Getting Started with XML Schemas

This chapter introduces the W3C's XML Schema Recommendation and covers the mechanisms by which XML Schema allows us to define the elements and attributes we want to allow in our XML documents. We will also see how we can constrain element content and attribute values to have a particular datatype.

In particular this chapter will cover:

- The aims behind the W3C XML Schema specification
- How to create a simple XML Schema
- How to declare elements and attributes
- How to use some of the built-in simple datatypes: `string`, `integer` and `date`
- How to validate an XML document against a schema

This chapter will serve simply as a starting point for you. There are lots of other topics that you need to understand in order to take full advantage of XML Schemas, and as we go through the chapters of this book, you will be building up your experience and writing increasingly complex schemas.

> *To get you familiar with these concepts, the XML documents in this chapter do not use XML Namespaces, and the markup we are creating does not belong to a namespace. If you wish to write documents that make use of namespaces make sure that you read Chapter 6 before writing your schemas.*

Why do we Need Schemas?

XML is intended to be a self-describing data format, allowing authors to define a set of element and attribute names that describe the content of a document. As XML allows the author such flexibility, we need to be able to define what element and attribute names are allowed to appear in a conforming document in order to make that document useful. Furthermore, we need to be able to indicate what sort of content each of these elements and attributes are allowed to contain. Only then can people share the meaning of the markup used in an XML document, be it for human or application consumption.

Sometimes authors require flexibility in what markup they can use to describe a document's content, while at other times they may be forced to adopt a very specific structure. For example, if we were working on an application for a publishing company, we might define a set of elements such as `Book`, `Chapter`, `Heading1`, `Heading2`, `Heading3`, `Paragraph`, `Table`, `CrossReference`, and `Diagram`. Each `Book` element would be allowed to contain any number of `Chapter` elements, which in turn would contain `Heading` and `Paragraph` elements. The `Paragraph` elements may then contain text, tables, cross references and diagrams. In such a case, the people marking up the book's content need a flexible way of indicating what information is held within each element as no two books are going to have exactly the same content. By contrast, if we were writing an e-commerce system, it would be the job of an application, rather than a human, to create and process the XML documents. Each part of the process would require a different type document, one structure for catalogs, one for purchase orders, one for receipts, and so on. In such situations, rather than there being a requirement for flexibility, the application would expect a predictable, rigid structure; it would need certain pieces of information in order to fulfill any given task.

As XML becomes more widely used in applications, there is an increasing demand for support of primitive datatypes found in languages like SQL, Java, Visual Basic or C++ (the concepts of strings, dates, integers, and so on). XML Schema introduces a powerful type mechanism that not only allows us to specify primitive datatypes, but also types of structures, allowing us to integrate principles of object-oriented development such as inheritance into our schemas.

> **A schema defines the allowable contents of a class of XML documents. A class of documents refers to all possible permutations of structure in documents that will still conform to the rules of the schema.**

Background to XML Schemas

When XML was created, it was written as a simplified form of an existing markup language, called SGML, which was used for document markup. SGML, however, was so complex that it was not widely adopted, and browser manufacturers made it clear that they were not going to support it in their products. The simpler relative, XML, became a popular alternative, and was soon adopted by all kinds of programmers, not just those involved in document markup. When XML 1.0 became a W3C recommendation, it contained a mechanism for constraining the allowable content of a class of XML document, which you are probably familiar with, in the form of **Document Type Definitions** or **DTD**s. The syntax of DTDs, however, fell short of the requirements of those who were putting XML to new uses, in particular data transfer, and as a result the W3C wanted to create an alternative schema language, namely XML Schema.

The W3C XML Schema Working Group has had the incredibly tough task of creating a schema specification that would satisfy a wide range of users, from programmers to content architects, many of whom have been waiting for XML Schema with much anticipation because they see it as a much more powerful way to define document structures. Indeed, it has been a long time in coming, and there was a gap of over two years between the working group releasing a set of requirements they aimed to achieve with the new schema language, back in February 1999, and the recommendation's release in May 2001.

> *In the time the W3C have taken to release the XML Schema Recommendation, a number of alternative schema technologies have been released. While this one is likely to achieve wide support because of its endorsement by the W3C, the competing technologies offer alternative approaches to constraining allowable contents of an XML document. This book mainly focuses on the W3C XML Schema Recommendation, although we do look at some of the other schema efforts in Chapter 14.*

The aims of the W3C XML Schema Working Group were to create a schema language that would be more expressive than DTDs and written in XML syntax. In addition it would also allow authors to place restrictions on the allowable element content and attribute values in terms of primitive datatypes found in languages such as SQL and Java.

In terms of defining structure of documents, the aims included:

- Providing mechanisms for constraining document structures and content
- Allowing tighter or looser constraints upon classes of documents than those offered by DTDs
- The ability to validate documents composed from markup belonging to multiple namespaces
- Mechanisms to enable inheritance for element, attribute, and datatype definitions, so that they can formally represent *kind-of* relations (for example, a car is a *kind-of* vehicle)
- Mechanism for embedded documentation

In terms of offering primitive data typing, the aims included:

- Support for primitive datatypes such as byte, date, and integer, as found in languages like SQL and Java
- Definition of a type system that would support import and export of data as XML to and from relational, object and OLAP database systems
- The ability to allow users to define their own datatypes that derive from existing datatypes by constraining certain of their properties, such as range and length

> *The full requirements can be seen at: http://www.w3.org/TR/NOTE-xml-schema-req*

The result is a powerful and flexible language for expressing permissible content of a class of XML documents. The added capabilities, however, come at a cost: the resulting language is complicated, especially when we begin to experiment with its more advanced aspects.

The W3C XML Schema Recommendation

The W3C Recommendation for XML Schema comes in three parts:

- **XML Schema Part 0: Primer** The first part is a descriptive, example-based document, which introduces some of the key features of XML Schema by way of sample schemas. It is easy to read, and is a good start for getting to grips with XML Schemas and understanding what they are capable of. It can be read at http://www.w3.org/TR/xmlschema-0/.
- **XML Schema Part 1: Structures** The next part describes how to constrain the structure of XML documents – where the information items (elements, attributes, notations, and so on) can appear in the schema. Once we have declared an element or an attribute, we can then define allowable content or values for each. It also defines the rules governing schema-validation of documents. It can be read at http://www.w3.org/TR/xmlschema-1/.
- **XML Schema Part 2: Datatypes** The third part defines a set of built-in datatypes, which can be associated with element content and attribute values; further restricting allowable content of conforming documents and facilitating the management of dates, numbers, and other special forms of information by software processing of the XML documents. It also describes ways in which we can control derivation of new types from those that we have defined. It can be read at http://www.w3.org/TR/xmlschema-2/.

As we shall see throughout the course of this chapter and the rest of the book, there are a number of advantages to using XML Schemas over DTDs. In particular:

- As they are written in XML syntax (which DTDs were not), we do not have a new syntax to learn before we can start learning the rules of writing a schema. It also means that we can use any of the tools we would use to work with XML documents (from authoring tools, through SAX and DOM, to XSLT), to work with XML Schemas.
- The support for datatypes used in most common programming languages, and the ability to create our own datatypes, means that we can constrain the document content to the appropriate type required by applications, and / or replicate the properties of fields found in databases.
- It provides a powerful class and type system allowing an explicit way of extending and re-using markup constructs, such as content models, which is far more powerful than the use of parameter entities in DTDs, and a way of describing classes of elements to facilitate inheritance.
- The support for XML Namespaces allows us to validate documents that use markup from multiple namespaces and means that we can re-use constructs from schemas already defined in a different namespace.
- They are more powerful than DTDs at constraining mixed content models.

