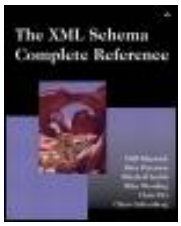


XML Schema Complete Reference, The

By Cliff Binstock, Dave Peterson, Mitchell Smith, Mike Wooding, Chris Dix, Chris Galtenberg

Publisher: Addison Wesley
Pub Date: September 27, 2002
ISBN: 0-672-32374-5
Pages: 1008

With the successful implementation of XML Schema, developers are learning how to increase productivity, improve software reliability, minimize development time, and decrease time to market. This in-depth reference is an all-in-one resource designed to help developers leverage the power and potential of XML schemas by offering a complete roadmap to their creation, design, and use.

~~This authoritative reference and tutorial is filled with practical insights and detailed examples. The book begins by providing a conceptual introduction to XML Schema. From there, coverage shifts to the W3C Schema Recommendation and how to apply schemas to specific business goals. The authors provide insight and instruction throughout on integrating XML schemas into existing technologies such as .NET, Java, Visual Basic, Oracle, and more. The book concludes with a complete case study designed to reinforce and illustrate material covered.~~

Additional topics include:

- Applications for schemas
- Simple and complex types
- XML schema processing and validation
- Namespaces in XML
- Using schemas with DOM and SAX
- XML schema document syntax
- XML Information Sets
- XML Schema applications of XPath

~~Whether designing a schema from scratch or integrating schemas into contemporary technologies, *The XML Schema Complete Reference* is the most complete and definitive sourcebook available for the XML Schema environment.~~

777

PART -III is talks about how the XML Instance or XML Schema document is being loaded into object representation. to learn about xml schema writing no need that part-3.
if you find time spend with part-3 to get more idea about APPLICATION DESIGNING. since processor are designed by big ARCHITECT.

right now i have deleted that part-3. when need, just down load this same book and read it. (search by title of book or authors of this book)

i can say this is very good book for learning xml schema

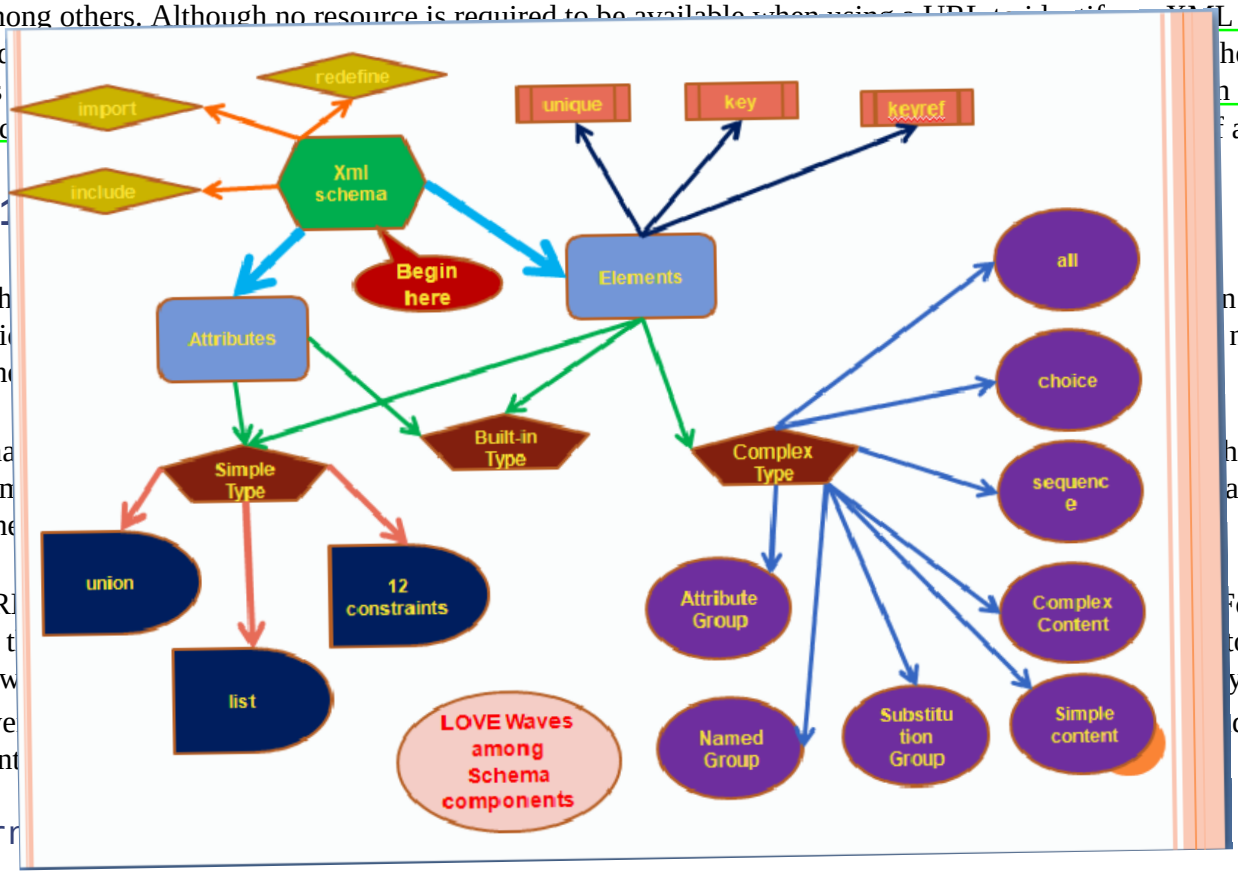
read each chapter.

where ever i have stroked out lines, no need to learn or no need to have more concentration.

-paramasivam.

it just for differentiation

among others. Although no resource is required to be available when using a URL to identify an XML Namespace, people—and



Wh n protocols to ensure uni naming clashes with oth

Tha heme-specific string is a don ain rights in forbidding othe

URI orce (IETF). After you go t o register an NID, how y it as *experimental.*

Eve d be to use a domain you cont

```
urn
```

where '`example.com`' is a registered domain name under the control of the author. You might have noticed that the periods used in the URL were turned into dashes in the URN. ~~Both URLs and URNs have specific rules about which characters are allowed. For more information on creating URLs and URNs, see the IETF RFCs:~~

- ~~RFC 1630, *Universal Resource Identifiers in WWW*: http://www.ietf.org/rfc/rfc1630.txt~~
- ~~RFC 2141, *URN Syntax*: http://www.ietf.org/rfc/rfc2141.txt~~
- ~~RFC 1738, *Uniform Resource Locators (URL)*: http://www.ietf.org/rfc/rfc1738.txt~~

3.1.2 Using RDDL with Namespace URLs

When deciding whether to use a URL or a URN, a good rule is to use a URL if something is retrievable from the identified location. This could be an XML schema, documentation for the namespace, or something else. If you don't have anything to retrieve, use a URN. ~~When using a URL to identify a namespace, the question arises of what you get when you access the URL with a browser. There's no consensus in the XML community on this. Some in the community think it should be a DTD or schema. Others feel it should be documentation or a package of various resources related to the namespace being used.~~

it is like UDDI

To help solve this problem, an XML vocabulary called Resource Directory Description Language (RDDL) was developed by a group of individuals on the XML-DEV mailing list. This vocabulary uses the XLink Recommendation, *XML Linking Language (XLink) Version 1.0* (http://www.w3.org/TR/xlink), to identify a number of resources that can be associated with a namespace. Here's an example from the XML namespace that identifies a DTD as well as a schema:

```
<rddl:resource
    xlink:title="DTD for validation"
    xlink:arcrole=
        "http://www.rddl.org/purposes#validation"
    xlink:role=
```

```
        "http://www.isi.edu/in-notes/iana/assignments/media-types/text/xml-dtd"
    xlink:href="XMLSchema.dtd"/>
<rddl:resource id="xmlschema"
    xlink:title="XML Schema schema document"
    xlink:role="http://www.w3.org/2001/XMLSchema"
    xlink:arcrole=
        "http://www.rddl.org/purposes#schema-validation"
    xlink:href="XMLSchema.xsd"/>
```

When processing a RDDL document, you can use the `xlink:arcrole` attribute to determine what resources are available, and then use the `xlink:href` attribute to retrieve them. This gives you the flexibility to associate a variety of resources with a document. The application that processes the document can use whichever of the resources is most appropriate. For more information on RDDL, see http://www.rddl.org/.

3.2 Namespace Components

Namespaces enable you to create unique names that appear in an XML document. You associate a URI with the names that you use in your document. When a name is associated with a URI, it is called a "qualified name" and is said to be in that namespace. You can also have names in your document that are not in any namespace. These names are called "unqualified." You associate names in your document with a namespace by using a *prefix*. A prefix is an arbitrary string of characters that acts as shorthand for the namespace URI so that names are not overly long. The prefix is also optional, because you can declare a namespace to be a default, meaning all unprefixed element names are in that namespace by default.

3.3 Declaring Namespaces

Namespaces are declared by using a family of reserved attributes. In XML 1.0, attributes starting with the three letters `'xml'` (including variants with capital letters) are reserved for W3C use. To declare a default namespace, you add an attribute named `'xmlns'` that has a value of the URI you choose to associate the namespace with. For example:

```
<Customer xmlns="http://www.example.com/foo">
      *      *      *      *      *
</Customer>
```

that why in SAX event method, we check "" empty instead of null in startTag / endTag method

In the preceding example, a default namespace is declared. All unprefixed elements will be in the namespace associated with the URI `'http://www.example.com/foo'`

Note

As specified in the Namespace Recommendation, all unprefixed *attributes* are *unqualified*—not part of any namespace.

There needs to be a way to declare multiple in-scope namespaces. This can be done by using prefixes. A prefix is a short sequence of characters that appears at the left of an element or attribute name and separated from it by a colon. To declare a namespace that is associated with a prefix, you specify an attribute whose *name* is formed by prefixing *your* prefix with the special prefix `'xmlns'`

```
<foo:Customer xmlns:foo="http://www.example.com/foo">
```

```
    *       *       *       *       *
</foo:Customer>
```

In the preceding example, we declare a namespace that has the prefix '`foo`' bound to it. Any elements or attributes that have the prefix '`foo`' will be in the namespace `http://www.example.com/foo`. Because no default namespace is declared, all unprefixed elements are not in any namespace.

It is important to remember that a prefix is arbitrary, except for any prefix that starts with the characters '`xml`' (lowercase or capitals or mixed), which are reserved for W3C use. For example, when looking at schema examples in this book, you might see the prefix '`xs`' or '`xsd`' bound to the schema namespace and '`xsi`' bound to the schema instance namespace—however, any prefixes can be used as long as they are bound to the proper namespace.

Follow in Coding

i can use the same style

3.4 Qualified Names and QNames

The XML Recommendation defines a *name* as a character string that has certain restrictions as to what characters it may contain. An XML document that conforms to the Namespace Recommendation may include "qualified names." Elements, attributes, and entities are some of the XML document components that are required to have a name.

A *qualified name* has additional namespace information and is composed of three parts and may contain one colon. If there is a colon, the character string preceding it is the optional *prefix*. If there is no colon, the prefix is an empty string. The *local part* is what follows the colon. A qualified name also has a *namespace URI* that identifies the namespace. You do not see the namespace URI as part of the name when viewing the document, because it is declared higher up in the tree. The namespace URI is associated with the qualified name through the namespace declaration, the optional *prefix*, and the namespace scoping rules.

yes. a name space no need to have prefix at all. it is optional at all

You use a prefix when you are not using a default namespace or when you want to include names outside any default namespace. A prefix is required to specify attributes that belong to a namespace, because unprefixed attributes never belong to any namespace. Even though unprefixed attributes are not in a namespace, there is an implied association between them and the element they appear on. The prefix is an arbitrary sequence of characters that work as a shorthand notation for the namespace URI.

Schema represents qualified names with the `QName` datatype. For more information on the Schema datatype `QName`, see Section 12.4.1.

The following XML shows an example document that uses both qualified and unqualified names. The element parent declares a default namespace, one with no prefix, and another namespace that is bound to the prefix '`bar`'. Because of namespace scoping rules, '`Parent`' is a qualified name that is in the `http://www.example.com/foo` namespace and does not have a prefix. The element type name '`bar:Child`' is a qualified name that is in the `http://www.example.com/bar` namespace and is bound to the prefix '`bar`'. Last, because unprefixed attributes do not belong in any namespace, the attribute name '`attr`' is not a qualified name and is not associated with any namespace.

xmlns:foo="https://www.example.com/foo"

```
<?xml version="1.0" encoding="UTF-8"?>
<Parent "xmlns=http://www.example.com/foo"
  xmlns:bar="http://www.example.com/bar">
  <bar:Child attr="abcdefg"/>
</Parent>
```

The preceding example correctly describes the lexical representation of a `QName` in a schema. However, the value space is

care full

slightly different. Because the prefix is arbitrary and is only used as a shorthand notation, it has no value in defining a `QName`. Therefore, the value space of a `QName` in a schema contains only the local part and namespace URI. See Section 12.4.1 for more details.

3.4.1 Qualified Names as Values

The Schema Recommendation takes the notion of qualified names further than the definition in the Namespace Recommendation. It allows attribute and element values to be qualified names, and defines a `QName` datatype. The following example schema shows how this works. This schema does not use a default namespace, so every name that is part of a namespace will contain a prefix.

```
<xs:schema targetNamespace="http://www.example.com/foo"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:foo="http://www.example.com/foo"
    elementFormDefault="qualified"
    attributeFormDefault="unqualified"
    version="1.0">
    <xs:element name="ZipCode" type="foo:ZipCodeType">
            <xs:annotation>
                <xs:documentation>Declares an element of type ZipCodeType</
xs:documentation>
            </xs:annotation>
        </xs:element>
        <xs:simpleType name="ZipCodeType">
            <xs:annotation>
                <xs:documentation>Represents a ZipCode</xs:documentation>
            </xs:annotation>
            <xs:restriction base="xs:string">
                <xs:length value="5"/>
                <xs:pattern value="\d{5}"/>
            </xs:restriction>
        </xs:simpleType>
</xs:schema>
```

This schema defines a simple type, `ZipCodeType`. Because this schema has a target namespace of `http://www.example.com/foo` that is bound to the prefix '`foo`', all definitions in this schema, such as `ZipCodeType`, are in the target namespace. When you define an element to be of a certain named structure type, the `type` attribute in the defining `element` has a structure type of `QName`. As such, either its value must contain a prefix that is bound to a namespace that is in scope, or it must be in the scope of a default namespace.In our example, the `ZipCode` element has a type of `foo:ZipCodeType`. The prefix lets a schema processor correctly identify the type being used and also prevents naming collisions if other namespaces contain different `ZipCodeType` definitions.

There is currently some debate in the W3C as to whether a default namespace will apply to QNames that are part of attribute values. By allowing the use of default namespaces, you can decrease the number of characters within an attribute value. The XML Schema Working Group takes the position that default namespace processing is used for attributes with a datatype of `QName`. However, there currently isn't consensus on this within the W3C, so a final resolution has yet to be worked out. In the meantime, always use prefixed namespaces to ensure correct results.

3.5 Namespace Scoping

Namespaces have scoping rules that let you have a namespace in effect for only part of your document. A namespace declaration applies to the element where it is specified and to all elements contained within it. You can also override the binding of a prefix to a namespace URI by binding it to a different namespace URI, and you can turn off default namespace processing by setting the value of the `xmlns` attribute to an empty string. For example: Follow in Coding

```
<root xmlns="http://www.example.com/root">
  <inrootnamespace/>
  <child xmlns="http://www.example.com/child">
    <inchildnamespace/>
    <nonamespace xmlns="">
    * * * * *
    </nonamespace>
  </child>
</root>
```

APPROVED
By paramasivam at 11:29 am, May 15, 2008

In the preceding example, the `root` and `inrootnamespace` elements are in the `http://www.example.com/root` namespace. The `child` and `inchildnamespace` elements are in the `www.example.com/child` namespace. In addition, the `nonamespace` element and its subelements do not belong in any namespace. Prefixes work the same way. It is possible to redefine a prefix to be associated with a different namespace. However, this is not a good thing to do. Because a prefix is just an arbitrary character string, you should not reuse a prefix in a document. The document will be error-prone and hard to read.

3.6 XML Schema and Namespaces

You associate an XML Schema with a namespace by using the `targetNamespace` attribute. A single XML Schema document can support at most one XML Namespace. Also, you can compose an XML Schema from multiple XML Schema documents that all share the same target namespace. You can also import elements and types from other namespaces in your schema by using the `import` element. By importing schemas from other namespaces, you can compose schemas that combine multiple vocabularies. Chapter 7 covers use of the `import` element.

what about targetnamespace while import another schema doc

In XML schemas, structure types are identified by QNames. This means they either need to be prefixed, there needs to be an in scope default namespace, or there must be no target namespace for the schema. With default processing, you might use the following schema start-tag to associate all the elements you define using default namespace processing:

```
<xs:schema
targetNamespace=http://www.myserver.com/myName
xmlns="http://www.myserver.com/myName"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xml:lang="en">
```

When creating schemas, it is preferable not to set a default namespace and to always use prefixes. By doing this, you cannot accidentally associate a name with the wrong URI. Follow in Coding

The following example uses prefixes for all namespaces:

```
<xs:schema
```

```
targetNamespace=http://www.myserver.com/myName
xmlns:mn="http://www.myserver.com/myName"
xmlns:xs="http://www.w3.org/2001/XMLSchema"
xml:lang="en">
```

CONTENTS

CONTENTS

Chapter 6. Overview of an XML Schema Document

IN THIS CHAPTER

this chapter talks, whole xml schema in a single chapter in briefly

-paramasivam

An XML schema is an abstract model that constrains a corresponding XML instance. An XML schema document is an XML representation of an XML schema. More precisely, multiple XML schema documents might comprise an XML schema. This chapter introduces an XML schema document. The chapter also provides terms and concepts to discuss XML schemas and XML schema documents. The goal of this chapter is to provide enough background so you can select the correct component to place in an XML schema document. Details on attribute and content options of individual components are provided in Chapter 8, Chapter 9, Chapter 10, and Chapter 11, respectively.

Listing 6.1 provides an example of a very simple, yet complete XML schema document. Listing 6.2 provides a corresponding XML instance. Both listings make up the *compact* example. These examples are discussed in enough detail to instill familiarity with the construction of an XML schema document. The goal of the chapter is to portray enough about a schema so that much of this book makes sense—even if you do not read the remaining chapters sequentially. Because of the myriad combinations possible in a schema, the compact example does not cover all schema elements and attributes; some of the sections in this chapter refer to the thematic catalog example introduced in Chapter 1 and presented in Appendix C.

Listing 6.1 A Compact XML Schema Document (`compact.xsd`)

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

<xsd:complexType name="globalComplexType"
                 abstract="true">
```

```
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A global complex type.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence>
        <xsd:element name="basicTokenElement" type="xsd:token"/>
        <xsd:element name="sixToOneHundred"
                     minOccurs="0"
                     maxOccurs="unbounded">
            <xsd:simpleType>
                <xsd:restriction base="xsd:positiveInteger">
                    <xsd:minExclusive value="5"/>
                    <xsd:maxInclusive value="100"/>
                </xsd:restriction>
            </xsd:simpleType>
        </xsd:element>
        <xsd:element name="tokenElement">
            <xsd:complexType>
                <xsd:simpleContent>
                    <xsd:extension base="xsd:token">
                        <xsd:attribute name="tokenAttribute"
                                       type="xsd:token"/>
                    </xsd:extension>
                </xsd:simpleContent>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="encompassingElement">
    <xsd:complexType>
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                A local complex type.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:complexContent>
            <xsd:extension base="globalComplexType">
                <xsd:sequence>
                    <xsd:element
                         name="emptyContentWithAttribute">
                        <xsd:complexType>
                            <xsd:attribute
                                 name="decimalAttribute"
```

this must be complex type. cannot use any built-in type

```
                                    type="xsd:decimal"/>
                            </xsd:complexType>
                        </xsd:element>
                    </xsd:sequence>
                </xsd:extension>
            </xsd:complexContent>
        </xsd:complexType>
</xsd:element>

</xsd:schema>
```

Listing 6.2 portrays an XML instance that corresponds to the XML schema document portrayed in Listing 6.1.

Listing 6.2 A Compact XML Instance (`compact.xml`)

```
<encompassingElement
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:noNamespaceSchemaLocation=
  "http://www.XMLSchemaReference.com/examples/Ch06/compact.xsd">

    <basicTokenElement>Element Value</basicTokenElement>
    <sixToOneHundred>50</sixToOneHundred>
    <sixToOneHundred>22</sixToOneHundred>
    <sixToOneHundred>87</sixToOneHundred>
    <sixToOneHundred>6</sixToOneHundred>
    <tokenElement tokenAttribute="aValue">eValue</tokenElement>
    <emptyContentWithAttribute decimalAttribute="1000"/>

</encompassingElement>
```

6.1 The Enclosing `schema` Element

Each XML schema document has exactly one `schema` element. The attributes of the `schema` element specify various defaults. Some of these attributes apply to namespaces. The content of the `schema` element ultimately specifies the valid set of elements and attributes, as well as the valid ranges for the element values and attribute values, permissible in a corresponding XML instance. The XML schema document disperses the specification of these elements and attributes across element types, attribute types, simple types, complex types, and other specific schema components.

Chapter 7 details the attributes and content options for a `schema` element. Chapter 7 also covers some miscellaneous content options (`annotation`, `import`, `include`, `notation`, and `redefine`). Chapters 8 through 11 cover the remainder of the specific elements that are valid content options for a `schema` element. Chapters 8 through11 cover `element`, `attribute`, `simpleType`, and `complexType`, respectively. Chapter 15 covers the schema components represented by the elements in schema documents.

6.2 Namespaces

Specifying namespaces, default namespaces, and a target namespace takes only a few lines of code in an XML schema document, yet namespaces seem to be one of the most confusing aspects when learning about XML schemas. This section presents an overview of namespaces, including a discussion of how an XML schema applies namespaces. Chapter 3 presents a thorough discussion of namespaces.

With the exception of anonymous schema components (see Section 1.3), all schema components have a name. Each name nominally belongs to a namespace. Many schemas explicitly have no namespace. A schema specifies the value of a namespace as a URI, although the URI does not have to exist (or be accessible to the XML validator).

Although all the components nominally belong to the same namespace, each namespace has multiple partitions. Each component type has a unique partition, or *symbol space*. For example, a schema that has both a `fred` element type and a `fred` simple type is perfectly valid. This partitioning has two exceptions:

- Simple types and complex types share the same symbol space.
- Locally scoped names have no namespace. Furthermore, distinct enclosing elements can specify local elements with the same name.

attribute, element, group,type has separate symbol space. so, name of components can be shared across different symbol space. for example the same name can be used in attribute and type.

Although trivial to imp XML schema is determining the target and default nam ot contain any namespace information other than where to loc documents evolve to include namespaces and target . A schema should absolutely specify a target namespace, ho et or in any public schema that a user might integrate with ot ive, a public schema document should not specify a default n multiple schemas with multiple namespaces.

PREFIX, which associate schema component to a particular name space.

Often an XM ment specifies a namespace with the `xmlns` attribute of a `schema` element. The following code snippet associates the prefix '`xsd`' with the namespace http://www.w3.org/2001/XMLSchema:

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

Typically, a qualifier is several characters. Many of the examples in this book specify prefixes such as '`xsd`', '`cat`', and '`addr`'. A *qualified name*, which is a prefix in conjunction with a local name, identifies a component. An *unqualified name* does not have an associated namespace. A qualified name has the form `namespace:name`. For example, `xsd:token` is a reference to the built-in datatype `token` in the schema for schemas namespace specified by the prefix `xsd`.

6.2.1 The Default Namespace

A schema may reference many namespaces. A schema may optionally specify a default namespace. When a schema specifies a default namespace, the default namespace is implicit for any reference to a global component that does not

include a namespace qualification. For example if the default namespace is the URI associated with the prefix `xsd`, the name `token` implies the qualified name `xsd:token`. The default namespace applies only to references within an XML schema; the default namespace has no impact on XML instances.

The `xmlns` attribute of the `schema` element—without a namespace declaration such as `xsd` in the previous example—specifies a default namespace:

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

Tip

Although not imperative, a public schema document should not specify a default namespace if the schema provides functionality via the Internet or in any public schema that a user might integrate other public or proprietary schemas. A public schema with a default namespace can lead to confusion when integrating multiple schemas with multiple namespaces.

6.2.2 The Target Namespace

A schema may optionally specify a target namespace. The target namespace specifies the namespace for all the schema components described in the XML schema. In other words, the names of all components must be unique within the appropriate symbol space in the target namespace. When the schema does not specify a target namespace, each component has explicitly no namespace.

this is better if it is going to include in another schema

The `targetNamespace` attribute of the `schema` element specifies a target namespace:

```
targetNamespace=
    "http://www.XMLSchemaReference.com/examples/theme/catalog"
```

Tip

A schema should absolutely specify a target namespace if the schema provides functionality via the Internet or in any public schema that a user might integrate other public or proprietary schemas. A public schema with explicitly no namespace is much more likely to have naming collisions with other schemas that also have no target namespace.

6.2.3 Namespaces and the XML Instance

The elements and attributes in a corresponding XML instance must reference components by name. This name is relative to the target namespace specified in the corresponding XML schema document. The XML schema document may explicitly specify no target namespace.

When the XML schema document does specify a target namespace, the XML instance can specify a qualified name to identify components. There are alternatives to qualifying every element and attribute name. The `schema` element

may specify the attributes `elementFormDefault` and `attributeFormDefault`. When these attributes have a value of `unqualified` (the default value), the XML instance is globally relieved of incessantly qualifying references to global ele-ment types and attribute types, respectively. Instead, the XML validator assumes the target namespace. In addition to the `schema` element attributes, either global element types or global attribute types may specify a `form` attribute with the same effect. The only difference is that the name qualification—or lack thereof—applies only to the specific element type or attribute type.

6.3 Scope

Like many programming languages, scope may limit one component from referencing another. Element types, attribute types, and most of the other kinds of components in a schema may be global to the schema or local to another component.

6.3.1 Global Components

With some exceptions, one global component might reference global components in the same namespace or even in multiple namespaces. Global element types and attribute types also have the distinction of being generally available for use in an XML instance.

6.3.1.1 Referencing Global Components from an XML Schema Document

In general, an element in an XML schema document can reference a named global component—typically with a `type` or `ref` attribute. Deriving complex types or simple types or associating an element type or attribute type with a structure type requires a `type` attribute in the representing document. For example, the following represents an element type by associating the name '`phoneNumber`' with the (global) built-in `string` datatype:

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

A reference to a global element type or global attribute type requires a `ref` attribute. For example, the following is a reference to the previously defined global `phoneNumber` element type:

```
<xsd:element ref="phoneNumber"/>
```

Some components may restrict their reference within a schema via blocking. This type of blocking occurs when the component's representation has a `final` attribute with an appropriate value.

6.3.1.2 Referencing Global Components from an XML Instance

An element or an attribute in an XML instance claims to be an instance of a particular element or attribute type by using that type's name: an element, by using that name as its *type name*; an attribute by using that name as its name. For example, the following element has '`phoneNumber`' as its *type name*, thereby referencing the `phoneNumber` element type and claiming to be an instance of that class:

```
<phoneNumber>503-555-1212</phoneNumber>
```

Finally, an element may explicitly assert that it is to validate against a global simple or complex type with the `xsi:type` attribute:

```
<amount xsi:type="salePriceType">123.45</amount>
```

The type must be a valid derivation of the structure type of the referenced element type. In the preceding example, the structure type of `amount` is `dollarPriceType`. Furthermore, `salePriceType` derives from `dollarPriceType`. Both Chapter 10 and Chapter 11 have detailed discussions about deriving types. These chapters also cover how to invoke `xsi:type` in elements.

An XML instance may not reference a global element type or complex type if that type is not instantiable. The XML representation of a non-instantiable component has the `abstract` attribute set to `'true'`. Section 6.10 has more details about component instantiability.

Certain components can prohibit direct or indirect references from an XML instance via blocking. This type of blocking occurs when the component's representation has a `block` attribute with an appropriate value (see the discussion on blocking in Section 8.2.3).

6.3.2 Local and Anonymous Components

A schema may reference any element type, attribute type, complex type, or simple type in the context, or scope, of another type. Element types and attribute types always have a name, because XML instances must be able to reference them using their names. *Anonymous* simple and complex types are unnamed local components. A schema may specify but not reference local components.

i dont understand

only simple type and complex type can be anonymous. none of other schema components.

6.4 Annotating Elements

Annotations are a great way to make notes about element types, attribute types, and almost any other element in an XML schema document. An annotation provides what a developer would typically call a "comment" to the XML schema document. Comments destined for developers are stored in a `documentation` subelement. `appinfo` subelements support automated processing of annotations—a discussion beyond the scope of this chapter.

Documentation in an XML schema document comes in two forms: annotations and comments. An annotation takes the form of an `annotation` element. The following annotation, which annotates the `schema` element, comes from the XML schema document `address.xsd`:

```
<xsd:annotation>
    <xsd:documentation xml:lang="en">
        This XML Schema Document describes a customer
        address in great detail.  In particular, this
        document illuminates all of the attributes and
        content options for elements and attributes.
    </xsd:documentation>
```

```
</xsd:annotation>
```

A real comment, on the other hand, is an XML comment. An XML comment looks like this:

```
<!-- This is a comment -->
```

Annotations have the advantage that they are preserved when creating a schema from schema documents. They provide a way to document any type of component. Any element, from the all-encompassing `schema` element to a local simple type nested in a local attribute type, can contain an `annotation`. A program can easily read an `annotation` and process the contents.

Tip

An `annotation` element is a great way to document individual elements in an XML schema document, such as `attribute`, `complexType`, `simpleType`, and many others.

An XML comment is a way to provide documentation that might apply across elements or, perhaps, within an element. Comments have no particular format (other than the start and end punctuation characters, of course) and are not particularly associated with an element. Annotations are appropriate when describing the behavior of an element; XML comments are appropriate for providing guidance to other schema document maintainers.

Almost any element in an XML schema document may offer an annotation by specifying a nested `annotation`:

```
<xsd:simpleType name="partNameType"
                final="list union"
                id="catalog.partName.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A part name can be almost anything.
            The name is a short description.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token"
                     id="pnt-rst">
        <xsd:minLength value="1"/>
        <xsd:maxLength value="40"/>
    </xsd:restriction>
</xsd:simpleType>
```

An annotation is part of the element. Furthermore, the XML Schema Recommendation even specifies when an annotation is valid in the XML representation of an element. See the various "Content Options" sections of this book for explanations of applicability of the `annotation` subelement to each type of element is a schema.

6.5 Constraining Elements

Perhaps the most important goal of the XML Schema Recommendation is to provide a language for constraining XML instances. A large part of this effort is constraining elements. An element type broadly specifies either simple content or complex content. An element type bound by simple content contains only a text value. This value may be constrained by built-in datatypes or other simple types derived from those datatypes. An element type whose content is complex might specify attribute types or nested element types (or both).

6.5.1 Simple Content

An element type that constrains an element to contain simple content places restrictions on the text value of the element. An element type that specifies simple content may specify attribute types as well.

6.5.1.1 Lexically Constrained Values

An element type lexically constrains the value of a corresponding element instance by specifying a simple type as its content type. The following two examples are elements whose structure types could be the built-in datatypes `token` and `decimal`, respectively:

```
<basicTokenElement>Element Value</basicTokenElement>
```

and

```
<sixToOneHundred>50</sixToOneHundred>
```

An element type representation may specify a built-in datatype or another global simple type with a `type` attribute:

```
<xsd:element name="basicTokenElement" type="xsd:token"/>
```

Alternatively, an element type representation may specify a local simple type with a `simpleType` element:

```
<xsd:element name="sixToOneHundred"
                    minOccurs="0"
                    maxOccurs="unbounded">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:minExclusive value="5"/>
            <xsd:maxInclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

6.5.1.2 Simple Content and Attribute Types

An element type that specifies simple content may also permit attribute types by specifying a complex type with

simple content. The following example is a `tokenElement` with a `tokenAttribute` attribute.

```
<tokenElement tokenAttribute="aValue">eValue</tokenElement>
```

The corresponding element type must specify a complex type constrained to simple content:

```
<xsd:element name="tokenElement">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:restriction base="xsd:token">
                <xsd:attribute name="tokenAttribute"
                               type="xsd:token"/>
            </xsd:restriction>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

The XML representation of an element type can nest a local complex type, as in the previous example, or the element type can reference a global complex type using the `type` attribute.

6.5.2 Complex Content

An element type that constrains an element instance to contain complex content may specify subelements, empty content, mixed content, element wildcards, or attributes.

6.5.2.1 Nested Elements

Many XML instances require nested elements. Nested elements provide a mechanism for describing a hierarchy of data. An element type must specify a complex type that specifies a model group. The model group specifies the nested elements. Listing 6.2 is an example of an `encompassingElement`, which contains the nested elements `basicTokenElement`, `sixToOneHundred`, `elementWithAttribute`, and `emptyContentWithAttribute`.

6.5.2.2 Empty Content

An element that has no value—as distinct from having a value that is an empty string—has *empty content*. Typically, although not necessarily, an element type that specifies empty content also specifies one or more attribute types. The following example is an element `emptyContentWithAttribute` with a single attribute, `decimalAttribute`:

```
<emptyContentWithAttribute decimalAttribute="1000"/>
```

The corresponding element type specifies a complex type that does not provide content:

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

```
    <xsd:complexType>
        <xsd:attribute name="decimalAttribute"
                       type="xsd:decimal"/>
    </xsd:complexType>
</xsd:element>
```

Element types that would otherwise require a value may specify that the corresponding element is nillable. The complete XML representation of `phoneNumber`, from `address.xsd`, permits a phone number to have empty content:

```
<xsd:element name="phoneNumber"
             type="xsd:string"
             nillable="true"/>
```

to make nillable we have to use nillable attribute if it is non-complexType,

to make a complex type a empty content that is nullable, simply dont provide any element

The corresponding element explicitly has empty content:

```
<phoneNumber xsi:nil="true"/>
```

6.5.2.3 Mixed Content

An element may have *mixed content*. An element with mixed content may contains text, usually interspersed with nested elements. (The text it not constrained by a simple type.) The compact example (Listing 6.1) does not demonstrate mixed content. However, the thematic catalog example specifies mixed content for an instance of `description`:

```
<description>
    This is made up of both
    <partList>UX002 UX003</partList>
    pieces which makes it
    better than
    <partList>ASM2000</partList>
    and better than the competition.
</description>
```

An element type must specify a complex type that permits mixed content. Either the `complexType` or the enclosed `complexContent` must specify a `mixed` attribute. The structure type that corresponds to the `description` element type specifies mixed content:

```
<xsd:complexType name="catalogEntryDescriptionType"
                 mixed="true"
                 id="catalogEntryDescriptionType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Allow the description of a part
            to include part number references.
```

```
            The "catalogEntryDescriptionType"
            is a good example of a complex type
            with "mixed" content.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="partList" type="partNumberListType"/>
    </xsd:sequence>
</xsd:complexType>
```

For completeness, the following is the `description` element type:

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

Chapter 11 discusses how to implement mixed content.

6.5.2.4 Element Wildcards

An XML schema may specify element wildcards with the `any` element. An element wildcard is a mechanism for specifying a set of namespaces from which the corresponding XML instance selects element types. The compact example does not demonstrate element wildcards. Chapters 8 and 15 covers element wildcards.

6.5.2.5 Complex Content and Attribute Types

Like simple content, complex content may optionally specify attribute types.

6.6 Constraining Attributes

Constraining an attribute is similar to constraining an element. Compared to element types, attribute types are severely limited: An attribute type may only specify simple types.

6.6.1 Simple Attribute Values

An element may have zero or more attributes. The following example is a `tokenElement` element with a single `tokenAttribute` attribute:

```
<tokenElement tokenAttribute="aValue">eValue</tokenElement>
```

The XML representation of the `tokenAttribute` attribute type looks very much like the XML representation of an element type:

```
<xsd:attribute name="tokenAttribute" type="xsd:token"/>
```

The structure type of an element type or a complex type may specify a reference to a global attribute type or a complete local attribute type.

6.6.2 Named Attribute-use Groups

A schema may specify attributes as a group instead of as a list of individual attribute types. A *named attribute-use group* is little more than a set of attribute types. Like other global components, the XML schema may reference named attribute-use groups in many places.

The XML representation of a named attribute-use group is the `attributeGroup` element, which might look like the following, for `saleAttributeGroup`:

```
<xsd:attributeGroup name="saleAttributeGroup"
                    id="pricing.sale.ag">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything that is on sale (or free,
            which is a type of sale) must
            have an authorization defined.
            This is someone's name,
            initials, ID, etc.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="employeeAuthorization" type="xsd:token"/>
    <xsd:attribute name="managerAuthorization" type="xsd:token"/>
</xsd:attributeGroup>
```

When an element type or complex type specifies the `saleAttributeGroup`, the effect is the same as specifying both `employeeAuthorization` and `managerAuthorization` at the same time.

Chapters 9 and 15 cover attribute groups.

6.6.3 Attribute Wildcards

An XML schema may specify attribute wildcards with the `anyAttribute` element. An attribute wildcard is a mechanism for specifying a set of namespaces from which the corresponding XML instance selects attribute types. The compact example does not demonstrate attribute wildcards. Chapters 9 and 15 cover attribute wildcards.

6.7 Simple Types

A simple type specifies a value range. An element type or attribute type may specify a simple type to impose a range of values on a corresponding XML instance. In the simplest case, a built-in datatype imposes the value range. Frequently, a simple type derived from a built-in datatype provides further constraints. A *constraining facet* on the

derived simple type specifies a restriction of the value range. Multiple constraining facets may apply to a single simple type.

6.7.1 Built-in Datatypes

The built-in datatypes fall mostly into three categories: strings, numbers, and dates. Each of these categories has multiple specific datatypes. For example, a date might or might not include the time. Alternatively, perhaps the only interesting part of a date is the year. Similarly, numbers can be broken into integer and floating point, positive and negative, and other subcategories. An element type, attribute type, simple type, or complex type may reference a built-in datatype. A trivial example is the XML representation of `basicTokenElement`, whose structure type is the built-in `token` datatype described in Section 6.5.1.1:

```
<xsd:element name="basicTokenElement" type="xsd:token"/>
```

Similarly, the `decimalAttribute` attribute type specifies a decimal number:

```
<xsd:attribute name="decimalAttribute" type="xsd:decimal"/>
```

Chapter 12 discusses all built-in datatypes.

6.7.2 Deriving Simple Types by Restriction

simply can use only RESTRICTION but not EXTENSTION

A built-in datatype does not always provide appropriate constraints. A custom simple type creates an appropriate constraint by restricting a built-in datatype or another custom simple type. The XML representation of the element type `sixToOneHundred` extracted from Listing 6.1 specifies a simple type that constrains the built-in datatype `positiveInteger` to a number between 6 and 100:

```
<xsd:element name="sixToOneHundred"
             minOccurs="0"
             maxOccurs="unbounded">
    <xsd:simpleType>
        <xsd:restriction base="xsd:positiveInteger">
            <xsd:minExclusive value="5"/>
            <xsd:maxInclusive value="100"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
```

Note that the value range is *exclusive* of the value 5 and *inclusive* of the value 100.

6.7.3 Constraining Facets

A simple type applies zero or more constraining facets during derivation. The constraining facets limit the range of values, as demonstrated by the use of `minExclusive` and `maxInclusive` in the element type

sixToOneHundred described in the previous example. Table 6.1 introduces all the constraining facets. This introduction includes a brief description of each constraining facet.

Table 6.1. The Constraining Facets

Element	*Description*
enumeration	The value of an enumeration constraining facet is a set of specific valid values for a corresponding element. Each member of the set is represented by a separate enumeration element.
fractionDigits	The value of a fractionDigits constraining facet limits the number of digits after the decimal point required to represent a decimal value.
length	The value of a length constraining facet specifies the length of a string in characters. The value of a length constraining facet may also specify the length of a list (that is, the number of items in the list). For other datatypes, the units (characters, items) may be something different, appropriate to the datatype.
maxExclusive	The value of a maxExclusive constraining facet specifies an upper bound on a numeric value. This boundary excludes the value specified.
maxInclusive	The value of a maxInclusive constraining facet specifies an upper bound on a numeric value. This boundary includes the value specified.
maxLength	The value of a maxLength constraining facet specifies the maximum number of characters in a string or the maximum number of items in a list (or the maximum "something" appropriate to the datatype constrained).
minExclusive	The value of a minExclusive constraining facet specifies a lower bound on a numeric value. This boundary excludes the value specified.
minInclusive	The value of a minInclusive constraining facet specifies a lower bound on a numeric value. This boundary includes the value specified.
minLength	The value of a minLength constraining facet specifies the minimum number of characters in a string or the minimum number of items in a list.
pattern	The value of a pattern constraining facet is a regular expression used to validate a character string.
totalDigits	The value of a totalDigits constraining facet limits the total number of digits required to represent a decimal value.
whiteSpace	The value of a whiteSpace constraining facet provides for various kinds of normalization of spaces, carriage returns, and line feeds when determining the value of an instance of a simple type.

Each constraining facet applies to a subset of the datatypes. Chapter 12 discusses all the constraining facets.

6.7.4 Lists

A simple type may specify a list of values. A simple type list derives from yet another simple type that constrains each value in the list. The values in a list are space-delimited. The thematic catalog example specifies a list of part numbers. An XML instance of this part number list might look like the following:

```
<partList>UX002 UX003</partList>
```

Because of the complexity of creating a list, discussion about creating the XML representation of a list simple type is left to Section 10.7.

6.7.5 Unions

A simple type may specify a value space and lexical space in terms of a union of other simple types. A simple type may form a union from fundamentally different simple types. The thematic catalog example portrays the simple type `assemblyPartStatusType` that specifies a union of a date, an integer, and a string. A corresponding XML instance must be an instance of one of these three valid value classes.

Because of the complexity of creating a union, discussion about creating the XML representation of a union simple type is left to Section 10.8.

6.8 Complex Types

A complex type specifies the possible content of an element. A complex type specifies subelements, empty content, or mixed content, and attributes. A complex type has two flavors: a complex type that specifies simple content and a complex type that specifies complex content.

6.8.1 Simple Content

Complex types that specify simple content can further restrict value ranges with functionality identical to simple type restriction. Complex types with simple content provide the following functionality:

- Restricting value ranges (identical functionality to simple type restriction)
- Adding attribute types to a simple type or to a base complex type that specifies simple content
- Modifying or removing attribute types from a base complex type that specifies simple content

Section 11.7 covers simple content.

6.8.2 Complex Content

Complex types that specify complex content describe the content range for an element instance. Complex content provides the following functionality:

- The allowable subelement structure
- Specifying empty content
- Specifying mixed content, which includes the structure of nested elements
- Adding, modifying, or removing attribute types from a base complex type

Section 11.8 covers complex content.

6.8.3 Deriving Complex Types by Extension

A derived complex type may extend a base complex type by adding any of the following:

- Content (individual element types or via entire model groups)
- Attribute types (or attribute-use groups, or attribute wildcards)

Listing 6.3 demonstrates the derived complex type `assemblyCatalogEntryType` that extends `baseAssemblyCatalogEntryType` by adding two element types: `partList` and `status`.

Listing 6.3 Extending a Complex Type

```
<xsd:complexType name="baseAssemblyCatalogEntryType"
                 abstract="true"
                 block="#all"
                 final="restriction"
                 id="baseAssemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An assembled item is similar to the
            other catalog entries.  The part number
            is restricted to an assembly number.
            In addition, there may be no options.
            Finally, a part list is also needed.
            Note that the "includedQuantity" has
            a default of one, but can be overridden
            in instances.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType"
                         id="bacet.rst">
            <xsd:sequence>
                <xsd:element ref="assemblyID"/>
                <xsd:element name="partName" type="partNameType"/>
                <xsd:element name="partNumber"
                             type="assemblyPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"
                             minOccurs="0"
                             maxOccurs="0"/>
                <xsd:element name="description"
```

```
                        type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                        type="xsd:positiveInteger"
                        default="1"/>
                <xsd:element name="customerReview"
                        type="customerReviewType"
                        minOccurs="0"
                        maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                        type="categoryType"
                        fixed="assembly"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="assemblyCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="assemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The actual definition of an assembly,
            including the contained parts.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:extension base="baseAssemblyCatalogEntryType"
                       id="acet.ext">
            <xsd:sequence>
                <xsd:element name="partList" type="partNumberListType"/>
                <xsd:element name="status" type="assemblyPartStatusType"/
>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
```

Section 11.9 covers extending complex types.

6.8.4 Deriving Complex Types by Restriction

Many complex types derive from other complex types via restriction. A restriction is a reduction in the range of values for specific element types or attribute types—or even the elimination of element types and attribute types.

Technically, the restrictions modify the "use" (as in `minOccurs` or `maxOccurs`), or the value specified by structure types. The XML representation of a restricted complex type must have element types and attribute types derived from those specified by the base complex type. A derived complex type provides the following functionality:

- Altering attributes of an element type or attribute type
- Substituting a derivation of a structure type for any structure type
- Substituting an element in the context of a substitution group
- Restricting value ranges of simple types

altering type of element or attribute is not possible. since subtype must have subset of super type. (i checked)

Derived complex types often alter the `minOccurs` and `maxOccurs` attributes of an element type: This may result in prohibiting an element in a corresponding XML instance. Similarly, a derived complex type frequently alters the `use` attribute of attribute types: ~~This may result in prohibiting an attribute in a corresponding XML instance~~.

Note

~~Complex type specifications which use the shorthand notation (neither~~ `restriction` ~~nor~~ `extension` ~~are specified) are implicitly restrictions of~~ `anyType`. (~~*Every* complex type other than~~ `anyType` ~~is *ultimately* a restriction of~~ `anyType`.~~)~~

Listing 6.4, whose source is the thematic catalog example, shows that `bulkCatalogEntryType` is a restriction of `baseCatalogEntryType`. Note that both of these complex types have exactly the same nesting of element types (`sequenceID`, `partName`, `partNumber`, `partOption`, `description`, `priceGroup`, `includedQuantity`, and `customerReview`) as well as the same `category` attribute type.

Listing 6.4 Restricting a Complex Type

```
<xsd:complexType name="baseCatalogEntryType"
                 abstract="true"
                 id="baseCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A catalog entry must have:
                * A database ID
                * Part Name
                * Part Number
                * Options available
                * Description
                * Price
                * Included Quantity when ordering
                  one item.
            The "baseCatalogEntryType" is
            non-instantiable:  a derived type must
            be created before a catalog
            entry can be instantiated.
```

```
                -- Shorthand Notation--
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence id="bacet-seq">
        <xsd:element ref="sequenceID"/>
        <xsd:element name="partName" type="partNameType"/>
        <xsd:element name="partNumber" type="partNumberType"/>
        <xsd:element name="partOption" type="partOptionType"/>
        <xsd:element name="description"
                     type="catalogEntryDescriptionType"/>
        <xsd:group ref="priceGroup"/>
        <xsd:element name="includedQuantity"
                     type="xsd:positiveInteger"/>
        <xsd:element name="customerReview"
                     type="customerReviewType"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="category"
                   type="categoryType"
                   use="required"/>
</xsd:complexType>

<xsd:complexType name="bulkCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="bulkCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A bulk item is just like any
            other, except that the part
            number is restricted to a
            bulk part number.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType">
            <xsd:sequence>
                <xsd:element ref="bulkID"/>
                <xsd:element name="partName" type="partNameType"/>
                <xsd:element name="partNumber" type="bulkPartNumberType"/
>
                <xsd:element name="partOption" type="partOptionType"/>
                <xsd:element name="description"
                             type="catalogEntryDescriptionType"/>
```

```
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                           type="categoryType"
                           fixed="bulk"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

In fact, `bulkCatalogEntryType` substitutes `bulkID` for `sequenceID`. Section 6.10 covers this substitution.

Section 11.10 covers restricting complex types.

6.8.5 Blocking Complex Types

The XML Schema Recommendation provides for blocking restriction or extension of complex types. Blocking can prevent derivations of a complex type. Blocking can also prevent substituting derived types in an XML instance via `xsi:type`. Because of the complexity of blocking, all further discussion of blocking is deferred to Chapter 11.

6.9 Model Groups

it can use only with complex type

A model group specifies a pattern for element type instances. There are three kinds of model groups:

- An *all* model group requires instances of all members of a set of element types or other groups, in no particular order. (Each member may have occurrence restrictions.)
- A *choice* model group requires an instance of one member of a set of element types or other groups. (Each member may have occurrence restrictions.)
- A *sequence* model group requires instances of all members of a set of element types or other groups, in a specified order. (Each member may have occurrence restrictions.)

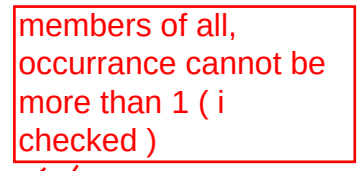
A *named model group* specifies any one of the other model groups.

6.9.1 The All Model Group

A complex type with complex content specifies an unordered set of element types with an all model group. The XML representation of `partOptionType` in Listing 6.5 specifies an unordered set of element types. Excluding the `minOccurs` attribute (which in this case specifies that the corresponding element instance is optional), a

corresponding XML instance must include a `color` and a `size`, although these can appear in either order.

Listing 6.5 An All Model Group




```
<xsd:complexType name="partOptionType"
                 block="#all"
                 final="#all"
                 id="partOptionType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Appropriate parts can have a color,
            a size, or both.  Note that the use
            of the "all" element indicates that
            the "color" and "size" are unordered.
            That is, they can appear in either
            order.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:all id="pot.all">
        <xsd:element name="color"
                     type="colorOptionType"
                     minOccurs="0"
                     maxOccurs="1"/>
        <xsd:element name="size"
                     type="sizeOptionType"
                     minOccurs="0"
                     maxOccurs="1"/>
    </xsd:all>
</xsd:complexType>
```

Section 11.11 covers the all model group.

6.9.2 The Choice Model Group

A complex type with complex content specifies the selection of one element type from a set of element types with a choice model group. Listing 6.6 portrays a choice of pricing schemes. A corresponding XML instance must select exactly one of the pricing schemes.

Listing 6.6 A Choice Model Group

```
<xsd:group name="priceGroup">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A price is any one of the following:
```

```
                * Full Price (with amount)
                * Sale Price (with amount and authorization)
                * Clearance Price (with amount and authorization)
                * Free (with authorization)
            </xsd:documentation>
        </xsd:annotation>
        <xsd:choice id="pg.choice">
            <xsd:element name="fullPrice"
                         type="fullPriceType"/>
            <xsd:element name="salePrice"
                         type="salePriceType"/>
            <xsd:element name="clearancePrice"
                         type="clearancePriceType"/>
            <xsd:element name="freePrice" type="freePriceType"/>
        </xsd:choice>
</xsd:group>
```

Section 11.12 covers the choice model group.

6.9.3 The Sequence Model Group

A complex type with complex content specifies an ordered set of element types with a sequence model group. The XML representation of `encompassingElement` in Listing 6.1 specifies an ordered set of element types. This ordering ensures that a corresponding XML instance always contains nested elements in the same order. This order may enhance readability or processing.

Section 11.13 covers the sequence model group.

6.9.4 The Named Model Group

A named model group takes a model group, names it, and makes it global so that it can be referenced; sequence, all, and choice model groups are always local to a complex type or a named model group. The main purpose of the named model group is to provide a reusable model group. `priceGroup` from Listing 6.5 is global. Many element types in the thematic catalog example require a price; these element types reference the global `priceGroup`.

Section 11.14 covers the named model group.

6.10 Substitution Groups

it is like RunTime poly morphism I in java

A substitution group consists of a set of element types. An element type associates itself with a substitution group by specifying a `substitutionGroup` attribute. The value of that attribute indicates that the new element type is a valid substitution for the referenced element type. The substitutions occur in complex type derivations or in an XML instance. Listing 6.7 is the `sequenceID` substitution group.

substitution group apply for ELEMENT a kind of inheritance
extension/ restriction apply for TYPE inheritance.

Listing 6.7 A Substitution Group

```
<xsd:element name="sequenceID"
             type="sequenceIDType"
             abstract="true">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element type is
            non-instantiable:  the element
            must be replaced by a substitution
            group in either a derived type or
            an instance.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="unitID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for unit items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="bulkID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for bulk items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="assemblyID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
```

it act as super class. its value is overridden by sub-class element. But here no explicit extend / restriction. it is achieved by substutitionGroup attribute

in substitutionGroup, the replacing element name is different.

Even it is not COMPLEX TYPE

then, what is difference between CHOICE and SUBSTITIONGroup ????????

type of base element

NAME OF base element

```
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for assembled items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
```

In the context of the thematic catalog example, there are various IDs. A base complex type requires a `sequenceID` attribute. Derived complex types substitute a specific ID, such as a `bulkID`. An element type can block substitution in derived complex types or in XML instances.

block attribute can be used only for avoid substitution group

Chapter 8 covers substitution groups as well as blocking substitutions.

6.11 Instantiability

Not all element types are instantiable in an XML instance. Element types may directly prohibit the existence of corresponding instances. Similarly, a complex type functioning as the structure type of an element type may indirectly prohibit the existence of corresponding instances. This section illuminates restrictions on instantiability of element types, attribute types, simple types, and complex types.

6.11.1 Element Type Instantiability

An element type is *explicitly* non-instantiable when the element type's representation specifies the `abstract` attribute with a value of '`true`'. An element type may be *implicitly* non-instantiable due to its structure type being explicitly or implicitly non-instantiable.

implicitly mean ---> if a element associated with a complex type and that complex type has abstract = true

Section 8.2 discusses the instantiability of element types.

6.11.2 Attribute Type Instantiability

An attribute type cannot derive another attribute type. Similarly, an attribute cannot be abstract. Therefore, all attribute types are instantiable.

6.11.3 Simple Type Instantiability

The XML Schema Recommendation does not yet permit non-instantiable simple types despite the fact that the XML schema writer might want to specify a non-instantiable base simple type. It is possible—we hope, likely—that the capability might be added in a future release of the Schema Recommendation.

6.11.4 Complex Type Instantiability

Complex types are *explicitly* non-instantiable when the complex type's representation specfies the `abstract` attribute with a value of '`true`'. A complex type may be *implicitly* non-instantiable due to enclosed element types (that is, element types in the content model) being implicitly or explicitly non-instantiable.

Section 11.5 covers the instantiability of complex types.

implicitly mean ---> if a complex type associated with a element and that element has abstract = true

6.12 Identity Constraint Definitions

An XML schema generally validates the structure of an XML instance in terms of element nesting and attributes. The XML schema also validates content primarily in the form of simple types. Identity constraints provide an additional mechanism to validate contents. The three types of elements for creating identity constraints are `unique`, `key`, and `keyref`.

A `unique` element specifies that repeated elements are unique given the values of one or a combination of keys. For example, Listing 6.8 demonstrates that a `partNumber` can appear only once with a particular value in the entire `completePartList`.

Listing 6.8 Unique Part Numbers (sdDemo.xsd)

```
<xsd:element name="completePartList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="part"
                         minOccurs="0"
                         maxOccurs="unbounded">
                <xsd:complexType>
                    <xsd:attribute name="partNumber"
                                   type="xsd:string"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>

<xsd:unique name="uniquePartNumbers">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                Ensure unique part numbers
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="part"/>
        <xsd:field xpath="@partNumber"/>
    </xsd:unique>

</xsd:element>
```

The `key` and `unique` identity constraints interact with the `keyref` identity constraints. The set of keys specified by `keyref` must exist in the set of keys specified by `key` or `unique`. This functionality parallels foreign keys in databases.

Chapter 13 provides many examples and detailed explanations of identity constraints.

6.13 Notations

A notation component in an XML schema supports the XML notations documented in the XML Recommendation. They constitute information to be simply passed on to the application.

> **Note**
>
> The XML Recommendation states that "XML processors must provide applications with the name and external identifier(s) of any notation declared and referred to in an attribute value, attribute definition, or entity declaration. They may additionally resolve the external identifier into the system identifier, file name, or other information needed to allow the application to call a processor for data in the notation described."

The value of an element or attribute in an XML instance may refer to a notation. The corresponding element type or attribute type must have a structure type that derives from the built-in `NOTATION`.

Section 7.10 covers notations.

notation is just data type. not a element ??

6.14 Imports and Includes

An XML schema document may reference components outside the document by importing or including these components from another XML schema document. The `include` element adds components to the current schema. In particular, the `include` element provides a mechanism to assemble an XML schema from a set of XML schema documents. An `import` element, on the other hand, permits a schema to include components in a different namespace.

Sections 7.8 and 7.9 provide details and examples of how to use `include` and `import`, respectively.

6.15 Locating XML Schemas and XML Schema Components

The most common mechanism for locating an XML schema document is a complete URL. The catalog schema, for example, is located at the following address:

```
xmlns:cat=
 "http://www.XMLSchemaReference.com/examples/theme/catalog.xsd"
```

An XML schema may specify a component via an XML fragment (see the W3C XML Fragment Interchange Recommendation at http://www.w3.org/TR/xml-fragment.html).

An XML schema may also specify components with an XPointer (see the W3C XML Pointer Language Recommendation at http://www.w3.org/TR/WD-xptr).

6.16 Schema Element IDs

it is with XPath or Fragment identifier

In an effort to demonstrate all the features offered by the XML Schema Recommendation, many of the examples in this book specify a unique ID for each schema component. Specifically, the elements have an `id` attribute. Each ID must be unique across the entire schema, not just within an XML schema document. Unless you are using the IDs for something meaningful, such as fragment identifiers, you might prefer not to specify IDs. The IDs tend to clutter the XML schema document.

Chapter 7. Creating an XML Schema Document

IN THIS CHAPTER

26 Pages

Every XML schema document contains a `schema` element that is the document element, which encloses all other elements. This chapter covers the attribute options and content options of a `schema` element. This chapter also discusses a few elements that pertain to XML schemas in general: `annotation`, `import`, `include`, `notation`, and `redefine`.

7.1 A Simple XML Schema Document Example

Listing 7.1 is a trivial XML schema document that specifies only an element type whose structure type is the built-in `string` datatype.

Listing 7.1 A Trivial XML Schema Document (`simpleSchema.xsd`)

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

A few features to point out about the preceding trivial schema:

- The only attribute of the `schema` element—`xmlns:xsd`—specifies the namespace http://www.w3.org/2001/XMLSchema. Furthermore, the `xmlns:xsd` attribute specifies the corresponding namespace prefix, '`xsd`'. The specification of the XML schema namespace implies a corresponding XML schema document `XMLSchema.xsd`, which is the Schema for Schemas.
- `xmlns:xsd` is not a `schema` attribute per se; the `xmlns:xsd` attribute applies to any XML element.
- There is explicitly no target namespace. Note that in the subsequent example, a namespace prefix does not precede the element name '`tokenElement`'.

Given the XML schema document in Listing 7.1, the following is a valid XML instance:

```
<stringElement
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation=
   "http://www.XMLSchemaReference.com/examples/Ch07/notation.xsd
>A String Value</stringElement>
```

Note that there is exactly one element in the XML instance: a `stringElement`. The value of that `stringElement` is `'A String Value'`.

7.2 A `schema` Element with Every Attribute

Section 7.4.1 discusses all the possible XML schema attributes of the `schema` element. Listing 7.2 is a complete XML schema in which the `schema` element has all possible attributes. Many of the ensuing explanations in Section 7.4.1 refer to Listing 7.2.

Listing 7.2 A `schema` Element with Every Attribute (`fullFeaturedSchema.xsd`)

```
<xsd:schema
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ffs="http://www.XMLSchemaReference.com/examples/Ch07/
fullFeaturedSchema"
    xmlns:ra="http://www.XMLSchemaReference.com/examples/Ch07/randomAttributes"
    xmlns="http://www.XMLSchemaReference.com/examples/Ch07/fullFeaturedSchema"
    targetNamespace="http://www.XMLSchemaReference.com/examples/Ch07/
fullFeaturedSchema"
    elementFormDefault="unqualified"
    attributeFormDefault="unqualified"
    id="Full-Featured-Schema"
    blockDefault="#all"
    finalDefault="restriction"
    version="FFS:1"
    xml:lang="en-US">

<xsd:import
    namespace="http://www.XMLSchemaReference.com/examples/Ch07/
randomAttributes"
    schemaLocation="http://www.XMLSchemaReference.com/examples/Ch07/
randomAttributes.xsd"/>
<xsd:include
    schemaLocation="http://www.XMLSchemaReference.com/examples/Ch07/ffs_include.
xsd"/>

</xsd:schema>
```

7.3 Concepts and Observations

Chapter 6 has many sections that cover concepts or observations that might otherwise appear in this chapter. You should read

Chapter 6 before proceeding with this chapter.

The only observation that seems worth repeating pertains to namespaces. Although trivial to implement, one of the most difficult tasks when planning an XML schema is determining the target and default namespaces. Frequently, initial XML schema documents do not contain any namespace information other than where to locate the Schema for Schemas (see Listing 7.1). The documents evolve to include namespaces and target namespaces as needed. This approach is often acceptable. A schema should absolutely specify a target namespace, however, if the schema provides functionality over the Web or in any "public" schema that a user might integrate with a proprietary schema. Section 6.2 has a more complete discussion of namespaces.

7.4 The `schema` Element

The XML representation of an XML schema is one or more XML schema documents. Each XML schema document has exactly one `schema` element that encompasses all other elements in the document. Only an XML comment may appear before or after the `schema` element.

7.4.1 Attributes of a `schema` Element

Most attributes of a `schema` element pertain to namespaces. Two attributes pertain to blocking. The remaining attributes provide minor functionality. Table 7.1 provides a summary of all of the attributes of a `schema` element.

These four attributes are simply like MAIN switch.

Table 7.1. Attribute Summary for a `schema` Element

Attribute	*Description*
`attributeFormDefault`	The value of the `attributeFormDefault` attribute determines the default value of the `form` attribute for all attribute types.
`blockDefault`	The value of the `blockDefault` attribute determines the default value of the `block` attribute for element types and complex types.
`elementFormDefault`	The value of the `elementFormDefault` attribute determines the default value of the `form` attribute for all element types.
`finalDefault`	The value of the `finalDefault` attribute determines the default value of the `final` attribute for element types, simple types, and complex types.
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`targetNamespace`	The value of the `targetNamespace` is the namespace for any component described in the XML schema document. There may be explicitly no target namespace.
`version`	The `version` attribute has no special meaning. The schema writer might wish to version the XML schema or the XML schema document.
`xml:lang`	The value of the `xml:lang` attribute indicates the language of all human-readable information in a schema.

xmlns	The value of an xmlns attribute specifies an XML namespace. The xmlns attribute identifies existing namespaces to which qualified names might apply. Without a namespace prefix, the value of this attribute may also identify a default namespace for the XML schema document.

7.4.1.1 The attributeFormDefault Attribute of a schema Element

The attributeFormDefault attribute specifies a default value for the form attribute of all attribute types. Because the form attribute of an attribute type is not required, and because the attributeFormDefault defaults to 'unqualified', the overall default is that attributes in a corresponding XML instance do not require qualification.

Attribute Overview

schema: attributeFormDesign

Value:	'qualified' or 'unqualified'.
Default:	'unqualified'.
Constraints:	None.
Required:	No.

The impact on a corresponding XML instance is rather complex. Section 9.5.1.3, explains qualification of attribute instances, including the impact of the schema-wide attributeFormDefault.

Listing 7.2 is an entire XML schema document. The enclosing schema element has an attributeFormDefault attribute which specifies that, without an explicit override by an individual attribute type, an attribute type reference in a corresponding XML instance does not require qualification.

7.4.1.2 The blockDefault Attribute of a schema Element

The blockDefault attribute specifies a default value for the block attribute of all element types and complex types. Because element types and complex types do not require a block attribute, and because blockDefault is not required, the default state for all schemas does not block extensions, restrictions, or substitutions in corresponding XML instances.

Attribute Overview

schema: blockDefault

Value:	'#all' or a space-delimited list consisting of any or all of 'extensio 'restriction', or 'substitution'.
Default:	'unqualified'.
Constraints:	None.
Required:	No.

extension and restriction are apply to complex type only. like that substitution apply only to element only.

block cannot apply to simple type

The impact of blocking is complicated. Section 8.3.1.2 explains blocking of element types. Section 11.6.1.2 explains blocking

of complex types. Both of these explanations include the impact of the schema-wide `blockDefault`.

Note

i know. i understood my self

When the value for the `blockDefault` attribute includes '`substitution`', '`substitution`' does not apply to complex types; any other values apply. Another way of saying this is that an XML validator ignores a value of '`substitution`' in the `blockDefault` attribute when processing complex types.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has a `blockDefault` attribute which specifies that, without an explicit override, all types prohibit an XML instance from explicitly or implicitly extending, restricting, or substituting.

7.4.1.3 The `elementFormDefault` Attribute of a `schema` Element

The `elementFormDefault` attribute specifies a default value for the `form` attribute of all element types. Because the `form` attribute of an element type is not required, and because the `elementFormDefault` defaults to '`unqualified`', the overall default is that elements in a corresponding XML instance do not require qualification.

Attribute Overview

`schema: elementFormDefault`

Value:	'`qualified`' or '`unqualified`'.
Default:	'`unqualified`'.
Constraints:	None.
Required:	No.

The impact on an XML instance is rather complicated. Section 8.3.1.6 explains qualification of element instances, including the impact of the schema-wide `elementFormDefault`.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has an `elementFormDefault` attribute which specifies that, without an explicit override by an individual element type, an element type reference in a corresponding XML instance does not require qualification.

