By setting *final* attribute to *"#all"* you can restrict the element from being used as the head of a substitution group completely. The following example shows the declaration of an element with *final* attribute set to *#all*. This element cannot be used for substitution, by restriction, or by extension.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee" final="#all">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

***Listing 5.72: A declaration showing the final attribute with "#all."***

The following schema is equivalent to the one given above. By specifying *restriction* and *extension* together, we could generate the same restriction effect as *#all.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee" final="restriction extension">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.73: A combination of "restriction" and "extension" will produce the same effect as "#all."

finalDefault

If the *final* attribute is not specified, the schema processor will look for a *finalDefault* attribute in the *schema* element declaration. If the *finalDefault* attribute is not present in the declaration of *schema* element, no restriction will be applied on substitution.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            finalDefault="#all">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.74: A schema with "finalDefault" set to "#all."

The *finalDefault* attribute of the above schema declaration is set to *"#all,"* and, hence, substitution of the *Employee* element is not permitted.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 5.75: A schema declaration without "final" and "finalDefault" attributes.

The above schema does not apply any restriction on substitution because neither *final* nor *finalDefault* attributes are declared.

Only global element declarations can use the *final* attribute.

Attributes: *minOccurs* and *maxOccurs*

These two attributes control the occurrence of the elements. *minOccurs* specifies the minimum number of times the element should appear in the XML instance. *maxOccurs* specifies the maximum number of times the element can appear within its parent.

The default value of *minOccurs* and *maxOccurs* is 1. You can make an element optional by setting *minOccurs* to 0.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="Employee" >
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
        <xsd:element name="MiddleName" minOccurs="0"/>
        <xsd:element name="LastName"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
```

```
SET @x = '
<Employee>
      <FirstName>Jacob</FirstName>
      <LastName>Sebastian</LastName>
</Employee>'
```

Listing 5.76: Example showing the declaration of an optional element.

The above example shows the declaration of an optional element: *MiddleName*.

> *minOccurs* and *maxOccurs* are applicable to local element declarations only. We will have a detailed look into it when we examine complex types in Chapter 10.

Attribute: *ref*

Local element declarations can refer to other elements declared globally in the same schema. This is achieved by using the *ref* attribute in an element declaration.

Referring to an element in this manner adds a certain level of reusability. You can declare the element once, and then refer it multiple times at different locations of the schema. This could avoid some duplication of code.

Let us look at an example in order to understand this.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <!-- Employee element -->
  <xsd:element name="EmployeeInfo" >
    <xsd:complexType>
      <xsd:attribute name="FirstName"/>
      <xsd:attribute name="LastName"/>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="Manager">
    <xsd:complexType>
      <xsd:sequence>
```

```
        <xsd:element ref="EmployeeInfo"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<Manager>
      <EmployeeInfo FirstName="Jacob" LastName="Sebastian"/>
</Manager>'
```

Listing 5.77: Example showing the usage of "ref" attribute.

Attribute: *form*

The *form* attribute specifies whether the element needs to be qualified with a namespace prefix. The value of *form* may be *"qualified"* or *"unqualified."*

When the value of *form* attribute is set to *"qualified,"* the element needs to be qualified with a namespace prefix at all times. If the value is set to *"unqualified,"* the element should not be qualified by a namespace prefix.

Let us look at an example:

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.jacob.com">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName" form="qualified"/>
        <xsd:element name="LastName" form="unqualified"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
```

```
<jac:Employee xmlns:jac="http://www.jacob.com">
     <jac:FirstName>Jacob</jac:FirstName>
     <LastName>Sebastian</LastName>
</jac:Employee>'
```

Listing 5.78: An example showing the usage of the "form" attribute.

Note that the *form* attribute of *FirstName* is declared as *qualified* and, therefore, the XML instance should take a namespace prefix. The next attribute, *LastName,* is declared with *form* attribute set to *unqualified* and therefore cannot take a namespace prefix.

If the *form* attribute is not present, the schema processor will look for the *elementFormDefault* attribute in the declaration of the *schema* element. The following example shows the declaration of a schema with *elementFormDefault* attribute.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http://www.jacob.com"
           elementFormDefault="qualified">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="FirstName"/>
        <xsd:element name="LastName" form="unqualified"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<jac:Employee xmlns:jac="http://www.jacob.com">
     <jac:FirstName>Jacob</jac:FirstName>
     <LastName>Sebastian</LastName>
</jac:Employee>'
```

Listing 5.79: An example showing "elementFormDefault" attribute.

Note that we have not specified the *form* attribute for *FirstName* element. However, the *form* of *FirstName* is considered as *"qualified"* because the *elementFormDefault* attribute is set with *"qualified."*

Chapter Summary

This chapter provided a detailed overview of element declarations. When an element is declared as a direct child of the *schema* element, it is a global element declaration. When an element is declared within other schema components (Complex Types, Element Groups, etc.), it is a local element declaration.

An element declaration takes several parameters, out of which the only required parameter is the *name* of the element. There are several other parameters (attributes) that are optional.

The *type* attribute is used to associate an element declaration with a given data type. When an element declaration is associated with a type, SQL Server will perform a data type validation on the value stored by the element. If the value is not valid for the given data type, SQL Server will generate an error.

The *default* attribute is used to assign a predefined value to an element declaration. The schema processor will assign this value to the element if the element is empty. The *fixed* attribute works in a similar manner. If the element contains a value, it should be the same value declared with the *fixed* attribute. If the element does not have a value, the schema processor will assign the value declared with *fixed* attribute to the element.

The *nillable* attribute is helpful when we need to distinguish between an element that is blank and an element that is empty. It is also helpful in cases where we need to perform a certain validation only if a value is present in the XML instance. When an element is declared as *"nillable"* and the element contains the *attribute"xsi:nil"* in the *XML* instance, the XSD processor will not assign a default value to the element even if it is empty.

The *substitutionGroup* attribute is used to declare elements capable of serving as *variable content containers*. *Abstract* is used to specify that the element cannot be instantiated directly. To instantiate such an element, a substitution element needs to be created. The *block* and *final* attributes are used to control substitution of the element. *Final* restricts substitution at schema level and *block* restricts it at instance level.

minOccurs and *maxOccurs* are used to control the occurrence of the elements. The default value of both attributes is 1. By setting the value of *minOccurs* to 0, you make the element optional.

ref is used to insert a globally declared element at a specific location within a schema (usually under a Complex Type or Element Group). The *form*

attribute specifies where the element should be fully qualified or not in the instance document.

CHAPTER 6

UNDERSTANDING ATTRIBUTE DECLARATIONS

Elements and attributes are the basic building blocks of XML. XSD, being an XML document, is composed of elements and attributes at the most granular level. We have discussed element declarations in Chapter 5. This chapter will focus on *attribute declarations* and will examine the following.

- Elements vs. Attributes
- Basic attribute declarations
- Global and local attribute declarations
- Attributes of an Attribute declaration
- Attribute groups

Elements vs. Attributes

When designing the structure of an XML document, one question returns time and time again. Should this piece of data go in an element or in an attribute? Sometimes you don't have a choice; there are times when only an element will do.

On many occasions, your piece of data could be stored equally well in an attribute as in an element with no loss of information. This is where you have to make the decision – element or attribute? The good news is that at the end of day, it doesn't really matter which you choose. They both do the same job. Even the experts are divided on this question – to such a degree that in fact it ends up being largely a matter of personal taste. For this reason we'll not be entering into this debate in this book, as ultimately – while the topic may be interesting – it won't help you write schemas that work any better.

In this section we're going to focus on those occasions where you don't have a choice. While there are no occasions when you *have* to use an attribute over an element, there *are* occasions when only an element will do.

The following restrictions on attributes compared to elements may compel you to use an element.

- Attributes are parasites! They can only exist within an element; thus, any top level declaration must be an element.
- Attributes can only store a value, while elements can also store child elements and attributes.
- An element can appear more than once within a parent node, but an attribute can appear only once. Thus, an attribute always has a one to (zero or) one relationship with the element it describes.
- The order of attributes is not significant and there is no way to control the order of attributes in XSD. If order is important to you then you have to use elements.

If an attribute is still an option, then you should be aware of some differences in behavior between elements and attributes.

Both element declarations and attribute declarations can take a default value. However, the processing of the default value is slightly different. The default value is assigned to the attribute only if the attribute is missing (not present). If the attribute is present the default value is not assigned, even if the value of the attribute is an empty string.

On the other hand, the default value of elements is assigned only when the element is present and is empty. If the element is missing, the default value is not assigned.

Also note that elements are mandatory by default while attributes are optional by default.

As I mentioned previously, many times you can store a piece of information in the form of an attribute or as an element. Let us look at an example:

```
<Employees>
   <Emplyee FirstName="Jacob" LastName="Sebastian"/>
</Employees>
```

Listing 6.1: Storing employee information in attributes.

```
<Employees>
   <Emplyee>
      <FirstName>Jacob</FirstName>
      <LastName>Sebastian</LastName>
   </Emplyee>
</Employees>
```

Listing 6.2: Storing employee information in elements.

The first example stores *FirstName* and *LastName* as attributes and the second example stores them as elements.

A Basic Attribute Declaration

Let us start with a basic attribute declaration. The simplest form of an attribute declaration is given below:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:attribute name="status" />
</xsd:schema>
```

Listing 6.3: A basic attribute declaration.

Though this is a valid schema, an attribute declaration really does not make sense without an element. Hence, a basic attribute declaration that makes some sense should be something like the following:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
     <xsd:complexType>
       <xsd:attribute name="status" />
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 6.4: A complexType with an attribute declaration.

> **Only a *Complex Type* can have attributes; hence, the *Employee* element is declared as *Complex Type*. *Complex Type* is explained in Chapter 10.**

An instance document that validates with the above schema is given below.

```
<Employee status="Active"/>
```

Listing 6.5: An XML instance showing an attribute.

An attribute is declared with **<xsd:attribute>** element. This element can take a number of different attributes that control the behavior of the attribute being declared. I will call them *"attributes of an attribute declaration"* throughout this book.

The only mandatory attribute of an attribute declaration is *"name."* There are several other attributes that an attribute declaration can take. We will examine those attributes a little later in this chapter.

Global and Local attribute declarations

When an attribute is declared right under the *"<xsd:schema>"* element, it is called *global attribute declaration*. When it is declared within a *Complex Type,* it is called *Local attribute declaration.* Let us see an example of a global attribute declaration.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="status" />
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute ref="status"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.4: A global attribute declaration.

When an attribute is declared globally, it can be referenced within other *Complex Types* declared in the same schema. The *"ref"* attribute is used to insert a globally defined attribute to a given location.

Here is a different version of the above schema, using a *local* attribute declaration.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="status" />
    </xsd:complexType>
```

```
    </xsd:element>
</xsd:schema>
```

Listing 6.5: A local attribute declaration.

We just saw two different ways of declaring an attribute. The first example used a global attribute declaration and the second example used a local attribute declaration. They both describe an XML instance with the same structure. Here is an XML instance that validates with both the schemas.

```
<Employee status="Active"/>
```

Listing 6.6

Let us create a schema collection and see this in action.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="status" />
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute ref="status"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Employee status="Active"/>'
```

Listing 6.7: A schema that uses a global attribute declaration.

```
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create new schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
```

```
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="status" />
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate an XML instance
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Employee status="Active"/>'
```

Listing 6.8: A schema showing a local attribute declaration.

Global attribute declarations are useful when an attribute is declared with several validations. By declaring the attribute global, we could avoid rewriting the rules every time we need to declare the same attribute.

For example, the following two schemas describe the same XML instance. The second one is more manageable.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="Manager">
          <xsd:complexType>
            <xsd:attribute name="Phone">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:minLength value="10"/>
                  <xsd:maxLength value="12"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Supervisor">
          <xsd:complexType>
            <xsd:attribute name="Phone">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:minLength value="10"/>
                  <xsd:maxLength value="12"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>              </xsd:element>
      </xsd:all>
```

```
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.9: An example that uses a local attribute declaration and resulting in duplication of code.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:minLength value="10"/>
        <xsd:maxLength value="12"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="Manager">
          <xsd:complexType>
            <xsd:attribute ref="Phone"/>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Supervisor">
          <xsd:complexType>
            <xsd:attribute ref="Phone"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.10: A schema that uses a global attribute declaration and reuses the code.

Both examples above describe the following XML instance.

```
<Employees>
  <Manager Phone="9999999999"/>
  <Supervisor Phone="9999999999"/>
</Employees>
```

Listing 6.11: Employee information XML document.

The first example shows two local attribute declarations that define an attribute named *"Phone."* Each attribute declaration is local and is not visible to the other. Hence, we had to write the validation rules twice.

The second example shows a global attribute declaration. The *Manager* and *Supervisor* elements refer to the attribute declared globally. The

validation rules are defined only once (in the global declaration). Thus, global attribute declarations provide some sort of re-usability.

Attributes of Attribute Declaration

We saw attribute declarations earlier in this chapter. An attribute is declared using <xsd:attribute /> element. This declaration can take a number of attributes that control the way the attribute is validated by the schema processor. Let us examine each of these attributes in detail.

Just as with element declarations, the only mandatory attribute that an attribute declaration should take is the *"name"* attribute.

A *global* attribute declaration can take the following attributes:

- **name**
- **id**
- **type**
- **default**
- **fixed**

A *local* attribute declaration can take all the attributes of *global* attribute declarations mentioned above (*name, id, type, default* and *fixed*) plus the following three attributes:

- **ref**
- **use**
- **form**

Let us look at each of these attributes in detail.

Attribute: *name*

This attribute refers to the name of the attribute as it should appear in the XML instance. This is a mandatory attribute. No attributes can be declared without a name. The first character of the name should be a letter or underscore. Other characters (except for the first character) can contain any combination of letters, digits, underscores, hyphens and periods.

The name of an attribute should be unique in its scope. It means that there can be only one global attribute with a given name. However, two different complexTypes can have attributes with the same name.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="name" />
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="name"/>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="Department">
    <xsd:complexType>
      <xsd:attribute name="name"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.12: Example of global and local attributes having the same name.

The above schema has three attribute declarations with the same name. However, each of them appears in a different scope and each is valid. The first is a global declaration and does not conflict with any other attributes declared globally. The other two declarations are local and they are unique within each complexType.

It may be a good idea to avoid using hyphens and periods within the name of an attribute. Though the usage of those characters is permitted, it might confuse certain XML parsers.

Attribute: *id*

"id" is an optional attribute that an attribute declaration can take. The presence or absence of this attribute does not affect the XML instance in any manner. *"id"* is used by the schema processor to uniquely identify the XSD components within a given schema.

The same naming rules as the *"name"* attribute are applicable to *"id,"* too. To some people the name *"id"* may give a feeling that it can take a numeric value. However, that is not true. The following is invalid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="EmployeeName" id="101"/>
    </xsd:complexType>
```

```
    </xsd:element>
  </xsd:schema>
```

Listing 6.13: The value of "id" cannot start with a digit.

The value of *"id"* cannot start with a digit. It should start with a letter or an underscore. Just like the *"name"* attribute it can contain digits, periods and hyphens (except for the first character). The following is valid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="EmployeeName" id="_101"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.14: The value of "id" can start with an underscore or letter and can contain digits, hyphens, periods and letters.

The value of the *"id"* attribute should be unique within the schema. Every *"id"* declared in a schema has global scope and the following is invalid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="EmployeeName" id="EmpName"/>
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="EmployeeName" id="EmpName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.15: Each "id" declared in a schema has global scope; hence, there cannot be two "id" with same value.

Two elements within a schema cannot have the same id. The following is invalid, too.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="EmployeeName"/>
  <xsd:element name="Employee" id="EmpName">
    <xsd:complexType>
      <xsd:attribute name="EmployeeName" id="EmpName"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.16: The above schema is invalid because the "EmpName" is duplicated.

The uniqueness of the *"id"* attribute is to be ensured not only within the attribute elements, but also within all other XSD components declared in the schema.

Attribute: *type*

"type" associates a data type with an attribute and thus facilitates data type specific validation on the value of the attribute. The *"type"* of an attribute can be declared as one of the built-in data types or a *simpleType* defined globally in the schema document.

Simple Types are explained in Chapter 8.

The *"type"* attribute behaves the same way it does with element declarations. A simple example is the *Age* element we saw when we discussed element declarations. In the absence of a *type* declaration, the *Age* attribute of an *Employee* might contain values which may not make sense at all. For example:

```
<Employee Age="twnty"/>
```

Listing 6.17

By associating the *Age* attribute with *integer* data type, we could make sure that only valid integers are accepted. Going a step further, we could even use *xsd:positiveInteger* to make sure that only values greater than zero are accepted.

```
<Employee Age="20"/>
```

Listing 6.18

Here is the schema that declares the above validation.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Age" type="xsd:positiveInteger"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.19: positiveInteger restricts negative values and zero.

> XSD native data types are discussed in Chapter 7. XSD
> Derived data types are discussed in Chapter 9.

Attribute: *default*

The *default* attribute assigns a default value to an attribute declaration. The attribute is missing in the instance document; the *default* value will be assigned to the attribute.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Name" type="xsd:string"/>
      <xsd:attribute name="Gender" type="xsd:string"
       default="Male"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 6.20: A schema using "default" attribute.

```
DECLARE @e XML(ExampleSchema)

-- if the attribute exists and is empty, the default
-- value is not assigned
SET @e = '<Employee Name="Jacob" Gender=""/>'
SELECT @e
/*
<Employee Name="Jacob" Gender="" />
*/

-- The default value will be assigned only
-- If the attribute is missing
SET @e = '<Employee Name="Jacob"/>'
SELECT @e
/*
<Employee Name="Jacob" Gender="Male" />
*/
```

Listing 6.21: the default value of an attribute is assigned only if the attribute is missing.

The default value is assigned to the attribute only if the attribute is missing (not present). If the attribute is present the default value is not assigned, even if the value of the attribute is an empty string.

This behavior is totally different from that of elements. The default value of elements is assigned only when the element is present and is empty. If the element is missing, the default value is not assigned.

Attribute: *fixed*

The behavior of the *"fixed"* attribute in an attribute declaration is similar to the behavior of the *"fixed"* attribute in an element declaration. It prevents the attribute from taking any value other than the pre-defined one. If the attribute is not present, the value declared with *fixed* attribute will be used. If the attribute is present, it can store only the value declared with the *fixed* attribute.

Let us look at an example:

```sql
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Document">
    <xsd:complexType>
      <xsd:attribute name="lang" type="xsd:string" fixed="EN-US"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 6.22: A schema showing the "fixed" attribute.

```sql
DECLARE @x XML(ExampleSchema)

-- If the attribute is present, it should store
-- the same value declared with the "fixed" attribute
SET @x = '<Document lang="EN-US" />'
SELECT @x
/*
<Document lang="EN-US" />
*/

-- If the attribut is missing, the "fixed" value
-- will be assigned to it.
```

```
SET @x = '<Document />'
SELECT @x
/*
<Document lang="EN-US" />
*/
```

Listing 6.23: If the attribute is missing, it is created and assigned with the value declared with the "fixed" attribute.

The first example assigns the value *"EN-US"* to the *"lang"* attribute. The second example does not have the attribute and the schema processor adds the attribute with value *"EN-US."*

When an attribute declaration has a *"fixed"* attribute, the attribute cannot take any value other than the one declared in the schema. The following is an invalid XML instance because the value *"FR"* is not accepted (Only *"EN-US"* will be accepted).

```
DECLARE @x XML(ExampleSchema)
SET @x = '<Document lang="FR" />'
```

Listing 6.24: This XML instance is invalid because the "lang" attribute is declared with a "fixed" value, "EN-US."

Attribute: *ref*

Earlier in this chapter we have discussed *global attribute declaration* and *local attribute declaration.* A globally declared attribute can be inserted into a complex type by using the *"ref"* attribute. This adds a reference to the globally declared attribute

Let us look at an example.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attribute name="status" />
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute ref="status"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.25: A schema showing the "ref" attribute.

This is particularly useful if the attribute needs complex validations and needs to be used in more than one complex type. In such a case you could declare the attribute globally, with all the validations, and add a reference to

it in one or more complex types. This allows a certain level of reusability of code. For example:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone attribute -->
  <xsd:attribute name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[0-9]{10}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:attribute>

  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <!-- Manager -->
        <xsd:element name="Manager">
          <xsd:complexType>
            <xsd:attribute ref="Phone"/>
          </xsd:complexType>
        </xsd:element>
        <!-- Developer -->
        <xsd:element name="Developer">
          <xsd:complexType>
            <xsd:attribute ref="Phone"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.26: An example showing a global attribute declaration and reusing the global declaration in two complex types.

Attribute: *use*

This attribute specifies whether the attribute is optional or mandatory. You could manage this by setting the value of this attribute to *optional* or *required.* The default value is *"use"* is *optional;* hence, all attributes are optional by default. It can take a third value, *prohibited* to restrict the attribute from being included in an XML instance.

The following example sets an attribute to be mandatory.

```sql
-- Drop previous schema collection
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO
```

148

```
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="status" use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 6.28: An example showing a mandatory attribute.

The schema processor will generate an error if the XML instance does not contain the "*status"* attribute. The following is a correct XML instance that validates with the above schema.

```
DECLARE @x XML(ExampleSchema)
SET @x = '<Employee status="Active" />'
```

Listing 6.29: "status" is a mandatory attribute and the XML instance will be accepted only if this attribute is present.

The schema processor will raise an error if the *"status"* attribute is missing.

```
DECLARE @x XML(ExampleSchema)
SET @x = '<Employee />'
```

Listing 6.30: This XML instance is invalid because the "status" attribute is missing.

You can make the attribute optional by declaring *"use"* with *"optional."* Here is an example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="status" use="optional"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.31: An example showing an optional attribute.

The default value of *"use"* is *"optional."* Hence, the following schema is also equivalent to the one given above.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
```

```
      <xsd:complexType>
        <xsd:attribute name="status"/>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 6.31.a: The default value of "use" is optional and the "status" attribute in this schema is optional.

Note that the *"use"* attribute is not declared in this version of the schema. If the *"use"* attribute is not declared, *"optional"* is assumed.

We have seen the usage of *optional* and *required*. The *use* attribute can take one more value: *prohibited*. *"prohibited"* is used to restrict the use of an attribute in a derived type.

Sometimes it can happen that you want to derive a new type from a base type, but want to restrict one or more attributes. In such cases, you can declare the attribute as *prohibited* so that the XML instance that validates against the derived type cannot have that attribute. We will see this when we examine *Complex Type derivation* in Chapter 11.

Attribute: *form*

The *form* attribute specifies whether the attribute needs to be qualified by a namespace prefix or not, in the XML instance. This is very similar to the *form* attribute of element declarations. The *form* attribute can take two values: *qualified* and *unqualified*. When the value is set to *qualified*, the attribute should be qualified by a namespace prefix. When it is set to *unqualified*, the attribute should not be qualified by a namespace prefix.

Let us see an example in order to understand this.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.jacob.com">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee" >
          <xsd:complexType>
```

```
                  <xsd:attribute name="FirstName" form="qualified"/>
                  <xsd:attribute name="LastName" form="unqualified"/>
            </xsd:complexType>
         </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<jac:Employees xmlns:jac="http://www.jacob.com">
      <Employee jac:FirstName="Jacob" LastName="Sebastian"/>
</jac:Employees>'
```

Listing 6.32: An example showing "qualified" and "unqualified" forms.

Note that the attribute *"FirstName"* is declared as *qualified* and should always take a namespace prefix. *"LastName"* is declared as *unqualified* and cannot take a namespace prefix.

If the *form* attribute is not present, the *attributeFormDefault* attribute of the schema element declaration will be used as the default value.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="http://www.jacob.com"
            attributeFormDefault="qualified">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee">
          <xsd:complexType>
            <xsd:attribute name="FirstName"/>
            <xsd:attribute name="LastName" form="unqualified"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
```

```
<jac:Employees xmlns:jac="http://www.jacob.com">
      <Employee jac:FirstName="Jacob" LastName="Sebastian"/>
</jac:Employees>'
```

Listing 6.33: An example showing the usage of "attributeFormDefault" attribute.

In the above example the *attributeFormDefault* attribute of the schema element is set to *"qualified"* and, accordingly, all the attributes without *form* will take *"qualified"* as their form. If neither *form* nor *attributeFormDefault* is present, the value will default to *unqualified.*

Attribute Groups

Attribute Groups provide a convenient means to reuse attribute declarations in multiple complex types. At the beginning of this chapter we covered global attribute declarations and found that they provide a certain extent of reusability. Attribute Groups provide a better level of reusability by grouping one or more attribute declarations into a named group.

Let us look at an example that would explain this well. Here is an XML instance which stores some information about employees of a Technology Company.

```
<Employees>
   <Manager name="Jacob" department="IT"/>
   <TechLead name="Bob" department="SW"/>
</Employees>
```

Listing 6.34: Employee information XML document.

Let us write a simple schema to describe this XML instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <!-- Declaration of root element "Employees" -->
   <xsd:element name="Employees">
     <xsd:complexType>
       <xsd:sequence>
         <!-- Declaration of "Manager" element -->
         <xsd:element name="Manager">
           <xsd:complexType>
             <xsd:attribute name="name">
               <xsd:simpleType>
                 <xsd:restriction base="xsd:string">
                   <xsd:maxLength value="20"/>
                 </xsd:restriction>
               </xsd:simpleType>
             </xsd:attribute>
             <xsd:attribute name="department">
```

```
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:length value="2"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
            </xsd:complexType>
          </xsd:element>
          <!-- Declaration of "TechLead" element -->
          <xsd:element name="TechLead">
            <xsd:complexType>
              <xsd:attribute name="name">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="20"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
              <xsd:attribute name="department">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:length value="2"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

*Listing 6.35: A schema that describes the employee information
XML document.*

Note that the declaration of *name* and *department* attributes is repeated
under *Manager* and *TechLead* elements. By using an attribute group, you
can avoid this repetition of code. *An Attribute Group* can simplify this code
further, as given below.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:attributeGroup name="EmpAttributes">
    <!-- Declaration of attribute "name" -->
    <xsd:attribute name="name">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:maxLength value="20"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <!-- Declaration of attribute "department" -->
    <xsd:attribute name="department">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:length value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
```

```
    </xsd:attributeGroup>
    <!-- Declaration of root element "Employees" -->
    <xsd:element name="Employees">
      <xsd:complexType>
        <xsd:sequence>
          <!-- Declaration of "Manager" element -->
          <xsd:element name="Manager">
            <xsd:complexType>
              <xsd:attributeGroup ref="EmpAttributes"/>
            </xsd:complexType>
          </xsd:element>
          <!-- Declaration of "department" element -->
          <xsd:element name="TechLead">
            <xsd:complexType>
              <xsd:attributeGroup ref="EmpAttributes"/>
            </xsd:complexType>
          </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 6.36: A schema that shows the usage of an attribute group.

Attribute groups can contain references to other attribute groups, as given in the below example.

```
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'AttributeTest'
) BEGIN
    DROP XML SCHEMA COLLECTION AttributeTest
END

GO
CREATE XML SCHEMA COLLECTION AttributeTest AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Attributes of employee -->
  <xsd:attributeGroup name="EmpAttributes">
    <xsd:attribute name="name"/>
    <xsd:attribute name="department"/>
  </xsd:attributeGroup>
  <!-- Attributes of Tech Lead -->
  <xsd:attributeGroup name="TechLeadAttributes">
    <xsd:attributeGroup ref="EmpAttributes"/>
    <xsd:attribute name="email"/>
  </xsd:attributeGroup>
  <!-- Attributes of Manager -->
  <xsd:attributeGroup name="ManagerAttributes">
    <xsd:attributeGroup ref="TechLeadAttributes"/>
    <xsd:attribute name="phone"/>
  </xsd:attributeGroup>
  <!-- Declaration of root element "Employees" -->
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <!-- Declaration of "Manager" element -->
        <xsd:element name="Manager">
          <xsd:complexType>
```

```
                    <xsd:attributeGroup ref="ManagerAttributes"/>
            </xsd:complexType>
        </xsd:element>
        <!-- Declaration of "department" element -->
        <xsd:element name="TechLead">
            <xsd:complexType>
                <xsd:attributeGroup ref="TechLeadAttributes"/>
            </xsd:complexType>
        </xsd:element>
        </xsd:sequence>
      </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 6.37: A schema that shows attribute groups that refers to other attribute groups.

You could see a chain of attribute group hierarchies in this example. *TechLeadAttributes* inherits from *EmpAttributes* and adds an *"email"* attribute to it. *ManagerAttributes* inherits from *TechLeadAttributes* and adds a *"phone"* attribute to it. Here is an XML instance which validates with this schema.

```
DECLARE @x XML(AttributeTest)
SET @x =
'<Employees>
  <Manager name="Jacob" department="IT" email="a@b.com"
      phone="222-222-2222"/>
  <TechLead name="Bob" department="SW" email="a@b.com" />
</Employees>'
```

Listing 6.38

LAB1: Write schema for the Order Processing Application – The Root element

In Chapter 3 we discussed the order processing application of North Pole Corporation. We have seen the structure of the XML document that the application expects and saw what the validation rules required. In Chapter 5 we learned element declarations and saw attribute declarations in this chapter. We are armed with enough knowledge to write the XSD schema for the root element of the order information XML document.

This is how the root element should look.

```
<OrderInfo AgencyCode="S008">
   <Order />
   <Order />
   <Order />
</OrderInfo>
```

Listing 6.39: Structure of the Order Information Root element.

The schema that we write for the root element should implement the following rules.

- The name of the root element should be *OrderInfo.*
- There may be multiple *Order* elements under the root element. Each *Order* element will hold the data of a single order. If the *OrderInfo* has the data of three orders, there should be three *order* elements under the *OrderInfo* element. *OrderInfo* element should contain at least one *Order* element and there is no maximum limit.
- *OrderInfo* element should have an attribute named *AgencyCode*.
- The *AgencyCode* attribute is mandatory and should be exactly four characters long.
- The first character of the *AgencyCode* should be an alpha and the other three should be digits.

Let us translate these rules into XSD code. Let us take the first rule and see how we could translate it into XSD.

Rule 1

The name of the root element should be "OrderInfo"

Let us see how the first rule can be translated to an XSD declaration. Here is the declaration of the first rule that defines the root element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo"/>
</xsd:schema>
```

Listing 6.40: Declaration of the "OrderInfo" element.

In Chapter 2 we have learned to create basic schema declarations. We discussed element declarations in Chapter 5. So the above schema must look very familiar to you, as it includes only the stuff we have learned so far.

Let us create an XML Schema Collection with the above schema definition and see the schema validation in action.

```
CREATE XML SCHEMA COLLECTION Lab1 AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo"/>
</xsd:schema>'
GO
```

Listing 6.41: Schema collection for the root element.

Let us now validate an XML instance with this schema

```
-- test with an XML variable
DECLARE @x1 XML(Lab1)
DECLARE @x2 XML(Lab1)


-- an empty element
SET @x1 = '<OrderInfo></OrderInfo>'
SET @x2 = '<OrderInfo />'

SELECT @x1 AS x1, @x2 AS x2

/*
output:

x1                 x2
----------------   ----------------
<OrderInfo />      <OrderInfo />
*/
```

Listing 6.42: Validating an XML instance.

Look a second at the above example. Both @x1 and @x2 store an empty element. SQL Server stores an empty element as *"<elementname />"* even if you provided the value as *"<elementname></elementname>."*

Since both XML variables are bound to the schema collection *Lab1*, SQL Server will validate the XML values being assigned to them. SQL Server will accept the value only if the element name is *"OrderInfo."* The *"OrderInfo"* element is not supposed to store any value; instead, it is meant to hold the child elements containing order information. But the current version of the schema allows a text value in the *"OrderInfo"* element, as shown in the example below.

```
-- declare a variable
DECLARE @x1 XML(Lab1)

-- assign a text value
SET @x1 = '<OrderInfo>some text</OrderInfo>'
```

157

```
-- read the value
SELECT @x1 AS 'x1'

/*
output:

x1
-------------------------------
<OrderInfo>some text</OrderInfo>
*/
```

Listing 6.43: Storing a text value to "OrderInfo" element.

This is an undesired behavior and will be restricted when the other rules are added to the schema definition.

We validated our schema collection with XML variables. Let us now perform the same validation against an XML column.

```
-- test with an XML column
-- create a memory table with an XML column
DECLARE @t TABLE (Data XML(Lab1))

-- insert some data
INSERT INTO @t(Data) SELECT '<OrderInfo></OrderInfo>'
INSERT INTO @t(Data) SELECT '<OrderInfo />'
INSERT INTO @t(Data) SELECT '<OrderInfo>Some text</OrderInfo>'

-- read values from the table
SELECT * FROM @t

/*
output:

Data
----------------
<OrderInfo />
<OrderInfo />
<OrderInfo>Some text</OrderInfo>
*/
```

Listing 6.44: Performing validations against an XML column.

You could see that the results are identical to what we have seen with XML variable.

Rule 2:

> **There may be multiple Order elements under the root element.**
>
> **Each Order element will hold the data of a single order.**
>
> **If the OrderInfo has the data of three orders, there should be three Order elements under the OrderInfo element.**
>
> **OrderInfo element should contain at least one Order element and there is no maximum limit.**

This rule describes the children of *"OrderInfo."* It has only one child element named *"Order."* There should be at least one *"Order"* element under *"OrderInfo"* and there is no maximum limit. Let us enhance the schema to include these rules.

I mentioned earlier that only a Complex Type can have child elements. We have not seen complex types in detail. Refer to Chapter 10 for a detailed discussion on Complex Types.

Before we could add a child element to *"OrderInfo,"* we should mark it as a *Complex Type*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.45: An element having Complex Type declaration.

All the child elements of a Complex Type should appear between the *complexType* element. As I mentioned earlier, the order of elements is significant in XML. So before we declare the child elements, we need to decide the order of child elements. If we need a specific order, the child element declarations should appear within a *sequence* block. If we don't need any specific order, we could use an *all* block.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:all>
      </xsd:all>
    </xsd:complexType>
```

```
   </xsd:element>
</xsd:schema>
```

Listing 6.46: Using order indicator "all."

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 6.47: Using order indicator "sequence."

There is only one child element under *"OrderInfo"* and it does not make a difference whether we use *"all"* or *"sequence."* However, we need to use *sequence* for this example because we need to allow unlimited number of *Order* elements. This works only with *sequence* indicator.

After the *sequence* block is declared, we could add the declaration of the *Order* element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd: sequence>
         <xsd:element name="Order"/>
       </xsd: sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 6.48: Declaration of "Order" element.

The next step is to control the *occurrence* of the *Order* element. This can be controlled by using the *"minOccurs"* and *"maxOccurs"* attributes.

The default value of *"minOccurs"* is 1; therefore, all elements should appear at least once. So the following definition is equivalent to the example given above.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Order"
                      minOccurs="1" />
       </xsd:sequence>
     </xsd:complexType>
```

```
    </xsd:element>
</xsd:schema>
```

Listing 6.49: Using "minOccurs" to control occurrence.

Though both usages are equivalent, I would prefer to use the second approach where we explicitly specify the occurrence for the sake of clarity.

Occurrence indicator *"maxOccurs"* is used to control the maximum occurrence of an element. By setting it to *"unbounded"* we could allow unlimited number of occurrences of an element.

> *"unbounded"* **can be used only when the** *order indicator* **is** *"sequence."* **It cannot be used with** *"all."*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                    minOccurs="1"
                    maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.50: using "maxOccurs" to control occurrence.

Let us create a Schema Collection with enhanced version of the schema and test it against an XML instance.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'Lab1'
) BEGIN
      DROP XML SCHEMA COLLECTION Lab1
END
GO

-- CREATE the SCHEMA COLLECTION with the updated
-- definition.
CREATE XML SCHEMA COLLECTION Lab1 AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                    minOccurs="1"
                    maxOccurs="unbounded"/>
```

```
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>'
GO
```

Listing 6.51: Recreating the Schema Collection with new definition.

As we have discussed in the previous chapters, there is no way to modify a Schema Collection. If we want to alter the schema definition, we need to drop the schema collection and create it again. This will be a pain if there are columns bound to the Schema Collection that we want to modify. If a Schema Collection is bound to a column, it cannot be dropped unless all the references are removed.

Let us try to see if the XML instance that validated with the previous version of the schema still gets validated or not.

```
DECLARE @x XML(Lab1)

SET @x =
'<OrderInfo>
</OrderInfo>'
```

Listing 6.52: This XML instance is invalid because it does not have an "Order" element, which is declared as mandatory.

If you run the above code, SQL Server will generate an error because there is no *"Order"* element present in the XML instance. Per the schema, there should be at least one *Order* element in the XML instance and, accordingly, this value is invalid.

Both the examples given below are valid because the Schema Collection accepts any number of *"Order"* elements.

```
DECLARE @x XML(Lab1)

SET @x =
'<OrderInfo>
      <Order />
</OrderInfo>'
```

Listing 6.53: OrderInfo with a single "Order" element.

```
DECLARE @x XML(Lab1)

SET @x =
'<OrderInfo>
      <Order />
```

```
      <Order />
      <Order />
</OrderInfo>'
```

Listing 6.54: "OrderInfo" with three "Order" elements.

Rule 3

OrderInfo element should have an attribute named "AgencyCode."

The third rule adds an attribute to the *"OrderInfo"* element. We have learned attribute declaration earlier in this chapter. Let us add an attribute declaration for *AgencyCode.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="AgencyCode"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 6.55: Declaration of "AgencyCode" attribute.

Note that the attribute declaration should occur inside the *complexType* declaration. If the *complexType* contains an order indicator (all, sequence, or choice) the attribute declaration should appear after the order indicator. The following Schema is invalid because the attribute declaration occurs before the order indicator (*sequence*) element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:attribute name="AgencyCode"/>
       <xsd:sequence>
         <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
       </xsd:sequence>
     </xsd:complexType>
```

```
   </xsd:element>
</xsd:schema>
```

Listing 6.56: Attributes should be declared after order indicators.

Rule 4

> **The "AgencyCode" attribute is mandatory and should be exactly four characters long.**

The previous rule added a declaration for the *AgencyCode* attribute. Now we need to enhance the attribute declaration with two validations. The first part of the rule is to make the attribute mandatory and the next part is to validate the length of the attribute value.

By default an attribute is optional. We can make an attribute mandatory by using the *"use"* attribute. To make an attribute optional, the *"use"* attribute should be set to *"optional"* and to make it mandatory *"use"* should be set to *"required."* The default value of *"use"* is *"optional"* and consequently the following two Schema definitions are equivalent.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="AgencyCode"
                     use="optional"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.57: Attribute "AgencyCode" is explicitly set to "optional."

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="AgencyCode" />
```

```
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 6.58: Attribute "AgencyCode" is implicitly set to "optional."

The following Schema sets *"AgencyCode"* as mandatory.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Order"
                      minOccurs="1"
                      maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="AgencyCode"
                      use="required"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 6.59: Using "required" to set an attribute as mandatory.

Rule 5

***The first character of the "AgencyCode" should be an alphabet
and the other three should be digits.***

The implementation of this rule needs a format validation. XSD supports a
regular expression language similar to the regular expression languages
supported by Microsoft .NET and Perl. We have seen a basic example
using a regular expression pattern in Chapter 4. Refer to Chapter 12 for a
detailed discussion on the regular expression language supported by XSD.

Before applying a format validation, the attribute should be declared as a
simpleType. Only a *simpleType* can take a pattern restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Order"
                      minOccurs="1"
                      maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="AgencyCode"
                      use="required">
       <xsd:simpleType>
       </xsd:simpleType>
```

```
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
```

Listing 6.70: Making an attribute "simpleType."

The next step is to add an *"xsd:restriction"* element to the *simpleType.* The restriction should be based on *"xsd:string"* because *AgencyCode* is an alpha-numeric value.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="AgencyCode"
                     use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 6.71: Adding a restriction to a simple Type.

After declaring the *"xsd:restriction"* element, we could add a *pattern* restriction with a regular expression pattern.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Order"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
      <xsd:attribute name="AgencyCode"
                     use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{1}[0-9]{3}" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
```

```
        </xsd:complexType>
     </xsd:element>
</xsd:schema>
```

Listing 6.72: Adding a regular expression pattern to an attribute.

The regular expression pattern specifies that the first character should be alphabets in either lower or upper case. Then the next three3 should be digits between 0 and 9.

[a-zA-z] stands for a single occurrence of a single lower case character within the range of *"a"* to *"z"* or an upper case character within the range of *"A"* to *"Z."*

{3} restricts the occurrence of the preceding characters to be EXACTLY three times. Hence, *[a-zA-Z]{3}* translates to the occurrence of three characters of the English alphabet in either lower or upper case.

Similarly, *[0-9]* stands for a single occurrence of any of the digits between 0 and 9. The whole expression *[0-9]{3}* stands for the occurrence of three digits between 0 and 9.

Refer to Chapter 12 for a detailed explanation of the Regular expression language of XSD.

Let us create a Schema Collection with the new schema definition and see the validation in action.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'Lab1'
) BEGIN
      DROP XML SCHEMA COLLECTION Lab1
END
GO

-- CREATE the SCHEMA COLLECTION with the updated
-- definition.
CREATE XML SCHEMA COLLECTION Lab1 AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="OrderInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Order"
                      minOccurs="1"
                      maxOccurs="unbounded"/>
       </xsd:sequence>
       <xsd:attribute name="AgencyCode"
                      use="required">
```

```
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{3}[0-9]{3}" />
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 6.73: Schema showing a pattern restriction.

Let us try to validate the XML instance that successfully got validated with the previous version of the schema.

```
DECLARE @x XML(Lab1)

SET @x =
'<OrderInfo>
      <Order />
      <Order />
      <Order />
</OrderInfo>'
```

Listing 6.74: Invalid XML value as it is missing the AgencyCode attribute.

If you run the above code, SQL Server will generate an error message. This XML instance was good for the previous version of the schema. But we added a new mandatory attribute in this version of the schema and, consequently, SQL Server will not accept this XML value.

This version of the schema will accept an XML value only if it satisfies all the rules defined with the previous version of the schema, plus the new rules. The XML instance should have an attribute named *AgencyCode* and it should be exactly six characters long. The first three characters should be alphabets and the other three should be digits. Here is an XML instance that validates with the above schema.

```
DECLARE @x XML(Lab1)

SET @x =
'<OrderInfo AgencyCode="JAC123" >
      <Order />
      <Order />
      <Order />
</OrderInfo>'
```

Listing 6.75: Example of a correct XML instance for Phase1 schema.

Try running different variations of the above XML instance and you will realize that SQL Server will accept only those values that pass all the validations we have defined in the schema so far.

Chapter Summary

We have had a comprehensive overview of *attribute declarations* in this chapter. We had a quick discussion on some fundamental differences between elements and attributes. Elements can hold other elements and attributes. Attributes can only store a value. Elements can appear without an attribute, but attributes can appear only within elements. Elements are mandatory by default, whereas attributes are optional. An element can appear more than once under a parent node, but attributes can appear only once. Position of elements is significant in XML but attributes can appear in any order.

Attribute declarations can be global or local. Global attribute declarations can be referred (re-used) within the declaration of other *complexTypes*. Global attribute declarations provide re-usability to a certain extend.

An attribute declaration takes a number of attributes. *"name"* is the only mandatory attribute that should exist in an attribute declaration. Both global and local attribute declarations should have a *"name"* attribute.

Global attribute declarations can have the following additional attributes: *id, type, default* and *fixed.* Local attribute declarations can have all the attributes of global attribute declarations, plus three additional attributes, namely: *ref, use* and *form.*

The *name* of an attribute must start with a letter or underscore. An attribute name can contain (except for the first character) letters, digits, underscores, hyphens and periods.

"id" is an optional attribute which uniquely identifies an attribute. The value of the *"id"* attribute should be unique within the schema document. The *"type"* attribute associates an XSD data type with an attribute. This provides better validation of the attribute value. The schema processor will make sure that the value of the attribute is valid for the data type declared in the schema.

The *"default"* attribute assigns a default value to the *attribute.* If the attribute is missing in the XML instance, the default value will be assigned to the attribute. However, if the attribute is present and holds an empty value, the default value is not assigned.

The *"fixed"* attribute restricts the value of the attribute to a predefined value. If the attribute is missing in the XML instance, the value declared with *fixed* is assumed. If the attribute is present, it should contain the same value declared with the *fixed* attribute.

"ref" can be used to refer to a global attribute declaration within a *complexType.* Attributes are optional by default. An attribute can be set to mandatory by declaring the *"use"* attribute with value:*" required."*

Attributes can be grouped into *Attribute Groups* and can be reused in one or more *complexType* declarations. This provides a greater level of reusability. Attribute group declarations can be nested.

CHAPTER 7
XSD PRIMITIVE DATA TYPES

XSD supports almost fifty built-in data types. While declaring an element or attribute, you can associate it with a data type to make sure that SQL Server will only accept a value which conforms to that data type. The same applies if you want to update the value – the new value will have to respect the associated data type.

In the previous chapters, we have seen several examples that use data types. By using data types, you perform an extra level of validation on the value stored in an element or attribute. One of the examples that we discussed several times is the *Employee* data where the *age* attribute is declared as *nonNegativeInteger*. This declaration makes sure that the *age* attribute does not accept negative values. Then we applied a minimum and maximum restriction to make sure that the values are always within the accepted range.

In this chapter, we will examine the Primitive data types of XSD. We will discuss the following.

- Importance of Data Types
- Characteristics of Data Types
- Primitive Data Types

After discussing the Primitive Data Types, we will do a hands-on lab which is a continuation of the lab we did in the previous chapter.

Importance of Data Types

I am pretty sure that the importance of data types is clear enough from many of the examples we saw in the previous chapters. Almost all programming languages use data types to make sure that correct values are stored to variables and correct operations are done using those variables. When we create a table we always create columns with specific data types based on the storage requirements.

Generally speaking, when you associate a variable or column to a data type you are basically restricting the values that the variable or column can store and restricting the permissible operations on them. For example,

numbers cannot be concatenated (they can be added) and strings cannot be added (they can be concatenated only).

At the minimum, most programming languages support strings, numbers and date data types. XSD supports a number of different data types to describe and validate almost all values that we might need to work with. Further, it supports deriving new data types from the built-in data types, in case a built-in data type does not adequately suit a specific validation requirement.

Data types help describe a certain piece of data more accurately and help validate them more efficiently. In the absence of a date data type, you might need to apply a very complex pattern restriction on a string value to make sure that the value follows *"yyyy-mm-dd"* format. Further, you'd need to add validations to make sure that the month is not more than 12 and the day is not over 30 when the month is 04, and so on. The existence of a *Date* data type simplifies this. When you associate a piece of data to a *Date* data types, all these validations are automatically applied on it.

This means that the more data types you have, the easier it is for you to write the schemas. Many of the validations that you need might already be part of an existing data type and, as a result, you can readily use them. As shown in the sample application we discussed in Chapter 3, we need to validate zip codes (only five digits allowed) and phone numbers (should follow the format: (999) 999 9999). If there were data types available for *zipCode* and *phoneNumber* it would have been easier for us to write those validations.

XSD supports almost fifty data types. They can be divided into the following categories.

- Primitive Data Types
  Primitive Data Types are the base data types of XSD. This means that they themselves have not been derived from another type.
- Derived Data Types
  These are Data Types derived directly or indirectly from Primitive Data Types.

This chapter will discuss the Primitive Data Types. We will discuss the Derived Data Types in Chapter 9.

Characteristics of XSD Data Types

There are a few characteristics of an XSD data type that I would like to discuss here to make sure that you understand them correctly. You will hear these characteristics frequently discussed when people talk about XSD data types.

Note: You may skip this topic at the first reading. If it sounds too confusing, jump to the section *facets* below.

Value Space

The value space of a data type is the set of values for that data type. Let us look at an example to understand this.