Getting Started with XML Schemas

The best way to start learning the syntax for XML Schemas is to jump in with an example. To start with, we will create a schema for the following simple document:

```
<?xml version = "1.0" ?>
<Customer>
   <FirstName>Raymond</FirstName>
   <MiddleInitial>G</MiddleInitial>
   <LastName>Bayliss</LastName>
</Customer>
```

A document conforming to a schema is known as an instance document, so let's have a look at an XML Schema for this instance document; we will go through it line by line in a moment (name the file `Customer.xsd`):

```
<?xml version = "1.0" ?>
<schema xmlns = "http://www.w3.org/2001/XMLSchema">
   <element name = "Customer">
      <complexType>
         <sequence>
            <element name = "FirstName" type = "string" />
            <element name = "MiddleInitial" type = "string" />
            <element name = "LastName" type = "string" />
         </sequence>
      </complexType>
   </element>
</schema>
```

> **XML Schema files are saved with the `.xsd` extension.**

As you can see, the `Customer.xsd` schema is itself an XML document, and the root element of any XML Schema document is an element called `schema`. In the opening `schema` tag we declare the namespace for the XML Schema Recommendation:

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

The next line indicates how we declare our first element, the `Customer` element:

```
   <element name = "Customer">
...
   </element>
```

As XML is intended to be a self-describing data format, it is hardly surprising that we declare elements using an element called `element`, and we specify the intended name of the element as a value of an attribute called `name`. In our case, the root element is called `Customer`, so we give this as the value of the `name` attribute.

We will come back to the `complexType` element that appears on the next line in just a moment, but looking further down the schema we can see the declarations for the three other elements that appear in the document: one called `FirstName`, one called `MiddleInitial`, and one called `LastName`.

```
<sequence>
   <element name = "FirstName" type = "string" />
   <element name = "MiddleInitial" type = "string" />
   <element name = "LastName" type = "string" />
</sequence>
```

You may be able to guess from the way in which the elements are declared, nested inside an element called `sequence`, that they would have to appear in that same order in a conforming document. The `sequence` element is known as a **compositor**, and we are required to specify a compositor inside the `complexType` element – we will meet other types of compositor in Chapter 3.

In addition, the element declarations carry a `type` attribute, whose value is `string`. XML Schema introduces the ability to declare types such as `string`, `date` and `integer`, as we would find in languages such as SQL and Java; this is how we specify such types.

Let's now come back to the element we have not looked at yet, called `complexType`, which contains the declarations of the elements that appear as children of the `Customer` element in our sample XML document. XML Schema makes a distinction between simple types and complex types.

The Difference Between Simple and Complex Types

There are two kinds of type in XML Schema: simple types and complex types, both of which constrain the allowable content of an element or attribute:

- **Simple types** restrict the text that is allowed to appear as an attribute value, or text-only element content (text-only elements do not carry attributes or contain child elements)
- **Complex types** restrict the allowable content of elements, in terms of the attributes they can carry, and child elements they can contain

Let's have a closer look at what this means.

Simple Types

All attribute values and text-only element content simply consists of strings of characters. The ability for XML Schema to support datatypes means that we can place restrictions on the characters that can appear in attribute values and text-only element content.

An example of such a restriction is the representation of a Boolean value, in which case XML Schema only allows the character strings: `true`, `false`, `1`, or `0`. After all, an instance document should not be allowed to use values such as "`maybe`" or "`4`" in attributes or elements that are supposed to represent a Boolean value. Alternatively, if we wanted to represent a byte, we would only want characters that are an integer whose value is between `-128` and `127`, so that `1445` would not be allowed and neither would `ff23`.

An XML Schema aware processor is required to support a number of **built-in simple types** that are considered common in programming languages and databases, and a number of datatypes that the working group thought were important to XML document authors. This is why we were allowed to specify that the content of the `FirstName`, `MiddleInital`, and `LastName` elements were **strings** (which places very little restriction on the allowable text of the element content):

```
<element name = "FirstName" type = "string" />
<element name = "MiddleInitial" type = "string" />
<element name = "LastName" type = "string" />
```

In addition to the built-in simple types, XML Schema allows us to derive our own simple types that restrict the allowable content of the built-in simple types already defined in XML Schema.

We will look into all of the built-in simple types in the next chapter. The rest of this chapter will stick to using the built-in types of `string`, `date`, and `integer`.

Complex Types

Complex types define the attributes an element can carry, and the child elements that an element can contain. Whenever we want to allow an element to carry an attribute or contain a child element, we have to define a complex type.

The `Customer` element declared in the `Customer.xsd` example is allowed to contain three child elements (`FirstName`, `MiddleInitial`, and `LastName`), and therefore needs to be a complex type. We gave the `Customer` element a complex type using the `complexType` element nested inside the element that declared `Customer`. We then declared the number of child elements the element `Customer` is allowed to contain inside the `complexType` element and its compositor `sequence`, like so:

```
<complexType>
   <sequence>
      <element name = "FirstName" type = "string" />
      <element name = "MiddleInitial" type = "string" />
      <element name = "LastName" type = "string" />
   </sequence>
</complexType>
```

Note that we cannot just nest the other element declarations inside each other. The following would *not* be allowed:

```
<element name = "Customer">
   <element name = "FirstName" type = "string" />
   <element name = "MiddleInitial" type = "string" />
   <element name = "LastName" type = "string" />
</element>
```

This is not allowed because we need to define the complex type in order for the `Customer` element to contain child elements.

The complex type defined above is known as an **anonymous complex type**. This is because it is nested within the element declaration (`Customer`, in this case). If we wanted more than one element to contain the *same* child elements and carry the *same* attributes, then we would create a **named complex type**, which would apply the same restrictions to the content of our new element. We look at named complex types in Chapter 3.

Let's quickly add to the `Customer` element in our example XML document, by giving it an attribute called `customerID`, so that we can see how we declare attributes. We want the new document to look as follows:

```
<?xml version = "1.0" ?>
<Customer customerID = "24332">
   <FirstName>Raymond</FirstName>
   <MiddleInitial>G</MiddleInitial>
   <LastName>Bayliss</LastName>
</Customer>
```

To add the attribute we can just declare it within the `complexType` definition, after the closing `sequence` compositor tag and just before the closing `complexType` tag:

```
<?xml version = "1.0" ?>
<schema xmlns = "http://www.w3.org/2001/XMLSchema">
   <element name = "Customer">
      <complexType>
         <sequence>
            <element name = "FirstName" type = "string" />
            <element name = "MiddleInitial" type = "string" />
            <element name = "LastName" type = "string" />
         </sequence>
         <attribute name = "customerID" type = "integer" />
      </complexType>
   </element>
</schema>
```

We declare an attribute using an element called `attribute`. As with the element declaration, it carries an attribute called `name` whose value is the name of the attribute. Remember the value of an attribute is always a simple type; in this case we want our `customerID` attribute to be represented as an integer, so we can use the built-in type of `integer` to restrict the value of the attribute to an integer value.

> **Note the distinction that elements and attributes are declared, while simple and complex types are defined.**

Let's start to look at each of the schema constructs in greater depth.

Element Declarations

The declaration of an element involves associating a name with a type. Earlier, we saw how to declare an element using an element called `element`, and that it's name is given as the value of the `name` attribute that the element declaration carries. The type meanwhile would be a simple type if the element had text-only content, otherwise it would be a complex type. The type of the element can be given in one of two ways:

- A type definition can be anonymous, and nested inside the element declaration, as we saw with the child elements of `Customer` in the first example.
- A type can be referred to, by putting the name of the type as the value of a `type` attribute, as we have been doing with the value `string`.

In the following example we can see a mix of the two approaches. The `Address` element declaration contains an anonymous type, while the child elements are all given a simple type of `string`:

```
<element name = "Address">
   <complexType>
      <sequence>
         <element name = "Street" type = "string" />
         <element name = "Town" type = "string" />
         <element name = "City" type = "string" />
         <element name = "StateProvinceCounty" type = "string" />
         <element name = "Country" type = "string" />
         <element name = "ZipPostCode" type = "string" />
      </sequence>
   </complexType>
</element>
```

Here is an example of an `Address` element that conforms to this schema:

```
<Address>
   <Street>10 Elizabeth Place</Street>
   <Town>Paddington</Town>
   <City>Sydney</City>
   <StateProvinceCounty>NSW</StateProvinceCounty>
   <Country>Australia</Country>
   <ZipPostCode>2021</ZipPostCode>
</Address>
```

If we do not specify a type, then the element can contain any mix of elements, attributes and text. This is known as the **ur-type** type in XML Schema, although you do not actually refer to it by name, it is just the default if you do not specify a type.

