

**SYBEX Sample Chapter**

# XML Schemas

Chelsea Valentine; Lucinda Dykes; Ed Tittel

## Chapter 5: Understanding XML Schema

Copyright © 2001 SYBEX Inc., 1151 Marina Village Parkway, Alameda, CA 94501. World rights reserved. No part of this publication may be stored in a retrieval system, transmitted, or reproduced in any way, including but not limited to photocopy, photograph, magnetic or other record, without the prior agreement and written permission of the publisher.

ISBN: 0-7821-4045-9

SYBEX and the SYBEX logo are either registered trademarks or trademarks of SYBEX Inc. in the USA and other countries.

**TRADEMARKS:** Sybex has attempted throughout this book to distinguish proprietary trademarks from descriptive terms by following the capitalization style used by the manufacturer. Copyrights and trademarks of all products and services listed or described herein are property of their respective owners and companies. All rules and laws pertaining to said copyrights and trademarks are inferred.

This document may contain images, text, trademarks, logos, and/or other material owned by third parties. All rights reserved. Such material may not be copied, distributed, transmitted, or stored without the express, prior, written consent of the owner.

The author and publisher have made their best efforts to prepare this book, and the content is based upon final release software whenever possible. Portions of the manuscript may be based upon pre-release versions supplied by software manufacturers. The author and the publisher make no representation or warranties of any kind with regard to the completeness or accuracy of the contents herein and accept no liability of any kind including but not limited to performance, merchantability, fitness for any particular purpose, or any losses or damages of any kind caused or alleged to be caused directly or indirectly from this book.

Sybex Inc.  
1151 Marina Village Parkway  
Alameda, CA 94501  
U.S.A.  
Phone: 510-523-8233  
[www.sybex.com](http://www.sybex.com)

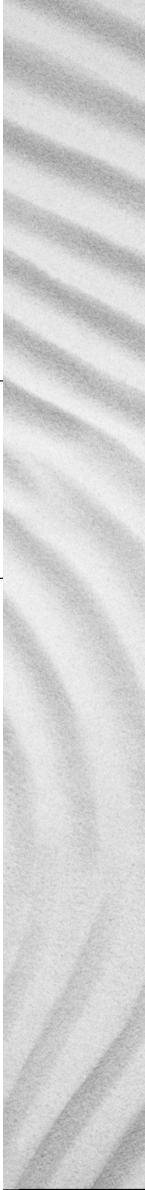
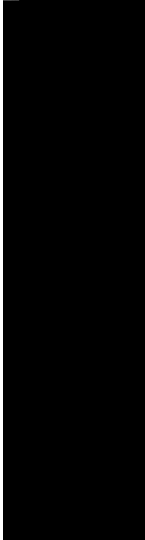


## CHAPTER 5

---

# Understanding XML Schema

---

- Introducing XML Schema terms and concepts
  - Understanding XML Schema components
  - Introducing XML Schema datatypes
  - Understanding how XML namespaces fit in
  - Introducing advanced XML Schema concepts
- 
- 

It took a while for XML Schema to make it to the W3C Recommendation phase—the proposal was under development for over two years. With its release, however, came an essential component meant to help XML reach its full potential. With XML Schema, users can describe a document’s structure as well as define datatypes for element and attribute content. Datatyping brings to XML document modeling an added functionality that increases a developer’s ability to define electronic commerce systems, and it helps those using databases or needing to manipulate large volumes of data on the Web. In addition to datatypes, XML Schema is namespace aware and allows document authors to take advantage of namespace functionality.

The XML Schema language is defined by three separate specification documents. The first, XML Schema Part 1: Structures (the Structures document), defines constructs for describing the structure of XML documents. The second, XML Schema Part 2: Datatypes (the Datatypes document), defines a set of simple datatypes. And finally, the third, XML Schema Part 0: Primer, provides developers with a primer that explains what schemas are and how to build them.

This chapter is a starting point for understanding XML Schema. It’s organized as an introduction to how the W3C’s XML Schema language defines the mechanisms that allow developers to define document models and datatypes.

## XML Schema Terms and Concepts

The term *schema* is commonly used in the database community and refers to the organization or structure for a database. When this term is used in the XML community, it refers to the structure (or model) of a class of documents. This model describes the hierarchy of elements and allowable content in a valid XML document. In other words, the schema defines constraints for an XML vocabulary. XML Schema is a formal definition for defining a schema for a class of XML documents.

The sheer volume of text involved in defining the XML Schema language can be overwhelming to an XML novice, or even to someone making the move from Document Type Definitions (DTDs) to XML Schema. To completely understand XML Schema, you have to be familiar with at least two specification documents (XML Schema Part 1: Structures and XML Schema Part 2: Datatypes), if not more. The following section is dedicated to introducing you to basic schema constructs and concepts.

## Purpose and Requirements

XML is a self-describing language that provides developers with flexibility. However, too much flexibility can sometimes be detrimental for XML applications. In some cases, free-form

XML can be used (an XML document without a defined document model); however, defining strict document models that your XML documents must conform to can make them more useful. When you're designing XML applications, there are instances where validation is a must. For example, schema validation is helpful for organizing data and documents. You'll also find schemas necessary when supporting data interchange between different platform systems and services. These are among many scenarios that served as the primary motivation for a new validation model.

There were several application needs that developers wanted met but were not possible using DTDs. For example, developers wanted to be able to incorporate namespaces into their document models that would then provide better assurance in electronic commerce (e-commerce) transactions and greater security against unauthorized changes to validation rules. In addition, many developers needed the ability to define strict datatypes for attribute and element content to ensure data integrity before attempting to exchange that data.

As defined by the XML Schema Requirements document, the purpose of XML Schema is to provide an inventory of XML markup constructs with which to write schemas. XML Schema is expected to define a document model for a class of XML documents. XML Schema document models are designed to define the usage and relationships of various schema components, such as the following:

- Datatypes
- Elements and their content
- Attributes and their values
- Reusable components and their content
- Notations

In an effort to define a document model, the XML Schema Working Group was asked to address the following issues:

**Structural schemas** The working group was expected to describe constructs for defining document models. These constructs are similar to those defined by DTDs in XML 1.0. The working group was expected to move beyond basic DTDs and define greater functionality, such as integration with namespaces and integration of primitive datatypes. These constraints are defined by XML Schema Part 1: Structures.

**Primitive datatypeing** This working group focused on allowing datatype definitions. This would allow developers to define constraints for integers, dates, and the like. These datatypes would be selected and defined based on experiences with Structured Query Language (SQL) and Java primitives. The constraints are defined by XML Schema Part 2: Datatypes.

**Conformance** This working group was expected to define the relationship between the schema and the XML document instances. In addition, it was expected to define obligations for schema-aware processors. These constraints are defined by XML Schema Part I: Structures and XML Schema Part 2: Datatypes.

In the following subsections, we take a look at the design principles, usage scenarios, and requirements defined by the XML Schema Working Group in an effort to guide the creation of XML Schema.

### Usage Scenarios

There are many reasons document authors are turning to XML Schema as their modeling language of choice. If you have a schema model, you can ensure that a document author follows it. This is important if you're defining an e-commerce application and you need to make sure that you receive exactly what you expect—nothing more and nothing less—when exchanging data. The schema model will also ensure that datatypes are followed, such as rounding all prices to the second decimal place, for example.

Another common usage for XML Schema is to ensure that your XML data follows the document model before the data is sent to a transformation tool. For example, you may need to exchange data with your parent company, and because your parent company uses a legacy document model, your company uses different labeling (`bookPrice` versus `price`). In this case, you would need to transform your data so it conforms to the parent company's document model. However, before sending your XML data to be transformed, you want to be sure that it's valid because one error could throw off the transformation process.

Another possible scenario is that you're asked to maintain a large collection of XML documents and then apply a style sheet to them to define the overall presentation (for example, for a CD-ROM or Web site). In this case, you need to make sure that each document follows the same document model. If one document uses a `para` instead of a `p` element (the latter of which the style sheet expects), the desired style may not be applied.

These are only a few scenarios that require the use of XML Schema (or a schema alternative). There are countless other scenarios that would warrant their use. The XML Schema Working Group carefully outlined several usage scenarios that it wanted to account for while designing XML Schema. As defined by the XML Schema Requirements document, the following usage scenarios were used to help shape and develop XML Schema:

- Publishing and syndication
- E-commerce transaction processing
- Supervisory control and data acquisition
- Traditional document authoring/editing governed by schema constraints

- Using schema to help query formulation and optimization
- Open and uniform transfer of data between applications, including databases
- Metadata interchange

### Design Principles

The design principles outlined by the XML Schema Requirements document are fairly straightforward. XML Schema documents should be created so they are as follows:

- More expressive than XML DTDs
- Expressed in XML
- Self-describing
- Usable in a wide variety of applications that employ XML
- Straightforwardly usable on the Internet
- Optimized for interoperability
- Simple enough to implement with modest design and runtime resources
- Coordinated with relevant W3C specs, such as XML Information Set, XML Linking Language (XLink), Namespaces in XML, Document Object Model (DOM), the Hyper-text Markup Language (HTML), and the Resource Description Framework (RDF) Schema.

### Requirements

Before the participants of the XML Schema Working Group sat down to define XML Schema, they identified some key structural and datatype requirements. These requirements are defined in the following lists.

XML Schema has the following structural requirements (as stated by the W3C at [www.w3.org/TR/NOTE-xml-schema-req#Structural](http://www.w3.org/TR/NOTE-xml-schema-req#Structural)):

- Mechanisms for constraining document structure (namespaces, elements, and attributes) and content (datatypes, entities, and notations)
- Mechanisms to enable inheritance for element, attribute, and datatype definitions
- Mechanisms for URI reference to standard semantic understanding of a construct
- Mechanisms for embedded documentation
- Mechanisms for application-specific constraints and descriptions
- Mechanisms for addressing the evolution of schemata
- Mechanisms to enable integration of structural schemas with primitive datatypes

The datatype requirements as stated by the W3C at [www.w3.org/TR/NOTE-xml-schema-req#Datatype](http://www.w3.org/TR/NOTE-xml-schema-req#Datatype) are as follows:

- Provide for primitive datatyping, including byte, date, integer, sequence, and SQL and Java primitive datatypes.
- Define a type system that is adequate for import/export from database systems.
- Distinguish requirements relating to lexical data representation vs. those governing an underlying information set.
- Allow creation of user-derived datatypes, such as datatypes that are derived from an existing datatype and that may constrain some of its properties.

