XSD Schemas

XSD is the commonly used name for the proposed XML Schema Definition standard of the World Wide Web Consortium (W3C). In the previous chapter, we saw how to use XDR schemas to return data as XML. Microsoft created XDR so that it could start using XML schemas as soon as possible, even though the W3C's work on XSD was not near completion. As such, XDR is based on a snapshot of the W3C's schema activity from March 1999, and is purely a Microsoft product. The W3C has made progress on XSD since then, and it now provides a much richer set of functionality than XDR. Consequently, although the XSD schema syntax is not presently supported in SQL Server, it is important enough for us to cover.

This chapter will examine the following:

- What XSD schemas are
- The present status of XSD with the W3C
- How XSD schemas compare to XDR schemas
- How to design and use XSD schemas
- How to validate your XSD schema
- SQL Server and XSD
- Some translation tools available to convert from XDR to XSD

What are XSD Schemas?

New standards for defining XML documents have become desirable because of the limitations imposed by Document Type Definitions (DTDs). The W3C XML Schema Definition (XSD) standards were promoted from the Candidate Recommendation Phase to the Proposed Recommendation Phase in March 2001. Once the W3C's director approves the standards, they will become a full Recommendation. Until the XSD standards reach the full Recommendation phase, however, they are subject to further review and changes. Thus, the information contained in this chapter is subject to change. However, according to members of the Schema group, this last step is almost purely administrative and the proposed recommendation is good enough for software development.

> *Please consult the W3C's web site at http://www.w3.org for current documentation and up-to-date information about the status of XSD. For a good overview of XSD, please see the XSD primer document on the W3C's web site at http://www.w3.org/TR/xmlschema-0.*

XSD Schemas vs. XDR Schemas

In the previous chapter, we learned that schemas provide an XML-based syntax for defining what elements and attributes are allowed in a given document. We saw examples of how to create schemas using the XDR schema syntax. So how does XDR compare to XSD? As we said earlier, XDR is Microsoft's own version of the W3C's early 1999 work-in-progress version of XSD. XSD provides a richer set of functionality than XDR and is vendor neutral.

Let's take a quick look at an example of how XSD compares to XDR. Don't worry yet about understanding the details of the XSD syntax. Consider the following XML document from the XDR chapter:

```
<?xml version="1.0"?>
   <Students>
      <Student>
         <ID>12345</ID>
         <GPA>3.5</GPA>
      </Student>
      <Student>
          <ID>67890</ID>
         <GPA>4.0</GPA>
      </Student>
   </Students>
```

Suppose that the XML shown above represents the proper syntax for the document. A valid XDR schema for this document could look like this:

```
<?xml version="1.0"?>
   <Schema name="StudentsSchema"
   xmlns="urn:schemas-microsoft-com:xml-data">
      <ElementType name="ID" content="textOnly"/>
      <ElementType name="GPA" content="textOnly"/>
      <ElementType name="Student" content="eltOnly">
         <element type="ID" minOccurs="1" maxOccurs="1"/>
         <element type="GPA" minOccurs="1" maxOccurs="1"/>
      </ElementType>
      <ElementType name="Students" content="eltOnly" model="closed">
         <element type="Student" minOccurs="0" maxOccurs="*"/>
      </ElementType>
   </Schema>
```

To define the same schema following the XSD syntax, on the other hand, the schema definition might look like:

```
<?xml version="1.0"?>
 <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
    <xsd:complexType>
     <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
 </xsd:schema>
```

As you can see from a quick glimpse at these two examples, there are distinct differences in the syntax between XDR and XSD. The XSD syntax will be explained in greater detail later in this chapter.

Key Features

XSD schemas offer the capabilities of XDR schemas, DTDs, and much more. Here is a brief summary of the key features of an XSD schema document:

- Like XDR schemas, XSD schemas *are* XML documents, so you don't have to learn another syntax to use them (unlike with DTDs).
- As with XDR schemas, data types in XSD schemas can be specified for an element or attribute.
- XSD schemas allow you to define your own data types, or use one of the 44+ pre-defined data types.
- XSD schemas offer the ability to define keys on data elements for uniqueness.
- XSD schemas support object-oriented style inheritance where one schema can inherit from another. This is a huge benefit and allows you to create re-usable schemas.
- XSD schemas allow you to define elements that can be substituted for each other.
- XSD schemas allow you to define elements with Null content.

Like XDR, XSD schemas define the format, content, and data of an XML document. When a document that references an XSD schema is validated by a parser that supports XSD, it confirms whether or not the document meets the criteria defined in the schema. If the validation fails, an error occurs.

Designing XSD Schemas

Let's walk through the details of the student XSD schema presented earlier to see how it works.

XML Declaration

An XSD schema is an XML document, so the first line can be a typical XML declaration:

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

In this example, `version` is the only attribute expressed in the declaration. Other attributes, such as `encoding`, can be specified to determine which encoding is used to represent characters.