Global versus Local Element Declarations

It is important to distinguish between the global and local element declarations:

- **Global element declarations** are children of the root `schema` element
- **Local element declarations** are nested further inside the schema structure and are not direct children of the root `schema` element

Once elements have been declared globally, any other complex type can use that element declaration, by creating a **reference** to it. This is especially helpful when an element and its content model are used in other element declarations and complex type definitions, as they enable us to re-use the content model (A content model simply refers to anything within an element declaration that affects the structure of the element in the instance document. This could be attributes or other elements within an element).

> *You should be aware that, if your instance documents make use of namespaces, there are greater differences between local and global element declarations. This is because when you use namespaces, globally declared elements must be explicitly qualified in the instance document, whereas local declarations should not always be qualified. We look into the issues that this introduces and the ways in which it might affect how you write XML Schemas in Chapter 6.*

Imagine that we wanted to alter our `Customer.xsd` schema so that we could represent name and address details for customers as below. Note we have also added a containing element for the name details called `Name`:

```
<?xml version = "1.0" ?>
<Customer customerID = "242552">
   <Name>
      <FirstName>Raymond</FirstName>
      <MiddleInitial>G</MiddleInitial>
      <LastName>Bayliss</LastName>
   </Name>
   <Address>
      <Street1>10 Elizabeth Place</Street1>
      <Town>Paddington</Town>
      <City>Sydney</City>
      <StateProvinceCounty>NSW</StateProvinceCounty>
      <Country>Australia</Country>
      <ZipPostCode>2021</ZipPostCode>
   </Address>
</Customer>
```

We also want it to be able to use the same schema to validate details about employees. In this case the details would be contained in an `Employee` element, although the child elements of each are the same:

```
<?xml version = "1.0" ?>
<Employee employeeID = "133">
   <Name>
      <FirstName>Raymond</FirstName>
      <MiddleInitial>G</MiddleInitial>
      <LastName>Bayliss</LastName>
   </Name>
   <Address>
      <Street1>10 Elizabeth Place</Street1>
      <Town>Paddington</Town>
      <City>Sydney</City>
      <StateProvinceCounty>NSW</StateProvinceCounty>
      <Country>Australia</Country>
      <ZipPostCode>2021</ZipPostCode>
   </Address>
</Employee>
```

Seeing as both the `Customer` and `Employee` elements contain a `Name` element and an `Address` element, both of which have the same content models, we can define the `Name` and `Address` elements globally, and then use a reference to the global declarations inside the declarations for the `Customer` and `Employee` elements. When we want to create a reference to a globally declared element, we use the `ref` attribute on the element declaration, whose value is the name of the element that we are referencing.

In order to use references to elements we have declared, we will qualify all of the elements defined by XML Schema using a **namespace prefix** (We will look into the reasons behind this in Chapter 6). This is what the schema looks like now:

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

   <xs:element name = "Customer">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref = "Name" />
            <xs:element ref = "Address" />
         </xs:sequence>
         <xs:attribute name = "customerID" type = "integer" />
      </xs:complexType>
   </xs:element>

   <xs:element name = "Employee">
      <xs:complexType>
         <xs:sequence>
            <xs:element ref = "Name" />
            <xs:element ref = "Address" />
         </xs:sequence>
         <xs:attribute name = "employeeID" type = "integer" />
      </xs:complexType>
   </xs:element>

   <xs:element name = "Name">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "FirstName" type = "string" />
            <xs:element name = "MiddleInitial" type = "string" />
            <xs:element name = "LastName" type = "string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name = "Address">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "Street1" type = "string" />
            <xs:element name = "Town" type = "string" />
            <xs:element name = "City" type = "string" />
            <xs:element name = "StateProvinceCounty" type = "string" />
            <xs:element name = "Country" type = "string" />
            <xs:element name = "ZipPostCode" type = "string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

</xs:schema>
```

Firstly you will notice the use of the `xs:` prefix on all of the elements defined by XML Schema. This is declared in the root `schema` element:

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

Next you can see that both the `Customer` element and the `Employee` element declarations contain a reference to the `Name` and `Address` elements:

```
<xs:element name = "Employee">
   <xs:complexType>
      <xs:sequence>
         <xs:element ref = "Name" />
         <xs:element ref = "Address" />
      </xs:sequence>
      <xs:attribute name = "employeeID" type = "integer" />
   </xs:complexType>
</xs:element>
```

This enables re-use of element declarations and saves repeating the element declarations inside each element. Any other element or complex type definition in the schema could use these globally defined elements. It is helpful whenever we have an element that may appear in more than one place in a document instance.

It is important to note, however, that any globally defined element can be used as the root element of a document. The only way of enforcing only one root element in a document is to only have one globally defined element, and to carefully nest all other element declarations inside complex type definitions. The benefits of this approach are that we can create a structure that can be used to validate fragments of documents without having to define separate schemas for each fragment, and that it allows us to define one schema for several classes of document. So, we would be able to validate the following document against this schema:

```
<?xml version = "1.0" ?>
<Address>
   <Street1>10 Elizabeth Place</Street1>
   <Town>Paddington</Town>
   <City>Sydney</City>
   <StateProvinceCounty>NSW</StateProvinceCounty>
   <Country>Australia</Country>
   <ZipPostCode>2021</ZipPostCode>
</Address>
```

This document is considered valid because the `Address` element has been declared globally. This may not be desirable, and we will see other approaches as we go through the book. We look into this topic more in Chapter 7.

Note that an element declaration that carries a `ref` attribute cannot also carry a `name` attribute, nor can it contain a complex type definition.

Element Occurrence Indicators

By default, when we declare an element in an XML Schema it is required to appear once and once only. However there are times when we might want to make the appearance of an element in a document optional. For example, we might want to make the `MiddleInitial` child element of our `Customer` element optional in case the customer does not have a middle name. Indeed there may be times when we want an element to be repeatable; for example, we might want to allow several `MiddleInitial` elements if the customer has several middle names.

To replicate the functionality offered by the cardinality operators in DTDs, namely `?`, `*`, and `+`, which indicate how many times an element could appear in an instance document, XML Schema introduces two occurrence constraints which take the form of attributes on the element declaration: `minOccurs` and `maxOccurs`. Their value indicates how many times the element can appear, and are a lot simpler to use than the cardinality operators in DTDs because we just specify a minimum and maximum number of times that an element can appear. The `maxOccurs` attribute can also take a value of `unbounded`, which means that there is no maximum number of times the element can appear in the document instance.

The following table shows the mapping of DTD cardinality operators to the equivalent values of `minOccurs` and `maxOccurs` XML Schema attributes:

Cardinality Operator	minOccurs Value	maxOccurs Value	Number of Child Element(s)
[none]	`1`	`1`	One and only one
`?`	`0`	`1`	Zero or one
`*`	`0`	`unbounded`	Zero or more
`+`	`1`	`unbounded`	One or more

Let's look at some examples. To start, if we want an element to appear once and once only, then we do not have to add anything to the declaration, as the default values for both attributes if not included are `1`. However, for clarity we could explicitly state that the `MiddleInitial` element must appear once and only once:

```
<element name = "MiddleInitial" type = "string" minOccurs = "1"
         maxOccurs = "1" />
```

If we wanted to make the element optional, so that the element could appear but is not required to do so, and that when it did appear it could only appear once we could use the following:

```
<element name = "MiddleInitial" type = "string" minOccurs = "0"
         maxOccurs = "1" />
```

If we wanted to require at least one `MiddleInitial` element, yet allow no more than 4 we could use the following:

```
<element name = "MiddleInitial" type = "string" minOccurs = "1"
         maxOccurs = "4" />
```

If we wanted to make sure that there were at least two `MiddleInitial` elements, but that there were no upper limits on the number of times the element could appear, we could use the following:

```
<element name = "MiddleInitial" type = "string" minOccurs = "2"
         maxOccurs = "unbounded" />
```

Note that you cannot declare `minOccurs` and `maxOccurs` on global elements, only on local element declarations.

While we cannot use the `minOccurs` and `maxOccurs` attributes on a global element declaration, we can add them to a local element declaration that references a global declaration using the `ref` attribute:

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

   <element name = "Customer">
      <complexType>
         <sequence>
            <element ref = "FirstName" minOccurs = "0" maxOccurs = "1" />
            <element ref = "MiddleInitial"
                     minOccurs = "0" maxOccurs = "unbounded" />
            <element ref = "LastName" minOccurs = "1" maxOccurs = "1" />
         </sequence>
         <attribute name = "customerID" type = "integer" />
      </complexType>
   </element>

   <element name = "FirstName" type = "string" />
   <element name = "MiddleInitial" type = "string" />
   <element name = "LastName" type = "string" />

</schema>
```

Here the `FirstName` is optional, the `MiddleInitial` element is optional although it can appear as many times as the document author requires, and the `LastName` is required.