The value space of *boolean* data type is a two valued logic (true/false). The value space of a date data type specifies what values are acceptable as the *year, month* and *day* part of a *Date* data type.

The value space of a *Date* data type restricts the maximum value of the *Month* part to be 12. Further, it will validate the *day* part to make sure that only certain months are allowed to have 31, 30 and 29 days.

Lexical Space

The lexical space of a data type is the set of valid literals for that data type. To understand this, let us take the same example we took for understanding the value space of a data type.

The lexical space of a *boolean* data type (in XSD) is *"1," "0," "true" and "false."* These are the four different literals that a *boolean* data type can accept. While *true* and *false* are in the lexical space of *boolean, yes* and *no* are not. Hence, a *boolean* data type cannot accept the literal *yes* or *no.*

The Lexical space of a date data type is *"yearpart-mothpart-daypart"* where *"yearpart"* can take an optional negative sign along with a four-digit year, *"monthpart"* can take a value between 1 and 12 and *"daypart"* can take a value between 1 and 31.

Another example is the lexical space of float data type. *"100"* and *"1.0E2"* are two different literals from the lexical space of float data type. Both literals refer to the same value and they are accepted as valid float values.

Lexical Representation

Each literal in the lexical space of a data type is its *lexical representation*. The members of the lexical space of *boolean* are *"true," "false," "1"* and *"0."* We could say that *"true," "false," "1"* and *"0"* are lexical representations of *boolean.*

Canonical Lexical Representation

In the previous section, we saw the lexical space of a data type. *"true"* and *"1"* are valid literals from the lexical space of *boolean.* Both *"1"* and *"true"* refer to the same value. Similarly, in the previous section we saw two different literals from the value space of float. *"100"* and *"1.0E2"* both are from the lexical space of *float*, referring to the same value.

Thus, sometimes, a value in the value space of a data type may be represented by more than one lexical representation. A *Canonical Lexical Representation* is a set of literals among the valid set of literals for a data type such that there is a one-to-one mapping between literals in the canonical lexical representation and values in the value space

Primitive Data Types

Primitive Data Types are base data types from which other data types are derived. XSD has nineteen primitive data types and SQL Server supports eighteen of them. SQL Server does not support XSD data type *"NOTATION."* The following is the list of primitive data types of XSD.

string	boolean	decimal
float	double	duration
dateTime	time	date
gYearMonth	gYear	gMonthDay
gDay	gMonth	hexBinary
base64Binary	anyURI	QName
NOTATION		

Some of these data types must be familiar to you. You might find them in other programming languages as well. *String, boolean, decimal, float, double, dateTime, time, date,* etc., are present in almost all programming languages.

However, many of you will be new to data types like *duration, gYearMonth, QName, NOTATION, anyURI, gYear, gMonthDay, gDay, gMonth,* etc. In this chapter we will go over all the XSD primitive data types in detail.

Data Type: string

I am pretty sure that the *string* data type does not need any explanation at all. Let me simply say that it is used to store a sequence of characters. Almost all programming languages support *the string datatype.*

Let us declare the schema of a simple XML element having *string* data type.

```
--Create XML SCHEMA collection
CREATE XML SCHEMA COLLECTION StringDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="myString" type="xsd:string" />
</xsd:schema>'
GO

-- Declare a variable bound to the schema
DECLARE @x XML(StringDemo)
SET @x = '<myString>this is a string</myString>'
```

Listing 7.1: An example using "string" data type.

The above example is very elementary and explains the usage of a *string* data type. Now let us have a look at a few interesting points related to the *string* data type.

There are certain characters that are not allowed in the value of an element or attribute. These are some control characters having some specific meaning and cannot be mixed up with values of elements and attributes. The following characters have special meaning in XML; therefore, you cannot use them in the value of an attribute or element.

- (&) – Ampersand
- (<) – Less Than
- (>) – Greater Than
- (') – Apostrophe
- (") – Quotation Mark

Most XML processors will generate an error if they find any of these characters present in the value of an element or attribute.

When you work with XML in SQL Server, you need to encode only two characters from the above list: "&" and "<."

SQL Server will generate an error if you attempt to run the following code.

```
DECLARE @x XML(StringDemo)
SET @x = '<myString>Jacob & John</myString>'
```

Listing 7.2: This XML instance is invalid because "&" is a restricted character.

The problem is the *"&"* character. It is an illegal character and should not be present as the value of an element or attribute. If you want to represent this character, you should use the named entity **&.**

The following code will execute successfully.

```
DECLARE @x XML(StringDemo)
SET @x = '<myString>Jacob &amp; John</myString>'
```

Listing 7.3: Use the named entity instead of "&" to represent an ampersand.

When you read the value from the XML variable, the entity name will be correctly decoded and you will get an "&" back.

```
DECLARE @x XML(StringDemo)
SET @x = '<myString>Jacob &amp; John</myString>'

-- read value from the XML variable
SELECT @x.value('myString[1]','VARCHAR(20)') AS Name

/*
Name
--------------------
Jacob & John
*/
```

Listing 7.4: SQL Server will do a decoding of named entities.

The same rules are applicable to *"<"* character. If the value of an element or attribute contains this character, SQL Server will raise an error. The *"<"* character has to be encoded using the entity name **<.**

```
-- the following code will generate an error because
-- of the "<" character in the value of <myElement>
DECLARE @x XML(StringDemo)
SET @x = '<myString>if x < y then g to end</myString>'

-- However, the code will execute correctly if we encode it
-- using &lt; literal.
DECLARE @x XML(StringDemo)
SET @x = '<myString>if x &lt; y then g to end</myString>'
```

Listing 7.5: Encoding is required for "<" as well.

Interestingly, SQL Server accepts certain characters that are not usually accepted by other XML parsers without encoding. All three examples given in the following snippet will execute correctly in SQL Server.

```
-- lets try ">" first. It works without encoding.
DECLARE @x1 XML(StringDemo)
SET @x1 = '<myString>if x > y then go to end</myString>'

-- well, next is " (double quotes)
DECLARE @x2 XML(StringDemo)
SET @x2 = '<myString>He shouted: "Stop There!"</myString>'

-- let us try ' (single quotes) too. (Note that I used two
-- quotes. The second quote is used to escape the character
-- because the whole XML value is enclosed by single quotes.)
DECLARE @x3 XML(StringDemo)
SET @x3 = '<myString>That is Robert''s hat</myString>'
```

Listing 7.6: No encoding is required for ">," double quotes ("), and single quotes (').

The behavior of the string data type is almost identical for elements and attributes, with the exception of handling double quotes ("). Elements can accept double quotes without encoding, but attributes need double quotes to be encoded.

The following code declares the schema of an XML document which has a *string* type attribute.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'StringDemo'
) BEGIN
      DROP XML SCHEMA COLLECTION StringDemo
END
GO

--Create XML SCHEMA collection
CREATE XML SCHEMA COLLECTION StringDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="myString" >
     <xsd:complexType>
        <xsd:attribute name="stringData" type="xsd:string" />
     </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Declare a variable bound to the schema
DECLARE @x XML(StringDemo)
SET @x = '<myString stringData="My String Data" />'
```

Listing 7.7: Example of a schema showing an attribute with string data type.

The following is illegal. SQL Server will throw an error while processing the following code.

```
-- Here is the correct code
DECLARE @x XML(StringDemo)
SET @x = '<myString stringData="He shouted: "Hi There!" " />'
```

Listing 7.8: Attributes cannot take double quotes without encoding.

Use entity name **"** to represent a double quote in the value of an attribute.

```
-- Here is the correct code
DECLARE @x XML(StringDemo)
SET @x = '<myString stringData="He shouted: &quot;Hi There!&quot;
          " />'
```

Listing 7.9: Named entity " represents a double quote.

Data Type: boolean

SQL Server's implementation of *boolean* data type accepts *"true," "false," "1"* and *"0"* as valid boolean values.

The following example shows the usage of the boolean data type.

```
--Create an XML SCHEMA collection
CREATE XML SCHEMA COLLECTION BooleanDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="isAlive" type="xsd:boolean" />
</xsd:schema>'
GO

-- supports true
DECLARE @x1 XML(BooleanDemo)
SET @x1 = '<isAlive>true</isAlive>'

-- supports false
DECLARE @x2 XML(BooleanDemo)
SET @x2 = '<isAlive>false</isAlive>'

-- supports 1
DECLARE @x3 XML(BooleanDemo)
SET @x3 = '<isAlive>1</isAlive>'

-- supports 0
DECLARE @x4 XML(BooleanDemo)
SET @x4 = '<isAlive>0</isAlive>'
```

Listing 7.10: An example showing the usage of boolean data type.

When you create an XML instance, you can use literal *"1"* to represent *"true"* and *"0"* to represent *"false."* SQL Server stores the value internally as *"true"* or *"false"* only. There is no way you can get the *"1"* or *"0"* back from a boolean attribute or element.

```
DECLARE @x1 XML(BooleanDemo)
SET @x1 = '<isAlive>true</isAlive>'
SELECT @x1.value('isAlive[1]','varchar(10)') AS IsAlive

/*
IsAlive
----------
true
*/
DECLARE @x2 XML(BooleanDemo)
SET @x2 = '<isAlive>1</isAlive>'
SELECT @x2.value('isAlive[1]','varchar(10)') AS IsAlive

/*
IsAlive
----------
true
*/

SELECT @x2
/*
-----------------------
<isAlive>true</isAlive>
*/
```

Listing 7.11: SQL Server's internal representation of boolean data type uses "true" and "false." It does not use "1" or "0."

If you attempt to cast a boolean value to a number, you will get an error. Here is an example.

```
DECLARE @x2 XML(BooleanDemo)
SET @x2 = '<isAlive>1</isAlive>'

SELECT @x2.value('isAlive[1]','int') AS IsAlive

/*
IsAlive
-----------
JAC\SQL2005(JAC\JACOB): Msg 245, Level 16, State 1, Line 4
Conversion failed when converting the nvarchar value 'true' to
data type int.
*/
```

Listing 7.12: Since SQL Server stores a boolean value as "true" or "false," and not as "1" or "0," the casting of a boolean value to integer will fail.

Also, note that the values *true* and *false* are case sensitive. The following code will generate an error.

```
DECLARE @x2 XML(BooleanDemo)
SET @x2 = '<isAlive>True</isAlive>'

/*
JAC\SQL2005(JAC\JACOB): Msg 6926, Level 16, State 1, Line 2
XML Validation: Invalid simple type value: 'True'. Location:
/*:isAlive[1]
*/
```

Listing 7.13: Literals "true" and "false" are case sensitive.

Data Type: decimal

XSD *decimal* data type represents a subset of real numbers. It takes a negative or positive value, with or without decimal fraction.

The following example shows an XSD schema which declares a *decimal* element and assigns different values to it.

```
--Create an XML SCHEMA collection for "decimal" type
CREATE XML SCHEMA COLLECTION DecimalDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Wealth" type="xsd:decimal" />
</xsd:schema>'
GO

-- declare variables bound to the schema
DECLARE
      @x1 XML(DecimalDemo),
      @x2 XML(DecimalDemo),
      @x3 XML(DecimalDemo),
      @x4 XML(DecimalDemo)

-- assign values to each XML variable
SELECT
      @x1 = '<Wealth>0012</Wealth>',
      @x2 = '<Wealth>+12</Wealth>',
      @x3 = '<Wealth>-12</Wealth>',
      @x4 = '<Wealth>123456789.98765432100</Wealth>'
```

Listing 7.14: example showing decimal data type.

Data Type: float

XSD *float* data type is single precision 32-bit floating point type. Unlike *decimal*, *float* can store special values like *Positive infinity, Negative Infinity* and *NaN (Not a Number)*.

Let us create the schema of an XML document that has a *float* data type.

```
--Create an XML SCHEMA collection for "float" type
CREATE XML SCHEMA COLLECTION FloatDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="AFloatValue" type="xsd:float" />
</xsd:schema>'
GO
```

Listing 7.15: An example showing the "float" data type.

Let us look at an example that shows how the above schema behaves with different float values.

```
-- declare variables bound to the schema
DECLARE
      @x1 XML(FloatDemo),
      @x2 XML(FloatDemo),
      @x3 XML(FloatDemo),
      @x4 XML(FloatDemo),
      @x5 XML(FloatDemo),
      @x6 XML(FloatDemo),
      @x7 XML(FloatDemo),
      @x8 XML(FloatDemo),
      @x9 XML(FloatDemo)

-- assign values to each XML variable
SELECT
      @x1 = '<AFloatValue>0</AFloatValue>',
      @x2 = '<AFloatValue>-0</AFloatValue>',
      @x3 = '<AFloatValue>INF</AFloatValue>',
      @x4 = '<AFloatValue>-INF</AFloatValue>',
      @x5 = '<AFloatValue>10e-2</AFloatValue>',
      @x6 = '<AFloatValue>-1e3</AFloatValue>',
      @x7 = '<AFloatValue>1234.5678e9</AFloatValue>',
      @x8 = '<AFloatValue>16.3e-2</AFloatValue>',
      @x9 = '<AFloatValue>125</AFloatValue>'
```

Listing 7.16: A TSQL example demonstrating the different values that a float data type can store.

Note the usage of *-INF, INF* and *-0*. Though the XSD specification supports a special value: *NaN (Not a Number),* the XSD implementation of SQL Server does not support it. Hence, the following code will fail.

```
DECLARE @x1 XML(FloatDemo)
SET @x1 = '<AFloatValue>NaN</AFloatValue>'

/*
JAC\SQL2005(JAC\JACOB): Msg 6926, Level 16, State 1, Line 2
```

```
XML Validation: Invalid simple type value: 'NaN'. Location:
/*:AFloatValue[1]
*/
```

Listing 7.17: SQL Server does not support "NaN."

Data Type: double

XSD *double* data type shares the same characteristics as *float* except that *double* data type is double precision 64-bit floating point type. Other than that, *double* shares the same characteristics as *float* data type.

```
--Create an XML SCHEMA collection for "double" type
CREATE XML SCHEMA COLLECTION DoubleDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ADoubleValue" type="xsd:double" />
</xsd:schema>'
GO

-- declare variables bound to the schema
DECLARE
      @x1 XML(DoubleDemo),
      @x2 XML(DoubleDemo),
      @x3 XML(DoubleDemo),
      @x4 XML(DoubleDemo),
      @x5 XML(DoubleDemo),
      @x6 XML(DoubleDemo),
      @x7 XML(DoubleDemo),
      @x8 XML(DoubleDemo),
      @x9 XML(DoubleDemo)

-- assign values to each XML variable
SELECT
      @x1 = '<ADoubleValue>0</ADoubleValue>',
      @x2 = '<ADoubleValue>-0</ADoubleValue>',
      @x3 = '<ADoubleValue>INF</ADoubleValue>',
      @x4 = '<ADoubleValue>-INF</ADoubleValue>',
      @x5 = '<ADoubleValue>10e-2</ADoubleValue>',
      @x6 = '<ADoubleValue>-1e3</ADoubleValue>',
      @x7 = '<ADoubleValue>1234.5678e9</ADoubleValue>',
      @x8 = '<ADoubleValue>16.3e-2</ADoubleValue>',
      @x9 = '<ADoubleValue>125</ADoubleValue>'
```

Listing 7.18: An example demonstrating the different values accepted by "double" data type.

Just like *float,* SQL Server's XSD implementation of *double* data type does not support *NaN*.

Data Type: duration

XSD duration data type represents duration of time. Duration of time is represented by number of years, number of months, number of days, number of hours, number of minutes and number of seconds.

```
--Create an XML SCHEMA collection for "duration" type
CREATE XML SCHEMA COLLECTION DurationDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ElapsedTime" type="xsd:duration" />
</xsd:schema>'
GO
```

Listing 7.19: A schema with an element of "duration" type.

A duration value should always start with the letter "P" in upper case. Then each value should be separated by indicators Y (Years), M (Months), D (Days), H (Hours), M (Minutes) and S (Seconds). Letter "T" should appear as a separator between Year-Month-Day and Hour-Minute-Second.

The following snippet shows a few examples.

```
DECLARE  @x XML(DurationDemo)

-- 20 seconds
SET @x = '<ElapsedTime>PT20S</ElapsedTime>'

-- 3 minutes
SET @x = '<ElapsedTime>PT3M</ElapsedTime>'

-- 3 minutes and 15 seconds
SET @x = '<ElapsedTime>PT3M15S</ElapsedTime>'

-- 6 hours
SET @x = '<ElapsedTime>PT6H</ElapsedTime>'

-- 2 days
SET @x = '<ElapsedTime>P2D</ElapsedTime>'

-- 2 days and 5 hours
SET @x = '<ElapsedTime>P2DT5H</ElapsedTime>'

-- 3 months
SET @x = '<ElapsedTime>P3M</ElapsedTime>'

-- 1 year
SET @x = '<ElapsedTime>P1Y</ElapsedTime>'

-- 2 years and 3 days
SET @x = '<ElapsedTime>P2Y3D</ElapsedTime>'
```

Listing 7.20: "duration" data type example.

Duration can take a negative value, too. In the case of a negative value, the negative sign should occur at the beginning of the value, on the left of **"P."**

```
DECLARE  @x XML(DurationDemo)

-- 2 years back
SET @x = '<ElapsedTime>-P2Y</ElapsedTime>'
```

Listing 7.21: "duration" data type can take a negative value too.

Data Type: dateTime

XSD *dateTime* data type represents a value that contains date and time information. It is very close to the DATETIME data type of SQL Server.

The format of XSD *dateTime* is **-yyyy-mm-ddThh:mm:ss.sZ**. The format is self explanatory, except for the part marked in yellow. Note that there is a **"T"** which separates the date part and the time part.

Part of the format marked in yellow needs some attention. The year value can take a negative sign which indicates date in BC. So **-0002-01-01T12:00:00** represents BC 2, January 1 midnight.

The **time** part can store the fraction of seconds after the **"ss"** part. The separator between seconds and fraction of seconds is a period. Finally, time zone representation should follow the **dateTime** value.

Time zone information is optional in XSD. However, SQL Server 2005's implementation of **dateTime** data type requires time zone information along with the date value. This requirement has been removed in SQL Server 2008.

Let us look at a few examples now.

```
--Create an XML SCHEMA collection for "dateTime" type
CREATE XML SCHEMA COLLECTION dateTimeDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MeetingSchedule" type="xsd:dateTime" />
</xsd:schema>'
GO
```

Listing 7.22: A schema having a "dateTime" element.

```
DECLARE  @x XML(dateTimeDemo)

-- We do not have a fraction of second in this example
```

```
SET @x = '<MeetingSchedule>2007-11-01T14:15:00Z</MeetingSchedule>'

-- here we have added the fraction of seconds
SET @x = '<MeetingSchedule>2007-11-
01T14:15:00.00Z</MeetingSchedule>'

-- You can make the 'fraction of second' incredibly long :-)
SET @x = '<MeetingSchedule>
          2007-11-01T14:15:00.00000000000000000000000000000Z
        </MeetingSchedule>'

-- this date falls in BC
SET @x = '<MeetingSchedule>-0002-11-
01T14:15:00Z</MeetingSchedule>'

-- but this one is illegal. You can't have a '+' sign.
--SET @x = '<MeetingSchedule>+0002-11-
01T14:15:00Z</MeetingSchedule>'
```

Listing 7.23: "dateTime" data type example.

Note the usage of **'Z'** at the end of the time value to represent the time zone information. **'Z'** indicates that the time value is in UTC (GMT). You can also express time in your local time zone by specifying the time difference. The following example shows this.

```
-- the following two dates are equal
-- representation in UTC
SET @x = '<MeetingSchedule>2007-11-01T13:30:00Z</MeetingSchedule>'
-- representation in local time zone (IST)
SET @x =
'<MeetingSchedule>2007-11-01T19:00:00+05:30</MeetingSchedule>'
```

Listing 7.24: Time zone can be specified either in UTC or by the time difference

The hour part can take a value of 24 if the minute and seconds are 0. In this case, it represents the first second of the next day.

```
DECLARE  @x XML(dateTimeDemo)

-- hour can be 24 if the minutes and seconds are 0
-- it represents the first second of the next day
SET @x = '<MeetingSchedule>2007-11-01T24:00:00Z</MeetingSchedule>'

-- the following is illegal
--SET @x = '<MeetingSchedule>2007-11-
01T24:00:00.01Z</MeetingSchedule>'
```

Listing 7.25: The "hour" part of a dateTime value can accept 24, if "minutes" and "seconds" are zero.

Enhancements in SQL Server 2008

SQL Server 2008 added two enhancements to the date/time data types (*date*, *time* and *dateTime*). Time zone information was mandatory in SQL Server 2005, though the XSD specification states that time zone is optional for *date*, *time* and *dateTime* data types. In SQL Server 2008, time zone information is optional for *date, time* and *dateTime* data types.

The XML instance given in the following example will not validate in SQL Server 2005, because time zone information is missing in the *dateTime* value.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MeetingSchedule" type="xsd:dateTime" />
</xsd:schema>'
GO

DECLARE  @x XML(ExampleSchema)
SET @x = '<MeetingSchedule>2007-11-01T10:30:00</MeetingSchedule>'
```

Listing 7.26: Time zone information is mandatory in SQL Server 2005.

However, this will run without an error in SQL Server 2008.

The second enhancement is about preserving the time zone information. In SQL Server 2005 time zone information is mandatory, but SQL Server does not preserve the time zone information. The *date*, *time* or *dateTime* value is normalized to UTC time.

To understand this, run the following code in SQL Server 2005.

```
-- Drop previous schema collection
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create schema collection
```

```
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Meeting" type="xsd:dateTime" />
</xsd:schema>'
GO

DECLARE  @x XML(ExampleSchema)
SET @x = '<Meeting>2007-11-01T10:30:00+05:30</Meeting>'

SELECT @x
/*
output:
<Meeting>2007-11-01T05:00:00Z</Meeting>
*/
```

Listing 7.27: Time zone information is not preserved in SQL Server 2005.

Note that we assigned a date/time value with time zone information, but SQL Server 2005 normalized it to UTC date/time and did not store the time zone information we provided. There is no way to identify the original time zone value we used while assigning the value.

SQL Server 2008 has fixed this limitation. Let us run the same code in SQL Server 2008.

```
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Meeting" type="xsd:dateTime" />
</xsd:schema>'
GO

DECLARE  @x XML(ExampleSchema)
SET @x = '<Meeting>2007-11-01T10:30:00+05:30</Meeting>'

SELECT @x
/*
output:
<Meeting>2007-11-01T10:30:00+05:30</Meeting>
*/
```

Listing 7.28: SQL Server 2008 preserves time zone information.

Note that the time zone information is preserved in the XML value. This is an important enhancement added in SQL Server 2008.

Data Type: time

XSD *time* data type represents a time value which contains hour, minute, second, fraction and time zone. It would be easier to say that the *time* data type represents everything after the *'T'* of a *dateTime* data type.

```
--Create an XML SCHEMA collection for "time" type
CREATE XML SCHEMA COLLECTION timeDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ArrivalTime" type="xsd:time" />
</xsd:schema>'
GO
```

Listing 7.29: A schema with a "time" element.

```
DECLARE  @x XML(timeDemo)

-- We do not have a fraction of second in this example
SET @x = '<ArrivalTime>14:15:00Z</ArrivalTime>'

-- here we have added the fraction of seconds
SET @x = '<ArrivalTime>14:15:00.00Z</ArrivalTime>'

-- You can make the 'fraction of second' incredibly long :-)
SET @x = '<ArrivalTime>
                 14:15:00.0000000000000000000000000000Z
          </ArrivalTime>'
```

Listing 7.30: "time" data type example.

The time zone rules of *dateTime* data type are applicable for *time* data type, too.

```
DECLARE  @x XML(timeDemo)
-- the following two values are equal
-- representation in UTC
SET @x = '<ArrivalTime>13:30:00Z</ArrivalTime>'
-- representation in local time zone (IST)
SET @x = '<ArrivalTime>19:00:00+05:30</ArrivalTime>'
```

Listing 7.31: Time zone information is mandatory in SQL Server 2005.

Hour can be 24 if minutes and seconds are 0. When hour is 24, it represents the first second of the next day.

```
DECLARE  @x XML(timeDemo)

-- hour can be 24 if the minutes and seconds are 0
-- it represents the first second of the next day
SET @x = '<ArrivalTime>24:00:00Z</ArrivalTime>'
```

```
-- the following is illegal
--SET @x = '<ArrivalTime>24:00:00.001Z</ArrivalTime>'
```

Listing 7.32: "hour" part of a "time" data type can accept 24 if "minute" and "second" part are zero.

Enhancements in SQL Server 2008

The enhancements we discussed for the *dateTime* data type are applicable to the *time* data type, too. SQL Server 2008 has made time zone information optional. If time zone information is present, it is preserved. We have seen details of these enhancements and examples when we discussed *dateTime* data type.

Data Type: date

XSD *date* data type represents a date value which contains *year, month, date* and optional *time zone* information.

Time Zone information is optional per the XSD specification. However, SQL Server 2005's implementation of XSD *date* data type requires the time zone information along with the date value. The validation will fail if the time zone information is not present in the given date value. SQL Server 2008 has made time zone information optional.

The *date* data type is very close to the *dateTime* data type that we examined earlier. If you strip it of the *time* part from the *dateTime* data type, it will result in the *date* data type.

```
--Create an XML SCHEMA collection for "date" type
CREATE XML SCHEMA COLLECTION dateDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="DueDate" type="xsd:date" />
</xsd:schema>'
GO
```

Listing 7.33: A schema with a "date" element.

```
DECLARE  @x XML(dateDemo)

-- in UTC format
SET @x = '<DueDate>2007-11-01Z</DueDate>'
```

```
-- using time difference
SET @x = '<DueDate>2007-11-01+05:30</DueDate>'
```

Listing 7.34: "date" data type example.

Enhancements in SQL Server 2008

The enhancements we discussed for the *dateTime* data type are applicable to the *date* data type, too. SQL Server 2008 has made time zone information optional. If time zone information is present, it is preserved. We have seen details of these enhancements and examples when we discussed *dateTime* data types.

Data Type: gYearMonth

XSD *gYearMonth* data type represents a specific month in a specific year in the Gregorian calendar. It includes a year value and a month value.

```
--Create an XML SCHEMA collection for "gYearMonth" type
CREATE XML SCHEMA COLLECTION gYearMonthDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SalesPeriod" type="xsd:gYearMonth" />
</xsd:schema>'
GO
```

Listing 7.35: A schema with a "gYearMonth" element.

```
DECLARE  @x XML(gYearMonthDemo)

-- time zone information is optional
SET @x = '<SalesPeriod>2007-11</SalesPeriod>'
SET @x = '<SalesPeriod>2007-11Z</SalesPeriod>'
SET @x = '<SalesPeriod>2007-11+05:30</SalesPeriod>'

-- Refer to BC by making the year negative
SET @x = '<SalesPeriod>-2007-11</SalesPeriod>'
```

Listing 7.36: "gYearMonth" data type example.

Data Type: gYear

XSD *gYear* data type represents a Gregorian calendar year.

```
--Create an XML SCHEMA collection for "gYear" type
CREATE XML SCHEMA COLLECTION gYearDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="YearOfBirth" type="xsd:gYear" />
</xsd:schema>'
GO
```

Listing 7.37: A schema with a "gYear" element.

```
DECLARE  @x XML(gYearDemo)

-- Year of birth
SET @x = '<YearOfBirth>1975</YearOfBirth>'

-- accepts time zone too
SET @x = '<YearOfBirth>1975Z</YearOfBirth>'
SET @x = '<YearOfBirth>1975+05:30</YearOfBirth>'

-- negative values are accepted too.
SET @x = '<YearOfBirth>-0002</YearOfBirth>'

-- maximum and minumum years
SET @x = '<YearOfBirth>-9999</YearOfBirth>'
SET @x = '<YearOfBirth>9999</YearOfBirth>'
```

Listing 7.38: "gYear" data type example.

Data Type: gMonthDay

XSD *gMonthDay* data type represents a month-day pair in a Gregorian calendar year. This data type is good for storing dates which occur every year, a Birthday or Anniversary, for example.

```
--Create an XML SCHEMA collection for "gMonthDay" type
CREATE XML SCHEMA COLLECTION gMonthDayDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Anniversary" type="xsd:gMonthDay" />
</xsd:schema>'
GO
```

Listing 7.39: A schema with a "gMonthDay" element.

A *gMonthDay* value starts with two *"-"* signs followed by two digit month, another *"-"* sign and two digit day of the month. *(--MM-DD)*

```
DECLARE  @x XML(gMonthDayDemo)

-- November 25
SET @x = '<Anniversary>--11-25</Anniversary>'

-- Takes a time zone too
SET @x = '<Anniversary>--11-25Z</Anniversary>'
SET @x = '<Anniversary>--11-25+05:30</Anniversary>'
```

Listing 7.40: "gMonthDay" data type example.

Data Type: gDay

XSD *gDay* data type represents a specific day of the Gregorian calendar year that recurs every month.

```
--Create an XML SCHEMA collection for "gDay" type
CREATE XML SCHEMA COLLECTION gDayDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="WritePayCheck" type="xsd:gDay" />
</xsd:schema>'
GO
```

Listing 7.41: A schema with a "gDay" element.

A *gDay* value starts with three *"-"* signs followed by two digit day-of-the month.

```
DECLARE  @x XML(gDayDemo)

-- 10th of Every Month
SET @x = '<WritePayCheck>---10</WritePayCheck>'

-- Takes a time zone too
SET @x = '<WritePayCheck>---10Z</WritePayCheck>'
SET @x = '<WritePayCheck>---10+05:30</WritePayCheck>'
```

Listing 7.42: "gDay" data type example.

Data Type: gMonth

XSD *gMonth* data type represents a specific month of the Gregorian calendar year.

```
--Create an XML SCHEMA collection for "gMonth" type
CREATE XML SCHEMA COLLECTION gMonthDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ProfitSharing" type="xsd:gMonth" />
</xsd:schema>'
GO
```

Listing 7.43: A schema with a "gMonth" element.

A *gMonth* value starts with two *"-"* signs followed by two-digit month number.

```
DECLARE  @x XML(gMonthDemo)

-- Month of october
SET @x = '<ProfitSharing>--10</ProfitSharing>'

-- Takes a time zone too
SET @x = '<ProfitSharing>--10Z</ProfitSharing>'
SET @x = '<ProfitSharing>--10+05:30</ProfitSharing>'
```

Listing 7.44: "gMonth" data type example.

Data Type: hexBinary

XSD *hexBinary* data type represents HEX encoded data. An XML document cannot store binary data. In order to store binary data into an XML document, it has to be encoded to a set of characters that XML supports. The most popular encoding are *hexBinary encoding* and *base64 Encoding*.

The most common usage of *hexBinary* data type is to store binary data. However, for the purpose of learning, I am presenting an example which encodes a plain ASCII text string to hexBinary.

```
--Create an XML SCHEMA collection for "hexBinary" type
CREATE XML SCHEMA COLLECTION hexBinaryDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SecretMessage" type="xsd:hexBinary" />
</xsd:schema>'
GO
```

Listing 7.45: A schema with a "hexBinary" element.

I encoded the string "This is a secret message" using the online *Text-to-Hex* tool available at http://tools.elitehackers.info/Hex.php.

```
DECLARE  @x XML(hexBinaryDemo)

SET @x = '
<SecretMessage>
54 68 69 73 20 69 73 20 61 20 73 65 63 72 65 74 20 6d 65 73 73 61
67 65
</SecretMessage>'
```

Listing 7.46: "hexBinary" data type example.

Data Type: base64Binary

XSD *base64Binary* data type represents base64 encoded data. The most common usage of *base64Binary* data type is to store binary data.

```
--Create an XML SCHEMA collection for "base64Binary" type
CREATE XML SCHEMA COLLECTION base64BinaryDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SecretMessage" type="xsd:base64Binary" />
</xsd:schema>'
GO
```

Listing 7.47: A schema with a "base64Binary" element.

Let us encode the previous message "*This is a secret message*" to base64 using the online conversion tool available at http://www.motobit.com/util/base64-decoder-encoder.asp

```
DECLARE  @x XML(base64BinaryDemo)

SET @x = '
<SecretMessage>
VGhpcyBpcyBhIHNlY3JldCBtZXNzYWdl
</SecretMessage>'
```

Listing 7.48: "base64Binary" data type example.

Data Type: anyURI

XSD *anyURI* data type an absolute or relative URI

```
--Create an XML SCHEMA collection for "anyURI" type
CREATE XML SCHEMA COLLECTION anyURIDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

194

```
  <xsd:element name="SalesURL" type="xsd:anyURI" />
</xsd:schema>'
GO
```

Listing 7.49: A schema with an "anyURI" element.

```
DECLARE  @x XML(anyURIDemo)

-- A fully qualified URL
SET @x = '<SalesURL>http://www.mysite.com/sales.html</SalesURL>'

-- A relative URL
SET @x = '<SalesURL>sales.html</SalesURL>'

-- Not a URL at all, but still accepted
SET @x = '<SalesURL>do you think this is a URL?</SalesURL>'

-- It is a good practice to encode spaces with "%20"
SET @x = '<SalesURL>sales%20centre.html</SalesURL>'
```

Listing 7.50: "anyURI" data type example.

Note that there is no strict validation on the format of the URI. The schema processor will accept even strings that are not valid URI values.

Data Type: QName

XSD *QName* data type stands for a **Qualified Name** per the XML specification. A Qualified Name usually takes the format of **prefix + ':' + name**. **"xsd:element"** is such a qualified name. It is also valid to ignore the prefix and just use the **name** part. An example of such a Qualified Name is **"attribute."**

Assume that we need to write the schema for an XML document which refers to the elements of another XML document. Now we need to make sure that values should be Qualified XML names. Here is an example.

```
--Create an XML SCHEMA collection for "QName" type
CREATE XML SCHEMA COLLECTION QNameDemo AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="XmlNodeName" type="xsd:QName" />
</xsd:schema>'
GO
```

Listing 7.51: A schema with a "QName" element.

```
DECLARE  @x XML(QNameDemo)

-- valid QName
SET @x = '<XmlNodeName>CustomerName</XmlNodeName>'

-- Double Quotes are not allowed
--SET @x = '<XmlNodeName>Customer"Name</XmlNodeName>'

-- + sign is not allowed
--SET @x = '<XmlNodeName>Customer+Name</XmlNodeName>'

-- spaces are not allowed
--SET @x = '<XmlNodeName>Customer Name</XmlNodeName>'

-- digits are allowed
SET @x = '<XmlNodeName>Customer2Customer</XmlNodeName>'

-- underscores are allowed
SET @x = '<XmlNodeName>Customer_Name</XmlNodeName>'

-- can not start with a number
--SET @x = '<XmlNodeName>1CustomerName</XmlNodeName>'
```

Listing 7.52: "QName" data type example.

QName data type can accept a colonized value (a namespace prefix and a string value separated by a colon) only if there is a valid namespace declaration within the scope where the value is used. For example, the following is invalid.

```
DECLARE @x XML(QNameDemo)
SELECT @x = '<XmlNodeName>cust:CustomerName</XmlNodeName>'
```

Listing 7.52.a

Validation of the above XML instance fails because the namespace prefix *"cust"* is not defined. The following works:

```
DECLARE @x XML(QNameDemo)
SELECT @x = '
<XmlNodeName xmlns:cust="http://www.customer.com">
     cust:CustomerName
</XmlNodeName>'
```

Listing 7.52.b

Data Type: NOTATION

SQL Server does not support the XSD *NOTATION* data type.

LAB2: Write Schema for the Order Processing Application – The *Order* Element

In the previous chapter we did the first lab where we created the schema for the root element. In this lab, let us write the schema for the *Order* element.

The *Order* Element

The definition of the root element, *OrderInfo,* says that it should contain one or more *Order* elements. Each *Order* element should contain complete information about a particular order and should look like the example given below.

```
<Order OrderNumber="20002">
   <OrderDate>2008-01-01Z</OrderDate>
   <DeliveryDate>2008-01-16T09:00:00-08:00</DeliveryDate>
   <Customer />
   <Items />
   <OrderNote>Delivery needed before 8 AM</OrderNote>
   <InvoiceNote>
      Adjust the previous credit note with this invoice
   </InvoiceNote>
   <Discount />
</Order>
```

Listing 7.53: Example of an Order element.

The *Order* element should comply with the following rules:

- The *"Order"* element should have a mandatory attribute named *OrderNumber.* This attribute should not be empty and cannot have more than twenty characters. It can hold any combination of alphas and numerics in any case (lower, upper or mixed case).

- *Order* element can have the following child elements. The child elements should follow the same order as given below.
  - OrderDate
  - DeliveryDate
  - Customer
  - Items
  - OrderNote
  - InvoiceNote
  - Discount
- *OrderDate, DeliveryDate, Customer* and *Items* are mandatory elements. *OrderNote, InvoiceNote* and *Discount* are optional elements. None of the elements can appear more than once under an *Order* element.
- *OrderDate* should be a *date* value and should not contain *time* information. *DeliveryDate* should be a *datetime* value which should contain *date* as well as *time* information.
- *OrderNote* and *InvoiceNote* are optional and if present they can store a text note as long as 500 characters.

Let us first review our understanding of XSD and make sure that we have learned everything required to write the schema needed for this lab. To write the schema needed in this lab, we need to have the following understanding.

- Declaring elements with mandatory attributes (Rule #1)
- Validating the length and format of attribute values (Rule #1)
- Declaring child elements under a given element and controlling the order and occurrence of the child elements (Rule #2)
- Declaring optional and mandatory elements (Rule #3)
- Using *date* and *dateTime* data types (Rule #4)
- Validating the length of the text stored in an element (Rule #5)

Let us see each of the requirements in detail. Some of the learning requirements listed above are already discussed in the previous chapters.

Declaring elements with mandatory attributes

We have seen attribute declarations in Chapter 6. An attribute can be declared as mandatory by setting the *"use"* attribute to *"required."* Refer to Chapter 6 for an explanation of the *"use"* attribute.

Validating the length and format of attribute values

The length of the attribute values can be validated by declaring restrictions. A restriction is declared using *"xsd:restriction"* element. We have seen this in the previous lab, where we restricted the length of the *AgencyCode* element.

Declaring child elements under a given element and controlling the order and occurrence of the child elements

Only a *complexType* can have child elements. We have seen this in the previous lab. Chapter 10 covers Complex Types in detail. In the previous lab we created a Complex Type, *OrderInfo,* which holds one or more *Order* elements. The order of elements is significant in XML. A Schema can define the order of the elements using *order indicators*. We have seen an example in the previous lab. *Order indicators* are explained in Chapter 10.

The occurrence of child elements can be controlled by using *minOccurs* and *maxOccurs* occurrence indicators. We have seen an example in the previous lab. A more detailed explanation can be found in Chapter 10.

Declaring optional and mandatory elements

Elements are mandatory by default. An element can be set to mandatory by setting the *minOccurs* attribute to 1 or a higher value. An element can be declared as optional by setting the *minOccurs* attribute to 0. Occurrence indicators are explained in Chapter 5.

Using *date* and *dateTime* data types

XSD has a number of built-in data types that store date and time information. SQL Server does not have an equivalent data type to represent many of those data types. For example, SQL Server does not have a data type that can represent XSD *duration, gDay, gMonthDay, gYear,* etc.

XSD *date, time* and *dateTime* data types map to the DATETIME data type in SQL Server 2005. SQL Server 2008 introduced DATE and TIME data types that can map to XSD *time* and *date* data types.

There are a few points to note while using the following XSD date/time data types:

- *date*
- *time*
- *dateTime*

The *date* data type stores a date value which contains year, month, day and time zone information. The *time* data type represents a time value which contains hour, minute, second and time zone information. The *dateTime* data type contains the information stored by both *date* and *time* data types. We have examined these data types earlier in this chapter.

Note that under SQL Server 2005, all three data types mentioned above take time zone information along with the values. SQL Server 2005's implementation of XSD *date, time* and *dateTime* data types requires time zone information along with the value. If the time zone information is missing, the validation will fail. SQL Server 2008 has relaxed this requirement. In SQL Server 2008, time zone information is optional.

```
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'DataTypeTest'
) BEGIN
      DROP XML SCHEMA COLLECTION DataTypeTest
END
GO

CREATE XML SCHEMA COLLECTION DataTypeTest AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="DateOfBirth" type="xsd:date"/>
        <xsd:element name="ReportingTime" type="xsd:time"/>
        <xsd:element name="Arrival" type="xsd:dateTime"/>
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 7.54

Here is a valid XML instance which passes the validations rules defined by the above schema.

```
DECLARE @x XML(DataTypeTest)
SET @x = '
<Employee>
  <DateOfBirth>1980-01-01Z</DateOfBirth>
```

200

```
    <ReportingTime>10:00:00Z</ReportingTime>
    <Arrival>2008-03-19T13:10:00Z</Arrival>
</Employee>'
```

Listing 7.55: date, time and dateTime values should contain time zone information.

As mentioned earlier, in SQL Server 2005 a *date, time* or *dateTime* value should contain the time zone information, also. The usage of *"z"* at the end of the value indicates a UTC date/time. It is also possible to specify the time zone by means of the time difference from GMT. Here is an example:

```
DECLARE @x XML(DataTypeTest)
SET @x = '
<Employee>
   <DateOfBirth>1980-01-01+05:30</DateOfBirth>
   <ReportingTime>10:00:00+05:30</ReportingTime>
   <Arrival>2008-03-19T13:10:00+05:30</Arrival>
</Employee>'
```

Listing 7.56: time zone can be stored as the time difference from GMT, too.

The value *"+05:30"* stands for Indian Standard Time. We have seen a detailed explanation of date/time data types earlier in this chapter.

The following XML instance will be invalid in SQL Server 2005, but will validate successfully under SQL Server 2008.

```
DECLARE @x XML(DataTypeTest)
SET @x = '
<Employee>
   <DateOfBirth>1980-01-01</DateOfBirth>
   <ReportingTime>10:00:00</ReportingTime>
   <Arrival>2008-03-19T13:10:00</Arrival>
</Employee>'
```

Listing 7.57

Validating the length of text stored in an element

The length of a string value can be restricted using the *length, minLength, maxLength* facets. We have seen an example in the previous lab. A more detailed explanation of the facets of data types is given in Chapter 8.

Start writing the schema

We have learned enough to write the schema needed for this lab. Let us start writing each rule needed for this lab. In each lab, we will focus on a specific element and will create it as a stand-along schema. Once the schema declaration for all the elements is created, we can assemble them and build the final schema that contains the declaration for all the elements.

Rule 1:

The "Order" element should have a mandatory attribute named "OrderNumber."

This attribute should not be empty and cannot have more than twenty characters.

It can hold any combination of digits and alphabets in any case (lower, upper or mixed case).

Let us declare the *"Order"* element and then enhance it with the rest of the validation rules. Here is the basic declaration of the *"Order"* element. We have learned element declarations in Chapter 5.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order"/>
</xsd:schema>
```

Listing 7.58

Let us add the *"OrderNumber"* attribute to the *"Order"* element. Only a *complexType* can have an attribute; therefore, we will declare the *Order* element as *complexType.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.59

Next, let us make the *OrderNumber* attribute mandatory. This can be done by setting the *"use"* attribute to *"required."*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.60

Next we need to add a restriction to make sure that the *OrderNumber* attribute is never left empty. Let us add a validation using the *minLength* facet. Before we could apply a restriction on an attribute value, we should declare it as *simpleType.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.61

The next step is to add an *xsd:restriction* element to the *simpleType* declaration. Since *OrderNumber* is a string value, the *"base"* attribute of the restriction element points to *"xsd:string."*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.62

Now let us restrict the length of the string by using *minLength* and *maxLength* facets.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.63

Next we should restrict the format of the value. We should allow only letters and digits. We will do this by using a pattern restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="20"/>
            <xsd:pattern value="[a-zA-Z0-9]"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.64

The above pattern restricts the values to *lower case alphabets a to z, upper case alphabets A to Z and digits 0 to 9.* An alternate way of writing this pattern is as follows:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
          </xsd:restriction>
        </xsd:simpleType>
```

```
        </xsd:attribute>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 7.65

Note that in this version we don't have the *minLength* and *maxLength* elements anymore. The length restriction is specified along with the pattern.

The regular expression language of XSD is explained in Chapter 12.

Rule 2

> ***Order element can have the following child elements and should appear in the same order as given below: OrderDate, DeliveryDate, Customer, Items, OrderNote, InvoiceNote and Discount.***

Next, let us declare the child elements of the *"Order"* node. Rule two says that the elements should appear in a specific order. Hence, we need to define the child elements using *sequence* indicator.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate"/>
        <xsd:element name="DeliveryDate"/>
        <xsd:element name="Customer"/>
        <xsd:element name="Items"/>
        <xsd:element name="OrderNote"/>
        <xsd:element name="InvoiceNote"/>
        <xsd:element name="Discount"/>
      </xsd:sequence>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
```

```
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.66

Note that any attribute declaration should appear after the declaration of elements.

Rule 3

> **OrderDate, DeliveryDate, Customer and Items are mandatory elements. OrderNote, InvoiceNote and Discount are optional elements. None of the elements can appear more than once under an Order element.**

The next rule defines certain elements as mandatory and certain other elements as optional. All elements are mandatory by default. (This is the opposite with the default behavior of attributes. Attributes are optional by default.)

An element can be declared optional by setting the *minOccurs* attribute to 0. The default value of *minOccurs* for element declarations is 1; thus, elements are mandatory by default.

The last part of the rule says that none of the elements can appear more than once in the XML instance. By using *minOccurs* and *maxOccurs* attributes, the number of occurrences of elements can be controlled. By setting *maxOccurs* to 1, we can make sure that the element is not allowed to appear more than once in the XML instance.

The default value of both *maxOccurs* and *maxOccurs* is 1, which indicates that the element should appear EXACTLY once within the parent element. So, if you do not specify *maxOccurs* or *minOccurs*, the schema processor will use the default value of these attributes while validating the element. Therefore, the following two schema definitions are equivalent.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate"/>
        <xsd:element name="DeliveryDate"/>
        <xsd:element name="Customer"/>
        <xsd:element name="Items"/>
      </xsd:sequence>
```

```
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 7.67

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="DeliveryDate"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Customer"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Items"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.68

Since the default value of *minOccurs* and *maxOccurs* is 1, both schema definitions given below are equivalent.

Rule 4

> ***OrderDate should be a date value and should not contain time information. DeliveryDate should be a datetime value which should contain date as well as time information.***

This rule can be implemented by using *"xsd:date"* and *"xsd:dateTime"* data types. We have seen these data types earlier in this chapter.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer"/>
        <xsd:element name="Items"/>
        <xsd:element name="OrderNote" minOccurs="0"/>
        <xsd:element name="InvoiceNote" minOccurs="0"/>
        <xsd:element name="Discount" minOccurs="0"/>
      </xsd:sequence>
```

```
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.69

Now it is the responsibility of the schema processor to make sure that the value in the XML instance is valid for the given data type.

Rule 5

> ***OrderNote and InvoiceNote are optional and if present they can store a text note as long as 500 characters.***

This rule is pretty simple and can be easily written using the restriction code we wrote to validate the length of the *OrderNumber*.

Here is the final version of the schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer"/>
        <xsd:element name="Items"/>
        <xsd:element name="OrderNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="InvoiceNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Discount" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
```

```
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 7.70

Note that we have declared *OrderNote* and *InvoiceNote* as *simpleTypes*. We are trying to apply a restriction on the value; thus, those elements should be defined as *simpleTypes.*

Now, let us create a Schema Collection and validate the XML instance we saw earlier in this lab with the schema.