Here are the conformance requirements as stated by the W3C at [www.w3.org/TR/NOTE-xml-schema-req#Conformance](http://www.w3.org/TR/NOTE-xml-schema-req#Conformance):

- Describe the responsibilities of conforming processors.
- Define the relationship between schemas and XML documents.
- Define the relationship between schema validity and XML validity.
- Define the relationship between schemas and XML DTDs and their information sets.
- Define the relationship among schemas, namespaces, and validity.
- Define a useful XML schema for XML schemas.

## Basic Terminology

XML terminology is thrown around, sometimes recklessly, within the XML community. Understanding this terminology will help you understand conversations about XML a little more. Before we introduce XML Schema, you need to be familiar with a few terms defined by XML 1.0 (Second Edition) and the XML Schema languages.

### Well Formed

A document is considered to be *well formed* if it meets all the well-formedness constraints defined by XML 1.0 (Second Edition). These constraints are as follows:

- The document contains one or more elements.
- The document consists of exactly one root element (also known as the document element).
- The name of an element's end tag matches the name defined in the start tag.
- No attribute may appear more than once within an element.
- Attribute values cannot contain a left-angle bracket (<).
- Elements delimited with start and end tags must nest properly within each other.

**WARNING**

We have listed only those well-formedness constraints required for a free-form XML document (one without a DTD). There are approximately 10 other well-formedness constraints that define requirements for internal and external DTD subsets. Because we're not concerned with DTDs, we did not define these constraints.

**Validity**

According to the strict definition of *validity*, as defined by XML 1.0 (Second Edition), an XML document is valid if it has an associated document type (DOCTYPE) declaration and if the document complies with the constraints expressed in it. On a more general level, to say that an XML document is valid is to assume that it adheres to a defined model for a class of XML documents. In terms of XML Schema, *validity* means that an XML document conforms to the constraints expressed by the associated XML Schema documents.

**TIP**

A valid document does not ensure semantic perfection. Although XML Schema defines stricter constraints on element and attribute content than XML DTDs do, it cannot catch all errors. For example, you might define a price datatype that requires two decimal places. However, you might enter 1200.00 when you meant to enter 12.00 and the schema document wouldn't catch the error.

When dealing with validity, you need to keep in mind that there are three ways an XML document can exist:

- As a free-form, well-formed XML document that does not have DTD or schema associated to it
- As a well-formed and valid XML document, adhering to a DTD or schema
- As a well-formed document that is not valid because it does not conform to the constraints defined by the associated DTD or schema

XML Schema validity encompasses two separate but important functions:

**Content model validity** Ensures that the element hierarchy and document structure are correct. Checks to make sure that elements are ordered and nested correctly.

**Datatype validity** Ensures that element and attribute content adheres to the defined datatype. A datatype can define a scope for legal values as well as define a base type such as integer, decimal, or string.



## Datatype

A broad definition of a *datatype* is a set of data with values that have predefined characteristics. Datotyping provides you with a way of assigning how the value associated with an element or attribute is to be represented, interpreted, or understood. Most datatypes devolve to a collection of so-called *data primitives*, which represent basic types to associate with values, including well-understood primitives such as integer, real number, character, string, and so forth.

According to the Datatypes specification, a datatype consists of a set of distinct values (value space), a set of lexical representations (lexical space), and a set of facets that characterize properties of the value space—individual values or lexical items. Datatypes are defined in greater detail later in this chapter.

## Schema Component

A schema is said to be made up of *schema components*, which are the building blocks that make up the abstract data model of the schema. Element and attribute declarations, complex types, simple types, and notations are all examples of schema components.

## XML Schema Structures

The XML Schema vocabulary is defined by XML Schema Part 1: Structures. The Structures document is burdened with defining the very nature of XML Schema and all of its components. In addition, the Structures document defines the vocabulary (elements and attributes) that can be used to define schema components that in turn make up the schema. In a nutshell, the Structures specification outlines constructs for defining, describing, and cataloging XML vocabularies for classes of XML documents.

This section is dedicated to introducing you to the Structures specification as well as the concepts defined by it. One of the best ways to understand the XML Schema language is to take a look at it. Therefore, the first thing we do is provide you with a brief example of a simple XML Schema document in Listing 5.1. Then in Listing 5.2, we show you an XML document instance that conforms to the schema defined in Listing 5.1.



### Listing 5.1

### XML Schema Document

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:element name="publications">
    <xsd:complexType>
```

```

<xsd:sequence>
  <xsd:element name="book" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="author" type="xsd:string"/>
        <xsd:element name="description" type="xsd:string"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:sequence>
</xsd:complexType>
</xsd:element>
</xsd:schema>

```



## Listing 5.2 XML Document Instance

```

<?xml version="1.0"?>
<publications>
  <book>
    <title>Mastering XHTML</title>
    <author>Ed Tittel</author>
    <description>Newly revised and updated, Mastering XHTML
      is a complete guide to the markup language that is
      leading the world of Web development from HTML to
      XML.
    </description>
  </book>
  <book>
    <title>Java Developer's Guide to E-Commerce with
      XML and JSP</title>
    <author>William Brogden</author>
    <description>Covering the latest Servlet and JSP APIs
      and the current XML standard, this book guides you through
      all the steps required to build and implement a cohesive,
      dynamic, and profitable site.
    </description>
  </book>
</publications>

```

Before trying to understand each line of Listing 5.1, try to pick out constructs that make sense to you, such as one of the following two lines, for example:

```

<xsd:element name="publications">
  <xsd:sequence>

```

Semantically these elements make sense. In Listing 5.1, `element` uses the `name` attribute to define an element name (`publications`). Then, the `sequence` element is a compositor that

tells the processor that the child elements nested with the `sequence` element must occur in that order when used as a part of an XML document instance. There's much more to our example, but from the beginning, you should have some idea of what is going on in the schema document.

As we dive deeper into XML Schema, you'll have to be aware of a few distinctions about commonly used schema components. These distinctions are described in the following sections.

### Definitions vs. Declarations

A *definition* describes a complex or simple type that either contains element or attribute declarations or references element or attribute declarations defined elsewhere in the document. A *declaration* defines an element or attribute, specifying the name and datatype for the component.

Here's a definition:

```
<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="description" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

---

**TIP**

Type definitions can define complex or simple types. In this case, we provide an example of a complex type definition.

Here's a declaration:

```
<xsd:element name="book">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title" type="xsd:string"/>
      <xsd:element name="author" type="xsd:string"/>
      <xsd:element name="description" type="xsd:string"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>
```

### Simple Type vs. Complex Type

A *simple type definition* constrains the text that is allowed to appear as content for an attribute value or text-only element without attributes. A *complex type definition* constrains the allowable content of elements. Both types govern possible attribute and child elements.

Here's a simple type definition:

```
<xsd:simpleType name="isbnType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{10}" />
  </xsd:restriction>
</xsd:simpleType>
```

Here's a complex type definition:

```
<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element name="title" type="xsd:string"/>
    <xsd:element name="author" type="xsd:string"/>
    <xsd:element name="description" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```

## Global vs. Local

Declarations and definitions can be declared globally or locally. A *globally defined component* is defined as a child of the schema element. A *locally defined component* (also known as *inline*) is defined as a child of another schema component.

A globally defined component would look like this:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:element name="publications">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="book" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>

  <xsd:element name="book" maxOccurs="unbounded">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="title"/>
        <xsd:element ref="author"/>
        <xsd:element ref="description"/>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
```

```
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="author" type="xsd:string"/>
<xsd:element name="description" type="xsd:string"/>

</xsd:schema>
```

Here's an example of a locally defined component:

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:element name="publications">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="book" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
              <xsd:element name="author" type="xsd:string"/>
              <xsd:element name="description" type="xsd:string"/>
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

## Schema Components

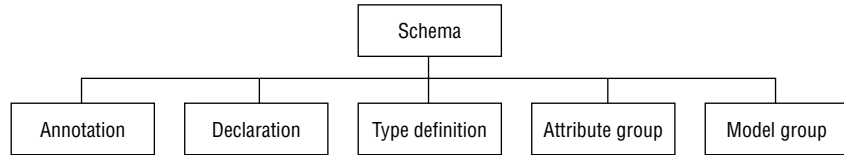
The Structures document defines the various schema components that make up a schema document. These components, such as type definitions and element declarations, make up the heart of the schema document. At the top level of any XML Schema document, a document author may define one of the five following basic schema components:

- Annotation
- Type definition
- Declaration
- Attribute group
- Model group

Figure 5.1 illustrates this relationship.

**FIGURE 5.1:**

Top level of an XML  
Schema structure  
hierarchy

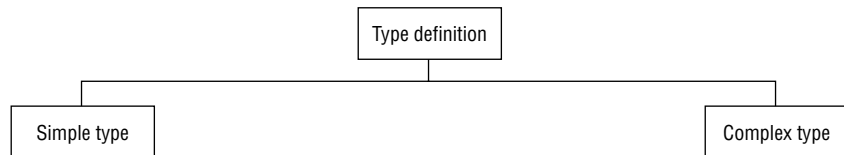


## Type Definitions

Type definitions are at the heart of many schema documents. There are two different types of definitions in addition to two different ways each type definition can be defined. We'll start with the two different types of definitions: simple and complex types (see Figure 5.2). Simple type definitions are used to define attributes and elements that can contain only data. Complex type definitions define content models for elements that may consist of elements and attributes.

**FIGURE 5.2:**

Type definitions can  
be divided into simple  
or complex types.