Value Constraints on Element Content – Default and Fixed Content

With DTDs we could supply a default attribute value for an attribute that was left empty in an instance document, but there was no equivalent mechanism for elements. With XML Schema, we can supply a default value for text-only element content.

If we specify a default value for an element, and that element is empty in the instance document, an XML Schema aware processor would treat the document as though it had the default value when it parses the document. In the following example we have a fragment of an XML instance document, which is used to profile a member's subscription to a web site:

```
<MailOut>
   <Subscribe></Subscribe>
</MailOut>
```

We want the default content of the `Subscribe` element to be `yes`, so we add a `default` attribute to the element declaration, whose value is the simple element content we want:

```
<element name = "Subscribe" type = "string" default = "yes" />
```

Once parsed, if the `Subscribe` element were empty in the instance document, the schema processor would treat the `Subscribe` element as if it had contained the string `yes`.

There is another attribute that we can add to an element declaration, called `fixed`. When `fixed` is used on an element declaration, the element's content must either be empty (in which case it behaves like `default`), or the element content must match the value of the `fixed` attribute. If the document contained a value other than that expressed by the `fixed` attribute it would not be valid.

For example, if we wanted a `SecurityCleared` element to either contain the `boolean` value of `true`, or if empty to be treated as if it contains `true`, we would use the fixed attributes like this:

```
<element name = "SecurityCleared" type = "boolean" fixed = "true" />
```

Therefore, the following would be valid:

```
<SecurityCleared>true</SecurityCleared>
```

As would either of these:

```
<SecurityCleared></SecurityCleared>
<SecurityCleared />
```

In either of the above cases, the processor would treat the element as if it had the content `true`. However, the three examples below would not be valid:

```
<SecurityCleared>false</SecurityCleared>
<SecurityCleared>no</SecurityCleared>
<SecurityCleared><UserID>001</UserID></SecurityCleared>
```

It should be noted that the value of the element is measured against the permitted values for the datatype. We will look at datatypes in more detail in the next chapter, but the examples here are not valid because the only allowed values for a `boolean` whose value is `true`, are the string `true` or the value `1`. The following would be a valid example, because `1` is an allowed value for the datatype:

```
<SecurityCleared>1</SecurityCleared>
```

This would be helpful in preventing any documents being validated if they explicitly contained any content other than the string `true`.

Note that we could not add both a `default` and a `fixed` attribute to the same element declaration.

> **Together the `default` and `fixed` attributes are known as value constraints, because they constrain the values allowed in element content.**

Attribute Declarations

We declare attributes in a similar way to declaring elements. The key differences are:

- They cannot contain any child information items. Attribute values are always simple types.
- They are unordered; we cannot specify the order in which attributes should appear on a parent element.

This means that the value of the `type` attribute on an attribute declaration is always a simple type – a restriction upon the value of the attribute. If we do not specify a type, then by default it is the simple version of the ur-type definition, whose name is **anySimpleType**. This represents any legal character string in XML that matches the `Char` production in the XML 1.0 Recommendation, but we need to be aware that if we need to use characters such as angled brackets ([ ]) or an ampersand (&), these should be escaped using the escape characters or numeric character references defined in the XML 1.0 Recommendation.

Attributes are added to an element inside the complex type definition for that element; they are added after the content of the element is defined within the complex type:

```
<?xml version = "1.0" ?>
<schema>
   <element name = "Customer">
      <complexType>
         <sequence>
            <element name = "FirstName" type = "string" />
            <element name = "MiddleInitial" type = "string" />
            <element name = "LastName" type = "string" />
         </sequence>
         <attribute name = "customerID" type = "integer" />
      </complexType>
   </element>
</schema>
```

Here we can see that we have added the `customerID` attribute to the `Customer` element by including its declaration at the end of the complex type.

Global versus Local Attribute Declarations

As with element declarations, attribute declarations can either be local or global. If they are global declarations they are direct children of the `schema` element, meaning that any complex type definition can make use of the attribute.

> *As with global and local element declarations, you should be aware that, if your instance documents make use of namespaces, there are greater differences between local and global attribute declarations. This is because globally declared attributes must be explicitly qualified in the instance document, whereas local declarations should not always be qualified. We look into the issues that this introduces and the ways in which it might affect how you write XML Schemas in Chapter 6.*

Occurrence of Attributes

By default, when we declare an element to carry an attribute, its presence in an instance document is optional. While there is no provision for `minOccurs` and `maxOccurs` attributes on our attribute declarations, because an attribute can only appear once on any given element, we might want to specify that an attribute *must* appear on a given element.

If we want to indicate that an attribute's presence is required, or explicitly state that an attribute is optional, we can add an attribute called `use` to the attribute declaration, which can take one of the following values:

- `required` when indicating that an attribute must appear
- `optional` when it can either appear once or not at all (the default value)
- `prohibited` when we want to explicitly indicate that it must not appear

> **Note that we cannot add the `use` attribute to globally declared attributes.**

For example, if we just want to ensure that an attribute is present on the element, we can just add the `use` attribute to the attribute declaration with a value of `required`:

```
<attribute name="dateReceived" use="required" />
```

If the attribute is optional, we can use the value of `optional`, although this is not required as it is the default value:

```
<attribute name="child" use="optional" />
```

Value Constraints on Attributes

As we would expect from working with attributes in DTDs, we can supply default and fixed content for an attribute's value in the XML Schema. This works rather like the value constraints on the element declarations.

If an attribute is not included in an element in an instance document, we can use the schema to tell the processor: "When processing the document, treat the element as if it had this attribute with the value given in the schema". We give an attribute a default value by adding the `default` attribute to the attribute declaration, like this:

```
<attribute name = "currency" default = "US$" />
```

> **If you have a `default` value for an attribute, then the `use` value must be set to `optional`.**

Imagine that we wanted to be able to validate an XML document in the following format:

```
<CreditAccount currency = "US$">
   <AccountName>Ray Bayliss</AccountName>
   <AccountNumber>27012</AccountNumber>
   <Amount>200.00</Amount>
</CreditAccount>
```

In this example, we want to ensure that if the `CreditAccount` element does not have the `currency` attribute in the instance document, the processor acts as though the attribute is there, and that its value is `US$`. Here is an extract from a schema that will ensure this behavior:

```
<element name = "CreditAccount">
   <complexType>
      <sequence>
         <element name = "AccountName" type = "string" />
         <element name = "AccountNumber" type = "integer" />
         <element name = "Amount" type = "string" />
      </sequence>
      <attribute name = "currency" default = "US$" />
   </complexType>
</element>
```

If we want to indicate that the value of an attribute is the same as the value we prescribe in the schema, whether or not the attribute is present in the instance document, we can use the `fixed` attribute on the element declaration, like so:

```
      <attribute name = "currency" fixed = "US$" />
```

If the attribute does not appear in the document, the value of `fixed` would act as the `default` attribute, and the processor would treat the document as though the attribute were there and had the value specified.

For example, if the `CreditAccount` element was declared to have the following attribute declaration:

```
<element name = "CreditAccount">
   <complexType>
      <sequence>
         <element name = "AccountName" type = "string" />
         <element name = "AccountNumber" type = "integer" />
         <element name = "Amount" type = "string" />
      </sequence>
      <attribute name = "currency" fixed = "US$" />
   </complexType>
</element>
```

Then the following document instance would not be valid because the currency attribute has a value of `AUS$` not `US$`:

```
<CreditAccount currency = "AUS$">
   <AccountName>Ray Bayliss</AccountName>
   <AccountNumber>2701 2202</AccountNumber>
   <Amount>200.00</Amount>
</CreditAccount>
```

If the attribute were missing, the schema processor would treat the `CreditAccount` element as though it was carrying a currency attribute whose value is `US$`.