```
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'Lab2'
) BEGIN
      DROP XML SCHEMA COLLECTION Lab2
END
GO


CREATE XML SCHEMA COLLECTION Lab2 AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer"/>
        <xsd:element name="Items"/>
        <xsd:element name="OrderNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="InvoiceNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Discount" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
```

```
   </xsd:element>
</xsd:schema>'
```

Listing 7.71

Here is the XML instance that validates with the above schema.

```
DECLARE @x XML(Lab2)
SELECT @x = '
<Order OrderNumber="20002">
  <OrderDate>2008-01-01Z</OrderDate>
  <DeliveryDate>2008-01-10T09:00:00-08:00</DeliveryDate>
  <Customer />
  <Items />
  <OrderNote>Delivery needed before 8 AM</OrderNote>
  <InvoiceNote>
    Adjust the previous credit note with this invoice
  </InvoiceNote>
  <Discount />
</Order>'
```

Listing 7.72

Try running different variations of the above XML instance and you will realize that SQL Server will accept only those values that pass all the validations we have defined in the schema so far.

We are done with the *Order* element. We will write schema for the other elements in the coming labs. We will assemble these declarations and build the final schema when we are done with the labs.

Chapter Summary

We examined XSD Primitive data types in this chapter. We discussed some theoretical stuff like value spaces, lexical spaces, lexical representation, lexical mapping, canonical mapping, canonical representation, canonical lexical representation, etc. If they look too complicated, you can ignore them in the first reading.

We saw that data types are very important in XSD. Having a richer set of data types will help make writing schemas better and more efficiently, as well. XSD supports a rich set of data types that can be classified as Primitive data types and Derived data types. Primitive data types are the base data types of XSD. Derived data type derives from the Primitive data types directly or indirectly. XSD allows you to create custom data types for cases where a built-in data type does not help perform a given validation.

We have examined all the primitive data types of XSD, namely: string, boolean, decimal, float, double, duration, dateTime, date, time, gYearMonth, gYear, gMonthDay, gDay, gMonth, hexBinary, base64Binary, anyURI, QName and NOTATION. SQL Server supports all the Primitive data types of XSD except NOTATION.

Finally, we did a hands-on lab, where we wrote the schema needed for the Order element.

CHAPTER 8
SIMPLE TYPES

We have discussed the XSD Primitive Data Types in the previous chapter. When we discussed Primitive data types, I mentioned that there are another set of built-in data types called Derived Data types. Those data types are derived from the Primitive data types, directly or indirectly.

All those data types are Simple Types that derive from the built in data types. Before we examine the Derived Data types, I would like to go over Simple Types and type derivation. We have seen simple types in many of the examples discussed previously, but have never had a detailed discussion.

In this chapter we will discuss the following:

- Simple Types and Complex Types
- Simple Types – Local and global
- Simple Types – Named and anonymous
- Deriving from simple types
- Enhancements to List and Union types added in SQL Server 2008
- Inheritance and restrictions
- Restricting type derivation

After discussing the above we will do a hands-on lab, which is a continuation of the labs we did in the previous chapters.

Simple Types and Complex Types

We have discussed simple types and complex types several times in the previous chapters. The basic distinction between simple types and complex types is that only a complex type can contain child elements and attributes. Simple types can't do it. We will discuss Complex Types in Chapter 10.

Simple types can only store a value. They cannot have child elements. They cannot have attributes. An element or attribute can have a simple type. They are simple, as the name indicates.

Simple Types – Global and Local

Simple Types can be declared *globally* or *locally.* When a *Simple Type* is declared right under the *xsd:schema* element, it is a global declaration. When a simple type is declared globally, it must always have a name.

When a *Simple Type* is declared within the scope of an element or attribute it is a local declaration. Local declaration of simple types cannot take a name. The following code snippet shows examples of simple types, both global and local.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="zipType" />
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.1: Example of Simple Type (Global Declaration).

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode">
    <xsd:simpleType>
      <xsd:restriction base="xsd:integer">
        <xsd:maxInclusive value="99999"/>
        <xsd:minInclusive value="10000"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 8.2: Example of Simple Type (Local Declaration).

Globally declared simple types provide a good example of reusability. These simple types can be used just like other built in types. Global simple types help reuse the definitions as well as help organize and maintain the schema.

This is particularly helpful when the same set of validations is to be performed on more than one element or attribute. For example, think of the case where we need to validate the zip code of Shipping and Billing location. Here is an example.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Sales invoice element-->
  <xsd:element name="SalesInvoice">
    <xsd:complexType>
```

```
      <xsd:attribute name="BillingZip" type="zipType"/>
      <xsd:attribute name="ShippingZip" type="zipType"/>
    </xsd:complexType>
  </xsd:element>

  <!-- zipType -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.3: An example showing a global simple type declaration.

The above schema would have been more complex if we had declared the *Simple Type* locally. If we need to go with a local declaration, we need to write the same validation rules twice. Here is the version of the schema that uses local simple type declarations to validate zip codes.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Sales invoice element-->
  <xsd:element name="SalesInvoice">
    <xsd:complexType>
      <xsd:attribute name="BillingZip">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer">
            <xsd:maxInclusive value="99999"/>
            <xsd:minInclusive value="10000"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="ShippingZip">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer">
            <xsd:maxInclusive value="99999"/>
            <xsd:minInclusive value="10000"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.4: An example showing a local simple type declaration.

Simple Types – Named and Anonymous

When Simple Types are declared globally, they must be named. Such a named simple type can be reused at other parts of the schema. When simple types are declared locally, they cannot have a name; hence, they are called *Anonymous.*

All global declarations of simple types are *named* and all local declarations of Simple types are *anonymous.*

Deriving from Simple Types

You can derive a new Simple Type from one of the built-in data types or from another simple type which is already derived from a built-in type. A new *Simple Type* can be derived from a base type in one of the following ways.

* Derive by restriction
* Derive by list
* Derive by Union

Let us see each of these methods in detail.

Deriving by Restriction

Usually new *Simple Types* are created to perform additional validations or restrictions that an existing type does not do. This involves identifying a base type (built-in or user-defined) that is close to what we are looking for, and adding the additional restrictions or validation rules.

For example, assume that we need to create a new *Simple Type* to validate the Age of employees. We need to make sure that the age should be between 18 and 65 and no decimals allowed. We could use *xsd:integer* and add a restriction on the minimum and maximum allowed values. We could also use *xsd:decimal* and then add restrictions on min/max values and set decimals to 0. In this example, a better option is to go with *xsd:integer* because it already has a restriction on the decimals.

Let us now understand the restrictions in detail.

Assume that we need to create the schema for a sales invoice. One of the values that we need to store is the zip code. Zip code should be a number and should have *exactly* five digits. Let us create a *Simple Type* to describe and validate a zip code.

A basic declaration of a *Simple Type* would look like this:

```
<xsd:simpleType/>
```

Listing 8.5: Basic declaration of a simple type (empty).

However, this declaration is incomplete. A global declaration of a *Simple Type* should always have a name. Let us name the type *"zipType."*

```
<xsd:simpleType name="zipType"/>
```

Listing 8.6: A global simple type declaration should always be named.

A restriction is defined by adding *xsd:restriction* to the *Simple Type* declaration.

```
<xsd:simpleType name="zipType">
   <xsd:restriction />
</xsd:simpleType>
```

Listing 8.7: Adding a restriction to a simple type.

The next step is to specify the parent type from which we need to derive our new *Type*. We could either go with *xsd:string* or *xsd:integer*. If we go with *xsd:string*, we need to add restrictions so that only digits are accepted. On the other hand, if we go with *xsd:integer* we don't need this validation. So let us go with *xsd:integer.*

```
<xsd:simpleType name="zipType">
   <xsd:restriction base="xsd:integer"/>
</xsd:simpleType>
```

Listing 8.8: Deriving from a base type.

Next, we need to restrict the length of the value to five digits. Each data type has a number of properties that restricts the set of values it can accept. These properties are called *facets*. While writing a schema, we could use these facets to restrict the values an element or attribute can accept. When you *derive* a new data type *by restriction*, you restrict one or more facets.

> The facets of XSD built-in data types are explained in Chapter 9.

All numeric data types have a facet called *totalDigits.* Let us use the *totalDigits* facet to restrict the number of digits to five. Here is the schema that applies this restriction.

```
<xsd:simpleType name="zipType">
  <xsd:restriction base="xsd:integer">
    <xsd:totalDigits value="5"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.9: Restricting the number of digits with "totalDigits" facet.

Well, the *totalDigits* facet will make sure that the *Simple Type* does not allow more than five digits. In our case, we need one more validation to make sure that the values have at least five digits.

This can be achieved in a number of ways. Two examples are given below.

```
<xsd:simpleType name="zipType">
  <xsd:restriction base="xsd:integer">
    <xsd:totalDigits value="5"/>
    <xsd:minInclusive value="10000"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.10: "totalDigits" restricts the number of digits in the value and "minInclusive" defines the smallest value accepted by the type.

```
<xsd:simpleType name="zipType">
  <xsd:restriction base="xsd:integer">
    <xsd:maxInclusive value="99999"/>
    <xsd:minInclusive value="10000"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.11: "minInclusive" and "maxInclusive" can be used together to restrict the values to a given range.

Both examples given above restrict the zip code to exactly five digits.

In the case of Zip codes we do not accept leading zeroes. If we need to accept leading zeroes, the above validation will not work. In such a case, we need to use a *pattern* restriction. *Pattern* restriction uses a Regular Expression pattern to validate the value. The Regular Expression language supported by XSD is explained in Chapter 12.

We have created a Simple Type named *zipType.* Now let us declare an element that is bound to *zipType*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="zipType" />
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.12: Declaring an element of simple type: zipType.

Note that we have set the *type* attribute to *"zipType"* to indicate that the element *ZipCode* should be validated using the rules defined for *zipType.*

Now let us create a schema collection and see if SQL Server validates the new data type correctly.

```
CREATE XML SCHEMA COLLECTION RestrictionDemo
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="zipType" />
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
```

Listing 8.13: Schema collection that validates a zip code.

```
DECLARE @x XML(RestrictionDemo)
SET @x = '<ZipCode>12345</ZipCode>'

-- the following will fail
-- SET @x = '<ZipCode>1234</ZipCode>'
-- SET @x = '<ZipCode>123456</ZipCode>'
-- SET @x = '<ZipCode>00001</ZipCode>'
-- SET @x = '<ZipCode>abcde</ZipCode>'
```

Listing 8.14: TSQL code validating a zip code.

Deriving by List

Another way of creating a new *Simple Type* is by deriving by List. When deriving by list, the new data type can store a SPACE separated list of values accepted by the base type. You can define additional restrictions if needed. Space is the only delimiter allowed in a list. Hence, if a value contains spaces as part of it, it will be interpreted as two separate values.

Note that the base type of a *Simple Type* derived by list cannot be a *List type.* In other words, a list of list is not supported.

Here is an example of a *Simple Type* that derives by list from *xsd:integer.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCodes" type="ZipCodeList" />
  <xsd:simpleType name="ZipCodeList">
    <xsd:list itemType="xsd:integer" />
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.15: Deriving by list.

The above schema defines an element named *ZipCodes.* The data type of the element is *ZipCodeList. ZipCodeList* is a *Simple Type* that derives from *xsd:integer* by *list*.

```
<xsd:simpleType name="ZipCodeList">
    <xsd:list itemType="xsd:integer" />
</xsd:simpleType>
```

Listing 8.16: A list type that derives from "integer."

The above example shows the definition of a Simple Type that derives from *xsd:integer* by *list.* When deriving by list, the *itemType* attribute of *xsd:list* should is set to the base type.

We have not seen xsd:integer when we discussed Primitive Data Types. This is because xsd:integer is a Simple Type that derives from xsd:decimal by restriction. We will examine the XSD Derived Data Types in Chapter 9.

Let us create a Schema Collection with this definition and see how it works in SQL Server.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCodes" type="ZipCodeList" />
  <xsd:simpleType name="ZipCodeList">
    <xsd:list itemType="xsd:integer" />
  </xsd:simpleType>
</xsd:schema>
'
GO
```

Listing 8.17: Creating schema collection to validate zip codes.

```
-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<ZipCodes>10001 10002 10003 1 205 409</ZipCodes>'
SET @x = '<ZipCodes></ZipCodes>'
SET @x = '<ZipCodes />'

-- The following will fail because the list is of integer
-- data type.
--SET @x = '<ZipCodes>1001 100A</ZipCodes>'
--SET @x = '<ZipCodes>10001 1001.0</ZipCodes>'
```

Listing 8.18: Validating zip codes.

Note that SQL Server allows empty elements as well as a list of integer values. However, it will not allow anything other than digits. It does not allow decimals because the list is defined as integer type.

Note also that we are still able to store values like 1, 205, etc., which are not valid ZIP codes. This is because SQL Server does not know what a zip code is and, thus, will accept values as long as they are integers. One way to get this fixed is by deriving a new type from integer by restriction and adding validations so that it will accept only integer values with five digits.

Let us assume that we need to write the schema for an element that accepts a list of zip codes in Washington State. The zip codes in Washington State range from 98000 to 99499. Let us do it in a two-step process. First, let us create a simple type that validates the zip code. Then we will create another type that derives by list from it.

Here is the definition of the new type that validates zip codes in Washington State.

```
<!-- Definition of WashingtonZipCode -->
<xsd:simpleType name="WashingtonZipCode">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="98000"/>
    <xsd:maxInclusive value="99499"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.19: Simple Type that validates the zip codes in Washington State.

Note that we have applied a restriction on the accepted range of values using *minInclusive* and *maxInclusive.* Now let us create the list type.

```
<!-- Define the list type -->
<xsd:simpleType name="ZipCodeList">
  <xsd:list itemType="WashingtonZipCode" />
</xsd:simpleType>
```

Listing 8.20: Creating a list type from "WashingtonZipCode."

Now let us create a schema collection and test it.

```
CREATE XML SCHEMA COLLECTION ZipCodeDemo
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- element declaration -->
  <xsd:element name="ZipCodes" type="ZipCodeList" />

  <!-- Define the list type -->
  <xsd:simpleType name="ZipCodeList">
    <xsd:list itemType="WashingtonZipCode" />
  </xsd:simpleType>

  <!-- Definition of ZipCode -->
  <xsd:simpleType name="WashingtonZipCode">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="98000"/>

      <xsd:maxInclusive value="99499"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
```

Listing 8.21: A schema collection that validates the zip codes of Washington State.

The new definition does not allow the list to store those incorrect values that the previous example allowed. SQL Server will accept values only within the range of 98000 to 99499.

```
DECLARE @x XML(ZipCodeDemo)

-- the following will fail, because the zip codes
-- are out-of-range
-- SET @x = '<ZipCodes>10001 10002 10003</ZipCodes>'

-- this is the correct version
SET @x = '<ZipCodes>98000 98001 98002</ZipCodes>'
```

Listing 8.22: Validating zip codes with schema collection: "ZipCodeDemo."

> **SQL Server 2005 does not allow creating a *list* of *union* types. Support for creating *list* of *union* types is added in SQL Server 2008 and is explained later in this chapter.**

Deriving by Union

We have just seen derivation by restriction and by list. A third method of deriving a simple type is by union. As the name indicates, you can derive a new type from the union of one or more base types. The derived type can store the values acceptable to any of the base types from which the new type is derived.

Assume that we need to write a schema for the XML instance given below.

```
<Locations>
  <Location>NY</Location>
  <Location>90002</Location>
  <Location>NJ</Location>
</Locations>
```

Listing 8.23: An XML instance storing location data.

The *Location* element in the above example accepts a zip code or a two-letter city code. If the value is a zip code, then a set of validations defined for zip codes should be performed on the value. If the value is a city code, then another set of validations need to be performed on the value.

Let us start writing this schema. First of all let us create two simple types; one to validate city codes and the other to validate zip codes. Here is the schema that validates city codes.

```
<xsd:simpleType name="CityType">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="NY"/>
    <xsd:enumeration value="NJ"/>
```

```
    <xsd:enumeration value="WA"/>
    <!-- other cities here -->
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.24: A simple type using enumeration to restrict values.

Let us now create another simple type to validate zip codes.

```
<xsd:simpleType name="ZipType">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="10000"/>
    <xsd:maxInclusive value="99999"/>
  </xsd:restriction>
</xsd:simpleType>
```

Listing 8.25: A simple type using range restrictions to validate zip codes.

Now, let us create another simple type that derives from *ZipType* and *CityType* by union.

```
<!-- Define the ZipCityUnion -->
<xsd:simpleType name="ZipCityUnion">
  <xsd:union>
    <xsd:simpleType>
      <xsd:restriction base="ZipType"/>
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="CityType"/>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Listing 8.26: A union type that derives from CityType and ZipType.

We have created the components to build a schema that validates the XML instance we discussed earlier. Let us now build the schema.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- element declaration -->
  <xsd:element name="Locations">
```

```
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Location" maxOccurs="unbounded"
                     type="ZipCityUnion"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <!-- Define the ZipCityUnion -->
  <xsd:simpleType name="ZipCityUnion">
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="ZipType"/>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="CityType"/>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>

  <!-- CityType -->
  <xsd:simpleType name="CityType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NY"/>
      <xsd:enumeration value="NJ"/>
      <xsd:enumeration value="WA"/>
      <!-- other cities here -->
    </xsd:restriction>
  </xsd:simpleType>

  <!-- ZipType -->
  <xsd:simpleType name="ZipType">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="10000"/>
      <xsd:maxInclusive value="99999"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
GO
```

Listing 8.27: Creating a schema collection to validate locations.

```
-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '
<Locations>
  <Location>NY</Location>
  <Location>90002</Location>
  <Location>NJ</Location>
</Locations>'
```

Listing 8.28: The schema processor accepts zip codes as well as city codes.

Note that the schema allows the element to store either a zip code or a city code. Each value will be validated against the rules defined in the base types until it passes the validation of one of the base types. The value will be accepted only if it validates successfully with one of the base types.

224

> It is legal to have a *union of unions* or a *union of union of unions*.

> SQL Server 2005 does not allow creating a *union of list* types. Support for creating *union of list* types is added in SQL Server 2008 and is explained later in this chapter.

Enhancements to List and Union Types added in SQL Server 2008

SQL Server 2008 added a few enhancements to some of the XSD data types. We saw some of those new enhancements when we discussed the date/time data types. SQL Server 2008 adds the following features to List and Union types.

- **You can create List types that contain Union types**
- **You can create Union types that contain List types**

Creating a List of Union Types

Earlier in this chapter we saw examples of Union types. We saw an example schema which defined a *ZipCityUnion* type that accepted either a Zip Code or a two letter city code.

Let us have a look at another XML document.

```
<Search>
  <Find what="Apartment" where="NY"/>
  <Find what="Hotel" where="98000"/>
</Search>
```

Listing 8.29: XML document containing input value for a search application.

This is the input that a search application takes. The search application tries to find the information requested by the *"what"* attribute in the location specified by the *"where"* attribute. The *"where"* attribute can take either a zip code or a two letter city code.

We could easily write a schema for this XML document based on what we learned earlier in this chapter. Here is the schema.

8 – Simple types

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- element declaration -->
  <xsd:element name="Search">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Find">
          <xsd:complexType>
            <xsd:attribute name="what" type="xsd:string"/>
            <xsd:attribute name="where" type="ZipCityUnion"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <!-- Define the union type -->
  <xsd:simpleType name="ZipCityUnion">
    <xsd:union>
      <!-- Definition of Zip Code -->
      <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
          <xsd:minInclusive value="98000"/>
          <xsd:maxInclusive value="99499"/>
        </xsd:restriction>
      </xsd:simpleType>
      <!-- definition of City Code-->
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:length value="2"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Search>
  <Find what="Apartment" where="NY"/>
  <Find what="Hotel" where="98000"/>
</Search>'
```

Listing 8.30: A schema with a union type that accepts a zip code or city code.

226

In the example we saw above, the *"where"* attribute could take either a zip code or a two-letter city code. However, you can specify only one location at a time. Assume that the requirement changes and we need to allow multiple locations in the *"where"* attribute, such as the example given below.

```
<Search>
   <Find what="Hotel" where="NY NJ 98003"/>
</Search>
```

Listing 8.31: A new version of the XML document that takes a list of zip codes and city codes.

We learned that we can use a *list type* to store a list of values. In this specific case we need to create a list of a union type, because we should allow zip codes as well as city codes.

Let us first create a type that validates zip codes.

```
<!-- Zip Type -->
<xsd:simpleType name="ZipType">
   <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="10000"/>
      <xsd:maxInclusive value="99999"/>
   </xsd:restriction>
</xsd:simpleType>
```

Listing 8.32: A simple type to validate zip codes.

Now let us create another type to validate city codes.

```
<!-- City Type -->
<xsd:simpleType name="CityType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NY"/>
      <xsd:enumeration value="NJ"/>
      <xsd:enumeration value="WA"/>
      <!-- other cities here -->
   </xsd:restriction>
</xsd:simpleType>
```

Listing 8.33: A simple type to validate city codes.

Next, we write a union type that accepts either a *ZipType* or a *CityType*.

```
<!-- Define the union type -->
<xsd:simpleType name="ZipCityUnion">
   <xsd:union>
      <xsd:simpleType>
         <xsd:restriction base="ZipType"/>
```

```
    </xsd:simpleType>
    <xsd:simpleType>
      <xsd:restriction base="CityType"/>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Listing 8.34: A type that derives from ZipType and CityType by union.

Now, let us create a list from *ZipCityUnion* type we just created.

```
<!-- Define list of ZipCityUnion -->
<xsd:simpleType name="ZipCityUnionList">
  <xsd:list itemType="ZipCityUnion"/>
</xsd:simpleType>
```

Listing 8.35: A list type that derives from a union type.

Now let us write the element declaration.

```
<!-- element declaration -->
<xsd:element name="Search">
  <xsd:complexType>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="Find">
        <xsd:complexType>
          <xsd:attribute name="what" type="xsd:string"/>
          <xsd:attribute name="where" type="ZipCityUnionList"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing 8.36: A schema that uses a list of union type.

Let us try to create a Schema Collection in SQL Server 2005 and see what happens.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- element declaration -->
  <xsd:element name="Search">
```

```xml
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Find">
          <xsd:complexType>
            <xsd:attribute name="what" type="xsd:string"/>
            <xsd:attribute name="where" type="ZipCityUnionList"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <!-- Define list of ZipCityUnion -->
  <xsd:simpleType name="ZipCityUnionList">
    <xsd:list itemType="ZipCityUnion"/>
  </xsd:simpleType>

  <!-- Define the union type -->
  <xsd:simpleType name="ZipCityUnion">
    <xsd:union>
      <xsd:simpleType>
        <xsd:restriction base="ZipType"/>
      </xsd:simpleType>
      <xsd:simpleType>
        <xsd:restriction base="CityType"/>
      </xsd:simpleType>
    </xsd:union>
  </xsd:simpleType>

  <!-- Zip Type -->
  <xsd:simpleType name="ZipType">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="10000"/>
      <xsd:maxInclusive value="99999"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- City Type -->
  <xsd:simpleType name="CityType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NY"/>
      <xsd:enumeration value="NJ"/>
      <xsd:enumeration value="WA"/>
      <!-- other cities here -->
    </xsd:restriction>
  </xsd:simpleType>

</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Search>
  <Find what="Hotel" where="NY NJ 98003"/>
</Search>'
```

Listing 8.37: SQL Server 2005 does not support lists of union types.

Unfortunately, this will not run in SQL Server 2005. The XSD implementation of SQL Server 2005 does not support *lists* of *union* types. If you run this code, SQL Server 2005 will generate the following error.

```
Msg 6978, Level 16, State 1, Line 2
Invalid item type for list type 'ZipCityUnionList'. The item type
of a list may not itself be a list, and union types and types
derived from ID may not be used as item types in this release.
```

This restriction has been removed in SQL Server 2008. You can successfully run this code in SQL Server 2008.

Creating a Union of List Types

We just saw an example that creates a list of union types. Now let us see another example that creates a union of list types. To understand this, let us take a slightly different version of the XML document we examined earlier.

Assume that there is a change in the way the search application processes input parameters. This change requires that the XML document should contain either a list of *zip codes* or a list of *city codes* in the *where* attribute.

```
<Search>
  <Find what="Hotel" where="NY NJ WA"/>
  <Find what="Apartment" where="98000 98002 98010"/>
</Search>
```

Listing 8.38: A new version of the XML document that requires a union of list types.

Let us re-write the schema for this XML structure. The *where* attribute should accept either a *list* of city codes or a list of *zip* codes. To achieve this we should create a union type that accepts a *list of zip codes* as well as a *list of city codes*.

In the previous section, we have created types to validate zip codes as well as city codes. Now let us create a union type that contains lists of *ZipType* and *CityType*.

```
<!-- Define the union type -->
<xsd:simpleType name="ZipCityListUnion">
  <xsd:union>
    <!-- Definition of Zip Code List -->
    <xsd:simpleType>
      <xsd:list itemType="ZipType"/>
    </xsd:simpleType>
```

```
    <!-- definition of City Code List-->
    <xsd:simpleType>
      <xsd:list itemType="CityType"/>
    </xsd:simpleType>
  </xsd:union>
</xsd:simpleType>
```

Listing 8.39: A union of list types.

Now, let us write the element declaration using the *ZipCityListUnion* type we just created.

```
<!-- element declaration -->
<xsd:element name="Search">
  <xsd:complexType>
    <xsd:sequence maxOccurs="unbounded">
      <xsd:element name="Find">
        <xsd:complexType>
          <xsd:attribute name="what" type="xsd:string"/>
          <xsd:attribute name="where" type="ZipCityListUnion"/>
        </xsd:complexType>
      </xsd:element>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

Listing 8.40: element declaration using a union of list types.

The schema is ready! Let us test it in SQL Server 2005.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- element declaration -->
  <xsd:element name="Search">
    <xsd:complexType>
      <xsd:sequence maxOccurs="unbounded">
        <xsd:element name="Find">
          <xsd:complexType>
            <xsd:attribute name="what" type="xsd:string"/>
            <xsd:attribute name="where" type="ZipCityListUnion"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

```
    <!-- Define the union type -->
    <xsd:simpleType name="ZipCityListUnion">
      <xsd:union>
        <!-- Definition of Zip Code List -->
        <xsd:simpleType>
          <xsd:list itemType="ZipType"/>
        </xsd:simpleType>
        <!-- definition of City Code List-->
        <xsd:simpleType>
          <xsd:list itemType="CityType"/>
        </xsd:simpleType>
      </xsd:union>
    </xsd:simpleType>

    <!-- Zip Type -->
    <xsd:simpleType name="ZipType">
      <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="10000"/>
        <xsd:maxInclusive value="99999"/>
      </xsd:restriction>
    </xsd:simpleType>

    <!-- City Type -->
    <xsd:simpleType name="CityType">
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="NY"/>
        <xsd:enumeration value="NJ"/>
        <xsd:enumeration value="WA"/>
        <!-- other cities here -->
      </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Search>
  <Find what="Hotel" where="NY NJ WA"/>
  <Find what="Apartment" where="98000 98002 98010"/>
</Search>'
```

Listing 8.41: SQL Server 2005 does not support union of list types.

This will not run in SQL Server 2005. If you try to run it, SQL Server 2005 will generate the following error.

```
Msg 6977, Level 16, State 1, Line 3
Invalid member type 'xs-
nun(/simpleType(ZipCityListUnion)/simpleType()[1])' in union type
'ZipCityListUnion'. Unions may not have complex member types.
```

The XSD implementation of SQL Server 2005 does not allow union of list types. The restriction has been removed in SQL Server 2008. You can run this code successfully in SQL Server 2008.

Inheritance and restrictions

We have seen how to inherit new *Simple Types* from existing ones. When you derive a new type from an existing one, you can **only** apply a restriction which is equal to or more restrictive than the base type.

For example, *xsd:integer* derives from *xsd:decimal* by applying a restriction on the fraction digits. *xsd:integer* sets *fractionDigits* facet to 0. If you derive a new *Simple Type* from *xsd:integer*, you cannot set the *fractionDigits* facet to a value greater than 0.

Let us look at an example. Let us go back to the *zipType* we defined earlier in this chapter.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.42: A simple type that validates zip codes.

We defined the above *Simple Type* to validate zip codes. It will accept any value between 10000 and 99999. Now assume that we are writing this schema to validate the sales invoice of a company that ships only to the state of Washington. So the Zip code in the shipping address should be between 98000 and 99499.

Let us define one more *Simple Type* to validate the Zip codes of Washington. We will derive a new type from *zipType*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Zip codes of Washington-->
  <xsd:simpleType name="washingtonZipType">
    <xsd:restriction base="zipType">
      <xsd:minInclusive value="98000"/>
      <xsd:maxInclusive value="99499"/>
    </xsd:restriction>
```

```
    </xsd:simpleType>
</xsd:schema>
```

Listing 8.43: A simple type that validates the zip codes of Washington.

The above example tries to derive a new *Simple Type* named *washingtonZipType* from *zipType*. The derived type makes the value space of the base type more restrictive.

It is not allowed to make the value space of a derived type less restrictive than the base type. The following is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Zip codes of Washington-->
  <xsd:simpleType name="washingtonZipType">
    <xsd:restriction base="xsd:integer">
      <xsd:minInclusive value="98000"/>
      <xsd:maxInclusive value="99499"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- any zip type -->
  <xsd:simpleType name="anyZipType">
    <xsd:restriction base="washingtonZipType">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.44: It is illegal to have the derived type less restrictive than the base type

The above example derives a new type from *washingtonZipCode*. However, it tries to make the value space less restrictive than the base type. This is not allowed and SQL Server will generate an error if you attempt to create a schema collection with such a definition.

Locking facets with "fixed" attribute

We have discussed earlier that each XSD data type has a certain number of facets that control its value space. When we derive a new *Simple Type* from another, the new type will inherit all the facets of the base type. The derived type can then make one or more facets more restrictive.

There may be times when you do not want the derived types to modify a certain facet. In such cases, you can set the *"fixed"* attribute of the given facets to *"true"* to make sure that the derived types do not modify those facets.

Let us go back to the example of *ZipType* that we saw earlier. In the previous example – the derived type – *washingtonZipType* was able to modify *minInclusive* and *maxInclusive* facets. Now let us try to mark *maxInclusive* as a *fixed* facet and see what happens.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="washingtonZipType"/>

  <!-- zipType -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Zip codes of Washington-->
  <xsd:simpleType name="washingtonZipType">
    <xsd:restriction base="zipType">
      <xsd:minInclusive value="98000"/>
      <xsd:maxInclusive value="99499"/>
    </xsd:restriction>  </xsd:simpleType>
</xsd:schema>
```

Listing 8.45: When a facet is marked as "fixed," it cannot be modified in the derived type.

Note that the facet *maxInclusive* is marked as *"fixed,"* and as a result you cannot modify its value in the derived type. If you modify this and try to create a schema collection, SQL Server will generate the following error.

```
Invalid type definition, fixed facets can not be redefined
```

However, you can modify the *minInclusive* facet because it is not a fixed facet in the base type. The following will run successfully.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
```

```
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="washingtonZipType"/>

  <!-- zipType -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Zip codes of Washington-->
  <xsd:simpleType name="washingtonZipType">
    <xsd:restriction base="zipType">
      <xsd:minInclusive value="98000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<ZipCode>98000</ZipCode>'
```

Listing 8.46

Restricting derivation with "final"

Inheritance is a great feature. However, there are times when you want to protect a *Simple Type* from being inherited. XSD provides a way to protect your Simple Type so that no other *Types* can inherit from it. This is achieved by using the *"final"* attribute.

The *"final"* attribute can take the following values:

- *restriction*
- *list*
- *union*
- *extension*
- *#all*

We will examine each of these values in the next few paragraphs.

Preventing derivation by restriction

When the *"final"* attribute is set to *"restriction,"* the Simple Type is protected from being inherited by restriction. Look at the following example:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="washingtonZipType"/>

  <!-- zipType -->
  <xsd:simpleType name="zipType" final="restriction">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" />
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- Zip codes of Washington-->
  <xsd:simpleType name="washingtonZipType">
    <xsd:restriction base="zipType">
      <xsd:minInclusive value="98000"/>
      <xsd:maxInclusive value="99499"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.47: When "final" is set to "restriction," the type cannot be inherited by restriction.

This will generate the following error:

```
Invalid type definition for type 'washingtonZipType', the
derivation was illegal because 'final' attribute was specified on
the base type
```

When the *"final"* attribute is set to *"restriction,"* you can still derive new Simple Types by list or by union.

Preventing derivation by list

When *"final"* is set to *"list,"* you cannot derive a new type by list. However, you can still derive by restriction or by union from the given base type.

The following is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="ZipTypeList"/>

  <!-- zipType -->
  <xsd:simpleType name="zipType" final="list">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ZipTypeList">
    <xsd:list itemType="zipType"/>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.48: When "final" is set to "list," the type cannot be used as the base of a list.

If you try to create a schema collection with the above code, SQL Server will generate the following error.

```
Invalid type definition for type 'ZipTypeList', the derivation was
illegal because 'final' attribute was specified on the base type
```

Preventing derivation by union

When *"final"* is set to *"union,"* the type cannot be used as one of the base types in a union. You can still derive from it by restriction or by list.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ZipCode" type="ZipCityUnion"/>

  <!-- zipType -->
  <xsd:simpleType name="zipType" final="union">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <!-- City Type -->
  <xsd:simpleType name="CityType">
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="NY"/>
      <xsd:enumeration value="NJ"/>
      <xsd:enumeration value="WA"/>
      <!-- other cities here -->
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ZipCityUnion">
    <xsd:union memberTypes="CityType zipType"/>
```

```
    </xsd:simpleType>
</xsd:schema>
```

Listing 8.49: Types having "final" set to "union" cannot be used as one of the base types in a union.

If you try to create a schema collection with the above schema, SQL Server will generate the following error.

```
Invalid type definition for type 'ZipCityUnion', the derivation
was illegal because 'final' attribute was specified on the base
type
```

Preventing derivation by extension

Earlier in this chapter we saw how to derive new *Simple Types* by list, union and restriction. But I have not mentioned derivation by *extension* yet.

Extension refers to deriving a new type from a Simple Type that results in a Complex Type. When you extend a simple type, the result will always be a complex type. That is the reason why I have not mentioned it when we discussed deriving *Simple Types.*

Setting the *"final"* attribute to *"extension"* protects the Simple Type from being extended. We will see *extension* in Chapter 11 when we discuss derivation of complex types.

Preventing more than one type of derivation

The *"final"* attribute can take more than one value at a time to prevent the Simple Type from being inherited by any of the specified derivation methods. For example, the following code prevents the Simple Type from being inherited by list and by union.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType" final="list union">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
```

```
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.50

The next example shows a *Simple Type* that cannot be inherited by union or by restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType" final="restriction union">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.51

The *final* attribute can take a list that contains any combination of *restriction, union, list* and *extension.* However, you cannot specify a derivation method more than once.

Preventing derivation completely

When the *"final"* attribute is set to *"#all,"* the *Simple Type* cannot be inherited at all. We have seen how to protect a Simple Type from being inherited by list, restriction, union or by extension. When *"final"* is set to *"#all"* the *Simple Type* cannot be inherited at all.

Having *"final"* set to *"#all"* is equivalent to setting it to *list, union, restriction* and *extension.* For example, the following two are equivalent.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType"
                  final="restriction union list extension">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.52

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType" final="#all">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.53

The following is illegal because *"#all"* cannot be used with any other derivation method.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- zipType -->
  <xsd:simpleType name="zipType" final="restriction #all">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999" fixed="true"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 8.54

LAB3: Write schema for the Order Processing Application –
The *Customer* element.

It is time for another Lab. In the previous labs we have developed the schema for the *OrderInfo* and *Order* elements. In this lab we will write the schema for the *customer* element.

The Customer Element

This is how the *Customer* element should look.

```xml
<Customer CustomerNumber="LAZYK">
  <CustomerName>Lazy K Kountry Store</CustomerName>
  <Billing />
  <Shipping />
```

```
    <Terms>30 Days Credit</Terms>
    <Contact />
</Customer>
```

Listing 8.55: Example of a Customer element.

The *Customer* element should follow the rules given below:

- Each *Customer* element should contain an attribute named *CustomerNumber.* It is mandatory and should be EXACTLY five characters long. It should contain only upper case letters (A-Z).
- A *Customer* element can have five child elements EXACTLY in the order given below.
  - CustomerName
  - Billing
  - Shipping
  - Terms
  - Contact
- *CustomerName* is optional, and if present should not be more than fifty characters long. The element can appear only once.
- *Billing* and *Contact* are mandatory and can appear only once.
- *Shipping* is optional. If not present, the address given in *Billing* is assumed to be the shipping location. There should be only one *shipping* element.
- *Terms* is mandatory and cannot appear more than once. The value should be one of the following:
  - 30 Days Credit
  - 60 Days Credit
  - 90 Days Credit
  - Against Delivery

Before we jump into writing the schema, let us see if we have learned enough to translate all the above rules to XSD declarations. Writing the above schema requires the following XSD skills.

- Declaring mandatory elements and validating the length and format of attribute values.
- Declaring the children of an element and controlling the order of the child elements*.*
- Declaring optional and mandatory elements.
- Controlling occurrences of elements.
- Defining an enumeration.

We have developed most of these skills in the previous chapters and have used them in the previous labs. The only skill that is new to you must be the enumeration that we need to define for the *Terms* element.

Defining Enumerations

Sometimes we will come across requirements where we need to apply a certain restriction to an element or attribute so that only a set of predefined values can be stored. For example, the *PaymentMethod* element of *Invoice* may allow only *Cash, Check* or *Credit Card*. This can be achieved by either using a Regular Expression or by using an enumeration.

Let us see an example. First of all, the element or attribute should be declared as *simpleType*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="PaymentMethod">
     <xsd:simpleType />
   </xsd:element>
</xsd:schema>
```

Listing 8.56

The next step is to add a *restriction* element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="PaymentMethod">
     <xsd:simpleType>
        <xsd:restriction base="xsd:string"/>
     </xsd:simpleType>
   </xsd:element>
</xsd:schema>
```

Listing 8.57

Since the element stores string values, let us define a restriction based on *xsd:string.* Within the restriction element, we could declare a few *enumeration* elements.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="PaymentMethod">
     <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:enumeration value="Cash"/>
          <xsd:enumeration value="Check"/>
          <xsd:enumeration value="Credit Card"/>
        </xsd:restriction>
```

```
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 8.58

The above schema declares an element named *PaymentMethod* and associates it with an enumeration that accepts only the following values: *Cash, Check* or *Credit Card*. Let us create a Schema Collection and see the validation in action.

```
CREATE XML SCHEMA COLLECTION EnumerationTest AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentMethod">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="Cash"/>
        <xsd:enumeration value="Check"/>
        <xsd:enumeration value="Credit Card"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 8.59

```
DECLARE @x XML(EnumerationTest)

-- cash
SET @x = '<PaymentMethod>Cash</PaymentMethod>'

-- check
SET @x = '<PaymentMethod>Check</PaymentMethod>'

-- Credit Card
SET @x = '<PaymentMethod>Credit Card</PaymentMethod>'

/*
Success!!!
*/
```

Listing 8.60

```
DECLARE @x XML(EnumerationTest)
SET @x = '<PaymentMethod>cash</PaymentMethod>'
/*
error: "cash" is invalid. First letter should be capitalized
*/
```

Listing 8.61

```
DECLARE @x XML(EnumerationTest)
SET @x = '<PaymentMethod>Paypal</PaymentMethod>'
/*
error: "Paypal" is not a valid value
*/
```

Listing 8.62

The above can be achieved with a *pattern* restriction as well.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentMethod">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="(Check|Cash|Credit Card)"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 8.63

Let us create an XML schema collection and see whether it performs the same set of validation or not.

```
CREATE XML SCHEMA COLLECTION PatternTest AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentMethod">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="(Check|Cash|Credit Card)"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 8.64

```
DECLARE @x XML(PatternTest)

-- cash
SET @x = '<PaymentMethod>Cash</PaymentMethod>'

-- check
SET @x = '<PaymentMethod>Check</PaymentMethod>'

-- Credit Card
SET @x = '<PaymentMethod>Credit Card</PaymentMethod>'
```

```
/*
Success!!!
*/
```

Listing 8.65

The schema collection does not accept any values other than the ones defined in the pattern.

```
DECLARE @x XML(PatternTest)
SET @x = '<PaymentMethod>cash</PaymentMethod>'
/*
error: "cash" is invalid. First letter should be capitalized
*/
```

Listing 8.66

```
DECLARE @x XML(PatternTest)
SET @x = '<PaymentMethod>Paypal</PaymentMethod>'
/*
error: "Paypal" is not a valid value
*/
```

Listing 8.67

Though we could write a schema to perform the same validation with a *pattern restriction* as well as an *enumeration*, it might be a better choice to go with enumerations as it will make the schema simpler to manage and understand.

Start Writing the Schema

Let us start writing the schema. We have seen the rules earlier and examined ourselves to make sure that we have enough skills to write the *Customer* schema.

Rule 1

> *Each Customer element should contain an attribute named CustomerNumber. It is mandatory and should be EXACTLY five characters long. It should contain only upper case letters.*

Let us start with a basic element declaration for the *Customer* element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Customer"/>
</xsd:schema>
```

Listing 8.68

Now, let us add the *CustomerNumber* attribute.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Customer">
     <xsd:complexType>
        <xsd:attribute name="CustomerNumber"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 8.69

An attribute can be set as mandatory by using the *use* attribute.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Customer">
     <xsd:complexType>
        <xsd:attribute name="CustomerNumber" use="required"/>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 8.70

To apply a restriction on the value of the attribute, the attribute should be declared as a *simpleType.* After the attribute is declared as a simple type, a restriction element can be added to define a restriction.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Customer">
     <xsd:complexType>
       <xsd:attribute name="CustomerNumber" use="required">
         <xsd:simpleType>
            <xsd:restriction base="xsd:string"/>
         </xsd:simpleType>
       </xsd:attribute>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 8.71

The length of the attribute value can be restricted either by restricting the *length* facet or by a *pattern* restriction. The format of the attribute value can be restricted only by a *pattern* restriction.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:attribute name="CustomerNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z]{5}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.72

The above pattern restricts the value of the attribute to be five characters long and would allow only alphabets 'A' to 'Z' in upper case. The *pattern* restriction uses a Regular Expression to specify the format of the value. This Regular Expression language is explained in Chapter 12.

Rule 2

> *A Customer element can have five child elements EXACTLY in the order given below: CustomerName, Billing, Shipping, Terms and Contact.*

At this step, let us declare the child elements of *Customer* element. The rule says that the elements should appear in a specific order. Hence, the child elements of *Customer* element should be placed within a *sequence* element.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"/>
        <xsd:element name="Billing"/>
        <xsd:element name="Shipping"/>
        <xsd:element name ="Terms"/>
        <xsd:element name="Contact"/>
      </xsd:sequence>
      <xsd:attribute name="CustomerNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{5}"/>
          </xsd:restriction>
```

```
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.73

If an element has child elements as well as attributes, the attributes should appear after the declaration of the elements.

Rule 3

> ***CustomerName is optional, and if present should not be more than fifty characters long. The element can appear only once.***

By default, elements are mandatory. An element can be marked as optional by setting *minOccurs* to 0. To indicate that the element cannot appear more than once, we should set the *maxOccurs* attribute to 1. The default value of *maxOccurs* is 1.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"/>
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name="Billing"/>
        <xsd:element name="Shipping"/>
        <xsd:element name ="Terms"/>
        <xsd:element name="Contact"/>
      </xsd:sequence>
      <xsd:attribute name="CustomerNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{5}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.74

The length of the value that an element can store is restricted by applying a restriction on the *maxLength* facet. The element should be declared as a *simpleType* and a restriction element needs to be added within the *simpleType* declaration.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"
                     minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <!-- other declarations here -->
      </xsd:sequence>
      <!-- attribute declaration here -->
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.75

Rule 4

Billing and Contact are mandatory and can appear only once.

As mentioned earlier, all elements are mandatory by default. An element can be marked as mandatory by setting *minOccurs* attribute to 1.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"
                     minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Billing"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Shipping"/>
        <xsd:element name ="Terms"/>
        <xsd:element name="Contact"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <!-- attribute declaration here -->
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.76

Rule 5

Shipping is optional. If not present, the address given in Billing is assumed to be the shipping location. There should be only one shipping element.

An element can be marked as optional by setting *minOccurs* to 0.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"
                     minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Billing"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Shipping"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name ="Terms"/>
        <xsd:element name="Contact"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <!-- attribute declaration here -->
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.77

Rule 6

Terms is mandatory and cannot appear more than once. The value should be one of the following: 30 Days Credit, 60 Days Credit, 90 Days Credit and Against Delivery.

We will use an *enumeration* restriction to implement this rule. We have seen an example a little earlier. Here is how the declaration would look.

```xsd
<xsd:element name ="Terms">
  <xsd:simpleType>
    <xsd:restriction base="xsd:string">
      <xsd:enumeration value="30 Days Credit"/>
      <xsd:enumeration value="60 Days Credit"/>
      <xsd:enumeration value="90 Days Credit"/>
      <xsd:enumeration value="Against Delivery"/>
```

```
        </xsd:restriction>
      </xsd:simpleType>
  </xsd:element>
```

Listing 8.78

Here is the final version of the schema we developed at this lab. The following schema describes the *Customer* element and validates it against all the rules we discussed.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"
                     minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Billing"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Shipping"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name ="Terms">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="30 Days Credit"/>
              <xsd:enumeration value="60 Days Credit"/>
              <xsd:enumeration value="90 Days Credit"/>
              <xsd:enumeration value="Against Delivery"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Contact"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="CustomerNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{5}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 8.79

Let us create a Schema Collection and see the validation in action.

```
CREATE XML SCHEMA COLLECTION CustomerSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"
                     minOccurs="0" maxOccurs="1">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="50"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Billing"
                     minOccurs="1" maxOccurs="1"/>
        <xsd:element name="Shipping"
                     minOccurs="0" maxOccurs="1"/>
        <xsd:element name ="Terms">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="30 Days Credit"/>
              <xsd:enumeration value="60 Days Credit"/>
              <xsd:enumeration value="90 Days Credit"/>
              <xsd:enumeration value="Against Delivery"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Contact"
                     minOccurs="1" maxOccurs="1"/>
      </xsd:sequence>
      <xsd:attribute name="CustomerNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-Z]{5}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
```

Listing 8.80

Here is a correct XML instance that validates with the schema we defined above.

```
DECLARE @x XML(CustomerSchema)
SELECT @x = '
<Customer CustomerNumber="LAZYK">
  <CustomerName>Lazy K Kountry Store</CustomerName>
  <Billing />
  <Shipping />
  <Terms>30 Days Credit</Terms>
  <Contact />
</Customer>'
```

Listing 8.81

It is time for you to play with different XML instances and see how well SQL Server 2005 validates your XML documents, based on the Schema Collection we just defined.

Let us merge this version of the schema with the one we developed in the previous lab.

```
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'OrderInfo'
) BEGIN
      DROP XML SCHEMA COLLECTION OrderInfo
END
GO
```

Listing 8.82

```
CREATE XML SCHEMA COLLECTION OrderInfo AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CustomerName"
                           minOccurs="0" maxOccurs="1">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="50"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Billing"
                           minOccurs="1" maxOccurs="1"/>
              <xsd:element name="Shipping"
                           minOccurs="0" maxOccurs="1"/>
              <xsd:element name ="Terms">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="30 Days Credit"/>
                    <xsd:enumeration value="60 Days Credit"/>
                    <xsd:enumeration value="90 Days Credit"/>
                    <xsd:enumeration value="Against Delivery"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Contact"
                           minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="CustomerNumber" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="[a-zA-Z]{5}"/>
                </xsd:restriction>
              </xsd:simpleType>
```

```
                </xsd:attribute>
              </xsd:complexType>
            </xsd:element>
            <xsd:element name="Items"/>
            <xsd:element name="OrderNote" minOccurs="0">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:maxLength value="500"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="InvoiceNote" minOccurs="0">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:maxLength value="500"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:element>
            <xsd:element name="Discount" minOccurs="0"/>
          </xsd:sequence>
          <xsd:attribute name="OrderNumber" use="required">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:attribute>
        </xsd:complexType>
      </xsd:element>
    </xsd:schema>'
```

Listing 8.83

The following is a correct XML instance that validates with the above schema.