You can define simple or complex types at the root level, in which case you must name them and reference them later (known as *named types*). Or, you can define them nested within other schema components (known as *anonymous types*). In this case, there's no need to reference them because they're defined as a part of the content model

## Simple Types

*Simple types* are used to define all attributes and elements that contain only text and do not have attributes associated with them. When defining simple types, you can use XML Schema datatypes to restrict the text contained by the element or attribute. For example, you can require an attribute value to be an integer or date. You can even restrict that integer to consist of only three numbers. In addition to specifying simple restrictions on length, you can define patterns and other properties for your datatypes. This is done by using derivation techniques defined by the Structures document.

Simple datatypes are defined by derivation from other datatypes, either predefined and identified by the W3C XML Schema namespace or defined elsewhere in your schema. We discuss this topic in more detail later in this chapter and in Chapter 6.

## Complex Types

An element is considered to be a *complex type* if it contains attribute or child elements. Complex types are used to define complex content models. For example, if you want to define a book element that contains `title` and `author` child elements, you would have to use a complex type definition. As with simple types, you can use derivation techniques to manipulate a complex type after it's defined. As the document author, you can restrict or extend a defined content model, or even substitute the model completely. We discuss this topic in more detail later in this chapter and in Chapter 6.

## Declarations

The Structures document defines three types of declaration components: element, attribute, and notation. These three declarations are similar to those defined by XML 1.0.

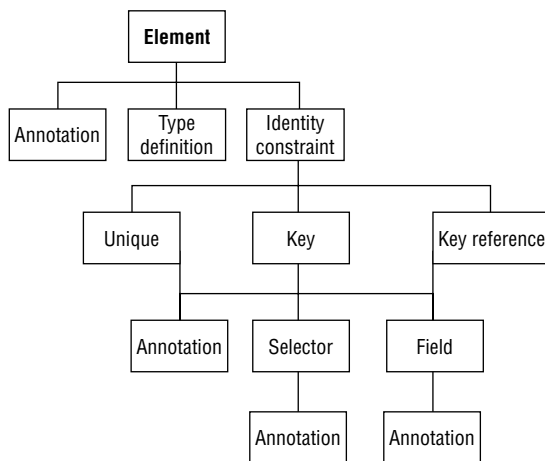
### Element Declarations

*Element declarations* define element names and types and can optionally define identity constraints. XML Schema allows document authors to define complex content models. However, it may take a little time to understand how to construct those models. Figure 5.3 illustrates allowable content within an element declaration.

---

**FIGURE 5.3:**

Element declaration  
structure hierarchy



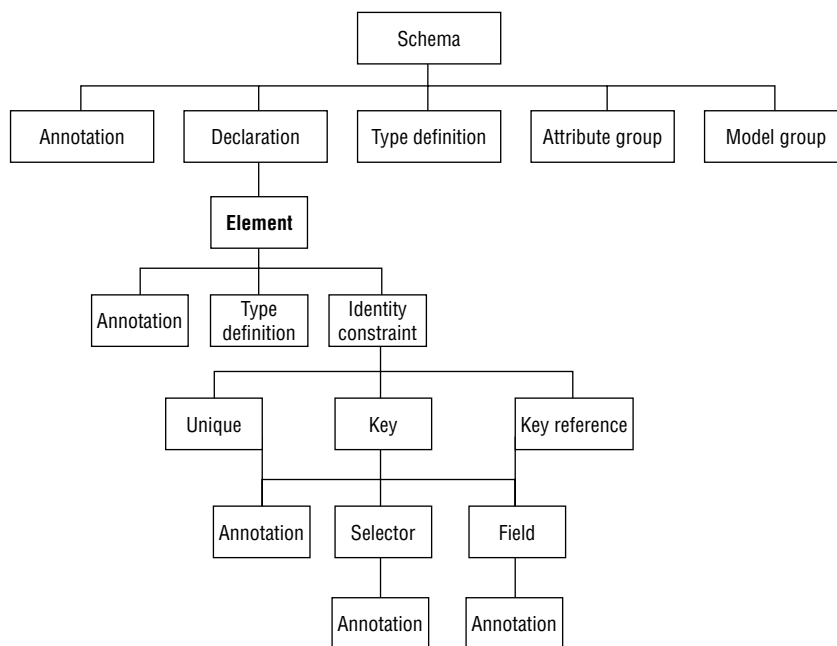
An element declaration may contain one of the following:

- An annotation to provide information for human or application consumption
- A type definition (simple or complex) to define allowable content and/or attributes
- An identity constraint to require uniqueness

There are a few ways you can declare an element:

- The element declaration can be defined globally at the top level of the schema document (see Figure 5.4).
- The element declaration can be defined locally within a complex type definition (see Figure 5.5). The complex type definition can be globally defined or defined within an element declaration.

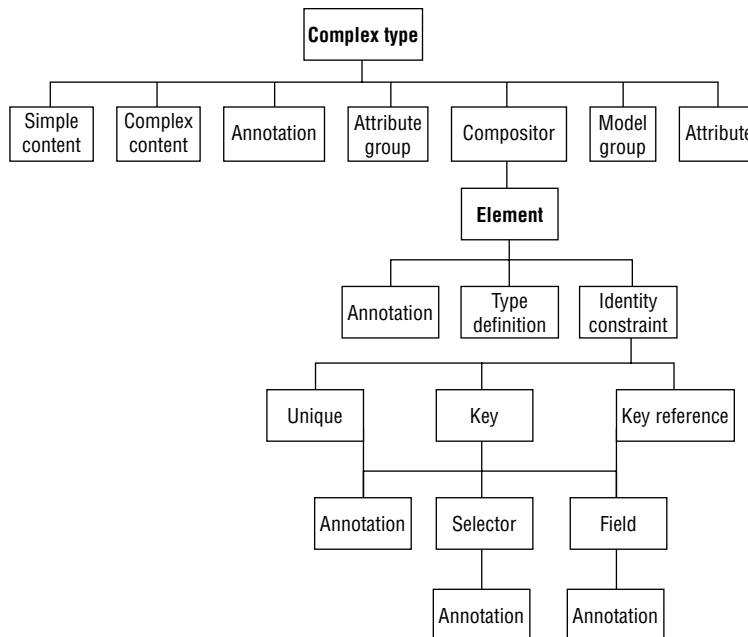
**FIGURE 5.4:**  
Declaration structure  
hierarchy





**FIGURE 5.5:**

Complex type  
definition structure  
hierarchy



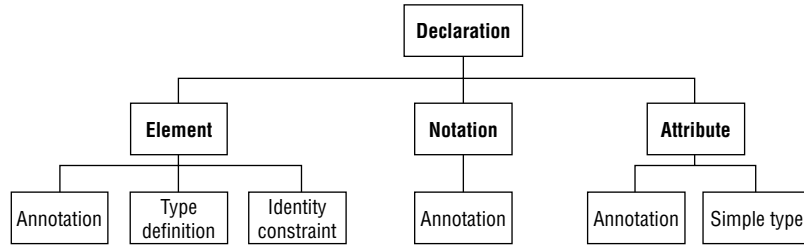
### Attribute Declarations

An *attribute declaration* defines the attribute name, the simple type definition that specifies the datatype for the value or a built-in datatype, occurrence information, and (optionally) a default value. Document authors have access to the 44 built-in XML Schema datatypes and can develop their own. There are several ways an attribute declaration can be defined:

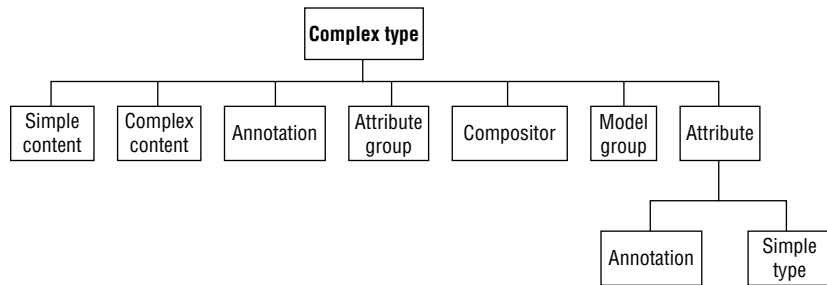
- Globally at the top level of the schema document as shown in Figure 5.6 (If declared globally, the declaration would have to be referenced from within a complex type definition.)
- Locally within a complex type definition that is defined globally as shown in Figure 5.7
- Locally within a complex type definition that is defined locally within an element declaration as shown in Figure 5.8

**FIGURE 5.6:**

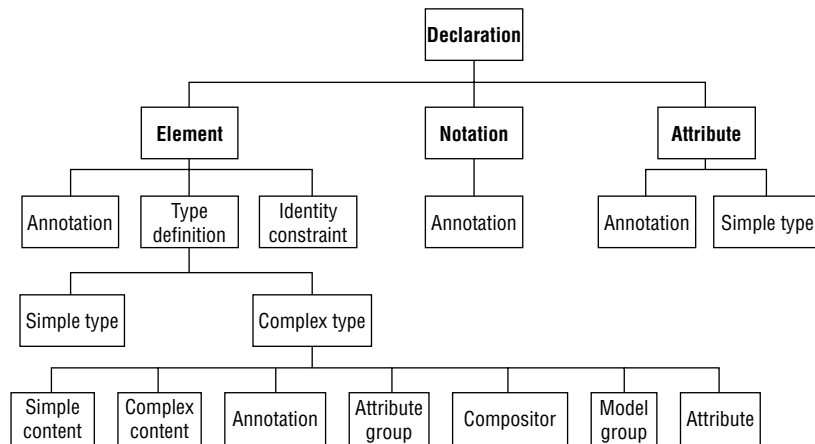
Declaration structure hierarchy

**FIGURE 5.7:**

Complex type definition structure hierarchy

**FIGURE 5.8:**

Element declaration structure hierarchy



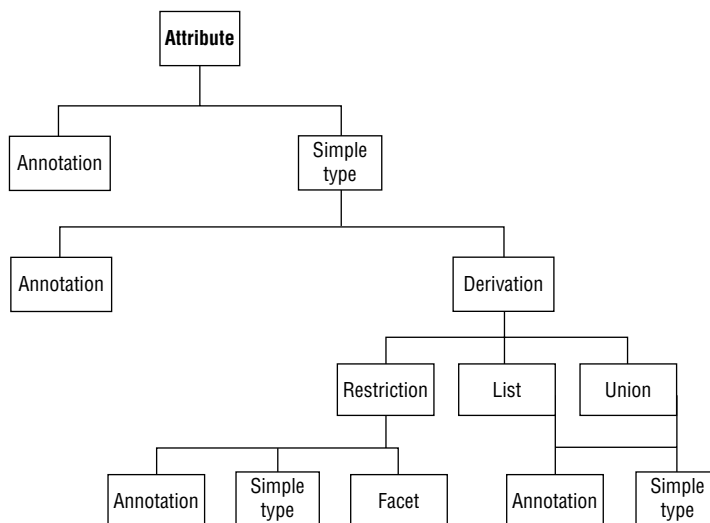
Attribute declarations contain one of the following:

- Annotation for human or application consumption.
- Simple type definition to define the datatype for the attribute. If a simple type is not defined locally, datatype information is defined using an attribute. That attribute value can define a built-in datatype or reference a simple type that is defined globally.