<schema>

The next line in our XSD schema is the root element itself. For XSD schemas, the root element is always `<schema>`. To identify the root element to an XML parser as an XSD schema, a specific namespace is referenced in the `<schema>` element. Here is an example:

```
<?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  </xsd:schema>
```

The namespace in the declaration is referencing the W3C's Proposed Recommendation version of XSD from 2001. In older examples, you may see the Candidate Recommendation version of XSD referenced, as shown below:

```
<?xml version="1.0"?>
  <xsd:schema xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
  </xsd:schema>
```

Note the `xsd:` prefix used in the above examples. This prefix is used to designate that an XSD schema is being used, although any prefix can actually be specified. The prefix must match the one specified prior to the namespace declaration, as shown below:

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

The purpose of using the prefixes throughout the schema is to show that the contents of the schema belong to the vocabulary of the designated namespace (XSD), versus that of another schema author.

It is worth noting that, if you make this namespace the default, then you don't have to use the prefixes at all. However, you should be aware that using default namespaces can lead to problems later, especially when you import schemas. Even the W3C's examples use the `xsd:` prefix throughout the code rather than setting the default namespace.

<element>

After identifying the schema, we then move on to the heart of the schema – the elements. We learned a lot about elements in the XDR chapter. So let's dive right into the syntax for creating elements with XSD.

Assigning values to the following attributes can modify the behavior of the element:

- `name`

  Required. Refers to the name of the element.

- `type`

  Refers to a simple type (for example `xsd:string`) or the name of a complex type. The `type` attribute can be used in the declaration of a simple type (when it is not being restricted) but not with a complex type, as will be demonstrated in this section.

- `minOccurs`

  This attribute determines if the element is optional. This attribute is not required. If unspecified, the default is `1`. The table below shows the possible values that may be assigned:

Value	Description
`0`	The element is optional.
`Integer > 0`	The element must occur at least the specified number of times.

- `maxOccurs`

  This attribute determines how many elements are allowed. This attribute is not required. If unspecified, the default is `1`. The table below shows the possible values that may be assigned:

Value	Description
`Integer > 0`	The element can only appear up to the specified number of times.
`Unbounded`	The element may appear an unlimited amount of times.

With this understanding of `<element>`, let's take a closer look at our schema example:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Looking at the schema, we notice the following definitions and rules:

- The `<ID>` and `<GPA>` contain string values.
- The `<Student>` element must contain only one `<ID>` and `<GPA>` element (the `<ID>` and `<GPA>` have a `minOccurs` and `maxOccurs` of `1`). Note that the `minOccurs` and `maxOccurs` attributes were not necessary in the above example because `1` is the default value if they are not specified.
- There can be unlimited `<Student>` elements or none at all (`minOccurs` is `0` and `maxOccurs` is unbounded).
- By specifying `<sequence>`, the `<ID> and <GPA>` elements are required to appear in the order listed in the schema.

In other words, we have defined that documents adhering to this XSD schema may list as many students as needed, as long as an ID and GPA are provided for each student. This "contract" can now allow multiple parties to share student information in an agreed format.

Now let's look at the detailed syntax of these element declarations.

<complexType> vs. <simpleType>

In the previous chapter, we saw how to use the `<ElementType>` element in XDR to describe characteristics – such as whether the element can contain child elements – by simply changing the values of the element attributes (for example `textOnly`, `eltOnly`, etc.). XSD takes a different approach. With XSD, you explicitly declare an element to be either a **complex type** or a **simple type**. When do you use each one?

The `<complexType>` element should be used:

- When your element will contain child elements, *and/or*
- When your element will contain attributes.

The `<simpleType>` element should be used:

- When you want to create a new data type from a built-in simple type, *and/or*
- When your element will *not* contain child elements or attributes.

To illustrate the differences, let's take a look at some examples. In the student example we have been using, each student has an ID and a GPA. Thus, if student is an element, then ID and GPA would be considered children of that element. Because the student element has child elements, we have to declare it as a complex type, as shown below:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
</xsd:schema>
```

Now let's modify our example to demonstrate the simple type element concept. Remember that we cannot use a simple type if there are children or attributes of the element. Suppose that we want to create a new data type called `StudentGPA` that would define the proper format for the GPA. In this scenario, we want to extend an existing simple data type (for example `xsd:string`) and customize it for our own purposes. These customizations are referred to as restrictions on the existing data type. We want to restrict the string so that the GPA has to be in a format like 4.00, for example. That is, we want there to be an integer followed by a decimal point and then two more integers. The code to declare this looks like:

```
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Following the simple type element declaration are the restrictions. Notice first how the `base` data type that we are starting with is the `string` data type. Next, the restrictions that are being placed on the `string` data type are specified. In this case, we are requiring the `length` of the `StudentGPA` to be `4` characters and the `pattern` to be one digit, then a decimal, and then two more digits. It is important to know that this `pattern` is written in the Unicode **Regular Expression** language, which is similar in its syntax to the Perl programming language.

Let's digress for a moment to take a look at the regular expression language in more detail and then we'll come back to the `pattern` of our specific example, and see if it makes more sense. The table below summarizes some of the most common uses of regular expressions:

Regular Expression	Explanation	Valid Example(s)
`\d`	Digit	1, 2, 3, etc.
`[a-z]`	Lower case ASCII characters	a, b, c, etc.
`[A-Z]`	Upper case ASCII characters	A, B, C, etc.
`*`	Wildcard	A*Z = ABZ, ABCZ, ABCCZ, etc.
`?`	Single placeholder	A?Z = ABZ, ACZ, ADZ, etc.
`+`	Inclusive of at least the specified values, but more are allowed	A+Z = AZ, ABZ, ABCZ, etc.
`(value1|value2)`	`OR`	(A|Z)+Q = AQ, ZQ, ABQ, ZBQ, etc.
`[abcde]`	Another way to specify `OR`, but with single characters only	[abc] = a, b, or c
`[^0-9]`	Any non-digit character	A, B, C, a, b, c, etc.
`{integer}`	The number of occurrences that there must be of the previous value	az{2} = azz \d{3} = 123, 456, 789, 444, etc. (az){2} = azaz

Now, back to our example from before:

```
      <xsd:pattern value="\d{1}.\d{2}"/>
```

The `\d{1}` is specifying that there must be one and only one digit prior to the decimal point. Then, after the decimal point, there must be two more digits, as specified by the `\d{2}` syntax.