```
 DECLARE @x XML(OrderInfo)
 SELECT @x = '
 <Order OrderNumber="20002">
   <OrderDate>2008-01-01Z</OrderDate>
   <DeliveryDate>2008-01-10T09:00:00-08:00</DeliveryDate>
   <Customer CustomerNumber="LAZYK">
     <CustomerName>Lazy K Kountry Store</CustomerName>
     <Billing />
     <Shipping />
     <Terms>30 Days Credit</Terms>
     <Contact />
   </Customer>
   <Items />
   <OrderNote>Delivery needed before 8 AM</OrderNote>
   <InvoiceNote>
     Adjust the previous credit note with this invoice
   </InvoiceNote>
   <Discount />
 </Order>'
```

Listing 8.84

Try a few different XML instances and make sure that all the validation rules we have discussed so far are correctly defined with the schema collection and that SQL Server validates the XML instances as expected.

Chapter Summary

We have had a good look into *Simple Types* and learned how to derive new types from existing ones. *Simple Types* can be local or global. When declared globally they are always named, and when declared locally they are anonymous. When a *Simple Type* is declared globally, it can be re-used in the declaration of other elements and attributes.

A *Simple Type* can be derived by *List, Union* or by *restriction.* SQL Server 2005 does not allow union of list types and lists of union types. SQL Server 2008 removed this restriction. With SQL Server 2008, you can create union of list types as well as list of union types.

When you derive a new type, you can only set the facets to be more restrictive than the base type.

You can freeze a given facet of a *Simple Type* so that a derived type cannot change the value of that facet. This is done by setting the *fixed* attribute to *"true."* You can restrict a *Simple Type* from being inherited by setting the *final* attribute to *list, union, restriction* or *extension.* The *final* attribute can also take a list of different derivation methods to restrict more than one method of derivation. You can also set it to *"#all"* to restrict all derivation methods.

CHAPTERS 9

XSD BUILT-IN DERIVED DATA TYPES

We have seen XSD Primitive data types in Chapter 7. XSD supports eighteen Primitive data types and SQL Server supports seventeen of them. (SQL Server does not support NOTATION data type.)

In this chapter we will examine the Built-in Derived Data Types of XSD. We will discuss the following:

- XSD Built-in Data Types: Primitive and Derived Data Types
- Facets of built-in data types
- XSD Built-in Derived data types

XSD Built-in Data Types: Primitive and Derived Data Types

We have seen several XSD data types in the topics we covered so far. One of the data types that I mentioned often in the previous discussions is *xsd:integer.* In Chapter 7 we examined XSD Primitive Data Types, but *xsd:integer* was not discussed there.

The reason is that *xsd:integer* is a simple type that derives from one of the primitive data types: *xsd:decimal*. XSD has twenty-five such data types that derive directly or indirectly from one of the Primitive Data types we examined earlier. I will cover all those data types later in this chapter.

In Chapter 8 we learned how to derive new types from existing ones. All the built-in Derived Data Types of XSD are derived in the same manner. For example, *xsd:integer* is derived from *xsd:decimal* by restriction with the following definition.

```
<xs:simpleType name="integer" id="integer">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#integer"/>
  </xs:annotation>
  <xs:restriction base="xs:decimal">
```

```
    <xs:fractionDigits fixed="true" value="0"
                       id="integer.fractionDigits"/>
    <xs:pattern value="[\-+]?[0-9]+"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.1: Definition of built-in simple type "integer."

Let us try to understand this.

```
<xs:simpleType name="integer" id="integer">
```

Listing 9.2: A basic simple type declaration

This statement defines a new type named *integer*. We have used similar statements to define simple types in the previous topics. In none of our examples, we used the *id* attribute. The *id* attribute does not add anything to the validation of the XML instance. This attribute is used only to identify each schema component unique within a given schema. We discussed the *id* attribute when we examined *Element Declarations* and *Attribute Declarations* in Chapter 5 and 6, respectively.

```
<xs:annotation>
  <xs:documentation
    source="http://www.w3.org/TR/xmlschema-2/#integer"/>
</xs:annotation>
```

Listing 9.3: Annotations are used to add documentation to schema components.

We discussed annotations briefly in Chapter 4. I will give a little more detailed explanation in Chapter 13. Annotations are used to add documentation to schema components.

```
<xs:restriction base="xs:decimal">
```

Listing 9.4: "integer" data type derives from "decimal" by restriction.

This statement indicates that *integer* type is derived from XSD Primitive Data Type, *decimal*.

```
<xs:fractionDigits fixed="true" value="0"
                   id="integer.fractionDigits"/>
```

Listing 9.5: "integer" data type does not support decimals.

258

There is something new in this declaration. We have not seen *fractionDigits* previously. This is a facet exposed by all numeric data types and it restricts the number of decimal places in the value. Note that *integer* data type sets *fractionDigits* to 0, to indicate that it should not accept decimals.

The attribute *fixed* indicates that any type that derives from *integer* is not allowed to modify the value of *fractionDigits* facet. The *fixed* attribute is explained later in this chapter.

```
<xs:pattern value="[\-+]?[0-9]+"/>
```

Listing 9.6: A pattern is used to restrict the value space of the "integer" data type.

And finally, it applies a pattern restriction to validate the format of the value. We have seen pattern restrictions a few times in the previous chapters. It uses a Regular Expression language which is explained in Chapter 12.

We just saw the *definition* of the *integer* data type. Just as with *integer,* each of the XSD built-in derived data types derives from one of the Primitive Types directly or indirectly. We will examine XSD Built-in Derived Data Types later in this chapter.

Facets of Data Types

Each data type has a certain set of characteristics that can be used to perform additional validations on the value. For example, by setting the data type of an attribute to *integer* you can make sure that only numbers are accepted. However, sometimes we might need to restrict a type to allow only numbers within a given range (e.g., age of an employee). This could be possible if the data type has a *min* and *max* property. Each of the XSD data types has a certain number of such properties that add additional restrictions on the value. These properties are called *facets* in XSD.

Not all data types support the same facets. Facets of *string* are different from the facets of *float, for example*. Facets like *length, minLength,* and *maxLength* are meaningful for *string* data type, but they are not meaningful with a *float* value. Facets like *fractionDigits* and *totalDigits* are meaningful for *float* values, but not for *date* values.

We will see the facets supported by each data type later in this chapter. Here is a list of all the facets supported by the different data types of XSD.

- length
- minLength
- maxLength
- pattern
- enumeration
- whitespace
- totalDigits
- fractionDigits
- maxInclusive
- minInclusive
- maxExclusive
- minExclusive

Let us examine each of these facets in detail.

Facet: length

This facet restricts the number of characters a *value* can accept. It should always be a non-negative value. 0 is accepted but negative values are not.

When *length* is specified the value being assigned to the given element or attribute should be exactly as long as the value specified in the *length* attribute.

```
CREATE XML SCHEMA COLLECTION StringLength AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ProductCode">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:length value="5"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.7: A string type with "length" restriction.

```
DECLARE @x XML(StringLength)
SET @x = '<ProductCode>ABCDE</ProductCode>'
/*
success!!!
*/
```

Listing 9.8: A "length" restriction makes sure that the value contains the specified number of characters.

```
DECLARE @x XML(StringLength)
SET @x = '<ProductCode>ABCD</ProductCode>'
/*
error!!!
XML Validation: Invalid simple type value: 'ABCD'. Location:
/*:ProductCode[1]
*/

SET @x = '<ProductCode>ABCDEF</ProductCode>'
/*
error!!!
XML Validation: Invalid simple type value: 'ABCD'. Location:
/*:ProductCode[1]
*/
```

Listing 9.9: The "length" validation will fail if the value does not contain the specified number of characters.

If *length* is specified, *minLength* and *maxLength* cannot be used. They are mutually exclusive. The following definition will give an error. (*minLength* and *maxLength* facets are explained later in this chapter)

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="test">
    <xsd:restriction base="xsd:string">
      <xsd:length value="5"/>
      <xsd:maxLength value="2" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 9.10: "length" and "maxLength" are mutually exclusive.

```
Invalid type definition for type 'test', 'Length' can not be
greater than 'maxLength'
```

Let us try to set the *length* and *maxLength* to five and see if it fixes the error.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="test">
    <xsd:restriction base="xsd:string">
      <xsd:length value="5"/>
      <xsd:maxLength value="5" />
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 9.11: "length" and "maxLength" are mutually exclusive, too.

```
Invalid type definition for type 'test', 'maxLength' facet is not
restricting the value space
```

When *length* is used, facets *minLength* or *maxLength* cannot be used.

Facet: minLength

minLength defines the minimum length of the value. The value of *minLength* facet should be 0 or a positive integer. *minLength* cannot be used with *"length"* facet.

The following schema defines a non-empty string element.

```
CREATE XML SCHEMA COLLECTION StringMinLength AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Notes">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:minLength value="1"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.12: "minLength" can be set to 1 to define a non-empty string.

```
DECLARE @x XML(StringMinLength)
SET @x = '<Notes>Urgent!!!</Notes>'
/*
success!!!
*/
```

Listing 9.13: When "minLength" is set to 1, the value should be non empty.

```
DECLARE @x XML(StringMinLength)
SET @x = '<Notes></Notes>'
/*
error!!!
```

```
XML Validation: Invalid simple type value: ''. Location:
/*:Notes[1]
*/
```

Listing 9.14: When "minLength" is set to 1, empty values will be rejected.

The following schema declares an element that allows an empty string.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'StringMinLength'
) BEGIN
      DROP XML SCHEMA COLLECTION StringMinLength
END
GO

CREATE XML SCHEMA COLLECTION StringMinLength AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Notes">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:minLength value="0"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.15: A schema showing a string type with "minLength" set to 0.

```
DECLARE @x XML(StringMinLength)
SET @x = '<Notes>Urgent!!!</Notes>'
/*
success!!!
*/
SET @x = '<Notes></Notes>'
/*
success!!!
*/
SET @x = '<Notes />'
/*
success!!!
*/
```

Listing 9.16: When "minLength" is set to 0, empty values are accepted.

Facet: maxLength

maxLength defines the maximum length of the value. *maxLength* cannot be used with the *length* facet. The value of *maxLength* restriction should be 0 or a positive integer.

If *maxLength* is set to 0, the element or attribute cannot take a value. It always has to be empty. The following schema declares an element that cannot store a value.

```
CREATE XML SCHEMA COLLECTION StringMaxLength AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Notes">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:maxLength value="0"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.17: A schema showing a string type with "maxLength" set to 0.

```
DECLARE @x XML(StringMaxLength)
SET @x = '<Notes>Urgent!!!</Notes>'
/*
error!!!
XML Validation: Invalid simple type value: 'Urgent!!!'. Location:
/*:Notes[1]
*/
```

Listing 9.18: When "maxLength" is set to 0, validation will fail if the element or attribute contains a value.

```
DECLARE @x XML(StringMaxLength)
SET @x = '<Notes></Notes>'
/*
success!!!
*/
SET @x = '<Notes />'
/*
success!!!
*/
```

Listing 9.19

maxLength and *minLength* can be used together to generate the same effect as the *length* restriction.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'StringLength'
) BEGIN
    DROP XML SCHEMA COLLECTION StringLength
END
GO

CREATE XML SCHEMA COLLECTION StringLength AS '
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ProductCode">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:maxLength value="5"/>
        <xsd:minLength value="5"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
Listing 9.20 – "minLength"and "maxLength"can be used to restrict
the length of the value
DECLARE @x XML(StringLength)
SET @x = '<ProductCode>ABCDE</ProductCode>'
/*
success!!!
*/
```

Listing 9.21

```
DECLARE @x XML(StringLength)
SET @x = '<ProductCode>ABCD</ProductCode>'
/*
error!!!
XML Validation: Invalid simple type value: 'ABCD'. Location:
/*:ProductCode[1]
*/

SET @x = '<ProductCode>ABCDEF</ProductCode>'
/*
error!!!
XML Validation: Invalid simple type value: 'ABCD'. Location:
/*:ProductCode[1]
*/
```

Listing 9.22

When *minLength* and *maxLength* are both used, *maxLength* should be greater than or equal to *minLength*.

Facet: pattern

By using *pattern* restriction, you can specify a Regular Expression to validate the value. If you are new to Regular Expressions, you can find some tutorials in the web URLs given below.

- http://www.regular-expressions.info/tutorial.html
- http://www.codeproject.com/dotnet/RegexTutorial.asp
- http://www.amk.ca/python/howto/regex/

Regular Expressions are very popular in programming languages like Perl, Microsoft .NET, Python, etc. While the Regular Expression language of

XSD is slightly different than the Regular Expression languages used in those programming languages, the core syntax and techniques are common to them all.

> **The Regular Expression language supported by XSD is described in Chapter 12**

The following code snippet shows a simple example using a regular expression pattern.

```
CREATE XML SCHEMA COLLECTION StringPattern AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="CustomerNumber">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z]{3}[0-9]{2}" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.23: A schema using "pattern" restriction.

The above schema defines a simple type element named *CustomerNumber.* The format of the customer number is defined by using a pattern restriction. The pattern specifies that the first three characters should be upper case alphas (A to Z), and the next two should be digits (0 to 9). It also ensures that the length of Customer Number is always five characters.

The following example shows an XML instance that successfully validates against the above schema collection.

```
DECLARE @x XML(StringPattern)
SET @x = '<CustomerNumber>JAC01</CustomerNumber>'
/*
success!!!
*/
```

Listing 9.24

None of the XML instances given below will validate successfully.

```
DECLARE @x XML(StringPattern)
SET @x = '<CustomerNumber>JAC001</CustomerNumber>'
SET @x = '<CustomerNumber>jac01</CustomerNumber>'
SET @x = '<CustomerNumber>10001</CustomerNumber>'
/*
```

```
error!!!
XML Validation: Invalid simple type value: Location:
/*:CustomerNumber[1]
*/
```

Listing 9.25

We will examine Regular Expression patterns in Chapter 12.

Pattern restrictions are usually applied with string data types. However, they can be applied on any data type. Depending on the specific requirement, a pattern restriction may be applied to a numeric, date or boolean data type.

Let us see an example. Boolean types accept string literals: *"true," "false,"* *"1"* or *"0."* If you only want to allow *"true"* or *"false"* you can define a pattern restriction to achieve this.

The following example shows a schema which restricts a *boolean* type to accept only *"true"* or *"false."*

```
CREATE XML SCHEMA COLLECTION BooleanPattern
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="IsActive">
    <xsd:simpleType>
      <xsd:restriction base="xsd:boolean">
        <xsd:pattern value="true|false" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.26: A schema showing a pattern restriction.

You will get the following warning when you try to create this schema.

```
Warning: Type 'xs-nun(/IsActive/simpleType())' is restricted by a
facet 'pattern' that may impede full round-tripping of instances
of this type
```

You can ignore this warning for the time being. I have explained this in Chapter 14.

```
declare @x xml(BooleanPattern)
set @x = '<IsActive>true</IsActive>'
set @x = '<IsActive>false</IsActive>'
```

```
/*
success!!!
*/
```

Listing 9.27

```
declare @x xml(BooleanPattern)
set @x = '<IsActive>1</IsActive>'
set @x = '<IsActive>0</IsActive>'
/*
error!!!
XML Validation: Invalid simple type value: '1'. Location:
/*:IsActive[1]
*/
```

Listing 9.28

A *pattern* restriction does not make much sense to a numeric data type as it does to a string data type. However, there may be times when you might need to use a pattern restriction on a numeric type to restrict the format of the values to match a specific business requirement.

The following schema defines a decimal type that allows only odd digits (1, 3, 5, 7 and 9). If you try to use an even digit (2, 4, 6, 8 and 0), SQL Server will generate an error.

```
CREATE XML SCHEMA COLLECTION DecimalPattern
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:pattern value="[1|3|5|7|9]{1,5}" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.29: Definition of a "decimal" type that does not accept even numbers.

Just as in the case with boolean, you will receive the following warning when you try to create this schema collection. I have explained this in Chapter 14.

```
Warning: Type 'xs-nun(/Quantity/simpleType())' is restricted by a
facet 'pattern' that may impede full round-tripping of instances
of this type
declare @x xml(DecimalPattern)
set @x = '<Quantity>1</Quantity>'
set @x = '<Quantity>13</Quantity>'
```

```
set @x = '<Quantity>135</Quantity>'
set @x = '<Quantity>57</Quantity>'
set @x = '<Quantity>77777</Quantity>'
/*
success!!!
*/
```

Listing 9.30

Facet: enumeration

enumeration facet is used to restrict the values to a set of predefined choices. The following example defines a schema which accepts only *USA, Canada, England* and *India* in the list of countries.

```
CREATE XML SCHEMA COLLECTION StringEnumeration AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ShipToCountry">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:enumeration value="USA"/>
        <xsd:enumeration value="Canada"/>
        <xsd:enumeration value="England"/>
        <xsd:enumeration value="India"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.31: A schema showing enumeration facet.

```
DECLARE @x XML(StringEnumeration)
SET @x = '<ShipToCountry>Canada</ShipToCountry>'
SET @x = '<ShipToCountry>India</ShipToCountry>'
/*
success!!!
*/
```

Listing 9.32

```
DECLARE @x XML(StringEnumeration)
SET @x = '<ShipToCountry>Australia</ShipToCountry>'
/*
error!!!
XML Validation: Invalid simple type value: 'Australia'. Location:
/*:ShipToCountry[1]
*/
```

Listing 9.33

An *enumeration* restriction can be applied on any data type. Most of the time this may be applied on string data types. However, based on the specific validation requirements it may be applied on other data types as well. The following example shows an enumeration restriction applied on a numeric type. It restricts the value of *ShipToZip* element to a given set of zip codes.

```
CREATE XML SCHEMA COLLECTION DecimalEnumeration AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ShipToZip">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:enumeration value="10001"/>
        <xsd:enumeration value="10002"/>
        <xsd:enumeration value="10003"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.34: A schema showing enumeration restriction applied on decimal type.

```
DECLARE @x XML(DecimalEnumeration)
SET @x = '<ShipToZip>10003</ShipToZip>'
/*
success!!!
*/
```

Listing 9.35

```
DECLARE @x XML(DecimalEnumeration)
SET @x = '<ShipToZip>10004</ShipToZip>'
/*
error!!!
XML Validation: Invalid simple type value: '10004'. Location:
/*:ShipToZip[1]
*/
```

Listing 9.36

Facet: whiteSpace

whiteSpace restriction defines the way whitespaces is processed by the schema processor. It instructs the XSD processor what needs to be done when it encounters a white space.

SPACES, TABs, Carriage Returns (CR) and Line Feeds (LF) come into the category of whitespaces in XSD parlance. There are three white-space processing modes, namely: *preserve, replace* and *collapse.*

Preserve

When the mode is set to *preserve*, no change is made to the value.

```
CREATE XML SCHEMA COLLECTION whiteSpacePreserve
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Address">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="preserve" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.37: Example showing "preserve" whitespace processing mode.

```
DECLARE @x XML(whiteSpacePreserve)
-- Note that we have plenty of spaces, TAB characters
-- and CR + LF in the following address.
SET @x = '<Address>
              401
                          ,
Time        square                </Address>'

-- Since we have set the whiteSpace restriction to "preserve"
-- the schema processor will not modify the value.
SELECT @x

/*
OUTPUT:

-------------------------------------------------------------
<Address>
              401
                          ,
Time        square                </Address>

(1 row(s) affected)
*/
```

Listing 9.38: When whitespace processing is set to "preserve," the schema processor preserves all TABS, SPACES, Carriage Returns and Line Feeds.

Replace

When *replace* mode is set, all instances of TAB, CR and LF are replaced by SPACES.

```
CREATE XML SCHEMA COLLECTION whiteSpaceReplace
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Address">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="replace" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.39: An example showing "replace" processing mode.

```
DECLARE @x XML(whiteSpaceReplace)
-- Note that we have plenty of spaces, TAB characters
-- and CR + LF in the following address.
SET @x = '<Address>
              401
                             ,
Time       square                 </Address>'

-- Note that TABS, Carriage Returns and Line Feeds are replaced
-- with spaces because the restriction mode is "replace"

SELECT @x

/*
OUTPUT:
--------------------------------------------------------------
<Address>                401   , Time   square    </Address>

(1 row(s) affected)
*/
```

Listing 9.40: When whitespace processing mode is set to "replace," all Spaces, Tabs, Carriage Returns and Line Feeds are replaced by spaces.

Collapse

When *collapse* mode is specified, the schema processor will first apply a *replace* operation. Then it will remove contiguous sequences of SPACES with a single SPACE character. It will then remove leading and trailing spaces from the value.

```
CREATE XML SCHEMA COLLECTION whiteSpaceCollapse
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Address">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:whiteSpace value="collapse" />
      </xsd:restriction>
```

```
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.41: Example showing "collapse" processing mode.

```
DECLARE @x XML(whiteSpaceCollapse)
-- Note that we have plenty of spaces, TAB characters
-- and CR + LF in the following address.
SET @x = '<Address>
                401
                                ,
Time        square                  </Address>'

-- Note that TABS, Carriage Returns and Line Feeds are replaced
-- with spaces, contiguous spaces are then replaced with a single
-- space and leading and trailing spaces are removed from the
value.

SELECT @x

/*
OUTPUT:
---------------------------------------------------------------
-
<Address>401 , Time square</Address>

(1 row(s) affected)
*/
```

Listing 9.42: When "collapse" mode is specified, all whitespace characters are first replaced with spaces, then contiguous sequences of spaces are replaced with a single space, and leading and trailing spaces are removed.

Each data type has a default white space processing mode. For example, the default white space processing mode of *string* type is *preserve.* The default mode of *boolean* is *collapse.*

When you derive a new simple type from a base type, you can alter the white space processing to a more restrictive mode. For example, you can change *preserve* to *collapse* or *replace.* But you cannot change *replace* to *preserve.*

The following table explains the white space restriction allowed on the derived type.

Base Type	Derived Type		
Preserve	Preserve	Replace	Collapse
Replace	Not allowed	Replace	Collapse
Collapse	Not allowed	Not Allowed	Collapse

If you try to change the *whiteSpace* processing mode to a less restrictive one, SQL Server will generate an error as given below.

```
Invalid facet value for facet 'whiteSpace' in type definition 'xs-
nun(/typename/simpleType())'
```

Facet: totalDigits

totalDigits restricts the number of digits the type can hold. It includes the number of digits in the integer part as well as in the decimal fraction part.

```
CREATE XML SCHEMA COLLECTION decimalDemo
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="5" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.43: An example showing "fraction Digits" restriction.

```
declare @x xml(decimalDemo)
set @x = '<Quantity>1</Quantity>'
set @x = '<Quantity>12</Quantity>'
set @x = '<Quantity>123</Quantity>'
set @x = '<Quantity>1234</Quantity>'
set @x = '<Quantity>12345</Quantity>'
set @x = '<Quantity>12.34</Quantity>'
```

```
set @x = '<Quantity>12.341</Quantity>'
/*
success!!!
*/
```

Listing 9.44

```
declare @x xml(decimalDemo)

set @x = '<Quantity></Quantity>'
/*
XML Validation: Invalid simple type value: ''. Location:
/:Quantity[1]
*/

set @x = '<Quantity>123456</Quantity>'
/*
XML Validation: Invalid simple type value: '123456'. Location:
/:Quantity[1]
*/

set @x = '<Quantity>12.3412</Quantity>'
/*
XML Validation: Invalid simple type value: '12.3412'. Location:
/:Quantity[1]
*/
```

Listing 9.45

Note that the decimal point is not counted when validating the number
of digits.

Facet: fractionDigits

fractionDigits restricts the number of digits in the decimal part of the value
(on the right of the decimal point). *fractionDigits* facet defines the maximum
number of digits allowed. Validation will fail if there are more digits than the
value specified with *fractionDigits*. It is acceptable to have a lesser number
of digits after the decimal point than the value specified in the restriction.

```
CREATE XML SCHEMA COLLECTION DecimalFraction
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="5" />
        <xsd:fractionDigits value="2" />
      </xsd:restriction>
```

```
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.46: An example showing "fractionDigits"restriction.

```
declare @x xml(DecimalFraction)

set @x = '<Quantity>1</Quantity>'
set @x = '<Quantity>12</Quantity>'
set @x = '<Quantity>123</Quantity>'
set @x = '<Quantity>1234</Quantity>'
set @x = '<Quantity>12345</Quantity>'
set @x = '<Quantity>12.34</Quantity>'
/*
success!!!
*/
```

Listing 9.47

```
declare @x xml(DecimalFraction)

set @x = '<Quantity>12.341</Quantity>'
/*
error!!!
XML Validation: Invalid simple type value: '12.341'. Location:
/*:Quantity[1]
*
```

Listing 9.48

Facet: maxInclusive

maxInclusive specifies the highest value the type can accept. The highest acceptable value is inclusive of the value specified with the restriction.

```
CREATE XML SCHEMA COLLECTION DecimalMaxInclusive
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:maxInclusive value="1000" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.49: An example showing "maxInclusive"restriction.

```
declare @x xml(DecimalMaxInclusive)
set @x = '<Quantity>1000</Quantity>'
/*
success!!!
*/
```

Listing 9.50

```
declare @x xml(DecimalMaxInclusive)
set @x = '<Quantity>1001</Quantity>'
/*
error!!!
XML Validation: Invalid simple type value: '1001'. Location:
/*:Quantity[1]
*/
```

Listing 9.51

Facet: maxExclusive

maxExclusive specifies the highest numeric value the type can accept, excluding the value specified in the restriction.

```
CREATE XML SCHEMA COLLECTION DecimalMaxExclusive
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:maxExclusive value="1000" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
```

Listing 9.52: An example showing "maxExclusive"restriction.

```
declare @x xml(DecimalMaxExclusive)
set @x = '<Quantity>999</Quantity>'
/*
success!!!
*/
```

Listing 9.53

```
declare @x xml(DecimalMaxExclusive)
set @x = '<Quantity>1000</Quantity>'
/*
error!!!
```

```
XML Validation: Invalid simple type value: '1000'. Location:
/*:Quantity[1]
*/
```

Listing 9.54

Facet: minInclusive

minInclusive defines the lowest value that the type can accept. The lowest acceptable value is inclusive of the value specified in the restriction.

```
CREATE XML SCHEMA COLLECTION DecimalMinInclusive
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:minInclusive value="12" />
      </xsd:restriction>\
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 9.55: An example using "minInclusive"restriction.

```
declare @x xml(DecimalMinInclusive)

set @x = '<Quantity>12</Quantity>'

/*
success!!!
*/
```

Listing 9.56

```
declare @x xml(DecimalMinInclusive)

set @x = '<Quantity>11</Quantity>'

/*
error!!!
XML Validation: Invalid simple type value: '11'. Location:
/*:Quantity[1]
*/
```

Listing 9.57

Facet: minExclusive

minExclusive defines the minimum value that the type can accept, excluding the value specified in the restriction.

```
CREATE XML SCHEMA COLLECTION DecimalMinExclusive
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Quantity">
    <xsd:simpleType>
      <xsd:restriction base="xsd:decimal">
        <xsd:minExclusive value="12" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 9.58: An example showing "minExclusive"restriction.

```
declare @x xml(DecimalMinExclusive)

set @x = '<Quantity>12</Quantity>'
/*
error!!!
XML Validation: Invalid simple type value: '12'. Location:
/*:Quantity[1]
*/
```

Listing 9.59

```
declare @x xml(DecimalMinExclusive)

set @x = '<Quantity>13</Quantity>'
/*
success!!!
*/
```

Listing 9.60

Facets of Primitive Data Types

As mentioned earlier, each data type supports a certain set of facets. We have examined each facet in detail. The following table summarizes the facets supported by each Primitive Data Type.

	pattern	whitespace	length	minLength	maxLength	enumeration	totalDigits	fractionDigits	minInclusive	maxInclusive	minExclusive	maxExclusive
string	✓	✓	✓	✓	✓	✓						
boolean	✓	✓										
decimal	✓	✓				✓	✓	✓	✓	✓	✓	✓
float	✓	✓				✓			✓	✓	✓	✓
double	✓	✓				✓			✓	✓	✓	✓
duration	✓	✓				✓			✓	✓	✓	✓
dateTime	✓	✓				✓			✓	✓	✓	✓
time	✓	✓				✓			✓	✓	✓	✓
date	✓	✓				✓			✓	✓	✓	✓
gYear	✓	✓				✓			✓	✓	✓	✓
gMonthDay	✓	✓				✓			✓	✓	✓	✓
gDay	✓	✓				✓			✓	✓	✓	✓
gMonth	✓	✓				✓			✓	✓	✓	✓
haxBinary	✓	✓	✓	✓	✓	✓						
base64 Binary	✓	✓	✓	✓	✓	✓						
anyURI	✓	✓	✓	✓	✓	✓						
QName	✓	✓	✓	✓	✓	✓						

Listing 9.61: Facets supported by each primitive data type.

XSD Built-in Derived Data Types

In Chapter 7 we examined the Primitive Data Types of XSD. In Chapter 8 we saw how to derive new types from existing ones. We saw three derivation methods: *By restriction, By List and By Union.*

XSD Built-in Derived Data Types are *Simple Types* that derive directly or indirectly from one of the Primitive Data Types. XSD has twenty-five such types and SQL Server supports twenty-two of them.

SQL Server does not support XSD data types: *ID, IDREF* and *IDREFS*.

Out of the twenty-two derived data types, only two types derive directly from one of the Primitive Data Types. The following are the types that directly derive from the Primitive Data Types.

Derived Type	Base Type
normalizedString	string
Integer	decimal

The following eight data types derive either from *normalizedString* or one of its derived types.

Derived Type	Base Type
Token	normalizedString
NMTOKEN	token
Name	token
Language	token
NMTOKENS	NMTOKEN
NCName	Name
ENTITY	NCName
ENTITIES	ENTITY

The following diagram shows the types that derive from "string" along with the derivation types, as well as the restrictions added to the facets:
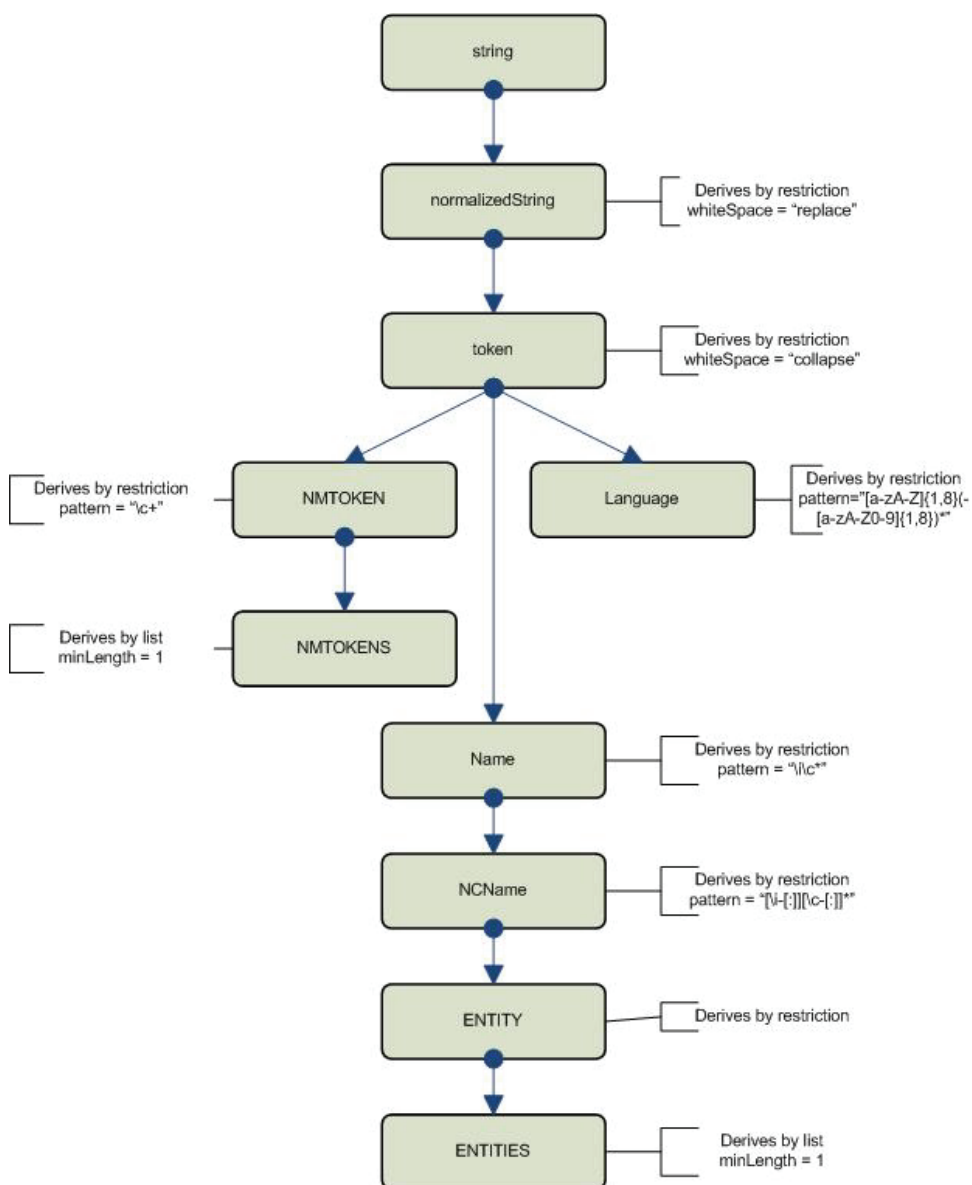
9 – XSD built-in derived data types

The following twelve data types derive either from *integer* or one of its derived types.

Derived Type	Base Type
nonPositiveInteger	integer
nonNegativeInteger	integer
Long	integer
negativeInteger	nonPositiveInteger
int	Long
short	int
byte	short
unsignedLong	nonNegativeInteger
positiveInteger	nonNegativeInteger
unsignedInt	unsignedLong
unsignedShort	unsignedInt
unsignedByte	unsignedShort

The following diagram shows the inheritance chain of XSD built-in data types that derive from "decimal."
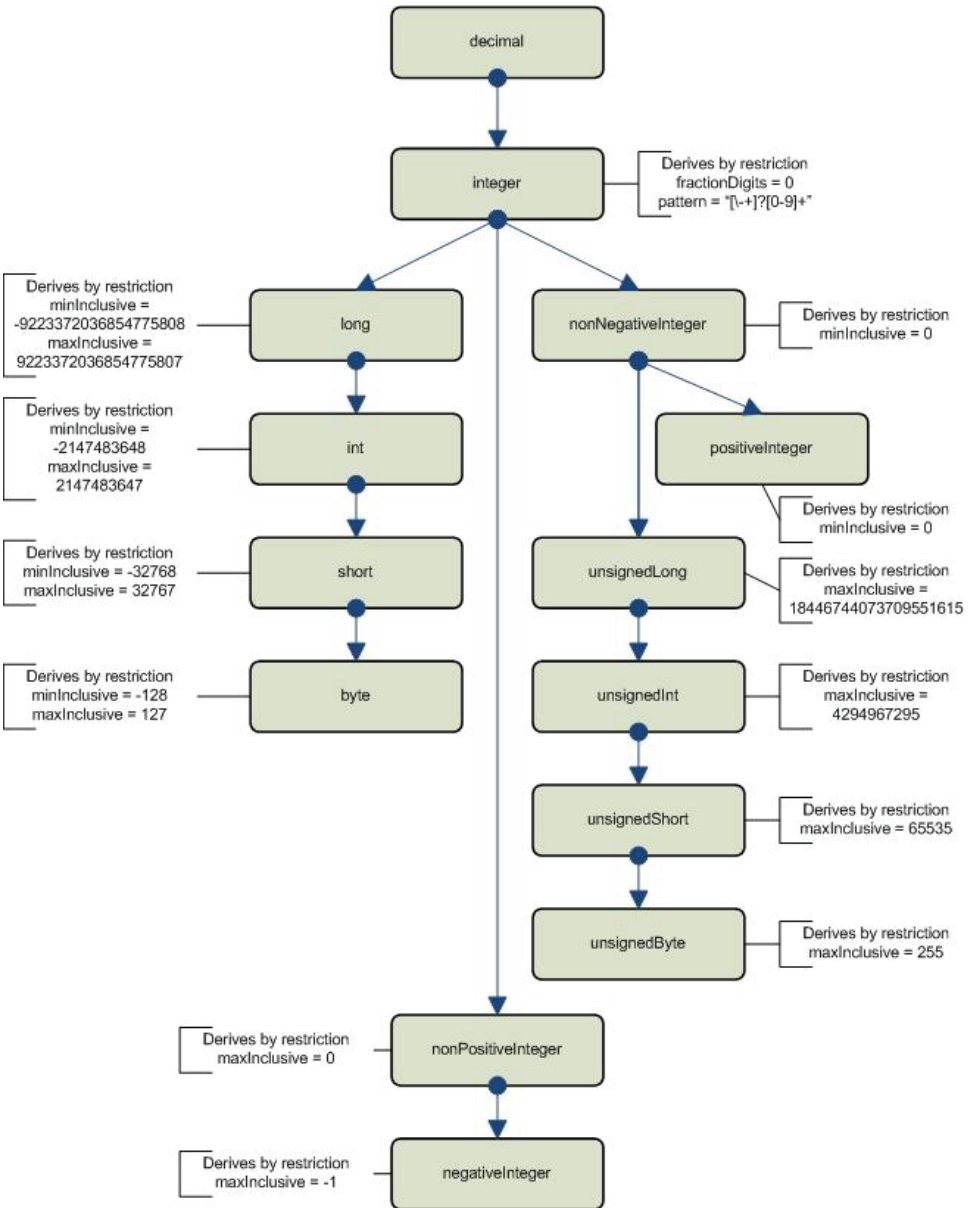
Let us examine each of these data types in detail.

normalizedString

XSD data type *normalizedString* stores a string that does not contain TABs, Carriage Returns or Line Feeds. It is derived by restriction from *string* by setting the whitespace processing mode to *"replace."*

```
<xs:simpleType name="normalizedString" id="normalizedString">
  <xs:annotation>
    <xs:documentation
    source="http://www.w3.org/TR/xmlschema-2/#normalizedString"/>
  </xs:annotation>
  <xs:restriction base="xs:string">
    <xs:whiteSpace value="replace"
                   id="normalizedString.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.62: Definition of XSD normalizedString type.

In Chapter 8 we saw how to derive new types by applying restrictions to one or more facets of a base type. XSD data type *normalizedString* is created by applying a restriction on the whitespace processing mode of the *string* data type.

The default whitespace processing mode of *string* data type is *"preserve."*This processing mode does not modify the value being assigned to a string element or attribute*. normalizedString* changes the white space processing mode to *"replace,"*which replaces TAB, CR and LF characters with SPACES. Thus, a *normalizedString* cannot store TAB, CR or LF characters. But it can contain leading or trailing spaces as well as contiguous blocks of spaces within the value.

```
CREATE XML SCHEMA COLLECTION normalizedString
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MyString" type="xsd:normalizedString" />
</xsd:schema>'
GO

DECLARE @x XML(normalizedString)

-- add a string which contains tabs and carriage return/LF
SET @x = '
<MyString>this
is    a
sales
order</MyString>'

SELECT @x

/*
OUTPUT:
```

```
------------------------------------------
<MyString>this  is a sales  order</MyString>

(1 row(s) affected)
*/
```

Listing 9.63: An example using normalizedString data type.

Note that the string that we assigned to the XML variable had TABS, Carriage Returns and Line Feeds. But the schema processor removed them before assigning the value to the XML element. When we query the variable, we do not find those characters anymore.

This shows that the whitespace restriction works little differently from other restrictions. It is more of a processing instruction rather than a validation rule. In the case of other restrictions, if the XML value does not follow the rules an error will be raised. But in the case of white space processing, the Schema processor performs whitespace handling in the way we defined in the restriction. I prefer to call this a processing instruction rather than a restriction.

normalizedString data type has the following facets.

length	minLength	maxLength
pattern	enumeration	whiteSpace

All the restrictions, except *whiteSpace* processing, work exactly the same way they work with the base data type; *string.* We saw in the previous chapter that, when a type is derived, it cannot be less restrictive than the parent type. Hence, the following is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MyString">
    <xsd:simpleType>
      <xsd:restriction base ="xsd:normalizedString">
        <xsd:whiteSpace value="preserve" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 9.64: A derived type cannot be less restrictive than the base type.

normalizedString derives from string by setting the white space processing mode to "replace." "preserve" is less restrictive than "replace" and, as a result, we cannot set the whitespace processing mode of a normalizedString derived type to "preserve."

The following is valid.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="MyString">
     <xsd:simpleType>
       <xsd:restriction base ="xsd:normalizedString">
         <xsd:whiteSpace value="Collapse" />
       </xsd:restriction>
     </xsd:simpleType>
   </xsd:element>
</xsd:schema>
```

Listing 9.65: A derived type can set a facet to be more restrictive than the base type if the facet is not "fixed."

"collapse" is more restrictive than *"replace,"* hence, the above usage is valid.

integer

We have seen *integer* data type in many of the examples we created in the previous chapters. *integer* is derived from *decimal* by restricting the number of decimal places to 0. Here is the definition of the *integer* data type.

```xsd
<xs:simpleType name="integer" id="integer">
   <xs:annotation>
     <xs:documentation
       source="http://www.w3.org/TR/xmlschema-2/#integer"/>
   </xs:annotation>
   <xs:restriction base="xs:decimal">
     <xs:fractionDigits fixed="true" value="0"
                        id="integer.fractionDigits"/>
     <xs:pattern value="[\-+]?[0-9]+"/>
   </xs:restriction>
</xs:simpleType>
```

Listing 9.66: Definition of XSD integer data type.

integer sets *fractionDigits* to 0 so that no digits will appear after the decimal point. Further, it sets a pattern value which restricts even the decimal point. The pattern says that *the value may have a + or – sign followed by digits.*

Note the usage of the *fixed* attribute. It instructs the schema processor to restrict any derived types from modifying the *fractionDigits* facet. It is similar to the *sealed* attribute some of the OOP languages support.

The following is illegal:

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MyInteger">
    <xsd:simpleType>
      <xsd:restriction base ="xsd:integer">
        <xsd:fractionDigits value="2" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 9.67: A derived type cannot modify a "fixed" facet.

However, it is acceptable if you set the value of *fractionDigits* to 0 because it does not modify the base type definition.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MyInteger">
    <xsd:simpleType>
      <xsd:restriction base ="xsd:integer">
        <xsd:fractionDigits value="0" />
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 9.68: A derived type can set the value of a "fixed"facet to the same value as defined in the base type.

Here is an example that uses *integer* data type.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Age" type="xsd:integer"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Age>30</Age>'
```

Listing 9.69: An example using integer data type.

integer data type has the following facets.

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

token

token is derived from *normalizedString* by setting the whitespace processing mode to *"collapse."* Here is the definition of *token* data type.

```xml
<xs:simpleType name="token" id="token">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#token"/>
  </xs:annotation>
  <xs:restriction base="xs:normalizedString">
    <xs:whiteSpace value="collapse" id="token.whiteSpace"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.70: Definition of XSD token data type.

When white space processing is set to *"collapse,"* all TAB, Carriage Return and Line Feed characters will be replaced with spaces. Then leading and trailing spaces are removed from the values. Further, contiguous occurrences of more than one space will be replaced with a single space.

token has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

All token facets, except *whiteSpace* processing, work exactly the same way as with *string* or *normalizedString.* Any type that derives from *token* cannot modify the whitespace processing mode because *"collapse"*is the most restrictive whitespace processing mode.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
```

289

```
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Message" type="xsd:token"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Message>
      Hi Jacob


                  bye            </Message>'

-- read the value
SELECT @x
/*
<Message>Hi Jacob bye</Message>
*/
```

Listing 9.71: An example using token data type.

Note that the schema processor removed leading and trailing spaces. It also replaced TAB, CR and LF characters with spaces and replaced contiguous sequences of spaces with a single space.

language

language derives from *token* by applying the following restriction.

```
<xs:simpleType name="language" id="language">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#language"/>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value="[a-zA-Z]{1,8}(-[a-zA-Z0-9]{1,8})*"
                id="language.pattern">
      <xs:annotation>
        <xs:documentation
          source="http://www.ietf.org/rfc/rfc3066.txt">
          pattern specifies the content of section 2.12 of XML
          1.0e2 and RFC 3066 (Revised version of RFC 1766).
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.72: Definition of XSD language data type.

This data type represents the language identifiers defined in RFC 3066. The pattern specifies the following:

- *language* can have two blocks of one to eight character-long values, separated by a hyphen (-)
- The second block is optional
- The hyphen should be present only if the second block is not empty
- The first part can have letters a-z in upper or lower case.
- The second part can have letters a-z in upper or lower case, as well as digits 0-9.

language has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Lang" type="xsd:language"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Lang>EN-US</Lang>'
```

Listing 9.73: An example using language data type.

NMTOKEN

NMTOKEN (Name Token) is a more restricted version of *token* that does not permit white spaces and special characters. It derives from *token* with the following restriction.

```
<xs:simpleType name="NMTOKEN" id="NMTOKEN">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#NMTOKEN"/>
```

```
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value="\c+" id="NMTOKEN.pattern">
      <xs:annotation>
        <xs:documentation
          source="http://www.w3.org/TR/REC-xml#NT-Nmtoken">
          pattern matches production 7 from the XML spec
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.74: Definition of XSD NMTOKEN data type.

The Regular Expression given in the pattern restriction represents one or more occurrences of normal a character. We will see more about the Regular Expression language of XSD in Chapter 12.

NMTOKEN has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="TableName" type="xsd:NMTOKEN"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<TableName>Customers</TableName>'
```

Listing 9.75: An example using NMTOKEN data type.

NMTOKENS

NMTOKENS stores a space separated list of *NMTOKEN* values. It derives from *NMTOKEN* by *list.*

```xml
<xs:simpleType name="NMTOKENS" id="NMTOKENS">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasFacet name="length"/>
      <hfp:hasFacet name="minLength"/>
      <hfp:hasFacet name="maxLength"/>
      <hfp:hasFacet name="enumeration"/>
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality"
                       value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#NMTOKENS"/>
  </xs:annotation>
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="xs:NMTOKEN"/>
    </xs:simpleType>
    <xs:minLength value="1" id="NMTOKENS.minLength"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.76: Definition of XSD NMTOKENS data type.

In the case of types derived from *string* by restriction, the *length* refers to the count of characters. However, when applied on a type which derives by *list*, it refers to the number of items in the list. Each item in the list is separated by a white space.

NMTOKENS has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO
```

```
-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Keys" type="xsd:NMTOKENS"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Keys>DivisionID CustmerID</Keys>'
```

Listing 9.77: An example using NMTOKENS data type.

Name

Name represents a valid XML name value which is very close to *NMTOKEN,* except that *Name* does not allow a digit at the beginning of the value. It derives from *token* with the following restriction.

```
<xs:simpleType name="Name" id="Name">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#Name"/>
  </xs:annotation>
  <xs:restriction base="xs:token">
    <xs:pattern value="\i\c*" id="Name.pattern">
      <xs:annotation>
        <xs:documentation
          source="http://www.w3.org/TR/REC-xml#NT-Name">
          pattern matches production 5 from the XML spec
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.78: Definition of XSD Name data type.

Name has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
```

```
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Key" type="xsd:Name"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
-- all these are valid values
set @x = '<Key>AAA</Key>'
set @x = '<Key>_abc</Key>'
set @x = '<Key>a124</Key>'
set @x = '<Key>a:b</Key>'

-- the following value is invalid
-- set @x = '<Key>1Abc</Key>'
```

Listing 9.79: An example using Name data type.