7.4.1.4 The `finalDefault` Attribute of a `schema` Element

The `finalDefault` attribute specifies a default value for the `final` attribute of all element types, simple types, and complex types. Because element types, simple types, and complex types do not require a `final` attribute, and because `finalDefault` is not required, the default state for all schemas is to not block extensions or restrictions in the XML schema document.

Attribute Overview

what is the difference between final and block ?

i hope both final and block more or less same except, block can have additionally 'substitution'

`schema: finalDefault`

Value:	'#all' or a space-delimited list consisting of any or all of 'extension' or 'restriction'.
Default:	'unqualified'.
Constraints:	None.
Required:	No.

The impact of blocking is complicated. Section 8.3.1.4 explains blocking of element types. Section 11.6.1.3 explains blocking of complex types. Both of these explanations include the impact of the schema-wide `finalDefault`.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has a `finalDefault` attribute which specifies that, without an explicit override, all types prohibit further restrictions (but not extensions) in the schema.

7.4.1.5 The `id` Attribute of a `schema` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`schema: id`

Value:	An ID.
Default:	None.
Constraints:	In general, the value of an ID-valued attribute must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has an `id` attribute.

7.4.1.6 The `targetNamespace` Attribute of a `schema` Element

The target namespace is the namespace for all the components in a schema document. In particular, all component names must be appropriately unique within this namespace. The target namespace is optional. When the XML schema document does not specify a target namespace, all enclosed components explicitly have no namespace; all references to these components must be unqualified. The exception to this rule is for components that explicitly belong to another namespace because of an `import` (see Section 7.9.). Section 6.2 has a more complete discourse on the interaction of namespaces.

Attribute Overview

`schema: targetNamespace`

Value:	A URI or the empty string (for example, 'targetNamespace=""'), which explicitly indicates no namespace.
Default:	No namespace.

Constraints:	None; in fact, there is no requirement that the location specified by the URI can be dereferenced.
Required:	No.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has a `targetNamespace` attribute.

7.4.1.7 The `version` Attribute of a `schema` Element

The `version` attribute of a `schema` element has no semantics. The value of the `version` attribute could represent the version of the XML schema or the XML schema document.

Attribute Overview

`schema: version`

Value:	Any string that conforms to the built-in `token` datatype.
Default:	None.
Constraints:	None.
Required:	No.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has a `version` attribute.

7.4.1.8 The `xml:lang` Attribute of a `schema` Element

An element in a schema document can specify various "special" attributes using the keyword '`xml`', which—to the casual observer—has the appearance of an attribute. The value of `xml:lang` specifies the default human language of all text in a schema document. `xml:lang` is not defined by the XML Schema Recommendation. Rather, `xml:lang` is a reserved attribute defined by the XML Recommendation. See RFC 1766 (http://www.ietf.org/rfc/rfc1766.txt) for more information on the language choices. For the English language, the value is '`en`'. The value for *United States* English is '`en-US`'.

Warning

Part 0 of the XML Schema Recommendation states that "you may indicate the language of all information in a schema by placing an `xml:lang` attribute on the `schema` element." Because multiple XML schema documents might represent an XML schema, the behavior of conflicting `xml:lang` attributes is indeterminate. The schema documents should respect other similar constraints by using the same value for `xml:lang` in all XML schema documents.

Attribute Overview

`schema: xml:lang`

Value:	Any string that conforms to the built-in `language` datatype.

Default:	None.
Constraints:	None.
Required:	No.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has an `xml:lang` attribute that specifies United States English.

7.4.1.9 The `xmlns` "Attributes" of a `schema` Element

An element in a schema document can specify a namespace declaration with `'xmlns'`, which—to the casual observer—has the appearance of an attribute. `xmlns` provides functionality to specify namespaces, including the default namespace. `xmlns` is not defined by the XML Schema Recommendation. Rather, `xmlns` is a reserved namespace as defined by the Namespace and Infoset Recommendations.

Attribute Overview

`schema: xmlns`

Value:	A URI.
Default:	None.
Constraints:	None.
Required:	No.

A namespace declaration, which starts with `'xmlns'`, has several functions:

- Associating a namespace prefix with a namespace
- Specifying a default namespace
- Removing a default namespace

The following example associates the namespace prefix `'cat'` with the namespace `http://www.XMLSchemaReference.com/examples/theme/catalog`:

```
xmlns:cat=
  "http://www.XMLSchemaReference.com/examples/theme/catalog"
```

The next example specifies a default namespace, which corresponds to the catalog in the previous example:

```
xmlns=
  "http://www.XMLSchemaReference.com/examples/theme/catalog"
```

Note that the namespace prefix is missing.

Finally, when the value of `xmlns` is empty, there is no default namespace:

```
xmlns=""
```

Tip

By default, there is no default namespace for an XML schema. In addition, an element nested within the `schema` element can remove the default namespace with the attribute specification '`xmlns=""`'.

Listing 7.2 is an entire XML schema document. The enclosing `schema` element has several `xmlns` attributes.

7.4.2 Content Options for a `schema` Element

The content options for a `schema` element pertain almost entirely to creating components that belong to the XML schema. The elements that do not represent components have to do with the details of describing an XML schema in XML schema documents, such as `include`, `import`, and `redefine`. Table 7.2 provides a summary of the numerous content options for a `schema` element.

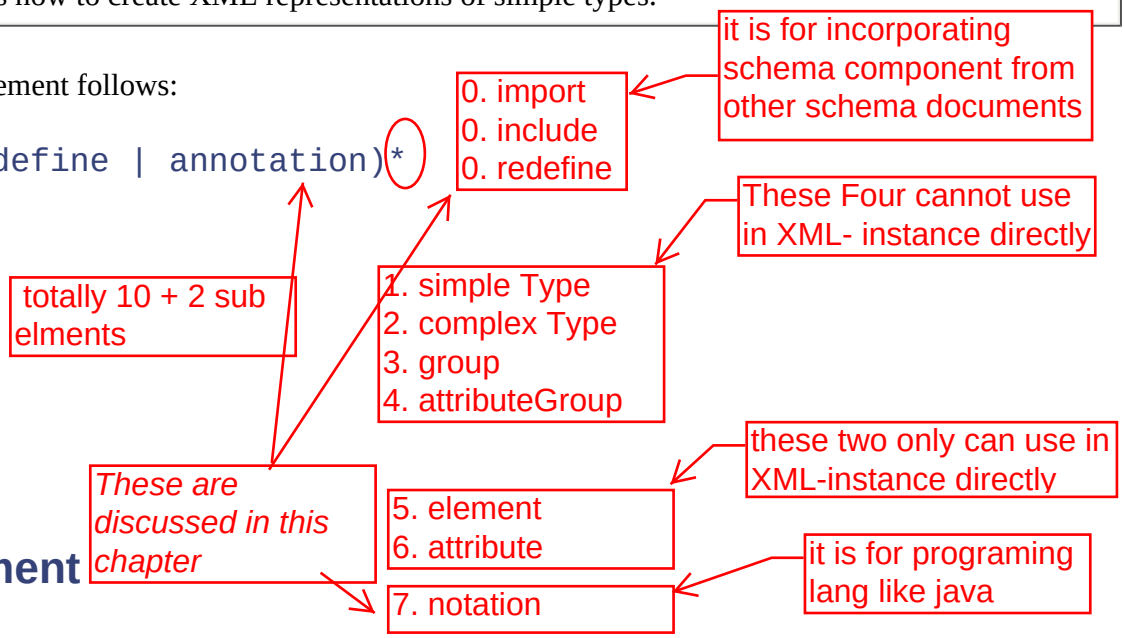
Table 7.2. Content Options for a `schema` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`attribute`	The XML schema document describes a global attribute type with an `attribute` element. Chapter 9 illuminates how to create XML representations of attribute types.
`attributeGroup`	The XML schema document describes a global set of attribute types with an `attributeGroup` element. Chapter 9 illuminates how to create XML representations of attribute use groups.
`complexType`	The XML schema document describes a global complex type with a `complexType` element. Chapter 11 illuminates how to create XML representations of complex types.
`element`	The XML schema document describes a global element type with an `element` element. Chapter 8 illuminates how to create XML representations of element types.
`group`	The XML schema document describes a global named model group with a `group` element. Section 11.14 illuminates how to create XML representations of named model groups.
`include`	An XML schema document can `include` components defined in the same namespace with the `include` element. The intent of the `include` element is to build a single XML schema from a set of XML schema documents. Section 7.8 illuminates how to describe an `include` element.
`import`	An XML schema document may reference components from a different or *foreign* target namespace with an `import` element. Section 7.9 illuminates how to describe an `import` element.
`notation`	A `notation` element describes XML notations (as documented in the XML Recommendation) in an XML schema document. A notation typically contains a reference to an executable program. For example, another program can refer to these notations to spawn the process identified in the notation. Section 7.10 illuminates how to create XML representations of notations.
`redefine`	An XML schema document describes a `redefine` element to replace a component. The new version of the component applies to the entire schema, not some portion thereof. Section 7.11 illuminates how to redefine existing schema components.

`simpleType`	The XML schema document describes a global simple type with a `simpleType` element. Chapter 10 illuminates how to create XML representations of simple types.

The content pattern for the schema element follows:

```
((include | import | redefine | annotation)*
 ((simpleType |
   complexType |
   group |
   attributeGroup) |
   element |
   attribute |
   notation |
   annotation*)*)
```



7.5 The `annotation` Element

The `annotation` element provides a mechanism for documenting most other schema elements. Unlike an XML comment, which looks like `<!-- comment text -->`, the annotation is part of the schema component. An `annotation` provides documentation intended for human consumption in one or more languages, as well as documentation intended for programmatic consumption, such as meaningful URIs.

Listing 7.3 demonstrates the use of an annotation that describes a Private Mail Box (PMB). The following example contains an `annotation` of `pmb`.

Listing 7.3 An `annotation` Element (`address.xsd`)

```
<xsd:element name="pmb"
             type="xsd:string"
             substitutionGroup="addressLine"
             minOccurs="1"
             maxOccurs="1">
    <xsd:annotation id="customerRecord.annotation.pmb">
        <xsd:documentation
            source="http://new.usps.com/cgi-bin/uspsbv/scripts/content.jsp?
D=13647"
            xml:lang="en">
            A PMB is a "Private Mail Box" that is provided
            by an entity other than the U S Postal Service.
        </xsd:documentation>
        <xsd:documentation xml:lang="en">
            Developer Note:  Someone should probably come up
            with a way to actually validate PMBs.  In fact,
            it would be great if we could validate
            every <pmb/> and <POBox/> element.
        </xsd:documentation>
        <xsd:appinfo
             source="http://www.XMLSchemaReference.com/examples/java/
```

```
extractJava">
            // A PMB is a "Private Mail Box" that is provided
            // by an entity other than the U S Postal Service.
            // -- create a class for the pmb
            public class pmb
                {
           *     *     *     *     *
                }
        </xsd:appinfo>
        <xsd:appinfo source="http:// www.XMLSchemaReference.com/examples/perl/
extractPerl">
            # A PMB is a "Private Mail Box" that is provided
            # by an entity other than the U S Postal Service.
            # -- create a variable for the PMB
            $pmb=""
        </xsd:appinfo>
    </xsd:annotation>
</xsd:element>
```

Note that XML schema document annotations do not affect the XML instances. The value of the `pmb` element is a Private Mail Box number; there is no annotation corollary:

```
<pmb>12345</pmb>
```

7.5.1 Attributes of an `annotation` Element

Since an `annotation` does not have a function with respect to XML validation, an `annotation` is limited to the `id` attribute. Table 7.3 provides an overview of the `id` attribute.

Table 7.3. Attribute Summary for an `annotation` Element

Attribute	*Description*
`id`	The `id` of the current `annotation`

7.5.1.1 The `id` Attribute of an `annotation` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`annotation: id`

Value:	An ID.
Default:	None.

Constraints:	In general, the value of an `id` must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

See Listing 7.3, which is the XML representation of `pmb`, which has an `id` attribute.

7.5.2 Content Options for an `annotation` Element

The `annotation` element can contain human- or machine-readable documentation. Table 7.4 summarizes the content options for the `annotation` element.

Table 7.4. Content Options for an `annotation` Element

Element	*Description*
`appinfo`	Provides a mechanism for specifying machine-readable documentation that pertains to the enclosing element. See Section 7.6 for details on the `appinfo` element.
`documentation`	Provides a mechanism for specifying human-readable documentation that pertains to the enclosing element. See Section 7.7 for details on the `documentation` element.

The content pattern for the `annotation` element follows:

both or either one

```
(appinfo | documentation)*
```

7.6 The `appinfo` Element

Only an `annotation` element can contain `appinfo` elements. The `appinfo` elements support automated processing of annotations—a discussion beyond the scope of this book. The following example demonstrates that the schema might support a mechanism for automatically creating source code that mirrors the XML schema document. One could, for example, create a program that extracts objects from the XML schema document in which each object mirrors a schema component. Listing 7.3 contains an `annotation` element, which encloses several `appinfo` elements. The following is a suitable excerpt:

```
<xsd:appinfo
    source=
"http://www.XMLSchemaReference.com/examples/java/extractJava">
    // A PMB is a "Private Mail Box" that is provided
    // by an entity other than the U.S Postal Service.
    // -- create a class for the pmb
    public class pmb
        {
          *     *     *     *     *
        }
</xsd:appinfo>
```

7.6.1 Attributes of an appinfo Element

The appinfo element has only one attribute, a source. Table 7.5 provides an overview of the source attribute.

Table 7.5. Attribute Summary for an appinfo Element

Attribute	*Description*
source	The value of the source attribute is a URI. Note that an XML processor does not validate this URI (or any other values within an annotation). The URI represents any documentation deemed relevant.

so it can be URN or URL

7.6.1.1 The source Attribute of an appinfo Element

The source attribute points to a URI that could be the location of a program to run, documentation about a program, a component, or just about anything that seems pertinent.

Attribute Overview

appinfo: source

Value:	A URI.
Default:	None.
Constraints:	None. In fact, there is no requirement that the URI can be dereferenced.
Required:	No.

7.6.2 Content Options for an appinfo Element

The appinfo element has no constraints on content. Therefore, an XML validator only ensures that an appinfo element is well-formed XML: The validator does not attempt to validate the contents of an appinfo element.

7.7 The documentation Element

Only an annotation element can contain a documentation element. The value of a documentation element is human-readable text that describes the element that encloses the annotation. The content, in an XML instance, of a documentation element is not constrained. Listing 7.3 contains an annotation element, which encloses several appinfo elements. The following is a suitable excerpt that demonstrates a documentation element with mixed content:

```
<xsd:documentation xml:lang="en">
    Developer Note:  Someone should probably come up
    with a way to actually validate PMBs.  In fact,
    it would be great if we could validate
    every <pmb/> and <POBox/> element.
</xsd:documentation>
```

An element that contains mixed content, as just demonstrated, may have content that intersperses elements with text. Chapter 11 provides more information about mixed content.

7.7.1 Attributes of a `documentation` Element

A `documentation` element may specify the language of the human-readable text. Table 7.6 provides a summary of the attributes for a `documentation` element.

Table 7.6. Attribute Summary for a `documentation` Element

Attribute	*Description*
`source`	The value of the `source` attribute is a URI. Note that an XML processor does not validate this URI (or any other values within an annotation). The URI points to any documentation deemed relevant.
`xml:lang`	The value of the `xml:lang` attribute specifies the language of the documentation. Note that an `annotation` might contain many `documentation` elements. Hence, each `documentation` element could present redundant verbiage in a distinct language.

7.7.1.1 The `source` Attribute of a `documentation` Element

The value of a `source` attribute is a URI that presumably assists the XML schema document reader in understanding the annotation.

Attribute Overview

`documentation: source`

Value:	A URI.
Default:	None.
Constraints:	None. In fact, there is not requirement that the URI can be dereferenced.
Required:	No.

The following documentation element tells the developer what a PMB is and references a document (URI) from the United States postal service that discusses PMBs:

```
<xsd:documentation
    source="http://new.usps.com/cgi-bin/uspsbv/scripts/content.jsp?D=13647"
    xml:lang="en">
                PMB is a "Private Mail Box" that is provided
                by an entity other than the U.S. Postal Service.
</xsd:documentation>
```

7.7.1.2 The `xml:lang` Attribute of a `documentation` Element

The `xml:lang` attribute indicates the human language contained within the `documentation` element. The `xml:lang` attribute is not defined by the XML Schema Recommendation. Rather, the `xml:lang` attribute is a reserved attribute defined by the XML Recommendation. See RFC 1766 for more information on the language choices. For the English language, the value is '`en`'. The value for *United States* English is '`en-US`'.

7.7.2 Content Options for a `documentation` Element

The `documentation` element has no constraints on content other than being well-formed XML. An XML validator does not validate the contents of a `documentation` element.

7.8 The `include` Element

Any given XML schema document may describe only a subset of an entire schema. An XML schema document may construct a larger subset of an entire schema, including the entire schema, by including other XML schema documents that contain different subsets of the components that comprise the schema. One XML schema document includes another via an `include` element.

Tip

An `include` element adds components from another schema, with a different namespace, to the current schema. An `import` element adds components from another schema, but associates them with the target namespace.

target name space
of including docs

The schema in Listing 7.2 includes components from the file `ffs_include.xsd`:

```
<xsd:include
    schemaLocation="http://www.XMLSchemaReference.com/examples/Ch07/ffs_include.
xsd"/>
```

Similarly, the thematic catalog example, `catalog.xsd`, makes use of multiple `include` statements. For example, `catalog.xsd` includes `pricing.xsd`; the latter describes components for specifying the price of catalog items. The catalog XML schema document includes the pricing XML schema document via the following `include` element:

```
<xsd:include
    schemaLocation="C:\XMLSchemaExample\theme\pricing.xsd"/>
```

still i am confusion.
because those two

7.8.1 Attributes of an `include` Element

An `include` element specifies the location of an XML schema document. Table 7.7 provides a summary of the attributes for an `include` element.

Table 7.7. Attribute Summary for an `include` Element

Attribute	*Description*

`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`schemaLocation`	The value of the `schemaLocation` attribute specifies an XML schema document that contains other components for the same XML schema.

7.8.1.1 The `id` Attribute of an `include` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`include: id`

Value:	An ID.
Default:	None.
Constraints:	In general, the value of an `id` must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

understood

7.8.1.2 The `schemaLocation` Attribute of an `include` Element

The value of the `schemaLocation` attribute is a URI that indicates one of the following:

- The location of an XML schema document
- A fragment that indicates a part of an XML document (via an HTTP address whose "content-type" is `application/xml` or `text/xml`)
- An XPointer that resolves to a `schema`

Attribute Overview

`include: schemaLocation`

Value:	A URI.
Default:	None.
Constraints:	The target namespace of the included document must be absent— which implies the current target namespace—or the target namespace of the included document must match the target namespace of the current document.
Required:	Yes.

care full

Unlike an `import` element, if the URI does not resolve to a valid XML schema document or an appropriate XML document, the `include` element has no effect. In other words, an XML validator ignores an `include` element whose `schemaLocation` is not valid.

7.8.2 Content Options for an `include` Element

Table 7.8 identifies the only content option available to an `include` element: the ubiquitous `annotation` element.

Table 7.8. Content Options for an `include` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the include element follows:

```
annotation?
```

7.9 The `import` Element

One XML schema can appropriate components from another XML schema that has a different namespace. An `import` element identifies a schema (usually by identifying a schema document that represents it) using a different namespace, and generates equivalent components in the new schema. The components retain their original namespace in the new schema.

Tip

An `include` element adds components from another schema, with a different namespace, to the current schema. An `import` element adds components from another schema, but associates them with the target namespace.

Section 6.15 provides a complete discussion on how the `namespace` and `schemaLocation` attributes affect locating schema components. The following list is a brief overview of how the schema components are located and identified:

- When an `import` element specifies both a `schemaLocation` and a `namespace`, the XML validator builds the new components into the namespace specified by the `namespace` attribute, *which must be the same as that specified by the source* `targetNamespace`.
- When an `import` element specifies only a `schemaLocation`, the source `schema` must not specify a `targetNamespace`; the new components explicitly have no target namespace.
- When an `import` element specifies only a `namespace`, the XML validator deduces the schema location from other known schemas and namespaces when possible. The source `schema` must specify the same namespace as its `targetNamespace`; that is, the namespace of the new components.
- If an `import` element specifies neither a namespace nor a schema (for example, the empty element '`<import />`'), the XML validator is given no clues as to where to locate foreign components; if through outside means it can find that place, the source `schema` must not specify a `targetNamespace` and the new components explicitly have no target namespace.

if including doc has targetNamespace it must use namespace attribute for import while importing

A schema frequently infers imports by specifying a namespace as an attribute of the `schema` element. The following example demonstrates how to explicitly import the Schema for Schemas:

Note, here not issue about prefix. so, prefix can be anything

```
<import namespace="http://www.w3.org/2001/XMLSchema"
```

so, more than one schema document can have the same name space.

```
        schemaLocation="http://www.w3.org/2001/XMLSchema.xsd"
        id="import.XMLSchema"/>
```

while more than one doc has same namespace if i dont dont give schema location, which one be identified by validator among them?? obviously the first one

Similarly, Listing 7.2 imports attributes defined in another namespace and another file:

```
<xsd:import
    namespace="http://www.XMLSchemaReference.com/examples/Ch07/
randomAttributes"
    schemaLocation="http://www.XMLSchemaReference.com/examples/Ch07/
randomAttributes.xsd"/>
```

7.9.1 Attributes of an `import` Element

The attributes of an `import` element provide an XML validator clues for locating foreign components. Table 7.9 provides a summary of the attributes of an `import` element.

Table 7.9. Attribute Summary for an `import` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`namespace`	The value of the `namespace` attribute specifies a target namespace for the foreign components.
`schemaLocation`	The value of the `schemaLocation` attribute specifies an XML schema document that contains foreign components.

7.9.1.1 The `id` Attribute of an `import` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`import`: `id`

Value:	An ID.
Default:	None.
Constraints:	In general, the value of an `id` must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

7.9.1.2 The `namespace` Attribute of an `import` Element

The value of the `namespace` attribute specifies the namespace for foreign components. Specifically, subsequent elements may contain references to components in the foreign namespace—with an appropriate namespace prefix.

Attribute Overview

`import: namespace`

Value:	A URI.
Default:	None.
Constraints:	Whether this attribute is present or absent, it must match the `targetNamespace` attribute on the source `schema` and be different from the target namespace of the schema being created. (If the namespace is absent then the XML validator creates the new components explicitly with no target namespace and the schema being created must have a target namespace.)
Required:	No.

7.9.1.3 The `schemaLocation` Attribute of an `import` Element

The value of the `schemaLocation` attribute is a URI that indicates the location of an XML schema document. Other permissible values are an XML document of type `application/xml` or `text/xml` that contains a fragment, or an XPointer notation that resolves to a `schema`.

Attribute Overview

`import: schemaLocation`

Value:	A URI.
Default:	None.
Constraints:	An XML validator requires a valid schema location.
Required:	No.

Unlike an `include` element, the URI *must* resolve to a valid XML schema document or a `schema` within another document. Note, however, that the `schemaLocation` attribute is optional.

7.9.2 Content Options for an `import` Element

Table 7.10 identifies the only content option available to an `import` element: the ubiquitous `annotation` element.

Table 7.10. Content Options for an `import` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the `import` element follows:

```
annotation?
```

7.10 The `notation` Element

A notation provides a mechanism for an XML validator to locate external programs or processing instructions. An XML validator does not intrinsically validate notations. However, during validation of a corresponding XML instance, a `notation` must exist for each reference from an element or attribute whose corresponding structure type specifies an enumeration of the built-in `notation` datatype. While there are no specific requirements, the intention of a notation is that the value for a notation in an XML instance is somehow relevant to URIs specified by the notation.

Each `notation` must specify a `system` attribute, a `public` attribute, or both. The value of a `system` attribute frequently identifies a file, which might be a program. There is no requirement that the file exists on the parsing machine, or that the file exists at all. The value of a `public` attribute frequently identifies an external HTTP address. This URI might likewise represent a document or a program such as an Active Server Page (ASP) or JavaServer Page (JSP). Conversely, there is no requirement that the URI can be dereferenced. Note that "public" is relative to the current system: A URI could point to another machine accessible only within the same company.

Listing 7.4 is the XML representation of two `notation` elements, which specify the location of a Perl and a Python interpreter.

Listing 7.4 A `notation` Element (`built-in.xsd`)

```
<xsd:notation name="perlCode"
              system="/usr/bin/perl"
              public="http://www.company.com/runPerl.pl"
              id="notation.perl">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            value of corresponding element
            should contain perl code to execute.
        </xsd:documentation>
    </xsd:annotation>
</xsd:notation>
<xsd:notation name="pythonCode"
              system="/usr/bin/python"
              public="http://www.company.com/runPython.py"
              id="notation.companyMascot">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            value of corresponding element
            should contain python code to execute.
        </xsd:documentation>
    </xsd:annotation>
</xsd:notation>
```

Typically, the value associated with a notation is an attribute; the value of the enclosing element is a value that applies to the

notation. In the next example, the `demoNotation` element contains a `source` attribute. The value of `source` is '`perlCode`'; the value of the element is a line of Perl code:

```
<xsd:element name="demoNotation">
    <xsd:complexType>
        <xsd:simpleContent>
            <xsd:extension base="xsd:string">
                <xsd:attribute name="source">
                    <xsd:simpleType>
                        <xsd:restriction base="xsd:notation">
                           <xsd:enumeration value="perlCode"/>
                           <xsd:enumeration value="pythonCode"/>
                        </xsd:restriction>
                    </xsd:simpleType>
                </xsd:attribute>
            </xsd:extension>
        </xsd:simpleContent>
    </xsd:complexType>
</xsd:element>
```

Note

An attribute (the normal scenario) or an element cannot specify a `notation` as its structure type: The structure type must be a simple type that is an enumeration of other `notation` simple types.

Given the preceding element type, the following element is valid in a corresponding XML instance:

```
<demoNotation
  source="perlCode">print "Hello, World\n"</demoNotation>
```

7.10.1 Attributes of a `notation` Element

The attributes of a `notation` element must specify a name and a way to obtain the corresponding processing instructions. Table 7.11 provides a summary of the attributes for a `notation` element.

Table 7.11. Attribute Summary for a `notation` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`name`	The value of the `name` attribute is the name of the `notation`.
`public`	The value of the `public` attribute is a character string; it may be a URI that represents the location of a corresponding document or program. While by no means a requirement, there is some expectation that the dereferenced URI is available on a system other than the one upon which the XML validator runs.

system	The value of the system attribute is a URI that represents the location of a corresponding document or program. While by no means a requirement, there is some expectation that the dereferenced URI is available locally—perhaps a local file.

7.10.1.1 The id Attribute of a notation Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

notation: id

Value:	An ID.
Default:	None.
Constraints:	In general, the value of an id must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

7.10.1.2 The name Attribute of a notation Element

An XML validator places the value of the name of a notation element in the target namespace. An element or attribute in an XML instance refers to the notation with this name. The element or attribute must have a corresponding element type or attribute type whose structure type specifies an enumeration of built-in notation datatypes. An application that interacts with an XML validator may do anything—or nothing—with the data from the corresponding notation.

Attribute Overview

notation: name

Value:	An NCName.
Default:	None.
Constraints:	No intrinsic constraints. However, there must be a corresponding notation element for each name referenced in an XML instance.
Required:	Yes.

7.10.1.3 The public Attribute of a notation Element

The value of the public attribute may be a URI that the XML validator may use as a processing instruction. For example, this URI might be the address of an HTML file or a Java servlet. The schema does not guarantee that the URI can be dereferenced.

Attribute Overview

notation: public

Value:	A character string; often a URI.
Default:	None.
Constraints:	None. In fact, there is no requirement that the URI can be dereferenced.
Required:	A `notation` element must specify either or both of the `system` or `public` attributes.

7.10.1.4 The `system` Attribute of a `notation` Element

The value of a `system` attribute is a URI. For example, the path might point to a program to render a graphics file or any other executable program. The schema does not guarantee that the URI can be dereferenced, or if it can, that the dereferenced string has any meaning.

Attribute Overview

`notation: system`

Value:	A URI.
Default:	None.
Constraints:	None. In fact, the representative file may not exist or may not be accessible on the validating machine.
Required:	A `notation` element must specify either or both of the `system` or `public` attributes.

7.10.2 Content Options for a `notation` Element

Table 7.12 identifies the only content option available to a `notation`: the ubiquitous `annotation` element.

Table 7.12. Content Options for a `notation` Element

Element	*Description*
`annotation`	Provides a way to document schema elements. Section 7.5 demonstrates how to apply annotations.

The content pattern for the `notation` element follows:

```
annotation?
```

7.11 The `redefine` Element

An XML schema document may redefine a schema component in the current schema or in another schema by including a `redefine` element. A schema can redefine the following components: simple types, complex types, named model groups, and named attribute-use groups.

The target namespace implied by the schema location of the `redefine` element must be the same as the target namespace of the current schema. However, the `redefine` element can reference a component from another schema in which the referenced component explicitly has no namespace. In the latter case, *the components become part of the current namespace*.

Note

A redefined component is always a restriction or extension of the original component.

The following list itemizes how to redefine each kind of component:

- A redefined simple type is a restriction of the original component. This means the value of the `base` attribute must be the name of the original component.
- A redefined complex type is an extension or a restriction of the original component. This means the value of the `base` attribute *is* the name of the redefined component.
- A redefined named model group is a superset or subset of the original named model group. A superset of the original named model group must include the original named model group via a reference (that is, a `group` element that has a `ref` attribute whose value is the original named model group). A subset of the original named model group must have identical subcomponents with appropriate restrictions. An appropriate restriction of a named model group is a modification of a `minOccurs` or `maxOccurs` attribute of a contained element type.
- A redefined named attribute-use group is a superset or subset of the original named attribute-use group. A superset of the original named attribute-use group must include the original named attribute-use group via a reference (that is, an `attributeGroup` element that has a `ref` attribute whose value is the original named attribute-use group). A subset of the original group must have identical subcomponents with appropriate restrictions. An appropriate restriction is the modification of a `use` attribute for a contained attribute type.

~~Although XML schema documents may redefine a component many times, there is ultimately only one instance of any of the redefined components in a schema: The same schema does not have different "versions" of a component.~~

Finally, a redefined component may cause a schema or a corresponding XML instance to become invalid: The previous iteration of the component does not exist in the schema.

redefine, is almost overriding. so, the base one will be gone off

Listing 7.5 is an XML schema document that describes `aSimpleType`.