This is just one of the many possible ways that regular expressions can be used to specify `patterns`. A detailed explanation of regular expressions is beyond the scope of this chapter. For more information about regular expressions, please consult Appendix D of the W3C's primer document at http://www.w3.org/TR/xmlschema-0.

Let's get back on track with our working example of declaring a custom data type called `StudentGPA`, which is shown again here to refresh your memory:

```
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

When declaring the complex type `<Student>` element with `<ID>` and `<GPA>` as children, we can make use of our new `StudentGPA` simple type by defining the GPA to be of this type instead of just a simple string type. Here's an example:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="StudentGPA"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

We could put the `simpleType` declaration format for the `<GPA>` element inline with the `<GPA>` element itself, to accomplish basically the same result. Here's how:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1"
                  maxOccurs="1"/>
       <xsd:simpleType>
         <xsd:restriction base="xsd:string">
          <xsd:length value="4"/>
          <xsd:pattern value="\d{1}.\d{2}"/>
         </xsd:restriction>
          </xsd:simpleType>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

In this second version, the simple type declaration and the restrictions immediately follow the <GPA> element declaration line.

> **The difference between these two methods is that the first allows you to access the `StudentGPA` data type in multiple places in the schema. With the second example, the restrictions are specific to the `<GPA>` element and cannot be re-used by name outside that element.**

Now that we have seen some examples of the different ways to declare elements as simple and complex types, let's summarize what we have learned. There are actually three ways to declare elements. Under the first method, you list the name, type, minOccurs, and maxOccurs attributes. This method is actually an implied way of defining simple type elements without specifying the <simpleType> element syntax explicitly. You don't have to explicitly state that it is a simple type because you are using one of the built-in simple types already:

```
<xsd:element name="name" type="type" minOccurs="int" maxOccurs="int"/>
```

Remember that the values of the attributes are merely placeholders.

This syntax should look familiar to you. We declared the <ID> and <GPA> elements this way in one of our previous examples:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

With the second method for declaring an element, the name, minOccurs, and maxOccurs attributes are specified for the parent, which is then declared as a complexType so that it can have child elements:

```
<xsd:element name="name" minOccurs="int" maxOccurs="int"/>
  <xsd:complexType>
  </xsd:complexType>
</xsd:element>
```

The <Student> element in the same example was declared with name, minOccurs, and maxOccurs attributes, and as a complexType with <ID> and <GPA> as children:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

With the third method, the `name`, `minOccurs`, and `maxOccurs` attributes are specified, and then a `simpleType` is declared to describe the element restrictions:

```
<xsd:element name="name" minOccurs="int" maxOccurs="int"/>
  <xsd:simpleType>
    <xsd:restriction base = "type">
    </xsd:restriction>
  </xsd:simpleType>
</xsd:element>
```

An example of this was shown in our custom-defined GPA example:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="StudentGPA"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

The ability to create your own data types, as in the example above, is a very powerful feature of XSD. We will walk through more examples of how to create your own data types in the *Data Types* section of this chapter.

Have you noticed how the student example we have been working with is element-centric? Suppose that we decided that `<ID>` should be an **attribute** of `<Student>`, instead of a child element. What changes need to be made to our schema? Let's take a look at other parts in the XSD schema that define attributes in an XML document.

<attribute>

Defining attributes in an XSD schema is very similar to defining elements. Here is the list of attributes that are associated with the `<attribute>` element:

- `name`
  Required. This is the name of the attribute.
- `type`
  Any simple type such as `xsd:string`, `xsd:integer`, `StudentGPA` (the custom simple type we created above), etc. can be specified. This identifies the data type of the attribute.

- `use`

Value	Description
`Required`	The attribute must appear in the element.
`Default`	The attribute will use the default value if none is specified. (See the `value` attribute).
`Fixed`	The attribute contains a fixed value that will never change. (See the `value` attribute).
`Optional`	The attribute is optional.
`Prohibited`	The attribute is prohibited.

- `value`

  Specifies the value of the attribute. This is only used when the `use` attribute is `Default` or `Fixed`.

We learned earlier that complex types can contain children or attributes. So we already know that an attribute has to be declared within a complex type. However, attributes themselves can only have simple types. They cannot contain child elements.

There are two ways to define an attribute. The first method is done on one line and is used in scenarios when the attribute is based on an existing simple type (either a built-in simple type or a simple type defined elsewhere in the document). The syntax for this method is shown below (again, using placeholders):

```
<xsd:attribute name="name" type="simple type" use="how used" value="value"/>
```

The second method for defining an attribute allows you to specify explicit restrictions on the attribute, such as the format that it must be in to be valid:

```
<xsd:attribute name="name" use="how used" value="value">
  <xsd:simpleType>
    <xsd:restriction base="simple type">
      <xsd:facet value="value"/>
    </xsd:restriction>
  </xsd:simpleType>
</xsd:attribute>
```

Facets will be discussed in greater detail in the *Data Types* section. In place of `facet` above, you can specify any particular facet available for a given simple type (for example `length`, `pattern`, `enumeration`, etc. for `xsd:string`). For now, let's move on to seeing how each of these methods can be used, by modifying our ongoing example.

Suppose we want the `<ID>` element to be an attribute of the `<Student>` element. The schema would then look something like this:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
```

```
        <xsd:attribute name="ID" type="xsd:string" use="required"/>
     </xsd:complexType>
    </xsd:element>
   </xsd:schema>
```

There are a couple of important aspects to take note of here. First, notice how the attribute is defined within the `<complexType>` element. We recall from before that attributes can only exist within a complex type. In this instance, we have made `ID` an attribute of the `<Student>` element, instead of an element itself. Secondly, note how the `ID` declaration now follows the `GPA` declaration instead of coming before it. Attributes need to be declared as the last items in the complex type, following all the elements.

These revisions now indicate the following requirements:

- The `<Student>` element must contain an `ID` attribute.
- The `ID` attribute cannot be used in any other elements, as it is declared within the `<Student>` element.

If we later introduced other elements to the schema that could use an `ID` attribute, we could simply declare the attribute outside of any element to make it accessible to the schema as a whole.

What if we decide that we want to require the `ID` to be a fixed length string of 5 characters that can only contain numeric values? In this instance, we want to place restrictions on the values and format the `ID` will contain. The following example shows how we can do this:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:attribute name="ID" use="required">
       <xsd:simpleType>
         <xsd:restriction base="xsd:string">
           <xsd:length value="5"/>
           <xsd:pattern value="\d{5}"/>
         </xsd:restriction>
       </xsd:simpleType>
     </xsd:attribute>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Notice how, in this instance, we have declared the attribute and then declared a simple type with the restrictions that we want to implement. We specify in the `length` facet that the `ID` must be `5` characters long and in the `pattern` facet that it must contain five digits.

Now that we have a basic understanding of how to declare elements and attributes, let's take a look at how to reference an XSD schema from an XML document. We learned how to do this by referencing an XDR schema in the previous chapter. Now let's take a look at how to do this with XSD. Here is an XML document example with the XSD schema assumed to reside in the main directory on the `www.mycollege.org` namespace:

```
<?xml version="1.0"?>
   <Students xmlns = "http://www.mycollege.org"
             xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="http://www.mycollege.org/StudentSchema1.xsd">
      <Student>
         <ID>12345</ID>
         <GPA>3.5</GPA>
      </Student>
      <Student>
         <ID>67890</ID>
         <GPA>4.0</GPA>
      </Student>
   </Students>
```

Save the schema as a file named `StudentSchema1.xsd`. Notice how a separate namespace, `http://www.w3.org/2001/XMLSchema-instance`, is used with the `xsi:` prefix. The W3C actually created this separate namespace to allow you to tie a document to its schema. If the XML document above is validated by an XML parser that supports XSD, it will succeed.

If you aren't using a namespace, then there is another way to reference the schema in your XML document. An example is shown below:

```
<?xml version="1.0"?>
   <Students xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
             xsi:noNameSpaceSchemaLocation="StudentSchema1.xsd">
      <Student>
         <ID>12345</ID>
         <GPA>3.5</GPA>
      </Student>
      <Student>
         <ID>67890</ID>
         <GPA>4.0</GPA>
      </Student>
   </Students>
```

Thus, if the author of a document makes use of namespaces to indicate the intended interpretation of names in the document, the `xsi:schemaLocation` attribute should be used to specify the location of the XSD schema that can be used to validate the document (which, in this case, is assumed to reside in the same directory as the XML document itself).

If the author does not need or want a namespace, the `xsi:noNameSpaceSchemaLocation` attribute can be used to locate the XSD schema used to validate the document.

So far, we have illustrated the basics of defining simple XSD schemas using elements and attributes and have seen how to reference those schemas from XML documents. The next section will look at some alternative ways to structure the syntax, so we can accomplish better results.

Structure Alternatives

There are a few different ways to structure an XSD schema. One way is to define elements and their attributes within the complex type declaration itself. Another way to define elements is to declare them as immediate children of `<schema>` (that is, outside the complex type itself) and then just make reference to the elements within the complex type. By declaring the elements nested within the complex type declarations, their scope is local to that complex type – they are only available to that complex type and cannot be referenced by other elements in that schema, or in any other schema for that matter. By declaring the elements outside the complex type, on the other hand, their scope is global. This has the effect of allowing those elements to be utilized from anywhere within the schema or from other schemas.

So, in situations where you do not need the elements to be referenced from within the same schema or from other schemas, it is perfectly fine to declare them nested within other elements. However, in situations where you want to be able to *re-use* a specific element, you should define that element as a child of `<schema>`.

Let's walk through some examples to further clarify these concepts. Recall our example from earlier:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

First, notice how the `<Student>` element is declared as a child of the root element `<schema>`. This means that the `<Student>` element is global in scope and can be referenced from this or other schemas.