NCName

NCName stands for *Non-colonized Name*. A *colonized name* has a namespace related prefix. For example, *"xsd:string"* is a colonized name. If you take *"xsd"* and *"string"* separately, they are both *Non-Colonized Names*.

NCName derives from *Name* with the following restriction.

```
<xs:simpleType name="NCName" id="NCName">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#NCName"/>
  </xs:annotation>
  <xs:restriction base="xs:Name">
    <xs:pattern value="[\i-[:]][\c-[:]]*" id="NCName.pattern">
      <xs:annotation>
        <xs:documentation
          source="http://www.w3.org/TR/REC-xml-names/#NT-NCName">
          pattern matches production 4 from the Namespaces
          in XML spec
        </xs:documentation>
      </xs:annotation>
    </xs:pattern>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.80: Definition of XSD NCName data type.

NCName has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

All the examples we tried for *Name* will work for *NCName,* except the example that takes a colon in its value.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Key" type="xsd:NCName"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
-- all these are valid values
set @x = '<Key>AAA</Key>'
set @x = '<Key>_abc</Key>'
set @x = '<Key>a124</Key>'

-- this will work for Name, but not for NCName
--set @x = '<Key>a:b</Key>'
```

Listing 9.81: An example using NCName data type.

ENTITY

ENTITY derives from *NCName* but does not add any additional restrictions. So the same validation rules which are applicable to *NCName* are applicable to *ENTITY,* too.

```
<xs:simpleType name="ENTITY" id="ENTITY">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#ENTITY"/>
```

```
        </xs:annotation>
    <xs:restriction base="xs:NCName"/>
</xs:simpleType>
```

Listing 9.82: Definition of XSD ENTITY data type.

ENTITY has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Key" type="xsd:ENTITY"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
-- all these are valid values
set @x = '<Key>AAA</Key>'
set @x = '<Key>_abc</Key>'
set @x = '<Key>a124</Key>'

-- this will work for Name, but not for ENTITY
--set @x = '<Key>a:b</Key>'
```

Listing 9.83: An example using ENTITY data type.

ENTITIES

ENTITIES data type derives from *ENTITY* by list. It stores a space-separated list of *ENTITY* values.

```
<xs:simpleType name="ENTITIES" id="ENTITIES">
   <xs:annotation>
     <xs:appinfo>
       <hfp:hasFacet name="length"/>
       <hfp:hasFacet name="minLength"/>
       <hfp:hasFacet name="maxLength"/>
       <hfp:hasFacet name="enumeration"/>
```

```
      <hfp:hasFacet name="whiteSpace"/>
      <hfp:hasFacet name="pattern"/>
      <hfp:hasProperty name="ordered" value="false"/>
      <hfp:hasProperty name="bounded" value="false"/>
      <hfp:hasProperty name="cardinality"
                       value="countably infinite"/>
      <hfp:hasProperty name="numeric" value="false"/>
    </xs:appinfo>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#ENTITIES"/>
  </xs:annotation>
  <xs:restriction>
    <xs:simpleType>
      <xs:list itemType="xs:ENTITY"/>
    </xs:simpleType>
    <xs:minLength value="1" id="ENTITIES.minLength"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.84: Definition of XSD ENTITIES data type.

ENTITIES has the following facets:

length	minLength	maxLength
pattern	enumeration	whiteSpace

Note that the facets *length, minLength* and *maxLength* refer to the number of items in the list. They do not refer to the number of characters in an item.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Keys" type="xsd:ENTITIES"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<Keys>CustomerID ProductID</Keys>'
```

Listing 9.85: An example using ENTITIES data type.

nonPositiveInteger

nonPositiveInteger represents an integer value which is less than or equal to 0. The highest value allowed is 0. The negative sign (-) is mandatory except for 0. It derives from *integer* with the following restriction.

```
<xs:simpleType name="nonPositiveInteger" id="nonPositiveInteger">
  <xs:annotation>
    <xs:documentation
    source="http://www.w3.org/TR/xmlschema-
2/#nonPositiveInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:maxInclusive value="0"
id="nonPositiveInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.86: Definition of XSD nonPositiveInteger data type.

nonPositiveInteger has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:nonPositiveInteger"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
-- all these are valid values
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>-0</SomeValue>'
```

```
set @x = '<SomeValue>+0</SomeValue>'
set @x = '<SomeValue>-1</SomeValue>'
set @x = '<SomeValue>-1999</SomeValue>'
```

Listing 9.87: An example using nonPositiveInteger data type.

negativeInteger

negativeInteger derives from *nonPositiveInteger*. The only difference from its base type is that *negativeInteger* does not allow 0 as its value. The highest value allowed is -1.

It derives from *nonPositiveInteger* with the following restriction.

```xml
<xs:simpleType name="negativeInteger" id="negativeInteger">
  <xs:annotation>
    <xs:documentation
        source="http://www.w3.org/TR/xmlschema-
                2/#negativeInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:nonPositiveInteger">
    <xs:maxInclusive value="-1"
      id="negativeInteger.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.88: Definition of XSD negativeInteger data type.

negativeInteger has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO


-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:negativeInteger"/>
```

```
</xsd:schema>'
GO


-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>-1</SomeValue>'
set @x = '<SomeValue>-1999</SomeValue>'
```

Listing 9.89: An example using negativeInteger data type.

long

long derives from *integer* with the following restriction.

```
<xs:simpleType name="long" id="long">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasProperty name="bounded" value="true"/>
      <hfp:hasProperty name="cardinality" value="finite"/>
    </xs:appinfo>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#long"/>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="-9223372036854775808"
                     id="long.minInclusive"/>
    <xs:maxInclusive value="9223372036854775807"
                     id="long.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.90: Definition of XSD long data type.

long has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive


```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
```

```
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:long"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>-9223372036854775808</SomeValue>'
set @x = '<SomeValue>9223372036854775807</SomeValue>'
```

Listing 9.91: An example using long data type.

int

int derives from *long* and restricts the minimum and maximum values supported by the base type. Here is the definition of *int* data type.

```
<xs:simpleType name="int" id="int">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#int"/>
  </xs:annotation>
  <xs:restriction base="xs:long">
    <xs:minInclusive value="-2147483648" id="int.minInclusive"/>
    <xs:maxInclusive value="2147483647" id="int.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.92: Definition of XSD int data type.

int has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
   <xsd:element name="SomeValue" type="xsd:int"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>-2147483648</SomeValue>'
set @x = '<SomeValue>2147483647</SomeValue>'
```

Listing 9.93: An example using int data type.

short

short derives from *int* and restricts the minimum and maximum values supported by the base type. Here is the definition of *short.*

```
<xs:simpleType name="short" id="short">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#short"/>
  </xs:annotation>
  <xs:restriction base="xs:int">
    <xs:minInclusive value="-32768" id="short.minInclusive"/>
    <xs:maxInclusive value="32767" id="short.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.94: Definition of XSD short data type.

short has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:short"/>
```

```
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>-32768</SomeValue>'
set @x = '<SomeValue>32767</SomeValue>'
```

Listing 9.95: An example using short data type.

byte

byte derives from *short* and restricts the minimum and maximum values supported by the base type. Here is the definition of *byte.*

```
<xs:simpleType name="byte" id="byte">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#byte"/>
  </xs:annotation>
  <xs:restriction base="xs:short">
    <xs:minInclusive value="-128" id="byte.minInclusive"/>
    <xs:maxInclusive value="127" id="byte.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.96: Definition of XSD byte data type.

byte has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:byte"/>
```

```
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>-128</SomeValue>'
set @x = '<SomeValue>127</SomeValue>'
```

Listing 9.97: An example using byte data type.

nonNegativeInteger

nonNegativeInteger derives from *integer* with the following restriction on the minimum and maximum values supported by the base type.

```
<xs:simpleType name="nonNegativeInteger" id="nonNegativeInteger">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-
2/#nonNegativeInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="0"
id="nonNegativeInteger.minInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.98: Definition of XSD nonNegativeInteger data type.

nonNegativeInteger has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:nonNegativeInteger"/>
</xsd:schema>'
```

```
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>127</SomeValue>'
```

Listing 9.99: An example using nonNegativeInteger.

unsignedLong

unsignedLong derives from *nonNegativeInteger* with the following restriction.

```
<xs:simpleType name="unsignedLong" id="unsignedLong">
  <xs:annotation>
    <xs:appinfo>
      <hfp:hasProperty name="bounded" value="true"/>
      <hfp:hasProperty name="cardinality" value="finite"/>
    </xs:appinfo>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#unsignedLong"/>
  </xs:annotation>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:maxInclusive value="18446744073709551615"
                     id="unsignedLong.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.100: Definition of XSD unsignedLong data type.

unsignedLong has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
   <xsd:element name="SomeValue" type="xsd:unsignedLong"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>18446744073709551615</SomeValue>'
```

Listing 9.101: An example using unsignedLong data type.

unsignedInt

unsignedInt derives from *unsignedLong* with the following restriction.

```
<xs:simpleType name="unsignedInt" id="unsignedInt">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#unsignedInt"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedLong">
    <xs:maxInclusive value="4294967295"
                     id="unsignedInt.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.102: Definition of XSD unsignedInt data type.

unsignedInt has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="SomeValue" type="xsd:unsignedInt"/>
</xsd:schema>'
GO
```

```
-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>4294967295</SomeValue>'
```

Listing 9.103: An example using unsignedInt data type.

unsignedShort

unsignedShort derives from *unsignedInt* with the following restriction.

```
<xs:simpleType name="unsignedShort" id="unsignedShort">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#unsignedShort"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedInt">
    <xs:maxInclusive value="65535"
                     id="unsignedShort.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.104: Definition of XSD unsignedShort data type.

unsignedShort has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:unsignedShort"/>
</xsd:schema>'
GO

-- Validate
```

```
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>65535</SomeValue>'
```

Listing 9.105: An example using unsignedShort data type.

unsignedByte

unsignedByte derives from *unsignedShort* with the following restriction.

```
<xs:simpleType name="unsignedByte" id="unsignedByte">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#unsignedByte"/>
  </xs:annotation>
  <xs:restriction base="xs:unsignedShort">
    <xs:maxInclusive value="255" id="unsignedByte.maxInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.106: Definition of XSD unsignedShort data type.

unsignedByte has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:unsignedByte"/>
</xsd:schema>'
GO
```

```
-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>0</SomeValue>'
set @x = '<SomeValue>255</SomeValue>'
```

Listing 9.107: An example using unsignedByte data type.

positiveInteger

positiveInteger derives from *nonNegativeInteger* with the following restriction.

```
<xs:simpleType name="positiveInteger" id="positiveInteger">
  <xs:annotation>
    <xs:documentation
      source="http://www.w3.org/TR/xmlschema-2/#positiveInteger"/>
  </xs:annotation>
  <xs:restriction base="xs:nonNegativeInteger">
    <xs:minInclusive value="1" id="positiveInteger.minInclusive"/>
  </xs:restriction>
</xs:simpleType>
```

Listing 9.108: Definition of XSD positiveInteger data type.

positiveInteger has the following constraining facets:

totalDigits	fractionDigits	pattern
whiteSpace	enumeration	maxInclusive
minInclusive	maxExclusive	minExclusive

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="SomeValue" type="xsd:positiveInteger"/>
</xsd:schema>'
GO
```

```
-- Validate
DECLARE @x XML(ExampleSchema)
set @x = '<SomeValue>1</SomeValue>'
set @x = '<SomeValue>989</SomeValue>'
```

Listing 9.109: An example using positiveInteger data type.

Facets of XSD Built-in Derived Data Types

Here is a summary of the facets supported by XSD Built-in Derived data types.

All the string data types that derive from *string* support the following facets.

- length
- minLength
- maxLength
- pattern
- enumeration
- whiteSpace

All the numeric data types that derive from *decimal* support the following facets.

- pattern
- enumeration
- whitespace
- totalDigits
- fractionDigits
- maxInclusive
- maxExclusive
- minInclusive
- minExclusive

Chapter Summary

XSD has defined eighteen Primitive Data Types and twenty-five Derived data types. Out of the eighteen Primitive Data Types, SQL Server supports seventeen (it does not support NOTATION). Out of the twenty-five

311

Derived data types, SQL Server supports twenty-two (it does not support ID, IDREF, IDREFS).

The XSD Built-in Derived data types derive from the Primitive types directly or indirectly. There are only two data types that directly derive from the primitive data types: *integer* derives from *decimal* and *normalizedString* derives from *string.* Eight data types derive from *normalizedString* and twelve data types derive from *integer.*

Each data type has a certain number of facets that can be restricted to perform additional validations on the value. Data types derive from *string* has a different set of facets than the data types that derive from *decimal.*

CHAPTER 10
COMPLEX TYPES

In the previous chapters we covered a few examples of complex types. However, we have not examined them in detail. In this chapter we will have a detailed look into complex types. We will discuss the following:

- Overview of complex types and simple types
- Complex Types – Local and Global
- Content Model of complex types
- Order Indicators
- Occurrence Indicators
- Element Groups

Just as we did in the previous chapters, we will do a hands-on lab at the end of this chapter.

Complex Types vs. Simple Types

We have seen several examples of Simple Types and Complex Types in the previous chapters. A complex type can have child elements and/or attributes. When an element has a Simple Type, it cannot have child elements or attributes. It can store only a text value.

Only element declarations can have Complex Types. Attribute declarations cannot have Complex Types because an attribute cannot have another element or attribute as it child.

Named Complex Types

Just as with Simple Types, Complex Types can also be declared globally. All global declarations should have a name. Global declarations should appear right under the *<xsd:schema>* element and the name should be unique (within other named complex types).

A *Named Complex Type* can be used within other Complex Types. Here is an example showing a named complex type.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Address Type -->
  <xsd:complexType name="AddressType">
    <xsd:all>
      <xsd:element name="Address"/>
      <xsd:element name="Street"/>
      <xsd:element name="City"/>
      <xsd:element name="Zip"/>
      <xsd:element name="State"/>
    </xsd:all>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.1: A named complex type.

The above example shows a complex type named *"AddressType."* This type can be used within the declaration of other complex types, just as we did with simple types. Here is an example.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Address Type -->
  <xsd:complexType name="AddressType">
    <xsd:all>
      <xsd:element name="Address"/>
      <xsd:element name="Street"/>
      <xsd:element name="City"/>
      <xsd:element name="Zip"/>
      <xsd:element name="State"/>
    </xsd:all>
  </xsd:complexType>
  <!-- Customer Information -->
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"/>
        <xsd:element name="Address" type="AddressType"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.2

Named Complex Types provide a great deal of reusability. For example, the *AddressType* we defined above can be used for validating *Billing* as well as *Shipping* information. Further, they help organize the code better and make it easier to understand and maintain.

Anonymous Complex Types

Though *Named Complex Types* provides a great extent of reusability, in real life you will find many complex types that are used only once. Hence,

sometimes you will find it easier to define them as anonymous types. The declaration of anonymous types appears within the complex type that owns the element.

Let us look at an example.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Customer Information -->
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="CustomerName"/>
        <xsd:element name="Address">
          <xsd:complexType>
            <xsd:all>
              <xsd:element name="Address"/>
              <xsd:element name="Street"/>
              <xsd:element name="City"/>
              <xsd:element name="Zip"/>
              <xsd:element name="State"/>
            </xsd:all>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.3

The above example shows an anonymous complex type. Such declarations cannot be reused. They are also called *Local* complex type declarations.

Content Model

The structure of the child elements of a Complex Type is called its *Content Model*. A complex type can have either Simple Content or Complex Content. The following XML fragment shows an element that has Simple Content.

```xml
<Phone location="Office">999 999 9999</Phone>
```

Listing 10.4: Simple Content Example.

Elements having Simple Content can store a text value and can hold attributes. However, they cannot have child elements.

When an element has Complex Content, it can be either *empty, element-only* or *mixed* type. We say an element has empty content when it does not have child elements and does not have a text value. However, it may have attributes. The following example shows an XML fragment having *empty* content.

```
<br />
```

Listing 10.5: Empty Content Example.

Another example:

```
<br someAttribute="someValue" />
```

Listing 10.6: Empty Content Example having an attribute.

An element is said to have element-only content when it has child elements – and optionally attributes, too – but doesn't hold a text value. (Its child elements may, however, hold a text value.) The following is an example of an element which has element-only content.

```
<Name>
   <First>Jacob</First>
   <Last>Sebastian</Last>
</Name>
```

Listing 10.7: Element-only content example.

```
<Name Title="Mr">
   <First>Jacob</First>
   <Last>Sebastian</Last>
</Name>
```

Listing 10.8: Element-only content example having an attribute.

The *name* element has *element-only* content model. Note that *element-only* content model allows attributes.

When an element has *mixed* content model it can store child elements, attributes and text value.

Here is an example:

```
<Email Priority="High">
  Dear <name>Jacob</name>,
  Your order has been
  shipped on <date>2008-01-01</date>
</Email>
```

Listing 10.9: Mixed content example.

Let us examine each of these content models in detail.

Simple Content

As discussed earlier, when an element has Simple Content it can store a text value and can have attributes. Complex Types, having Simple Content, are very similar to Simple Types. The only difference is that Simple Types cannot have attributes while Complex Types, having simple content, can have attributes. Here is an example of an element having Simple Content.

```
<Phone>999 999 9999</Phone>
```

Listing 10.10: Simple Content Example.

```
<Phone location="Office">999 999 9999</Phone>
```

Listing 10.11: Simple Content Example.

Both examples given above show an element having Simple Content. The first element has a text value and does not have attributes. The second example shows an element that has a text value, as well as an attribute.

How do we write the schema for an XML instance such as the above example? This is something new to us. We learned how to declare complex types that hold attributes. We also learned simple types that can take a value. But we have not seen complex types that take a value and attributes.

Well, the example above shows a Complex Type having *Simple Content.* Simple Content does not allow child elements. However, such an element can have attributes. It can store a text value as well. Let us see how to define a Complex Type having Simple Content.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:complexType>
      <xsd:simpleContent/>
```

```
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 10.12

The first step is to add a *simpleContent* element to the Complex Type.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Phone">
     <xsd:complexType>
       <xsd:simpleContent>
         <xsd:extension base="xsd:string"/>
       </xsd:simpleContent>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 10.13

We can derive a *simpleContent* by either extension or restriction. We will see extensions and restrictions in Chapter 11. For the purpose of this example, let us derive a Complex Type having Simple Content from *xsd:string*.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Phone">
     <xsd:complexType>
       <xsd:simpleContent>
         <xsd:extension base="xsd:string">
           <xsd:attribute name="location" type="xsd:string"/>
         </xsd:extension>
       </xsd:simpleContent>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 10.14

The above example derives a Complex Type from *xsd:string*. The derived type has Simple Content; hence, it can store a text value. It has an attribute, also, which stores the location of the phone number. Let us create a Schema Collection to validate this.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO
```

```
-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:complexType>
      <xsd:simpleContent>
        <xsd:extension base="xsd:string">
          <xsd:attribute name="location" type="xsd:string"/>
        </xsd:extension>
      </xsd:simpleContent>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Phone location="home">999 999 9999</Phone>'
```

Listing 10.15

New Complex Types can be derived from Complex Types having Simple Content. The derived type must also have Simple Content. It is not allowed to derive Complex Content from Simple Content. New Types can be derived from a Simple Content by either restriction or extension. We will examine Complex Type derivation in Chapter 12.

Complex Content

We just saw that Simple Content cannot store child elements. Complex Content can store child elements and attributes as well as text values. Complex Types having Complex Content can be classified into three groups.

1. Empty content
2. Element-only content
3. Mixed Content

We have seen an example of each of these content models earlier in this chapter. Now it is time to examine these content models in detail.

Empty Content

Complex Types having *empty content* are very close to the ones having *simple content*, except that they cannot store a text value. They can store zero or more attributes, but no text values. The most common example I

see is the XHTML <br/> element, which does not have an attribute or a text value.

```
<br />
```

Listing 10.16

When a Complex Type has empty content model, it cannot store a text value or child elements. However, it can have attributes. Here is an example.

```
<Phone Office="999 999 9999" Home="888 888 8888"/>
```

Listing 10.17

The *Phone* element given above is having empty content model. Let us look at the schema of it.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:complexType>
       <xsd:attribute name="Home"/>
       <xsd:attribute name="Office"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Phone Office="999 999 9999" Home="888 888 8888"/>'
```

Listing 10.18

320

Element-only content

These are Complex Types that hold elements and/or attributes, but no text values. Such elements cannot store a text value. However, they can hold child elements or attributes. We have seen several examples of such Complex Types earlier in this book.

The following example shows an element *Contact* that has two child elements, but no attributes. *Contact* is an element-only complex type.

```
<Contact>
   <Phone>999 999 9999</Phone>
   <Fax>888 888 8888</Fax>
</Contact>
```

Listing 10.19

Here is the schema that describes the above XML instance.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Phone"/>
        <xsd:element name="Fax"/>
      </xsd:sequence>
      <xsd:attribute name="Primary"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Contact>
  <Phone>999 999 9999</Phone>
  <Fax>888 888 8888</Fax>
</Contact>'
```

Listing 10.20

In the below example the *Contact* element has two attributes, but no child elements.

```
<Contact Phone="999 999 9999" Fax="888 888 8888" />
```

Listing 10.21

And here is the schema that describes the above XML instance.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:attribute name="Phone"/>
      <xsd:attribute name="Fax"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Contact Phone="999 999 9999" Fax="888 888 8888" />'
```

Listing 10.22

Here is a third variation that shows an XML element having child elements and attributes.

```
<Phone Primary="Home">
  <Home>999 999 9999</Home>
  <Office>888 888 8888</Office>
</Phone>
```

Listing 10.23

Here is the schema that describes the above XML instance.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Home"/>
        <xsd:element name="Office"/>
      </xsd:sequence>
      <xsd:attribute name="Primary"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Phone Primary="Home">
  <Home>999 999 9999</Home>
  <Office>888 888 8888</Office>
</Phone>'
```

Listing 10.24

The three examples we saw above show elements having *element-only* content.

Mixed Content

Mixed Content is a combination of Simple Content and Element-only content. When a Complex Type has Mixed Content, it can store child elements and attributes as well as text values. Let us look at an example:

```
<InvoiceNote>
  Call <name>Steve</name> on <mobile>999 999 9999</mobile> before
  shipping the order.
</InvoiceNote>
```

Listing 10.25

Do you find anything special in this XML instance? Yes, this XML instance is much different than what we have seen so far. The *InvoiceNote* element contains text values as well as child nodes.

Mixed content type makes text values more meaningful. For example, the above XML instance is more meaningful to an XML parser than the one given below.

```
<InvoiceNote>
   Call Steve on 999 999 9999 before
   shipping the order.
</InvoiceNote>
```

Listing 10.26

Though both examples contain the same information, the first example is more meaningful to an XML parser because the information can be extracted more accurately from the first example. Name of the contact person and mobile number can easily be parsed from the first example. However, it would be too complex to identify and extract the same information from the second example.

We have seen an example of Mixed Content. Now let us see how to write the schema for a Complex Type that has Mixed Content.

```
CREATE XML SCHEMA COLLECTION MixedContentTest AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="InvoiceNote">
     <xsd:complexType mixed="true">
       <xsd:sequence>
          <xsd:element name="name"/>
          <xsd:element name="mobile"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 10.27

You will notice that the only difference from *element-only* content model is the presence of the attribute *"mixed."* By setting this to *true* we indicate that the Complex Type has Mixed Content. Here is an XML instance that validates with the above schema.

```
DECLARE @x XML(MixedContentTest)
SELECT @x = '
<InvoiceNote>
```

```
   Call <name>Steve</name> on <mobile>999 999 9999</mobile> before
   shipping the order.
</InvoiceNote>'
```

Listing 10.28

There are many cases where we would come across mixed types. The best example is an XHTML document. For example, look at the following.

```
<p>
   This <i>XHTML</i> document is a good example of a
   <b>mixed</b> type. Some text may be <b>bold</b>, others
   may be <u>underlined</u> or may be <i>italicised</i>.
</p>
```

Listing 10.29

This is something very familiar to us. The <p> tag has mixed content; it can hold other child elements for text formatting.

Let us try to write the schema for the above XML instance.

```
CREATE XML SCHEMA COLLECTION XhtmlTest AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="p">
     <xsd:complexType mixed="true">
       <xsd:sequence>
         <xsd:choice minOccurs="0" maxOccurs="unbounded">
           <xsd:element name="u"/>
           <xsd:element name="i"/>
           <xsd:element name="b"/>
         </xsd:choice>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 10.30

```
DECLARE @x XML(XhtmlTest)
SELECT @x = '
<p>
   This <i>XHTML</i> document is a good example of a
   <b>mixed</b> type. Some text may be <b>bold</b>, others
   may be <u>underlined</u> or may be <i>italicised</i>.
</p>'
```

Listing 10.31

Order of Elements and Attributes

As mentioned several times in the previous chapters, a Complex Type can have other child elements as well as attributes. When a Complex Type contains elements and attributes, the attribute declarations should appear after the element declarations. For example, the following declaration is invalid.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:attribute name="CustomerNumber"/>
      <xsd:sequence>
        <xsd:element name="name"/>
        <xsd:element name="phone"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.32

Here is the correct schema.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name"/>
        <xsd:element name="phone"/>
      </xsd:sequence>
      <xsd:attribute name="CustomerNumber"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.33

Note that the attribute declaration is moved towards the bottom of the Complex Type declaration. Attribute declarations should always appear after element declarations.

Attributes of an element can appear in any order. The order/position is not significant for attributes. However, the order/position of elements is significant in XML. When you define your schema, you can specify for each Complex Type whether the children should follow a specific order or not.

Order Indicators

Order indicators are used to define the order or position of child elements within an XML node. There are times when we need the elements to appear in a specific order. For example, an order processing application might require the *OrderDate* to appear before *ShipDate* so that it can perform certain validations without going backward while parsing the XML. There may be times when you do not need the information in any specific order. XSD uses *order indicators* to specify the order in which elements should appear inside an XML node.

XSD defines the following order indicators:

* sequence
* all
* choice

We will see each of these *Order Indicators* in detail.

sequence Indicator

"sequence" indicator is used to specify that the elements should appear in exactly the same order as they are defined in the schema. The following schema specifies that child elements *Name, Phone* and *Address* should be placed in exactly the same order as they are defined in the schema. The first element should be *Name,* followed by *Phone,* and the last element should be *Address.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" />
        <xsd:element name="Phone" />
        <xsd:element name="Address" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.34

The XML value should follow exactly the same order. Here is a valid XML instance that passes the above schema validation.

```xml
<Customer>
  <Name>Jacob</Name>
  <Phone>999-999-9999</Phone>
  <Address>401, TIME SQUARE</Address>
</Customer>
```

Listing 10.35

If the XML instance does not follow the correct order, the schema validation will fail and SQL Server will not accept the XML instance.

all Indicator

"all" indicator is used to specify that the child elements can appear in any order. Here is the updated version of a previous schema that uses *"all"* indicator to specify that the elements can appear in any order.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:all>
        <xsd:element name="Name" />
        <xsd:element name="Phone" />
        <xsd:element name="Address" />
      </xsd:all>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.36

With the new version of the schema that uses "all" indicator, all six of the XML instances given below are valid.

```xml
<Customer>
  <Name>Jacob</Name>
  <Phone>999-999-9999</Phone>
  <Address>401, TIME SQUARE</Address>
</Customer>
```

Listing 10.37

```
<Customer>
  <Name>Jacob</Name>
  <Address>401, TIME SQUARE</Address>
  <Phone>999-999-9999</Phone>
</Customer>
```

Listing 10.38

```
<Customer>
  <Phone>999-999-9999</Phone>
  <Name>Jacob</Name>
  <Address>401, TIME SQUARE</Address>
</Customer>
```

Listing 10.39

```
<Customer>
  <Phone>999-999-9999</Phone>
  <Address>401, TIME SQUARE</Address>
  <Name>Jacob</Name>
</Customer>
```

Listing 10.40

```
<Customer>
  <Address>401, TIME SQUARE</Address>
  <Phone>999-999-9999</Phone>
  <Name>Jacob</Name>
</Customer>
```

Listing 10.41

```
<Customer>
  <Address>401, TIME SQUARE</Address>
  <Name>Jacob</Name>
  <Phone>999-999-9999</Phone>
</Customer>
```

Listing 10.42

> **When *all* is specified, an element cannot appear more than once. *maxOccurs* should always be 1 and *minOccurs* can be 0 or 1.**

choice Indicator

The *"choice"* indicator is used when only one element from a list of child elements should appear in the XML instance. The following schema defines an XML instance that stores payment details.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Payment">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="CashDetails" />
        <xsd:element name="CheckDetails" />
        <xsd:element name="CreditCardDetails" />
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 10.43

Out of the three elements declared under *Payment* node, only one should be present in the XML instance. All of the following are valid XML instances.

```
<Payment>
  <CashDetails>details here</CashDetails>
</Payment>
```

Listing 10.44

```
<Payment>
  <CheckDetails>detaile here</CheckDetails>
</Payment>
```

Listing 10.45

```
<Payment>
  <CreditCardDetails>detaile here</CreditCardDetails>
</Payment>
```

Listing 10.46

Occurrence Indicators

One of the basic differences between an element and attribute is that an element can appear more than once – if the schema allows. Attributes cannot appear more than once within the parent element.

There are times when you need to control the occurrence of child elements. Some elements may be mandatory, some optional and some may appear more than once. You can control the occurrence of elements by using *minOccurs* and *maxOccurs* attributes of element declaration.

The default value of *minOccurs* and *maxOccurs* is 1; hence, every element should appear EXACTLY once by default. By setting *minOccurs* to 0, you can make an element optional. The following table shows a few examples that demonstrate how to control the occurrences of elements by using *minOccurs* and *maxOccurs.*

minOccurs	maxOccurs	Result
1	1	The element is mandatory and should appear only once.
0	1	The element is optional. It may appear once but not more than once.
0	5	The element is optional and can appear up to 5 times.
1	5	The element is mandatory and can appear 5 times maximum.
0	unbounded	The element is optional and can appear any number of times.
1	unbounded	The element is mandatory and can appear any number of times
2	2	The element should appear exactly 2 times.

Element Groups

Element Groups are reusable groups that contain one or more elements. These groups can be inserted into other complex types within the same schema.

Element groups provide reusability to a certain extent. Further, they make a schema easier to modify and maintain. When modifications are required, the schema author can change in a single location and the changes will reflect at all places where the element group is referenced.

```sql
-- Drop previous schema collection
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">

  <xsd:group name="ContactInfo">
    <xsd:sequence>
      <xsd:element name="email"/>
      <xsd:element name="phone"/>
    </xsd:sequence>
  </xsd:group>

  <xsd:element name="Manager">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:group ref="ContactInfo"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SET @x = '
<Manager>
     <email>name@organization.com</email>
     <phone>123456789</phone>
</Manager>'
```

Listing 10.47: An example showing the usage of element groups.

In the above example, an element group named *ContactInfo* is created with *email* and *phone* elements. This group is then inserted into the declaration of the *Manager* element.

LAB4: Write schema for the Order Processing Application – Billing and Shipping address

In the last three labs we have developed the schema for the *OrderInfo, Order* and *Customer* elements. In this lab we will write the schema for the *Billing* and *Shipping* address.

The *Customer* element should contain billing and shipping information. Only the billing address is mandatory. If shipping address is not specified, the billing address is assumed to be the shipping location.

Billing and *Shipping* elements have the same structure and they follow the same validation rules. The following example shows how shipping and billing addresses should look.

```
<Billing City="Eugene" State="OR" Zip="97403">
   <Address>2732 Baker Blvd.</Address>
   <Street>Main st.</Street>
</Billing>
```

Listing 10.48: Example of a Billing Address.

```
<Shipping City="Eugene" State="OR" Zip="97403">
   <Address>2732 Baker Blvd.</Address>
   <Street>Main st.</Street>
</Shipping>
```

Listing 10.49: Example of a Shipping Address.

As mentioned earlier, *Billing* and *Shipping* elements should follow the same validation rules. Here are the rules that these elements should follow:

- Should have three mandatory attributes: *City, State* and *Zip.*
- The value of *City* should not be empty and should not be longer than thirty characters.
- The value of *State* should be exactly two characters long. Only letters A to Z in upper case are permitted.
- The length of *Zip* should be a five-digit number without leading zeros.
- *Address* is mandatory and should not be more than fifty characters long.
- *Street* is optional. If present, it should not be more than twenty characters long and should appear after the *Address* element.

Defining Complex Type: *AddressType*

Since the Billing and Shipping elements share the same validation rules, let us create a complex type named *AddressType* that validates address information. We can reuse this complex type while declaring Shipping and Billing elements.

Let us start writing the rules for *AddressType.* Let us start with a basic complex type declaration. *AddressType* should be a named complex type.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="AddressType"/>
</xsd:schema>
```

Listing 10.50

Let us now add the validation rules to the complex type definition.

Rule 1

AddressType should have three mandatory attributes: City, State and Zip.

Let us add the attribute declarations to the complex type. To indicate that the attributes are mandatory, let us set the *"use"* attribute to *"required."*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:complexType name="AddressType">
     <xsd:attribute name="City" use="required"/>
     <xsd:attribute name="State" use="required"/>
     <xsd:attribute name="Zip" use="required"/>
   </xsd:complexType>
</xsd:schema>
```

Listing 10.51

Rule 2

The value of City should not be empty and should not be longer than thirty characters.

Let us modify the declaration of the *City* attribute and add a restriction based on string. To restrict the length of the value, let us use the *minLength* and *maxLength* facets.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:attribute name="City" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:minLength value="1"/>
          <xsd:maxLength value="30"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="State" use="required"/>
    <xsd:attribute name="Zip" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.52

Rule 3

> **The value of State should be exactly two characters long. Only letters A to Z in upper case are permitted.**

Let us add a restriction to the *State* attribute. Let us use a pattern restriction to implement the restriction stipulated by Rule 3.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:attribute name="City" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.53

Rule 4

> **The length of Zip should be exactly five characters and should contain only digits 0 to 9; leading zeros are not allowed.**

We created a simple type named *ZipType* in Chapter 8. Let us use it to validate the zip code.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:attribute name="City" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.54

Rule 5

Address is mandatory and should not be more than fifty characters long.

Let us add an element declaration for the *Address* element. Let us add a sequence indicator and write the declaration of the address element. Elements are mandatory by default, and we don't need to specify anything to make it mandatory. Let us use *minLength* and *maxLength* restrictions to validate the length of the value.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Address">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="50"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="City" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.55

Rule 6

> **Street is optional. If present, it should not be more than twenty characters long and should appear after the Address element.**

Let us add the declaration for the *Street* element. To make it optional, let us set *minOccurs* to 0. Then let us add a restriction to specify the maximum allowed length of the value.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Address">
        <!-- other declarations here -->
      </xsd:element>
      <xsd:element name="Street" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="City" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <!-- other declarations here -->
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 10.56

We have created the complex type to validate Address information. Let us now build the schema that validates the shipping and billing address information.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Customer Element -->
```

```xml
  <xsd:element name="Customer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Billing" type="AddressType"/>
        <xsd:element name="Shipping" type="AddressType"
                     minOccurs="0"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <!-- Address Type -->
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Address">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="50"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Street" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="City" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:minLength value="1"/>
          <xsd:maxLength value="30"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
  <!-- Zip Type -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
GO
```

Listing 10.57

```sql
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Customer>
      <Billing City="Eugene" State="OR" Zip="97403">
```

```
        <Address>2732 Baker Blvd.</Address>
        <Street>Main st.</Street>
      </Billing>
      <Shipping City="Eugene" State="OR" Zip="97403">
        <Address>2732 Baker Blvd.</Address>
        <Street>Main st.</Street>
      </Shipping>
 </Customer>'
```

Listing 10.58

In the previous lab we had created the schema to validate the *OrderInfo*, *Order* and *Customer* elements. Let us merge the schema we created in this lab to the previous version and come up with a combined version that validates *OrderInfo, Order, Customer, Billing and Shipping* elements.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CustomerName"
                           minOccurs="0" maxOccurs="1">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="50"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Billing" type="AddressType"/>
              <xsd:element name="Shipping" type="AddressType"
                           minOccurs="0"/>
              <xsd:element name ="Terms">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="30 Days Credit"/>
                    <xsd:enumeration value="60 Days Credit"/>
                    <xsd:enumeration value="90 Days Credit"/>
                    <xsd:enumeration value="Against Delivery"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Contact"
```

```
                            minOccurs="1" maxOccurs="1"/>
            </xsd:sequence>
            <xsd:attribute name="CustomerNumber" use="required">
              <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                  <xsd:pattern value="[a-zA-Z]{5}"/>
                </xsd:restriction>
              </xsd:simpleType>
            </xsd:attribute>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="Items"/>
        <xsd:element name="OrderNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="InvoiceNote" minOccurs="0">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="500"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Discount" minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="OrderNumber" use="required">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[a-zA-ZO-9]{1,20}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
  <!-- Address Type -->
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Address">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="50"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Street" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="City" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:minLength value="1"/>
          <xsd:maxLength value="30"/>
        </xsd:restriction>
      </xsd:simpleType>
```

```
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
  <!-- Zip Type -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
'
GO
```

Listing 10.59

Here is an XML instance that validates with the above schema.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Order OrderNumber="20002">
  <OrderDate>2008-01-01Z</OrderDate>
  <DeliveryDate>2008-01-10T09:00:00-08:00</DeliveryDate>
  <Customer CustomerNumber="LAZYK">
    <CustomerName>Lazy K Kountry Store</CustomerName>
    <Billing City="Eugene" State="OR" Zip="97403">
      <Address>2732 Baker Blvd.</Address>
      <Street>Main st.</Street>
    </Billing>
    <Shipping City="Eugene" State="OR" Zip="97403">
      <Address>2732 Baker Blvd.</Address>
      <Street>Main st.</Street>
    </Shipping>
    <Terms>30 Days Credit</Terms>
    <Contact />
  </Customer>
  <Items />
  <OrderNote>Delivery needed before 8 AM</OrderNote>
  <InvoiceNote>
    Adjust the previous credit note with this invoice
  </InvoiceNote>
  <Discount />
</Order>'
```

Listing 10.60

341

Chapter Summary

We have taken a closer look at Complex Types in this chapter. Complex Types are more content-rich than Simple Types because they can have child elements and attributes.

The data and the structure of an element and its children constitute its *Content Model.* Complex Types can have four different content models, namely Simple, Empty, Element-Only and Mixed.

All of the four content models mentioned above can have attributes. The only difference is in terms of the ability to hold child elements and character content.

Elements having *Simple Content* can store character data as well as child elements. They are very close to Simple Types but can hold attributes, which Simple Types cannot. Elements having empty content can have attributes but no child elements. Elements having element-only content can hold child elements and attributes, but they cannot store character data. When an element has *Mixed* type it can store child elements, attributes and character data.

Complex Types can be Named or Anonymous. Named Types are defined globally and can be re-used within other complex types. Anonymous types are declared inline within the body of other complex types and the declaration can be used only once.

Sequence indicator is used to control the order of the child elements within the XML instance. When elements are declared within a *sequence* indicator, the XML instance must contain elements in the same order as declared in the *sequence* indicator. If a specific order is not required, the *all* indicator should be used. When *all* is used child elements can appear in any order and none of the elements should appear more than once. *Choice* indicator can be used to specify that only one element from a given list of elements should appear in the XML instance.

Elements are mandatory by default. An element can occur more than once if the schema permits. The occurrences of elements are controlled by *minOccurs* and *maxOccurs* attributes. The default value of *minOccurs* and *maxOccurs* is 1; hence, all elements should appear exactly once unless specified differently. By setting *minOccurs* to 0, you can make an element optional. By setting *maxOccus* to *"unbounded"* you can allow an element to appear an unlimited number of times.

When a complex type contains elements and attributes, attribute declarations should appear after the element declarations. Element groups can be used to define frequently used elements. A named element group can be created with such declarations and other complex types can refer to a named element group.

CHAPTER 11
COMPLEX TYPE DERIVATION

In Chapter 8 we have seen how to derive new types from Simple types. We discussed the different ways to derive a new type from an existing simple type. In Chapter 10 we examined complex types and discussed the different content models supported by complex types. Just as we saw with simple types, new types can be derived from complex types, too.

New complex types can be derived from existing complex types by restriction or by extension. In this chapter we will discuss the following:

- Deriving complex types from simple types
- Deriving from simple content complex types
- Deriving from empty content complex types
- Deriving from element-only content complex types
- Deriving from mixed content complex types
- Controlling type derivation of complex types

Deriving Complex Types from Simple Types

We have examined simple types in many of the previous chapters. When we examined the *final* attribute of simple type declaration, I had mentioned that a complex type can be derived from a simple type by extension. Let us look at an example in order to understand this.

Let us create a simple type that validates a phone number.

```
xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Type -->
  <xsd:simpleType name="PhoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 11.1: A simple type that validates a phone number.

We know how to derive other simple types by restriction, by union or by list. But we have not seen deriving complex types from simple types.

Let us try to derive a new complex type from this. Let us add a new attribute to *PhoneType* so that it can take an additional property that specifies whether the value refers to *"work"* or *"home"* phone.

When you derive a complex type from a simple type, it can only have a simple content. Let us start by adding a simple content element to the element definition.

```
<xsd:complexType name="PhoneTypeEx">
   <xsd:simpleContent/>
</xsd:complexType>
```

Listing 11.2

Next, we need to add an extension element that derives from *PhoneType.*

```
<xsd:complexType name="PhoneTypeEx">
   <xsd:simpleContent>
     <xsd:extension base="PhoneType"/>
   </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.3

And finally, let us add the *Type* attribute to it.

```
<xsd:complexType name="PhoneTypeEx">
   <xsd:simpleContent>
     <xsd:extension base="PhoneType">
       <xsd:attribute name="Type" use="required"/>
     </xsd:extension>
   </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.4

Let us create a schema collection with this definition.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO
```

```sql
-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone" type="PhoneTypeEx"/>
  <!-- Extended Phone Type -->
  <xsd:complexType name="PhoneTypeEx">
    <xsd:simpleContent>
      <xsd:extension base="PhoneType">
        <xsd:attribute name="Type" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
  <!-- Phone Type -->
  <xsd:simpleType name="PhoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Phone Type="home">999-999-9999</Phone>'
```

Listing 11.5

Restricting Extension of Simple Types

In Chapter 8 we saw the *"final"* attribute of simple type declaration. This attribute can be used to restrict the methods by which the type can be inherited. If *"final"* is set to *"list,"* derivation by list will not be allowed. Similarly, by setting *"final"* to *"union"* or *"restriction"* we could prohibit derivation by *union* and *list,* respectively.

When the *final* attribute is set to *"extension,"* the simple type cannot be extended. In the previous example we extended *PhoneType* and created *PhoneTypeEx*. Now let us write a different version of *PhoneType* that prevents extension.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone" type="PhoneTypeEx"/>
  <!-- Extended Phone Type -->
  <xsd:complexType name="PhoneTypeEx">
    <xsd:simpleContent>
      <xsd:extension base="PhoneType">
        <xsd:attribute name="Type" use="required"/>
      </xsd:extension>
    </xsd:simpleContent>
```

```
   </xsd:complexType>
   <!-- Phone Type -->
   <xsd:simpleType name="PhoneType" final="extension">
     <xsd:restriction base="xsd:string">
       <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
     </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

Listing 11.6: When "final" is set to "extension," the type cannot be derived by extension.

Since the *PhoneType* does not allow extension, the definition of *PhoneTypeEx* is invalid. If you try to create a schema collection with this version of the schema, SQL Server will generate the following error.

```
Invalid type definition for type 'PhoneTypeEx', the derivation was
illegal because 'final' attribute was specified on the base type
```

Deriving from Complex Types

We just saw how to derive a complex type from a simple type. In Chapter 10 we learned that complex types can have four different content models, namely simple content, empty content, mixed content and element-only content. The derivation behavior of each content model is slightly different from others. Let us examine how to derive new complex types from the four different content models.

Throughout this chapter I will be referring to the different content models supported by complex types often. I will be using *element-only type, element-only content type, and element only content complex type* to refer to a complex type having element only content type. I will use the same naming pattern to refer to the other content models, as well.

Deriving from Simple Content

We have learned simple content in Chapter 10. An element having simple content can store a text value as well as attributes. However, it cannot have child elements. The following example shows an XML element having simple content.

```
<Phone Type="Home" CallOnWeekend="true">999-999-9999</Phone>
```

Listing 11.7

And here is the schema that describes this XML instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="PhoneType"/>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="Type" type="xsd:string"/>
        <xsd:attribute name="CallOnWeekend" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>
```

Listing 11.8

The above example shows a complex type having simple content. There are two ways to derive a new type from this. We can derive a new type by restriction or by extension. Let us examine both options in detail.

Deriving from Simple Content by Restriction

When we derive by restriction from a complex type having simple content, we can apply the following restrictions.

- We can add restrictions to the content/text-value of the element.
- We can add restrictions to the attributes
- We can remove one or more of the attributes.

Let us derive a new type from complex type shown above and try to add the following restrictions.

- At present there is no validation on the format of the phone number. Let us add a pattern restriction to restrict values to the format *"999-999-9999."*
- The *Type* attribute does not have any validation. Let us restrict it to *"Home"* and *"Work"* by using an enumeration restriction.
- Let us remove the attribute *"CallOnWeekend."*

First of all, let us derive a new type from *PhoneType* by restriction.

348

```
<!-- Restricted Phone Type -->
<xsd:complexType name="RestrictedPhoneType">
  <xsd:simpleContent>
    <xsd:restriction base="PhoneType"/>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.9

Now let us add a pattern restriction to validate the content of the element.

```
<!-- Restricted Phone Type -->
<xsd:complexType name="RestrictedPhoneType">
  <xsd:simpleContent>
    <xsd:restriction base="PhoneType">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.10

Let us add restrictions to the *"Type"* attribute. Let us use an enumeration restriction to restrict the values to *"Home"* and *"Work"* only.

```
<!-- Restricted Phone Type -->
<xsd:complexType name="RestrictedPhoneType">
  <xsd:simpleContent>
    <xsd:restriction base="PhoneType">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
      <xsd:attribute name="Type">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:enumeration value="Home"/>
            <xsd:enumeration value="Work"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.11

Finally, let us remove the attribute *"CallOnWeekend."*

```
<!-- Restricted Phone Type -->
<xsd:complexType name="RestrictedPhoneType">
  <xsd:simpleContent>
    <xsd:restriction base="PhoneType">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
      <xsd:attribute name="Type">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
```

```
            <xsd:enumeration value="Home"/>
            <xsd:enumeration value="Work"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="CallOnWeekend" type="xsd:boolean"
                     use="prohibited"/>
    </xsd:restriction>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.12

Note the usage of *"prohibited."* This is the example I mentioned when we discussed attribute declarations in Chapter 6. When you derive a new complex type, all the attributes of the base type will pass down to the derived type. If you want to eliminate one or more attributes, you need to redefine them in the child type and set the *"use"* attribute to *"prohibited."*

Let us create a schema collection with this code.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="RestrictedPhoneType"/>

  <!-- Restricted Phone Type -->
  <xsd:complexType name="RestrictedPhoneType">
    <xsd:simpleContent>
      <xsd:restriction base="PhoneType">
        <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
        <xsd:attribute name="Type">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:enumeration value="Home"/>
              <xsd:enumeration value="Work"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
        <xsd:attribute name="CallOnWeekend" type="xsd:boolean"
                       use="prohibited"/>
      </xsd:restriction>
    </xsd:simpleContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
```

```
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="Type" type="xsd:string"/>
        <xsd:attribute name="CallOnWeekend" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>'
GO
```

Listing 11.13

Since we have marked the attribute *"CallOnWeekend"* as *"prohibited,"* SQL Server will generate an error if the XML instance contains this attribute. The following XML instance will not be validated.