Listing 7.5 Defining `aSimpleType`

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
    <xsd:simpleType name="aSimpleType">
    <xsd:restriction base="xsd:token"/>
    </xsd:simpleType>
</xsd:schema>
```

Listing 7.6 redefines `aSimpleType`, originally defined in Listing 7.5.

Listing 7.6 Redefining `aSimpleType`

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

```
    schemaLocation="http://www.XMLSchemaReference.com/examples/Ch07/redefine1.
xsd">
<xsd:simpleType name="aSimpleType">
    <xsd:restriction base="aSimpleType">
        <xsd:maxLength value="40"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:redefine>
</xsd:schema>
```

7.11.1 Attributes of a `redefine` Element

The attributes of a `redefine` element include the ubiquitous `id` and the `schemaLocation`, which identifies a schema document. Table 7.13 provides a summary of the attributes for a `redefine` element.

Table 7.13. Attribute Summary for a `redefine` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`schemaLocation`	The value of the `schemaLocation` attribute identifies a schema document that describes the original components. The target namespace of the original components must be identical to the enclosing schema document. Alternatively, there can be no target namespace for the original components, in which case the components adopt the enclosing schema's target namespace.

7.11.1.1 The `id` Attribute of a `redefine` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`redefine: id`

Value:	An ID.
Default:	None.
Constraints:	In general, the value of an `id` must be unique within an XML document. The XML Schema Recommendation further constrains this uniqueness to the entire XML schema.
Required:	No.

7.11.1.2 The `schemaLocation` Attribute for a `redefine` Element

The value of the `schemaLocation` attribute is a URI that indicates the location of an XML schema document. Another permissible value is an XML document of type `application/xml` or `text/xml` that contains a fragment or XPointer

notation that resolves to a `schema`.

Attribute Overview

`redefine: schemaLocation`

Value:	A URI.
Default:	None.
Constraints:	The XML validator requires a valid URI.
Required:	Yes.

7.11.2 Content Options for a `redefine` Element

Table 7.14 identifies the content options available to a `redefine` element. Other than the frequently seen `annotation`, the contents of a `redefine` are the redefined components.

Table 7.14. Content Options for a `redefine` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`simpleType`	A restriction of a previously defined simple type.
`complexType`	A restriction or extension of a previously defined complex type.
`group`	A restriction or extension of a previously defined named model group.
`attributeGroup`	A restriction or extension of a previously defined named attribute-use group.

The content pattern for the `redefine` element follows:

```
(annotation | simpleType | complexType | group | attributeGroup)*
```

CONTENTS

CONTENTS

Chapter 8. Element Types

IN THIS CHAPTER

The main purpose of an *element type* is to associate an element name with a *structure type*. The structure type is a simple type or a complex type. Furthermore, the structure type may be global to the schema, or it may be local to the element type. The XML representation of an element type is an `element` element.

The simplest form of an element type is one whose only attribute is `name`. An element type that does not explicitly specify a structure type implicitly specifies `anyType` as the structure type. The content of an element in an XML instance whose structure type is `anyType` is unconstrained.

8.1 An Example of a Trivial Element Type

The most basic element type is one that associates an element name with a built-in datatype. The following element demonstrates the XML representation of `city`, which associates the name '`city`' with the built-in `token` datatype:

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

The value of a `city` element, which is an instance of the `city` element type, is a string of any length. Of course, this string is constrained by the constraining facets that specify `token` (Chapter 12 discusses all of the built-in datatypes including `token`). Given the preceding element type, the following element is valid in an XML instance:

```
<city>New York</city>
```

> **Tip**
>
> The `city` element type example can—and probably should—cover a large percentage of the element types specified in an XML schema document. Good programming style dictates the creation of global simple and complex types with references to these types as the structure type of primarily local element types.

8.2 Concepts and Observations

Element types have very sophisticated, and often very confusing, capabilities. The next four sections attempt to expose some of the confusing issues surrounding element types. These sections cover global and local element types,

substitution groups, blocking, `nil` elements (those with no value), and the use of namespaces vis-à-vis element types.

8.2.1 Global and Local Element Types

An element type is global or local. A global element type is a child of the `schema` element in the XML schema document. Otherwise, the element type is local to a complex type.

Listing 8.1 portrays the `city` element type introduced in Section 8.1 as local to the complex type `addressType`.

Listing 8.1 A Local Element Type (`address.xsd`)

```
<xsd:complexType name="addressType">
    <xsd:sequence>
        <xsd:element ref="addressLine"
                     minOccurs="1"
                     maxOccurs="2"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="country"
                     type="xsd:string"
                     fixed="US"/>
        <xsd:element name="zip" type="xsd:string"/>
        <xsd:element ref="effectiveDate"/>
    </xsd:sequence>
    <xsd:attribute ref="zipPlus4" use="optional"/>
</xsd:complexType>
```

8.2.2 Substitution Groups

A substitution group provides functionality that parallels derivation in that appropriate element types in a substitution group replace instantiable or non-instantiable element types. The parallel is that when an instantiable-derived type replaces a base type, substitutable element types replace base element types. The substitution of element types occurs in an XML instance or in an XML schema document. Listing 8.2 is a substitution group that provides a foundation for complete addresses.

Listing 8.2 The `addressLine` Substitution Group (`address.xsd`)

extension and restriction gives only reuseability of oops. BUT only subtitution group gives RUNTIME POLIMORPHISM

```
<xsd:element name="addressLine"
             id="customerRecord.base.addressLine"
             type="xsd:string"
             abstract="true">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The "addressLine" element type
            is a base non-instantiable element
            type whose structure type is
```

```
            a built-in string datatype.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="street"
             type="xsd:string"
             substitutionGroup="addressLine">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Street is a substitution group
            for addressLine.  This particular
            substitution only substitutes the
            name; there are no further restrictions.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="POBox"
             substitutionGroup="addressLine">
    <xsd:simpleType>
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                The POBoxType demonstrates that
                a Substitution Group can have a
                type that is derived from the
                type used by the related
                non-instantiable base element.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:restriction base="xsd:string">
            <xsd:maxLength value="10"/>
            <xsd:pattern value="[0-9]+"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>

<xsd:element name="pmb"
             type="xsd:string"
             substitutionGroup="addressLine">
    <xsd:annotation id="customerRecord.annotation.pmb">
        <xsd:documentation
             source="http://new.usps.com/cgi-bin/uspsbv/scripts/content.jsp?
D=13647"
             xml:lang="en">
            A PMB is a "Private Mail Box" that is
            provided by an entity other than the
```

```
            U S Postal Service.
          </xsd:documentation>
        <xsd:documentation xml:lang="en">
            Developer Note:  Someone should probably
            come up with a way to actually validate PMBs.
          In fact, we should validate every <pmb/>
          and <POBox/> element.
        </xsd:documentation>
        <xsd:appinfo
            source="http://www.XMLSchemaReference.com/examples/java/
extractJava">
            // A PMB is a "Private Mail Box" that is
            // provided by an entity other than the
            // U S Postal Service.
            // -- create a class for the pmb
            public class pmb
                {
          *     *     *     *     *
                }
         </xsd:appinfo>
         <xsd:appinfo
             source="http://www.XMLSchemaReference.com/examples/perl/
extractPerl">
             # A PMB is a "Private Mail Box" that is provided
             # by an entity other than the U S Postal Service.
             # -- create a variable for the PMB
             $pmb=""
         </xsd:appinfo>
     </xsd:annotation>
</xsd:element>
```

Substitution groups provide quite a bit of flexibility:

super. it is like java interface

- The head element type can be instantiable or non-instantiable.
- The structure types associated with the element types in a substitution group can be different from each other. Substitutable element types can specify a structure type that is a derivation of the structure type associated with the head element type.
- The structure types associated with the element types in a substitution group can be simple types or complex types. The only limitation is that the structure types are suitable derivations.

The XML representation of an element type whose structure type is the complex type `addressType` introduced in Listing 8.2 might look like the following element type:

only elements INSIDE (structure types) of replacing element suitable for derivation

```
<xsd:element name="address" type="addressType"/>
```

Given the preceding element type, any of the following elements are valid in an XML instance. Note the inline

replacement of addressLine with street, POBox, and pmb, respectively:

```
<address>
    <street>123 Gravel Road</street>
    <city>Nowheresville</city>
    <state>OR</state>
    <country>US</country>
    <zip>97000</zip>
    <effectiveDate></effectiveDate>
</address>

<address zipPlus4="20000-1234">
    <POBox>123</POBox>
    <city>Small Town</city>
    <state>VA</state>
    <country>US</country>
    <zip>20000</zip>
    <effectiveDate>2001-02-14</effectiveDate>
</address>

<address>
    <street>123 Main Street</street>
    <pmb>12345</pmb>
    <city>Metropolis</city>
    <state>CO</state>
    <country>US</country>
    <zip>80000</zip>
    <effectiveDate>2001-02-14</effectiveDate>
</address>
```

Frequently, the substitution occurs in the XML instance. The next example shows POBox substituted for the non-instantiable addressLine:

```
<address>
    <POBox>123</POBox>
    <city>Small Town</city>
    <state>VA</state>
    <country>US</country>
    <zip>20000</zip>
    <effectiveDate>2001-02-14</effectiveDate>
</address>
```

Listing 8.3 provides an XML representation for POBoxAddressType. This listing demonstrates substituting POBox for addressLine in an XML schema document that specifies a derived complex type.

Listing 8.3 Specifying a Substitution Group in a Derived Type (substGroup.xsd)

```
<xsd:complexType name="POBoxAddressType">
    <xsd:complexContent>
        <xsd:restriction base="addressType">
            <xsd:sequence>
                <xsd:element ref="POBox"
                             minOccurs="1"
                             maxOccurs="1"/>
                <xsd:element name="city" type="xsd:string"/>
                <xsd:element name="state" type="xsd:string"/>
                <xsd:element name="country"
                             type="xsd:string"
                             fixed="US"/>
                <xsd:element name="zip" type="xsd:string"/>
                <xsd:element ref="effectiveDate"/>
            </xsd:sequence>
            <xsd:attribute ref="zipPlus4" use="optional"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

while restricting or extending of SUBSTItUTIOn group can remove or add any more elements ??

not addressLine

Tip

Sections 8.3.1.2 and 8.3.1.4 discuss how to prevent element type substitutions by specifying the `block` and `final` attributes.

8.2.3 Blocking Substitution

The `block` attribute of an element type, in conjunction with the `block` attribute of a complex type, restricts the ability to substitute, in an XML instance, specific members of a substitution group. Listing 8.4 is a complex set of substitution groups. The listing includes element types and the relevant structure types. The listing sets up the discussion surrounding the `block` attribute.

Tip

A solid understanding of complex types (see Chapter 11), including the `block` and `final` attributes (Sections 11.6.1.2 and 11.6.1.3, respectively), is valuable before attempting to use the `block` or `final` attributes of an element type.

Likewise, a solid understanding of substitution groups (see Section 8.2.2) is imperative.

There are three ways to impose blocking:

#all

- When the head of a substitution group blocks substitution, an XML instance cannot "replace" the head with

another member of the substitution group.

- When the head of a substitution group blocks extension, an XML instance cannot "replace" the head with another member of the substitution group whose structure type is an extension of the structure type associated with the head. In addition, an XML instance cannot specify a structure type, via `xsi:type`, that is derived—even indirectly—by extension from the structure type associated with the head.
- When the head of a substitution group blocks restriction, an XML instance cannot "replace" the head with another member of the substitution group whose structure type is a restriction of the structure type associated with the head. In addition, an XML instance cannot specify a structure type, via `xsi:type`, that is derived—even indirectly—by restriction from the structure type associated with the head.

Note

A complex type can also specify blocking. The blocking is transitive. That is, when an element type's structure type is a complex type, a block on that complex type effectively places a block on the element type.

super.. .

In general, substitution groups are transitive. Listing 8.4 specifies two substitution groups: The first one contains the members 'A', 'B', and 'C', with 'A' being the head; the second one contains 'C' and 'D', with 'C' being the head. Transitively (through 'C'), 'B', 'C', and 'D' are valid substitutions for 'A'. Blocking, however, is *not* transitive. Specifically, a block on 'C' (or its structure type, 'T3'), does not restrict the ability of 'D' to be a valid substitution for 'A'.

Listing 8.4 Multiple Substitution Groups (`substGroup.xsd`)

```
<xsd:element name="A" type="T1" abstract="true"
             block="restriction">
    <xsd:annotation>
        <xsd:documentation>
            Head of substitution group A.
            An A element cannot appear in an XML
            instance, since this element type is
            abstract.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
<xsd:element name="B" type="T1" substitutionGroup="A">
    <xsd:annotation>
        <xsd:documentation>
        Member of substitution group A.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
<xsd:element name="C" type="T3" substitutionGroup="A"
             block="substitution">
    <xsd:annotation>
        <xsd:documentation>
            Member of substitution group A.
```

so, to make any EXITING element as head of a substitution group no need any CHANGE in that existing tag

```
            Head of substitution group C.
            Blocking 'substitution' or 'extension' on C,
            or blocking '#all' or 'extension' on T3,
            would prevent D from being a valid substitution
            for C.  D would, however, retain
            its transitive ability to be a valid
            substitution for A.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
<xsd:element name="D" type="T4" substitutionGroup="C">
    <xsd:annotation>
        <xsd:documentation>
            Member of substitution group C.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
<xsd:complexType name="T1">
    <xsd:sequence>
        <xsd:element name="_1" type="xsd:token"/>
    </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="T2">
    <xsd:complexContent>
        <xsd:extension base="T1">
            <xsd:sequence>
                <xsd:element name="_2" type="xsd:token"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="T3">
    <xsd:complexContent>
        <xsd:extension base="T2">
            <xsd:sequence>
                <xsd:element name="_3" type="xsd:token"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="T4">
    <xsd:complexContent>
        <xsd:extension base="T3">
            <xsd:sequence>
                <xsd:element name="_4" type="xsd:token"/>
            </xsd:sequence>
        </xsd:extension>
```

So, D become replacement for A. but it is due to transitive of C, but D cannot be replacement for C.

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

8.2.4 Element Type Instantiability

Normally, element types are instantiable. Furthermore, there is no way to "derive" one element type from another. The only utility of non-instantiable element types is in a substitution group: The head of the substitution group might be non-instantiable; the other members of the substitution group are instantiable, and can therefore "replace" the head.

Only element types and complex types can be *explicitly* non-instantiable. An XML instance cannot specify instances of an element type whose `abstract` attribute is `'false'`. Some element types are *implicitly* non-instantiable: An XML instance cannot specify instances of an element type whose structure type refers to a non-instantiable complex type.

Listing 8.2 demonstrates a non-instantiable `addressLine`, along with the appropriate substitutions `street`, `POBox`, and `pmb`. Note that the structure type associated with `addressLine` is `xsd:string`. The substitution `street` has the same structure type: `xsd:string`. The structure type associated with `POBox`, on the other hand, is a derivation of `xsd:string`.

Listing 8.4 is another example. Element type `A` is non-instantiable. All of the other element types in this listing are valid instantiable substitutions for `A`.

8.2.5 Nillable Element Types

An element type can specify a `nillable` attribute. When this occurs, the value of an element instance can be nil. A nil value is not the same as an *empty* value. For a string, "empty" means a string of zero length; nil means no string at all. Listing 8.5 specifies the nillable `phoneNumber` element type.

my current parser does not work as said. i tried nillable as false and instance element as self closed. but i have not got error

Listing 8.5 A Nillable Element Type (`address.xsd`)

```
<xsd:element name="phoneNumber"
             type="xsd:string"
             nillable="true"/>
```

The XML instance must use the special `nil` attribute from the XML schema instance namespace to set the value to nil:

```
<phoneNumber xsi:nil="true"/>
```

For clarity, a `phoneNumber` with an empty string has just a start-tag and an end-tag, or is an empty element:

```
<phoneNumber></phoneNumber> or </phoneNumber/>
```

Note that the element that specifies the `nil` value—or an enclosing element—must declare the http://www.w3.org/2001/XMLSchema-instance namespace (see Chapter 7). The following declares this namespace tied to the `'xsi'` qualifier, as used in the preceding XML instance with a nil value:

```
xmlns:xsi=http://www.w3.org/2001/XMLSchema-instance
```

Of course, an element whose element type permits a nil value can still have a value or content that is valid given the element type's structure type. Listing 8.5 specifies `xsd:string` as the structure type of `phoneNumber`. The following element, with a string value, is a valid instance:

```
<phoneNumber>212-555-0000</phoneNumber>
```

8.2.6 Element Types and Namespaces

This section discusses how an XML instance must specify namespaces for elements in order to locate the appropriate element types described throughout this chapter. Chapter 3 discusses how namespaces apply to the element types in the schema.

In an XML instance, the namespace of an element must match the namespace identified by the element type. Typically, element types do not specify the target namespace. Normally, the `schema` element specifies the target namespace. If a namespace is not specifically identified, and the target namespace is not specified, the element must be unqualified (that is, specified without a namespace).

In an XML instance, an element may be *qualified* (namespace specified) or *unqualified* (namespace not specified). An XML validator determines whether the namespace is required, by evaluating the following rules in order:

- The element must be appropriately qualified when the element type specifies a `form` attribute (which has a value of '`qualified`' or '`unqualified`')
- When the element type does not specify the `form` attribute, the `elementFormDefault` attribute (which also has a value of '`qualified`' or '`unqualified`') of the `schema` element enclosing the element type determines whether the name of the element instance is qualified.
- When the element type does not specify the `form` attribute, and the schema does not specify the `elementFormDefault` attribute, the element must be unqualified. In other words, the default value for `elementFormDefault` is '`unqualified`'

Section 8.3.1.6, along with Listings 8.7 and 8.8, provides an example of how the `form` attribute of an element type works in conjunction with the `elementFormDefault` attribute of the `schema` element.

8.3 The `element` Element

The XML representation of an element type is an `element` element. Most element types are relatively straightforward: The element type only associates a name with a structure type. Despite the fact that this simple association covers the majority of element types, element types support a wide range of functionality. Element type functionality, described in this section, includes default values, element instance cardinality, substitution groups, blocking, and more.

8.3.1 Attributes of an `element` Element

One of the great features about XML schemas is the ability to specify—in great detail—the structure and contents of element types. Table 8.1 summarizes the individual attributes used to constrain an `element` element.

Table 8.1. Attribute Summary of an `element` Element

Attribute	*Description*
`abstract`	The value of the Boolean `abstract` attribute deems an element type instantiable or non-instantiable. A substitution group provides functionality for inheriting from non-instantiable element types.
`block`	The value of the `block` attribute is a space-delimited list containing any combination of '`extension`', '`restriction`', and '`substitution`'. These values constrain an element instance from respectively extending, restricting, or substituting the element type directly in an XML instance. The value '#all', which implies all of these values, indicates that an XML instance cannot transform the element type in any way.
`default`	An XML parser substitutes the value of the `default` attribute of an element type when an element instance has no value.
`final`	The value of the `final` attribute is a space-delimited list containing any combination of '`extension`' or '`restriction`'. These values restrict the schema from extension, restriction, or substitution, respectively. The value '#all', which implies all of these values, indicates that the schema cannot directly reference this element type.
`fixed`	The `fixed` attribute is similar to the default attribute. However, when an element instance does specify a value, that value must be identical to the value of the fixed attribute specified by the element type.
`form`	The `form` attribute of a local element type determines whether the element instance must—or must not—specify an appropriate namespace.
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`maxOccurs`	The value of the `maxOccurs` attribute determines the maximum number of times that an element instance can appear in an enclosing element. Applies only to local element types.
`minOccurs`	The value of the `minOccurs` attribute determines the minimum number of times that an element instance can appear in an enclosing element. Applies only to local element types.
`name`	The value of the `name` attribute is the name used to reference an element type in the rest of the schema or in an XML instance.
`nillable`	The value of the `nillable` attribute determines if nil is a valid value for an element instance.
`ref`	The value of the `ref` attribute is a global element type. In an XML instance, this reference permits an enclosing element to contain an element whose element type is global.
`substitutionGroup`	The value of the `substitutionGroup` attribute is a global element type. Any element types in a substitution group may be substituted in the schema or in an XML instance—barring any blocks from a `final` or `block` attribute.
`type`	The value of the `type` attribute is the structure type that constrains the value or content of the element type.

8.3.1.1 The `abstract` Attribute of an `element` Element

An XML instance cannot contain instances of element types whose representation sets the `abstract` attribute value to '`true`'. However, An XML instance can replace non-instantiable element types with instantiable element types simply by refering to the name of a valid instantiable substitution. In an XML schema, complex types reference these non-instantiable element types as well as instantiable substitutions. Non-instantiable element types are only useful in the context of substitution groups. See Sections 8.2.2 and 8.3.1.14 for more information on substitution groups.

The `block` and `final` attributes of the element type control the ability to substitute element types. Sections 8.3.1.2 and 8.3.1.4 discuss the `block` attribute and the `final` attribute, respectively.

Attribute Overview

`element: abstract`

Value:	A Boolean (that is, '`true`' or '`false`')
Default:	'`false`'—The typical element type may be used in an XML instance.
Constraints:	None.
Required:	No.

Section 8.2.2 discusses and demonstrates how to use non-instantiable element types and their related substitution groups.

Listing 8.2 portrays the XML representation of the non-instantiable `addressLine`. Only a complex type or another element type that specifies a `substitutionGroup` attribute may reference a non-instantiable element type. The `street` element type described in Listing 8.2 is a valid reference to a non-instantiable element type.

An XML instance may not reference a non-instantiable element type. Therefore, the following element type is not valid:

```
<addressLine>123 Illegal Address</addressLine>
```

Warning

Some XML parsers permit the XML schema or an XML instance to incorrectly reference non-instantiable element types.

8.3.1.2 The `block` Attribute of an `element` Element

The `block` attribute constrains the ability to substitute members of a substitution group. The `block` attribute can also constrain the ability to specify a derived structure type (via `xsi:type`) in an XML instance. See Section 8.2.3 for a comprehensive discussion on blocking.

so, in both complexType and element, use this attribute to control in XML document, not deal with schema document. use final for control with in schema document

Attribute Overview

element: block

Value:	A space-delimited list containing one or more of 'extension', 'restriction', or 'substitution'. The value '#all' is analogous to a list containing all three values.
Default:	If the element type does not specify a block attribute, it inherits the value of the blockDefault attribute of the schema element.
Constraints:	None.
Required:	No.

Listing 8.4 portrays several complete substitution groups. The block on A (that is, 'block="restriction"') prevents the structure type of member element types from being restriction derivations. Specifically, the structure types of the member element types are either identical (as for B) or an extension (as for C). In addition, the XML instance cannot specify a restriction of the structure type with xsi:type.

The block on C (that is, 'block="substitution"') prevents substituting member element types (in this case, just D) from being a valid substitution for C. However, D is still, transitively, a valid substitution for A.

For clarity, note that blocking with 'substitution' is not the same as blocking with 'restriction extension', as the structure types of the member element types in a substitution group can be identical.

8.3.1.3 The default Attribute of an element Element

An element type can specify a default value for element instances by specifying the default attribute. An XML parser substitutes this default value when an element instance does not explicitly specify a value.

Attribute Overview

element: default

Value:	Any string that conforms to the attribute's type.
Default:	None.
Constraints:	The default and fixed attributes are mutually exclusive.
Required:	No.

The global element type effectiveDate has a default value of January 01, 1900.

```
<xsd:element name="effectiveDate"
             type="xsd:date"
             default="1900-01-01"/>
```

Given the preceding element type, the following element with no explicit value is valid in an XML instance:

```
<effectiveDate></effectiveDate>
```

Of course, the XML instance can have a value:

```
<effectiveDate>1900-01-01</effectiveDate>
```

In either case, the XML parser should provide the same result: an `effectiveDate` element with the value '`1900-01-01`'

the default parser in JDK dont support this feature

Warning

Some of the XML schema parsers do not substitute the default value; instead, the element remains empty or creates an error.

8.3.1.4 The `final` Attribute of an `element` Element

The `final` attribute prohibits any combination of extension, restriction, or substitution of the element type in a derived type.

Tip

You should have a thorough understanding of complex types (see Chapter 11) before attempting to understand element blocking. Further, you should have a comprehensive understanding of the `block` and `final` attributes of complex types (discussed in Sections 11.6.1.2 and 11.6.1.3, respectively).

Likewise, you should have a thorough understanding of substitution groups, discussed in Section 8.2.2.

Attribute Overview

`element: final`

Value:	A space-delimited list containing one or more of '`extension`', '`restriction`', or '`substitution`'. The value '#all' is analogous to a list containing all three values: a derivation must use the element type without any restrictions, extensions, or substitutions.
Default:	If the element type does not specify a `final` attribute, the element type inherits the value from the `finalDefault` attribute of the schema element.
Constraints:	None.
Required:	No.

Listing 8.6 is a substitution group whose base non-instantiable element type is the `sequenceID` element type.

Listing 8.6 A Substitution Group (`sequence.xsd`)

```
<xsd:element name="sequenceID"
             type="sequenceIDType"
             abstract="true">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element type is
            abstract:  the element must be
            replaced by a substitution group
            in either a derived type or
            an instance.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="unitID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for unit items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="bulkID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for bulk items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="assemblyID"
             type="sequenceIDType"
```

```
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for assembled items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:simpleType name="sequenceIDType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A Sequence ID is generated by
            the database.  Sequences are
            integers that start with "0"
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:nonNegativeInteger"/>
</xsd:simpleType>
```

The `baseCatalogEntryType` complex type includes a reference to the non-instantiable sequenceID:

```
<xsd:complexType name="baseCatalogEntryType"
                 abstract="true"
                   id="baseCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A catalog entry must have:
                * A database ID
                * Part Name
                * Part Number
                * Options available
                * Description
                * Price
                * Included Quantity when ordering
                  one item.
            The "baseCatalogEntryType" is
            non-instantiable:  a derived type must
            be created before a catalog
            entry can be instantiated.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence id="bacet-seq">
```

```
        <xsd:element ref="sequenceID"/>
        <xsd:element name="partName" type="partNameType"/>
        <xsd:element name="partNumber"
                     type="partNumberType"/>
        <xsd:element name="partOption"
                     type="partOptionType"
                     minOccurs="0"/>
        <xsd:element name="description"
                     type="catalogEntryDescriptionType"/>
        <xsd:group ref="priceGroup"/>
        <xsd:element name="includedQuantity"
                     type="xsd:positiveInteger"/>
        <xsd:element name="customerReview"
                     type="customerReviewType"
                     minOccurs="0"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
</xsd:complexType>
```

Subsequently, the `unitCatalogEntryType` substitutes the element type `unitID`, which is a valid substitution:

```
<xsd:complexType name="unitCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="unitCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A unit item contains nothing more
            or less than a basic catalog entry ID:
                * A database ID
                * Part Name
                * Part Number
                * Options available
                * Price
                * Included Quantity when ordering
                  one item (always one for unit items).
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent id="ucet.cc">
        <xsd:restriction base="baseCatalogEntryType">
            <xsd:sequence>
                <xsd:element ref="unitID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="unitPartNumberType"/>
```

```
                <xsd:element name="partOption"
                             type="partOptionType"
                             minOccurs="1"/>
                <xsd:element name="description
                             type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             fixed="1"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Finally, if `sequenceID` had a `final` attribute set to '`substitution`', as in the following example, the substitution in `unitCatalogEntryType` would have been invalid:

```
<xsd:element name="sequenceID"
             type="sequenceIDType"
             abstract="true"
             final="substitution">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element type is
            abstract:  the element must be
            replaced by a substitution group
            in either a derived type or
            an instance.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
```

8.3.1.5 The `fixed` Attribute of an `element` Element

The `fixed` attribute behaves very much like the `default` attribute: An XML parser substitutes the value of this attribute when the element instance has no explicit value. The only difference is that if the element specified in the XML instance has an explicit value, that value must be identical to the value specified by the `fixed` attribute.

Attribute Overview

`element: fixed`

if not match will get validation error. i checked.

if a element has **fixed** attribute it must not have **abstract.** i checked. So, both are mutually exclusive

Value:	Any string that conforms to the structure type of the element's element type.
Default:	None. i hope only abstract and default are mutually exclusive
Constraints:	The `final` attribute is mutually exclusive with the `default` attribute.
Required:	No.

The following example is the element type `country`, which has a `fixed` attribute set to 'US':

```
<xsd:element name="country" type="xsd:string" fixed="US"/>
```

Given the preceding element type, the following example with no explicit value is valid in an XML instance:

```
<country></country>
```

Of course, the XML instance can have the explicit value 'US':

```
<country>US</country>
```

In either case, the XML parser should provide the same result: a `country` element with the value 'US'.

Warning

Some of the XML parsers do not substitute the fixed value; instead, the element remains empty or creates an error.

Finally, for clarity, an XML instance cannot have a value other than US. The next example is invalid in an XML instance:

```
<country>United States</country>
```

8.3.1.6 The `form` Attribute of an `element` Element

The `form` attribute determines if a local element type requires qualification of the start- and endtags of an element instance. An element is *qualified* when a namespace prefix and a colon precede the respective name. For example, in Listing 8.7, `xse:nsQualified` is the *type name* of an element called '`nsQualified`'; the qualifier for the related namespace is '`xse`'. The subsequent examples use the `xse` namespace qualifier. The `form` attribute is not frequently used: Normally, the XML processor accesses the default value provided by the global `elementFormDefault` attribute of the `schema` element. The `form` attribute provides a mechanism to override the default qualification at the element type level. This attribute has no effect upon a global element type.

Attribute Overview

`element: form`

Value:	'`qualified`' or '`unqualified`'.
Default:	The element type inherits the value of the `elementFormDefault` attribute of the schema element.
Constraints:	The form attribute only applies to local element types, not global element types.
Required:	No.

The interaction of namespaces, default namespaces, and target namespaces is somewhat complicated. Because of this, the subsequent two examples include an entire schema and an entire XML instance respectively. Listing 8.7 provides an XML representation for the `formElementDemo` element. The `formElementDemo` element type in turn specifies three local element types that specify the `form` attribute as '`qualified`', '`unqualified`', and the default value.

Listing 8.7 A Complete XML Schema That Demonstrates the `form` Attribute

```
<xsd:element name="formElementDemo">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="nsUnqualified"
                         type="xsd:string"
                         form="unqualified"/>
            <xsd:element name="nsQualified"
                         type="xsd:string"
                         form="qualified"/>
            <xsd:element name="nsDefault"
                         type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

Listing 8.8 is an XML instance for the schema specified in Listing 8.7.

Listing 8.8 A Complete XML Instance That Demonstrates the `form` Attribute

```
<xse:formElementDemo
 xmlns:xse="http://www.XMLSchemaReference.com/examples"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.XMLSchemaReference.com/examples
                     http://www.XMLSchemaReference.com/examples/Ch08/
formElement.xsd">

<nsUnqualified>abc</nsUnqualified>
<xse:nsQualified>abc</xse:nsQualified>
<nsDefault>xyz</nsDefault>

</xse:formElementDemo>
```

Note that `nsUnqualified` is not qualified, and `nsQualified` is qualified—as specified by the respective `form` attributes. `nsDefault` is not qualified, because the `elementFormDefault` attribute of `schema` element specifies a default of '`unqualified`'

8.3.1.7 The `id` Attribute of an `element` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema. The example of the `city` element type in Section 8.1 provides an example of the `id` attribute.

Attribute Overview

`element: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Most of the examples in this chapter uniquely identify each element type.

Warning

Many of the existing XML parsers do not properly enforce the uniqueness of the `id`.

8.3.1.8 The `maxOccurs` Attribute of an `element` Element

The `maxOccurs` attribute determines the maximum number of times that an element instance can appear in an enclosing element.

Attribute Overview

`element: maxOccurs`

Value:	A non-negative integer, or '`unbounded`'.
Default:	'`1`'.
Constraints:	The `maxOccurs` attribute applies only to local element types.
Required:	No.

Refer to Listing 8.1, which demonstrates the complex type `addressType`. An element instance of `address` (whose structure type is `addressType`) may have one or two `addressLine` elements. The following example shows an

address that has a second address line (the apartment number):

```
<address>
    <street>88888 Mega Apartment Bldg</street>
    <street>Apt 5315</street>
    <city>New York</city>
    <state>NY</state>
    <country>US</country>
    <zip>10000</zip>
    <effectiveDate>2001-02-14</effectiveDate>
</address>
```

Setting the value of `maxOccurs` to '`unbounded`' permits an element instance to occur an unlimited number of times with respect to the enclosing element. Listing 8.9 allows for an XML instance that specifies a `customerList` to specify any number of `customer` records.

Listing 8.9 Specifying the `maxOccurs` Attribute for an Unbounded List (`catalog.xsd`)

```
<xsd:element name="customerList">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="customer"
                         minOccurs="0"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

> **Tip**
>
> Setting the value of the `maxOccurs` attribute to '`0`' is valuable when a restriction of a complex type must prohibit element instances of an element type specified in a base complex type.

Setting the value of `maxOccurs` to '`0`' is useful for overriding an existing `maxOccurs` value in a derived complex type. In the thematic catalog example, an assembly does not have any of the standard color or size options associated with all other types of catalog entries. Listing 8.10 specifies the `baseAssemblyCatalogEntryType` that removes the nested `partOption` element type specified by `baseCatalogEntryType`. Setting the value of the `maxOccurs` attribute to '`0`' prohibits an XML instance from containing a `partOption` element.

Listing 8.10 Removing an Element in a Derived Restricted Complex Type (`catalog.xsd`)

```
<xsd:complexType name="baseAssemblyCatalogEntryType"
                 abstract="true"
                 block="#all"
                 id="baseAssemblyCatalogEntryType.catalog.cType">
```

```
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An assembled item is similar to the
            other catalog entries.  The part number
            is restricted to an assembly number.
            In addition, there may be no options.
            Finally, a part list is also needed.
            Note that the "includedQuantity" has
            a default of one, but can be overridden
            in instances.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType"
                         id="bacet.rst">
            <xsd:sequence>
                <xsd:element ref="assemblyID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="assemblyPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"
                             minOccurs="0"
                             maxOccurs="0"/>
                <xsd:element name="description"
                             type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             default="1"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

8.3.1.9 The `minOccurs` Attribute of an `element` Element

The value of the `minOccurs` attribute determines the minimum number of times that an element instance can appear in an enclosing element.

Attribute Overview

element: minOccurs

it cannot have "unbounded" as value

Value:	A non-negative integer.
Default:	'1'.
Constraints:	The maxOccurs attribute applies only to local element types.
Required:	No.

Listings 8.9 and 8.10 provide examples of the minOccurs attribute.

8.3.1.10 The name Attribute of an element Element

The name attribute identifies the element type. The name of an element type may be referenced in one of three ways:

- An XML instance may specify instances of global element types.
- A complex type can specify a global element type with the ref attribute.
- A complex type can specify a local element type. An XML instance can specify this element type in the context of an element whose structure type is this complex type.

Attribute Overview

element: name

Value:	An NCName.
Default:	None.
Constraints:	The name attribute is mutually exclusive with the ref attribute.
Required:	Either name or ref is required.

Chapter 3 discusses namespaces. Chapter 7 describes how a schema makes use of namespaces.

8.3.1.11 The nillable Attribute of an element Element

When an element type specifies that an element instance is nillable, the XML parser does not require that element instances have a value. Section 8.2.5 provides a complete discussion on nillable element types.