Figure 5.9 illustrates the allowable content for an attribute declaration.

**FIGURE 5.9:**

Attribute declaration  
structure hierarchy



### Notation Declarations

Similar to DTD notations, an XML notation component is used to declare links to external non-XML data and then associate an external application to handle the non-XML data. Examples of non-XML data that would need a notation declaration are image, audio, and sound files. The notation declaration is used in association with the NOTATION datatype.

The declaration uses attributes to identify the notation name, identifier, and external application. Notation declarations are empty elements and, therefore, cannot contain other schema components.

### Working with Declarations

When you first begin to work with XML Schema, you'll realize that there are several ways you can define declarations:

- You can define elements and attributes as you need them, nesting the declarations within other element declarations (see “Inline Declarations”).
- You can define declarations at the top level and reference them from within element declarations (see “Referencing Declarations”).
- You can define named type definitions that are then referenced within element or attribute declarations (see “Named Types”).

### Inline Declarations

Inline declarations are defined locally within other schema components. For example, you can nest an element declaration within a complex type definition, which can also be nested within element declarations. Using this recursive behavior is a common method for defining elements. Attribute declarations can also be defined locally within a complex type definition. Listing 5.3 provides an example of inline element and attribute declarations.



#### Listing 5.3 An XML Schema Using Inline (or Local) Declarations

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:element name="publications">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="book" maxOccurs="unbounded">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="title" type="xsd:string"/>
              <xsd:element name="author" type="xsd:string"/>
              <xsd:element name="description" type="xsd:string"/>
            </xsd:sequence>
            <xsd:attribute name="isbn" type="xsd:string"/>
          </xsd:complexType>
        </xsd:element>
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>
```

### Referencing Declarations

If you defined your declarations globally, you can then reference them from within other complex or simple type definitions. This modular approach is a popular way to handle complex schema documents. When you modularize your complex definitions, they're easier to maintain and manipulate using derivation and substitution techniques. Listing 5.4 shows an element referencing globally defined declarations.



#### Listing 5.4 An XML Schema Document Referencing Globally Defined Declarations

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">
```

```

<xsd:element name="publications">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="book" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="book" maxOccurs="unbounded">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element ref="title" />
      <xsd:element ref="author" />
      <xsd:element ref="description" />
    </xsd:sequence>
    <xsd:attribute name="isbn" type="xsd:string" />
  </xsd:complexType>
</xsd:element>

<xsd:element name="title" type="xsd:string" />
<xsd:element name="author" type="xsd:string" />
<xsd:element name="description" type="xsd:string" />

</xsd:schema>

```

---

## Named Types

The method of defining type definitions globally is also known as defining *named types* (because you name the type definition and then reference it later). Named types can be used like XML DTD parameter entities; you can define a content model or datatype and then reference that definition multiple times in the schema document. This approach is commonly used when working with complex datatypes or if you have content models that will be used more than once in the schema document. Listing 5.5 provides an example of named type definitions.



### Listing 5.5

### An XML Schema Document Using Named Type Definitions

```

<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:simpleType name="isbnType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{10}" />
    </xsd:restriction>
  </xsd:simpleType>

```

```
<xsd:complexType name="pubType">
  <xsd:sequence>
    <xsd:element ref="book"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element ref="title"/>
    <xsd:element ref="author"/>
    <xsd:element ref="description"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="publications" type="pubType"/>
<xsd:element name="book" type="bookType" maxOccurs="unbounded"/>
<xsd:element name="title" type="xsd:string"/>
<xsd:element name="author" type="xsd:string"/>
<xsd:element name="description" type="xsd:string"/>
</xsd:schema>
```

---

## Attributes and Grouping

An *attribute group definition* is an association between a name and a set of attribute declarations, enabling reuse of the same set in several complex type definitions. This functionality is similar to using XML DTD parameter entities. There are many instances in which several attributes will be used in multiple content models. For example, the XHTML vocabulary allows attributes such as `id`, `style`, and `title` to be used with most of its elements. Instead of redefining these declarations, or even referencing these declarations in multiple places, you can define them as a group. Any content model that uses all three of these attributes can reference that group instead of having to reference each attribute individually, as shown here:

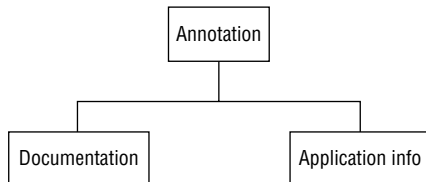
```
<xsd:attributeGroup name="commonAttributes">
  <xsd:attribute name="title" type="xsd:string"/>
  <xsd:attribute name="id" type="xsd:ID"/>
  <xsd:attribute name="style" type="xsd:string"/>
</xsd:attributeGroup>
```

## Annotations

*Annotations* are helpful schema components because they provide formal definitions for information intended for humans or applications. Annotations can be defined at the top level, under the schema element, as well as within almost every other schema component. The annotation component may contain only two different elements (see Figure 5.10).

**FIGURE 5.10:**

The annotation structure hierarchy



Although XML Schema defines these constructions, the specification does not define how this information should be interpreted. The following example contains an annotation meant for the document author, or any other two-legged creator who would need to use the schema document:

```

<xsd:element name="book">
  <xsd:annotation>
    <xsd:documentation xml:lang="en">
      Child elements content models defined using named
      complex types
    </xsd:documentation>
  </xsd:annotation>
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="title" type="titleType"/>
      <xsd:element name="description" type="descType"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

```

## An Overview of XML Schema Datatypes

XML 1.0 DTDs allow for limited facilities for applying datatypes to document content.

However, for many XML applications, a higher level of datatype checking is needed.

Listing 5.6 defines an XML document that contains several values that would require strictly defined datatypes.



### Listing 5.6

### A Schema Document That Uses Multiple Simple Type Definitions

```

<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:year"/>
  </xsd:simpleType>

```

```
<xsd:simpleType name="isbnType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{10}" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="editionType">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="10" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="ppType">
  <xsd:restriction base="xsd:nonNegativeInteger">
    <xsd:minInclusive value="1" />
    <xsd:maxInclusive value="2000" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:element name="author" type="xsd:string"/>

<xsd:complexType name="authorsType">
  <xsd:sequence>
    <xsd:element ref="author" maxOccurs="5" />
  </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element ref="title" />
    <xsd:element name="authors" type="authorsType" />
    <xsd:element name="pubDate" type="pubDateType" />
    <xsd:element ref="publisher" />
    <xsd:element name="size" type="sizeType" />
    <xsd:element name="topics" type="topicsType" />
    <xsd:element name="errata" type="errataType"
      minOccurs="0" />
    <xsd:element ref="description" />
    <xsd:element ref="website" minOccurs="0" />
  </xsd:sequence>
  <xsd:attribute name="isbn" type="isbnType"
    use="required" />
  <xsd:attribute name="edition" type="editionType"
    use="required" />
  <xsd:attribute name="cat" type="xsd:NMTOKENS"
    use="required" />
  <xsd:attribute name="id" type="xsd:ID" />
</xsd:complexType>
```



```
<xsd:element name="description" type="xsd:string"/>

<xsd:complexType name="errataType">
  <xsd:attribute name="code" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pubDateType">
  <xsd:attribute name="year" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:element name="publications">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="book" type="bookType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="publisher" type="xsd:string"/>

<xsd:complexType name="sizeType">
  <xsd:attribute name="pp" type="ppType" use="required"/>
</xsd:complexType>

<xsd:element name="title" type="xsd:string"/>
<xsd:element name="topic" type="xsd:string"/>

<xsd:complexType name="topicsType">
  <xsd:sequence>
    <xsd:element ref="topic" maxOccurs="unbounded"/>
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="website" type="xsd:string"/>
</xsd:schema>
```

---

In Listing 5.6 there are four defined datatypes for the following attributes:

**year** We use the built-in year datatype.

**isbn** We derived our own datatype that defines a pattern for the ISBN.

**edition** We restrict our datatype to only allow for an integer from 1 to 10.

**pp** We restrict our datatype to only allow for an integer from 1 to 2000.

If we were to use XML 1.0 DTDs, we would not be able to express the necessary validity constraints for the content in attributes such as *isbn*, *edition*, and *pp*. However, XML Schema Part 2: Datatypes provides document authors with a robust, extensible datatype system for XML.

This datatype system is built on the idea of derivation. Beginning with one basic datatype, others are derived. In total, the Datatypes specification defines 44 built-in datatypes (datatypes that are *built into* the specification) that you can use. In addition to these built-in datatypes, you can derive your own datatypes using techniques such as restricting the datatype, extending the datatype, adding datatypes, or allowing a datatype to consist of a list of datatypes. Each of these techniques (restriction, extension, list, and union) is defined by the Structure document.