```
DECLARE @x XML(ExampleSchema)
SET @x = '
<Phone Type="Home" CallOnWeekend="true">999-999-9999</Phone>'
```

Listing 11.14

The validation of the above XML instance will fail with the following error.

```
XML Validation: Attribute 'CallOnWeekend' is not permitted in this
context. Location: /*:Phone[1]/@*:CallOnWeekend
```

Note that when you add restrictions to the content as well as attributes, you cannot make the values less restrictive. We have discussed this when we discussed derivation of simple types. Similarly, if an attribute is *"required"* in the base type, you cannot make it *"optional"* in the derived type. However, if the attribute is declared with *"optional"* in base type, you can make it *"required"* in the derived type.

If an attribute is declared as *"required"* in base type, it should be *"required"* in the derived type as well. If an attribute is *"required"* in base type, the derived type cannot make it *"prohibited."*

Base Type	Derived Type		
Attribute	**Optional**	**Required**	**Prohibited**
Optional	Yes	Yes	Yes
Required	No	Yes	No
Prohibited	No	No	Yes

Listing 11.15

Deriving from Simple Content by Extension

When you derive by extension from a complex type with simple content the result will always be a simple content. You cannot create a new complex content type from a simple content type. The only operation you can do when you extend a simple content type is to add new attributes.

Let us take the example of the *PhoneType* we saw in *Listing 11.8* and derive a new type from it by extension. Let us add a new attribute named *"CallOnHolidays."*

First of all, let us create a new type and add an extension element having base type set to *PhoneType.*

```xml
<!-- Extended Phone Type -->
<xsd:complexType name="ExtendedPhoneType">
  <xsd:simpleContent>
    <xsd:extension base="PhoneType"/>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.16

Then let us add the new attribute.

```xml
<!-- Extended Phone Type -->
<xsd:complexType name="ExtendedPhoneType">
  <xsd:simpleContent>
    <xsd:extension base="PhoneType">
      <xsd:attribute name="CallOnHolidays" type="xsd:string"/>
    </xsd:extension>
  </xsd:simpleContent>
</xsd:complexType>
```

Listing 11.17

Let us create a schema collection and validate an XML instance with this version of the schema.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="ExtendedPhoneType"/>

  <!-- Extended Phone Type -->
  <xsd:complexType name="ExtendedPhoneType">
    <xsd:simpleContent>
      <xsd:extension base="PhoneType">
        <xsd:attribute name="CallOnHolidays" type="xsd:string"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:simpleContent>
      <xsd:extension base="xsd:string">
        <xsd:attribute name="Type" type="xsd:string"/>
        <xsd:attribute name="CallOnWeekend" type="xsd:boolean"/>
      </xsd:extension>
    </xsd:simpleContent>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '
<Phone Type="Home" CallOnWeekend="true" CallOnHolidays="true">
    999-999-9999
</Phone>'
```

Listing 11.18

Deriving from Element-only Content

We have discussed element-only content model in Chapter 10. A complex type with element-only content model can have child elements and attributes. However, it cannot store a text value as a simple content element can.

The following example shows an xml instance having element-only content.

```xml
<Contact Name="Jacob" Title="Manager">
  <Phone>999-888-7777</Phone>
  <Email>jacob@jacob.com</Email>
</Contact>
```

Listing 11.19

The *Contact* element given in the above example has element-only content model. It has an attribute as well as child elements. *Phone* and *Email* are simple content elements. Here is the schema that describes the above XML instance.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="ContactType"/>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType">
    <xsd:sequence>
      <xsd:element name="Phone" minOccurs="0"/>
      <xsd:element name="Email" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name"/>
    <xsd:attribute name="Title"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 11.20

A complex type with element-only content can be extended or restricted to create new types.

Deriving from Element-only content by Restriction

When deriving from an element-only content type, we could create either an empty content complex type or another element-only content complex type.

The most common usage of deriving by restriction from an element-only content type is to remove one or more elements or attributes.

Deriving an element-only complex type from an element-only complex type

When deriving by restriction from an element-only complex type, you can add restrictions to the base type's elements and attributes as well as remove one or more elements and attributes.

Let us derive a new type from *ContactType* and remove the *"Email"* element and *"Title"* attribute. Further, let us add a pattern restriction to the *Phone* element and add a length restriction to the *Name* attribute.

First of all, let us create a new complex type that derives from *ContactType* by restriction.

```xsd
<!-- Restricted Contact Type -->
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType"/>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.21

Now let us add the element declarations. We need to remove the *Email* element and keep only the *Phone* element.

```xsd
<!-- Restricted Contact Type -->
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType">
      <xsd:sequence>
        <xsd:element name="Phone"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.22

Elements from the base type are not passed down to the derived type by default. The derived type has to explicitly declare the elements it wants to inherit from the parent (rather than list those to exclude.).

In the above example the derived type has declared only the *Phone* element, and as a result its content model will have only one element. This would result in the elimination of the *Email* element declared in the base type.

Note that *Email* element is declared in base type as optional. If it were mandatory, we would not have been able to eliminate it from the derived type. All mandatory elements in the base type must exist in the derived type as well.

If the elements in the base type are declared within a *sequence* group, the derived type cannot change it to *all* or *choice* groups. If the base type has *all*, the derived type can change it to *sequence.* If the base type has *choice,* the derived type must also have *choice.*

Let us now add a format restriction to the *Phone* element.

```xsd
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType">
      <xsd:sequence>
        <xsd:element name="Phone">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.23

The next step is to remove the *"Title"* attribute.

```xsd
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType">
      <xsd:sequence>
        <xsd:element name="Phone">
          <!-- Removed for brevity -->
        </xsd:element>
      </xsd:sequence>
      <xsd:attribute name="Title" use="prohibited"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.24

All attributes of the base type are passed down to the derived type by default. Hence, if you want to remove an attribute you need to re-declare it in the derived type as *"prohibited."* A derived type cannot eliminate an attribute declared as mandatory in the base type. Listing *11.15* explains the rules to be followed while deriving attributes.

You need to redefine an attribute in the derived type only if you want to restrict the value space of the attribute. In the above example, the *Name* attribute is not redefined. When not redefined in the derived type, the attributes of the base type will exist in the derived type with the same validation rules defined in the base type. We discussed earlier that we need to add a length restriction to the *Name* attribute. To achieve this, we need to redefine the *Name* attribute in the derived type.

```xml
<!-- Restricted Contact Type -->
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType">
      <!-- Removed for brevity -->
      <xsd:attribute name="Title" use="prohibited"/>
      <xsd:attribute name="Name">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.25

Let us build a schema collection with this definition.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="RestrictedContactType"/>

  <!-- Restricted Contact Type -->
  <xsd:complexType name="RestrictedContactType">
    <xsd:complexContent>
      <xsd:restriction base="ContactType">
        <xsd:sequence>
          <xsd:element name="Phone">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
```

```
        </xsd:sequence>
        <xsd:attribute name="Title" use="prohibited"/>
        <xsd:attribute name="Name">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:maxLength value="20"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType">
    <xsd:sequence>
      <xsd:element name="Phone" minOccurs="0"/>
      <xsd:element name="Email" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name" />
    <xsd:attribute name="Title"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Contact Name="Jacob">
  <Phone>999-888-7777</Phone>
</Contact>'
```

Listing 11.26

Deriving an empty content complex type from an element-only complex type

In the above example, we saw how to derive an element-only complex type from a base type having element-only content model. It is also possible to derive an empty content complex type from an element-only content type. An empty content type can hold attributes, but no child elements or text value. Let us see an example.

```
<!-- Restricted Contact Type -->
<xsd:complexType name="RestrictedContactType">
  <xsd:complexContent>
    <xsd:restriction base="ContactType"/>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.27

Note that the *restriction* element is left empty. This indicates that the derived type does not want to inherit any of the elements declared in the base type. We saw earlier that elements from the base type are not passed down to the derived type. The content model of the derived type is composed of only the elements redefined in the derived type. By keeping the *restriction* element empty, we can derive an empty content type.

This does not affect the attributes. Attributes of the base type are always passed down to the derived type. If you want to remove one or more attributes, you need to redefine them in the derived type and set the *"use"* attribute to *"prohibited."*

You can derive an empty content type from an element only content type only if all the elements in the base type are optional.

Let us create a schema collection with this version of the schema.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="RestrictedContactType"/>

  <!-- Restricted Contact Type -->
  <xsd:complexType name="RestrictedContactType">
    <xsd:complexContent>
      <xsd:restriction base="ContactType"/>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType">
    <xsd:sequence>
      <xsd:element name="Phone" minOccurs="0"/>
      <xsd:element name="Email" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name"/>
    <xsd:attribute name="Title"/>
  </xsd:complexType>
</xsd:schema>'
GO
```

```
-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Contact Name="Jacob"/>'
```

Listing 11.28

Deriving from Element-only content by Extension

Usually, complex types are extended to add new elements and attributes. When you derive a new type from an element-only type by extension, you can add new elements and attributes.

Let us take the example of *ContactType* we discussed earlier and add a *Fax* element to it. Let us also add a new attribute named *"Department."*

```
<!-- Extended Contact Type -->
<xsd:complexType name="ExtendedContactType">
  <xsd:complexContent>
    <xsd:extension base="ContactType">
      <xsd:sequence>
        <xsd:element name="Fax"/>
      </xsd:sequence>
      <xsd:attribute name="Department"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.29

When deriving by extension, all elements and attributes in the base type will be passed down to the derived type by default. You don't need to redefine them in the derived type. The new elements will be added to the bottom of the elements in the base type.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="ExtendedContactType"/>
```

```
  <!-- Extended Contact Type -->
  <xsd:complexType name="ExtendedContactType">
    <xsd:complexContent>
      <xsd:extension base="ContactType">
        <xsd:sequence>
          <xsd:element name="Fax"/>
        </xsd:sequence>
        <xsd:attribute name="Department"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType">
    <xsd:sequence>
      <xsd:element name="Phone"/>
      <xsd:element name="Email" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name" use="required"/>
    <xsd:attribute name="Title"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Contact Name="Jacob" Title="Manager" Department="IT">
  <Phone>999-888-7777</Phone>
  <Email>jacob@jacob.com</Email>
  <Fax>888-999-3333</Fax>
</Contact>'
```

Listing 11.30

The derived type of a complex type with element-only content model will be another element-only type. You cannot derive a mixed content or empty content type from an element-only type.

Deriving from Mixed Content Complex Types

You can derive new complex types from a mixed content complex type by restriction or by extension. A mixed type or element-only content type can be derived from a mixed type by restriction. However, when deriving by extension only a mixed type can be created.

Here is an example of a mixed type. We saw this example in Chapter 10 when we discussed mixed types.

```
<InvoiceNote Priority="High">
  Call <name>Steve</name> on <mobile>999-999-9999</mobile> before
  shipping the order.
</InvoiceNote>
```

Listing 11.31

Here is the schema that describes the above XML instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
  <xsd:element name="InvoiceNote" type="NoteType"/>

  <!-- Invoice Note Type -->
  <xsd:complexType name="NoteType" mixed="true">
    <xsd:sequence>
      <xsd:element name="name" minOccurs="0"/>
      <xsd:element name="mobile" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Priority"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 11.32

Derivation of complex types with mixed content is very similar to the derivation of element-only types. You can derive a new type either by extension or by restriction.

Deriving from Mixed Content Type by Restriction

We saw earlier that only an element-only content type can be derived from a complex type having element-only content. However, when deriving by restriction from a mixed type, you can derive a mixed type, element-only type or empty content type.

Deriving a mixed content type from a mixed content complex type

For the purpose of this example, let us derive a new mixed content complex type from *NoteType* by restriction. Let us remove the *mobile* element from the derived type.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
```

```
  <xsd:element name="InvoiceNote" type="RestrictedNoteType"/>

  <!-- Restricted Note Type -->
  <xsd:complexType name="RestrictedNoteType" mixed="true">
    <xsd:complexContent>
      <xsd:restriction base="NoteType">
        <xsd:sequence>
          <xsd:element name="name"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Note Type -->
  <xsd:complexType name="NoteType" mixed="true">
    <xsd:sequence>
      <xsd:element name="name" minOccurs="0"/>
      <xsd:element name="mobile" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Priority"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 11.33

This is very similar to the example we saw when we discussed element-only content types. The only difference is the presence of the *mixed* attribute, which makes the type a *mixed type*.

Let us create a schema collection with the above schema.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
  <xsd:element name="InvoiceNote" type="MixedNoteType"/>

  <!-- Restricted Note Type -->
  <xsd:complexType name="MixedNoteType" mixed="true">
    <xsd:complexContent>
      <xsd:restriction base="NoteType">
        <xsd:sequence>
          <xsd:element name="name"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>
```

```
   <!-- Note Type -->
   <xsd:complexType name="NoteType" mixed="true">
     <xsd:sequence>
       <xsd:element name="name" minOccurs="0"/>
       <xsd:element name="mobile" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="Priority"/>
   </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<InvoiceNote Priority="High">
  Call <name>Steve</name> on 999-999-9999 before
  shipping the order.
</InvoiceNote>'
```

Listing 11.34

Note that you can eliminate a base type element only if it is declared as optional in the base type. For example, if you set the *mobile* attribute as mandatory in the base type and try to eliminate it in the derived type, SQL Server will generate the following error.

```
Invalid restriction for type 'RestrictedNoteType'. Invalid model
group restriction.
```

The rules of derivation are similar to the rules of element-only content model that we examined earlier in this chapter. Elements declared in the parent type do not pass down to the derived type. You need to explicitly declare the elements you want to include in the derived type.

Deriving an element-only content type from a mixed-type

When deriving by restriction from a mixed type, you can also create an element-only type. This could be achieved by simply removing the *"mixed"* attribute from the complex type declaration.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
```

```
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
  <xsd:element name="InvoiceNote" type="ElementOnlyNoteType"/>

  <!-- Restricted Note Type -->
  <xsd:complexType name="ElementOnlyNoteType">
    <xsd:complexContent>
      <xsd:restriction base="NoteType">
        <xsd:sequence>
          <xsd:element name="name"/>
        </xsd:sequence>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Note Type -->
  <xsd:complexType name="NoteType" mixed="true">
    <xsd:sequence>
      <xsd:element name="name" minOccurs="0"/>
      <xsd:element name="mobile" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Priority"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<InvoiceNote Priority="High">
  <name>Steve</name>
</InvoiceNote>'
```

Listing 11.35

Note that the only difference is the absence of the *mixed* attribute. This schema and XML instance is presented only for understanding how type derivation works. The actual validation rules and the information stored in the above XML instances are not very meaningful. They are used only for the purpose of understanding type derivation.

Deriving an empty-content type from a mixed-type

When deriving from a mixed type by restriction, we could create an empty content type if all the elements in the base type are optional. To understand this, let us derive a new empty-content type from *NoteType*.

```
<!-- Deriving an Empty Content -->
<xsd:complexType name="EmptyNoteType">
  <xsd:complexContent>
    <xsd:restriction base="NoteType"/>
  </xsd:complexContent>
```

```
    </xsd:complexType>
```

Listing 11.36

Just as in the case with element-only content types, by keeping the restriction element empty we could eliminate all the elements declared in the base type (if those elements are declared optional in the base type). To eliminate attributes, you need to declare them in the derived type with *prohibited* attribute. You can eliminate an attribute only if it is declared as optional in the base type. The schema in the following example restricts the *priority* attribute, also.

```
<!-- Deriving an Empty Content -->
<xsd:complexType name="EmptyNoteType">
  <xsd:complexContent>
    <xsd:restriction base="NoteType">
      <xsd:attribute name="Priority" use="prohibited"/>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.37

Here is a schema collection that validates an XML instance using the above schema.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
  <xsd:element name="InvoiceNote" type="EmptyNoteType"/>

  <!-- Deriving an Empty Content -->
  <xsd:complexType name="EmptyNoteType">
    <xsd:complexContent>
      <xsd:restriction base="NoteType">
        <xsd:attribute name="Priority" use="prohibited"/>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Invoice Note Type -->
  <xsd:complexType name="NoteType" mixed="true">
    <xsd:sequence>
```

```
        <xsd:element name="name" minOccurs="0"/>
        <xsd:element name="mobile" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="Priority"/>
   </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<InvoiceNote/>'
```

Listing 11.38

Deriving simple content from mixed-type

Though the XSD specification allows deriving a simple content from a mixed-content type, SQL Server does not allow that. The following schema is valid per XSD specification.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <!-- Invoice Note Element -->
   <xsd:element name="InvoiceNote" type="SimpleNoteType"/>

   <!-- Simple Content Note Type -->
   <xsd:complexType name="SimpleNoteType">
     <xsd:simpleContent>
       <xsd:restriction base="NoteType">
         <xsd:simpleType>
           <xsd:restriction base="xsd:string"/>
         </xsd:simpleType>
       </xsd:restriction>
     </xsd:simpleContent>
   </xsd:complexType>

   <!-- Note Type -->
   <xsd:complexType name="NoteType" mixed="true">
     <xsd:sequence>
       <xsd:element name="name" minOccurs="0"/>
       <xsd:element name="mobile" minOccurs="0"/>
     </xsd:sequence>
     <xsd:attribute name="Priority"/>
   </xsd:complexType>
</xsd:schema>
```

Listing 11.39

The above schema tries to derive a simple content from a mixed type. Though this is valid per XSD specification, SQL Server will generate an error if you try to create a schema collection with this definition.

```
Derivation with both a 'base' attribute and an embedded type
definition is not supported in this release
```

Deriving from Mixed Content by Extension

We saw earlier that when deriving from mixed content complex type by restriction, we could create either a mixed content type or an element-only content type. However, when deriving by extension, we could create only a mixed content complex type.

Let us derive a new type from *NoteType* by extension and add an *email* element.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Invoice Note Element -->
  <xsd:element name="InvoiceNote" type="ExtendedNoteType"/>

  <!-- Extended Note Type -->
  <xsd:complexType name="ExtendedNoteType" mixed="true">
    <xsd:complexContent>
      <xsd:extension base="NoteType">
        <xsd:sequence>
          <xsd:element name="email"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Note Type -->
  <xsd:complexType name="NoteType" mixed="true">
    <xsd:sequence>
      <xsd:element name="name"/>
      <xsd:element name="mobile"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<InvoiceNote>
  Call <name>Steve</name> on <mobile>999 999 9999</mobile> or
```

```
   <email>steve@somedomain.com</email> before shipping the order.
</InvoiceNote>'
```

Listing 11.40

Deriving from Empty Content

Just as with all other types we discussed in this chapter, you can derive a new type from an empty content element by restriction or by extension.

Here is an example showing an empty content type.

```
<Phone Work="999 999 9999" Home="888 888 8888"/>
```

Listing 11.41

Here is the schema that describes the above XML instance.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="PhoneType"/>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:attribute name="Home"/>
    <xsd:attribute name="Work"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 11.42

We will use this example in the sections below to understand how type derivation works with empty content types.

Deriving from Empty Content by Restriction

Since an empty content complex type can have only attributes, the only purpose of deriving from an empty content type by restriction is to eliminate attributes or add additional validations to the attributes.

Let us derive a new type from *PhoneType* by restriction. Let us remove the *Work* attribute from the derived type and add a pattern restriction to the *Home* attribute.

```xml
<!-- RestrictedPhoneType -->
<xsd:complexType name="RestrictedPhoneType">
  <xsd:complexContent>
    <xsd:restriction base="PhoneType">
      <xsd:attribute name="Work" use="prohibited"/>
      <xsd:attribute name="Home">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.43

You can eliminate an attribute from the derived type only if it is optional in the base type.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="RestrictedPhoneType"/>

  <!-- RestrictedPhoneType -->
  <xsd:complexType name="RestrictedPhoneType">
    <xsd:complexContent>
      <xsd:restriction base="PhoneType">
        <xsd:attribute name="Office" use="prohibited"/>
        <xsd:attribute name="Home">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:attribute>
      </xsd:restriction>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:attribute name="Home"/>
    <xsd:attribute name="Work"/>
  </xsd:complexType>
</xsd:schema>'
```

```
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Phone Home="888-888-8888"/>'
```

Listing 11.44

Deriving from Empty Content by Extension

When you derive from an empty content complex type by restriction, you can create only an empty content type. However, when you derive by extension you can create an empty type, mixed type or element only content type.

Deriving from Empty Content by Extension – Creating an Empty Content type

When deriving new Empty Content by extension, you can add one or more attributes to the base type. Let us add a new attribute to the *PhoneType*.

```
<!-- EmptyPhoneType -->
<xsd:complexType name="EmptyPhoneType">
  <xsd:complexContent>
    <xsd:extension base="PhoneType">
      <xsd:attribute name="Mobile"/>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.45

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
```

```
  <xsd:element name="Phone" type="EmptyPhoneType"/>

  <!-- EmptyPhoneType -->
  <xsd:complexType name="EmptyPhoneType">
    <xsd:complexContent>
      <xsd:extension base="PhoneType">
        <xsd:attribute name="Mobile"/>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:attribute name="Home"/>
    <xsd:attribute name="Work"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Phone
  Home="888-888-8888"
  Work="777-777-7777"
  Mobile="666-666-6666"/>'
```

Listing 11.46

Deriving from Empty Content by Extension – Creating an element only content type

Let us see how to derive an element only content type from an empty content type by extension. In the previous example, we added a new attribute *Mobile* to the *PhoneType*. Let us change that schema and add it as an element so that it results in an element only content type. This example is purely for the purpose of understanding. In real life it makes sense to keep *Mobile* as an attribute if *Home* and *Work* are presented as attributes.

```
<!-- EmptyPhoneType -->
<xsd:complexType name="EmptyPhoneType">
  <xsd:complexContent>
    <xsd:extension base="PhoneType">
      <xsd:sequence>
        <xsd:element name="Mobile"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.47

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="EmptyPhoneType"/>

  <!-- EmptyPhoneType -->
  <xsd:complexType name="EmptyPhoneType">
    <xsd:complexContent>
      <xsd:extension base="PhoneType">
        <xsd:sequence>
          <xsd:element name="Mobile"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:attribute name="Home"/>
    <xsd:attribute name="Work"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Phone Home="888-888-8888" Work="777-777-7777">
  <Mobile>666-666-6666</Mobile>
</Phone>'
```

Listing 11.48

Deriving from Empty Content by Extension – Creating a Mixed content type

It is also possible to derive a mixed content type from an empty content type. The following example shows a mixed type derived from *PhoneType* by extension.

```
<!-- EmptyPhoneType -->
<xsd:complexType name="EmptyPhoneType" mixed="true">
  <xsd:complexContent>
    <xsd:extension base="PhoneType">
      <xsd:sequence>
```

```
        <xsd:element name="Mobile"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

Listing 11.49

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Phone Element -->
  <xsd:element name="Phone" type="EmptyPhoneType"/>

  <!-- EmptyPhoneType -->
  <xsd:complexType name="EmptyPhoneType" mixed="true">
    <xsd:complexContent>
      <xsd:extension base="PhoneType">
        <xsd:sequence>
          <xsd:element name="Mobile"/>
        </xsd:sequence>
      </xsd:extension>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Phone Type -->
  <xsd:complexType name="PhoneType">
    <xsd:attribute name="Home"/>
    <xsd:attribute name="Work"/>
  </xsd:complexType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Phone Home="888-888-8888" Work="777-777-7777">
  If not available at home or work, call me on my
  <Mobile>666-666-6666</Mobile>.
</Phone>'
```

Listing 11.50

Complex Type Derivation Summary

New complex types can be created from complex types by extension and restriction. We have seen how to extend the four different content types earlier in this chapter. The following table summarizes the type derivation support extended by each content model.

Base Content Type	Derived Content Type							
	Simple		Element Only		Mixed		Empty	
	Rest	Ext	Rest	Ext	Rest	Ext	Rest	Ext
Simple	Yes	Yes						
Element Only			Yes	Yes			Yes	
Mixed			Yes		Yes	Yes	Yes	
Empty				Yes		Yes	Yes	Yes

Listing 11.51

The table shows the different content models you can derive from a given base type. It also shows the derivation method (extension or restriction) to be used to create a given content type from a base type.

Controlling Complex Type Derivation

There may be times when you want to control the derivation of a given complex type. For example, you might have created a complex type that you don't want others to extend. Or you might not want others to restrict a certain complex type. The derivation of a complex type can be controlled by using the *final* attribute of *Complex Type Declaration.*

The *final* attribute of a complex type declaration can take one of the following three values.

1. restriction
2. extension
3. #all

Preventing derivation by *restriction*

By setting the *final* attribute to *"restriction,"* you can prevent derivation by restriction. The following code snippet shows an example.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="RestrictedContactType"/>

  <!-- Restricted Contact Type -->
  <xsd:complexType name="RestrictedContactType">
    <xsd:complexContent>
      <xsd:restriction base="ContactType"/>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType" final="restriction">
    <xsd:sequence>
      <xsd:element name="Phone" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name" />
  </xsd:complexType>
</xsd:schema>
```

Listing 11.52

In the above example *ContactType* is declared with *final restriction,* and you cannot derive a new complex type from it by restriction. If you try to create a schema collection with this schema definition, SQL Server will generate the following error.

```
Invalid type definition for type 'RestrictedContactType', the
derivation was illegal because 'final' attribute was specified on
the base type
```

The *ContactType* in the above example prevents only derivation by restriction. You can still derive a new type by extension.

Preventing derivation by *extension*

By setting the *final* attribute to *"extension,"* you can prevent derivation by extension. Let us look at an example.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Contact Element -->
  <xsd:element name="Contact" type="ExtendedContactType"/>

  <!-- Extended Contact Type -->
  <xsd:complexType name="ExtendedContactType">
    <xsd:complexContent>
      <xsd:extension base="ContactType"/>
    </xsd:complexContent>
  </xsd:complexType>

  <!-- Contact Type -->
  <xsd:complexType name="ContactType" final="extension">
    <xsd:sequence>
      <xsd:element name="Phone" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="Name" />
  </xsd:complexType>
</xsd:schema>
```

Listing 11.53

The above schema is invalid because *ContactType* is declared with *final extension* and it cannot be extended further. If you try to create a schema collection with this schema, SQL Server will generate the following error.

```
Invalid type definition for type 'ExtendedContactType', the
derivation was illegal because 'final' attribute was specified on
the base type
```

The above schema prevents only derivation by extension. You can still derive a new type by restriction from *ContactType.*

Preventing derivation completely

We have seen how to prevent type derivation by restriction or by extension. There may be times when you want to protect a complex type from derivation completely. You can achieve this by setting the *final* attribute to *"#all."*

```xsd
<!-- Contact Type -->
<xsd:complexType name="ContactType" final="#all">
  <xsd:sequence>
    <xsd:element name="Phone" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="Name" />
</xsd:complexType>
```

Listing 11.54

You can achieve the same results by specifying *restriction* and *extension* together. The following schema is equivalent to the one given in the above example.

```
<xsd:complexType name="ContactType" final="extension restriction">
  <xsd:sequence>
    <xsd:element name="Phone" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="Name" />
</xsd:complexType>
```

Listing 11.55

Chapter Summary

Based on the content model, complex types are categorized into *simple content, element only content, mixed content* and *empty* content. You can derive new complex types from these types by extension or by restriction.

When deriving by extension, you add elements or attributes to the base type. When deriving by restriction you eliminate elements or attributes. You can also add additional validations to the elements or attributes declared in the base type.

Simple types can be extended to create complex types having simple content. By setting the *final* attribute of a simple type to *extension,* you can prevent the simple type from being extended.

When deriving by restriction, the derived type needs to specify the elements to be included in its content model. Elements declared in the base type are not passed down to the derived type by default. Attributes declared in the base type are passed down to the derived type by default. You don't need to redefine the attributes in the derived type. You should redefine attributes only if you want to eliminate the attribute or if you want to apply restrictions on one or more attributes.

To eliminate an element, simply don't include it while re-declaring elements in the derived type. To eliminate an attribute in a derived type, the attribute needs to be re-declared with *use* attribute set to *"prohibited."* You can eliminate an element or attribute from a derived type only if it is declared optional in the base type. You cannot eliminate a mandatory attribute or element in a base type.

When deriving by extension, all the elements and attributes in the base type are passed down to the derived type. You don't need to re-declare them.

You can prevent derivation of complex types by using the *final* attribute. The *final* attribute can take *extension, restriction* or *#all.* When it is set to *restriction*, the complex type is protected from derivation by restriction. When set to *extension*, you cannot derive a new type by extension. By setting the *final* attribute to *#all,* you can prevent derivation completely.

CHAPTER 12

XSD REGULAR EXPRESSION LANGUAGE

In the previous chapters we have covered several examples that use pattern restrictions to validate the format of values. The pattern restriction uses a Regular expression pattern to perform a format validation. XSD supports a Regular Expression language similar to the Regular expression languages supported by popular programming languages such as .NET, Perl, Python, etc.

This regular expression language helps to define a pattern that will be matched against the value stored in a simple type. In this chapter we will examine the Regular Expression language supported by XSD. We will discuss the following.

- What are Regular Expressions?
- Understanding Regular Expression Patterns
- Regular Expression Meta Characters
- Case Sensitivity
- Shorthand Character Classes
- Negative Expressions
- Character Class Subtraction

After examining the above, we will do a hands-on-lab which is a continuation of the labs that we did in the previous chapters. This will be the final lab and we will write the missing parts to complete the schema needed for the order processing application.

What are Regular Expressions?

I am sure that all of you are familiar with the find functionality most applications provide (MS Word, web browser, text editors, etc). This functionality helps us to quickly find a given character or group of characters within the body of a document.

Sometimes we might come across situations where a normal string find operation is not enough. Think of an application that parses web pages and

extracts all email addresses. You can easily find all occurrences of a static value using the normal find operation. But how do you find all occurrences of *"a word followed by an '@' sign and then followed by two words separated by a period (without any spaces or special characters)"*?

We need a special notation language to write this kind of search requirement. A notation language may translate this requirement to a certain format that a search component understands. We then need a search component that understands this notation format and is capable of performing a search operation based on the requirement specified by the notation.

A Regular Expression Language is such a notation language that allows defining patterns to perform complex string matching operations. A component that understands such a pattern and is capable of performing the pattern matching operation is called a Regular Expression Engine. A pattern that represents a specific string matching requirement is known as a *Regular Expression,* or *Regular Expression Pattern*.

Programming languages like Perl, Microsoft .NET, Python, etc., have their own regular expression engines. While the syntax and notations used for creating regular expression patterns in those languages are similar, each vendor may add their own extensions or enhancements.

XSD also supports a regular expression language very similar to the regular expressions languages supported by applications mentioned above. XSD uses regular expressions in pattern restrictions. By using pattern restrictions you can validate the format of values stored in simple types. We have seen several examples in the previous chapters.

Let us look closer into the regular expression language supported by XSD.

Understanding Regular Expression Patterns

Many of the examples we discussed in the previous chapters used pattern restrictions to validate the format of values. Most of the examples we saw used very simple regular expression patterns. Let us revisit one of the examples we saw earlier and try to understand the anatomy of a regular expression pattern.

Assume that we need to restrict the value of a simple type that stores a phone number. For the purpose of this example, let us assume that we need the phone number to be in the following format: *"(999) 123-5678."*

Let us create a simple type with a pattern restriction that performs this validation.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\([0-9]{3}\) [0-9]{3}-[0-9]{4}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<Phone>(999) 999-9999</Phone>'
```

Listing 12.1

Let us try to understand the regular expression pattern used in the above example. To understand the anatomy of this pattern, let us divide it into the following seven parts.

Part	1	2	3	4	5	6	7
Pattern	\(	[0-9]{3}	\)		[0-9]{3}	-	[0-9]{4}
Value	(	999	)		123	-	4567

Let us look at each part of the pattern in detail.

Part	1
Pattern	\(
Value	(

The first part of the pattern is a backslash followed by a parenthesis. To understand this we need to understand meta characters and regular characters.

Meta characters are characters having some special meaning when used within a regular expression pattern. For example, a question mark is a meta character that has a special meaning when used in a pattern. A question mark stands for 0 or 1 occurrence of the preceding character or group of characters. When a normal character is used inside a regular expression pattern, it represents the occurrence of that character in the value being validated using the pattern.

A parenthesis is a meta character. It is used to define a group of characters or expressions. A backslash is a meta character, also. It is used as an escape sequence character. When a meta character is preceded by a backslash, the meta character loses its "meta" status and will be considered a normal character.

In the above example, since the open parenthesis is preceded by a backslash, the parenthesis is not considered a meta character. The parenthesis will be treated as a normal character that stands for the occurrence of a parenthesis in the value.

Part	2
Pattern	[0-9]{3}
Value	999

This part of the pattern has a few meta characters, also Square brackets and Curly braces are meta characters. A pair of square brackets is used to define a range of values. The first part of the pattern defines a range that represents a digit between 0 and 9, inclusive.

Curly braces are used to control the occurrence of the preceding character or group of characters. The value inside the curly braces indicates the number of times the preceding character, or group of characters, should occur in the value. The above pattern stands for 3 occurrences of a digit between 0 and 9, inclusive.

Part	3
Pattern	\)
Value	)

This pattern includes a backslash and a closing parenthesis. We saw that a backslash is used to escape a meta character. So the above pattern stands for a single occurrence of a closing parenthesis in the value at character position 5.

Part	4
Pattern	
Value	

The next part of the pattern is a space. **That is why you do not see anything in the cell above.** A space is a normal character which stands for the occurrence of a space in the value.

Part	5
Pattern	[0-9]{3}
Value	123

We have seen a similar pattern a little earlier. It stands for 3 occurrences of a digit within the range of 0 to 9.

Part	6
Pattern	-
Value	-

The next part of the pattern has a hyphen. A hyphen is a normal character and stands for the occurrence of a hyphen in the value.

Part	7
Pattern	[0-9]{4}
Value	4567

This part of the expression stands for four occurrences of a digit between 0 and 9, inclusive.

We saw a basic regular expression pattern and learned how to read the pattern and understand the format of the value described by the pattern. Now let us have a closer look at the building blocks of regular expression patterns.

Meta Characters

We have seen a few meta characters in the example we discussed earlier. Meta characters are characters having some special meaning when placed within a regular expression pattern.

XSD Regular Expression language uses the following meta characters.

- Dot – "."
- Backslash – "\"
- Question Mark – "?"
- Asterisk – "*"
- Plus – "+"
- Curly Braces – "{}"
- Parenthesis - "()"
- Square Brackets – "[]"
- Pipe – "|"

Let us examine each of these meta characters in detail.

Dot

A *Dot* is a Meta Character. It stands for the single occurrence of any character.

The following pattern defines a *CustomerName* element, which is twenty characters long and accepts any character.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="CustomerName">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value=".{20}"/>
      </xsd:restriction>
```

```
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.2

Question Mark

A Question Mark (?) represents 0 or 1 occurrence of the character, or group of characters, which immediately precedes it.

Let us go back to the pattern we created for validating phone numbers. Assume that the requirement changes and you need to make the space at position 6 and the hyphen at position 10 optional. The new version of the pattern should accept all of the values given below.

- (999) 123-4567
- (999)123-4567
- (999) 1234567
- (999)1234567

Let us rewrite the schema with a modified version of the pattern. Let us use a question mark to make the space and hyphen optional. Here is the modified version of the schema.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\([0-9]{3}\) ?[0-9]{3}-?[0-9]{4}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<Phone>(999) 999-9999</Phone>'
```

```
SET @x = '<Phone>(999)999-9999</Phone>'
SET @x = '<Phone>(999) 9999999</Phone>'
SET @x = '<Phone>(999)9999999</Phone>'
```

Listing 12.3

Asterisk

We saw that a question mark can be used to represent 0 or 1 occurrence of a character or group of characters. An asterisk represents 0 or more occurrences of a character or group of characters.

The following pattern validates a string that accepts any lower case letter any number of times. Note that it allows 0 occurrences, also; hence, an empty string will also be accepted.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="MiddleName">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-z]*"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.4

Plus

A *Plus* sign represents one or more occurrences of a character or group of characters.

Question mark, asterisk and plus sign are used to control the occurrences of characters or group of characters. Here is a table that summarizes the usage of these three meta characters.

Meta Character	Meaning
?	0 or 1 occurrence
*	0 or More occurrences
+	1 or More occurrences

Backslash

Backslash is used as an escape sequence character in XSD regular expression patterns. This is used to indicate that we are not interested in the escaped character as a meta character, but are simply interested in the character itself. We have seen the usage of the escape sequence character in the example we discussed earlier in this chapter.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Amount">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[\+-][0-9]+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.5

Note that we have two plus signs in the pattern. The first sign is escaped and as a result will match for the occurrence of a plus sign in the value. The second plus sign is not escaped, and is treated as a Meta Character. The above pattern accepts both values given below.

```
DECLARE @x XML(Amount)
SET @x = '<Amount>+12</Amount>'
SET @x = '<Amount>-12</Amount>'
```

Listing 12.6

How do we match with a backslash itself? For example, how do we write a pattern that accepts a back slash in the value? Well, the back slash can be used to escape itself.

Here is an example that demonstrates this.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
```

```
   <xsd:element name="path">
     <xsd:simpleType>
       <xsd:restriction base="xsd:string">
         <xsd:pattern value="[c-e]:\\jacob"/>
       </xsd:restriction>
     </xsd:simpleType>
   </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<path>c:\jacob</path>'
SET @x = '<path>d:\jacob</path>'
SET @x = '<path>e:\jacob</path>'
```

Listing 12.7

Parentheses

Parentheses are used to group a set of characters. To understand this, let us look at a different version of the pattern we created to validate phone numbers.

Assume that you want to accept a phone number with or without the three-digit area code. We saw earlier that we can use a question mark to indicate that a character or group of characters is optional. So, we need to define the area code as a single group and then make it optional by using a question mark.

Here is a schema that does this.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="(\([0-9]{3}\))? ?[0-9]{3}-?[0-9]{4}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

```
-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<Phone>(999) 999-9999</Phone>'
SET @x = '<Phone>999-9999</Phone>'
```

Listing 12.8

The first part of the pattern used in the above example (shown in bold) might look a bit confusing because of the two parentheses used together. The first and last parentheses are used to create an expression group. But the other two parentheses are escaped; hence, they represent the occurrence of an *opening* and *closing* parentheses within the value.

Square Brackets

Square brackets are used to represent a character from a value range or a character from a list of given characters. To understand this, let us look at an example. The pattern in the following example represents one or more occurrences of a character in the range of "a–z" in upper or lower case.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z]+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.9

The pattern shown in the schema below represents a word that starts with a vowel in upper or lower case.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[aeiouAEIOU][a-zA-Z]+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.10

Curly Braces

Curly Braces are used to express the number of times a character or group of characters should occur. You can specify the minimum and maximum occurrences using these meat characters. The following table summarizes the usages of curly braces.

Pattern	Meaning
{3}	Exactly 3 occurrences
{2,4}	Minimum 2 occurrences. Maximum 4 occurrences.
{2,}	Minimum 2 occurrences. No maximum Limit

The following pattern restricts the value of a simple type to be exactly three characters long.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z]{3}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.11

The pattern shown in the schema below restricts the value of a simple type to two or three characters. It accepts values with two characters as well as three characters. The first value in the curly braces defines the minimum occurrence and the second value defines the maximum occurrence.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z]{2,3}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.12

The pattern in this schema restricts the value of a simple type to be, at minimum, two characters long. There is no maximum limit specified.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Name">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-zA-Z]{2,}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.13

Pipe Symbol

The *Pipe* symbol is used as an OR operator between a set of values. For example, the following pattern accepts a URL that starts with *http, ftp* or *https.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="path">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="(http|ftp|https)://sub.domain.com"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.14

Case Sensitivity

Unlike other regular expression languages, unfortunately XSD regular expression language does not provide a way for a case insensitive match. Case sensitivity needs to be handled within the pattern expression itself.

It will be tedious to write a pattern that performs a case insensitive validation. For example, the following is a case insensitive pattern that validates a string against value: *SQL Server 2005."*

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
```

```
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Database">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[Ss][Qq][Ll] [Ss][Ee][Rr][Vv][Ee][Rr]
        2005"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SET @x = '<Database>SQL Server 2005</Database>'
SET @x = '<Database>SQL SERVER 2005</Database>'
SET @x = '<Database>Sql Server 2005</Database>'
SET @x = '<Database>sql server 2005</Database>'
```

Listing 12.15

Shorthand Character Classes

Shorthand Character Classes are predefined expressions that represent frequently used character expressions. For example, the character class *"\d"* represents a digit. It is equivalent to *"[0-9]."* Shorthand character classes make pattern writing easier.

XSD Regular Expression Language supports the following *Shorthand Character Classes.*

Character Class	Meaning
\d	Any single Digit
\s	A single White Space (Space, Tab, CR or LF)
\i	An XML 1.0 initial name character: a letter of alphabet or an underscore.
\w	"word" character, usually [a-zA-Z0-9]
\c	XML 1.0 Name characters - XML 1.0 initial name character plus ".", ":", "-" and digits.

Note that short hand character classes are case sensitive. All the classes above should be used in lower case. You can create a negation of each of the above classes by changing the class to upper case.

Character Class	Meaning
\D	Any single Non Digit Character
\S	Any single Non Whitespace character
\I	A character that is not an XML initial name character.
\W	A character that is not a word character
\C	A character that is not an XML 1.0 name character.

Negative Expressions

All the patterns that we discussed so far were instructing the pattern processor to match with an expression. Negative expressions are the opposite of that. They instruct the pattern processor not to accept a value if it matches with the given pattern.

A negative expression is created by using a caret sign (^). The following pattern matches with any word that does not start with a vowel.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Word">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[^aeiou].*"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.16

Character Class Subtraction

So far, we have seen several examples that demonstrated almost all features of the Regular Expression Language of XSD. If you have followed all the examples we discussed so far, I am pretty sure that you will be capable of writing most patterns that your programming life would demand.

Now let us see a feature I would call quite advanced. That is character class subtraction. Assume that you need to write the pattern to validate a word that starts with a consonant. How do you do that? I could see a few hands going up in the air with something resembling the following pattern:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="word">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[b-df-hj-np-tv-z][a-z]+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.17

You are absolutely right. The pattern is correct. But there is an easier way to write this pattern. That is by using a feature called *Character Class Subtraction.*

Character Class Subtraction means subtracting a pattern from another. Look at the example below:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="word">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[a-z-[aeiuo]][a-z]+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.18

Instead of defining all the consonants (which is a tedious job), we defined all the alphabet (a–z) and then subtracted the vowels from it.

The following pattern matches with any number that does not start with zero.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="word">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="[\d-[0]]\d+"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>
```

Listing 12.19

In the above pattern we defined "\d," which represents all the digits, and subtracted 0 from it. Character Class subtraction makes writing complex patterns easier.

LAB5: Write schema for the Order Processing Application – Writing the final schema

We have done four labs so far and developed most parts of the schema needed for the order processing application. In this lab, we will write the schema for the *Contact*, *Item* and *Discount* elements. Once those elements are created, we will assemble the final schema.

Contact element

Each *Customer* element should contain a *Contact* element. This element contains information about the contact person at the customer's organization. In case of any query or communication related to the order, this person should be contacted.

Here is an example of a *Contact* element.

```
<Contact Name="Howard Snyder" Title="Purchase Manager">
  <Email>hsnyder@greatlakes.com</Email>
  <Phone>(503) 555-7555</Phone>
  <Fax>(503) 555-2376</Fax>
</Contact>
```

Listing 12.20: Example of a Contact element.

The contact element should follow these rules:

- Should have two mandatory attributes: *Name* and *Title.*
- Value of *Name* should not be empty and should not contain more than twenty characters.
- Value of *Title* should not be empty and should not contain more than twenty characters

.

- *Contact* element can have the following child elements:
  - o Email
  - o Phone
  - o Fax
- The child elements should appear exactly in the order given above.
- *Email* is mandatory and is expected in the format of string1@string2.string3. The schema should contain only the following simple validations. [Steve did not want to make it too complex.]
- Only alpha-numeric characters are allowed in string1, string2 and string3.
- There should be EXACTLY one "@" sign in the whole email address and it should appear between *string1* and *string2.*
- There should be at least one "." between string2 and string3.
- *Phone* is mandatory and should be in the format: **(503) 555-7555**.
- *Fax* is optional. If present, it should be in the same format as the phone number.

Let us start writing the schema. Here is the basic declaration of the *Contact* element.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Contact"/>
</xsd:schema>
```

Listing 12.21

Rule 1

Contact element should have two mandatory attributes: Name and Title

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Contact">
      <xsd:complexType>
         <xsd:attribute name="Name"/>
         <xsd:attribute name="Title"/>
      </xsd:complexType>
   </xsd:element>
</xsd:schema>
```

Listing 12.22

397

Rule 2

Value of Name should not be empty and should not contain more than twenty characters.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:attribute name="Name">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="Title"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 12.23

Rule 3

Value of Title should not be empty and should not contain more than twenty characters.

Note that the validation needed for the *Name* and *Title* elements are the same. So, instead of writing those restrictions twice, let us create a simple type describing the required length validation. Once that is created, both *Name* and *Title* elements can be bound to the new simple type.

In the example below a new simple type is created (*NameType*), and *Name* and *Title* are declared as simple types of *NameType.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:attribute name="Name" type="NameType"/>
      <xsd:attribute name="Title" type="NameType"/>
    </xsd:complexType>
  </xsd:element>
  <!-- Name Type -->
  <xsd:simpleType name="NameType">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
      <xsd:maxLength value="20"/>
```

```
    </xsd:restriction>
   </xsd:simpleType>
</xsd:schema>
```

Listing 12.24

Rule 4

**Contact element can have the following child elements: Email,
Phone and Fax. Email should be the first element, followed by
Phone and then Fax.**

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Contact">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Email"/>
         <xsd:element name="Phone"/>
         <xsd:element name="Fax"/>
       </xsd:sequence>
       <xsd:attribute name="Name" type="NameType"/>
        <xsd:attribute name="Title" type="NameType"/>
     </xsd:complexType>
   </xsd:element>
   <!-- Name Type -->
   <xsd:simpleType name="NameType">
     <!-- Removed for Brevity -->
   </xsd:simpleType>
</xsd:schema>
```

Listing 12.25

Rule 5

**Email is mandatory and is expected in the format of
string1@string2.string3. Only alpha-numeric characters are
allowed in string1, string2 and string3. There should be EXACTLY
one "@" sign in the whole email address and it should appear
between string1 and string2. There should be at least one "."
between string2 and string3.**

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Contact">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Email">
```

```xml
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:pattern value="\w+@\w+\.\w+"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="Phone"/>
          <xsd:element name="Fax"/>
        </xsd:sequence>
        <xsd:attribute name="Name" type="NameType"/>
         <xsd:attribute name="Title" type="NameType"/>
      </xsd:complexType>
    </xsd:element>
    <!-- Name Type -->
    <xsd:simpleType name="NameType">
        <!-- Removed for Brevity -->
    </xsd:simpleType>
  </xsd:schema>
```

Listing 12.26

Rule 6

Phone is mandatory and should be in the following format of "(503) 555-7555." Fax is optional. If present, it should be in the same format as the phone number.

Phone and fax have the same validation requirements and, therefore, let us create a simple type that validates a phone/fax number.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Email">
          <!-- Removed for Brevity -->
        </xsd:element>
        <xsd:element name="Phone" type="PhoneType"/>
        <xsd:element name="Fax" type="PhoneType"
                     minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="Name" type="NameType"/>
       <xsd:attribute name="Title" type="NameType"/>
    </xsd:complexType>
  </xsd:element>
  <!-- NameType -->
  <xsd:simpleType name="NameType">
    <!-- Removed for Brevity -->
  </xsd:simpleType>
  <!-- PhoneType -->
  <xsd:simpleType name="PhoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\(([0-9]{3}\) [0-9]{3}-[0-9]{4}"/>
```

```
      </xsd:restriction>
    </xsd:simpleType>
</xsd:schema>
```

Listing 12.27

Here is the complete schema that validates the contact element.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Contact">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Email">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="\w+@\w+\.\w+"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Phone" type="PhoneType"/>
        <xsd:element name="Fax" type="PhoneType"
                    minOccurs="0"/>
      </xsd:sequence>
      <xsd:attribute name="Name" type="NameType"/>
        <xsd:attribute name="Title" type="NameType"/>
    </xsd:complexType>
  </xsd:element>
  <!-- NameType -->
  <xsd:simpleType name="NameType">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
      <xsd:maxLength value="20"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- PhoneType -->
  <xsd:simpleType name="PhoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\([0-9]{3}\) [0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Contact Name="Howard Snyder" Title="Purchase Manager">
  <Email>hsnyder@greatlakes.com</Email>
```

```
   <Phone>(503) 555-7555</Phone>
   <Fax>(503) 555-2376</Fax>
</Contact>'
```

Listing 12.28

Let us move to the next element. Let us create the declaration for the *Items* element.

Items element

Each order should have an *items* element which contains the details of items ordered. The following example shows the structure of the *Items* element.