The `<ID>` and `<GPA>` elements, on the other hand, are declared within the `Student` complex type itself:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

This means that the `<ID>` and `<GPA>` are local in scope to the `<Student>` element and cannot be re-used anywhere else.

But what if we really need to re-use `ID` and `GPA`? For instance, we might also have to track professor credentials, which could also consist of an `ID` and `GPA` of the same data types. Let's take a look at how we could declare the `ID` and `GPA` globally so that we can reference them in other places, including in a `<Professor>` element.

First, we will start by modifying our previous example to define `ID` and `GPA` globally, and then have the `<Student>` element reference the global declarations:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ID" minOccurs="1" maxOccurs="1"/>
      <xsd:element ref="GPA" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:element name="ID" type="xsd:string"/>
<xsd:element name="GPA" type="xsd:string"/>
</xsd:schema>
```

In the above example, the `ref` attribute refers to the `<ID>` and `<GPA>` elements that are declared outside the `Student complexType` declaration as children of the root `<schema>` element. This takes two extra lines of code to accomplish, but the benefit of code re-use far outweighs the extra lines required.

So, if we also want to implement the `<Professor>` element using these global elements, it might look something like this:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ID" minOccurs="1" maxOccurs="1"/>
      <xsd:element ref="GPA" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:element name="Professor" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="ID" minOccurs="1" maxOccurs="1"/>
      <xsd:element ref="GPA" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
<xsd:element name="ID" type="xsd:string"/>
<xsd:element name="GPA" type="xsd:string"/>
</xsd:schema>
```

Now, in the event that we want to modify the `<ID>` or `<GPA>` elements, we only have to change them in *one* place and both the `<Student>` and `<Professor>` elements will automatically reference the updates. Imagine the extra work required if we had duplicated the declarations in both places and then needed to modify them. Or, worse yet, what if we had duplicated them everywhere in multiple schemas? This could turn into a maintenance nightmare.

The fact that XSD provides you with a mechanism for re-using code in an efficient way is an incredible advantage. It is also a critical concept to master, so let's look at a few more examples to further fix it in your mind.

Recall when we looked at the two different syntaxes for declaring the `StudentGPA` restrictions. In one example, the restrictions on the GPA were nested within the element declaration itself:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1"
                  maxOccurs="1"/>
       <xsd:simpleType>
         <xsd:restriction base="xsd:string">
          <xsd:length value="4"/>
```

```
            <xsd:pattern value="\d{1}.\d{2}"/>
           </xsd:restriction>
           </xsd:simpleType>
      </xsd:sequence>
    </xsd:complexType>
   </xsd:element>
  </xsd:schema>
```

In this instance, the restrictions on the GPA are declared within the GPA element itself and are local in scope. Thus, they cannot be re-used. However, recall how the `StudentGPA` was declared separately in the other example, and then referenced as the `type` attribute for the `<GPA>` element:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="StudentGPA"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

In this example, the `<StudentGPA>` element is global and can be re-used from within this schema or from other schemas. So, if we have another schema that needs to make use of the `<StudentGPA>` element, we can import the above schema and reference it, just as if it were declared within the same schema. Pay close attention as this is really useful.

Referencing External Schemas

The first step in making use of another schema is to **import** or **include** that schema into the one you're working with. You include schemas that are in the *same* namespace as the one you are working in:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="URIgoeshere">
<xsd:include schemaLocation="XSDFileNameGoesHere"/>
...
</xsd:schema>
```

On the other hand, you import schemas that are in a *different* namespace from the one you are working in:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="URIgoeshere">
<xsd:import namespace="URIgoeshere"
     schemaLocation="XSDFileNameGoesHere"/>
...
</xsd:schema>
```

So, let's suppose that we are creating a new schema for professors and want to make use of the global <StudentGPA> element (forgive the inappropriate name) from a (hypothetical) StudentGPA1.xsd file in our new <Professor> element. Further, suppose that we want to extend the <Professor> element to include an additional element for the university that professor graduated from. The code to accomplish this looks like:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.xsdrocks.com/students">
<xsd:include schemaLocation="StudentGPA1.xsd"/>
<xsd:element name="Professor" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element ref="StudentGPA" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="University" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element
</xsd:schema>
```

Note that the include statement references the StudentGPA1.xsd document that must be present in the same location as the target namespace. Once included in this schema, the <Professor> element references the <StudentGPA> global element in exactly the same manner as if it had been declared within this document itself. Think of it as though you were typing them all into the same schema to begin with, since that is actually the net effect of an include statement. Then, the additional <University> element, which is unique to the <Professor> element, is defined after the reference to the <StudentGPA> element.