Attribute Overview

element: nillable

Value:	A Boolean (that is, 'true' or 'false')
Default:	'false'

Constraints:	None.
Required:	No.

Listing 8.5 provides an example of an element type that specifies the `nillable` attribute.

8.3.1.12 The `ref` Attribute of an `element` Element

The value of a `ref` attribute is a global element type. In particular, a complex type specifies nested element types with local element types or references to global element types with the `ref` attribute.

Attribute Overview

`element: ref`

Value:	A QName that refers to global element type.
Default:	None.
Constraints:	There are several constraints, itemized in the following list: • The qualified name must refer to a global element type. • The `ref` attribute is mutually exclusive with `name`. • Allowed: Only the `minOccurs` and `maxOccurs` attributes are permitted. Only the `annotation` element may be used as content. • Disallowed: The `nillable`, `default`, `fixed`, `form`, `block`, and `type` attributes. The elements `complexType`, `simpleType`, `key`, `keyref`, and `unique` are also prohibited as content.
Required:	Either `name` or `ref` is required.

The `ref` attribute provides a way to incorporate a global element type into one or more complex types. Regardless, there is only one XML representation of the element type. This is particularly useful if the element type has lots of attributes, restrictions, or other detailed functionality. The `existingCustomer` element type, specified in Listing 8.11, references the global `phoneNumber` element type, specified in Listing 8.5.

Listing 8.11 Referencing a Global Element Type (`address.xsd`)

```
<xsd:element name="existingCustomer">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An existing customer consists of a
            name, phone number, address, and some
            keys from the database.  This element
            is restricted from further extensions.
```

```
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
            <xsd:element ref="phoneNumber"
                         minOccurs="1"
                         maxOccurs="unbounded"/>
            <xsd:element ref="address"
                         minOccurs="1"
                         maxOccurs="unbounded"/>
        </xsd:sequence>
        <xsd:attribute name="customerID"
                       type="xsd:string"
                       use="required"/>
        <xsd:attribute name="alternateKey"
                       type="xsd:string"
                       use="optional"/>
        <xsd:attribute ref="sequenceID"/>
    </xsd:complexType>
</xsd:element>
```

An element type can specify other attributes along with the `ref` attribute, as demonstrated by the use of the `minOccurs` and `maxOccurs` attributes associated with the reference to `phoneNumber` in Listing 8.11.

8.3.1.13 The `type` Attribute of an `element` Element

The `type` attribute determines the structure type associated with an element type. The value of the `type` attribute is a built-in datatype (such as `token` or `decimal`), a simple type derived from a built-in datatype, or a complex type.

Attribute Overview

`element: type`

Value:	A QName that refers to a global simple or complex type.
Default:	The built-in `anyType`—that is, no constraints on content.
Constraint:	The `type` attribute is mutually exclusive with a local `simpleType` or `complexType`.
Required:	No.

There are two ways to associate a structure type with an element type. The first is to specify the structure type with a `type` attribute (most of the examples in this chapter). The other is to locally specify a simple or complex type by nesting a `simpleType` or `complexType` element. Refer to Listing 8.2, which specifies the element type `POBox`

with a local simple type.

Tip

When an element type provides a `substitutionGroup` attribute, the type attribute is not required. When an element type that is part of a substitution group does not specify a structure type, the element type inherits the structure type from the base element type. The base element type is the element type specified by the value of the `substitutionGroup` attribute.

8.3.1.14 The `substitutionGroup` Attribute of an `element` Element

The `substitutionGroup` attribute specifies an element type to be substitutable for an another element type. Section 8.2.2 provides a complete discussion on the general functionality of substitution groups.

Attribute Overview

`element: substitutionGroup`

Value:	A QName that refers to a global element type.
Default:	None.
Constraints:	A `block` or `final` attribute in a base element type might prohibit substitutions.
Required:	No.

Listing 8.2 is a complete substitution group that includes the base non-instantiable `addressLine` and the valid substitutions `POBox`, `pmb`, and `street`.

Tip

The use of `final` or `block` attributes on base elements might prohibit substitutions in certain circumstances.

8.3.2 Content Options for an `element` Element

An element type has a number of content options. Table 8.2 itemizes the valid content options. The elements representing the content options for `element` elements are also content options for many other elements. Therefore, this section provides no explicit detail on any of the options. Instead, the ensuing table references appropriate chapters of this book.

Table 8.2. Content Options for an `element` Element

Element	*Description*	*Reference*
annotation	The annotation element provides a way to document schema elements.	Section 7.5
simpleType	A local simple type. Most element types have a local simpleType element, a local complexType element, or a reference to a global simple or complex type via the type attribute.	Chapter 10
complexType	A local complex type. Most element types have a local simpleType element, a local complexType element, or a reference to a global simple or complex type via the type attribute.	Chapter 11
key	A key element asserts the uniqueness of a set of elements in the XML instance. In addition, the key element asserts the existence of the elements in the XML instance.	Chapter 13
keyref	A keyref element asserts that the values specified by a set of elements in an XML instance exist as values of elements identified by either a key or unique element.	Chapter 13
unique	A unique element asserts uniqueness of a set of elements in the XML instance.	Chapter 13

The content pattern for the element element is:

what is difference between key and unique ?

```
annotation? (simpleType | complexType)? (unique | key | keyref)*
```

8.4 The any Element

this tag is very good for to give different structure type for different name space. this is also some what run time polimorphism

The any element provides a mechanism for specifying elements with what the XML Schema Recommendation calls a wildcard. The element wildcard permits a complex type to loosely specify how an XML validator validates elements in an XML instance. This is in sharp contrast to the normal element element, which succinctly specifies element type.

An element wildcard generally specifies a set of namespaces against which the XML validator may validate. The XML validator searches each namespace for global element types that might correspond to the elements referenced in the XML instance.

Listing 8.12 is an element type that specifies the XML instance may contain a reference to a set of any known element types in any known namespaces. For demonstration purposes, assume that eStrictAnyNS is in a target namespace that is bound to the 'aeDemo' prefix. For clarity in later discussion, 'aeDemo' is bound to:

http://www.XMLSchemaReference.com/examples/Ch08/anyElementDemo

Listing 8.12 Specifying an any Element

i hope ANY element used with complex type.

```
<xsd:element name="eStrictAnyNS">
    <xsd:complexType>
        <xsd:choice minOccurs="0" maxOccurs="unbounded">
            <xsd:any id="aaStrictAny"
                     processContents="strict"
                     namespace="##any"/>
```

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

To demonstrate how to use the `eStrictAnyNS` element described in Listing 8.12, suppose that the following elements are also in a namespace that is bound to the '`aeDemo`' prefix:

```
<xsd:element name="e1" type="xsd:token"/>
<xsd:element name="e2" type="xsd:token"/>
<xsd:element name="e3" type="xsd:token"/>
<xsd:element name="e4" type="xsd:token"/>
<xsd:element name="e5" type="xsd:token"/>
<xsd:element name="e6" type="xsd:token"/>
```

since, it accept any name space

The XML instance might contain the following element:

```
<aeDemo:eStrictAnyNS>
    <aeDemo:e1>value1</aeDemo:e1>
    <aeDemo:e2>value2</aeDemo:e2>
</aeDemo:eStrictAnyNS>
```

The remainder of this section provides detail on the attribute and content options available for the `any` element.

8.4.1 Attributes of an any Element

The attributes of an `any` element are complicated. Table 8.3 provides a summary of the attributes for an `any` element.

Table 8.3. Attribute Summary for an any Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`namespace`	The value of the `namespace` attribute specifies a list of namespaces that might provide global element types for validating an element in an XML instance.
`processContents`	The value of the `processContents` attribute determines whether the XML validator validates the *typenames* of elements against element types, and the elements' content against the element types' structure types.

8.4.1.1 The `id` Attribute of an `any` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

any: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within the set of schema documents that comprise an XML schema.
Required:	No.

Listing 8.12 describes an `any` element that specifies an `id` attribute.

8.4.1.2 The namespace Attribute of an any Element

The value of the `namespace` attribute specifies the namespaces that an XML validator examines to determine the validity of an element in an XML instance. The element in the XML instance must correspond to a global element type specified in one of the namespaces, but only if the `processContents` attribute is `strict` (see Section 8.4.1.3).

Attribute Overview

any: namespace

this '##other' is used in schema definition of SOAP

Value:	'`##any`', '`##other`', or a space-delimited list containing any or all of various URIs, '`##targetNamespace`', or '`##local`'.
Default:	'`##any`'.
Constraints:	None.
Required:	No.

An element in an XML instance must correspond to global element types specified in the namespaces that may correspond to a specified URI, or the constants defined in the following list:

- The value '`##any`' indicates any of the namespaces available to the XML instance.
- The value '`##targetNamespace`' indicates the namespace specified by the `targetNamespace` attribute of the `schema` element of the current schema.
- The value '`##other`' indicates a namespace that is any namespace *except* the namespace specified by the `targetNamespace` attribute of the `schema` element of the current schema.
- The value '`##local`' indicates explicitly no namespace. Of course, the element in the XML instance must be unqualified.

Note that Listing 8.12 demonstrates the use of the `namespace` attribute. Furthermore, note that the value of the `namespace` argument could be '`##targetNamespace`' or '`http://www.XMLSchemaReference.com/`

examples/Ch08/anyElementDemo'—instead of '##all'—with the same effect, because the element e1 exists in the target namespace.

8.4.1.3 The processContents Attribute of an any Element

The processContents attribute interacts with the namespace attribute. In particular, some of the values applicable to the processContents attribute negate the value of using the namespace attribute because the XML validator either ignores the namespace or does minimal validation against it.

Attribute Overview

any: processContents

Value:	One of 'lax', 'skip', or 'strict'.
Default:	'strict'.
Constraints:	None.
Required:	No.

The processContents attribute has one of three values. These values have the following meaning:

- strict: The XML validator enforces that an element in an XML instance validates against an element type in one of the namespaces specified by the namespace attribute.
- skip: Other than enforcing that an element is well formed, the XML validator does not attempt to validate the element against an element type.
- lax: The XML validator validates the element when possible. When the validator locates a corresponding element type, the element must conform to the element type's structure type. Conversely, when there is no corresponding element type, the validator deems the element and contents valid.

Listing 8.12 demonstrates the use of the processContents attribute.

8.4.2 Content Options for an anyAttribute Element

Table 8.4 identifies the only content option available to a wildcard: the ubiquitous annotation element.

Table 8.4. Content Options for an anyAttribute Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the any element is:

annotation?

CONTENTS

CONTENTS

Chapter 9. Attribute Types

31 - may - 2008

26 pages

IN THIS CHAPTER

we are going to read 3 tags
1. attribute
2. attributeGroup
3. anyAttribute

An *attribute type* specifies constraints for corresponding attributes in an XML instance. The main purpose of an attribute type is to associate an attribute name with a structure type. The structure type of an attribute type is always a simple type. The structure type may be global, or the structure type may be local to the attribute type. The XML representation of an attribute type is an `attribute` element.

Because the structure type of an attribute is always a simple type, an attribute type cannot specify nested element types or attribute types (only a complex type—or, indirectly, an element type whose structure type is a complex type—may specify nested element types and attribute types). Other than this limitation, an attribute type is very similar to an element type.

In XML, an attribute can have only one occurrence per element; there is no notion of number of occurrences. More precisely, an element may specify at most one occurrence of an attribute.

A complex type specifies attribute uses by using an attribute group. An *attribute use* is a local attribute type, or a reference to a global attribute type. A *named attribute-use group* is nothing more than a named collection of attribute uses. Multiple complex types may reference the same named attribute-use group. However, the scope of the attribute types changes: The attribute types become local to the enclosing complex type. A named attribute-use group provides modularity: If the named attribute-use group changes, all the referencing complex types conceptually specify the new set of attribute types. The XML representation of a named attribute-use group is an `attributeGroup` element.

An *attribute wildcard* can loosely specify attributes. An attribute wildcard specifies a namespace from which the corresponding element in an XML instance selects global attribute types. The XML representation of an attribute wildcard is an `anyAttribute` element.

9.1 An Example of an Attribute Type

Attribute types are global or local. Listing 9.1 portrays the complex type `businessCustomer`, which

incorporates the global `sequenceID` attribute type as well as local `customerID` and `primaryContact` attribute types:

Listing 9.1 Local and Global Attribute Types (`address.xsd`)

```
<xsd:complexType name="businessCustomerType">
    <xsd:sequence>
        <xsd:element name="name" type="xsd:string"/>
        <xsd:element ref="phoneNumber"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
        <xsd:element name="URL"
                     type="xsd:token"
                     minOccurs="0"
                     maxOccurs="1"/>
        <xsd:element ref="address"
                     minOccurs="1"
                     maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="customerID"
                   type="xsd:token"
                   use="required"/>
    <xsd:attribute name="primaryContact"
                   type="xsd:token"
                   use="optional"/>
    <xsd:attribute ref="sequenceID"/>
</xsd:complexType>

<xsd:attribute name="sequenceID"
               type="sequenceIDType"
               use="optional"
               id="address.attr.sequenceID"/>
```

The XML representation of an element type whose structure type is the complex type `businessCustomerType` might look like the following:

```
<xsd:element name="businessCustomer"
             type="businessCustomerType"/>
```

The file `address.xsd` provides detail on the `address` element type. Given the preceding element type, the following is valid in an XML instance:

```
<businessCustomer customerID="SAM132F57">
    <name>Cliff Binstock</name>
```

```
    <phoneNumber>503-555-0000</phoneNumber>
    <address>
        <street>123 Gravel Road</street>
        <city>Nowheresville</city>
        <state>OR</state>
        <country>US</country>
        <zip>97000</zip>
        <effectiveDate></effectiveDate>
    </address>
</businessCustomer>
```

Because the `primaryContact` and `sequenceID` are optional, the following element, which specifies these two attributes in addition to the required `customerID`, is also valid:

```
<businessCustomer customerID="SAM132F58"
                  primaryContact="Ellie"
                  sequenceID="88742">
    <name>Ellen Boxer</name>
    <phoneNumber xsi:nil="true"/>
    <address zipPlus4="20000-1234">
        <POBox>123</POBox>
        <city>Small Town</city>
        <state>VA</state>
        <country>US</country>
        <zip>20000</zip>
        <effectiveDate>2001-02-14</effectiveDate>
    </address>
</businessCustomer>
```

9.2 An Example of a Named Attribute-use Group

A *named attribute-use group* provides a way to specify a set of attribute types that presumably apply to multiple complex types. Listing 9.2 portrays the named attribute-use group `saleAttributeGroup`. In addition, the listing demonstrates the complex types `freePriceType` and `salePriceType`, both of which reference `saleAttributeGroup`.

Listing 9.2 A Named Attribute-use Group (`pricing.xsd`)

```
<xsd:attributeGroup name="saleAttributeGroup"
                    id="pricing.sale.ag">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything that is on sale (or free,
```

```
            which is a type of sale) must
            have an authorization defined.
            This is someone's name,
            initials, ID, etc.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="employeeAuthorization" type="xsd:token"/>
    <xsd:attribute name="managerAuthorization" type="xsd:token"/>
</xsd:attributeGroup>

<xsd:complexType name="freePriceType"
                 block="#all"
                 final="#all"
                 id="freePriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything that is free has no
            value (i.e., price), but must
            have an authorization code.
            This is a complex type with
            "empty" content.
                 — Shorthand Notation —
</xsd:documentation>
    </xsd:annotation>
    <xsd:attributeGroup ref="saleAttributeGroup"/>
</xsd:complexType>

<xsd:complexType name="salePriceType"
                 block="#all"
                 final="extension"
                 id="salePriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything on sale must have a price
            and an authorization
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="dollarPriceType">
            <xsd:attributeGroup ref="saleAttributeGroup"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

i hope , attributeGroup **cannot** be extend or restriction. Yes, it cannot extended. simply it can be referenced

The XML representation of an element type whose structure type is the complex type `freePriceType` might look like the following:

```
<xsd:element name="freePrice" type="freePriceType"/>
```

Given the preceding element type, the following is valid in an XML instance:

```
<freePrice employeeAuthorization="CB"
           managerAuthorization="ALB"/>
```

9.3 An Example of `anyAttribute`

The `anyAttribute` element provides a mechanism to loosely specify attributes. The `anyAttribute` element is analogous to the `any` element, which provides a similar mechanism for loosely specifying elements. Listing 9.3 demonstrates a simple complete schema for specifying attributes `a1` through `a6`. The element type `eStrictAnyNS` specifies that a corresponding XML instance may reference any global attribute specified in the http://www.XMLSchemaReference.com/examples namespace. The entire schema is included in Listing 9.3 because the namespaces are an integral part of the attribute wildcard.

Listing 9.3 An Attribute Wildcard (`anyAttributeDemo.xsd`)

```
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     targetNamespace="http://www.XMLSchemaReference.com/examples/Ch09/
anyAttributeDemo"

     elementFormDefault="qualified"
     attributeFormDefault="qualified">

<xsd:attribute name="a1" type="xsd:token"/>
<xsd:attribute name="a2" type="xsd:token"/>
<xsd:attribute name="a3" type="xsd:token"/>
<xsd:attribute name="a4" type="xsd:token"/>
<xsd:attribute name="a5" type="xsd:token"/>
<xsd:attribute name="a6" type="xsd:token"/>

<xsd:element name="anyAttributeDemo">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="eStrictAnyNS">
                <xsd:complexType>
                    <xsd:anyAttribute id="aaStrictAny"
                                      processContents="strict"
```

anyAttribute can be only one in local or top level complex type

any global attribute with any name space

```
                                    namespace="##any"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="eStrictURI">
                <xsd:complexType>
                    <xsd:anyAttribute
                        processContents="strict"
                        namespace=
    "http://www.XMLSchemaReference.com/examples/Ch09/anyAttributeDemo"/
>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="eStrictTargetNamespace">
                <xsd:complexType>
                    <xsd:anyAttribute
                        processContents="strict"
                        namespace="##targetNamespace"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="eStrictOtherNS">
                <xsd:complexType>
                    <xsd:anyAttribute processContents="strict"
                                      namespace="##other"/>
                </xsd:complexType>
            </xsd:element>
            <xsd:element name="eLaxAnyNS">
                <xsd:complexType>
                    <xsd:anyAttribute id="aaLaxAny"
                                      processContents="lax"
                                      namespace="##any"/>
                </xsd:complexType>
            </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>

</xsd:schema>
```

An entire XML instance that corresponds to the XML schema specified in Listing 9.3 might look like the following:

```
<aaDemo:anyAttributeDemo
 xmlns:aaDemo="http://www.XMLSchemaReference.com/examples/Ch09/
anyAttributeDemo"
```

```
 xmlns:aaList="http://www.XMLSchemaReference.com/examples/Ch09/
anyAttributeList"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.XMLSchemaReference.com/examples/Ch09/
anyAttributeDemo
http://www.XMLSchemaReference.com/examples/Ch09/anyAttributeDemo.xsd
http://www.XMLSchemaReference.com/examples/Ch09/anyAttributeList
http://www.XMLSchemaReference.com/examples/Ch09/anyAttributeList.xsd">

<aaDemo:eStrictAnyNS aaDemo:a1="value1"
                     aaDemo:a2="value2"/>

<aaDemo:eStrictURI aaDemo:a3="value3"/>

<aaDemo:eStrictTargetNamespace aaDemo:a4="tns" aaDemo:a6="6"/>

<aaDemo:eStrictOtherNS aaList:o1="1"/>

<aaDemo:eLaxAnyNS aaDemo:a1="value1"
                  aaDemo:a2="value2"
                  aaDemo:totallyRandom="what?"/>

</aaDemo:anyAttributeDemo>
```

9.4 Concepts and Observations Regarding Attribute Types

~~Attribute types have fewer options than element types, simple types, or complex types. Because of the simplicity of attribute types, this section covers only three concepts: when to use an attribute type versus an element type, global and local attribute types, and how attribute types interact with namespaces in the corresponding XML instances.~~

9.4.1 When to Use an Attribute Type

~~In many XML schemas, the use of an attribute type versus an element type is an arbitrary choice. In the following example, the customer's ID is an attribute type:~~

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

Given the preceding element type, the following is valid in an XML instance:

```
<customer customerID="SAM132E57"/>
```

Alternatively, the next example shows a customer's ID specified as an element type:

```
<xsd:element name="customer">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="customerID" type="xsd:string"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
```

When the customer's ID is an element type, as specified by the preceding `customer`, the following is valid in an XML instance:

```
<customer>
    <customerID>"SAM132E57"</customerID>
</customer>
```

Neither solution is technically superior. The following list contains some guidelines to help identify good candidates for storing data in XML attributes:

- Use attribute types when the data uniquely identifies the element. The customer ID is a good example of this.
- An element may not specify multiple occurrences of an attribute.
- Attribute types specify only simple content. If the data requires nested element types—or if nested element types are likely to become a future requirement—use an element type. Element types are much more flexible. Altering the XML instance generators to add children to elements and altering the corresponding element types is much easier than conceptually recasting attribute types as element types. The schema writer should take into consideration maintenance efforts for software developers.
- Finally, and conversely, humans are more adept at reading attributes than additional elements: The number of lines in an XML instance tends to expand when elements—not attributes—contain the data. If people frequently read or validate the XML instances, the schema writer might consider using more attribute types and fewer element types.

9.4.2 Global and Local Attribute Types

Attribute types can be global or local. Typically, multiple complex types reference a global attribute type. This reference to a global attribute type can be direct via the `ref` attribute or indirect via a named attribute-use group. Listing 9.1 portrays the XML representation of both global and local attribute types.

9.4.3 Namespaces and Attribute Types

This section discusses how the XML instance specifies namespaces for attributes (that is, how to locate, for validation purposes, the attribute types discussed in this chapter). Chapter 3 discusses how to determine which attribute types in a schema reside in which namespace.

In an XML instance, the namespace of an attribute must match the namespace identified by the attribute type. Typically, attribute types do not specify the target namespace: Normally, the `schema` element specifies the target namespace. If a namespace is not specifically identified, and the target namespace is not specified, the attribute must be unqualified (that is, specified without a namespace).

In an XML instance, an attribute may be *qualified* (namespace specified) or *unqualified* (namespace not specified). An XML validator determines whether the XML instance requires a namespace by evaluating the following rules in order:

- The attribute must be appropriately qualified when the attribute type specifies a `form` attribute (which has a value of '`qualified`' or '`unqualified`').
- When the attribute type does not specify the `form` attribute, the `attributeFormDefault` attribute (which also has a value of '`qualified`' or '`unqualified`') of the `schema` element enclosing the attribute type determines whether the name of the attribute instance is qualified.
- When the attribute type does not specify the `form` attribute, and the schema does not specify the `attributeFormDefault` attribute, the attribute must be unqualified. In other words, the default value for `attributeFormDefault` is '`unqualified`'.

Section 9.5.1.3, along with the corresponding Listings 9.4 and 9.5, provides an example of how the `form` attribute of attribute types and the `attributeFormDefault` attribute of the `schema` element work.

9.5 The `attribute` Element

Attribute types do not have the complexity of element types, simple types, or complex types. In particular, an attribute type has many possible attributes but very few content options.

9.5.1 Attributes of an `attribute` Element

The XML representation of an attribute type is an `attribute` element. Table 9.1 summarizes the attributes of an `attribute` element.

Table 9.1. Attribute Summary for an `attribute` Element

Attribute	*Description*

`default`	The value of a `default` attribute must be a value that conforms to the structure type of the attribute type. When the XML validator encounters an element whose element type specifies this attribute type and the element does not specify the corresponding attribute, the XML validator inserts the attribute with this default value into the infoset.
`fixed`	This is similar to the `default` attribute, except that when the element in the corresponding XML instance does specify an attribute, the value of that attribute must match the value specified by the `fixed` attribute.
`form`	The value of the `form` attribute determines if a namespace prefix must or must not precede the reference to a globally defined attribute type.
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`name`	The value of the `name` attribute is the name used to reference an attribute type in the rest of the schema or in an XML instance.
`type`	The value of the `type` attribute is the simple type, or structure type, that constrains an attribute type. This simple type may be a built-in datatype.
`use`	The value of the `use` attribute determines when an attribute is required, prohibited, or optional in an XML instance.
`ref`	The value of the `ref` attribute is a global attribute type. This reference permits multiple complex types to specify the same global attribute type.

it cannot be complex type

9.5.1.1 The `default` Attribute of an `attribute` Element

The `default` attribute of an attribute type specifies the value, in an XML instance, of a missing attribute. When an element type specifies an optional attribute type with a default value, and the element instance does not specify an attribute, the abstract element acquires the attribute along with its `default` attribute.

Attribute Overview

`attribute: default`

Value:	Any string that conforms to the structure type of the attribute type.
Default:	None.
Constraints:	The `default` and `fixed` attributes are mutually exclusive. In addition, when an attribute type includes both the `use` and `default` attributes, the `use` attribute must have the value '`optional`'.
Required:	No.

yes, i checked

use - attribute itself does not in current schema. i hope it might be deprecated by W3C

Listing 9.4 is the `customerList` element type, which specifies the attribute type `source`, which in turn specifies a default value of '`Oracle`':

Listing 9.4 A Default Value for an Attribute Type (`catalog.xsd`)

```
<xsd:element name="customerList">
    <xsd:complexType>
        <xsd:sequence minOccurs="0"
                      maxOccurs="unbounded">
            <xsd:choice>
                <xsd:element name="businessCustomer"
                             type="businessCustomerType"/>
                <xsd:element name="privateCustomer"
                             type="privateCustomerType"/>
            </xsd:choice>
        </xsd:sequence>
        <xsd:attribute name="source"
                       type="xsd:string"
                       default="Oracle"/>
        <xsd:attribute name="deliverDataToCountry"
                       type="xsd:string"
                       fixed="US"/>
    </xsd:complexType>
</xsd:element>
```

The results of parsing the following `customerList` elements are identical:

```
<customerList>
</customerList>

<customerList source="Oracle">
</customerList>

<customerList country="US">
</customerList>

<customerList source="Oracle" country="US">
</customerList>
```

Warning

Some XML validators do not create an attribute with the default value; instead, the infoset continues to not have the attribute.

In any reference to `customerList`, the XML validator creates a `source` attribute with the value '`Oracle`' in the infoset.

9.5.1.2 The `fixed` Attribute of an `attribute` Element

The `fixed` attribute behaves very much like the `default` attribute. An XML validator creates the specified value in the infoset when the instance does not have an attribute that corresponds to the `name` attribute. The only difference between `default` and `fixed` is that if the XML instance does contain the attribute, the instance's attribute value must be identical to the value specified by the `fixed` attribute of the attribute type.

so, Fixed attribute act as SINGLE VALUED ENUM type.

Attribute Overview

`attribute: fixed`

default - the instance can have any value. but if not value there then only DEFAULT comes to play

Value:	Any string that conforms to the structure type of the attribute type.
Default:	None.
Constraints:	The `fixed` and `default` attributes are mutually exclusive.
Required:	No.

Listing 9.4 demonstrates the use of the `fixed` attribute.

Warning

Some XML validators incorrectly do not create an attribute when the attribute type specifies a `fixed` value; instead, the element continues to not have the attribute.

this is own !

In any reference to `customerList`, the XML validator inserts the attribute `deliverDataToCountry` with the value '`US`' into the infoset. If an XML instance specifies an attribute with an invalid value, the XML validator reports an error.

9.5.1.3 The `form` Attribute of an `attribute` Element

The `form` attribute determines if a local attribute type requires the name of an attribute instance to be qualified. An attribute name is *qualified* when a namespace prefix and a colon precede the local name. For example, in Listing 9.6, `xse:nsQualified` is the qualified name of the attribute `nsQualified`; the corresponding namespace is '`http://www.XMLSchemaReference.com/examples`'. The subsequent examples bind the qualifier '`xse`' to the namespace. The `form` attribute is not frequently used: The XML validator inserts the default value provided by the global `attributeFormDefault` attribute of the `schema` element. The `form` attribute provides a mechanism to override the default qualification at the attribute level. This attribute has no effect on a global attribute type.

this is contradiction between table 9.1

it affect only local attribute. but if global attribute referenced by element, it should prefix BECAUSE OF targetNamespace

Attribute Overview

attribute: form

Value:	'qualified' or 'unqualified'
Default:	The attribute type inherits the value of the attributeFormDefault attribute of the schema element.
Constraints:	The form attribute applies only to local attribute types, not global attribute types.
Required:	No.

The interaction of namespaces, default namespaces, and target namespaces is somewhat complicated. Because of this, the subsequent two examples include an entire schema and an entire XML instance, respectively. Listing 9.5 provides an XML representation of the abusedElement element type. The abusedElement element type specifies three local attribute types that specify the form attribute as 'unqualified', 'qualified', and the default value.

Listing 9.5 An XML Schema to Demonstrate the form Attribute

```
<xsd:schema
     xmlns:xsd="http://www.w3.org/2001/XMLSchema"
     xmlns:xse="http://www.XMLSchemaReference.com/examples"
     xmlns="http://www.XMLSchemaReference.com/examples"
     targetNamespace=
        "http://www.XMLSchemaReference.com/examples"
     elementFormDefault="unqualified"
     attributeFormDefault="unqualified">

<xsd:element name="formAttributeDemo">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="abusedElement">
                <xsd:complexType name="abusedElementType">
                    <xsd:attribute name="nsUnqualified"
                                   type="xsd:string"
                                   form="unqualified"/>
                    <xsd:attribute name="nsQualified"
                                   type="xsd:string"
                                   form="qualified"/>
                    <xsd:attribute name="nsDefault"
                                   type="xsd:string"/>
                </xsd:complexType>
            </xsd:element>
```

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

</xsd:schema>
```

Listing 9.6 demonstrates an XML instance that corresponds to the schema specified in Listing 9.5.

Listing 9.6 An XML Instance to Demonstrate the `form` Attribute

```
<xse:formAttributeDemo
 xmlns:xse="http://www.XMLSchemaReference.com/examples"
 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xsi:schemaLocation="http://www.XMLSchemaReference.com/examples
                     http://www.XMLSchemaReference.com/examples/Ch09/
formAttribute.xsd">
<abusedElement
    nsUnqualified="abc"
    xse:nsQualified="abc"
    nsDefault="abc"/>

</xse:formAttributeDemo>
```

Note that `nsUnqualified` is not qualified and `nsQualified` is qualified, as specified by the respective `form` attributes. `nsDefault` is not qualified because the `attributeFormDefault` attribute of the `schema` element specifies a default of '`unqualified`'

9.5.1.4 The `id` Attribute of an `attribute` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`attribute: id`

Value:	An ID
Default:	None.
Constraints:	Each `id` must be unique in an XML schema.
Required:	No.

Listing 9.1 portrays an example that specifies the `sequenceID` attribute type, which specifies an `id` attribute with a value of '`address.attr.sequenceID`'

Warning

Many of the validators on the market today do not properly enforce `id` uniqueness.

9.5.1.5 The `name` Attribute of an `attribute` Element

The `name` attribute of an attribute type is the name with which a complex type specifies an attribute type in an XML schema document. An element in an XML instance references this name when specifying attributes.

Attribute Overview

`attribute: name`

Value:	An NCName.
Default:	None.
Constraints:	The `name` and `ref` attributes are mutually exclusive.
Required:	Either the `name` or `ref` attribute is required.

Most of the listings in this chapter contain present attribute types that specify the `name` attribute.

9.5.1.6 The `type` Attribute of an `attribute` Element

The value of the `type` attribute specifies the structure type of the attribute type. The structure type must be a simple type. This simple type specifies the valid values for an attribute in an XML instance that corresponds to the attribute type. An attribute type specifies the simple type that represents valid values for a corresponding XML instance with either a `type` attribute or `simpleType` content.

Attribute Overview

type of an attribute can be ONLY either simple type or built-in type

`attribute: type`

Value:	An NCName.
Default:	The built-in `anyType`—that is, any (simple) content.

Constraints:	• The use of the `type` attribute is mutually exclusive with a local `simpleType` element. • The value of the `type` attribute must be the name of built-in datatype or the name of a custom global simple type.
Required:	No.

so, a attribute type can ONLY either built-in or simple type. but cannot be complex type

Listing 9.1 portrays the use of the `type` attribute with built-in datatypes (the `customerID` and `primaryContact` attribute types), as well as with a custom simple type (the `sequenceID` attribute type).

For comparison, Listing 9.8 portrays the XML representation of `addressType`, which specifies a local `zipPlus4`, which in turn specifies an anonymous `simpleType`

9.5.1.7 The `use` Attribute of an `attribute` Element

yes, attribute ELEMENT can be specified only in complex type.

The structure type of an element type might be a complex type. This complex type might specify an attribute type. This attribute type might specify a `use` attribute. The value of this `use` attribute determines whether an element instance (of the element type) might include an attribute instance (of the attribute type). The use options permit the attribute type to require, prohibit, or not constrain the use of the attribute instance. The `use` attribute has one of the following values:

- '`required`': The element instance must have the attribute.
- '`optional`': The element instance may or may not have the attribute.
- '`prohibited`': The element instance may not have the attribute.