```
<Items>
  <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
  <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
  <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
</Items>
```

Listing 12.29: An example of Items element.

The *Items* element can contain one or more *Item* elements. There should be one *Item* element for each item that the customer has ordered.

- There should be at least one *Item* element. There is no maximum limit.
- Each *Item* element should have three mandatory attributes:
    - ItemNumber
    - Quantity
    - Price
- *ItemNumber* should be exactly eight characters long. The first two characters of the *ItemNumber* should be upper case alphabets [A to Z], and the next six characters should be digits.
- *Quantity* should be a number and should be between 1 and 9999. Decimals are not allowed.
- *Price* should be a number between 0.01 and 999999.99. The value should have two decimal places.

Let us start writing the schema.

Rule 1

The items element should contain at least one Item element. There is no maximum limit.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Item"
                     type="ItemType"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 12.30

To simplify the schema, let us create a complex type that validates each item element. Let us create a type named *ItemType* that will describe an *Item* element.

Rule 2

Each Item element should have three mandatory attributes: ItemNumber, Quantity and Price.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="ItemType">
    <xsd:attribute name="ItemNumber"/>
    <xsd:attribute name="Quantity"/>
    <xsd:attribute name="Price"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 12.31

Rule 3

ItemNumber should be exactly eight characters long. The first two characters of the ItemNumber should be upper case alphabets [A to Z], and the next six characters should be digits.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="ItemType">
    <xsd:attribute name="ItemNumber">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}[0-9]{6}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Quantity"/>
    <xsd:attribute name="Price"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 12.32

Rule 4

Quantity should be a number and should be between 1 and 9999. Decimals are not allowed.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="ItemType">
    <!-- Other attributes here -->
    <xsd:attribute name="Quantity">
      <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
          <xsd:minInclusive value="1"/>
          <xsd:maxInclusive value="9999"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Price"/>
  </xsd:complexType>
</xsd:schema>
```

Listing 12.33

Rule 5

Price should be a number between 0.01 and 999999.99. The value should have two decimal places.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:complexType name="ItemType">
    <!-- Other attributes here -->
    <xsd:attribute name="Price">
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
          <xsd:fractionDigits value="2"/>
          <xsd:minInclusive value="0.01"/>
```

```
                      <xsd:maxInclusive value="999999.99"/>
                  </xsd:restriction>
              </xsd:simpleType>
          </xsd:attribute>
      </xsd:complexType>
</xsd:schema>
```

Listing 12.34

Here is the complete schema that validates the *Items* element.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Items">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Item"
                     type="ItemType"
                     maxOccurs="unbounded"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:complexType name="ItemType">
    <xsd:attribute name="ItemNumber">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}[0-9]{6}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Quantity">
      <xsd:simpleType>
        <xsd:restriction base="xsd:integer">
          <xsd:minInclusive value="1"/>
          <xsd:maxInclusive value="9999"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Price">
      <xsd:simpleType>
        <xsd:restriction base="xsd:decimal">
          <xsd:fractionDigits value="2"/>
          <xsd:minInclusive value="0.01"/>
          <xsd:maxInclusive value="999999.99"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
  </xsd:complexType>
</xsd:schema>'
```

```
GO

-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Items>
  <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
  <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
  <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
</Items>'
```

Listing 12.35

Discount element

Each *Order* element may contain an optional *Discount* element. Discount may be given in terms of a fixed amount or as a certain percentage of the total invoice amount. Here are some examples of *Discount* element.

```
<Discount>
   <Amount>300</Amount>
</Discount>
```

Listing 12.36: Discount as Amount.

```
<Discount>
   <Percent>7.5</Percent>
</Discount>
```

Listing 12.37: Discount as Percentage.

The *Discount* element should follow the rules given below:

- It should contain either an *Amount* element or a *Percent* element. These two elements are mutually exclusive. Either one of them can be present in a *Discount* element.
- If *Amount* is present, the minimum value should be 0.01. There is no maximum limit. The value should always have two decimals.
- If *Percent* is present, the value should be between 0.01 and 100.00. The value should always have two decimals.

Let us start writing the *Discount* element.

Rule 1

It should contain either an Amount element or a Percent element. These two elements are mutually exclusive. Either one of them can be present in a Discount element.

To represent a set of mutually exclusive elements we need to use a *choice* group. Let us add a *choice* group to the discount element.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Discount">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Amount"/>
        <xsd:element name="Percent"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 12.38

Rule 2

If Amount is present, the minimum value should be 0.01. There is no maximum limit. The value can have, at maximum, two decimals.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Discount">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Amount">
          <xsd:simpleType>
            <xsd:restriction base="xsd:decimal">
              <xsd:minInclusive value="0.01"/>
              <xsd:fractionDigits value="2"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Percent"/>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 12.39

Rule 3

If Percent is present, the value should be between 0.01 and 100.00. The value can have, at maximum, two decimals.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Discount">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Amount">
          <!-- Removed for Brevity -->
        </xsd:element>
        <xsd:element name="Percent">
          <xsd:simpleType>
            <xsd:restriction base="xsd:decimal">
              <xsd:fractionDigits value="2"/>
              <xsd:minInclusive value="0.01"/>
              <xsd:maxInclusive value="100"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
      </xsd:choice>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 12.40

Here is the final schema that validates a discount element.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Discount">
    <xsd:complexType>
      <xsd:choice>
        <xsd:element name="Amount">
          <xsd:simpleType>
            <xsd:restriction base="xsd:decimal">
              <xsd:minInclusive value="0.01"/>
              <xsd:fractionDigits value="2"/>
            </xsd:restriction>
          </xsd:simpleType>
        </xsd:element>
        <xsd:element name="Percent">
```

```
            <xsd:simpleType>
              <xsd:restriction base="xsd:decimal">
                <xsd:fractionDigits value="2"/>
                <xsd:minInclusive value="0.01"/>
                <xsd:maxInclusive value="100"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
        </xsd:choice>
      </xsd:complexType>
    </xsd:element>
</xsd:schema>'
GO
```

Listing 12.41

Let us use this schema to validate two variations of the *Discount* elements that we saw earlier in this lab.

```
-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Discount>
  <Amount>300</Amount>
</Discount>'
```

Listing 12.42

```
-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Discount>
  <Percent>7.5</Percent>
</Discount>'
```

Listing 12.43

Assemble the final schema

Let us merge the schema we created in this version with the previous version of the schema. Before we create the new version of the schema collection, let us drop the previous version.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
```

```
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO
```

Listing 12.44

Here is the TSQL code that creates the final version of the schema collection.

```
-- Create Schema Collection
CREATE XML SCHEMA COLLECTION ExampleSchema
AS
'<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Order">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="OrderDate" type="xsd:date"/>
        <xsd:element name="DeliveryDate" type="xsd:dateTime"/>
        <xsd:element name="Customer">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="CustomerName"
                        minOccurs="0" maxOccurs="1">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:maxLength value="50"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Billing" type="AddressType"/>
              <xsd:element name="Shipping" type="AddressType"
                        minOccurs="0"/>
              <xsd:element name ="Terms">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:enumeration value="30 Days Credit"/>
                    <xsd:enumeration value="60 Days Credit"/>
                    <xsd:enumeration value="90 Days Credit"/>
                    <xsd:enumeration value="Against Delivery"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:element>
              <xsd:element name="Contact">
                <xsd:complexType>
                  <xsd:sequence>
                    <xsd:element name="Email">
                      <xsd:simpleType>
                        <xsd:restriction base="xsd:string">
                          <xsd:pattern value="\w+@\w+\.\w+"/>
                        </xsd:restriction>
                      </xsd:simpleType>
                    </xsd:element>
                    <xsd:element name="Phone" type="PhoneType"/>
                    <xsd:element name="Fax" type="PhoneType"
                        minOccurs="0"/>
                  </xsd:sequence>
                  <xsd:attribute name="Name" type="NameType"/>
                  <xsd:attribute name="Title" type="NameType"/>
                </xsd:complexType>
```

```
                </xsd:element>
              </xsd:sequence>
              <xsd:attribute name="CustomerNumber" use="required">
                <xsd:simpleType>
                  <xsd:restriction base="xsd:string">
                    <xsd:pattern value="[a-zA-Z]{5}"/>
                  </xsd:restriction>
                </xsd:simpleType>
              </xsd:attribute>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="Items">
            <xsd:complexType>
              <xsd:sequence>
                <xsd:element name="Item"
                             type="ItemType"
                             maxOccurs="unbounded"/>
              </xsd:sequence>
            </xsd:complexType>
          </xsd:element>
          <xsd:element name="OrderNote" minOccurs="0">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="500"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="InvoiceNote" minOccurs="0">
            <xsd:simpleType>
              <xsd:restriction base="xsd:string">
                <xsd:maxLength value="500"/>
              </xsd:restriction>
            </xsd:simpleType>
          </xsd:element>
          <xsd:element name="Discount" minOccurs="0">
            <xsd:complexType>
              <xsd:choice>
                <xsd:element name="Amount">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:decimal">
                      <xsd:minInclusive value="0.01"/>
                      <xsd:fractionDigits value="2"/>
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:element>
                <xsd:element name="Percent">
                  <xsd:simpleType>
                    <xsd:restriction base="xsd:decimal">
                      <xsd:fractionDigits value="2"/>
                      <xsd:minInclusive value="0.01"/>
                      <xsd:maxInclusive value="100"/>
                    </xsd:restriction>
                  </xsd:simpleType>
                </xsd:element>
              </xsd:choice>
            </xsd:complexType>
          </xsd:element>           </xsd:sequence>
        <xsd:attribute name="OrderNumber" use="required">
          <xsd:simpleType>
            <xsd:restriction base="xsd:string">
              <xsd:pattern value="[a-zA-Z0-9]{1,20}"/>
            </xsd:restriction>
          </xsd:simpleType>
```

```xml
        </xsd:attribute>
      </xsd:complexType>
  </xsd:element>
  <!-- Address Type -->
  <xsd:complexType name="AddressType">
    <xsd:sequence>
      <xsd:element name="Address">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:minLength value="1"/>
            <xsd:maxLength value="50"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
      <xsd:element name="Street" minOccurs="0">
        <xsd:simpleType>
          <xsd:restriction base="xsd:string">
            <xsd:maxLength value="20"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:element>
    </xsd:sequence>
    <xsd:attribute name="City" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:minLength value="1"/>
          <xsd:maxLength value="30"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="State" use="required">
      <xsd:simpleType>
        <xsd:restriction base="xsd:string">
          <xsd:pattern value="[A-Z]{2}"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:attribute>
    <xsd:attribute name="Zip" use="required" type="zipType"/>
  </xsd:complexType>
  <!-- Zip Type -->
  <xsd:simpleType name="zipType">
    <xsd:restriction base="xsd:integer">
      <xsd:maxInclusive value="99999"/>
      <xsd:minInclusive value="10000"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- NameType -->
  <xsd:simpleType name="NameType">
    <xsd:restriction base="xsd:string">
      <xsd:minLength value="1"/>
      <xsd:maxLength value="20"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- PhoneType -->
  <xsd:simpleType name="PhoneType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="\([0-9]{3}\) [0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
  <!-- ItemType -->
  <xsd:complexType name="ItemType">
    <xsd:attribute name="ItemNumber">
      <xsd:simpleType>
```

```
          <xsd:restriction base="xsd:string">
            <xsd:pattern value="[A-Z]{2}[0-9]{6}"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="Quantity">
        <xsd:simpleType>
          <xsd:restriction base="xsd:integer">
            <xsd:minInclusive value="1"/>
            <xsd:maxInclusive value="9999"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
      <xsd:attribute name="Price">
        <xsd:simpleType>
          <xsd:restriction base="xsd:decimal">
            <xsd:fractionDigits value="2"/>
            <xsd:minInclusive value="0.01"/>
            <xsd:maxInclusive value="999999.99"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
</xsd:schema>'
GO
```

Listing 12.45

Let us validate an XML instance containing order information with the final version of the schema collection.

```
-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Order OrderNumber="20002">
  <OrderDate>2008-01-01Z</OrderDate>
  <DeliveryDate>2008-01-10T09:00:00-08:00</DeliveryDate>
  <Customer CustomerNumber="LAZYK">
    <CustomerName>Lazy K Kountry Store</CustomerName>
    <Billing City="Eugene" State="OR" Zip="97403">
      <Address>2732 Baker Blvd.</Address>
      <Street>Main st.</Street>
    </Billing>
    <Shipping City="Eugene" State="OR" Zip="97403">
      <Address>2732 Baker Blvd.</Address>
      <Street>Main st.</Street>
    </Shipping>
    <Terms>30 Days Credit</Terms>
    <Contact Name="Howard Snyder" Title="Purchase Manager">
      <Email>hsnyder@greatlakes.com</Email>
      <Phone>(503) 555-7555</Phone>
      <Fax>(503) 555-2376</Fax>
    </Contact>
  </Customer>
  <Items>
    <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
    <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
    <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
  </Items>
```

```
   <OrderNote>Delivery needed before 8 AM</OrderNote>
   <InvoiceNote>
     Adjust the previous credit note with this invoice
   </InvoiceNote>
   <Discount>
     <Amount>300</Amount>
   </Discount>
</Order>'
```

Listing 12.46

Here is another version of the XML instance that uses Discount percentage instead of discount amount.

```
-- Validate
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Order OrderNumber="20002">
   <OrderDate>2008-01-01Z</OrderDate>
   <DeliveryDate>2008-01-10T09:00:00-08:00</DeliveryDate>
   <Customer CustomerNumber="LAZYK">
     <CustomerName>Lazy K Kountry Store</CustomerName>
     <Billing City="Eugene" State="OR" Zip="97403">
       <Address>2732 Baker Blvd.</Address>
       <Street>Main st.</Street>
     </Billing>
     <Shipping City="Eugene" State="OR" Zip="97403">
       <Address>2732 Baker Blvd.</Address>
       <Street>Main st.</Street>
     </Shipping>
     <Terms>30 Days Credit</Terms>
     <Contact Name="Howard Snyder" Title="Purchase Manager">
       <Email>hsnyder@greatlakes.com</Email>
       <Phone>(503) 555-7555</Phone>
       <Fax>(503) 555-2376</Fax>
     </Contact>
   </Customer>
   <Items>
     <Item ItemNumber="FB001923" Quantity="12" Price="18.25" />
     <Item ItemNumber="SG060020" Quantity="80" Price="12.75" />
     <Item ItemNumber="FB019090" Quantity="24" Price="6.00" />
   </Items>
   <OrderNote>Delivery needed before 8 AM</OrderNote>
   <InvoiceNote>
     Adjust the previous credit note with this invoice
   </InvoiceNote>
   <Discount>
     <Percent>7.5</Percent>
   </Discount>
</Order>'
```

Listing 12.47

We have successfully created the schema collection needed for the order processing application. This is the last lab we will do in this book. I hope the labs we attended helped you understand the building blocks of XSD

and made you capable enough to undertake any schema development assignments that you might come across in your SQL Server programming life.

Chapter Summary

XSD supports Regular Expressions to validate the format of a value with a pattern. This Regular Expression language is close to the Regular Expression languages supported by programming languages like Perl, Python or Microsoft Dot Net class libraries.

A Regular Expression may contain Normal Characters as well as Meta Characters. Meta characters have some special meaning when they appear inside a Regular Expression. A backslash is used to escape a Meta character so that it will lose its special meaning and will become a normal character.

Meta characters Question mark (?), Asterisk (*) and Plus (+) are used to define repetition of a character or group of characters. Curly Braces are used to define the minimum and maximum number of occurrences of a character or group. Square brackets are used to define a range of characters, whereas Parenthesis is used to define groups of characters. A pipe sign is used to "OR" groups of characters, from which only one group or character is to be matched. A caret (^) sign is used to create a negative expression.

Character classes are those expressions which represent a group of characters. The most commonly used character classes are "\d," which represents any digit. Other character classes are "\s," "\w," "\i" and "\c." The negative version of the above character classes can be created either with a caret (^) or by making the character upper case. For example, "\d" represents a digit where as "\D" represents a non-digit.

CHAPTER 13

ADVANCED SCHEMA CONCEPTS

In the last few chapters we have discussed various schema components in detail. All the previous chapters were more focused on the most commonly used schema components. There are certain parts of schema development that we have not yet touched in detail. We have discussed them briefly in the previous chapters and it is time to have a closer look at those topics.

In this chapter, we will discuss:

- Attributes of a schema declaration
- Wildcard components and content validation
- Element and attribute wildcards
- Annotations

Attributes of a schema declaration

We have seen several dozen schema declarations in the previous chapters. The only mandatory attribute of a schema declaration is the namespace declaration. Here is the basic declaration of a schema.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Customers" />
</xsd:schema>
```

Listing 13.1

This schema declares an element named *Customers*. A schema is declared with *"<xsd:schema>"* element and it should have a namespace declaration pointing to *"http://www.w3.org/2001/XMLSchema."* Though you can specify a namespace prefix of your choice, you cannot alter the namespace URI. Also, note that the namespace URI is case sensitive. If the schema processor does not find the correct namespace declaration, it will throw an error.

Though the only attribute a schema declaration must have is the schema namespace declaration, it can take a number of other optional attributes. These attributes control the behavior of the schema processor while validating XML instances against the schema collection.

A schema declaration can take the following optional attributes:

- **id**
- **targetNamespace**
- **attributeFormDefault**
- **elementFormDefault**
- **blockDefault**
- **finalDefault**
- **version**
- **xml:lang**

Let us examine each of these attributes in detail.

Attribute: id

The *id* attribute of a schema declaration shares the same characteristics as the *id* attributes of *attribute declarations* and *element declarations*. It is used only to uniquely identify each schema component. It does not add anything to the meaning of the schema or its validation rules.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            id="mySchema">
  <xsd:element name="Customers" type="xsd:string"/>
</xsd:schema>
```

Listing 13.2

We have discussed the *"id"* attribute of element declarations in Chapter 5 and attribute declarations in Chapter 6. Refer to those chapters for a detailed discussion on this attribute.

Attribute: targetNamespace

None of the XML instances which we saw previously had a namespace declaration. In Chapter 2 we saw how namespaces help to avoid ambiguity. The examples we saw in the previous chapters were so simple that they did not need a namespace declaration. There are times when you need to make sure that the XML instance should take a specific namespace to be qualified for consumption by a specific application. You can achieve this by using the *targetNamespace* of schema declaration.

Assume that you are writing a schema that describes an ATOM feed. An ATOM feed is an XML document having a specific structure and usually

contains information about new or modified resources available on web sites.

ATOM is a popular web feed format. For further information about ATOM feeds, see:

http://en.wikipedia.org/wiki/Atom_(standard)

The ATOM specification says that a valid ATOM document should contain the following root element. Let us try to write a schema that performs this validation.

```
<feed xmlns="http://www.w3.org/2005/Atom">
  <!-- Other declarations here -->
</feed>
```

Listing 13.3

We know how to create a schema that describes the root element of the XML instance. But we have not seen schema declarations that make sure that the XML instance contains a specific namespace declaration.

This is achieved by adding *targetNamespace* attribute to the schema declaration. The following code creates a schema collection that validates the xml instance given in *Listing 13.3.*

```
-- Create the schema
CREATE XML SCHEMA COLLECTION AtomSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      targetNamespace="http://www.w3.org/2005/Atom">
  <xsd:element name="feed"/>
</xsd:schema>'
GO

-- Validate
DECLARE @x XML(AtomSchema)
SELECT @x = '
<feed xmlns="http://www.w3.org/2005/Atom">
  <!-- Other declarations here -->
</feed>'
```

Listing 13.4: An example showing the usage of targetNamespace.

When a schema declaration contains the *targetNamespace* attribute, the XML instance should contain the namespace declaration specified in this attribute.

Attribute: attributeFormDefault

This attribute specifies the default value of the *form* attribute of attribute declarations in the given schema. We have examined the *form* attribute when we discussed attribute declarations in Chapter 6.

If an attribute is declared without the *form* attribute, the schema processor will look for the value of the *attributeFormDefault* attribute while validating it. If the schema declaration does not contain the *attributeFormDefault* attribute, the schema processor will assume *unqualified* as the default form for all the attributes. The default value is used only if the attribute declaration does not contain the *form* attribute.

The schema definition in the following example sets *qualified* as the default form of all attributes. The *Number* attribute overrides the default value by using the *form* attribute.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.employee.com"
            attributeFormDefault="qualified">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Number" form="unqualified"/>
      <xsd:attribute name="Name"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '
<emp:Employee xmlns:emp="www.employee.com"
      Number="123" emp:Name="Jacob" />'
```

Listing 13.5

The *"Number"* attribute is in *unqualified* form because it is declared with the *form* attribute having value: *"unqualified."* The *"Name"* attribute is in *qualified* form because it does not have the *form* attribute; hence, the schema processor uses the value of *attributeFormDefault,* which is *"qualified."*

Attribute: elementFormDefault

This attribute specifies the default form of elements. As with attributes, elements can have *qualified* or *unqualified* forms. We discussed this in Chapter 5. If an element is declared without the *form* attribute, the schema processor will use the value of *elementFormDefault* while validating the element. If the *"elementFormDefault"* attribute is not present in the schema declaration, SQL Server will assume *"unqualified"* as the default form of all elements.

As mentioned earlier, if an element declaration contains the *form* attribute it will override the default value specified by the *elementFormDefault* attribute. The following example demonstrates that.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.employee.com"
            elementFormDefault="qualified">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Number" form="unqualified"/>
        <xsd:element name="Name"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '
<emp:Employee xmlns:emp="www.employee.com">
      <Number>1001</Number>
      <emp:Name>Jacob</emp:Name>
</emp:Employee>'
```

Listing 13.6

The schema declaration sets the default value of the *form* attribute of all elements to *"qualified."* The *Name* element is declared without the *form* attribute, and as a result the schema processor assumes its form as *"qualified."*

The declaration of the *Number* element sets its form as *"unqualified"* by using the *form* attribute.

Attribute: blockDefault

This attribute defines the default value of the *block* attribute. You can use the *block* attribute to prevent *substitution*, *extension* or *restriction* of an element. Refer to Chapter 5 for a detailed explanation of the *block* attribute.

If an element is declared without the *block* attribute SQL Server will use the value of *blockDefault,* if present. If *blockDefault* is not present in the schema declaration and the element is declared without the *block* attribute, the element can be substituted, extended or restricted.

Attribute: finalDefault

This attribute defines the default value of the *final* attribute. You can use this attribute to prevent the restriction or extension of a given element or type. We have discussed the *final* attribute in Chapter 5.

If an element is declared without the *final* attribute, SQL Server will use the value of *finalDefault* if it is present in the schema declaration. If the schema declaration does not have the *finalDefault* attribute and the element is declared without the *final* attribute, the element can be extended as well as restricted.

Attribute: version

The attribute *version* is used only for the documentation purpose. The schema processor does not use it while validating the schema components.

What can occur is that after you created a schema the requirement changes and you might need to write a new schema to support the additional requirements. You may handle those additional requirements by extending the existing schema components or by creating a completely new version of the schema. The *version* attribute of the schema declaration can be used to label the new schema with a different version number.

The following example shows a schema declaration that contains a version number.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
version="1.0">
  <xsd:element name="Employee"/>
</xsd:schema>
```

Listing 13.7

Attribute: xml:lang

This is another attribute that is used only for the documentation purpose. This attribute is used in XML documents to specify the formal language in which the content is written.

Just as the *version* attribute, *xml:lang* is used only for the documentation and the schema processor does not use it for validating the schema components. This attribute can take a valid language identifier specified in RFC 3066 (http://www.ietf.org/rfc/rfc3066.txt).

Wildcard components and content validation

All the examples we have seen in the previous chapters demonstrated how strictly SQL Server validates XML instances against a schema collection. The XML instance should strictly follow the structure and restrictions defined in the schema collection. This behavior makes sure that the XML instance follows the exact structure specified in the schema collection and that the elements and attributes contain acceptable values.

When you define a schema, you predefine the elements and attributes the XML instance should have. There are times when you need to give a little more flexibility so that the XML instance can hold elements or attributes not known at the time the schema was written. This can be achieved by using wildcard declarations in the schema definition.

Let us go back to the example of the order processing application we examined in Chapter 3. Partner agencies of North Pole Corporation will send order information to the web service exposed by the order processing. This application needs to send a response back to the client applications indicating the success or failure of the request.

This response message may contain a response code, description and some additional information that may vary from case to case and time to

time. We need to keep a provision that if, in the future, we need to send some additional information regarding the status of the operation, we should be able to do it without modifying the schema and the processing logic. Let us see how wildcard components can help in this case.

XSD supports two wildcard components:

1. Element wildcards
2. Attribute wildcards

Let us examine each of these in detail.

Element Wildcards

An element wildcard is used to represent an element that is not known at the time of writing the schema. Element wildcards are declared using *"<xsd:any>"* element. An element wildcard can only appear within *sequence* or *choice* model groups.

Before we examine element wildcards in detail, let us see an example that uses an element wildcard declaration.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderRegistrationResult">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="skip"/>
      </xsd:sequence>
      <xsd:attribute name="Code"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.8

Note wildcard declaration inside the complex type. The two XML instances given below will validate successfully against the above schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<OrderRegistrationResult Code="101">
      <Status>Order registered successfully!</Status>
</OrderRegistrationResult>'
```

Listing 13.9

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<OrderRegistrationResult Code="101">
      <Status>
              <ID>100</ID>
              <Text>Order registered successfully!</Text>
      </Status>
</OrderRegistrationResult>'
```

Listing 13.10

The XML instance given in the first example has an element named *Status* which is not declared in the schema definition. Similarly, the second XML instance has an element named *Message* which is not declared in the schema definition as well. The element wildcard that we added to the schema definition represents the occurrence of an unknown element in the XML instance. Hence, both the XML instances given above will validate successfully with the schema collection.

An element wildcard declaration has the following attributes:

- processContents
- id
- maxOccurs and minOccurs
- namespace

Let us examine each of these attributes in detail.

Attribute: processContents

This attribute instructs the schema processor how to validate the wildcard element. The value of this attribute can be *"skip"*, *"strict"* or *"lax."*

Processing wildcard elements with "skip"

When the value of *processContents* is set to *"skip"* the schema processor will not attempt to validate the wildcard element. We have seen this behavior in the previous example.

```
-- Create a schema collection
CREATE XML SCHEMA COLLECTION SkipDemo AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="skip"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.11

This schema declares a complex type element named *ContactInfo*. It has
an element wildcard declaration. Since the *processContents* attribute is set
to *skip*, SQL Server will not perform any validation against this element.
Hence, all the following XML instances will be accepted.

```
DECLARE @x XML(SkipDemo)
SELECT @x = '
<ContactInfo>
      <Phone>999-999-9999</Phone>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Fax>999-999-9999</Fax>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Email>a@b.com</Email>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Phone>
            <Home>999-999-9999</Home>
            <Work>888-888-8988</Work>
      </Phone>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Phone Work="888-888-8888" Home="999-999-9999" />
</ContactInfo>'
```

Listing 13.12

Note that a wildcard element declaration can represent a simple type or
complex type. The first three examples show simple types and the last two
examples show complex types.

Processing wildcard elements with "strict"

When *processContents* is set to *"strict,"* SQL Server will validate the wildcard element against schema definition from a given namespace. The following example sets the *namespace* to *"##any,"* which indicates the XML instance can have a replacement element from any namespace.

```
CREATE XML SCHEMA COLLECTION StrictDemo AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict" namespace="##any"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.13

None of the XML instances we tried in the previous example will work for this version of the schema. Try to run the following code.

```
DECLARE @x XML(StrictDemo)
SELECT @x = '
<ContactInfo>
      <Phone>999-999-9999</Phone>
</ContactInfo>'
```

Listing 13.14

If you run the above code, SQL Server will raise the following error.

```
XML Validation: Declaration not found for element 'Phone'.
Location: /*:ContactInfo[1]/*:Phone[1]
```

To demonstrate a working example that uses *processContents* with *strict*, let me present a little more complex schema definition having multiple target namespaces. Run the following code to add a few more element declarations having different target namespaces to the above schema.

```
ALTER XML SCHEMA COLLECTION StrictDemo ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="www.fax.com">
  <xsd:element name="Fax"/>
</xsd:schema>'

ALTER XML SCHEMA COLLECTION StrictDemo ADD '
```

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="www.phone.com">
   <xsd:element name="Phone"/>
</xsd:schema>'

ALTER XML SCHEMA COLLECTION StrictDemo ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
targetNamespace="www.email.com">
   <xsd:element name="Email"/>
</xsd:schema>'
```

Listing 13.15

All the following XML instances will validate successfully. The wildcard declaration accepts elements from *any* namespace; therefore, all three XML instances given below are valid.

```
DECLARE @x XML(StrictDemo)
SELECT @x = '
<ContactInfo>
      <Fax xmlns="www.fax.com">999-999-9999</Fax>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Phone xmlns="www.phone.com">999-999-9999</Phone>
</ContactInfo>'

SELECT @x = '
<ContactInfo>
      <Email xmlns="www.email.com">a@b.com</Email>
</ContactInfo>'
```

Listing 13.16

Note that each element will be validated against the schema definition found in the specified namespace. For example, the *Fax* element in the first example will be validated against the definition of the schema having target namespace *"www.fax.com."*

Processing wildcard elements with "lax"

We saw two validation modes in the previous examples. When *processContents* is set to *skip*, SQL Server does not validate the element. When it is set to *strict*, SQL Server will validate the element against the definition given in the specified namespace. If the namespace is not found in the schema declaration, SQL Server will generate an error.

There is a third validation mode, *lax,* which will perform a little more liberal validation. Under *lax* validation mode, SQL Server will check if the

namespace specified by the replacement element exists in the schema collection or not. If the namespace is not found, SQL Server will skip the validation of the element. If the namespace is found, SQL Server will check if it contains a declaration for the given element. If it is not found, SQL Server will skip the validation of the element. The replacement element of a wildcard declaration will be validated only if the declaration of the element is found in the specified namespace within the schema collection.

SQL Server 2005 supports *strict* and *skip* methods, but not *lax*. Support for *lax* validation is added in SQL Server 2008. The following example will run only in SQL Server 2008.

Let us create a schema collection with an element wildcard declaration that requests *lax* validation.

```
-- Create a schema collection
CREATE XML SCHEMA COLLECTION LaxDemo AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="ContactInfo">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:any processContents="lax" namespace="##any"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 13.17

Let us add another schema definition to the above schema collection. Note that the new schema definition has a different target namespace.

```
ALTER XML SCHEMA COLLECTION LaxDemo ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="www.phone.com">
   <xsd:element name="Phone">
     <xsd:simpleType>
           <xsd:restriction base="xsd:string">
                 <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
           </xsd:restriction>
     </xsd:simpleType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 13.18

The above schema definition declares an element named Phone and performs a format validation. Now let us try to validate a few different XML instances against this schema.

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <Fax>12345</Fax>
</ContactInfo>'
```

Listing 13.19

This will validate successfully because the element Fax is not associated with any namespace and SQL Server will just skip it because the processing mode is set to lax.

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <HomePhone xmlns="www.HomePhone.com">12345</HomePhone>
</ContactInfo>'
```

Listing 13.20

This will validate as well. The namespace "www.homephone.com" does not exist in the current schema collection; hence, SQL Server will exclude this element from validation.

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <HomePhone xmlns="www.Phone.com">12345</HomePhone>
</ContactInfo>'
```

Listing 13.21

This will also succeed. In this case, SQL Server will be able to find the namespace in the schema collection. However, the namespace*"www.phone.com"* does not contain a declaration for *HomePhone* element and as a result SQL Server will skip the validation of this element.

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <Phone xmlns="www.phone.com">12345</Phone>
</ContactInfo>'
```

Listing 13.22

The above code will not validate. SQL Server will validate the *Phone* element against the declaration specified in the schema collection. In the

case of this example, SQL Server will find the namespace declaration and will find a matching declaration for the *Phone* element. The validation will fail because the format of the value does not match with the pattern specified in the pattern restriction.

Here is the correct XML instance.

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <Phone xmlns="www.phone.com">123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.23

Note that namespace declarations are case sensitive. The following will succeed even though the value is not in the correct format, because SQL Server will not be able to find this namespace declaration in the schema collection and will skip the validation. (Note the *"p"* in *"www.Phone.com"*)

```
DECLARE @x XML(LaxDemo)
SELECT @x = '
<ContactInfo>
      <Phone xmlns="www.Phone.com">1234</Phone>
</ContactInfo>'
```

Listing 13.24

As mentioned earlier, SQL Server 2005 does not support *lax* validation. If you attempt to create a schema collection having a wildcard declaration with *lax* validation, you will receive the following error.

```
The XML Schema syntax 'processContents="lax"' is not supported.
```

Attribute: Id

We saw the *"id"* attribute when we discussed element declarations and attribute declarations. The attribute *"id"* is used only to uniquely identify each schema component. It does not add anything to the validation rules or meaning of the schema. Here is an example that declares a wildcard element having an *"id"* attribute.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="skip" id="SkipDemo"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
    <Phone>999-999-9999</Phone>
</ContactInfo>'
```

Listing 13.25

Attributes: maxOccurs and minOccurs

These two attributes are not new to us. We previously discussed them when we discussed element declarations. They control the number of times an element can appear within its parent element.

To make an element optional, set *minOccurs* to 0. To make an element mandatory, set *minOccurs* to a non-zero value. To restrict the occurrence of an element to exactly once, set *minOccurs* and *maxOccurs* to 1. You cannot set *minOccurs* to a value that is higher than *maxOccurs*.

Here is an example that declares a wildcard element and can appear a minimum of two times and a maximum of five times.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
```

```
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="skip"
                 minOccurs="2" maxOccurs="5"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
     <Phone>999-999-9999</Phone>
     <Fax>888-888-8888</Fax>
     <Email>a@b.com</Email>
</ContactInfo>'
```

Listing 13.26

Attribute: namespace

This attribute specifies the target namespace of the wildcard element. We have seen the usage of target namespace earlier in this chapter. While processing the wildcard element, the schema processor will check for a matching schema definition having the namespace declaration specified in the *namespace* attribute of the wildcard declaration.

This attribute can take one of the following values:

- ##any
- ##other
- ##targetNamespace
- ##local
- space separated list of namespaces

Let us examine the usages of each of these values in the *namespace* attribute.

##any

This specifies that the schema processor will accept any namespace. The XML instance can specify any namespace that is present in the schema collection and the schema processor will accept it. Let us see an example in order to understand this.

Let us first create a schema collection that declares a wildcard element.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict" namespace="##any"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.27

Let us add a few more schema components from different namespaces to the above schema collection. Let us add the declaration of *Phone* element from *"www.phone.com"* namespace.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.phone.com">
  <xsd:element name="Phone">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
      </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.28

Now, let us add the declaration of *Fax* element from *"www.fax.com"* namespace.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.fax.com">
  <xsd:element name="Fax">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
```

```
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:element>
</xsd:schema>'
GO
```

Listing 13.29

Now let us try to validate a few XML instances against this schema collection. Note that we have set *processContents* to *"strict."* SQL Server will look for a matching namespace declaration and will validate the element against the definition given in the schema collection. If the namespace or element declaration is not found, the validation will fail.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
        <Phone xmlns="www.phone.com">123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.30

The above XML instance will validate successfully because the wildcard element is declared such that it can accept an element from any namespace. The namespace given in the XML instance exists in the schema collection and it has the declaration of an element named *Phone*. The value of *Phone* element in the XML instance validates successfully against the rules specified in the schema collection.

The following will succeed, as well, for the same reason.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
        <Fax xmlns="www.fax.com">123-123-1234</Fax>
</ContactInfo>'
```

Listing 13.31

##other

When the *namespace* attribute of a wildcard declaration is set to *"##other,"* the element in the XML instance can take any namespace other than the target namespace of the parent element.

The following schema collection declares an element named *ContactInfo* and specifies *"www.contactinfo.com"* as its target namespace.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.contactinfo.com">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict" namespace="##other"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.32

Now let us add another schema definition to the above schema collection, with the definition of *Phone* element having *"www.phone.com"* as the target namespace.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.phone.com">
  <xsd:element name="Phone">
    <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.33

Let us add the declaration of an element named *Fax* from the same namespace as the *ContactInfo* element.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.contactinfo.com">
  <xsd:element name="Fax">
    <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
    </xsd:simpleType>
```

```
    </xsd:element>
</xsd:schema>'
GO
```

Listing 13.34

Now, let us try to validate a few XML instances against the above schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
      <Phone xmlns="www.phone.com">123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.35

This will validate successfully. The *Phone* element is not in the same namespace as its parent element. The specified namespace exists in the schema collection and the namespace contains the declaration of the *Phone* element.

Now, let us try another XML instance.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
      <Fax xmlns="www.contactinfo.com">123-123-1234</Fax>
</ContactInfo>'
```

Listing 13.36

Validation of the above XML instance will fail with the following error.

```
XML Validation: Invalid content. Expected element(s):##other:*
where element 'www.contactinfo.com:Fax' was specified. Location:
/*:ContactInfo[1]/*:Fax[1]
```

Though there is a *Fax* element declared in the given namespace, the schema processor will not accept it because the wildcard is declared with the namespace attribute set to *"##other."* When *namespace* is set to *##other*, the element that represents the wildcard declaration cannot take the same namespace as the *targetNamespace* of the parent element.

##targetNamespace

The behavior of *##targetNamespace* is almost the opposite of *##other*. When *##other* is specified, the XML element that represents the wildcard declaration can take any namespace other than the target namespace. When *##targetNamespace* is specified, it should contain the same namespace as the target namespace of the parent element.

Let us create a new version of the schema collection with a wildcard element declaration having *namespace* attribute set to *##targetNamespace.*

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.contactinfo.com">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict"
                 namespace="##targetNamespace"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.37

Just as we did with the previous example, let us add the declaration of *Phone* element having a different target namespace.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.phone.com">
  <xsd:element name="Phone">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
        </xsd:restriction>
    </xsd:simpleType>
```

```
    </xsd:element>
</xsd:schema>'
GO
```

Listing 13.38

And finally, let us add the declaration of *Fax* element from the same namespace as the root element.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             targetNamespace="www.contactinfo.com">
  <xsd:element name="Fax">
     <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
     </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.39

Let us validate an XML instance against this version of the schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
     <Fax xmlns="www.contactinfo.com">123-123-1234</Fax>
</ContactInfo>'
```

Listing 13.40

The above XML instance will validate successfully against the schema collection. However, the following will fail.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
     <Phone xmlns="www.phone.com">123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.41

SQL Server will throw the following error if you try to run the above code.

```
XML Validation: Invalid content. Expected
element(s):www.contactinfo.com:* where element
'www.phone.com:Phone' was specified. Location:
/*:ContactInfo[1]/*:Phone[1]
```

The validation of the above XML instance fails because the schema specifies that the XML element that represents the wildcard declaration cannot take a namespace other than the target namespace of the parent element. Note that you need not declare the namespace in the *Fax* element because you already have it declared in the parent element. The following will validate successfully.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
    <Fax>123-123-1234</Fax>
</ContactInfo>'
```

Listing 13.42

##local

When the *namespace* attribute of a wildcard declaration is set to *##local*, the element in the XML instance that represents the wildcard declaration should not be part of any namespace. The schema collection should contain a declaration for the replacement element and it should not be associated with any namespace.

Let us create a schema collection that declares a wildcard element having namespace set to *##local.*

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="www.contactinfo.com">
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict"
                namespace="##local"/>
      </xsd:sequence>
    </xsd:complexType>
```

```
   </xsd:element>
 </xsd:schema>'
 GO
```

Listing 13.43

The schema definition in the above schema collection is created with a target namespace. Let us add the definition of the *Phone* element to the same namespace.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.contactinfo.com">
   <xsd:element name="Phone">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
</xsd:schema>'
 GO
```

Listing 13.44

The pattern in the above restriction specifies that the group of digits in the phone number be separated by a hyphen. (Take note of this. We will come back to this a little later).

Now let us add one more schema definition to the above schema collection.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
   <xsd:element name="Phone">
      <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                  <xsd:pattern value="\d{3} \d{3} \d{4}"/>
            </xsd:restriction>
      </xsd:simpleType>
   </xsd:element>
</xsd:schema>'
 GO
```

Listing 13.45

Note that this schema, too, defines a *Phone* element. However, the pattern restriction is slightly different. This specifies that the group of digits in the phone number be separated by space characters.

At this point, we have two definitions of *Phone* element. One version resides in the same namespace as the root element and accepts phone numbers having groups of digits separated by a hyphen (example: 999-999-9999). The other version of *Phone* element does not belong to any namespace and accepts phone numbers having space separated groups of digits (example: 999 999 9999).

Let us try to validate an XML instance against this schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
      <Phone>123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.46

If you run the above code, SQL Server will throw the following error.

```
XML Validation: Invalid content. Expected element(s):* where
element 'www.contactinfo.com:Phone' was specified. Location:
/*:ContactInfo[1]/*:Phone[1]
```

The *Phone* element in the above XML instance does not have a namespace. In the absence of a namespace declaration, the schema processor will associate the namespace of the parent element to a child element. The element in the XML instance that represents the wildcard element cannot take a namespace. Though there is a *Phone* element in the given namespace and the element validates successfully against that element, the schema processor will not accept it.

Here is the correct XML instance that validates with the above schema.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo xmlns="www.contactinfo.com">
      <Phone xmlns="">123 123 1234</Phone>
</ContactInfo>'
```

Listing 13.47

Setting *namespace* attribute to specific namespace values

It is also possible to associate the wild card elements to one or more specific namespaces. When done so, the XML element that represents the

wildcard element should contain one of the namespaces declared in the *namespace* attribute of the wild card element declaration.

Let us create a schema collection that declares an element named *ContactInfo.*

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
  <xsd:element name="ContactInfo">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:any processContents="strict"
                 namespace="www.phone.com www.fax.com"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.48

Note that the wildcard element accepts two namespaces. The XML instance element that represents the wildcard declaration can take only one of the namespaces declared above.

Let us add the declaration of *Phone* element from *"www.phone.com"* namespace to the above schema collection.

```sql
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.phone.com">
  <xsd:element name="Phone">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.49

Let us add one more schema definition to the above schema collection. The example given below adds the declaration of *Fax* element from *"www.fax.com"* to the schema collection.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.fax.com">
  <xsd:element name="Fax">
     <xsd:simpleType>
            <xsd:restriction base="xsd:string">
                    <xsd:pattern value="\d{3}-\d{3}-\d{4}"/>
            </xsd:restriction>
     </xsd:simpleType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 13.50

Let us validate a few XML instances against this schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
     <Phone xmlns="www.phone.com">123-123-1234</Phone>
</ContactInfo>'
```

Listing 13.51

This succeeds because *"www.phone.com"* is one of the namespaces declared with the wildcard element declaration.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<ContactInfo>
     <Fax xmlns="www.fax.com">123-123-1234</Fax>
</ContactInfo>'
```

Listing 13.52

This will succeed, too. *"www.fax.com"* is also one of the namespaces declared in the wildcard element declaration. The schema processor will not accept any namespace other than the namespaces declared in the wildcard element declaration.

Attribute Wildcards

In the previous section, we examined element wildcards. An element wildcard declaration represents an attribute in the XML instance that is not known at the time of writing the schema.

Attribute wildcards work very similar to element wildcards. When a complex type contains an attribute wildcard declaration the XML instance can contain replacement attributes from the specified namespace.

An attribute wildcard is declared using *"<xsd:anyAttribute>"* element. Let us see an example.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            targetNamespace="www.employeeinfo.com"
            attributeFormDefault="qualified">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee">
          <xsd:complexType>
            <xsd:anyAttribute
                        namespace="##any"
                        processContents="strict"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:attribute name="FName"/>
  <xsd:attribute name="LName"/>
  <xsd:attribute name="Number"/>
</xsd:schema>'
GO
```

Listing 13.53

The following XML instance validates against the above schema collection.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<emp:Employees xmlns:emp="www.employeeinfo.com">
```

```
      <Employee emp:FName="Jacob" />
</emp:Employees>'
```

Listing 13.54

It may be surprising to know that the following will be validated, too.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<emp:Employees xmlns:emp="www.employeeinfo.com">
      <Employee emp:FName="Jacob" emp:LName="Sebastian"
            emp:Number="1001" />
</emp:Employees>'
```

Listing 13.55

Your XML instance can contain any number of replacement attributes against an attribute wildcard declaration. Unlike element wildcards, attribute wildcards do not have *minOccurs* or *maxOccurs* attributes. Hence, the attributes can appear 0, one or more times.

An attribute wildcard declaration can take the following attributes.

- id
- processContents
- namespace

The behavior of these attributes is the same as we have seen with element wildcards. Refer to the section *"Element Wildcards"* above for a detailed discussion on these attributes.

Annotations

Annotations are used to add documentation to the schema. You can add annotations to any schema component. Schema documentation can be added using *documentation* as well as *appinfo* elements. The *documentation* element is used to add user documentation and *appinfo* element is used to add processing instructions for applications.

User Documentation

The following example shows a schema declaration with a basic documentation example.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation>
      This element stores employee information.
    </xsd:documentation>
  </xsd:annotation>  <xsd:element name="Employees"/>
</xsd:schema>
```

Listing 13.56

The *documentation* element can take a few attributes, using which you can add additional documentation. You can add references to external resources as well as specify language identifiers. The following example shows an annotation element having documentation in two languages.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:annotation>
    <xsd:documentation
        source="http://www.sqlserverandxml.com/employee/en-us"
        xml:lang="en-us">
      This element stores employee information.
    </xsd:documentation>
    <xsd:documentation
        source="http://www.sqlserverandxml.com/employee/it"
        xml:lang="it">
      Questo element negozi dipendente informazioni.
    </xsd:documentation>
  </xsd:annotation>
  <xsd:element name="Employees"/>
</xsd:schema>
```

Listing 13.57

Application Information

The *appinfo* element of an annotation is used to add processing information to applications. For example, an application that reads information from a table or writes to a table can use appinfo to store the mapping between XML elements and columns/tables. An example of such an application is the XMLBulkLoad component of SQLXML.

The following example shows a schema that uses *appinfo* element to store processing instructions for XMLBulkLoad component.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
      xmlns:sql="urn:schemas-microsoft-com:mapping-schema">
  <xsd:annotation>
    <xsd:appinfo>
      <sql:relationship
        name="CustomerProduct"
        parent="Customers" parent-key="CustomerID"
```

```
        child="Orders" child-key="CustomerID"/>
    </xsd:appinfo>
  </xsd:annotation>  <xsd:element name="Orders"
sql:relation="Orders">
    <!-- Other declarations here -->
  </xsd:element>
</xsd:schema>
```

Listing 13.58

Chapter Summary

In this chapter we have discussed a few advanced schema topics. We discussed the attributes of schema declarations and examined each attribute in detail. The only mandatory attribute of a schema declaration is the namespace declaration. A schema declaration can take a few optional attributes, also.