Note that you could not add both a `default` and a `fixed` attribute to the same attribute declaration.

> **Together the `default` and `fixed` attributes are known as value constraints, because they constrain the value of the attribute.**

Annotations

XML Schema offers two kinds of annotation to a schema, both of which appear as children of an element called `annotation`:

- `documentation` is rather like the ability to add comments. Using the `documentation` element, we can add information that will help us and others understand the intended purpose of our documents.
- `appinfo` offers a place in which we can provide additional information to a processing application.

As with all areas of programming, the use of comments is very important (even if they can be a nuisance to add at the time of writing). Of course they help the original author when they come back to use the schema later, but their use is also important for anyone else wanting to use the schema to help them understand the constructs – whether they are authoring documents according to the schema or writing an application to process documents according to the schema. As such, they will be especially helpful if the document author or programmer is not used to the schema syntax.

If we intend that others should use our schema, we should provide enough information in documentation elements to clarify any ambiguity regarding the intended purpose of an element or type. Additional information may also help users get to grips with a schema quicker.

Good use of documentation could make the difference in getting our schema adopted by a group of users over an alternative schema that is not as well documented.

DTD authors are allowed to use comments using the same syntax used for XML comments:

```
<!-- comment goes here -->
```

Indeed, we can include comments in this form in an XML Schema because it is an XML document itself, but this is not a good way of documenting the XML Schema for these reasons:

- By putting documentation in a `documentation` element, you can add structured documentation including markup such as XHTML, whereas XML comments cannot.
- You can easily make the schema self-documenting by adding a stylesheet to it.
- XML parsers can ignore comments. By providing an explicit `documentation` element, the information becomes available to any processing application. If the processing application is an authoring tool, it can pass on information from the `documentation` element to document authors allowing them to use the markup as it is intended

The `annotation` element can appear at the beginning of most schema constructs, although it will most commonly be used inside `element`, `attribute`, `simpleType`, `complexType`, `group`, and `schema` elements. Where we place the `annotation` and its child `documentation` will affect what the documentation applies to.

In our simple `Customer` example that we have been looking at through this chapter, we could provide copyright and author information at the root of the schema, and indicate to document authors that the `MiddleInitial` element is optional, although if the `Customer` has a middle name we should use it:

```
<?xml version = "1.0" ?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:annotation>
      <xs:documentation>
         Schema for customer name information.
         Used in Professional XML Schemas
         Copyright Wrox Press Ltd 2001, all rights reserved
         1102 Warwick Road, Acocks Green, Birmingham, B27 6BH. UK
      </xs:documentation>
   </xs:annotation>
   <xs:element name = "Customer">
   <xs:annotation>
      <xs:documentation>
         MiddleInitial is optional, but should be used if the customer has a
         middle name to help distinguish between customers with like names.
      </xs:documentation>
   </xs:annotation>
      <xs:complexType>
         <xs:group ref = "NameGroup" />
      </xs:complexType>
   </xs:element>
   <xs:group name = "NameGroup">
      <xs:sequence>
         <xs:element name = "FirstName" type = "xs:string" />
         <xs:element name = "MiddleInitial" type = "xs:string" />
         <xs:element name = "LastName" type = "xs:string" />
      </xs:sequence>
   </xs:group>
</xs:schema>
```

The `appinfo` child of the `annotation` element is designed to pass information to a processing application, stylesheet, or other tool. This will be a particular advantage to schema users if XML Schema compliant parsers implement a way of passing this information to an application, because those who used XML 1.0 processing instructions to pass information to the processing application often had to write custom parsers in order to do this. Therefore, a lot of developers who could have made use of processing instructions ended up putting that information in application code, making the resulting application less flexible. By allowing information to be put into the `appinfo` element, programmers can either pass information to the application about how the section of a conforming document should be processed, or they can add extra code inside the `appinfo` elements.

The `appinfo` element is subject to the same rules for appearing in an XML Schema as the `documentation` element, as they are both contained in the `annotation` element. This means that it can be used within most schema constructs. In the following example we have nested some script inside the `appinfo` element, which is intended to indicate to an application what action to take, depending upon which of a choice of two elements a document instance contains:

```
   <xs:group name="CreditOrDebitGroup">
      <xs:annotation>
         <xs:appinfo>
            if (currentNode.firstChild != "Credit")
               docParser.load(debitURL);
            else
               document.write("Your account will be credited within 24
                              hours.");
```

```
            </xs:appinfo>
         </xs:annotation>
         <xs:choice>
            <xs:element name = "Credit" type = "CreditType" />
            <xs:element name = "Debit" type = "DebitType" />
         </xs:choice>
      </xs:group>
```

The script buried inside the `appinfo` element can be passed to an application that is using the schema to validate an instance of the document. In this case, the script in the `appinfo` element can be passed to a processing application to indicate how to handle each element in the choice group, depending upon which element the document contains.

We look at annotation in more detail in Chapter 10 on Schemas and XSLT. There is also an interesting example of using the `appinfo` element to contain Schematron rules in Chapter 14.

Validating an Instance Document

Having understood some of the basics for writing XML Schemas, we should look at how we validate document instances. You may have noticed that none of the sample XML documents in this chapter have indicated a link to the XML Schema they are supposed to correspond to. They have not included an equivalent of the Document Type Declaration (whether it refers to inline definitions or an external DTD). This is because there is no direct link of any kind between an instance document and its XML Schema.

A document author can indicate where a copy of the schema they used to write the document can be found using the `xsi:schemaLocation` attribute, whose value is a URL, but there is no requirement for the processor to use the indicated schema. For example we could use the following to indicate where the `Customer.xsd` file can be found:

```
<?xml version = "1.0" ?>
<Customer xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi:schemaLocation = "http://www.wrox.com/ProXMLSchemas/
          Customer.xsd">
   ...
</Customer>
```

Note that we have had to declare the XML Schema for Instance Documents namespace and its prefix `xsi:` in order to use the `schemaLocation` attribute (as the `schemaLocation` attribute is defined in that namespace).

Parsers can ignore or override the suggestion in the `schemaLocation` attribute; they may decide to use a different schema or use a cached copy of the suggested schema.

Sometimes it is helpful to be able to validate a document against a different schema than that which it was authored against. Therefore we can leave it up to the program that hands the XML document to the parser to say which schema to use to validate it.

Note also that we have not so far been indicating the intended namespace to which our schema belongs. This means that the markup we have been creating does not belong to a namespace. In this case we need to use the `xsi:noNamespaceSchemaLocation` attribute on the root element, like this:

```
<?xml version = "1.0" ?>
<Customer xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance"
          xsi:noNamespaceSchemaLocation = "Customer.xsd">
...
</Customer>
```

This indicates to the parser where it can find a copy of the schema that doesn't belongs to a namespace.

How the XML Schema Recommendation Specifies Validity

The XML Schema Recommendation does not indicate *how* an XML Schema aware processor should validate a document, so before we look at validation it is worthwhile taking a moment to understand how the XML Schema Recommendation determines validity. The XML Schema Recommendation is written in terms of an abstract model (rather like the DOM Recommendation). This corresponds to information items as defined in the **XML Information Set**.

The purpose of the XML Information Set (or **infoset**) is to provide a consistent set of definitions that can be used in other specifications that refer to information held within a well-formed XML document.

Any well-formed XML document has an **information set** (as long as it also conforms to the XML Namespaces Recommendation). This in turn means that an XML Schema and all instance documents must be well-formed in order for them to be processed by a parser. After all, a document that is not well-formed does not have an information set.

The infoset presents an XML document's information set as a modified tree. We should be clear however, that the XML Schema Recommendation does not require that an XML Schema aware processor's interfaces make the infoset available as a tree structure – the document may just as equally be accessed by an event-based approach (such as that implemented in SAX processors) or a query-based interface. However, the term information set can be treated as analogous to the term **tree**.