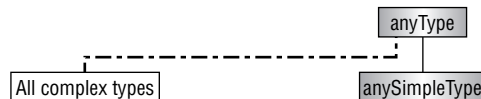
When we first think of datatypes, the assumption is that we're only talking about simple types (elements with text only or attribute values); however, the concept of datatypeing applies to both simple and complex types. In the Datatypes specification, you'll find a discussion of simple types (working with complex types refers to working with content models and is covered in the Structures specification). One of the easiest ways to begin to understand XML Schema datatypes is to look at their type hierarchy.

### Type Hierarchy

The *type hierarchy* identifies all 44 built-in datatypes and provides a road map for how each datatype was derived. All Schema datatypes (simple and complex) are derived from a root type. Figure 5.11 shows a snippet of the schema datatype hierarchy. In the figure, you should notice that a root type, `anyType`, is defined. When this type is used, it allows any content to appear. From `anyType`, the type hierarchy branches into simple types and complex types.

**FIGURE 5.11:**

XML Schema type hierarchy

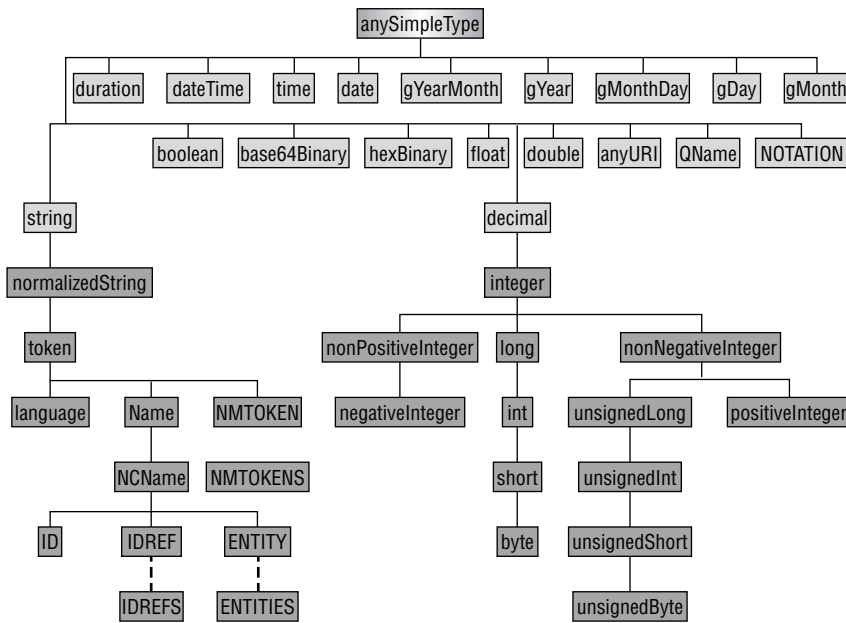


As covered previously in “Type Definitions,” elements defined as simple types differ from complex types in that they cannot have child elements or attributes. In addition to text-only elements, attributes are always defined as simple types. On the other hand, only elements that may contain child elements and/or attributes are considered complex types. As you follow the type hierarchy, you’ll notice that the root of all simple types is `anySimpleType`. Like `anyType`, this datatype can also be used. All 44 built-in simple types are derived from `anySimpleType`. You should note that each simple type is derived by restriction. None of the simple types is derived by extension; if you were to extend a simple type, you would, in theory, have to add a child element or attribute that contradicts the definition of a simple type.

If you need to use a datatype that is not defined by the specification, you have to derive your own. Figure 5.12 provides you with an illustration of the simple type definition component that is used to derive simple types. The only way to modify a built-in datatype is to restrict its content (restriction), permit a list of possible datatypes (list), or derive a new type that is based on the combination of other datatypes (union).

**FIGURE 5.12:**

Simple type structure hierarchy



## Datatype Properties

Every datatype is defined to consist of an ordered set of three properties: value space, lexical space, and facets. Each of these helps constrain the possible content.

### Value Space

The *value space* defines the allowable set of values for a given datatype. Each value in the value space has at least one corresponding literal in the lexical space. For example, let's assume the `street` element in the following markup is defined in a schema as a `string` datatype (a *string* is a sequence of characters). For a `string` type, the value space and the lexical space are the same. `Main` is part of the value space for a `string` type (it's a group of four characters) and `Main` is also the lexical representation—the `string` literal that represents the value:

```
<!-- instance document -->
<street>Main</street>
```

The `total` element is defined as a `float` datatype in our hypothetical schema document:

```
<!-- instance document -->
<total>135000.00</total>
```

In this case, the representation of `total` is in string format, but the actual value is a numeric value that can be represented in many ways, including 135000, 135000.00, and 13.5E4. These three representations have the same numeric value but are three different literals from the float lexical space.

### Lexical Space

The *lexical space* defines the set of valid literals for a given datatype. This is important because many values can oftentimes be expressed by multiple lexical representations. For example, 200.0 can also be expressed as 2.0e2.

### Facets

*Facets* provide a set of defining aspects of a given datatype. If you want to define a minimum or maximum constraint, or if you need to restrict the length of a string datatype, you'll have to use a facet. There are two types of facets: fundamental and constraining:

**Fundamental facets** Fundamental facets define the datatype that semantically characterizes the facet's value. The fundamental facets are as follows:

**equal** Values with an `equal` facet can be compared and determined to be equal or not equal. For any values *a* and *b* in a value space, only one of the following is true:

`a=b` or `a!=b`.

**ordered** Values with an `ordered` facet have an ordered relationship with each other; for example, groups of numbers or groups of words that can be placed in an ordered sequence relative to each other have an `ordered` facet. The `ordered` facet can take the values `false` (not ordered), `partial`, or `total`.

**bounded** Values with a `bounded` facet fit into a range of values with a specific lower and/or upper limit.

**cardinality** The number of values in a value space can be finite or countably infinite.

**numeric** Values can be classified as numeric (`value="true"`) or nonnumeric (`value="false"`).

**Constraining (or nonfundamental) facets** *Constraining facets* are optional properties that constrain the permitted values of a datatype. The following are allowable constraining facets:

**length** A nonnegative integer that defines the number of units of length. The units of length vary depending on the datatype. For example, for a string datatype, the units of length are measured in characters, whereas for a `hexBinary` datatype, units are measured in octets (8 bits) of binary data.

**minLength** A nonnegative integer that defines the minimum number of units of length. The units of length vary depending on the datatype.

**maxLength** A nonnegative integer that defines the maximum number of units of length. The units of length vary depending on the datatype.

**pattern** Constrains the lexical space to literals that must match a defined pattern. The value must be a regular expression (defined in Chapter 7). For example, you can define a pattern for ISBNs. Using the **pattern** facet, the value could be defined as follows: `<xsd:pattern value="[0-9]{10}" />`.

**enumeration** Constrains the value of the datatype to a defined set of values. For example, an **importance** attribute can be limited to accepting only the value **high** or **low**.

**whiteSpace** The value of this facet specifies how white space (tabs, line feeds, carriage returns, and spaces) is processed. The **whiteSpace** facet can accept only one of three values: **preserve**, **replace**, or **collapse**.

**maxInclusive** Defines the inclusive upper bound for a datatype with the **ordered** property. Because it's inclusive, the defined value may be included within the value space. For example, you can define an upper bound of 100 for an integer datatype; in this case, the datatype may accept any integer less than or equal to 100.

**maxExclusive** Defines the exclusive upper bound for a datatype with the **ordered** property. Because it's exclusive, the defined value may *not* be included within the value space. For example, you can define an upper bound of 100 for an integer datatype; in this case, the datatype may accept any integer less than 100.

**minInclusive** Defines the inclusive lower bound for a datatype with the **ordered** property. Because it's inclusive, the defined value may be included within the value space. For example, you can define an upper bound of 10 for an integer datatype; in this case, the datatype may accept any integer greater than or equal to 10.

**minExclusive** Defines the exclusive lower bound for a datatype with the **ordered** property. Because it's exclusive, the defined value may *not* be included within the value space. For example, you can define an upper bound of 10 for an integer datatype; in this case, the datatype may accept any integer greater than 10.

**totalDigits** Defines the maximum number of digits allowable for a datatype derived from the **decimal** type.

**fractionDigits** Defines the maximum number of digits allowable for the fractional part of a datatype derived from the **decimal** type.

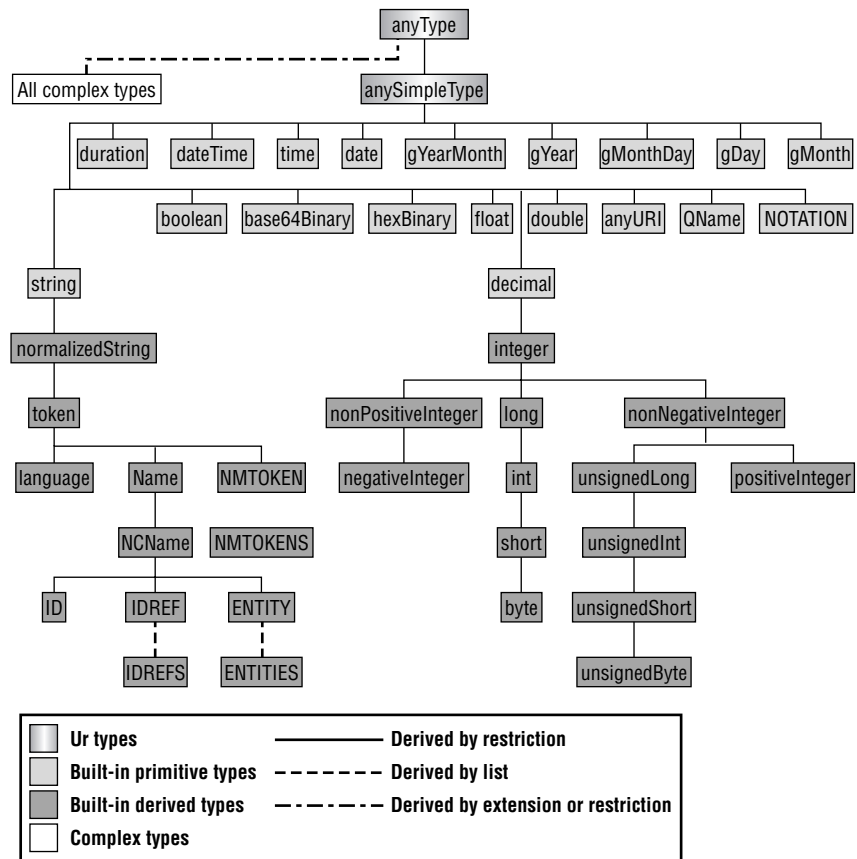
## Built-In and User-Derived

XML Schema provides document authors with a predefined set of datatypes, known as *built-in datatypes*. In addition to built-in datatypes, document authors may abstract their own datatype based on built-in types. User-derived datatypes provide an essential tool for constraining content for specific purposes.