What if the StudentGPA1.xsd document is not saved in the same location as the target namespace? In that case, you use the import statement instead of the include statement:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://www.xsdrocks.com/professors">
<xsd:import namespace="http://www.xsdrocks.com/students/"
     schemaLocation="StudentGPA1.xsd"/>
<xsd:element name="Professor" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ProfessorGPA" type="StudentGPA1:StudentGPA"
        minOccurs="1" maxOccurs="1"/>
     <xsd:element name="University" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element
</xsd:schema>
```

Notice that the import statement above specifies the namespace to import from, as well as the name of the schema to import. Further note that the <ProfessorGPA> element is defined as StudentGPA1:StudentGPA, with StudentGPA1: being the prefix as defined in the schema location.

If you would like more information about XSD structures, please see the XSD structures document on the W3C's web site at http://www.w3.org/TR/xmlschema-1.

Now that we have learned the basics of creating elements and attributes, and the varying ways to structure the code, we will take a look at how to document the schema so that it is easier to understand.

Annotations

Documenting your code is as important as writing the core functionality itself. If you do not describe what is happening so that you and others will be able to understand it later, then what good is it? XSD provides you with a way to document your schemas in a very clean and comprehensible way – **annotations**.

The `<annotation>` element can be used to document your schemas. You should use its `<documentation>` child element to provide comments to people, and use the `<appinfo>` child element to provide comments to applications. Here's an example of how an annotation might look:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
      <xsd:annotation>
       <xsd:documentation>The Student ID uniquely identifies a student
       </xsd:documentation>
       <xsd:appInfo>Student Identification Number
       </xsd:appInfo>
      </xsd:annotation>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Notice how the `<annotation>` for the `ID` immediately follows the element declaration line itself. The `<documentation>` and `<appInfo>` elements must be children of the `<annotation>` element. You can use either of them individually or both of them together.

> **It is important to note that you cannot put annotations anywhere you want in the schema. You can only put annotations at the beginning of the content model that you are annotating (for example immediately after a schema declaration or after an element declaration).**

The purpose of the `<documentation>` element is to describe what the code is doing for people looking at the source code. On the other hand, the purpose of the `<appInfo>` element is to display helpful information to the end user in the application. The `<appInfo>` element data is typically transformed and displayed to the user using a stylesheet, whereas the `<documentation>` element is simply left as-is.

Now that you have a good handle on creating and annotating schemas, let's take a look at the data types of XSD.

Data Types

Data type support is an extremely valuable feature of XSD. Data types allow for a deeper level of validation and proper format of elements and attributes. XSD schemas support a wide range of data types in addition to allowing you to define your own data types.

Simple Data Types

There are presently 44 simple data types in XSD that you can take advantage of. Some of these types are built into XSD, while others are derived from these built-in types. Both simple types and their derivations can be used in element and attribute declarations. We have already seen examples of this when we looked at how to declare elements and attributes. In doing so, we learned that you can specify the data type that you want the element or attribute to be by assigning a value to the `type` attribute, as shown below:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:string" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Notice how `xsd:string` is specified for the `type` attribute of both the `ID` and `GPA`. This means that both of these elements will accept string values. We discussed at the beginning of this chapter the reason for including the `xsd:` prefix throughout your schema – so that the parser knows that the syntax you are using belongs to the XSD language.

In building your schemas, you can assign the `type` attribute to any of the valid simple data types shown in the table below. Or, as we will discuss in more detail shortly, you can create your own data types to extend these simple types.

XSD Data Type	Description
`anyUri`	Uniform Resource Identifier (URI). Examples: http://www.sample.com, http://www.sample.com/index.html#ID2
`base64Binary`	MIME-style Base64 encoded binary data.
`hexBinary`	Hexadecimal-encoded binary data.
`boolean`	True (1) or False (0).
`byte`	-128 to 127.
`dateTime`	Date in a subset of the ISO 8601 format. Time is optional. Time Zone is optional. Example: 2001-04-06T11:45:33.000-05:00

Table continued on following page

XSD Data Type	Description
`date`	Date in a subset of the ISO 8601 format. Example: 2001-04-06
`decimal`	Positive or negative arbitrary precision decimal value. Note that, in the Candidate Recommendation version of XSD, this was called `Number` instead. Examples: -5.34, 0, 5.34, 5000.00
`double`	Equivalent to double-precision 64-bit floating point.
`duration`	Duration of time specified in years, months, days, hours, minutes, and seconds format, as defined in the ISO 8601 standards extended format P*n*Y*n* M*n*DT*n*H *n*M*n*S. nY is the number of years, nM is the number of months, and so on. The P is required but the other items are optional. For example, to specify a duration of 1 year and 2 months, you would specify: P1Y2M. To specify a duration of 1 year, 2 months, 3 days, 10 hrs, 30 minutes, and 12.3 seconds, you would specify: P1Y2M3DT10H30M12.3S
`ENTITIES`	XML 1.0 `ENTITIES` attribute type. `ENTITIES` contain a set of `ENTITY` values. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`ENTITY`	XML 1.0 `ENTITY` attribute type. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`float`	Equivalent to single-precision 32-bit floating point.
`gDay`	Day in Gregorian format. Example: `--31` (every 31st day, regardless of month)
`gMonth`	Month in Gregorian format. Example: `--06--` (every May)
`gMonthDay`	Month and day in Gregorian format. Example: `--07-31` (every July 31st)
`gYear`	Year in Gregorian format. Example: 2000
`gYearMonth`	Year and month in Gregorian format. Example: 2000-02
`ID`	The `ID` values must be unique throughout all elements in the XML document. This attribute is referenced by other attributes such as `idref` and `idrefs`. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`IDREF`	References the value in an `ID` attribute within the XML document. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`IDREFS`	References multiple `ID` type values separated by whitespace. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.

XSD Data Type	Description
`Int`	Integer. Sign is optional. Range: -2147483648 to 2147483647.
`Integer`	The standard mathematical concept of integer numbers. Range: an infinite set of negative or positive numbers.
`Language`	Any valid XML `Language` value as defined by RFC 1766. Example: en-US
`Long`	Integer. Range: -9223372036854775808 to 9223372036854775807.
`Name`	XML 1.0 `Name` type.
`NCName`	XML Namespace `NCName` (an XML `Name` without the prefix and colon).
`negativeInteger`	Range: negative infinity to –1.
`NMTOKEN`	Name token value. String consisting of one word in a set of letters, digits, and other characters in any combination. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`NMTOKENS`	List of name tokens separated by whitespace. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`nonNegativeInteger`	Range: 0 to infinity.
`nonPositiveInteger`	Range: Negative infinity to 0.
`normalizedString`	String of character data. Newline, tab, and carriage-return characters are converted to spaces before schema processing.
`NOTATION`	XML 1.0 `NOTATION` attribute type. To retain compatibility between XSD and XML DTD 1.0s, these should only be used with attributes.
`positiveInteger`	Range: 1 to infinity.
`QName`	XML Namespace `QName`.
`Short`	Range: -32768 to 32767.
`String`	String of character data (characters that match `Char` from XML 1.0).
`Time`	Time in hh:mm:ss.sss-TimeZone format. Time Zone is optional. The time zone is based on the number of hours ahead or behind Coordinated Universal Time (as defined in ISO 8601). Example: 11:45:33.20-05:00 where 05:00 means 5 hours behind Universal time

Table continued on following page

XSD Data Type	Description
Token	String of character data. Like normalizedString, the newline, tab, and carriage-return characters are converted to spaces before schema processing. In addition, adjacent space characters are collapsed to a single space and leading and trailing spaces are removed.
unsignedByte	Unsigned byte. Range: 0 to 255.
unsignedInt	Unsigned integer. Range: 0 to 4294967295.
unsignedLong	Unsigned long. Range: 0 to 18446744073709551615.
unsignedShort	Unsigned short. Range: 0 to 65535.

Now let's see an example of these data types in action. To invalidate the string "XSD is great!!" as content for the <GPA> element, our schema can define a floating-point number as the only valid data type. Here is how it looks:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="xsd:float" minOccurs="1" maxOccurs="1"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
</xsd:schema>
```