Tip

A derived complex type specifies the '`prohibited`' option to remove an attribute type in a restriction of a base complex type that already specifies the attribute type as optional.

Attribute Overview

`attribute: use`

Value:	One of '`required`', '`optional`', or '`prohibited`'.
Default:	'`optional`'.
Constraints:	When an attribute type includes both the `use` and `default` attributes, the `use` attribute must have the value '`optional`'.
Required:	No.

yes, it make sense.

In Listing 9.1, both the primaryContact and sequenceID attribute types specify the use attribute as 'optional'; customerID specifies the use attribute as 'required'.

Listing 9.7 portrays the complex type privateCustomerType, which derives from businessCustomerType specified in Listing 9.1. Note that businessCustomerType specifies the primaryContact attribute type as optional; privateCustomerType restricts this attribute type to be prohibited.

Listing 9.7 Prohibiting an Attribute Type

```
<xsd:complexType name="privateCustomerType">
    <xsd:complexContent>
        <xsd:restriction base="businessCustomerType">
            <xsd:sequence>
                <xsd:element name="name" type="xsd:string"/>
                <xsd:element ref="phoneNumber"
                             minOccurs="1"
                             maxOccurs="unbounded"/>
                <xsd:element name="URL"
                             type="xsd:token"
                             minOccurs="0"
                             maxOccurs="0"/>
                <xsd:element ref="address"
                             minOccurs="1"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="customerID"
                           type="xsd:token"
                           use="required"/>
            <xsd:attribute name="primaryContact"
                           type="xsd:token"
                           use="prohibited"/>
            <xsd:attribute ref="sequenceID"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

so, simply not repeating a super type attribute is not remove. explicitly has to declare as 'prohibited'

The XML representation of the element types whose structure type is privateCustomerType might look like the following:

```
<xsd:element name="privateCustomer"
             type="privateCustomerType"/>
```

Despite the fact that `privateCustomerType` is a derivation of `businessCustomerType`, a `privateCustomer` cannot specify a `primaryContact` because of the constraint on the `privateCustomer` complex type that specifies `'use="prohibited"'`. Given the preceding element type, the following is a valid element in an XML instance:

```
<privateCustomer customerID="SAM01234P"
                 sequenceID="88743">
    <name>I. M. Happy</name>
    <phoneNumber>303-555-0000</phoneNumber>
    <phoneNumber>303-555-1111</phoneNumber>
    <address>
        <street>123 Main Street</street>
        <pmb>12345</pmb>
        <city>Metropolis</city>
        <state>CO</state>
        <country></country>
        <zip>80000</zip>
        <effectiveDate>2001-02-14</effectiveDate>
    </address>
</privateCustomer>
```

9.5.1.8 The `ref` Attribute of an `attribute` Element

A complex type specifies a global attribute type with the `ref` attribute. The XML schema specifies a global attribute type once; the XML schema references a global attribute type potentially many times.

Attribute Overview

`attribute: ref`

it make sense

Value:	A `QName` name that corresponds to an attribute type.
Default:	None.
Constraints:	An attribute type may not contain any of the `name`, `form`, or `type` attributes along with the `ref` attribute. Also, only the `annotation` element is valid as a content option: The `simpleType` content option is not valid.
Required:	The `attribute` element must contain either a `name` attribute or a `ref` attribute.

it is same as ref attribute of element

The `ref` attribute provides a reference to a global attribute type from within a custom complex type. Listing 9.1

portrays the custom businessCustomerType that references the global sequenceID attribute type.

9.5.2 Content Options for an attribute Element

Table 9.2 contains the content options for an attribute element, which are limited to annotation and simpleType.

Table 9.2. Content Options for an attribute Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.
simpleType	A local simple type. An attribute type specifies the simple type that represents valid values for an attribute instance with either a type attribute or simpleType content.

it can be used to specify any of 11 constraints to built-in type.

The content pattern for the attribute element is:

annotation? simpleType?

It is simply say its own content type

an ASSOCIATION of an element with one or more attributes can be only BY COMLEX TYPE

Listing 9.8 portrays the XML representation of the complex type addressType, which specifies a local zipPlus4 attribute type, which in turn specifies an anonymous simpleType.

Listing 9.8 An Attribute Type That Specifies an Anonymous simpleType (address.xsd)

```
<xsd:complexType name="addressType">
    <xsd:sequence>
        <xsd:element ref="addressLine"
                     minOccurs="1"
                     maxOccurs="2"/>
        <xsd:element name="city" type="xsd:string"/>
        <xsd:element name="state" type="xsd:string"/>
        <xsd:element name="country" type="xsd:string" fixed="US"/>
        <xsd:element name="zip" type="xsd:string"/>
        <xsd:element ref="effectiveDate"/>
    </xsd:sequence>
        <xsd:attribute name="zipPlus4"
                       use="optional">
            <xsd:simpleType>
                <xsd:restriction base="xsd:string">
                    <xsd:pattern value="[0-9]{5}-[0-9]{4}"/>
```

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

Tip

Chapter 10 provides information on specifying custom simple types.

The XML representation of an element type whose structure type is the complex type `addressType` might look like the following:

```
<xsd:element name="address" type="addressType"/>
```

Given the preceding element type, the following is valid in an XML instance:

```
<address zipPlus4="20000-1234">
    <POBox>123</POBox>
    <city>Small Town</city>
    <state>VA</state>
    <country>US</country>
    <zip>20000</zip>
    <effectiveDate>2001-02-14</effectiveDate>
</address>
```

9.6 The `attributeGroup` Element

A *named attribute-use group* specifies a set of attribute uses. An *attribute use* is either a local attribute type, or a reference to a global attribute type. The XML schema specifies a named attribute-use group once; the XML schema references the named attribute-use group potentially many times. The named attribute-use group provides a convenient mechanism for defining complex types that require the same or similar attribute types. Additionally, named attribute-use groups provide modularity: Simply modifying a named attribute-use group indirectly modifies the complex types that reference that named attribute-use group.

The XML representation of a named attribute-use group is the `attributeGroup` element. Listing 9.2 portrays the specification and the use of a named attribute-use group.

Tip

The scope of local attribute types—but not global attribute type references—enclosed in a named attribute-use group changes depending on context. Specifically, the attribute types become local to the enclosing complex type.

9.6.1 Attributes of an `attributeGroup` Element

Because a named attribute-use group is little more than a set of attribute types, there are not many attributes needed to specify a named attribute-use group. Table 9.3 summarizes the attributes of an `attributeGroup` element.

Table 9.3. Attribute Summary for an `attributeGroup` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`name`	The value of the `name` attribute is the name used to reference a named attribute-use group in the rest of the schema.
`ref`	The value of the `ref` attribute is a named attribute-use group. A complex type specifies the `ref` attribute to reference a named attribute-use group. A complex type with a reference to a named attribute-use group has identical functionality to a complex type that references the individual attributes specified by the named attribute-use group.

9.6.1.1 The `id` Attribute of an `attributeGroup` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`attributeGroup: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema .
Required:	No.

Listing 9.2 includes the XML representation of the `saleAttributeGroup` named attribute-use group, which has an `id` attribute with a value set to '`pricing.sale.ag`'

Warning

Many of the validators on the market today do not properly enforce `id` uniqueness.

9.6.1.2 The name Attribute of an attributeGroup Element

A complex type specifies a name to include a named attribute-use group. This name must match the value of the `name` attribute of a named attribute-use group.

Attribute Overview

attributeGroup: name

Value:	An NCName.
Default:	None.
Constraints:	The `name` and `ref` attributes are mutually exclusive.
Required:	The `attributeGroup` element must contain either a `name` attribute or a `ref` attribute.

Listing 9.2 portrays the `saleAttributeGroup`, as well as references to this named attribute-use group.

9.6.1.3 The ref Attribute of an attributeGroup Element

A complex type references a named attribute-use group with the `ref` attribute. An XML schema specifies a named attribute-use group once; that XML schema references a named attribute-use group potentially many times.

Attribute Overview

attributeGroup: ref

Value:	A QName that corresponds to a named attribute-use group.
Default:	None.
Constraints:	The `name` and `ref` attributes are mutually exclusive.
Required:	The `attributeGroup` element must contain either a `name` attribute or a `ref` attribute.

Listing 9.2 portrays the `saleAttributeGroup`, as well as references to this named attribute-use group.

9.6.2 Content Options for an attributeGroup Element

Other than an `annotation`, the content options for named attribute-use groups directly or indirectly specify

attribute types. Table 9.4 illuminates the possible ways to specify attribute types.

Table 9.4. Content Options for an `attributeGroup` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`attribute`	An `attribute` element specifies a local attribute type, or references a global attribute type.
`attributeGroup`	A named attribute-use group may reference another named attribute-use group.
`anyAttribute`	A named attribute-use group may specify attributes by identifying attribute wildcards.

The content pattern for the `attribute` element is:

```
annotation? (attribute | attributeGroup)* anyAttribute?
```

9.7 The `anyAttribute` Element

The `anyAttribute` element provides a mechanism for specifying attributes with what the XML Schema Recommendation calls a wildcard. The attribute wildcard permits a complex type to loosely specify how an XML validator validates attributes in an XML instance. This is in sharp contrast to the normal `attribute` and `attributeGroup` elements, which succinctly specify one or a set of attribute types, respectively.

An attribute wildcard generally specifies a set of namespaces against which the XML validator may validate. The XML validator searches each namespace for global attribute types that might correspond to the attributes referenced in the XML instance.

9.7.1 Attributes of an `anyAttribute` Element

The attributes of an `anyAttribute` element are complicated. Table 9.5 provides a summary of the attributes for an `anyAttribute` element.

Table 9.5. Attribute Summary for an `anyAttribute` Element

Attribute	*Description*

id	The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
namespace	The value of the namespace attribute specifies a list of namespaces that might provide global attribute types for validating a corresponding attribute in an XML instance.
processContents	The value of the processContents attribute determines whether the XML validator validates the names of the attributes against attribute types, and the attribute values against the attribute types' structure types.

9.7.1.1 The id Attribute of an anyAttribute Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

anyAttribute: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 9.3 portrays an example that specifies the id attribute for an anyAttribute element.

9.7.1.2 The namespace Attribute of an anyAttribute Element

The value of the namespace attribute specifies the namespaces that an XML validator examines to determine the validity of an attribute in an XML instance. The attribute in the XML instance must correspond to a global attribute type specified in one of the namespaces, but only if the processContents attribute is 'strict' (see Section 9.7.1.3).

Attribute Overview

anyAttribute: namespace

what is difference between ##targetNamespace and ##local ???

Value:	'##any', '##other', or a space-delimited list containing URIs, '##targetNamespace' or '##local'.
Default:	'##any'.
Constraints:	None.

Required:	No.

An attribute in an XML instance must correspond to global attribute types specified in the namespaces that may correspond to the constants defined in the following list:

- The value '`##any`' indicates any of the namespaces available to the XML instance.
- The value '`##targetNamespace`' indicates the namespace specified by the `targetNamespace` attribute of the `schema` element in the corresponding XML Schema.
- The value '`##other`' indicates a namespace that is any namespace *except* the namespace specified by the `targetNamespace` attribute of the `schema` element in the corresponding XML Schema.
- The value '`##local`' indicates that the attribute in the XML instance does not pertain to a known namespace. Furthermore, the attribute must be unqualified.

Note that Listing 9.3 demonstrates the use of the `namespace` attribute. Furthermore, note that the value of the `namespace` argument could be '`##targetNamespace`' or '`http://www.XMLSchemaReference.com/examples`'—instead of '`##all`'—with the same effect, because the attribute `a1` exists in the target namespace.

9.7.1.3 The `processContents` Attribute of an `anyAttribute` Element

The `processContents` attribute interacts with the `namespace` attribute. In particular, some of the values applicable to the `processContents` attribute negate the value of using the namespace attribute because the XML validator either ignores the namespace or does minimal validation.

Attribute Overview

`anyAttribute: processContents`

Value:	One of '`lax`', '`skip`', or '`strict`' .
Default:	'`strict`'.
Constraints:	None.
Required:	No.

The `processContents` attribute has one of three values. These values have the following meaning:

- '`strict`': The XML validator enforces that an element in an XML instance validates against an element type in one of the namespaces specified by the `namespace` attribute.
- '`skip`': Other than enforcing that an element is well formed, the XML validator does not attempt to validate the element against an element type.

- 'lax': The XML validator validates the element when possible. When the validator locates a corresponding element type, the element must conform to the element type's structure type. Conversely, when there is no corresponding element type, the validator deems the element and contents valid.

Listing 9.3 demonstrates the use of the processContents attribute.

9.7.2 Content Options for an anyAttribute Element

Table 9.6 identifies the only content option available to an attribute wildcard: the ubiquitous annotation element.

Table 9.6. Content Options for an anyAttribute Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the anyAttribute is:

```
annotation?
```

CONTENTS

Chapter 10. Simple Types

just 30 pages

start 1- June - 08

end 3 - june -08

IN THIS CHAPTER

this chapter talks about FOUR tag. simple tag and its content

1. simpleType
2. restriction
3. list
4. union

This chapter details how to write simple types. Simple types determine the range of possible values of an element or attribute in an XML instance. Simple types derive from either a built-in datatype or another derived simple type. The built-in datatypes fall mostly into the broad categories of strings, numbers, and dates. Chapter 12 contains more detail on the built-in datatypes.

A `simpleType` element is the XML representation of a simple type in an XML schema document. Unlike an element type or an attribute type, but like a complex type, an XML instance cannot directly contain an instance of a simple type. The simple type must be the structure type of an element type or an attribute type.

A simple type restricts the value of an element or attribute instance. A *constraining facet* specifies a restriction. Each simple type can have multiple constraining facets.

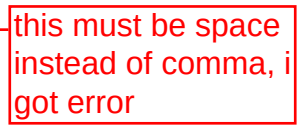
Warning

Some constraining facets affect the *lexical space* of a value; some affect the *value space*. Furthermore, an XML validator transliterates every value into its *canonical representation*. In order to avoid unexpected behavior, a thorough understanding of how constraining facet perform is imperative. See Section 10.4.1 for more information.

10.1 An Example of a Simple Type Derived from the Built-in `token` Datatype

The simpleType element restricts the value of another simple type. Listing 10.1 demonstrates the XML representation of partNameType. A part name represents a short description of a part. The name is limited to 40 characters, perhaps due to database or screen real estate limitations.

Listing 10.1 A Simple Type Derived from a Token (catalog.xsd)




```
<xsd:simpleType name="partNameType"
                final="list,union"
                id="catalog.partName.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A part name can be almost anything.
            The name is a short description.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token"
                     id="pnt-rst">
        <xsd:minLength value="1"/>
        <xsd:maxLength value="40"/>
    </xsd:restriction>
</xsd:simpleType>
```

An element type whose structure type is partNameType might look like the following element:

```
<xsd:element name="partName" type="partNameType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<partName>Short Description of Unit 1</partName>
```

10.2 An Example of a Pattern-constrained Simple Type

Another example of a simple type is the partNumberType in Listing 10.2. The partNumberType contains a pattern that determines the valid values for a part number in a corresponding XML instance. The annotation provides an explanation of the pattern.

Listing 10.2 A Pattern-constrained Simple Type (catalog.xsd)

```
<xsd:simpleType name="partNumberType"
                final="union"
                id="catalog.partNumber.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Declaration of a part number.
            Each part number consists of one to
            three alphabetic characters followed by
            one to eight digits. The following part
            numbers, for example, are valid:
                J1
                ABC32897
                ZZ22233344
        </xsd:documentation>
    </xsd:annotation>
        <xsd:restriction base="xsd:token">
            <xsd:pattern value="[A-Z]{1,3}\d{1,8}"/>
        </xsd:restriction>
</xsd:simpleType>
```

The next section demonstrates instantiating a derivation of `partNumberType`.

10.3 An Example of a Simple Type Derived from a User-derived Simple Type

The two previous examples demonstrate simple types derived from the built-in `token` type. Listing 10.3 portrays the `assemblyPartNumberType` derived from the `partNumberType` that is declared in Listing 10.2.

Listing 10.3 Restricting a User-derived Simple Type (`catalog.xsd`)

```
<xsd:simpleType name="assemblyPartNumberType"
                final="#all"
                id="catalog.assemblypartNumber.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An "assembly" represents a pre-built
            collection of unit items. The
            part number for an assembly
```

```
            always starts with "ASM."
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="partNumberType">
        <xsd:pattern value="ASM\d{1,8}"/>
    </xsd:restriction>
</xsd:simpleType>
```

An element type whose structure type is `assemblyPartNumberType` might look like the following element:

```
<xsd:element name="partNumber"
             type="assemblyPartNumberType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<partNumber>ASM9001</partNumber>
```

10.4 Concepts and Observations

Creating a simple type is easy. Nonetheless, there are a few things to consider when creating a new simple type. This section covers adding constraining facets to simple types, noting the lack of non-instantiable simple types, specifying the scope of simple types, and blocking of simple types.

Each simple type has a value space and a lexical space. Furthermore, every value in the value space has a canonical lexical representation. This section covers these concepts.

10.4.1 Constraining Facets

The Schema Recommendation limits the use of constraining facets to simple types. Constraining facets enforce additional restrictions to a simple type derived directly or indirectly from a built-in datatype.

Listing 10.1 contains a simple example of a constraining facet associated with the `partNameType`. The `partNameType` is a restriction of the built-in `token` type. The `minLength` and `maxLength` constraining facets restrict `partNameType` to a short description of 40 characters.

The constraining facets also apply to simple types that are lists or unions of other simple types. Sections 10.5.3, 10.6.3, 10.7.3, and 10.8.3 cover the constraining facets of `simpleType`, `restriction`, `list`, and `union`, respectively.

Tip

The `restriction` element provides functionality for adding constraining facets to a simple type.

A `restriction` element contains constraining facets. The previously itemized sections discuss the constraining facets associated with different types. Sometimes the constraining facets occur in places that are not always obvious. Lists and unions, for example, require further derivations to add constraining facets.

10.4.2 The Value Space

The *value space* defines the range of permissible values. The following simple type specifies an integer between -5 and 5, inclusive, by restricting the value space:

```
<xsd:simpleType name="valueAbsoluteLessEqualFiveType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This simple type specifies a range
            of integer values between -5 and 5,
            inclusive.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:integer">
        <xsd:minInclusive value="-5"/>
        <xsd:maxInclusive value="5"/>
    </xsd:restriction>
</xsd:simpleType>
```

The value space for this simple type contains the integers between -5 and 5. The lexical space automatically loses all numerals that don't name integers outside this range.

10.4.3 The Lexical Space

lexical means, it just characters. it is not yet interpreted as integer or float or string value. simply it is a token

The *lexical space* defines the range of character-string representations of the values. Compare the previous example with the next example, which directly constrains the lexical space with a `pattern`:

```
<xsd:simpleType name="lexicalAbsoluteLessEqualFiveType">
```

```
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This simple type specifies a range
            of integer values between -5 and 5,
            inclusive.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:integer">
        <xsd:pattern value="0|(-?[1-5])"/>
    </xsd:restriction>
</xsd:simpleType>
```

super regular expr.

This example differs from the previous one in that it disallows "redundant leading zero" representations such as '001'.

since it is not yet identified as number

~~In general, constraining facets should specify restrictions on the value space when appropriate. For example, specifying a pattern conceptually similar to the previous example, but for a *decimal* range between -5.0 and 5.0, is suspect at best.~~

10.4.4 The Canonical Lexical Representation

why i want to know about canonical lexical rep.. ?

it is useful only for float

Each value in the value space has a canonical lexical representation. The *canonical lexical representation* of a value is the "official" *lexical* representation of a value. For example, the canonical lexical representation of the float value having representations '5', '5.0', '0005.000', and '5.0E0'is '5.0E0'. Off hand, it appears that that canonical lexical representation is of no interest to the schema writer. On the contrary, the Schema Recommendation requires that all default values adhere to the following constraint:

If an element type or attribute type specifies a default or fixed value, that value must be valid with respect to the structure type.

This can lead to the presumably unintended situation where the canonical representation of the default value invalidates the simple type. The following scenario demonstrates this unfortunate situation. Listing 10.4 is a simple type that specifies a float value that might store dollar values.

Listing 10.4 A Simple Type to Store Dollar Values (`pricing.xsd`)

still i am not clear , even at second round study

```
<xsd:simpleType name="dollarType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
```

```
            This simple type specifies decimal
            values that conform to scientific
            notation.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:float">
<xsd:pattern value="[0-9]+\.[0-9][0-9]"/>
    </xsd:restriction>
</xsd:simpleType>
```

Note that the canonical form of a float requires exactly one digit before the decimal point.

The following element type is invalid, because the canonical lexical representation of the default value (even if represented as '`1.0E2`' in the element type) cannot conform to the pattern specified by `dollarType`:

```
<xsd:element name="dollar"
             type="dollarType"
             default="100.00"/>
```

so, there is no ABSTRACT attribute

10.4.5 Non-instantiable Simple Types

Simple types are always instantiable. The XML representation of a simple type cannot specify a non-instantiable simple type. ~~Furthermore, because a simple type does not contain element types or attribute types, there are no implicitly non-instantiable simple types. The inability to specify a simple type as non-instantiable means the XML representation of a schema cannot describe a simple type that an element cannot reference. For example, an element type is not supposed to associate the~~ `partNumberType` ~~in Listing 10.2 as a structure type. Instead, an element type is supposed to associate a derived type such as the~~ `assemblyPartNumberType`. ~~Unfortunately, there is no way to prevent an element type from directly using~~ `partNumberType`. ~~Conceptually,~~ `partNumberType` ~~is non-instantiable; in practice,~~ `partNumberType` ~~is always instantiable. To reiterate, the XML schema does not prohibit the ensuing element type:~~

```
<xsd:element name="partNumber" type="partNumberType"/>
```

Note that both complex types and element types may be non-instantiable.

~~Warning~~

~~The inability to declare a simple type non-instantiable could lead to unintended use of what~~

conceptually a non-instantiable base simple type. This unintended use would occur in the XML instance, such as the one portrayed in Section 10.4.2.

10.4.6 Global, Local, and Anonymous Simple Types

A simple type can be global or local. A local simple type can be anonymous; in other words, a local simple type does not require a name. Complex types, element types, attribute types, and other simple types can reference a global simple type. Sections 10.7 and 10.8 (which cover lists and unions, respectively) provide examples of both local and global simple types.

block only for complex type and element

10.4.7 Blocking

The `simpleType` element can have only the `final` attribute, not the `block` attribute. In general, the `final` attribute blocks the schema from further derivations of a simple type; the `block` attribute blocks a corresponding XML instance from specifying further derivations of a simple type. Presumably, because all simple types are instantiable, the World Wide Web Consortium (W3C) Schema Working Group found little value in supporting the `block` attribute.

10.5 The `simpleType` Element

The XML representation of a simple type is a `simpleType` element. A `simpleType` must contain exactly one of the following elements: `restriction`, `list`, or `union`. Because all three of the aforementioned elements are moderately complex, this chapter includes a section for each element. The `simpleType` and `restriction` elements cover the majority of simple types, which are frequently easy derivations of built-in datatypes.

10.5.1 Attributes of a `simpleType` Element

The `simpleType` element has only three attributes: `final`, `id`, and `name`. All three of these attributes behave similarly to the identically named attributes for the `element` element. Chapter 8 provides details on the `element` element. Table 10.1 provides an overview of the simple type attributes.

Table 10.1. Attribute Summary of a `simpleType` Element

Attribute	*Description*
`final`	The value of the `final` attribute is a space-delimited list containing any combination of '`list`', '`union`', or '`restriction`'. This further restricts derivations of simple types specifically `list`, `union`, or `restriction`, respectively. The value '`#all`' indicates that no further derivations are possible.
`id`	The value of the `id` attribute uniquely identifies an element.
`name`	The value of the `name` attribute is the name of a simple type. A derived simple type references this name as the base simple type. An element type references this name to associate a structure type.

10.5.1.1 The `final` Attribute of a `simpleType` Element

The `final` attribute restricts the ability to derive other types. Its values are defined as follows:

- '`list`': prevents a derived type from creating a list of the restricted type.
- '`union`': prevents a derived type from using the restricted type in a union.
- '`restriction`': prevents a derived type from using constraining facets to further restrict the restricted type.
- '`#all`': prohibits all further derivations.

i want see sample how to use list or union in derived simple type.

i will get in third round. i dont have net connection

Attribute Overview

`simpleType: final`

Value:	Any or all of '`list`', '`union`', or '`restriction`'. The value '`#all`' can be used as a shortcut for a list containing all three values.
Default:	In general, there is no default. However, a `schema` element can set a global default by setting the `finalDefault` attribute.
Constraints:	None.
Required:	No.

The `partNumberType` in Listing 10.2 has the `final` attribute set to `union`. Another simple type cannot be a union that contains `partNumberType` as a member type. Similarly, the `assemblyPartNumberType` in Listing 10.2 cannot have any derived classes, including those that

might further restrict the value of an element in a corresponding XML instance. Listings 10.1, 10.3, 10.6, 10.8, and 10.9 also contain examples that incorporate the `final` attribute.

Warning

Many existing XML parsers do not respect the `final` attribute of simple types or complex types.

10.5.1.2 The `id` Attribute of a `simpleType` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema. Listing 10.1 provides an example that associates an `id` attribute with a `simpleType` element.

Attribute Overview

`simpleType: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Most of the examples in this chapter uniquely identify the type.

Warning

Many of the existing XML parsers to not enforce the uniqueness of an `id`

10.5.1.3 The `name` Attribute of a `simpleType` Element

The value of `name` is the name used in the schema to reference a `simpleType`.

Attribute Overview

simpleType: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	Yes, unless the `simpleType` is local to an element type, another simple type, or a complex type—in which case the name is prohibited. A `simpleType` without a name is *anonymous*.

Section 10.4.3 discusses the usage and requirements of the `name` attribute. Listing 10.1 provides an example that associates the `name` attribute with a `simpleType` element.

10.5.2 Content Options for a `simpleType` Element

This section itemizes the possible child elements of a `simpleType` element. Table 10.2 provides a quick summary of all the content options.

Table 10.2. Content Options for a `simpleType` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`restriction`	The `restriction` element provides functionality for adding constraining facets to a derived simple type.
`list`	The `list` element provides functionality for creating a simple type that specifies a list of simple type values.
`union`	The `union` element provides functionality for creating a simple type that specifies a selection from a set of simple type values.

The content pattern for a `simpleType` element is:

only one can be there among three

```
annotation? (restriction | list | union)
```

10.5.3 Constraining Facets of a `simpleType` Element

there are 12 facts are there

Constraining facets can be associated with a simple type for three reasons:

- The user-derived simple type further restricts a built-in datatype or another, atomic user-derived simple type ultimately derived from a built-in datatype.
- The user-derived simple type restricts a list containing simple type values.
- The user-derived simple type restricts a union whose member types are other simple types.

For simple types ultimately derived as a restriction of a built-in datatype — as opposed to a list or union — the constraining facets apply to the base datatype. A constraining facet limits the values that are permissible in an XML instance. These limits are in addition to those constraining facets already associated with the base simple type. For example, the `partNameType` introduced in Listing 10.1 is a refinement, or restriction, of the built-in `token` type. Similarly, the values permissible for the `assemblyPartNumberType` presented in Listing 10.3 are constrained further than the `partNumberType` in Listing 10.2. Chapter 12 discusses constraining facets in detail.

Sections 10.7.3 and 10.8.3 illuminate the constraining facets of the `list` and `union` elements, respectively.

10.6 The `restriction` Element

The `restriction` element provides the functionality to add constraining facets to a simple type. For all simple types except those that represent lists and unions, the `simpleType` contains a `restriction`. In the case of a list or a union, a simple type derived from the list or union must add the restrictions.

i dont know how to test

10.6.1 Attributes of a `restriction`

A `restriction` has two attributes: `base` and `id` Table 10.3 provides a brief summary of the `base` and `id` attributes. The subsequent two sections discuss these attributes in detail.

Table 10.3. Attribute Summary for a `restriction` Element

Attribute	*Description*
`base`	The value of the `base` attribute identifies the base simple type, which is often a built-in datatype, of the current derived simple type.

id	The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

10.6.1.1 The base Attribute of a restriction Element

The value of a base attribute indicates the global base type. The base type must be a global simple type. The new simple type typically restricts the base simple type by adding constraining facets.

Attribute Overview

restriction: base

Value:	A QName.
Default:	None.
Constraints:	The base type must refer to a global simple type in the XML schema. The specification of the base attribute and the specification of a local simple type are mutually exclusive.
Required:	No.

Most of the listings in this chapter use the base attribute. In fact, most simple types, except for those specifying lists or unions, make use of the base attribute.

10.6.1.2 The id Attribute of a restriction Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

restriction: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.

Required:	No.

Listing 10.1 contains an example of a `restriction` that specifies an `id` attribute.

10.6.2 Content Options for a `restriction` Element

The content options for a `restriction` element, when nested within a `simpleType` element, are limited to an `annotation`, a `simpleType`, and a set of appropriate constraining facets. The base built-in datatype determines the appropriate constraining facets. Table 10.4 provides a summary of content options. Note that the entire chapter illuminates the limitations of an embedded `simpleType`. The sections on constraining facets portray multiple ways for applying constraining facets to restrictions, lists, and unions.

Table 10.4. Content Options for a `restriction` Element

The `enumeration` facet lists all allowed values. Applies to all simple types except `boolean`.

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`enumeration`	The value of an `enumeration` constraining facet specifies a possible (constant) value.
`fractionDigits`	The value of a `fractionDigits` constraining facet determines the number of digits after the decimal point in a decimal number.
`length`	The value of a `length` constraining facet specifies the exact length of a string of characters that must appear in an element instance, after normalization. This constraining facet also specifies the number of items in a list (see the caveats regarding `hexBinary` and `base64binary` datatypes in Section 12.4.3).
`maxExclusive`	The value of the `maxExclusive` constraining facet specifies an upper bound on a numeric value. This boundary excludes the value specified.
`maxInclusive`	The value of the `minInclusive` constraining facet specifies an upper bound on a numeric value. This boundary includes the value specified.
`maxLength`	The value of the `maxLength` constraining facet specifies the maximum number of characters in a string, after normalization or the maximum number of items in a list (see caveats on `hexBinary` and `base64Binary` in Section 12.4.3).

`minExclusive`	The value of the `minExclusive` constraining facet specifies a lower bound on a numeric value. This boundary excludes the value specified.
`minInclusive`	The value of a `minInclusive` constraining facet specifies a lower bound on a numeric value. This boundary includes the value specified.
`minLength`	The value of a `minLength` constraining facet specifies the minimum number of characters in a string, after normalization or the minimum number of items in a list.
`pattern`	The value of a `pattern` constraining facet specifies a regular expression often used to validate a character string. Specifically, the pattern constrains lexical representations.
`totalDigits`	The value of a `totalDigits` constraining facet specifies the total number of digits in a decimal number.
`simpleType`	The `simpleType` element specifies a local simple type.
`whiteSpace`	The value of a `whiteSpace` constraining facet determines the normalization of spaces, carriage returns, and line feeds in a string. This normalization frequently removes undesirable white space.

The content pattern for a `restriction` element is:

```
annotation? simpleType? (minExclusive |
                         minInclusive |
                         maxExclusive |
                         maxInclusive |
                         totalDigits |
                         fractionDigits |
                         length |
                         minLength |
                         maxLength |
                         enumeration |
                         whiteSpace |
                         pattern)*
```

maxLength, minLength, length, pattern, whitespace are work for STRING of characters

minExclusive, maxExlusive, minInclusive, minExclusive are work for NUMBERS

it might be deprecated by current impl

The content options for simple types vary according to which element includes the `restriction` element. Refer to Sections 10.7.3 and 10.8.3 for further clarification: These sections illuminate the constraining facets that apply to `list` and `union` elements, respectively. Chapter 12 portrays the applicability of constraining facets to each built-in datatype.

10.6.3 Constraining Facets of a `restriction` Element

The base built-in datatype associated with the simple type determines appropriate constraining facets of a `restriction` element. Chapter 12 provides a discussion on which constraining facets are applicable to which built-in datatype.

10.7 The `list` Element

A simple type can specify a space-delimited list consisting of values specified by other global or local simple types. An instance of a list declared in the context of a simple type has all the values contained within a single element:

```
<listElement>value1 value2 value3</listElement>
```

in the list we specify only data type of possible values in list in xml INSTANCE document.

Tip

in the ENUM we say possible VALUE itself not data type in SCHEMA file

The `list` element does *not* support the notion of a list comprised of repeating elements:

```
<element>value1</element>
<element>value2</element>
<element>value3</element>
```

Both the `element` element (covered in Chapter 8) and the `complexType` element (covered in Chapter 11) support repeating elements.

Listing 10.5 shows the XML representation of `partNumberListType`, which contains a list of `partNumberType`. Listing 10.2 portrays the `partNumberType` referenced by the `itemType` attribute of `partNumberListType`.

Listing 10.5 Specifying a List of Simple Types (`catalog.xsd`)

```
<xsd:simpleType name="partNumberListType"
                id="catalog.partNumber.list.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The "partNumberListType" describes the value
            for an element that contains a set of part
            numbers.  Given that a part number might look
            like any of the following:
```

```
                J1
                ABC32897
                ZZ22233344
          A list of these part numbers might look like:
                J1 ABC32897 ZZ22233344
        </xsd:documentation>
    </xsd:annotation>
    <xsd:list id="transaction.partNumberList"
              itemType="partNumberType">
    </xsd:list>
</xsd:simpleType>
```

An element type whose structure type is `partNumberListType` might look like the following element:

```
<xsd:element name="partNumberList" type="partNumberListType"/>
```

Given the preceding element type, the following element specifying the requisite spacedelimited list is valid in an XML instance:

```
<partNumberList>J1 ABC32897 ZZ22233344</partNumberList>
```

10.7.1 Attributes of a `list` Element

The `list` element has two optional attributes. The first is the ubiquitous `id` attribute. The other is the `itemType` attribute that specifies the valid simple type of values that are acceptable in the list. Table 10.5 provides an overview of the attributes appropriate for a `list` element.

Tip

A list of values is always space delimited. Therefore, the value specified by the `itemType` attribute or `simpleType` element must not specify values that contain a space.

Table 10.5. Attribute Summary for a `list` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`itemType`	The value of an `itemType` attribute is a global simple type. A value permitted by the simple type specified by the `itemType` is a valid item value in the list.

The space character (' ') delimits a list of values: A value in a list may not contain embedded spaces besides the delimiter. A list could contain the colors '`cyan`', '`magenta`', '`yellow`', and '`black`'. The resultant list might look like the following instance:

```
<colors>yellow cyan</colors>
```

The list could *not* contain '`Ocean`', '`Pink Grapefruit`', '`Sunshine`', and '`Midnight`'. The theoretical resultant list might look like the following invalid instance, which, examined carefully, appears to have *five* values:

```
<colors>Ocean Pink Grapefruit Sunshine Midnight<colors>
```

The Schema recommendation does not require or allow that the XML parsers distinguish that the previous example has four, not five, values in the list specified by the `colors` element.

10.7.1.1 The `id` Attribute of a `list` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`list: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` attribute must be unique within an XML schema.
Required:	No.

Listing 10.4 demonstrates the use of the `id` attribute in conjunction with a `list` element.

10.7.1.2 The `itemType` Attribute of a `list` Element

The value of the `itemType` attribute references a global simple type. The simple type referenced by the `itemType` specifies the value of individual items in the list.

Attribute Overview

`list: itemType`

Value:	A QName.
Default:	None.
Constraints:	The value must be the name of a global simple type.
Required:	No. Either the `itemType` attribute or a local `simpleType` element is required; they are mutually exclusive.

i know..

Listing 10.4 demonstrates the use of the `itemType` attribute in conjunction with a `list` element.

10.7.2 Content Options for a `list` Element

The `list` element has two optional content options: `annotation` and `simpleType`. The `annotation` element documents the `list`. The `simpleType` element specifies the valid individual values that may be contained in an XML instance of the list. The syntax for the `simpleType` element is recursive in the sense that virtually anything in this chapter might appear within the local `simpleType`. Table 10.6 provides an overview of the content options for the `list` element.

Table 10.6. Content Options for a `list` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`simpleType`	The valid contents of the list are determined either by this local `simpleType`, or by the simple type referenced in the value of an `itemType` attribute.

The content pattern for a `list` element is:

```
annotation? simpleType?
```

The XML representation of a list might contain local simple types. Listing 10.6 is a rewrite of the `partNumberListType` presented in Listing 10.5. The only difference is that `partNumberType` is anonymous in Listing 10.6.

Listing 10.6 A Simple Type with an Anonymous List Item Simple Type (`listExamples.xsd`)

```
<xsd:simpleType name="partNumberListType"
                final="list">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A transaction consists of an order (or return)
            of a list of parts.  Each part number consists
            of 1 to 3 alpha characters followed by 1 to 8
            numeric characters.  The following part numbers,
            for example, are valid:
                J1
                ABC32897
                ZZ22233344
           Note that the "list" of these part numbers
           might look like:
                J1 ABC32897 ZZ22233344
        </xsd:documentation>
    </xsd:annotation>
    <xsd:list id="transaction.partNumberList">
    <xsd:simpleType>
            <xsd:annotation>
                <xsd:documentation xml:lang="en">
                    Anonymous declaration of a part number.
                    Each part number consists of 1 to 3 alpha
                    characters followed by 1 to 8 numerics.
                    The following part numbers, for
                    example, are valid:
                        J1
                        ABC32897
```

sub type of this simple type cannot have list

```
                    ZZ22233344
                </xsd:documentation>
            </xsd:annotation>
            <xsd:restriction base="xsd:token">
                <xsd:pattern value="[A-Z]{1,3}\d{1,8}"/>
            </xsd:restriction>
        </xsd:simpleType>
    </xsd:list>
</xsd:simpleType>
```

Note that the XML representation of the element type is the same as when the simple type describing the list is global:

```
<xsd:element name="partNumberList" type="partNumberListType"/>
```

Not surprisingly, the same XML instance works too:

```
<partNumberList>J1 ABC32897 ZZ22233344</partNumberList>
```

10.7.3 Constraining Facets of a `list` Element

The values in a list are constrained by either the `itemType` attribute or a nested `simpleType`. The list, however, may be further constrained by constraining facets that apply to the list—not the list items. Like most simple types, only certain constraining facets apply to lists. Table 10.7 provides a summary of the constraining facets applicable to a `list`.

Warning

Part I of the W3C XML Schema specification incorrectly states, "Only length, minLength, maxLength, pattern and enumeration facet components are allowed among the {facets}." Part II, however, correctly specifies that the `whiteSpace` attribute is applicable. Part II also specifies that `whiteSpace` is fixed to '`collapse`' making the difference largely academic. However, because of the discrepancy between these two documents, XML parsers may or may not accept the `whiteSpace` attribute.

Table 10.7. Constraining Facets of a `list`

Constraining Facet	*Description*
`length`	The v… …the number of elements that r…
`maxLength`	The v… …ifies the maximum number of el…
`minLength`	The v… …ifies the minimum number of el…
`enumeration`	The v… …pecifies the actual values of the e… …nstance. These values are addit… …t's type.
`pattern`	The v… …s a regular expression of value… …ce. This pattern specifies the v… …o the constraining facets speci…
`whiteSpace`	The value of the `whiteSpace` constraining facet specifies the validity of spaces, carriage returns, and line feeds in a string. The `whiteSpace` constraining facet is redundant for a list because the value of the `whiteSpace` constraining facet must be COLLAPSE.

all the facts are character related.

Listing 10.7 demonstrates a `list` whose `length` is constrained to contain only one item.

Listing 10.7 A Constrained `list` (`listExamples.xsd`)

```
<xsd:simpleType name="oneItemPartNumberListType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            In some cases the list of part numbers
            must be restricted to just one item.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="partNumberListType">
        <xsd:length value="1"/>
    </xsd:restriction>
</xsd:simpleType>
```

10.8 The `union` Element

A simple type can specify a union of other simple types. The contents in the XML instance of an element constructed from a union can be any of the valid types. A union can be thought of as an "or" construct. That is, "a union b union c" is the conceptual representation of "a or b or c."

Tip

A `union` may contain member types that have completely distinct base types, as demonstrated in Listing 10.8.

Listing 10.8 demonstrates the XML representation of the `colorOptionType`, which is a `union` of the anonymously declared `standardColorOptionType`, `fancifulColorOptionType`, and `codedColorOptionType` enumerations.

Listing 10.8 A `union` Specifying Local `simpleType` Elements (`catalog.xsd`)

```
<xsd:simpleType name="colorOptionType"
                id="catalog.colorOption.union.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A part has one of the following color definitions:
                - a standard name (cyan, yellow, etc.),
                - a fanciful name (Ocean, Sunshine, etc.), or
                - an internal code 1..n
        </xsd:documentation>
    </xsd:annotation>
    <xsd:union id="colorOptionType.union">

<xsd:simpleType name="standardColorOptionType"
                final="restriction"
                id                              rOption.sType">
            <xsd:annotation>
                <xsd:documentation xml:lang="en">
                    Color selection is limited.
                    The colors apply to unit and
                    bulk items.
                </xsd:documentation>
            </xsd:annotation>
            <xsd:restriction base="xsd:token">
                <xsd:enumeration value="cyan"/>
```

i am not able to name these UNIONED simple type.

as per latest impl it can have only 'id' attribute. simple type is anonymous

this simple type cannot have 'name' attribute

```
                <xsd:enumeration value="magenta"/>
                <xsd:enumeration value="yellow"/>
                <xsd:enumeration value="black"/>
            </xsd:restriction>
        </xsd:simpleType>
```

this simple type cannot have 'name' attribute

```
<xsd:simpleType name="fancifulColorOptionType"
                final="restriction"
                id="catalog.fancifulColorOption.sType">
            <xsd:annotation>
                <xsd:documentation xml:lang="en">
                    Color selection is limited.
                    The colors apply to unit and
                    bulk items.
                </xsd:documentation>
            </xsd:annotation>
            <xsd:restriction base="xsd:token">
                <xsd:enumeration value="Ocean"/>
                <xsd:enumeration value="Pink Grapefruit"/>
                <xsd:enumeration value="Sunshine"/>
                <xsd:enumeration value="Midnight"/>
            </xsd:restriction>
        </xsd:simpleType>
```

this simple type cannot have 'name' attribute

```
<xsd:simpleType name="codedColorOptionType"
                id="catalog.codedColorOption.sType">
            <xsd:annotation>
                <xsd:documentation xml:lang="en">
                    A color can be defined by an
                    internal integer that maps
                    directly to a standard or
                    fanciful color
                    1 = cyan = Ocean
                    2 = magenta = Pink Grapefruit
                    etc.
                </xsd:documentation>
            </xsd:annotation>
            <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxInclusive value="4"/>
            </xsd:restriction>
```

```
        </xsd:simpleType>

</xsd:union>
</xsd:simpleType>
```

An element type whose structure type is `colorOptionType` might look like the following element:

```
<xsd:element name="color" type="colorOptionType"/>
```

Given the preceding element type, the following `color` element containing the standard color 'magenta' is valid in an XML instance:

```
<color>magenta</color>
```

A fanciful name is also valid as a color:

```
<color>Pink Grapefruit</color>
```

Finally, a coded integer can determine the color:

```
<color>2</color>
```

10.8.1 Attributes of a `union` Element

The `union` element has two optional attributes. The first is the ubiquitous `id` attribute. The other is the `memberTypes` attribute that specifies the set of simple type from whose conceptual values form a union. Table 10.8 provides an overview of the attributes associated with a `union` element.

Table 10.8. Attribute Summary for a `union` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`memberTypes`	The value of a `memberTypes` attribute is a list of global simple type names. Each member type specifies an acceptable set of values in the union.

10.8.1.1 The id Attribute of a union Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

union: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 10.8 demonstrates the use of the id attribute in conjunction with a union element.

10.8.1.2 The memberTypes Attribute of a union Element

The memberTypes attribute specifies a set of simple types. The union is the set of all values represented by these types. The union element can specify simpleType elements locally in addition to, or instead of, the simple types associated with the memberTypes attribute.

Attribute Overview

union: memberTypes

Value:	A list of QNames.
Default:	None.
Constraints:	Each QName in the value-list must refer to a global simple type.
Required:	No. The simple type can specify the memberTypes attribute, local simple types, or both.

this is the only one exception both(nested and attribute) allowed

The primary example for this section—Listing 10.8—shows the colorOptionType with anonymous member types. Listing 10.9 is a rewrite of Listing 10.8, demonstrating references to global member types; the memberTypes attribute references the global simple types that enumerate the color selections.

Listing 10.9 A union Specifying the memberTypes Attribute (unionExamples.xsd)

```
<xsd:simpleType name="standardColorOptionType"
                final="restriction">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Color selection is limited.
            The colors apply to unit and
            bulk items.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="cyan"/>
        <xsd:enumeration value="magenta"/>
        <xsd:enumeration value="yellow"/>
        <xsd:enumeration value="black"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="fancifulColorOptionType"
                final="restriction">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Color selection is limited.
            The colors apply to unit and
            bulk items.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="Ocean"/>
        <xsd:enumeration value="Pink Grapefruit"/>
        <xsd:enumeration value="Sunshine"/>
        <xsd:enumeration value="Midnight"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="codedColorOptionType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
```

```
            A color can be defined by an
            internal integer that maps
            directly to a standard or
            fanciful color
            1 = cyan = Ocean
            2 = magenta = Pink Grapefruit
            etc.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:positiveInteger">
        <xsd:maxInclusive value="4"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="colorOptionType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A part has one of the following color definitions:
            - a standard name (cyan, yellow, etc.),
            - a fanciful name (Ocean, Sunshine, etc.), or
            - an internal code 1..n
        </xsd:documentation>
    </xsd:annotation>
    <xsd:union id="colorOptionType.union"
               memberTypes="standardColorOptionType
                            fancifulColorOptionType
                            codedColorOptionType"/>
</xsd:simpleType>
```

10.8.2 Content Options for a `union` Element

The `union` element has only two options for associated content: an optional annotation and an optional simple type. The syntax for the simple type is recursive in the sense that virtually anything in this chapter might appear within the local simple type. Table 10.9 provides a quick summary of the available content options for `union`.

Table 10.9. Content Options for a `union` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`simpleType`	The values specified by any local `simpleType` elements are acceptable values in the union. These values are in addition to those specified by the `memberTypes` attribute.

so, the 'memberType' and content 'simpleType' are NOT MUTUALLY EXCLUSIVE

The content pattern for a `union` element is:

```
annotation? simpleType*
```

A union might contain multiple member types. Any or all of these member types can be anonymous. Listing 10.8 demonstrates anonymous member types.

10.8.3 Constraining Facets of a `union` Element

Only the `enumeration` and `pattern` constraining facets listed in Table 10.10 are applicable to the `union` element. Chapter 12 discusses constraining facets in detail.

Table 10.10. Constraining Facets of a `union` Element

Constraining	*Facet Description*
`enumeration`	The value of the `enumeration` constraining facet is one of perhaps many values that are valid in an XML instance that corresponds to the `union` element. Any enumerated value is a further restriction of the constraining facets associated with the member types that comprise the union.
`pattern`	The value of the `pattern` constraining facet is a regular expression representing the valid values in an XML instance that corresponds to the `union` element. Any pattern is a further restriction of the constraining facets associated with the member types that make up the union.

Listing 10.10 demonstrates a rather nonsensical restriction of the `colorOptionType` declared in Listing 10.8. The result of adding the pattern `'.*c.*'` to the new `cColorUnionRestrictionType` is that only values in the union that contain a 'c' are valid.

Therefore, only 'Ocean' and 'cyan' are valid values of cColorUnionRestrictionType.

Listing 10.10 Adding Constraining Facets of a union Element (unionExamples.xsd)

```
<xsd:simpleType name="cColorUnionRestrictionType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A restriction of all valid colors.
            In particular, the pattern only
            allows colors with the character 'c.'
            This limits the valid values to
            "cyan" and "Ocean."
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="colorOptionType">
        <xsd:pattern value=".*c.*"/>
    </xsd:restriction>
</xsd:simpleType>
```

 CONTENTS 

CONTENTS

Chapter 11. Complex Types

65 pages

IN THIS CHAPTER

Complex types provide sophisticated groupings of element types. Complex types specify the following functionality:

1. Adding attribute types to simple types.
2. Requiring *empty content*: An XML instance has no text or embedded elements. A complex type that specifies empty content may have attributes.
3. Nesting element types.
4. Permitting *mixed content*: An XML instance may contain text interspersed with elements.

A `complexType` element is the XML representation of a complex type in an XML schema document. Unlike an element type or an attribute type, but like a simple type, a complex type cannot be directly instantiated in an XML instance. The complex type must be associated with an element type (that is, the complex type must be the structure type of an element type).

11.1 An Example of a Complex Type Specifying Empty Content

A complex type can specify empty content. In other words, an XML instance that contains an element that is an instance of an element type whose structure type is this complex type does not contain any text or any other elements. The `freePriceType` complex type presented in Listing 11.1 is used to "price" items in the

catalog that have no cost associated with them. Any discount catalog item has an `authorization` attribute. The complex type `freePriceType` has no content (that is, no base simple type or nested element types):

Listing 11.1 A Complex Type Specifying Empty Content (`pricing.xsd`)

```
<xsd:complexType name="freePriceType"
                 block="#all"
                 final="#all"
                 id="freePriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything that is free has no
            value (i.e., price), but must
            have an authorization code.
            This is a complex type with
            "empty" content.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:attributeGroup ref="saleAttributeGroup"/>
</xsd:complexType>
<xsd:attributeGroup name="saleAttributeGroup">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything that is on sale (or free,
            which is a type of sale), must
            have an authorization defined.
            This is someone's name,
            initials, ID, etc.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:attribute name="authorization" type="xsd:token"/>
</xsd:attributeGroup>
```

An element type whose structure type is `freePriceType` might look like the following element:

```
<xsd:element name="freePrice" type="freePriceType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<freePrice authorization="CB"/>
```

11.2 An Example of a Complex Type That Adds Attributes to a Simple Type

A complex type can add attribute types to a simple type. Listing 11.2 is the complex type `salePriceType` `salePriceType` extends the globally described `dollarPriceType` simple type by adding the `saleAttributeGroup` attribute group. Note that the `simpleContent` element provides the functionality to extend simple types into complex types.

Listing 11.2 Adding an Attribute to a Simple Type (`pricing.xsd`)

```
<xsd:complexType name="salePriceType"
                 block="#all"
                 final="extension"
                 id="salePriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything on sale must have a price
            and an authorization.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="dollarPriceType">
            <xsd:attributeGroup ref="saleAttributeGroup"/>
    </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>

<xsd:attribute name="currency"
               type="xsd:token"
               fixed="U S Dollars">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            U S Dollars are the only currency
            currently allowed.  This attribute
            is a great example of using "ref"
            (elsewhere), but is not set up well
            for extending to other currencies
            later.  This should really be a
            type that keeps getting restricted.
        </xsd:documentation>
    </xsd:annotation>
```

```
</xsd:attribute>

<xsd:simpleType name="currencyAmountType"
                id="pricing.currencyAmount.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Limit all transactions to less than
            500,000.00 of any currency
            This can be represented as NNNNNN.NN
            or eight total digits, two of which are
            after the decimal point.

            ********************************************
               Note that the W3C XML Schema
               Recommendation does not support
               non-instantiable simple types.

               This simple type is conceptually
               not instantiable.  This type is not
               intended to be used directly, only
               indirectly, such as via the ...DollarType
               simple types.
            ********************************************
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:decimal">
        <xsd:totalDigits value="8" fixed="true"/>
        <xsd:fractionDigits value="2" fixed="true"/>
        <xsd:minExclusive value="0.00" fixed="true"/>
        <xsd:maxInclusive value="500000.00" fixed="true"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="restrictedDollarAmountType"
                id="pricing.restrictedDollarAmount.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Nothing sells for less than $1 or
            greater than or equal to $10,000.00.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="currencyAmountType">
        <xsd:minInclusive value="1.00"
```

```
                            fixed="true"/>
        <xsd:maxExclusive value="10000.00"
                            fixed="true"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:complexType name="dollarPriceType"
                 final="restriction"
                 block="restriction"
                 abstract="true"
                 id="dollarPriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Currently, currency is limited to
            U S Dollars. Note that this type is
            non-instantiable.  A derived type must
            defined that sets the range.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="restrictedDollarAmountType">
            <xsd:attribute ref="currency"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:simpleType>
```

this is business req.

printing mistake

The XML representation of an element type whose structure type is `salePriceType` might look like the following element:

```
<xsd:element name="salePrice" type="salePriceType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<salePrice>123.45</salePrice>
```

11.3 An Example of a Complex Type Specifying Nested Element Types

A complex type can specify multiple element types. An XML instance might contain an element whose element type's structure type is this complex type; the XML instance contains nested elements. Listing 11.3 demonstrates how to create an XML representation of a complex type that contains nested element types. The thematic catalog example has a `partOptionType` complex type. This `partOptionType` permits a

catalog entry to have a nested `color`, `size`, or both.

Listing 11.3 A Complex Type Specifying Nested Element Types (`catalog.xsd`)

```
<xsd:complexType name="partOptionType"
                 block="#all"
                 final="#all"
                 id="partOptionType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Appropriate parts can have a color,
            a size, or both.  Note that the use
            of the "all" element indicates that
            the "color" and "size" are unordered.
            That is, they can appear in either
            order.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:all id="pot.all">
        <xsd:element name="color"
                     type="colorOptionType"
                     minOccurs="0"
                     maxOccurs="1"/>
        <xsd:element name="size"
                     type="sizeOptionType"
                     minOccurs="0"
                     maxOccurs="1"/>
    </xsd:all>
</xsd:complexType>

<xsd:simpleType name="colorOptionType"
                id="catalog.colorOption.union.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A part has one of the following color definitions:
                - a standard name (cyan, yellow, etc.),
                - a fanciful name (Ocean, Sunshine, etc.), or
                - an internal code 1..n
        </xsd:documentation>
    </xsd:annotation>
    <xsd:union id="colorOptionType.union">
```

maxOccurs cannot have more than 1 since 'all' group used

```
<xsd:simpleType name="standardColorOptionType"
                final="restriction"
                id="catalog.standardColorOption.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Color selection is limited.
            The colors apply to unit and
            bulk items.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="cyan"/>
        <xsd:enumeration value="magenta"/>
        <xsd:enumeration value="yellow"/>
        <xsd:enumeration value="black"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="fancifulColorOptionType"
                final="restriction"
                id="catalog.fancifulColorOption.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Color selection is limited.
            The colors apply to unit and
            bulk items.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="Ocean"/>
        <xsd:enumeration value="Pink Grapefruit"/>
        <xsd:enumeration value="Sunshine"/>
        <xsd:enumeration value="Midnight"/>
    </xsd:restriction>
</xsd:simpleType>

<xsd:simpleType name="codedColorOptionType"
                id="catalog.codedColorOption.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A color can be defined by an
            internal integer that maps
            directly to a standard or
```

```
                    fanciful color
                    1 = cyan = Ocean
                    2 = magenta = Pink Grapefruit
                    etc.
                </xsd:documentation>
            </xsd:annotation>
            <xsd:restriction base="xsd:positiveInteger">
                <xsd:maxInclusive value="4"/>
            </xsd:restriction>
        </xsd:simpleType>

    </xsd:union>
</xsd:simpleType>

<xsd:simpleType name="sizeOptionType"
                final="#all"
                id="catalog.sizeOption.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Size selection is limited.
            The sizes apply to unit and
            bulk items.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:token">
        <xsd:enumeration value="tiny"/>
        <xsd:enumeration value="small"/>
        <xsd:enumeration value="medium"/>
        <xsd:enumeration value="large"/>
        <xsd:enumeration value="grandiose"/>
    </xsd:restriction>
</xsd:simpleType>
```

The XML representation of an element type whose structure type is `partOptionType` might look like the following element:

```
<xsd:element name="partOption" type="partOptionType"/>
```

Given the preceding element type, the following `partOption` element, with nested `size` and `color` elements, is valid in an XML instance:

```
<partOption>
    <size>grandiose</size>
```

```
    <color>cyan</color>
</partOption>
```

For clarity, note that an XML instance of `partOption` can contain just one of the `size` or `color` elements because of the `minOccurs` attribute of the respective element type:

```
<partOption>
    <color>cyan</color>
</partOption>
```

11.4 An Example of a Complex Type Specifying Mixed Content

A complex type can specify mixed content. Mixed content means that an element whose element type's structure type is this complex type can contain text interspersed with elements. The text is always free form (there is no way to specify constraints on the text). The elements are typically partially confined. Listing 11.4 is `catalogEntryDescriptionType`, which permits valid lists of part numbers to be interspersed with text.

Listing 11.4 A Complex Type Specifying Mixed Content (`catalog.xsd`)

```
<xsd:complexType name="catalogEntryDescriptionType"
                 mixed="true"
                 id="catalogEntryDescriptionType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Allow the description of a part
            to include part number references.
            The "catalogEntryDescriptionType"
            is a good example of a complex type
            with "mixed" content.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence minOccurs="0" maxOccurs="unbounded">
        <xsd:element name="partList" type="partNumberListType"/>
    </xsd:sequence>
</xsd:complexType>
```

mostly, in real life programming we NEVER need this feature of complex type

The XML representation of an element type whose structure type is `catalogEntryDescription` might look like the following element:

```
<xsd:element name="description"
```

```
                type="catalogEntryDescriptionType"/>
```

Given the preceding element type, the following `description` element is valid in an XML instance:

```
<description>
    This is made up of both
    <partList>UX002 UX003</partList>
    pieces which makes this assembly
    better than
    <partList>ASM2000</partList>
    and better than the competition.
</description>
```

11.5 Concepts and Observations

A complex type has many options. Accordingly, the XML representation of a complex type has many possible attributes and many choices for content. Because of the capabilities of a complex type, this section highlights quite a few observations about the Schema Recommendation.

11.5.1 Explicitly Non-instantiable Complex Types

Only complex types and element types can be explicitly non-instantiable. A complex type that specifies the `abstract` attribute with the value '`false`' cannot have instances. Listing 11.5 is the explicitly non-instantiable `dollarPriceType` complex type.

Warning

Read the discussion about terminology in Section 1.8 before reading about abstraction and instantiation. The Schema Recommendation uses the term *abstract* to mean both a conceptual data model and a non-instantiable complex or element type.

Listing 11.5 An Explicitly Non-instantiable Complex Type (`pricing.xsd`)

```
<xsd:complexType name="dollarPriceType"
                 final="restriction"
                 block="restriction"
                 abstract="true"
                 id="dollarPriceType.pricing.cType">
    <xsd:annotation>
<xsd:documentation xml:lang="en">
            Currently, currency is limited to
```

```
            U.S. Dollars. Note that this type is
            non-instantiable.  A derived type must be
            defined that sets the range.
</xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="restrictedDollarAmountType">
            <xsd:attribute ref="currency"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

If the complex type is explicitly non-instantiable, a derivation might be instantiable. Listing 11.6 has several instantiable complex types, which are derivations of the non-instantiable `dollarPriceType`.

Listing 11.6 An Instantiable Complex Type Derived from a Non-instantiable Complex Type (`pricing.xsd`)

```
<xsd:complexType name="fullPriceType"
                 block="#all"
                 final="#all"
                 id="fullPriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The pricing element for all items
            sold at full price have no elements
            or attributes.  The price is simply
            the amount, stored in the value
            of the element.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="dollarPriceType"/>
    </xsd:simpleContent>
</xsd:complexType>

<xsd:complexType name="salePriceType"
                 block="#all"
                 final="extension"
                 id="salePriceType.pricing.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything on sale must have a price
```

```
            and an authorization
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="dollarPriceType">
            <xsd:attributeGroup ref="saleAttributeGroup"/>
        </xsd:extension>
    </xsd:simpleContent>
</xsd:complexType>
```

An element type whose structure type is `fullPriceType` might look like the following element:

```
<xsd:element name="fullPrice" type="fullPriceType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<fullPrice>32.00</fullPrice>
```

An instance can specify an instantiable derivation by using the `xsi:type` attribute. An element type whose structure type is `dollarPriceType` might look like the following element:

```
<xsd:element name="amount" type="dollarPriceType"/>
```

Given the preceding element type, the following element is valid in an XML instance:

```
<amount xsi:type="salePriceType">123.45</amount>
```

Note that `salePriceType`, described in Listing 11.6, is a valid instantiable derivation of `dollarPriceType`.

11.5.2 Implicitly Non-instantiable Complex Types

A complex type can be implicitly non-instantiable because of a nested non-instantiable element type. Transitively, the element type can be non-instantiable either because the element type is explicitly non-instantiable or because the structure type associated with the element type is non-instantiable.

The `baseCatalogEntryType` in Listing 11.7 is explicitly non-instantiable (it has an `abstract` attribute whose value is '`true`'). The `baseCatalogEntryType` is also implicitly non-instantiable: It contains a reference to the global `sequenceID` element type, which is non-instantiable.

Tip

In theory, describing `baseCatalogEntryType` as non-instantiable is redundant. This explicit attribute may be—in practice—required. The Recommendation mandates that a complex type must be explicitly defined as non-instantiable when the intent is that the XML instance—not the schema—specifies the structure type (see various discussions on the use of `xsi:type` in this book). This tip applies only to complex types; a simple type is always instantiable.

but, i have not got any warning or info about this reduntancy

Listing 11.7 An Implicitly Non-instantiable Complex Type (`catalog.xsd`)

```
<xsd:complexType name="baseCatalogEntryType"
                 abstract="true"
                 id="baseCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A catalog entry must have:
                * A database ID
                * Part Name
                * Part Number
                * Options available
                * Description
                * Price
                * Included Quantity when ordering
                  one item.
            The "baseCatalogEntryType" is
            non-instantiable:  a derived type must
            be created before a catalog
            entry can be instantiated.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence id="bacet-seq">
        <xsd:element ref="sequenceID"/>
        <xsd:element name="partName" type="partNameType"/>
        <xsd:element name="partNumber" type="partNumberType"/>
        <xsd:element name="partOption" type="partOptionType"/>
        <xsd:element name="description"
                     type="catalogEntryDescriptionType"/>
        <xsd:group ref="priceGroup"/>
        <xsd:element name="includedQuantity"
                     type="xsd:positiveInteger"/>
    </xsd:sequence>
    <xsd:attribute name="category"
                   type="categoryType"
```

```
                    use="required"/>
</xsd:complexType>
```

The `baseCatalogEntryType` in Listing 11.7 requires the database sequence types from Listing 11.8.

Listing 11.8 Database Sequence Types (`sequence.xsd`)

```
<xsd:simpleType name="sequenceIDType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A Sequence ID is generated by
            the database.  Sequences are
            integers that start with "0"
        </xsd:documentation>
    </xsd:annotation>
    <xsd:restriction base="xsd:nonNegativeInteger"/>
</xsd:simpleType>

<xsd:element name="sequenceID"
             type="sequenceIDType"
             abstract="true"
             block="substitution">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element is
            non-instantiable:  the element
            must be replaced by a substitution
            group in either a derived type or
            an instance.
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="unitID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for unit items.
            This element provides a valid
            substitution for "sequenceID".
```

```
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="bulkID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for bulk items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>

<xsd:element name="assemblyID"
             type="sequenceIDType"
             substitutionGroup="sequenceID">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This element represents sequence
            IDs for assembled items.
            This element provides a valid
            substitution for "sequenceID".
        </xsd:documentation>
    </xsd:annotation>
</xsd:element>
```

Listing 11.9 is the `unitCatalogEntryType`, which is a derivation of the `baseCatalogEntryType` described in Listing 11.7. The `restriction` element specifies the base complex type. In addition, the instantiable `unitID` replaces the non-instantiable `sequenceID`.