## Built-In

Built-in datatypes are already defined for you by the Datatypes specification. Figure 5.13 provides the type hierarchy for all built-in datatypes.

**FIGURE 5.13:**  
XML Schema type  
hierarchy for built-in  
datatypes



According to the Datatypes document, XML Schema allows for two basic types of built-in datatypes: primitive and derived. Although there's a clear distinction defined by the

Datatypes document, this distinction does not greatly affect how these datatypes may be used:

**Primitive** Primitive datatypes are those not defined in terms of other datatypes. Primitive datatypes are the basis for all other datatypes. The primitive datatypes are identified in Table 5.1. (See Chapter 7 for a more detailed list of these datatypes.)

**TABLE 5.1:** Primitive Datatypes

Type Name	Type	Derivation
duration	Built-in primitive	anySimpleType
dateTime	Built-in primitive	anySimpleType
time	Built-in primitive	anySimpleType
date	Built-in primitive	anySimpleType
gYearMonth	Built-in primitive	anySimpleType
gYear	Built-in primitive	anySimpleType
gMonthDay	Built-in primitive	anySimpleType
gDay	Built-in primitive	anySimpleType
gMonth	Built-in primitive	anySimpleType
boolean	Built-in primitive	anySimpleType
base64Binary	Built-in primitive	anySimpleType
hexBinary	Built-in primitive	anySimpleType
float	Built-in primitive	anySimpleType
double	Built-in primitive	anySimpleType
anyURI	Built-in primitive	anySimpleType
QName	Built-in primitive	anySimpleType
NOTATION	Built-in primitive	anySimpleType
string	Built-in primitive	anySimpleType
decimal	Built-in primitive	anySimpleType

**Derived** A derived datatype is one that is derived from a base type. The derived type inherits its value space from its base type and may also constrain the derived value space to be an explicit subset of the base type's value space. The predefined derived datatypes are identified in Table 5.2.

**TABLE 5.2:** Derived Datatypes

Type Name	Type	Derivation
integer	Built-in derived	decimal
token	Built-in derived	normalizedString
nonPositiveInteger	Built-in derived	integer
long	Built-in derived	integer
nonNegativeInteger	Built-in derived	integer
language	Built-in derived	token
Name	Built-in derived	token
NMTOKEN	Built-in derived	token
negativeInteger	Built-in derived	nonPositiveInteger
int	Built-in derived	long
unsignedLong	Built-in derived	nonNegativeInteger
positiveInteger	Built-in derived	nonNegativeInteger
NCName	Built-in derived	Name
NMTOKENS	Built-in derived	NMTOKEN
short	Built-in derived	int
unsignedInt	Built-in derived	unsignedLong
ID	Built-in derived	NCName
IDREF	Built-in derived	NCName
ENTITY	Built-in derived	NCName
byte	Built-in derived	short
unsignedShort	Built-in derived	unsignedInt
IDREFS	Built-in derived	IDREF
ENTITIES	Built-in derived	ENTITY
unsignedByte	Built-in derived	unsignedShort

### User-Derived

Whereas the XML Schema specification offers you 44 datatypes, there will be occasions when you'll need to define your own datatypes. The only way to define your own datatype is to derive it using a predefined datatype as the base. There are three different ways a datatype can be derived:

**Restriction** Uses facets to restrict the allowable content.

**List** Derived datatype is a list of white-space-separated values of the base datatype.

**Union** There are two or more base datatypes, and the value space-derived datatype is formed from the union of the value spaces of the base types.

Deriving datatypes is covered in more detail in the following section and in Chapter 7.



### Datatype Categories

Datatypes in XML Schema are categorized into three groups called *characterization dichotomies*:

- Atomic vs. list vs. union datatypes
- Primitive vs. derived datatypes
- Built-in vs. user-derived datatypes

The definitions of these datatypes are specific to XML Schema and do not necessarily correspond exactly to the definition of these types in any other particular programming language.

#### Atomic

*Atomic datatypes* have a value that is considered indivisible and cannot be further broken down into other components. Atomic datatypes can be primitive or derived. All primitive datatypes are atomic.

#### List

*List datatypes* make up a group of defined, finite-length sequences of atomic values. The atomic datatype included in the definition of a list datatype is known as the `itemType` of that specific list datatype. List datatypes are always derived.

#### Union

*Union datatypes* are always derived from atomic or list datatypes and include the value spaces and lexical spaces of all the datatypes used to derive the union datatype. The datatypes in the group of datatypes included in a union datatype are known as the `memberTypes` of that specific union datatype.

We are now at the end of our introduction to XML Schema datatypes; however, there's much more to learn. Chapter 7 is dedicated to datatypes.

## The Role of Namespaces

One of the more notable advantages to using XML Schema is that it's namespace aware. To take advantage of namespaces and XML Schema, you first need to understand the role of namespaces in XML.

### Why Namespaces?

Namespaces were defined after XML 1.0 was formally presented to the public. After XML 1.0 was completed, the W3C set out to resolve a few problems, one of which dealt with naming conflicts. To understand the significance of this problem, first think about the future of the Web.

Shortly after the W3C introduced XML 1.0, an entire family of languages began to pop up, such as Mathematical Markup Language (MathML), Synchronized Multimedia Integration Language (SMIL), Scalable Vector Graphics (SVG), XLink, XForms, and the Extensible Hypertext Markup Language (XHTML). Instead of relying on one language to bear the burden of communicating on the Web, the idea was to present many languages that could work together. If functions were modularized, each language could do what it does best. The problem arises when a developer needs to use multiple vocabularies within the same application. For example, one might need to use SVG, SMIL, XHTML, and XForms for an interactive Web site. When mixing vocabularies, you have to have a way to distinguish between element types. Take the following example:

```
<html>
  <head>
    <title>Book List</title>
  </head>
  <body>
    <publications>
      <book>
        <title>Mastering XHTML</title>
        <author>Ed Tittel</author>
      </book>
      <book>
        <title>Java Developer's Guide to E-Commerce
          with XML and JSP</title>
        <author>William Brogden</author>
      </book>
    </publications>
  </body>
</html>
```

In this example, there's no way to distinguish between the two element types of the same name (`title`), even though they are semantically different.

A namespace solves this problem, providing a unique identifier for a collection of elements and/or attributes. The interesting thing about namespaces is that they're only symbolic references (the URL used typically doesn't point to anything specific). Listing 5.7 uses namespaces to resolve the name conflict in the preceding example.

**Listing 5.7****An XML Document That Uses Default Namespaces**

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Book List</title>
  </head>
  <body>
    <publications xmlns="http://www.lanw.com/namespaces/pub">
```

```
<book>
  <title>Mastering XHTML</title>
  <author>Ed Tittel</author>
</book>
<book>
  <title>Java Developer's Guide to E-Commerce
    with XML and JSP</title>
  <author>William Brogden</author>
</book>
</publications>
</body>
</html>
```

---

All of the text in bold belongs to the `publications` namespace, <http://www.lanw.com/namespaces/pub>, whereas all the XHTML elements belong to the XHTML namespace, <http://www.w3.org/1999/xhtml>.

## Namespace URLs

Namespaces can confuse XML novices because the namespace names are URLs and therefore often mistaken for a Web address that points to some resource. However, XML namespace names are URLs that don't necessarily have to point to anything. Most times they don't, in fact. For example, if you visit the XSLT namespace, you would find a document that contains a single sentence: "This is the XSLT namespace." The unique identifier is meant to be symbolic; therefore, there's no need for a document to be defined. To help out those who are first learning about XML namespaces, the W3C commonly defines a resource for the namespace name; however, this is not required by the XML Namespace recommendation. URLs were selected for namespace names because they contain domain names that can work globally across the Internet and they are unique.

## Declaring Namespaces

To declare a namespace, you need to be aware of the three possible parts of a namespace declaration:

- `xmlns` identifies the value as an XML namespace. This is required to declare a namespace and can be attached to any XML element.
- `:prefix` identifies a namespace prefix. It (including the colon) is only used if you're declaring a namespace prefix. If it's used, any element found in the document that uses the prefix (`prefix:element`) is then assumed to fall under the scope of the declared namespace.
- `namespaceURI` is the unique identifier. The value does not have to point to a Web resource; it's only a symbolic identifier. The value is required and must be defined within single or double quotation marks.

There are two different ways you can define a namespace:

**Default namespace** Defines a namespace using the `xmlns` attribute without a prefix, and all child elements are assumed to belong to the defined namespace.

**Prefixed namespace** Defines a namespace using the `xmlns` attribute with a prefix. When the prefix is attached to an element, it's assumed to belong to that namespace.

Listing 5.8 provides both an example of namespace defaulting and an example of namespace scooping (using a prefixed namespace).

**Listing 5.8****An XML Document That Uses Prefixed Namespaces**