Now, if an XML document references this schema and it contains this element:

```
<GPA>XSD is great!!</GPA>
```

An XSD Parser will generate an error indicating a problem with the data type.

You can make use of the other data types in this same manner. As you can see, it is very easy to take advantage of these data types. Since this is a relatively straightforward concept, let's move on to the more complicated details of creating your own data types.

Creating Your Own Data Types

You can create a simple data type by deriving from any one of the 44 listed on the previous table. We already saw an example of this concept with our StudentGPA data type that restricts the GPA to a certain format:

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
 <xsd:element name="Student" minOccurs="0" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
     <xsd:element name="ID" type="xsd:string" minOccurs="1" maxOccurs="1"/>
     <xsd:element name="GPA" type="StudentGPA"/>
    </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
</xsd:schema>
```

Facets

In the example above, we created a new data type called `StudentGPA` as an extension of the existing `xsd:string` simple type. We then specified the **facets** (`length` and `pattern`) that we wanted to implement to make the data type meet our objective.

In XSD, there are a number of different facets that can be specified to further restrict or define your new data type. The available facets vary depending on the data type you are deriving your new type from. For example, the `string` data type has the following facets:

- `enumeration`
- `length`
- `minLength`
- `maxLength`
- `pattern`
- `whitespace`

Our `StudentGPA` data type used the `length` facet to specify that the `GPA` must contain 4 characters, and the `pattern` facet to specify that the `GPA` must follow the particular format (for example 3.93):

```
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:length value="4"/>
    <xsd:pattern value="\d{1}.\d{2}"/>
  </xsd:restriction>
</xsd:simpleType>
```

Suppose that we wanted to restrict the `GPA` to be one of the following values: 2.0, 2.5, 3.0, 3.5, or 4.0. In such an instance, we could use the `enumeration` facet. Here's how that would look:

```
<xsd:simpleType name="StudentGPA" minOccurs="1" maxOccurs="1">
  <xsd:restriction base="xsd:string">
    <xsd:enumeration value="2.0"/>
    <xsd:enumeration value="2.5"/>
    <xsd:enumeration value="3.0"/>
    <xsd:enumeration value="3.5"/>
    <xsd:enumeration value="4.0"/>
  </xsd:restriction>
</xsd:simpleType>
```

The enumeration facet requires a document to contain one of its specified values for it to be valid. Note that, in this instance, there must be a 2.0, *or* a 2.5, *or* a 3.0, and so on. Yet, in the previous example, we required the length to be 4 *and* the pattern to follow the specified format. How can we tell when AND is being enforced, versus when OR is being enforced? The answer is simple: patterns and enumerations create OR scenarios. When you use a pattern or enumeration facet, the value must be one of those specified for the document element to be valid. All other facets create AND scenarios, which means that all values must be present for the document element to be valid.

Now, let's take a look at some examples using the int data type. For starters, you should know that the int data type has the following facets:

- enumeration
- fractionDigits
- maxExclusive
- maxInclusive
- minExclusive
- minInclusive
- pattern
- totalDigits
- whitespace

Suppose that we want to create a new data type to validate the course numbers that students are enrolled in. Each student can be enrolled in 0 to 10 classes a semester. Further, suppose that all course numbers are numbered in the following range: 1000 to 3000. We can use the int data type as our base type and then restrict the range of valid values using the minInclusive and maxInclusive facets:

```
<xsd:simpleType name="CourseNumber" minOccurs="0" maxOccurs="10">
  <xsd:restriction base="xsd:integer">
    <xsd:minInclusive value="1000"/>
    <xsd:maxInclusive value="3000"/>
  </xsd:restriction>
</xsd:simpleType>
<xsd:element name="Course" type="CourseNumber"/>
```

Now, if an XML document references this schema and it contains this element:

```
<Course>4000</Course>
```

an XSD parser will generate an error indicating a problem with the data type, because the number is out of the allowed range.

In this section, we have only begun to scratch the surface of the ways we can use the existing XSD data types or create our own.

For more information about XSD simple data types, or creating your own data types, please see the XSD data types document on the W3C's web site at http://www.w3.org/TR/xmlschema-2.

Validating and Using XSD Schemas

Everything we have learned up to this point concerns the basics of writing XSD schemas. But how are XSD schemas supported in SQL Server 2000 and other applications? What parsers are available for validating XSD today? How can you translate existing XDR schemas into XSD? Those topics will be covered in this section.

SQL Server Support of XSD

Presently, SQL Server 2000 only supports the XDR schema standard. It does not support the XSD standard. However, Microsoft has announced plans to make support for XSD part of its core XML services at some point after XSD becomes a Recommendation from the W3C. In fact, in April 2001, Microsoft announced the availability of the MSXML Parser 4.0 Technology Preview, which supports the Proposed Recommendation version of XSD. Thus, the official MSXML 4.0 release will have an XSD schema validator, and a future release of Microsoft SQL Server will also support the XSD schema.

Microsoft Office XP and the Beta 2 version of Microsoft Visual Studio.Net already have some support for XSD, so all of these factors are very good evidence that Microsoft will be providing future support for XSD as it becomes finalized.

Validating XSD Schemas

As mentioned previously, the MSXML Parser 4.0 Technology Preview supports XSD schema validation. There are also some validators that have been created by other parties. A number of validators presently available can be found on the W3C's web site at http://www.w3.org/XML/Schema.html.

So you can either go to the Microsoft web site and download the MSXML Parser 4.0 Technology Preview (or the release version of MSXML 4.0, once it is available) or work with one of the other validators. The bottom line is that you have plenty of resources available to start working with and validating your XSD schemas today.

Translating XDR Schemas into XSD Schemas

It is not possible to go from XSD to XDR with a translation tool, because of the functionality that wouldn't be supported. However, it is possible to translate from XDR into XSD, which is most likely what you would want to do anyway. There are already a few such translation tools available in beta formats today. Since the XSD schema itself is not finalized, however, these tools are far from being complete. One example of a tool to convert XDR to XSD is listed in the Microsoft .Net Framework beta documentation. It lists an XML Schema Definition Tool (`xsd.exe`) with a variety of features, including one that converts XDR to XSD. As at the writing of this book, information about the tool was present at the following link:

http://msdn.microsoft.com/library/dotnet/cptools/cpconxmlschemadefinitiontoolxsdexe.htm

Once XSD is finalized, there will likely be some very comprehensive conversion tools available to help users migrate from XDR to XSD.

> **Important Note: On May 2, 2001, the W3C promoted XML Schema Definitions (XSD) to the Recommendation Stage. Now that XSD is a full Recommendation, it is effectively the worldwide standard for XML Schemas. Please consult the W3C's web site at http://www.w3.org for the latest documentation.**
>
> **Additionally, on April 30, 2001, Microsoft released the Beta 1 Version of SQL Server 2000 Web Release 2. This version of Web Release 2 provides support for XSD in SQL Server. More information about this beta release can be found in Appendix C.**

Summary

XSD schemas provide a great way to define valid instances of XML documents. They are very powerful and more flexible than DTDs and XDR schemas. Let's review some of the key points highlighted in this chapter:

- XSD schemas are a powerful alternative to other technologies. They provide much richer functionality than DTDs and XDR schemas.
- XSD schemas allow you to create your own data types.
- XSD schemas allow you to implement object-oriented style inheritance.
- MSXML 4.0 will support XSD, and a technology preview of MSXML 4.0 that supports XSD was made available for download on Microsoft's web site in April 2001.
- You can begin working with XSD today and can validate your schemas using the MSXML 4.0 Technology Preview, as well as a number of other schema validators.
- There are some translation tools in beta versions right now that allow you to translate from XDR to XSD.
- XSD is the new worldwide vendor neutral standard for schema definitions, so you should start becoming familiar with it today!

XSD will have a great future. The only questions that remain regard what exact syntax it will employ. The industry has been anxiously awaiting a formal schema with this level of power and flexibility for a long time. If all goes well, XSD will be a full Recommendation by the end of 2001. Once it becomes a full Recommendation, more and more XSD support will be offered by applications like SQL Server. We are already seeing XSD support in the latest software tools being developed by Microsoft today, like Microsoft Office XP, Microsoft Visual Studio.Net, and MSXML 4.0.

Our next chapter gets us back on the SQL Server track, exploring how XML templates can be used and why they are so beneficial.