The *targetNamespace* attribute of the schema declaration can be used to specify a namespace to which the XML instance should belong.

The *attributeFormDefault* and *elementFormDefault* attributes specify the default value of the *form* attribute of all elements and attributes. The *blockDefault* and *finalDefault* attributes specify the default value for the *block* and *final* attributes of elements and types. The default value is used only if the element or attribute declaration does not contain the specified attribute.

Element and attribute wildcards can be used to create flexible schema components that can accept elements and attributes not known at the time the schema was written. SQL Server supports three modes to validate the content of wildcard declarations. When the validation mode is set to *skip,* SQL Server will skip the validation of the replacement element. When the validation mode is set to *strict*, SQL Server will locate the specified namespace in the schema collection and will validate the element or attribute against the element/attribute declaration given in the specified namespace. If the namespace is not found or it does not contain the declaration for the specified element or attribute, the validation will fail.

SQL Server 2008 supports a third validation mode: *lax.* When the processing mode is set to *lax,* SQL Server will validate the replacement element/attribute only if the specified namespace exists in the schema collection and the namespace contains the declaration for the specified element or attribute.

The behavior of element wildcards and attribute wildcards are almost similar, except that the XML instance can have 0, one or more attributes against a single attribute wild card declaration.

Annotations can be used to add documentation to the schema collection. The *documentation* element can be used to add user documentation and *appinfo* element can be used to add processing information for applications.

CHAPTER 14

SQL SERVER SCHEMA COLLECTIONS AND METADATA

SQL Server maintains XML schemas in schema collections. A schema collection is a system object just like tables, stored procedures, indexes, etc. A schema collection can store more than one schema definition. In this chapter we will examine XML Schema Collections in detail. We will discuss:

- Why is there a collection?
- How to create schema 'collections'
- How to alter schema collections
- Retrieving schema definition from SQL Server
- Adding constraints using facets: CONTENT and DOCUMENT
- XML Schema Collection Metadata
- Limitations of SQL Server Schema Collections

Why schema 'collection'?

It is not clear to me why there is no *xml schema* object in SQL Server. Instead, what we have is *xml schema collection* objects. Each *xml schema collection* object stores one or more schema definitions, so you need an *xml schema collection* even for storing a single *schema definition.*

Though a schema collection can store more than one schema definition, most of the time you would prefer to store only one schema definition in it. When you validate an XML instance against a schema collection, SQL Server will validate the XML instance against all the global elements declared in the entire schema collection until a suitable match is found. If the XML instance validates against any of the global elements declared in the entire schema collection, the insert/update/assignment operation will succeed.

A Schema Collection Example

Let us look at a schema collection having more than one schema definition. Let us think of an application used by a training school. They have a table named *"person"* that stores information of students as well as trainers. Assume that they need to store some additional information which is specific to the type of data stored in each row. For example, if the row stores information regarding a student they need to store the course and duration of the course in which he or she is enrolled. If the row stores information of a trainer, they need to store the qualification of the trainer and the details of industry experience he or she has.

Let us try to use an XML column to store this additional information. Let us create a schema collection that contains two schema definitions: one to validate student information and the other to validate trainer information.

Here is an example of student information.

```
<Student>
   <Course>SQL Server Training</Course>
   <Duration>6 Months</Duration>
</Student>
```

Listing 14.1

And the following example shows the structure of the trainer information XML document.

```
<Trainer>
   <Qualification>MCDBA</Qualification>
   <IndustryExperience>10 Years</IndustryExperience>
</Trainer>
```

Listing 14.2

Let us create a schema collection containing the schema definition to validate student information.

```
CREATE XML SCHEMA COLLECTION StudentOrTrainer AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <!-- Student Element -->
   <xsd:element name="Student">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Course"/>
         <xsd:element name="Duration"/>
       </xsd:sequence>
```

```
      </xsd:complexType>
    </xsd:element>
</xsd:schema>'
GO
```

Listing 14.3

You can add a new schema definition to an existing schema collection by using ALTER SCHEMA COLLECTION ADD command. Let us add the schema definition of the trainer information to the above schema collection.

```
ALTER XML SCHEMA COLLECTION StudentOrTrainer ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Qualification"/>
        <xsd:element name="IndustryExperience"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 14.4

Since this schema collection has the definition for a *Student* element as well as *Trainer* element, this will accept both XML instances that we saw earlier.

```
DECLARE @x XML(StudentOrTrainer)

-- Student Information
SELECT @X = '
<Student>
  <Course>SQL Server Training</Course>
  <Duration>6 Months</Duration>
</Student>'

-- Trainer Information
SELECT @X = '
<Trainer>
  <Qualification>MCDBA</Qualification>
  <IndustryExperience>10 Years</IndustryExperience>
</Trainer>'
```

Listing 14.5

The first schema definition has a global element declaration named *Student*. The second schema definition contains a global element declaration named *Trainer*. By adding both schema definitions to the same

schema collection, we could make sure that SQL Server will accept XML instances that validates against either one of the global elements declared in the schema collection.

It can also accept an XML instance that contains both *Student* and *Trainer* elements. The following XML instance will validate successfully.

```
DECLARE @x XML(StudentOrTrainer)

-- Student Information
SELECT @X = '
<Student>
  <Course>SQL Server Training</Course>
  <Duration>6 Months</Duration>
</Student>
<Trainer>
  <Qualification>MCDBA</Qualification>
  <IndustryExperience>10 Years</IndustryExperience>
</Trainer>'
```

Listing 14.6

Though we created two separate schema definitions and added them to the same schema collection, SQL Server will merge the definitions and will internally maintain only one schema definition. So the code shown in the following two code snippets are equivalent.

```
CREATE XML SCHEMA COLLECTION StudentOrTrainer AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Student Element -->
  <xsd:element name="Student">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Course"/>
        <xsd:element name="Duration"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

ALTER XML SCHEMA COLLECTION StudentOrTrainer ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Qualification"/>
        <xsd:element name="IndustryExperience"/>
      </xsd:sequence>
```

```
      </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 14.7: This example shows a schema collection that contains the definition of two global elements. Each element is added separately to the schema collection.

The above schema collection is equivalent to the following.

```
CREATE XML SCHEMA COLLECTION StudentOrTrainer AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <!-- Student Element -->
   <xsd:element name="Student">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Course"/>
         <xsd:element name="Duration"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
   <!-- Trainer Element -->
   <xsd:element name="Trainer">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Qualification"/>
         <xsd:element name="IndustryExperience"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 14.8: This example shows a schema collection having the definition of two global elements.

> **When you add a new schema definition to an existing schema collection, SQL Server will merge the schema definitions and group them by target namespace. We will see this later in this chapter.**

Altering a schema collection

I had mentioned several times earlier that you cannot alter a schema collection. Well, you can alter a schema collection to add more schema components. You can use ALTER XML SCHEMA COLLECTION command to add new components to an existing schema collection. We have seen an example of this in the previous section.

Though you can add a new schema component to an existing schema collection, you cannot modify the definition of existing schema components. The following example creates a schema collection with a *Student* element.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection with 'Student' element
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Student Element -->
  <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.9

You can alter this schema collection and add additional components. For example, the following code adds a *Trainer* element to this schema collection.

```
-- Add a new element 'trainer'
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer" type="xsd:string"  />
</xsd:schema>'
GO
```

Listing 14.10

However, you cannot alter the definition of the *Student* element. Let us try to change the *type* of *Student* element.

```
-- Add a new element 'trainer'
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Student" type="xsd:token"  />
</xsd:schema>'
GO
```

Listing 14.11

This will not run. SQL Server will generate the following error if you try to run this code.

```
Altering existing schema components is not allowed.  There was an
attempt to modify an existing XML Schema component, component
namespace: '' component name: 'Student' component kind:ELEMENT
```

If you want to modify the definition of an existing schema component, you need to drop the schema collection and create the schema collection again with the updated definition.

But SQL Server will not allow you to drop a schema collection if the schema collection is bound to an XML column or a stored procedure or function has a TYPED XML parameter bound to the given schema collection. So before you drop a schema collection, you need to remove all references to the given schema collection from columns, functions and stored procedures, etc.

This makes modification of schema definitions very difficult.

You can remove the binding of a schema collection with a column using the following TSQL code.

```
ALTER TABLE tableName ALTER COLUMN columnName XML
```

Listing 14.12

The above code changes the column to an UNTYPED XML column. You can also alter the column to some other XML schema collection if all the values in the given column are valid for the other schema collection.

Let us try to understand this by looking at an example. Create the following schema collection that validates student information.

```
CREATE XML SCHEMA COLLECTION StudentSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Student Element -->
  <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.13

Let us create one more schema collection that validates trainer information.

```
CREATE XML SCHEMA COLLECTION TrainerSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.14

Let us create a table having a TYPED XML column bound to the *SudentSchema* schema collection.

```
CREATE TABLE Person(PersonName VARCHAR(20), Data
XML(StudentSchema))
```

Listing 14.15

Let us insert a few rows to the above table.

```
INSERT INTO Person(PersonName, Data)
     SELECT 'Jacob', '<Student>Some StudentInfo</Student>'
INSERT INTO Person(PersonName, Data)
     SELECT 'Mike', '<Student>Some More StudentInfo</Student>'
```

Listing 14.16

At this point, the schema collection *StudentSchema* is bound to the *Data* column of *Person* table. I had mentioned earlier that, if you want to alter the definition of an existing schema collection, you need to drop the schema collection and create it again with the new definition.

Let us try to drop this schema collection and see if SQL Server allows that.

```
DROP XML SCHEMA COLLECTION StudentSchema
```

Listing 14.17

SQL Server will not allow you to drop this schema collection because it is bound to a column. If you try to run the above code, you will get the following error.

```
Specified collection 'StudentSchema' cannot be dropped because it
is used by object 'dbo.Person'.
```

So before you drop the schema collection, you need to change the schema binding. You need to remove the binding between the schema collection

and the XML columns as well as the TYPED XML parameters of functions and stored procedures. You can run the following code to unbind the *Data* column.

```
ALTER TABLE Person ALTER COLUMN Data XML
```

Listing 14.18

This will set the column to an UNTYPED XML column. Alternatively, you can bind the column to another schema collection if the data in the column validates with that schema collection.

When you alter the definition of an XML column and bind it to another XML schema collection, SQL Server will validate the values in the column with the new schema collection. The ALTER operation will succeed only if all the values in the column validate successfully with the new schema collection. For example, the following will fail.

```
ALTER TABLE Person ALTER COLUMN Data XML(TrainerSchema)
```

Listing 14.19

The values currently stored in the *"Data"* column contain XML fragments that validate with *StudentSchema* but not with *TranerSchema;* hence, the ALTER operation will fail with the following error.

```
XML Validation: Declaration not found for element 'Student'.
Location: /*:Student[1]
The statement has been terminated.
```

Just as in the case with columns, SQL Server will not allow you to drop a schema collection if any of the stored procedures or functions has a TYPED XML parameter bound to the given schema collection. Before you can drop the schema collection, you need to remove such references either by turning the parameters to UNTYPED XML or changing them to anther schema collection.

To summarize, altering schema collections is not very easy. You need to be very careful while writing your schema definitions so that frequent alterations of schema collections can be avoided.

Retrieving Schema Definition

Sometimes you might need to retrieve the definition of a schema collection from SQL Server. You can use *xml_schema_namespace()* function to retrieve the definition of a schema collection stored internally by SQL Server.

To experiment with this, let us create a new schema collection.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

CREATE XML SCHEMA COLLECTION ExampleSchema AS'
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Student Element -->
  <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.20

Let us try to retrieve the definition of this schema collection from SQL Server.

```
SELECT xml_schema_namespace('dbo','ExampleSchema')
```

Listing 14.21

The above code returns the following.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Student" type="xsd:string" />
</xsd:schema>
```

Listing 14.22: Schema definition retrieved from SQL Server.

Let us alter the schema collection and add one more element.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD'
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer" />
</xsd:schema>'
GO
```

Listing 14.23

This will add a new element to the schema collection that we created earlier. Now let us try to retrieve the modified schema definition from SQL Server.

```
SELECT xml_schema_namespace('dbo','ExampleSchema')
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Student" type="xsd:string" />
  <xsd:element name="Trainer" type="xsd:anyType" />
</xsd:schema>
```

Listing 14.24: Schema definition retrieved from SQL Server.

Note that SQL Server will strip off comments from the schema definition. Further, SQL Server will apply small modifications to your schema. In the above example, we did not specify the *type* attribute for the *Trainer* element and SQL Server added the *Type* attribute and set it to *anyType.*

Such modifications do not alter the meaning of your schema, nor do they change the validation rules. When you retrieve the definition of a schema collection for further modification you might find it bit confusing, because SQL Server will give you a slightly modified version of the original schema. Further, all the comments you had written will be gone, too. To make editing schema collections easier, you need to keep a copy of your schema definitions in an XML file or in a table. When you need to modify a schema collection, you can take the version of the schema that you have (in files or a table), modify it and recreate the schema collection.

It is possible to create a schema collection from a variable. Hence, it is possible to create a stored procedure that reads schema definition from a file or from a table and creates the schema collections. The following code snippets show how to create schema collections from a variable, from a table or from a disk file.

```
DECLARE @xml XML
SELECT @xml = '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <!-- Student Element -->
   <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'

CREATE XML SCHEMA COLLECTION SchemaFromVariable AS @xml
GO
```

Listing 14.25: Creating a schema collection from a variable.

```
DECLARE @xml XML

-- Load the schema definition from a file
SELECT @xml =CONVERT(XML, bulkcolumn, 2)
FROM OPENROWSET(BULK 'C:\temp\SchemaFile.xml', SINGLE_BLOB) AS x

-- create schema collection
CREATE XML SCHEMA COLLECTION SchemaFromFile AS @xml
GO
```

Listing 14.26: Creating a schema collection from the definition taken from a file.

```
DECLARE @xml XML

-- Load the schema definition from a table
SELECT
      @xml = SchemaDefinition
FROM SchemaTable
WHERE SchemaName = 'SchemaFromTable'

-- create schema collection
CREATE XML SCHEMA COLLECTION SchemaFromTable AS @xml
GO
```

Listing 14.27: Creating a schema collection from the definition taken from a table.

Multiple Target Namespaces

In the previous chapter we examined the *targetNamespace* attribute of schema declaration. This attribute is used to specify a namespace to which the elements in the XML instance should belong.

When you add more than one schema definition to a single schema collection, SQL Server groups schema definitions by target namespace. Let us revisit one of the examples we saw earlier in this chapter.

The following example shows a schema collection that stores student information.

```
CREATE XML SCHEMA COLLECTION ExampleSchema AS'
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Student Element -->
  <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.28

After creating the schema collection for storing student information, we added one more schema definition to the schema collection by using ALTER XML SCHEMA COLLECTION command.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD'
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <!-- Trainer Element -->
  <xsd:element name="Trainer" />
</xsd:schema>'
GO
```

Listing 14.29

When we added the second schema definition to the existing schema collection, SQL Server grouped the schema definitions together. We received the following schema when we retrieved the definition of the schema collection from SQL Server.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Student" type="xsd:string" />
  <xsd:element name="Trainer" type="xsd:anyType" />
</xsd:schema>
```

Listing 14.30: Definition of a schema collection retrieved from SQL Server.

Note that SQL Server merged the schema definitions to a single schema. This merging is done based on the target namespace declaration of each schema component. In the above example, both schema definitions did not have a target namespace declaration. Hence, SQL Server merged both schema definitions to a single schema.

Now let us see what happens if the schema definitions have different target namespace declarations.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
     SELECT * FROM sys.xml_schema_collections
     WHERE name = 'ExampleSchema'
) BEGIN
     DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection with 'Student' element
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.student.com">
  <xsd:element name="Student"  type="xsd:string" />
</xsd:schema>'
GO
```

Listing 14.31

Note that the student information element has a *targetNamespace* declaration that insists that the XML instance have the specified namespace declaration.

Now let us add the declaration of the *Trainer* element having a different target namespace declaration.

```
ALTER XML SCHEMA COLLECTION ExampleSchema ADD '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.trainer.com">
  <xsd:element name="Trainer" type="xsd:string"  />
</xsd:schema>'
GO
```

Listing 14.32

The *Trainer* element is declared with a different namespace declaration. The following are the valid XML instances that validate with the above version of the schema collection.

```
DECLARE @x XML(ExampleSchema)

-- student information
SELECT @x = '
<Student xmlns="http://www.student.com">
     Student Info
</Student>'

-- trainer information
SELECT @x = '
<Trainer xmlns="http://www.trainer.com">
     Trainer Info
</Trainer>'

-- Both student and trainer information
```

```
SELECT @x = '
<Student xmlns="http://www.student.com">
      Student Info
</Student>
<Trainer xmlns="http://www.trainer.com">
      Trainer Info
</Trainer>'
```

Listing 14.33

Now, let us retrieve the definition of this schema collection from SQL Server and see what it looks like.

```
SELECT xml_schema_namespace('dbo','ExampleSchema')
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:t="http://www.student.com"
            targetNamespace="http://www.student.com">
  <xsd:element name="Student" type="xsd:string" />
</xsd:schema>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:t="http://www.trainer.com"
            targetNamespace="http://www.trainer.com">
  <xsd:element name="Trainer" type="xsd:string" />
</xsd:schema>
```

Listing 14.34

You will find this schema slightly different from what we have seen previously. Again, you will see that it is a bit different from the schema definition you provided when you created the schema collection. Note the additional declaration of the namespace with prefix: *"t."* As I mentioned earlier in this chapter, SQL Server will slightly modify your schema definition without altering the meaning of your schema collection. It looks like SQL Server shreds the XML schema definition to a set of relational tables internally and re-assembles the schema when you call the *xml_schema_namespace()* function.

The above example shows that SQL Server groups schema definitions by target namespace. If you would like to retrieve the schema definition having a specific target namespace declaration, you can do it by specifying the target namespace as the third argument to the *xml_schema_namespace()* function. The following example retrieves the schema definition of the student element.

```
SELECT xml_schema_namespace('dbo','ExampleSchema',
      'http://www.student.com')
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:t="http://www.student.com"
```

```
            targetNamespace="http://www.student.com">
   <xsd:element name="Student" type="xsd:string" />
</xsd:schema>
```

Listing 14.35

You can achieve the same result by running the following XQuery method as well.

```
SELECT xml_schema_namespace('dbo','ExampleSchema').query('
/xs:schema[@targetNamespace="http://www.student.com"]
')
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:t="http://www.student.com"
            targetNamespace="http://www.student.com">
   <xsd:element name="Student" type="xsd:string" />
</xsd:schema>
```

Listing 14.36

Adding constraints using facets

A TYPED XML column or variable has two facets that control the type of XML value it can store. There are two facets: CONTENT and DOCUMENT. When you declare a TYPED XML variable or column, SQL Server automatically associates the CONTENT facet to it. When a TYPED XML column or variable is declared with CONTENT facet, it can store XML values having multiple top level elements. When it is declared with DOCUMENT facet, the XML value should have a single root element.

The following XML instance contains multiple top level elements. This can be stored to a TYPED XML column or variable declared with CONTENT facet.

```
<Employee>
   <Name>Jacob</Name>
</Employee>
<Employee>
   <Name>Mike</Name>
</Employee>
```

Listing 14.37

The XML instance given below has a single root element and can be stored to a TYPED XML column or variable declared with CONTENT or DOCUMENT facet.

```
<Employee>
   <Name>Mike</Name>
</Employee>
```

Listing 14.38

Let us create a schema collection that describes the above XML instance.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="Employee">
     <xsd:complexType>
       <xsd:sequence>
         <xsd:element name="Name"/>
       </xsd:sequence>
     </xsd:complexType>
   </xsd:element>
</xsd:schema>'
GO
```

Listing 14.39

Let us declare an XML variable with CONTENT facet.

```
DECLARE @x XML(CONTENT ExampleSchema)
SELECT @x = '
<Employee>
   <Name>Jacob</Name>
</Employee><Employee>
   <Name>Mike</Name>
</Employee>'
```

Listing 14.40

If you have not specified a facet, SQL Server will associate all TYPED XML columns and variables with CONTENT facet. Hence, the following is equivalent to the above example.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '
<Employee>
   <Name>Jacob</Name>
</Employee>
```

```
<Employee>
  <Name>Mike</Name>
</Employee>'
```

Listing 14.41

The example given below declares a typed XML variable with DOCUMENT facet.

```
DECLARE @x XML(DOCUMENT ExampleSchema)
SELECT @x = '
<Employee>
  <Name>Mike</Name>
</Employee>'
```

Listing 14.42

A TYPED XML column or variable having DOCUMENT facet can only store XML instances having a single root element. If you try to store XML instances with more than one top level element, SQL server will raise an error.

```
DECLARE @x XML(DOCUMENT ExampleSchema)
SELECT @x = '
<Employee>
  <Name>Jacob</Name>
</Employee>
<Employee>
  <Name>Mike</Name>
</Employee>'
```

Listing 14.43

If you try to run this code, SQL Server will generate the following error.

```
XML Validation: XML instance must be a document.
```

The following example creates a table having two TYPED XML columns: one with CONTENT facet and the other DOCUMENT facet.

```
CREATE TABLE FacetTest (
    documentData XML(DOCUMENT ExampleSchema),
    contentData XML(CONTENT ExampleSchema) )
GO
```

Listing 14.44

XML Schema Collection Metadata

SQL Server maintains information about schema collections in system metadata. There are a number of system catalog views that you can use to query information related to your schema collections. The following are the system catalog views that provide information about XML Schema Collections.

- **sys.xml_schema_collections**
- **sys.xml_schema_components**
- **sys.xml_schema_elements**
- **sys.xml_schema_attributes**
- **sys.xml_schema_model_groups**
- **sys.xml_schema_types**
- **sys.xml_schema_wildcards**
- **sys.xml_schema_namespaces**
- **sys.xml_schema_wildcard_namespaces**
- **sys.xml_schema_facets**
- **sys.xml_schema_component_placements**
- **sys.column_xml_schema_collection_usages**
- **sys.parameter_xml_schema_collection_usages**
- **sys.xml_indexes**

Let us examine each of these catalog views briefly.

Please refer to Books Online for a detailed explanation of the columns returned by each catalog view.

SQL Server 2005:

http://msdn.microsoft.com/en-us/library/ms189815(SQL.90).aspx

SQL Server 2008:

http://msdn.microsoft.com/en-us/library/ms189815.aspx

sys.xml_schema_collections

This catalog view retrieves all the schema collections in the current database. This view will have one row per schema collection.

To retrieve the information about the *ExampleSchema* we created earlier, run the following query.

```
SELECT *
FROM sys.xml_schema_collections c
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.45

sys.xml_schema_components

This view returns a row per XML schema component. It includes all the elements, attributes, model groups, simple types, complex types, etc. You can run the following query to retrieve the details of all the schema components of a given schema collection.

```
SELECT sc.*
FROM sys.xml_schema_components sc
INNER JOIN sys.xml_schema_collections c
      ON sc.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.46

sys.xml_schema_elements

This view returns a row per xml schema element declaration. The information this view returns is a subset of the information provided by the catalog view: *xml_schema_components.* You can join this view with *sys.xml_schema_collections* and retrieve all the elements declared in a given schema collection. Here is an example:

```
SELECT e.*
FROM sys.xml_schema_elements e
INNER JOIN sys.xml_schema_collections c
      ON e.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.47

sys.xml_schema_attributes

This view returns a row per xml schema attribute declaration. The information this view returns is a subset of the information provided by the catalog view: *xml_schema_components.* You can join this view with *sys.xml_schema_collections* and retrieve all the attributes declared in a given schema collection. Here is an example:

```sql
SELECT a.*
FROM sys.xml_schema_attributes a
INNER JOIN sys.xml_schema_collections c
    ON a.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.48

sys.xml_schema_model_groups

This view returns a row per model group declaration (*sequence*, *choice* and *all*) in all the schema collections. The information this view returns is a subset of the information provided by the catalog view: *xml_schema_components.* You can join this view with *sys.xml_schema_collections* and retrieve all the attributes declared in a given schema collection. Here is an example:

```sql
SELECT mg.*
FROM sys.xml_schema_model_groups mg
INNER JOIN sys.xml_schema_collections c
    ON mg.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.49

sys.xml_schema_types

This view returns a row per *model group*, *simple type* and *complex type* declarations in all the schema collections. The information this view returns is a subset of the information provided by the catalog view: *xml_schema_components.* You can join this view with *sys.xml_schema_collections* and retrieve all the attributes declared in a given schema collection. Here is an example:

```sql
SELECT t.*
FROM sys.xml_schema_types t
INNER JOIN sys.xml_schema_collections c
```

```
        ON t.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.50

sys.xml_schema_wildcards

This view returns a row per element wildcard declaration or attribute wildcard declaration from all the schema collections in the current database. The information provided by this view is a subset of the information provided by catalog view: *sys.xml_schema_components.* The following query returns all the element or attribute wildcard declarations of a given schema collection.

```
SELECT sw.*
FROM sys.xml_schema_wildcards sw
INNER JOIN sys.xml_schema_collections c
        ON sw.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.51

Refer to Chapter 13 for a detailed discussion on element and attribute wildcards.

sys.xml_schema_namespaces

This view returns a row per *target namespace* declaration. You can join this view with *sys.xml_schema_collections* and retrieve all the attributes declared in a given schema collection. Here is an example:

```
SELECT n.*
FROM sys.xml_schema_namespaces n
INNER JOIN sys.xml_schema_collections c
        ON n.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.52

sys.xml_schema_wildcard_namespaces

This view returns a row per wildcard namespace declaration. To retrieve the wildcard namespace declarations of a given schema collection, run the following query:

```
SELECT *
FROM sys.xml_schema_wildcard_namespaces wn
INNER JOIN sys.xml_schema_components sc
    ON sc.xml_component_id = wn.xml_component_id
INNER JOIN sys.xml_schema_collections c
    ON sc.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.53

Refer to Chapter 13 for a detailed discussion on wildcard namespaces.

sys.xml_schema_facets

This view returns all the facets of all simple types where a restriction is applied. You can read the facets used in a schema collection by running the following query.

```
SELECT
    sc.[name] AS ComponentName,
    sc.[kind_desc] AS ComponentType,
    sf.*
FROM sys.xml_schema_facets sf
INNER JOIN sys.xml_schema_components sc
    ON sf.xml_component_id = sc.xml_component_id
INNER JOIN sys.xml_schema_collections c
    ON sc.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.54

sys.xml_schema_component_placements

This view contains information about the placement (minimum and maximum occurrences) of each schema component. The following query shows how to read the placement information of a schema collection.

```
SELECT
      sc.[name] AS ComponentName,
      sc.[kind_desc] AS ComponentType,
      scp.*
FROM sys.xml_schema_component_placements scp
INNER JOIN sys.xml_schema_components sc
      ON scp.xml_component_id = sc.xml_component_id
INNER JOIN sys.xml_schema_collections c
      ON sc.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.55

sys.column_xml_schema_collection_usages

This view provides information about columns that are bound to schema collections. To find all the columns bound to a specific schema collection, you can run the following query.

```
SELECT tab.name AS TableName, col.name AS ColumnName
FROM sys.column_xml_schema_collection_usages cu
INNER JOIN sys.columns col
      ON col.object_id = cu.object_id
      AND col.column_id = cu.column_id
INNER JOIN sys.tables tab ON tab.object_id = col.object_id
INNER JOIN sys.xml_schema_collections c
      ON cu.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.56

sys.parameter_xml_schema_collection _usages

This view provides information about stored procedures and functions that takes a TYPED XML parameter. To find all the functions and stored procedures that take an XML parameter bound to a given schema collection, run the following query.

```
SELECT
      ao.name AS ObjectName,
      ao.type_desc AS ObjectType
FROM sys.parameter_xml_schema_collection_usages pu
INNER JOIN sys.all_objects ao
      ON pu.object_id = ao.object_id
```

```
INNER JOIN sys.xml_schema_collections c
      ON pu.xml_collection_id = c.xml_collection_id
WHERE c.[name] = 'ExampleSchema'
```

Listing 14.57

sys.xml_indexes

This view returns a row per xml index in the current database. SQL Server allows you to index xml columns. A detailed discussion on XML indexes is beyond the scope of this book. Please refer to Books Online for additional information on XML indexes. You can run the following query to retrieve information about all the XML indexes.

```
SELECT *
FROM sys.xml_indexes
```

Listing 14.58

SQL Server XSD Implementation – Limitations

The XSD implementation of SQL Server has a number of limitations. SQL Server 2008 added some enhancements to the XSD support of SQL Server 2005 and removed some of the limitations. Let us look at the limitations that the XSD implementation of SQL Server has.

SQL Server 2005 does not support Lax Validation

SQL Server 2005 did not support *lax* validation. We have seen lax validation in Chapter 13. Support for *lax* validation is added in SQL Server 2008.

Maximum value limit for minOccurs and maxOccurs

SQL Server has a limit in the maximum value that you can assign to the *minOccurs* and *maxOccurs* attributes. The values of these attributes

cannot be more than the size a 4 byte integer data type can store. The following will generate an error.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Employee" maxOccurs="2147483648"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 14.59

If you try to create a schema collection with the above XML schema definition, SQL Server will generate the following error.

```
Invalid value '2147483648' for the maxOccurs attribute. The value
has to be between 0 and 2147483647.
```

SQL Server 2005 does not support sqltypes:datetime and sqltypes:smalldatetime

SQL Server comes with an internal schema collection that defines data type mapping between SQL Server data types and XSD data types. When you create a schema collection, you can declare elements and attributes of the types defined in this schema collection.

SQL Server 2005's implementation of date/time data types requires that the values should have time zone information. But the definition of *sqltypes:datetime* and *sqltypes:smalldatetime* do not accept time zone information. So these data types are unusable in SQL Server 2005.

Look at the following schema collection.

```xsd
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:sqltypes="http://schemas.microsoft.com/sqlserver/
      2004/sqltypes">
  <xsd:import
 namespace="http://schemas.microsoft.com/sqlserver/2004/sqltypes"
    schemaLocation="http://schemas.microsoft.com/sqlserver/2004/
        sqltypes/sqltypes.xsd"/>
  <xsd:element name="date" type="sqltypes:datetime"/>
</xsd:schema>
```

Listing 14.60

The above schema collection declares an element named *"date"* and specifies the type of the element as *"sqltypes:datetime."* You can create a schema collection with this schema definition in SQL Server 2005 as well as SQL Server 2008. However, you can successfully validate an XML instance against this schema collection only in SQL Server 2008.

The following will work only in SQL Server 2008.

```
DECLARE @x XML(ExampleSchema)
SET @x='<date>2002-10-10T12:00:00</date>'
```

Listing 14.61

SQL Server does not support <xsd:include> element

SQL Server does not support *<xsd:include>* element. *<xsd:include>* is used to add the schema definition from the specified schema document to the current schema collection. SQL Server will generate an error if this element is present in the schema definition.

SQL Server does not support XSD schema components: <xsd:redefine>, <xsd:key>, <xsd:keyref> and <xsd:unique>

The XSD implementation of SQL Server 2005 and 2008 does not support the above schema components. If your schema definition contains any of those components, SQL Server will generate an error.

Limitation of *Choice* Groups

SQL Server does not allow a choice group to be empty unless the group is declared with *minOccurs* attribute having a value of 0. The following schema is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="PaymentMethod">
     <xsd:complexType>
       <xsd:choice />
```

```
      </xsd:complexType>
    </xsd:element>
</xsd:schema>
```

Listing 14.62

If you try to create a schema collection with the above schema definition, SQL Server will generate the following error.

```
Choice may not be empty unless minOccurs is 0
```

The following is valid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="PaymentMethod">
    <xsd:complexType>
      <xsd:choice minOccurs="0"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

Listing 14.63

Limitation of *Simple Types*

SQL Server does not support *NaN* value in simple type declarations. We saw this limitation when we discussed the XSD data types in Chapter 7.

The following are the other limitations that SQL Server applies on simple type values.

Data Type: duration

The *year* part has to be within the range of -2^{31} to $2^{31}-1$. The *month*, *day*, *hour*, *minute*, and *second* must all be within the range of 0 to 9999. The *seconds'* part has an additional three digits of precision to the right of the decimal point.

Data Type: date, time and dateTime

The hour part in the time zone subfield must be within the accepted range of -14 to +14. The year part must be within the range of 1 to 9999. The month part must be within the range of 1 to 12. The day part must be within the range of 1 to 31 and must be a valid calendar date

The *seconds'* part of a *time* and *dateTime* value in SQL Server 2005 could only store values with precision up to milliseconds. The following example demonstrates it.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderDate" type="xsd:dateTime"/>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '<OrderDate>2008-01-01T10:23:12.657913Z</OrderDate>'

SELECT @x
/*
<OrderDate>2008-01-01T10:23:12.658Z</OrderDate>
*/
```

Listing 14.64

Note that SQL Server 2005 rounded *657913* to *658*.

The *seconds'* part of a *time* and *dateTime* value in SQL Server 2008 can store values with precision up to 100 nanoseconds. Let us run the above code in SQL Server 2008.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="OrderDate" type="xsd:dateTime"/>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '<OrderDate>2008-01-01T10:23:12.657913</OrderDate>'
```

```
SELECT @x
/*
<OrderDate>2008-01-01T10:23:12.657913</OrderDate>
*/
```

Listing 14.65

In SQL Server 2005, *date, time* and *dateTime* values must contain time zone information. Further, SQL Server 2005 does not preserve the time zone information. It normalizes the date/time value to UTC date/time. This restriction has been removed in SQL Server 2008. In SQL Server 2008, time zone information is optional and if present, it will be preserved. This is explained in detail in Chapter 7.

In SQL Server 2005, a *date* or *dateTime* value can take a negative value in the *year* part. The following is valid in SQL server 2005.

```
DECLARE @x XML(ExampleSchema)
SELECT @ = '<OrderDate>-9999-01-01T10:23:12.123Z</OrderDate>'
```

Listing 14.66

However, SQL Server 2008 does not support negative years; hence, the above code will generate an error. Under SQL Server 2008, the *date* part of a *date* or *dateTime* value should be between 1 and 9999.

SQL Server does not preserve xsi:schemaLocation and xsi:noNamespaceSchemaLocation attributes in XML Instance

If an XML instance contains *xsi:schemaLocation* or *xsi:noNamespace SchemaLocation* attribute, SQL Server will ignore these values and the value of these attributes are not preserved in the data stored into the XML column or variable.

Limitations of *xsd:QName*

SQL Server does not allow deriving from *xsd:QName* by restriction. The following is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="RootElement">
    <xsd:simpleType>
      <xsd:restriction base="xsd:QName">
```

```
            <xsd:maxLength value="50"/>
        </xsd:restriction>
      </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

Listing 14.67

If you try to create a schema collection with the above schema definition, SQL Server will throw the following error.

```
Derivation from 'QName' by restriction is not supported in this
release
```

You cannot use *QName* as one of the members in a union type. The following is illegal.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <xsd:element name="RootElement">
     <xsd:simpleType>
       <xsd:union memberTypes ="xsd:QName xsd:string"/>
     </xsd:simpleType>
   </xsd:element>
</xsd:schema>
```

Listing 14.68

SQL Server will generate the following error if you attempt to create a schema definition with the above definition.

```
XML Validation: Invalid definition for type 'xs-
nun(/RootElement/simpleType())'. SQL Server does not currently
support inclusion of ID, QName, or list of QName among the member
types of a union type.
```

Limitations on adding members to a Substitution Group

We have examined substitution groups in Chapter 5. Substitution groups help to create variable content containers which give a good deal of flexibility to the schema.

Earlier in this chapter, we have seen that you can build a schema collection incrementally by using ALTER SCHEMA command. You can create a schema collection with the declaration of some components and then you

can use ALTER SCHEMA command to add more schema components to the schema collection.

However, you can't do this with the components that are part of a substitution group. If you try to do that, SQL Server will generate the following error.

```
SQL Server does not currently permit additions to existing
substitution groups via ALTER XML SCHEMA COLLECTION.
```

Limitations on pattern restriction

We have discussed lexical representation in Chapter 7. The lexical representation of *boolean* data type includes *true, false, 1* and *0.* SQL Server accepts *boolean true* value as *"true"* or as *"1."* Similarly, it accepts *boolean false* values as *"false"* or *"0."*

So, SQL Server accepts *boolean true* value in two different forms, as it has two lexical representations. However, when SQL Server serializes the values for output, it will use the *canonical representation.* The canonical representation of *boolean* value is *"true"* or *"false."*

Look at the following example.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Active" type="xsd:boolean"/>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO

DECLARE @x XML(ExampleSchema)
SELECT @x = '<Employee Active="1"/>'

SELECT @x
```

```
/*
<Employee Active="true" />
*/
```

Listing 14.69

Note that we used the lexical representation *"1"* in our XML instance, but when SQL Server generated the output it used the *canonical* representation. Instead of *"1"* you received back *"true."*

Usually this is not a problem. But if you have a pattern restriction on the simple type that does not allow the canonical representation, then this can cause an error. Let us look at an example.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Active">
        <xsd:simpleType>
          <xsd:restriction base="xsd:boolean">
            <xsd:pattern value="(0|1)"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 14.70

Look at the pattern restriction in this schema collection. It restricts the value of the attribute to *"1"* or *"0."* So what is the problem with this?

Well, this can cause you some trouble. Look at the following case. Assume that you created a schema collection with the above schema definition and created an XML variable bound to that schema collection. Suppose that you want to store the following value to the variable.

```
<Employee Active="1"/>
```

Listing 14.71

This value is valid per the schema definition. However, when you read the value back from SQL Server it will give you the output as follows.

```
<Employee Active="true"/>
```

Listing 14.72

But this value is not valid per the schema definition. You cannot assign this value back to the same variable. This is a serious problem, and to avoid this SQL Server will restrict this type of usage.

Let us try to create a schema collection with the above schema definition and see what happens.

```sql
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Active">
        <xsd:simpleType>
          <xsd:restriction base="xsd:boolean">
            <xsd:pattern value="(0|1)"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

Listing 14.73

SQL Server will generate the following warning but the schema collection will be successfully created.

```
Warning: Type 'xs-
nun(/Employee/complexType()/@Active/simpleType())' is restricted
by a facet 'pattern' that may impede full round-tripping of
instances of this type
```

However, if you try to validate an XML instance against this schema collection, SQL server will generate an error. Look at the following example.

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Employee Active="1"/>'
```

Listing 14.74

```
XML Validation: The canonical form of the value '1' is not valid
according to the specified type. This can result from the use of
pattern facets on non-string types or range restrictions or
enumerations on floating-point types. Location:
/*:Employee[1]/@*:Active
```

This will cause a problem only if you specify a pattern that does not match with the canonical representation of the values of the given data type. The following is valid.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
    SELECT * FROM sys.xml_schema_collections
    WHERE name = 'ExampleSchema'
) BEGIN
    DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee">
    <xsd:complexType>
      <xsd:attribute name="Active">
        <xsd:simpleType>
          <xsd:restriction base="xsd:boolean">
            <xsd:pattern value="(true|false)"/>
          </xsd:restriction>
        </xsd:simpleType>
      </xsd:attribute>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>'
GO
```

```
DECLARE @x XML(ExampleSchema)
SELECT @x = '<Employee Active="true"/>'
```

Listing 14.75

This is valid because the pattern we specified accepts the canonical representation of boolean values. You will receive the following warning when you create the schema collection, but you can safely ignore it in this case.

```
Warning: Type 'xs-
nun(/Employee/complexType()/@Active/simpleType())' is restricted
by a facet 'pattern' that may impede full round-tripping of
instances of this type
```

This restriction is applicable to the following data types.

- boolean
- decimal
- float
- double
- dateTime
- date
- time
- hexBinary
- base64Binary

You cannot apply a pattern restriction on any of these data types if the pattern does not accept the canonical representation of the given data type.

Limitations on facet length

SQL Server stores the values of *length, minLength* and *maxLength* facets as a long data type. This is a 32 bit type and the maximum value these facets can take is 2^{31}.

The following schema is invalid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employees">
    <xsd:simpleType>
      <xsd:restriction base="xsd:string">
        <xsd:maxLength value="2147483648"/>
      </xsd:restriction>
```

```
        </xsd:simpleType>
    </xsd:element>
</xsd:schema>
```

Listing 14.76

If you try to create a schema collection with this schema definition, SQL Server will generate the following error.

```
The value of 'maxLength' facet is outside of the allowed range
```

SQL Server does not preserve the *"id"* attribute

When you create a schema collection, SQL Server will ensure the uniqueness of the *id* attribute. However, it does not preserve this value.

The following schema is invalid because two element declarations have the same *id.*

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Employee" id="Employee" />
    <xsd:element name="Manger" id="Employee"/>
</xsd:schema>
```

Listing 14.77

If you try to create a schema collection with the above schema definition, SQL Server will generate the following error.

```
Duplicate id value found: 'Employee'
```

However, SQL Server does not preserve this value. To verify this, let us create a schema collection.

```
-- DROP the previous SCHEMA COLLECTION
IF EXISTS(
      SELECT * FROM sys.xml_schema_collections
      WHERE name = 'ExampleSchema'
) BEGIN
      DROP XML SCHEMA COLLECTION ExampleSchema
END
GO

-- Create a schema collection
CREATE XML SCHEMA COLLECTION ExampleSchema AS '
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:element name="Employee" id="Employee" />
</xsd:schema>'
```

```
GO
```

Listing 14.78

Now let us retrieve the schema definition from SQL Server metadata.

```
SELECT XML_SCHEMA_NAMESPACE('dbo','ExampleSchema')
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee" type="xsd:anyType" />
</xsd:schema>
```

Listing 14.79

Note that the *id* attribute is not present anymore in the schema. SQL Server uses the *id* for the validation but does not store it. So after you create the schema, the *id* attribute is lost.

Further, the *id* attribute cannot take a value longer than 1,000 characters. If you try to create a schema collection with an identifier longer than 1,000 characters, SQL Server will generate the following error.

```
Identifiers may not contain more than 1000 characters
```

SQL Server does not support XSD data types: *ID,IDREF,IDREFS* and *NOTATION*

SQL Server does not support schema components having any of the above data types. The following is invalid.

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Employee" type="xsd:ID" />
</xsd:schema>
```

Listing 14.80

If you try to create a schema collection with any of these data types, SQL Server will generate the following error.

```
Invalid definition for type or element 'Employee'. SQL Server does
not permit the built-in XML Schema types 'ID' and 'IDREF' or types
derived from them to be used as the type of an element or as the
basis for derivation by extension.
```

SQL Server 2005 does not support *lists* of *union* types

SQL Server 2005 does not support lists of union types. SQL Server 2008 removed this limitation and supports lists of union types. Refer to Chapter 8 for a detailed discussion on this.

SQL Server 2005 does not support *union* of *list* types

SQL Server 2005 does not support union of list types. SQL Server 2008 removed this limitation and supports union of list types. Refer to Chapter 8 for a detailed discussion on this.

Limitation on restricting complex types with complex content

SQL Server does not allow restricting a mixed content type to create a simple content type. Refer to Chapter 11 for a detailed discussion on this.

Out-of memory-conditions

When working with large XML Schema collections, out-of-memory condition might occur. Books Online provides the following solution to handle such conditions.

- When the system load is light, use the DROP XML SCHEMA COLLECTION command. If this fails, put the database in single-user mode by using the ALTER DATABASE statement and trying DROP XML SCHEMA COLLECTION again. If the XML schema collection exists in **master**, **model**, or **tempdb**, a server restart is required for single-user mode.
- When you call the XML_SCHEMA_NAMESPACE, you can try to retrieve a single XML schema namespace, you can try the call when the system load is lighter, or you can try the call in single-user mode.

Limitation on *final* and *block* attributes

We saw that the *final* and *block* attributes can take a space separated list of values. For a detailed discussion on those attributes, refer to Chapter 11.

Although these attributes can accept a space separated list of values, you cannot specify the same value more than once. The following is illegal.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:simpleType name="PhoneType" final="extension extension">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{3}-[0-9]{3}-[0-9]{4}"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:schema>
```

Listing 14.81

If you try to create a schema collection with the above schema definition, SQL Server will generate the following error.

```
The attribute 'final' cannot have a value of 'extension extension'
```

Limitation on *union* types

SQL Server does not support restriction on union types. The following is illegal.

```xml
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="EmployeeNumber">
    <xsd:simpleType>
      <xsd:restriction base="EmployeeNumberType">
        <xsd:length value="2"/>
      </xsd:restriction>
    </xsd:simpleType>
  </xsd:element>
  <!-- EmployeeNumberType (union)-->
  <xsd:simpleType name="EmployeeNumberType">
    <xsd:union memberTypes ="xsd:string xsd:integer" />
  </xsd:simpleType>
</xsd:schema>
```

Listing 14.82

If you try to create a schema collection with the above schema definition, SQL Server will generate the following error.

```
This type may not have a 'length' facet
```

Limitation on *decimal* types

SQL Server does not support variable precision decimals. The maximum value *totalDigits* facet can accept is 38 and the maximum value *fractionDigits* can accept is 10. If you try to create a schema collection with values outside this range, SQL Server will generate the following error.

```
Component 'xs-nun(/value/simpleType())' is outside of allowed
range. Maximum for 'fractionDigits' is 10 and maximum number of
digits for non fractional part is 28
```

Chapter Summary

SQL Server stores schema definitions as *xml schema collections.* An *xml schema collection* is a system object like tables, views, procedures, etc. Each schema collection can contain the definition of one or more schema definitions.

When a column or variable is bound to a schema collection that has more than one schema definition, SQL Server validates xml instances being assigned to the column/variable against all the global elements declared in the entire schema collection. The operation will succeed only if the xml instance validates successfully against any one of the global elements declared in the schema collection.

You can add the definition of one more schemas to an existing schema collection by using ALTER XML SCHEMA COLLECTION ADD command. This adds new schema definition to the existing schema collection. SQL Server will group the definition of schema components by target namespace. Further, it will strip off comments and might alter the schema definitions slightly without losing the original meaning.

You can retrieve the definition of a schema collection by using the system function: *xml_schema_namespace().* When you retrieve the schema definitions, it can happen that you get a slightly modified version of your original schema without losing the original meaning. Further, you will lose the comments you may have embedded to your original schema definition. This will make editing schema definitions more difficult. To overcome this, it is a good idea to keep the original definition of the schema collection in a file or a table. If you need to modify the schema collection, you can take the current definition from the file or table where it was previously stored. This will be harder to maintain, but easier in terms of modifying the schema definition.

Though you can add a new schema definition to an existing schema collection using ALTER SCHEMA COLLECTION ADD command, you cannot change the definition of a schema component. For example, if you want to change the data type of a simple type, you need to drop the schema collection and recreate it. The tough part is that SQL Server will not allow you to drop a schema collection if it is bound to a column or used as arguments to a stored procedure or function. Before you modify the schema collection you need to remove all references from the definition of columns, stored procedures, functions, etc. This makes modification of schema collections very difficult. Luckily, SQL Server has a number of system catalog views that give detailed information of the XML schema collection metadata.

The XSD implementation of SQL Server has a number of limitations. SQL Server 2008 has removed some of the issues that SQL Server 2005 had, and the last section of this chapter discussed those limitations in detail.