```
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Book List</title>
  </head>
  <body>
    <pub:publications
      xmlns:pub="http://www.lanw.com/namespaces/pub">
      <pub:book>
        <pub:title>Mastering XHTML</pub:title>
        <pub:author>Ed Tittel</pub:author>
      </pub:book>
      <pub:book>
        <pub:title>Java Developer's Guide to E-Commerce
          with XML and JSP</pub:title>
        <pub:author>William Brogden</pub:author>
      </pub:book>
    </pub:publications>
  </body>
</html>
```

## XML Schema Namespaces

After you understand the basics behind XML namespaces, you're ready to face the three namespaces defined by XML Schema:

- XML Schema namespace is used for W3C XML Schema elements. You can define this namespace as a default namespace or with a prefix. The namespace URI is `http://www.w3.org/2001/XMLSchema`.
- XML Schema Datatype namespace is used to define built-in datatypes. The XML Schema namespace (`http://www.w3.org/2001/XMLSchema`) also defines built-in datatypes; therefore, it's not necessary to define it if you have defined the XML Schema namespace. The namespace URI is `http://www.w3.org/2001/XMLSchema-datatypes`.

- XML Schema instance namespace is used for W3C XML Schema attributes used in XML instance documents. The namespace is used to introduce `xsi:type`, `xsi:nil`, `xsi:schemaLocation`, and `xsi:noNamespaceSchemaLocation` attributes in instance documents. This namespace should be defined with an `xsi` prefix. The namespace URI is `http://www.w3.org/2001/XMLSchema-instance`.

The XML Schema namespace must be defined for a schema document. In addition, the XML Schema namespace must be used in the XML document instance when associating the schema. You'll learn more about these namespaces in Chapter 6.

## Targeting Namespaces with XML Schema

If you want to use a namespace with your document instances, you have to define the namespace in your XML Schema document. To do this, you have to define a target namespace (`targetNamespace="http://www.yournamespace.com"`). The target namespace identifies the namespace to which the corresponding XML document should adhere. The following XML document uses a namespace (in bold):

```
<?xml version="1.0"?>
<publications xmlns="http://www.lanw.com/namespaces/pub">
  <book>
    <title>Mastering XHTML</title>
    <author>Ed Tittel</author>
  </book>
  <book>
    <title>Java Developer's Guide to E-Commerce with
      XML and JSP</title>
    <author>William Brogden</author>
  </book>
</publications>
```

XML Schema allows you to define a target namespace for any document that is to conform to the schema:

```
<schema targetNamespace="http://www.lanw.com/namespaces/pub">
...
</schema>
```

## Identity Constraints

Whereas XML 1.0 allowed document authors to define an ID datatype for attribute values that, in turn, required the value to be unique to a document instance, XML Schema offers a

more flexible and powerful mechanism for defining identity constraints. There are three different ways to define identity constraints using XML Schema:

- You can use the `key` and `keyref` elements to define relationships between elements. These are similar to the `ID`/`IDREF` connection; however, keys can be unique to a defined element set. XML Schema uses XPath expressions to identify an element set to which the key is required to be unique.
- You can use the `unique` element to specify that any attribute or element value must be unique within a defined scope. The `unique` element may contain a `selector` or `field` element. The `selector` element defines the element set for the uniqueness constraint. The `field` element is used to identify the attribute or element field relative to each selected element that has to be unique within the scope of the set of selected elements. Both the `selector` and `field` elements use an `xpath` attribute that contains an XPath expression.
- You can use an `ID` datatype. The `ID` datatype requires that the value be unique to the entire document instance. The `IDREF` datatype can be used to reference a defined `ID`.

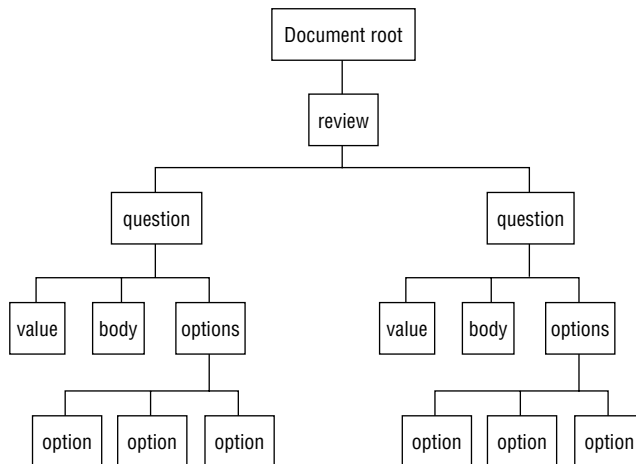
To take advantage of XML Schema identity constraints, you need to understand the XPath language. XPath is a language that allows you to address into an XML document structure using a tree representation of your data—similar to the DOM. The tree representation of your data consists of nodes that you can then navigate. Figure 5.14 is a simplified tree representation of the XML document in Listing 5.9.

**Listing 5.9****The XML Document Used to Show a Simplified Tree Representation in Figure 5.14**

```
<review set="1">
  <question value="1">
    <body>Do you like cats?</body>
    <options>
      <option value="a">yes</option>
      <option value="b">no</option>
      <option value="c">maybe</option>
    </options>
  </question>
  <question value="1">
    <body>Do you like dogs?</body>
    <options>
      <option value="a">yes</option>
      <option value="b">no</option>
      <option value="c">maybe</option>
    </options>
  </question>
</review>
```

**FIGURE 5.14:**

A simplified XPath tree representation



If you wanted to require that the `value` attribute is unique for every `option` element, you could use XPath to identify the `options` node set, in essence, requiring that the value of the `value` attribute be unique within each `question` element. This is a helpful tool for document modeling. With DTDs, you could only define identity constraints using IDs that had to be unique to the entire document instance, which doesn't give you much flexibility. By using XPath, you're able to identify node sets and therefore work on a modular level.

**TIP**

We take a closer look at XPath and identity constraints in Chapter 6.

## Unique Values

In almost every document model, there's a need for identity constraints. Unique identifiers are found everywhere in data: ISBNs, Social Security numbers, product codes, and customer IDs. By enforcing unique values, you're able to check data integrity before sending that data to an application. If you're familiar with XML 1.0, you already know how to use an ID datatype. Although XML Schema supports this datatype (which is helpful for backward compatibility), it offers a more flexible model for defining uniqueness.

There are several disadvantages to using XML 1.0 IDs. To begin with, they only apply to attribute values. There's no way to define an element value as unique using XML 1.0. Second, the attribute value must then be unique to the entire document, and therefore, only one set of unique values is allowed, which would not be helpful when designing a document model for Listing 5.9. And finally, one of the most unnerving disadvantages is that XML 1.0

IDs cannot begin with a number and they cannot contain spaces. Because most unique identifiers begin with numbers, this last limitation was a hard one to swallow.

Using XML Schema identity constraints, you gain flexibility. For example, both element and attribute values can be defined to be unique. In addition, that unique value can be any datatype and therefore can begin with a number. And, as you know, the numbers don't have to be unique to the document instance. Instead, you can define a scope for unique value.

The `unique` element is used to require an element or attribute's value to be unique within a specified range. When using the `unique` element, you use XPath to identify the range (`publications/book`).

```
<xsd:unique name="bookID">
  <xsd:selector xpath="publications/book"/>
  <xsd:field xpath="@id"/>
</xsd:unique>
```

## Key and Key References

Defining relationships is an important functionality, especially for those working with relational databases. XML Schema allows you to represent relationships using `key` and `keyRef` elements. These elements can also take advantage of XPath, thereby allowing you to specify the scope of a uniqueness constraint. The `key` element functions much like the `unique` element. The following shows how the `key` element is used:

```
<xsd:key name="bookKey">
  <xsd:selector xpath="publications/book"/>
  <xsd:field xpath="@id"/>
</xsd:key>

<xsd:keyref name="bookRef" refer="bookKey">
  <xsd:selector xpath="publications/review"/>
  <xsd:field xpath="book/@bookRef"/>
</xsd:keyref>
```

## About Derivation and Substitution

XML Schema gets interesting when you're able to manipulate content models to tailor a particular application. Using derivation and substitution, you can redefine, append, and restrict content models to suit your needs. For example, you can add elements or attributes using the `extension` element, or you can eliminate elements using the `restriction` element. Derivation techniques apply to both simple and complex types.



## Derivation

XML Schema introduces derivation to XML validation. *Derivation* allows you to derive new types by restricting or extending the type definition. Extension is only allowed for complex types.

With this new functionality, document authors are able to derive new types based on previously created types or ones that someone else may have defined. Deriving new types, whether they be simple or complex, is an improvement over having to use the static content models you could create using XML DTDs. XML Schema allows you to derive new complex and simple types.

## Complex Types

You can constrain the element and/or attribute declarations of an existing type and therefore derive new content models based on previously defined models. The only two elements allowed for this type of derivation are restriction and extension. The restriction element contains an attribute that identifies the base type and contains the new content model definitions. In the following example, we have restricted a base type (`bookType`) and redefined its content model to only allow for one `title` element and one `author` element:

```
<xsd:complexType name="newBookType">
  <xsd:complexContent>
    <xsd:restriction base="bookType">
      <xsd:sequence>
        <xsd:element name="title" type="xsd:string"/>
        <xsd:element name="author" type="xsd:string"/>
      </xsd:sequence>
    </xsd:restriction>
  </xsd:complexContent>
</xsd:complexType>
```

Restriction is not the only way to derive new complex types; you can also extend content models using the extension element. Using the extension element, you can add an element and/or elements to an existing content model. In the following example, we add a `description` element and an `isbn` element:

```
<xsd:complexType name="newBookType">
  <xsd:complexContent>
    <xsd:extension base="bookType">
      <xsd:sequence>
        <xsd:element name="description" type="xsd:string"/>
        <xsd:element name="isbn" type="xsd:string"/>
      </xsd:sequence>
    </xsd:extension>
  </xsd:complexContent>
</xsd:complexType>
```

## Simple Types

As with complex types, you can use derivation techniques to manipulate simple types. Using derivation, you can derive new simple types that serve as datatypes for content. Deriving simple types is not exactly the same as deriving complex types. One limitation is that you cannot extend a simple type. When deriving simple types, you can use one of the following elements: restriction, list, or union. Listing 5.10 provides an example of several derived simple types.



**Listing 5.10** An XML Schema Document Using Derivation

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:year"/>
  </xsd:simpleType>

  <xsd:simpleType name="isbnType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{10}"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="editionType">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="10"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ppType">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="2000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:element name="author" type="xsd:string"/>

  <xsd:complexType name="authorsType">
    <xsd:sequence>
      <xsd:element ref="author" maxOccurs="5"/>
    </xsd:sequence>
  </xsd:complexType>
```

```

<xsd:complexType name="bookType">
  <xsd:sequence>
    <xsd:element ref="title"/>
    <xsd:element name="authors" type="authorsType"/>
    <xsd:element name="pubDate" type="pubDateType"/>
    <xsd:element ref="publisher"/>
    <xsd:element name="size" type="sizeType"/>
    <xsd:element name="topics" type="topicsType"/>
    <xsd:element name="errata" type="errataType"
      minOccurs="0"/>
    <xsd:element ref="description"/>
    <xsd:element ref="website" minOccurs="0"/>
  </xsd:sequence>
  <xsd:attribute name="isbn" type="isbnType"
    use="required"/>
  <xsd:attribute name="edition" type="editionType"
    use="required"/>
  <xsd:attribute name="cat" type="xsd:NMTOKENS"
    use="required"/>
  <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<xsd:element name="description" type="xsd:string"/>

<xsd:complexType name="errataType">
  <xsd:attribute name="code" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:complexType name="pubDateType">
  <xsd:attribute name="year" type="xsd:string" use="required"/>
</xsd:complexType>

<xsd:element name="publications">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="book" type="bookType"
        maxOccurs="unbounded"/>
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="publisher" type="xsd:string"/>

<xsd:complexType name="sizeType">
  <xsd:attribute name="pp" type="ppType" use="required"/>
</xsd:complexType>

<xsd:element name="title" type="xsd:string"/>
<xsd:element name="topic" type="xsd:string"/>

<xsd:complexType name="topicsType">

```

```

    <xsd:sequence>
      <xsd:element ref="topic" maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>

  <xsd:element name="website" type="xsd:string" />
</xsd:schema>

```

---

## Restriction

The `restriction` element creates a datatype that is a subset of an existing type. You can apply a restriction using facets that limit properties of a datatype. For example, you can use the `maxLength` or `minLength` facets to define the maximum and minimum number of characters allowed for a given datatype. Each datatype has a specifically defined list of allowable facets. For example, you can't use the same facets for integer and string datatypes (facets are discussed in Chapter 7).

In the following example, we use the pattern facet to define the allowable pattern for an ISBN:

```

<xsd:simpleType name="isbnType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{10}" />
  </xsd:restriction>
</xsd:simpleType>

```

## List

The `list` element creates a datatype that consists of a white-space-separated list of values. The `list` element uses an attribute to define the base type that is to be used for the list:

```

<xsd:simpleType name="isbnType">
  <xsd:restriction base="xsd:string">
    <xsd:pattern value="[0-9]{10}" />
  </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="isbnTypeList">
  <xsd:list itemType="isbnType" />
</xsd:simpleType>

```

## Union

The `union` element creates a datatype that is derived from multiple datatypes. The `union` element uses an attribute to define all base types participating in the union. Any of the defined base datatypes are allowed to be used. In the following example, the `oneType` and `twoType` datatypes are allowed:

```

<xsd:simpleType name="exampleTypeUnion">
  <xsd:union memberTypes="oneType twoType" />
</xsd:simpleType>

```

## Substitution

XML Schema supports substitution mechanisms that allow one group of elements to be replaced by another group of elements. This is especially helpful when you're working with a predefined XML Schema document. There are many instances in which you might want to redefine a content model, or even substitute the entire model with one of your own. Substitution is only defined for complex types. Listing 5.11 defines a schema document, but then at the end of that document, it defines a substitution group for the `magType` and `bookType` complex type definitions. The purpose of this was to define a general content model (title, author, and description) that can be used for both for the time being. Later, we can remove the substitution group and work with the more complex models.



### Listing 5.11

### An XML Schema Document Using Derivation

```
<xsd:schema
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  elementFormDefault="qualified"
  targetNamespace="http://www.lanw.com/namespaces/pub"
  xmlns="http://www.lanw.com/namespaces/pub">

  <xsd:simpleType name="yearType">
    <xsd:restriction base="xsd:year"/>
  </xsd:simpleType>

  <xsd:simpleType name="isbnType">
    <xsd:restriction base="xsd:string">
      <xsd:pattern value="[0-9]{10}"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="editionType">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="10"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:simpleType name="ppType">
    <xsd:restriction base="xsd:nonNegativeInteger">
      <xsd:minInclusive value="1"/>
      <xsd:maxInclusive value="2000"/>
    </xsd:restriction>
  </xsd:simpleType>

  <xsd:element name="author" type="xsd:string"/>

  <xsd:complexType name="authorsType">
    <xsd:sequence>
```

```

        <xsd:element ref="author" maxOccurs="5"/>
    </xsd:sequence>
</xsd:complexType>

<xsd:complexType name="bookType">
    <xsd:sequence>
        <xsd:element ref="title"/>
        <xsd:element name="authors" type="authorsType"/>
        <xsd:element name="pubDate" type="pubDateType"/>
        <xsd:element ref="publisher"/>
        <xsd:element name="size" type="sizeType"/>
        <xsd:element name="topics" type="topicsType"/>
        <xsd:element name="errata" type="errataType"
            minOccurs="0"/>
        <xsd:element ref="description"/>
        <xsd:element ref="website" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="isbn" type="isbnType"
        use="required"/>
    <xsd:attribute name="edition" type="editionType"
        use="required"/>
    <xsd:attribute name="cat" type="xsd:NMTOKENS"
        use="required"/>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<xsd:complexType name="magType">
    <xsd:sequence>
        <xsd:element ref="title"/>
        <xsd:element name="authors" type="authorsType"/>
        <xsd:element name="pubDate" type="pubDateType"/>
        <xsd:element name="size" type="sizeType"/>
        <xsd:element name="topics" type="topicsType"/>
        <xsd:element ref="description"/>
    </xsd:sequence>
    <xsd:attribute name="publication" type="xsd:string"
        use="required"/>
    <xsd:attribute name="edition" type="editionType"
        use="required"/>
    <xsd:attribute name="cat" type="xsd:NMTOKENS"
        use="required"/>
    <xsd:attribute name="id" type="xsd:ID"/>
</xsd:complexType>

<xsd:element name="description" type="xsd:string"/>

<xsd:complexType name="errataType">
    <xsd:attribute name="code" type="xsd:string"
        use="required"/>
</xsd:complexType>

```

```
<xsd:complexType name="pubDateType">
  <xsd:attribute name="year" type="xsd:string"
    use="required" />
</xsd:complexType>

<xsd:element name="publications">
  <xsd:complexType>
    <xsd:sequence>
      <xsd:element name="book" type="bookType"
        maxOccurs="unbounded" />
      <xsd:element name="mag" type="magType"
        maxOccurs="unbounded" />
    </xsd:sequence>
  </xsd:complexType>
</xsd:element>

<xsd:element name="publisher" type="xsd:string" />

<xsd:complexType name="sizeType">
  <xsd:attribute name="pp" type="ppType" use="required" />
</xsd:complexType>

<xsd:element name="title" type="xsd:string" />
<xsd:element name="topic" type="xsd:string" />

<xsd:complexType name="topicsType">
  <xsd:sequence>
    <xsd:element ref="topic" maxOccurs="unbounded" />
  </xsd:sequence>
</xsd:complexType>

<xsd:element name="website" type="xsd:string" />

<xsd:element name="book" type="bookType"
  substitutionGroup="genType" />
<xsd:element name="mag" type="magType"
  substitutionGroup="genType" />

<xsd:complexType name="genType">
  <xsd:sequence>
    <xsd:element ref="title" />
    <xsd:element ref="author" />
    <xsd:element ref="description" />
  </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

---

## Summary

XML Schema brings powerful validation to the world of XML. There are many reasons one would want to use XML Schema rather than an alternative validation tool. However, this chapter doesn't focus on why you might want to use XML Schema; it assumes you already do. In this chapter, we focus on the basic tenets of the language. There are a handful of constructs that developers must be familiar with if they want to take advantage of XML Schema. Once a developer has the basic concepts down (e.g., the difference between simple and complex types, how to derive simple and complex types, etc.), the rest will fall into place.

Understanding the XML Schema vocabulary is not the difficult part of mastering the language—it's conceptually understanding the language's functionality. There are some basic tenets that you must understand, and before you move on to the next chapter, please take a few moments to make sure that you have an understanding of the primary ones.

The XML Schema language is broken into two separate documents. The first one defines the XML Schema components. These components are introduced using the schema vocabulary. For example, the `element` element declares an element, and the `attribute` element declares an attribute. There are a variety of schema components defined, the most basic of which are listed here:

**Type definitions** You can define two types of definitions: simple and complex.

**Declarations** There are three types of declarations that you can define; these include element, attribute, and notation.

**Annotations** You can add comments intended for humans or applications.

The XML Schema Datatypes document is the second part of the XML Schema language, and it defines the built-in datatypes that are predefined and ready for use. These datatypes serve as base types for any datatypes you might want to derive on your own. Although it's helpful to have 44 predefined datatypes, you'll undoubtedly want to derive your own. By using derivation, you can define patterns and limitations for your datatypes that will ensure data integrity and safeguard against errors. There are three ways you can derive datatypes: restriction, union, and list.

Not only can you derive simple types, but you can also use derivation to manipulate complex content models. Both derivation and substitution provide the document author with the flexibility to tailor any schema for a particular application. This is a great advantage that XML Schema enjoys over XML DTDs.

This is by no means the last stop. The next step to understanding the XML Schema language is to read Chapters 6 and 7. These two chapters make up the heart of the book because they define the XML Schema components and datatypes. With a firm understanding of the basic schema concepts, you're now ready for them.