An XML document's information set consists of a number of **information items**, each of which can be treated as analogous to a **node** on the tree. An information item is an abstract representation of some part of a document, and each information item has a set of associated properties. At minimum, a well-formed XML document will have a document information item. There are 14 information items in all; here are the ones that we are most concerned with:

- The **document information item** is the unique element in which all other markup is nested within a well-formed XML document. In the case of an XML Schema document, the document information item would correspond to the `schema` element.
- An **element information item** exists for every element that appears in an XML document.
- An **attribute information item** exists for each attribute, whether specified or defaulted, of each element in the document.
- A **character information item** exists for each data character in the document, whether literally or as a character reference, or within a CDATA section. Each character is a logically separate information item, although many processing applications chunk characters into larger groups.
- A **namespace information item** exists for each namespace that is in the scope for that element.

By talking in terms of an abstract tree representation, the schema specification can then ensure that each information item in an instance document respects the constraints imposed by the corresponding information item in the schema. This is known as **local schema-validity**.

There is a second level of schema validity, which represents the overall validation outcome for each item. This is where the local schema-validity of an information item corresponds with the results of the schema-validity assessments performed upon its descendents, if it has any. So, a parent element is checked against the schema-validity assessments of its child information items.

Therefore, the XML Schema Recommendation does not have to worry about how the validating processor is implemented. As long as the information items are locally schema-valid, and they correspond with child information items, an instance document will be valid. At each stage, augmentations (in the form of properties) may be added to the information items in the information set to record the outcome and help the processor achieve its task.

So, each of the components that make up any schema are used to determine whether an element or attribute in an instance document is valid. In addition, a processor may check augmentations (such as default values) placed upon those elements, attributes, and their descendents.

Validating with XSV

At the time of writing, XML Schema has only recently become a full W3C recommendation, and there are limited tools available for validating instance documents using the final recommendation. A proliferation of compliant tools is expected to follow, but many are in still in beta version. Check out Appendix D for full discussion of XML Schema tools and XML Schema-compliant parsers. For now, however, we are just going to focus on one, XSV.

XSV (XML Schema Validator) is an ongoing open source project, developed at the University of Edinburgh in the UK by Henry Thompson and Richard Tobin (Henry Thompson is also co-author of the XML Schema Recommendation, Part 1). Written in Python, it is available for download either as source, or as a Win32 executable. Alternatively, you can use it as an online utility. XSV is available from:

- http://www.ltg.ed.ac.uk/~ht/xsv-status.html (for download)
- http://www.w3.org/2001/03/webdata/xsv (to use online)

> *The easiest way to use XSV is via the online web form. You can validate schemas on their own by simply uploading the file from your own machine, but if you want to validate instance documents against your schema, then you need to be able to make them available online. If this is difficult for you – if you are behind a firewall for example – then you may prefer to download XSV and install it on your own machine.*

Since this is ongoing work, there are frequent updates to the tool, and full details concerning which parts of the XML Schema recommendation are implemented is available from the first URL above. At the time of writing, this tool appears to be the one most fully conformant with the W3C recommendation.

> *Warning: One of the main limitations of XSV at the time of writing is its lack of support for validating simple types. The only checks that XSV makes on simples types are on length and enumerations.*

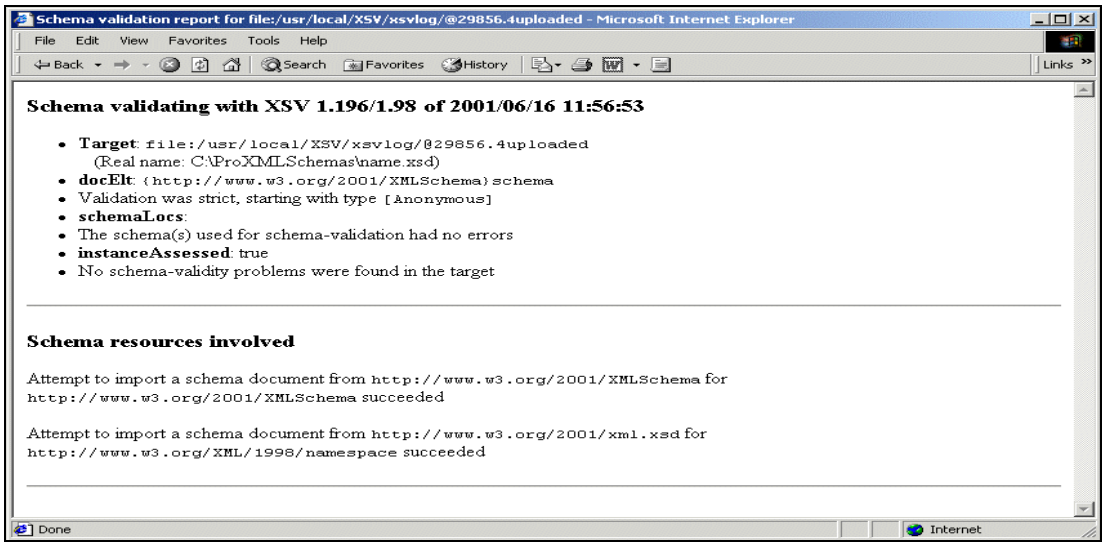
The download comes in the form of a self-installing executable for Win32. If you're working on a Unix platform, however, you'll need to download and compile the source files. Alternatively, you could check out some of the tools discussed in Appendix D, such as Turbo XML from TIBCO Extensibility Solutions.

Validating a Schema

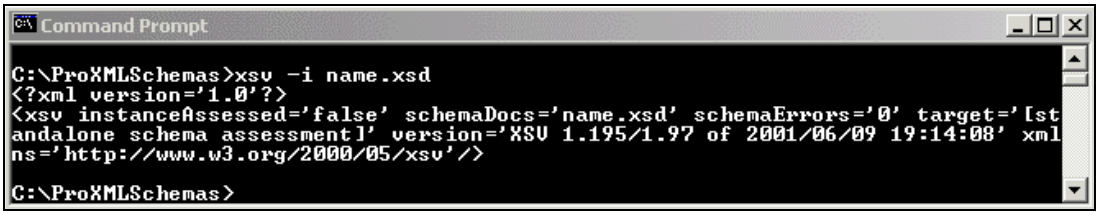
Let's start by validating a simple schema, `name.xsd`:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "Name">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "firstName" type = "xs:string" />
            <xs:element name = "middleInitial" type = "xs:string" />
            <xs:element name = "lastName" type = "xs:string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>
</xs:schema>
```

If you try validating this schema online, then you should see something like this:



Using the downloaded version of XSV, you can check that this is a valid schema by simply running it from the command line with a `-i` flag:

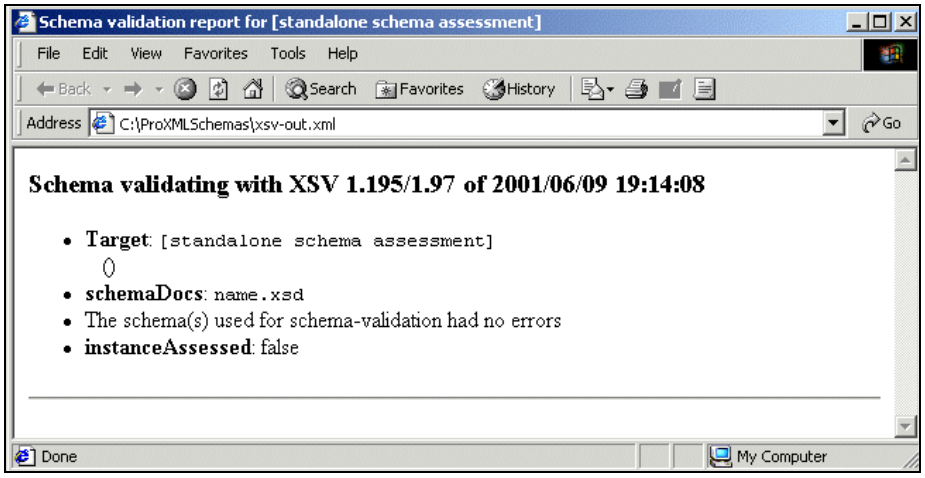


(Note that you'll need to have the folder in which XSV is installed included in your `PATH` variable). The output here isn't immediately obvious, so let's take a quick look at it (see the screenshot below). You can see that we are looking at a schema file here rather than an XML instance document since it says `instanceAccessed='false'`, and that the target is `[standalone schema assessment]`. Note that no schema errors are listed.

If you are running IE5 or above, you get a more user-friendly version of this and you can redirect the XML output to another file, including a stylesheet for display, with the command:

```
> xsv -o xsv-out.xml -s xsv.msxsl -i name.xsd
```

If you have MSXML 3 installed, you should replace `xsv.msxsl` with the XSLT 1.0 compliant version of the stylesheet, `xsv.xsl`. You can then view the result in your browser:

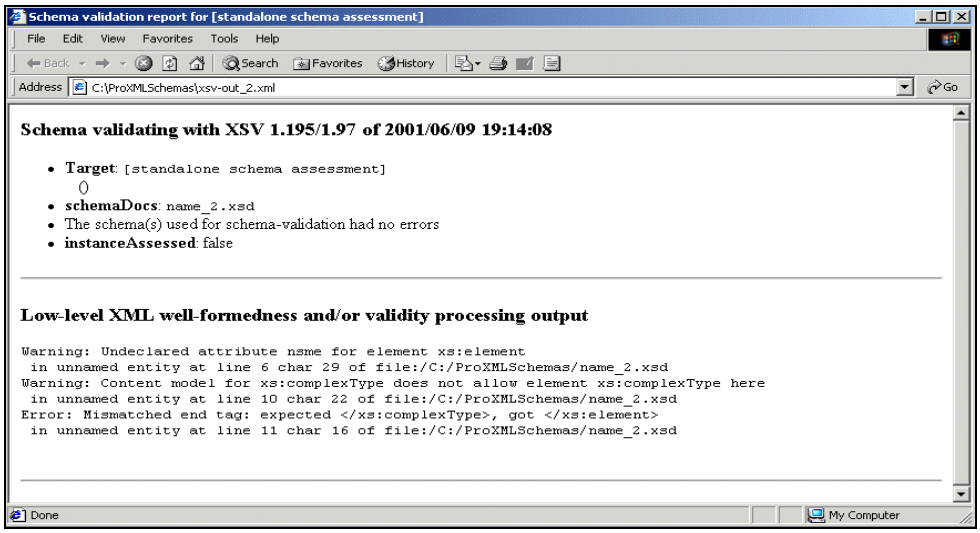


Note that you can use `xsv -?` for information on all the possible flags.

So that covers the basic ways of using XSV. Now let's take a look at some of the error messages that occur if our schema *isn't* error free. Suppose, for example, we make a simple typographical mistake, such as spelling the name attribute wrongly, or forgetting to close one of the elements:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "Name">
      <xs:complexType>
         <xs:sequence>
            <xs:element nsme = "firstName" type = "xs:string" />
            <xs:element name = "middleInitial" type = "xs:string" />
            <xs:element name = "lastName" type = "xs:string" />
         </xs:sequence>
      <xs:complexType>
   </xs:element>
</xs:schema>
```

In this case, XSV warns us that we have an undeclared attribute `nsme`, on our `element` element, and that we have a `complexType` declaration out of place:



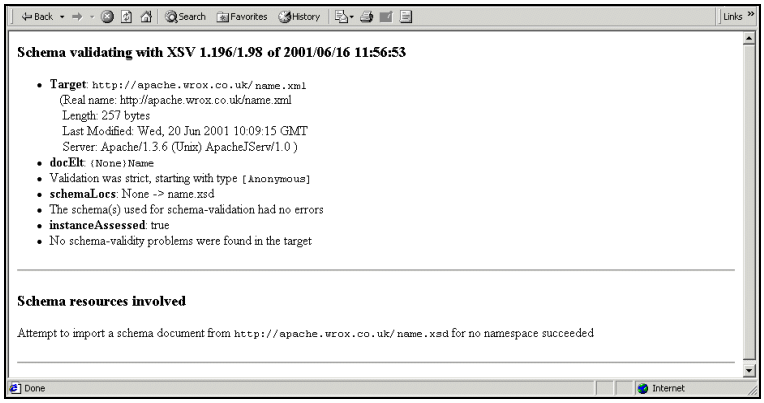
The reason the second error message takes this form is that XSV thinks that because we have forgotten to add a `/` in our closing tag, we are trying to nest a second `complexType` element inside the first, which is not allowed. Note that XSV also gives us the line number of each error. While the mistakes may be quite obvious in our simple schema, this information becomes very helpful when working with more complex examples.

Validating an Instance Document

Now let's try validating an instance document against our simple schema:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Name xmlns:xsi=" http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="name.xsd">
   <firstName>John</firstName>
   <middleInitial>J</middleInitial>
   <lastName>Johnson</lastName>
</Name>
```

Here, we have used the `xsi:noNamespaceSchemaLocation` attribute to indicate the location of the schema document to which the XML instance document conforms. In this case, it is in the same directory. Note that if you're validating this with the online version of XSV, both the XML file and the schema file need to be accessible over the web. Here's what the results look like for this file, `name.xml`:



The things to look out for here are the statement that there are no schema-validity problems in the target, and that the "Validation was strict": this means that the instance document has correctly validated against the schema. If you see the validation described as "lax", then you'll know that your document has not been validated, though it may be well formed. Note also the line at the bottom of the output, "Attempt to import a schema document from http://apache.wrox.co.uk/name.xsd for no namespace succeeded". This means that XSV has successfully found and loaded the correct schema document.

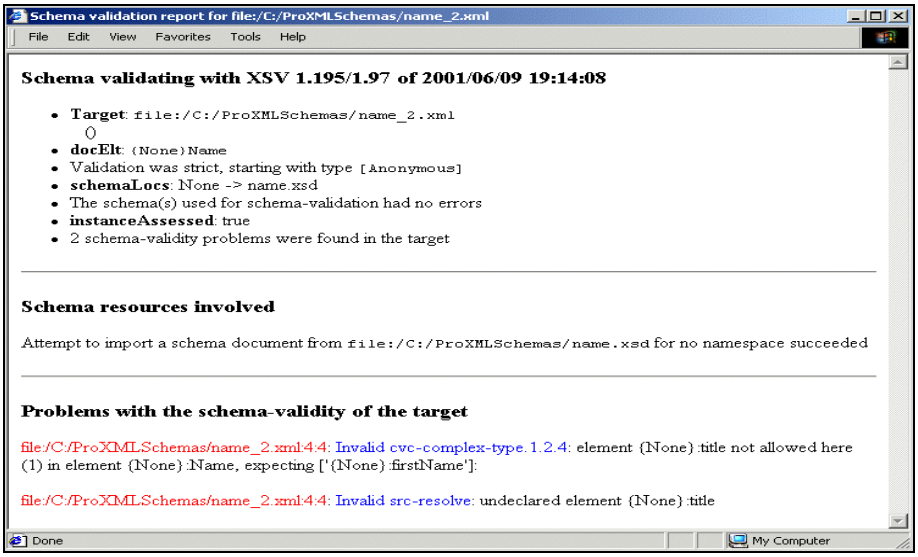
Let's take a look at an instance document with some problems, so you can see how XSV reports errors in document validation. Here, we've simply slipped in an extra `title` element that is not declared in our schema:

```
<?xml version = "1.0" encoding = "UTF-8"?>
<Name xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="name.xsd">
   <title>Dr</title>
   <firstName>John</firstName>
   <middleInitial></middleInitial>
   <lastName>Johnson</lastName>
</Name>
```

Let's try this one with our local version of XSV. In this case, we don't use the `-i` flag, as we are validating an instance document, not a schema, so we use the command:

> xsv -o xsv-out.xml -s xsv.msxsl name_2.xml

And this is what the output looks like:



In the first part of the output, we see the line, "2 schema-validity problems were found in the target". If you look at the section below, where the problems are listed in detail, you can clearly see that there is a `title` element that is not allowed according to the schema, and XSV was expecting the `firstName` element to appear in its place. The first number after the file name (in this case 4) indicates the line number on which the error occurred. Again, this information can be very useful when debugging schemas.

The output prefixes all of the element names with `{None}` to indicate that these elements are not part of a namespace. We'll be seeing how to create schemas with a target namespace in Chapter 6.

In the final part of this chapter, we'll be tying together the ideas that we have met so far in a slightly more complex example.

Example Schema: Delivery Receipt

In this example, we'll see how to create a schema for a delivery receipt called `DeliveryReceipt.xsd`. The schema contains constructs for names, addresses, and delivery items.

The delivery receipt is held within a root element called `DeliveryReceipt`, which has two attributes, `deliveryID` and `dateReceived`. The customer's name and address are then held within an element called `Customer`. Finally, the delivered items will be held within an `Items` element.

Here is a sample document marked up according to the `DeliveryReceipt.xsd` schema called `DeliveryReceipt.xml`:

```
<?xml version = "1.0" ?>
<DeliveryReceipt deliveryID = "44215" dateReceived = "2001-04-16"
  xsi:noNamespaceSchemaLocation = "http://file_Location/DeliveryReceipt.xsd"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
   <Customer>
      <Name>
         <FirstName>Ray</FirstName>
         <MiddleInitial>G</MiddleInitial>
         <LastName>Bayliss</LastName>
      </Name>
      <Address>
         <AddressLine1>10 Elizabeth Place</AddressLine1>
         <AddressLine2></AddressLine2>
         <Town>Paddington</Town>
         <City>Sydney</City>
         <StateProvinceCounty>NSW</StateProvinceCounty>
         <ZipPostCode>2021</ZipPostCode>
      </Address>
   </Customer>
   <Items>
      <DeliveryItem quantity = "2">
         <Description>Small Boxes</Description>
      </DeliveryItem>
   </Items>
</DeliveryReceipt>
```

Note how we indicate to a parser that it will be able to find a schema to validate the document using the `xsi:noNamespaceSchemaLocation` attribute in the root element. We use this because the constructs in the schema do not belong to a namespace. In order to use this attribute, we also need to declare the namespace for the XML Schema for instance documents:

```
<DeliveryReceipt deliveryID = "44215" dateReceived = "2001-04-16"
  xsi:noNamespaceSchemaLocation = "http://file_Location/DeliveryReceipt.xsd"
  xmlns:xsi = "http://www.w3.org/2001/XMLSchema-instance">
```

Now let's see the schema that we use for our Delivery Receipt documents. The schema is called `DeliveryReceipt.xsd`:

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs = "http://www.w3.org/2001/XMLSchema">
   <xs:element name = "DeliveryReceipt">
      <xs:complexType>
         <xs:sequence>

            <xs:element name = "Customer">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref = "Name" />
                     <xs:element ref = "Address" />
                  </xs:sequence>
               </xs:complexType>
            </xs:element>

            <xs:element name = "Items">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref = "DeliveryItem"
                                 minOccurs = "1" maxOccurs = "unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>

         </xs:sequence>
         <xs:attribute name = "deliveryID" type = "xs:integer" />
         <xs:attribute name = "dateReceived" type = "xs:date" />
      </xs:complexType>
   </xs:element>

   <xs:element name = "Name">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "FirstName" type = "xs:string" />
            <xs:element name = "MiddleInitial" type = "xs:string"
                                 minOccurs = "0" maxOccurs = "1" />
            <xs:element name = "LastName" type = "xs:string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name = "Address">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "AddressLine1"  type = "xs:string" />
            <xs:element name = "AddressLine2" type = "xs:string"
                        minOccurs = "0" maxOccurs = "1" />
            <xs:element name = "Town" type = "xs:string" />
```

```
            <xs:element name = "City" type = "xs:string"
                        minOccurs = "0" maxOccurs = "1" />
            <xs:element name = "StateProvinceCounty" type = "xs:string" />
            <xs:element name = "ZipPostCode"  type = "xs:string" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name = "DeliveryItem">
      <xs:complexType>
         <xs:sequence>
            <xs:element name = "Description"  type = "xs:string" />
         </xs:sequence>
         <xs:attribute name = "quantity" type = "xs:integer" />
      </xs:complexType>
   </xs:element>

</xs:schema>
```

There are a few things we should note about this schema:

- We have defined the `Name`, `Address`, and `DeliveryItem` elements globally, which also means that this schema could be used to validate documents only containing these elements
- We build the `Customer` element's content model using references to the globally declared `Name` and `Address` elements

Let's take a closer look at the schema. We start off declaring the namespace for XML Schema, which we use to prefix all of the elements defined by the XML Schema Recommendation:

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

We then define the root element `DeliveryReceipt`. Because it contains `Customer` and `Items` element elements (as opposed to being a text-only element), we have had to associate it with complex type using the `complexType` element. This also contains a `sequence` compositor, requiring that the `Customer` element appear before the `Items` element.

Between the closing `sequence` and `complexType` elements, we declare the two attributes that are carried by the `DeliveryReceipt` element: `deliveryID`, whose type is an `integer`, and `dateReceived`, whose type is a `date` type:

```
   <xs:element name = "DeliveryReceipt">
      <xs:complexType>
         <xs:sequence>

            <xs:element name = "Customer">
               ...
            </xs:element>

            <xs:element name = "Items">
               ...
            </xs:element>
```

```
            <xs:attribute name = "deliveryID" type = "xs:integer" />
            <xs:attribute name = "dateReceived" type = "xs:date" />
         </xs:complexType>
      </xs:element>
```

Inside the declaration of the `DeliveryReceipt` element we have a declaration of the `Customer` and `Items` elements. Both `Customer` and `Items` contain child elements, so we need to use a `complexType` element inside each of them, along with a compositor, which is the `sequence` element, to indicate the order in which they can appear. `Customer` and `Items` are made up of references to globally declared elements using the `ref` attribute:

```
            <xs:element name = "Customer">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref = "Name" />
                     <xs:element ref = "Address" />
                  </xs:sequence>
               </xs:complexType>
            </xs:element>

            <xs:element name = "Items">
               <xs:complexType>
                  <xs:sequence>
                     <xs:element ref = "DeliveryItem"
                                 minOccurs = "1" maxOccurs = "unbounded"/>
                  </xs:sequence>
               </xs:complexType>
            </xs:element>
```

We have already seen how we defined the `Name` and `Address` elements earlier in the chapter. The third globally declared element is the `DeliveryItem` element, which can occur one or more times. Note that we had to declare the occurrence constraints on the reference to the element, however, because you cannot add them to global declarations.

The `DeliveryItem` element also holds a `quantity` element, which is declared between the closing `sequence` and `complexType` elements. The `quantity` attribute has a type of `integer`:

```
      <xs:element name = "DeliveryItem">
         <xs:complexType>
            <xs:sequence>
               <xs:element name = "Description"  type = "xs:string" />
            </xs:sequence>
            <xs:attribute name = "quantity" type = "xs:integer" />
         </xs:complexType>
      </xs:element>
```

We specify the simple built-in string datatype on the `Description` element to restrict the allowable content of text-only elements; if we not did associate them with a type they could hold any well-formed combination of elements, attributes and characters that we had defined in the schema.

Summary

In this chapter we have looked at the basics of the W3C XML Schema syntax, and how we can declare which elements and attributes are allowed to appear in our XML documents. We have seen that in order to declare an element or an attribute, we must associate its name with a type, and how XML Schema introduces two categories of types:

- Simple types: which restrict text-only element content and attribute values
- Complex types: which are required to indicate when an element contains child elements and carries attributes

We have briefly touched on some of the other features that make XML Schema such a powerful language:

- The built-in types such as string, date and integer, which will make integration of XML with applications and data sources a lot easier
- The annotation mechanism for commenting and passing information to processing applications

We also alluded to some of the more complicated features we will be seeing in coming chapters, such as the use of namespaces, named complex types and different compositors.

Having addressed the basics of the element and attribute declarations and the differences between simple and complex types in XML Schema, you can go on to look at the built-in types in more depth in the next chapter. In Chapter 3, we'll move on to see how we can build more complicated structures.