Listing 11.9 An Instantiable Derivation of an Implicitly Non-instantiable Complex Type (`catalog.xsd`)

```
<xsd:complexType name="unitCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="unitCatalogEntryType.catalog.cType">
    <xsd:annotation>
```

```
        <xsd:documentation xml:lang="en">
            A unit item contains nothing more
            or less than a basic catalog entry ID:
                * A database ID
                * Part Name
                * Part Number
                * Options available
                * Price
                * Included Quantity when ordering
                  one item (always one for unit items).
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent id="ucet.cc">
        <xsd:restriction base="baseCatalogEntryType">
            <xsd:sequence>
                <xsd:element ref="unitID"/>
                <xsd:element name="partName" type="partNameType"/>
                <xsd:element name="partNumber"
                             type="unitPartNumberType"/>
                <xsd:element name="partOption" type="partOptionType"/
>
                <xsd:element name="description"
                             type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             fixed="1"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                           type="categoryType"
                           fixed="unit"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Tip

Some complex types are implicitly non-instantiable because they contain a reference to a global non-instantiable element type; an element in the XML instance can determine the instantiable element type by specifying an appropriate substitution group as its typename, or by specifying an instantiable structure type via `xsi:type`.

11.5.3 Adding Element Types or Attribute Types to a Derived Complex Type

A complex type can specify attribute types for simple or complex types. A `complexType` element, along with a nested `simpleContent` element, provides the functionality necessary to add attribute types to simple types. Similarly, a `complexType` element, along with a nested `complexContent` element, provides the functionality necessary to add element types—or attribute types—to complex types.

Listing 11.2 portrays a complex type that specifies a `simpleContent` element. Listing 11.9 portrays an XML representation of a complex type derivation that specifies a `complexContent` element.

11.5.4 Removing Element Types or Attribute Types from a Derived Complex Type

A derived complex type can remove element types or attribute types from a base complex type. A derived complex type removes a nested element type by setting the `maxOccurs` attribute of the appropriate element type to have a value of '`0`'. A derived complex type removes an attribute type by setting the `use` attribute of the appropriate attribute type to '`prohibited`'. Listing 8.13 portrays removing an element type from a complex type. Listing 9.7 portrays removing an attribute type from a complex type.

11.5.5 Prohibiting Extension or Restriction of a Complex Type

The `final` attribute prohibits further derivations of a complex type. In other words, when a complex type specifies a `final` attribute, the schema cannot contain derived complex types. In fact, the `final` attribute has a number of values that permit the complex type to limit only certain derivations. See Section 11.6.1.3 for more details.

The effects of the `block` attribute are convoluted. The `block` attribute affects substitution groups. The `block` attribute can also affect specifying a derived complex type in an XML instance. Both complex types and element types can specify a `block` attribute: There are complicated interactions. In order to avoid reiterating a lengthy discussion, please refer to Section 8.2.3 (Chapter 8 covers element types in general). Section 11.6.1.2 covers the options for specifying a `block` attribute for a complex type.

11.5.6 Shorthand Notation of a Complex Type

A complex type always has associated content (even empty content classifies as content). The complex type specifies its content as either simple or complex. The corollary XML representation is a `simpleContent` or a `complexContent`.

The XML Schema syntax permits a shorthand notation where a `complexType` element does not explicitly contain either a `simpleContent` or a `complexContent` element. The shorthand notation specifies that

when the content type is not explicit, the processor infers complex content that restricts a base of `anyType`.

The XML representation of `longhandPartOptionType` in Listing 11.10 is functionally equivalent to the XML representation of `partOptionType` in Listing 11.3.

Listing 11.10 Longhand Notation of a Complex Type

```
<xsd:complexType name="longhandPartOptionType"
                 block="#all"
                 final="#all">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The "partOptionType" is defined
            using the shorthand notation for
            complex types.  This element
            demonstrates the equivalence
            using the normal notation.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="xsd:anyType">
            <xsd:all>
                <xsd:element name="color"
                             type="colorOptionType"
                             minOccurs="0"
                             maxOccurs="1"/>
                <xsd:element name="size"
                             type="sizeOptionType"
                             minOccurs="0"
                             maxOccurs="1"/>
            </xsd:all>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

Tip

The shorthand notation used in Listing 11.10 restricts `anyType`, which permits any combination of elements and attributes.

Note that the shorthand notation permits the following subelements of a `complexType`:

- all
- anyAttribute
- attribute
- attributeGroup
- choice
- group
- sequence

Sections 11.11–11.14 discuss the all, choice, sequence, and group elements, respectively. Sections 9.5–9.7 cover the attribute, attributeGroup, and anyAttribute elements, respectively.

11.6 The complexType Element

Complex types provide a wide range of functionality. This wide range includes, but is not limited to, extending simple types and restricting or extending complex types, empty content, and mixed content. Most of the listings in this chapter, especially Listings 11.1–11.4, contain examples of complex types.

11.6.1 Attributes of a complexType Element

A complexType element can specify the name and id attributes that are also applicable to many other elements. The attributes abstract, block, and final provide the capability to derive instantiable complex types. The mixed attribute determines when a complex type permits or prohibits mixed content. Table 11.1 provides a summary of all of the attributes applicable to a complexType.

Table 11.1. Attribute Summary for a complexType Element

Attribute	*Description*
abstract	The value of the abstract attribute is a Boolean value that determines if a complex type is explicitly non-instantiable. An XML instance may not contain data corresponding directly to a non-instantiable complex type.
block	The value of the block attribute determines the capability of an XML instance to specify an instantiable derivation of a non-instantiable complex type. The block attribute also limits the ability to substitute member element types of a substitution group—when the structure type of the head of the substitution group is this complex type.
final	The value of the final attribute determines the capability of an XML schema to specify an instantiable derivation of a non-instantiable complex type.
id	The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

`mixed`	The value of the `mixed` attribute controls whether an XML instance can inter-sperse text between elements in the contents of a complex type.
`name`	The value of the `name` attribute is the name of a complex type. A derived complex type references this name as the base complex type. An element type references this name to specify a structure type.

11.6.1.1 The `abstract` Attribute of a `complexType` Element

The value of the `abstract` attribute is a Boolean value that determines if a complex type is explicitly non-instantiable. An XML instance may not contain data corresponding directly to a non-instantiable complex type.

Listing 11.5 contains sample code that creates a non-instantiable complex type.

Attribute Overview

`complexType: abstract`

Value:	A Boolean value (that is, '`true`' or '`false`').
Default:	'`false`'.
Constraints:	None.
Required:	No.

11.6.1.2 The `block` Attribute of a `complexType` Element

Conveying how complex types block is extremely complicated. The `block` attribute determines the capability of an XML instance to substitute a derived complex type. The XML instance can substitute a derived type in one of two ways:

- The XML instance can request element validation against an instantiable derivation of a non-instantiable complex type by specifying `xsi:type` as an attribute to an element. The requested complex type must be a valid derivation of the element's element type's structure type.
- When an element type is a member of a substitution group (see Section 8.2.2), the XML instance can refer to an element type that is a valid substitution for the element type specified by the structure type. Note that a substitution group requires that the structure type of a substitutable element type is a derived structure type (or the same structure type) as the root (of the substitution group) element type.

The `block` attribute can disallow either or both of the substitutions just discussed. Furthermore, the `block` attribute can prohibit substitutions based on the kind of derivation. For example, a complex type could prohibit an XML instance from substituting elements whose element type's structure type is a restriction of itself, but allow the XML instance to substitute elements whose element type's structure type is an extension.

Tip

A solid understanding of extending and restricting complex types (that is, the use of the `extension` and `restriction` elements) is valuable before attempting to use the `block` or `final` attributes.

Attribute Overview

`complexType: block`

Value:	An enumeration that consists of '`#all`', or a space-delimited list of any or all of '`extension`' and '`restriction`'
Default:	In general, there is no default. However, the `schema` element can create a global default using the `blockDefault` attribute.
Constraints:	None.
Required:	No.

Listing 11.11 revisits the `dollarPriceType` described in Listing 11.5. This time `dollarPriceType` has the `block` attribute set to '`restriction`'. Note that for complex types '`block="#all"`' provides the same functionality as '`block="restriction extension"`'.

Listing 11.11 Blocking Restrictions or Extensions of a Complex Type

```
<xsd:complexType name="dollarPriceType"
                 final="restriction"
                 block="restriction"
                 abstract="true">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Currently, currency is limited to
            U S Dollars. Note that this type is
            non-instantiable.  A derived type must be defined
            that sets the range.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="restrictedDollarAmountType">
            <xsd:attribute ref="currency"/>
        </xsd:extension>
```

```
    </xsd:simpleContent>
</xsd:complexType>
```

The global `amount` element is now not so valuable; an XML instance cannot substitute an instantiable derivation of the `dollarPriceType` structure type:

```
<xsd:element name="amount" type="dollarPriceType"/>
```

Note that another complex type could reference the `amount` element type.

The previously defined `dollarPriceType` prohibits an instance from extending the type; therefore, the following is not valid:

```
<amount xsi:type="salePriceType">123.45</amount>
```

Limitations on extension work similarly when the `block` attribute includes '`extension`'. Finally, the value of the `block` attribute can be set to '`#all`', which blocks both restriction and extension in XML instances.

Warning

Many of the existing XML validators do not properly respect the `block` attribute of complex types.

Blocking is not inherited. Because of this limitation, derived types—in most cases—repeat the `block` attribute (e.g., a hierarchy of complex types derived by extension might all have a '`block="restriction"`' attribute). The same is true for the `final` attribute.

11.6.1.3 The `final` Attribute of a `complexType` Element

The `final` attribute prohibits extending or restricting a complex type. In other words, the `final` attribute prohibits derived complex types. When the value of the `final` attribute is '`extension`', there can be no complex types derived by extension. The `final` attribute can similarly prohibit restrictions.

Attribute Overview

`complexType: final`

Value:	An enumeration that consists of '`#all`', or a space-delimited list of any or all of '`extension`' and '`restriction`'

Default:	In general, there is no default. However, the `schema` element can create a global default using the `finalDefault` attribute.
Constraints:	None.
Required:	No.

Tip

A solid understanding of extending and restricting complex types (that is, the use of the `extension` and `restriction` elements) is valuable before attempting to use the `block` or `final` attributes.

Listing 11.11 assumes the original `dollarPriceType` described in 11.5; the `final` attribute has the value '`restriction`'. Because the value of the `final` attribute is '`restriction`', `fullPriceType` is a valid derivation by extension.

Note

The Schema Recommendation states that a complex type that does not allow restrictions or extensions is "final". The complex type `fullPriceType` described in Listing 11.12 is "final".

Listing 11.12 A Final Complex Type

```
<xsd:complexType name="fullPriceType"
                 block="#all"
                 final="#all">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The pricing element for all items
            sold at full price have no elements
            or attributes.  The price is simply
            the amount, stored in the value
            of the element.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:extension base="dollarPriceType"/>
    </xsd:simpleContent>
</xsd:complexType>
```

This `fullPriceType` described in Listing 11.12 does not permit any restrictions or extensions. For example, there cannot be a derivation that restricts the dollar amount.

Like the `block` attribute, the `final` attribute can be set to '`restriction`' to prevent derivations by restriction, '`extension`' to prevent derivations by extension, or '`#all`' to prevent any further derivations.

Warning

Many of the existing XML validators do not properly respect the `final` attribute of complex type definitions.

Warning

The following excerpt is from the Schema Recommendation: "Finality is not inherited, that is, a type definition derived by restriction from a type definition which is final for extension is not itself, in the absence of any explicit final attribute of its own, final for anything."

Finality is not inherited. Because of this limitation on finality, derived types—in most cases—repeat the `final` attribute (e.g., a hierarchy of complex types derived by extension might all have a '`final="restriction"`' attribute). The same is true for the `block` attribute.

11.6.1.4 The `id` Attribute of a `complexType` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`complexType: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Many of the listings in this chapter contain examples of complex types that specify an `id`

11.6.1.5 The `mixed` Attribute of a `complexType` Element

Having mixed content is useful when the XML is less structured. A good use of mixed content is embedding style in documents; highly structured XML schemas do not typically specify complex types that permit mixed content.

Attribute Overview

complexType: mixed

Value:	A Boolean value (that is, 'true' or 'false')
Default:	'false'
Constraints:	None.
Required:	No.

The mixed attribute applies to either the complexType element or the complexContent element (when the shorthand notation is not used). Listing 11.4 demonstrates the use of the mixed attribute associated with the catalogEntryDescriptionType element.

The mixed attribute of a complexContent element is redundant with the mixed attribute of a complexType element. Regardless, the attribute applies to the content (that is, complexContent). The attribute appears on the complexType element primarily to allow the shorthand notation (see Section 11.5.6). If a base complexContent element sets the mixed attribute to 'false', a complexType that permits mixed content is not a valid derivation.

11.6.1.6 The name Attribute of a complexType Element

The name identifies a complex type. Another complex type can reference this name to create a derived complex type. An element type can use this name to associate a structure type. An XML instance can reference this name to specify a derivation of a non-instantiable complex type.

Attribute Overview

complexType: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	Yes, unless describing a local, or anonymous, complexType within the context of another element or complex type in which case the name attribute is prohibited.

All of the examples in this chapter have a name.

11.6.2 Content Options for a `complexType` Element

Complex types provide a wide range of functionality. This wide range includes, but is not limited to, extending simple types and restricting or extending complex types, permitting empty content, and permitting mixed content. Because of this complexity, there is a correspondingly large range of content options. Table 11.2 summarizes these content options.

Table 11.2. Content Options for a `complexType` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`all`	The `all` element, discussed in Section 11.11, specifies a model group that indicates an unordered set of element types; an XML instance must reference all the nested element types in any order. A `complexType` element containing an immediate `all` subelement is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
`anyAttribute`	The `anyAttribute` element, discussed in Section 9.7, specifies a set of namespaces, each of which may provide global attribute types; the XML instance may reference any of these attribute types. A `complexType` element directly containing an `anyAttribute` element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
`attribute`	The `attribute` element, discussed in Section 9.5, specifies a local attribute type or refers to a global attribute type; the XML instance must reference this attribute, unless the attribute is optional or prohibited. A `complexType` element directly containing an `attribute` element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.

attributeGroup	The attributeGroup element, discussed in Section 9.6, specifies a reference to a named attribute use group. An XML instance must reference each attribute type according to the appropriate nested model group. A complexType element directly containing an attributeGroup element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
choice	The choice element, discussed in Section 11.12, specifies a model group that indicates a selection of one element type from a set of element types; an XML instance must reference only one of the element types. A complexType element directly containing a choice element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
complexContent	The complexContent element provides functionality for restricting or extending global complex types. The complexContent element also provides functionality for describing empty content, mixed content, and nested element types. See Section 11.8 for details and examples.
group	The group element, discussed in Section 11.14, specifies a reference to a named model group. The named model group specifies another model group (that is, all, choice, or sequence). A complexType element directly containing a group element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
sequence	The sequence element, covered in Section 11.13, specifies a model group that indicates an ordered set of element types; an XML instance must reference all the nested element types in the schema order. A complexType element directly containing a sequence element is a form of the shorthand notation. See Section 11.5.6 for a clarification of the shorthand notation.
simpleContent	The simpleContent element provides functionality for adding attribute types to a global simple type. See Section 11.7 for details and examples.

The content pattern for a complexType element is:

```
annotation?
(simpleContent |
 complexContent |
 ((group | all | choice | sequence)?
  (attribute | attributeGroup)* anyAttribute?))
```

11.7 The simpleContent Element

simpleTYPE can have only restriction

The simpleContent element can specify attributes for simple types. This element can also extend or restrict attribute types on other complex types with simple content. Listing 11.2 provides an example of adding attribute types to a simple type.

11.7.1 Attributes of a simpleContent Element

The only attribute of the simpleContent element is the ubiquitous id attribute (see Table 11.3).

Table 11.3. Attribute Summary of a simpleContent Element

Attribute	*Description*
id	The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

11.7.1.1 The id Attribute of a simpleContent Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

simpleContent: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 11.12 demonstrates the use of the id attribute of a simpleContent element in the XML representation of clearancePriceType.

11.7.2 Content Options for a simpleContent Element

The simpleContent can specify attribute types (via extensions) and it can restrict existing attribute types

(via restrictions). Table 11.4 provides an overview of the content options of a `simpleContent` element.

The content pattern for a `simpleContent` element is:

```
annotation? (restriction | extension)
```

The `extension` element, nested within a `simpleContent` element, provides functionality for adding attribute types to simple types or to complex types with simple content. Listing 11.2 portrays adding attribute types to a simple type.

Table 11.4. Content Options for a `simpleContent` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`extension`	The `extension` element provides functionality to add attribute types to a complex type. See Section 11.9, which covers the `extension` element, and Section 11.2, which provides a more thorough description and example.
`restriction`	The `restriction` element provides functionality to restrict the values of attributes already associated with a base complex type. See Section 11.10 for more detail.

The `extension` element also provides functionality for creating instantiable derivations of noninstantiable complex types with simple content. Listing 11.6 provides an example that demonstrates how to create the instantiable complex type `fullPriceType`. The complex type `fullPriceType` is a derivation of the non-instantiable complex type `dollarPriceType`.

Tip

The use of the `final` attribute on the base complex type may prohibit the extension of that complex type. In addition, the use of a `fixed` attribute on a constraining facet of the base complex type may prohibit further modification of that constraining facet.

Much like simple types, the `restriction` element, nested within a `simpleContent` element, provides functionality to add or modify constraining facets that apply to the simple type. The `salePriceType` component has simple content; the XML representation of `salePriceType` associates attribute types but not element types. The complex type `clearancePriceType` described in Listing 11.13 restricts the complex type `salePriceType` (described in Listing 11.2) to have a value of no more than $10.00.

Listing 11.13 Adding Restrictions to a Complex Type with Simple Content (`pricing.xsd`)

```
<xsd:complexType name="clearancePriceType"
                 block="#all"
                 final="#all">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Anything on sale must have a price
            and an authorization
        </xsd:documentation>
    </xsd:annotation>
    <xsd:simpleContent>
        <xsd:restriction base="salePriceType">
            <xsd:maxInclusive value="10.00"/>
        </xsd:restriction>
    </xsd:simpleContent>
</xsd:complexType>
```

restricting value of an attribute. Mostly not trying to remove a attribute

11.8 The `complexContent` Element

A `complexContent` element can specify nested element types. This includes the special case of zero elements, also known as "empty content." The `complexContent` element also provides functionality that permits text interspersed with elements, known as "mixed content." Listings 11.1 and 11.4 portray empty content and mixed content, respectively.

11.8.1 Attributes of a `complexContent` Element

Table 11.5 details the limited attributes available for use with a `complexContent` element.

Table 11.5. Attribute Summary of a `complexContent` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`mixed`	The value of the `mixed` attribute determines the capability of the complex type to contain mixed content. Mixed content is usually a rigid element structure (like the majority of complex types) that has free-form text interspersed between the elements.

11.8.1.1 The `id` Attribute of a `complexContent` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`complexContent : id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Listing 11.9 demonstrates a `complexContent` element with an associated `id` attribute.

this is usefull when shorthand notation not used

11.8.1.2 The `mixed` Attribute of a `complexContent` Element

The `mixed` attribute controls whether a complex type contains only elements (`mixed="false"`) or embedded text and elements (`mixed="true"`).

Attribute Overview

`complexContent : mixed`

Value:	A Boolean value (that is, '`true`' or '`false`')
Default:	'`false`'
Constraints:	None.
Required:	No.

Having mixed content is useful when the XML is less structured. A good use of mixed content is embedding style in documents; highly structured XML schemas do not typically define complex types that allow mixed content. The `mixed` attribute applies to either the `complexType` element or the `complexContent` element (if the shorthand notation is not used). Listing 11.4 demonstrates the use of the `mixed` attribute.

wow

Warning

The `mixed` attribute on the `complexContent` element is redundant with the mixed attribute

on the `complexType` element. If a base `complexContent` element set the `mixed` to '`false`', a derived `complexType` that permits mixed content is not valid.

11.8.2 Content Options for a `complexContent` Element

Complex types that contain complex content are either extensions or restrictions of other base complex types. Note that even the shorthand notation—which does not explicitly describe extension or restriction—is a restriction of `anyType`. Table 11.6 provides a brief description of the content options for a `complexContent` element.

The content pattern for the `complexContent` element is:

printing mistake

```
annotation? (attribute | attributeGroup)* anyAttribute?
```

Table 11.6. Content Options for a `complexContent` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`extension`	The `extension` element provides functionality for deriving complex types that add attribute types or elements to an existing base `complexType`.
`restriction`	The restriction element provides functionality for deriving complex types that restrict attributes or constraining facets of attributes or elements associated with an existing base `complexType`.

11.9 The `extension` Element

The `extension` element applies to complex types that contain simple content as well as complex types that contain complex content. The `extension` element provides functionality for adding attribute types to simple content. For complex content, the `extension` element provides functionality for adding either attributes or nested element types. Finally, with respect to simple content, the base type can be either a simple type or a complex type that contains simple content.

like mixed attribute with TRUE, and empty content

Tip

The use of the `final` attribute on the base complex type may prohibit the extension of that complex type. In addition, the use of a `fixed` attribute on a constraining facet of the base

complex type may prohibit further modification of that constraining facet.

In the following example, an *assembly* represents an orderable item made of many smaller parts. Listing 11.14 demonstrates deriving a complex type that specifies this assembly—`assemblyCatalogEntryType`—from the base `baseAssemblyCatalogEntryType` by adding a list of part numbers. The simple type `partNumberListType` represents the list of part numbers.

Listing 11.14 Extending a Complex Type That Contains Complex Content (`catalog.xsd`)

```
<xsd:complexType name="assemblyCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="assemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The actual definition of an assembly,
            including the contained parts.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:extension base="baseAssemblyCatalogEntryType"
                       id="acet.ext">
            <xsd:sequence>
                <xsd:element name="partList"
                             type="partNumberListType"/>
                <xsd:element name="status"
                             type="assemblyPartStatusType"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="baseAssemblyCatalogEntryType"
                 abstract="true"
                 block="#all"
                 id="baseAssemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An assembled item is similar to the
            other catalog entries.  The part number
            is restricted to an assembly number.
            In addition, there may be no options.
```

this affect only derivation in XML instance. not in schema itself

```
            Finally, a part list is also needed.
            Note that the "includedQuantity" has
            a default of one, but can be overridden
            in instances.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType"
                         id="bacet.rst">
            <xsd:sequence>
                <xsd:element ref="assemblyID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="assemblyPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"
                             minOccurs="0"
                             maxOccurs="0"/>
                <xsd:element name="description"
                             type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             default="1"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="partNumberListType"
                id="catalog.partNumber.list.sType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The "partNumberListType" describes the value
            for an element that contains a set of part
            numbers.  Given that a part number might look
            like any of the following:
                J1
```

```
                ABC32897
                ZZ22233344
          A list of these part numbers might look like:
                J1 ABC32897 ZZ22233344
        </xsd:documentation>
    </xsd:annotation>
    <xsd:list id="transaction.partNumberList"
              itemType="partNumberType">
    </xsd:list>
</xsd:simpleType>
```

Tip

Use an `extension` to create an instantiable derivation of an otherwise non-instantiable complex type. Simply do not add any attributes or elements. Listing 11.6 provides an example of using `extension` in this manner.

11.9.1 Attributes of an `extension` Element

The only attributes applicable to the `extension` element are the ubiquitous `base` attribute and the `id` attribute. Table 11.7 identifies these two attributes.

Table 11.7. Attribute Summary of an `extension` Element

Attribute	*Description*
`base`	The value of the `base` attribute identifies the base simple or complex type of the current derived complex type.
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

11.9.1.1 The `base` Attribute of an `extension` Element

The value of a `base` attribute indicates the global base type. For an extension of simple content, the base type can be either a simple type or a complex type with simple content. Otherwise, the derived complex type is an extension of complex content, which means that the value of the `base` attribute must refer to a global complex type.

when to extend a complex type use complexContent
when to extend a simple type use simpleContent

Attribute Overview

extension: base

Value:	A QName.
Default:	None.
Constraints:	The base type must refer to a global simple or complex type in the XML schema.
Required:	No.

11.9.1.2 The id Attribute of an extension Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

extension: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 11.13 portrays an extension element with an associated id attribute.

11.9.2 Content Options for an extension Element

Both the complexContent and the simpleContent elements may contain an extension element. Some of the content options described in this section apply to the complexContent element, some to the simpleContent element, and some to both. The description of each element in Table 11.8 identifies the appropriate applicability.

Note the Applies To column in Table 11.8. The Applies To column indicates whether the element applies to simple content (the cell value is 'S'), complex content (the cell value is 'C'), or both (the cell value is 'S/C').

Table 11.8. Content Options for an extension Element

Element	*Description*	*Applies To*
`all`	The `all` element, discussed in Section 11.11, specifies a model group that indicates an unordered set of element types; an XML instance must reference all the nested element types in any order.	C
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.	S/C
`anyAttribute`	The `anyAttribute` element, discussed in Section 9.7, specifies a set of namespaces, each of which may provide global attribute types; the XML instance may reference any of these attribute types.	S/C
`attribute`	The `attribute` element, discussed in Section 9.5, specifies a local attribute type or refers to a global attribute type; the XML instance must reference this attribute, unless the attribute is optional or prohibited.	S/C
`attributeGroup`	The `attributeGroup` element, discussed in Section 9.6, specifies a reference to a named attribute use group. An XML instance must reference each attribute type according to the appropriate nested model group.	S/C
`choice`	The `choice` element, discussed in Section 11.12, specifies a model group that indicates a selection of one element type from a set of element types; an XML instance must reference only one of the element types.	C
`group`	The `group` element, discussed in Section 11.14, references to a named model group. The named model group specifies another model group (that is, `all`, `choice`, or `sequence`)	C
`sequence`	The `sequence` element, covered in Section 11.13, specifies a model group that indicates an ordered set of element types; an XML instance must reference all the nested element types in the schema order.	C

The content pattern for an `extension` element nested within a `simpleContent` element is:

```
annotation? (attribute | attributeGroup)* anyAttribute?
```

The content pattern for the `extension` element nested within a `complexContent` element is:

```
annotation?
(group | all | choice | sequence)?
(attribute | attributeGroup)* anyAttribute?
```

11.10 The `restriction` Element

The `restriction` element provides functionality for deriving a complex type that restricts a base complex

type. For simple content, these restrictions may only specify constraining facets that apply to simple content. For complex content, the restriction permits altering attributes of nested element or attribute types.

> **Tip**
>
> The use of the `final` attribute on the base complex type may prohibit the restriction of that complex type. In addition, the use of a `fixed` attribute on a constraining facet of the base complex type may prohibit further modification of that constraining facet.

Listing 11.15 demonstrates how to use a `restriction` element for complex content. The `baseAssemblyCatalogEntryType` is a derivation of `baseCatalogEntryType`. In particular, note that the `assemblyPartNumberType` is a derivation of `partNumberType`. In addition, the `partOptionType` cannot appear in an XML instance, because the value of the `maxOccurs` attribute is '`0`'.

Listing 11.15 Restricting a Complex Type That Contains Complex Content (`catalog.xsd`)

```
<xsd:complexType
        name="baseAssemblyCatalogEntryType"
        abstract="true"
        block="#all"
        final="restriction"
        id="baseAssemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An assembled item is similar to the
            other catalog entries.  The part number
            is restricted to an assembly number.
            In addition, there may be no options.
            Finally, a part list is also needed.
            Note that the "includedQuantity" has
            a default of one, but can be overridden
            in instances.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType"
                         id="bacet.rst">
            <xsd:sequence>
                <xsd:element ref="assemblyID"/>
                <xsd:element name="partName"
```

```
                    type="partNameType"/>
                <xsd:element name="partNumber"
                    type="assemblyPartNumberType"/>
                <xsd:element name="partOption"
                    type="partOptionType"
                    minOccurs="0"
                    maxOccurs="0"/>
                <xsd:element
                     name="description"
                     type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                    type="xsd:positiveInteger"
                    default="1"/>
                <xsd:element name="customerReview"
                    type="customerReviewType"
                    minOccurs="0"
                    maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                    type="categoryType"
                    fixed="assembly"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>
```

11.10.1 Attributes of a `restriction` Element

The attributes applicable to a `restriction` element are identical to the attributes applicable to an `extension` element: the `base` attribute and the `id` attribute. Table 11.9 identifies these two attributes.

Table 11.9. Attribute Summary of a `restriction` Element

Attribute	*Description*
`base`	The value of the `base` attribute identifies the base simple or complex type of the current derived complex type.
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

11.10.1.1 The base Attribute of a restriction Element

The value of a base attribute indicates the global base structure type. For an extension of simple content, the base type can be either a simple type or a complex type with simple content. Otherwise, the derived complex type is an extension of complex content, which means that base structure type is a complex type.

Attribute Overview

restriction: base

Value:	A QName.
Default:	None.
Constraints:	The base type must refer to a global simple or a complex type in the XML schema.
Required:	No.

11.10.1.2 The id Attribute of a restriction Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

restriction: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 11.14 portrays a restriction element with an associated id attribute.

11.10.2 Content Options for a restriction Element

Both the complexContent and the simpleContent elements may contain a restriction element. Some of the content options described later in this section apply to the complexContent element, some to the simpleContent element, and some to both. The description of each element identifies the appropriate applicability.

Note the Applies To column in Table 11.10. The Applies To column indicates whether the element applies to simple content (the cell value is 'S'), complex content (the cell value is 'C'), or both (the cell value is 'S/C').

Table 11.10. Content Options for a `restriction` Element

Element	*Description*	*Applies To*
`all`	The all element, discussed in Section 11.11, specifies a model group that indicates an unordered set of element types; an XML instance must reference all the nested element types in any order.	C
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.	S/C
`anyAttribute`	The `anyAttribute` element, discussed in Section 9.7, specifies a set of namespaces, each of which may provide global attribute types; the XML instance may reference any of these attribute types.	S/C
`attribute`	The `attribute` element, discussed in Section 9.5, specifies a local attribute type or refers to a global attribute type; the XML instance must reference this attribute, unless the attribute is optional or prohibited.	S/C
`attributeGroup`	The `attributeGroup` element, discussed in Section 9.6, specifies a reference to a named attribute use group. An XML instance must reference each attribute type according to the appropriate nested model group.	S/C
`choice`	The `choice` element, discussed in Section 11.12, specifies a model group that indicates a selection of one element type from a set of element types; an XML instance must reference only one of the element types.	C
`group`	The `group` element, discussed in Section 11.14, references to a named model group. The named model group specifies another model group (that is, `all`, `choice`, or `sequence`)	C
`enumeration`	The value of an `enumeration` constraining facet specifies a possible (constant) value.	S
`fractionDigits`	The value of a `fractionDigits` constraining facet determines the number of digits after the decimal point in a decimal number.	S
`length`	The value of a `length` constraining facet specifies the exact length of a string of characters that must appear in an element instance after normalization. This constraining facet also specifies the number of items in a list (see the caveats regarding `hexBinary` and `base64binary` datatypes in Section 12.4.3).	S

maxExclusive	The value of the maxExclusive constraining facet specifies an upper bound on a numeric value. This boundary excludes the value specified.	S
maxInclusive	The value of the minInclusive constraining facet specifies an upper bound on a numeric value. This boundary includes the value specified.	S
maxLength	The value of the maxLength constraining facet specifies the maximum number of characters in a string after normalization or the maximum number of items in a list (see caveats on hexBinary and base64Binary in Section 12.4.3).	S
minExclusive	The value of the minExclusive constraining facet specifies a lower bound on a numeric value. This boundary excludes the value specified.	S
minInclusive	The value of a minInclusive constraining facet specifies a lower bound on a numeric value. This boundary includes the value specified.	S
minLength	The value of a minLength constraining facet specifies the minimum number of characters in a string after normalization, or the minimum number of items in a list (see caveats on hexBinary and base64Binary in Section 12.4.3).	S
pattern	The value of a pattern constraining facet specifies a regular expression often used to validate a character string. Specifically, the pattern constrains lexical representations.	S
totalDigits	The value of a totalDigits constraining facet specifies the total number of digits in a decimal number.	S
simpleType	The simpleType element specifies a local simple type.	S
whiteSpace	The value of a whiteSpace constraining facet determines the normalization of spaces, carriage returns, and line feeds in a string. In certain cases, the value also specifies implied transformations. This normalization frequently removes undesirable white space.	S
sequence	The sequence element, covered in Section 11.13, specifies a model group that indicates an ordered set of element types; an XML instance must reference all the nested element types in the schema order.	C

The content pattern for restriction is similar to the content pattern for the extension element, with one huge difference. When the restriction applies to simple content, the restriction may contain constraining facets that apply to the base (or locally described) simpleType element. Chapter 10 covers simple types. Chapter 12 covers built-in datatypes.

The content pattern for a restriction element nested within a simpleContent element is:

```
annotation?
(simpleType? (minExclusive |
```

```
                minInclusive |
                maxExclusive |
                maxInclusive |
                totalDigits |
                fractionDigits |
                length |
                minLength |
                maxLength |
                enumeration |
                whiteSpace |
                pattern)*)?
(attribute | attributeGroup)* anyAttribute?
```

The content pattern for the `restriction` element nested within a `complexContent` element is

```
annotation?
(group | all | choice | sequence)?
(attribute | attributeGroup)* anyAttribute?
```

11.11 The `all` Element

The `all` element specifies an unordered set of element types. For each element type associated with an `all` element in an XML schema document, there must be an element in the XML instance. However, they may appear in any order. In fact, there may be zero or many elements for each type, depending on the values of the `minOccurs` and `maxOccurs` attributes associated with the appropriate element type.

maxoccurs cannot have more than 1

Tip

The Schema Recommendation generally permits XML elements that represent a model group to contain an element representing another model group. This nesting typically excludes the `all` element.

Listing 11.3 provides an example of the `partOptionType` that describes a complex type containing an `all` element.

11.11.1 Attributes of an `all` Element

Other than the ubiquitous `id` attribute, the attributes of the `all` element describe whether the entire model group is optional or required. Table 11.11 details the attributes of the `all` element.

Table 11.11. Attribute Summary of an `all` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`maxOccurs`	The value of the `maxOccurs` attribute determines the maximum number of occurrences of this model group. The attribute is irrelevant, because `maxOccurs` is fixed to the value '1'.
`minOccurs`	The value of the `minOccurs` attribute determines the minimum number of occurrences of this model group. In fact, the `minOccurs` attribute determines whether the entire model group is optional or required as the value is limited to '`0`' or '`1`'.

11.11.1.1 The `id` Attribute of an `all` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`all: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Listing 11.3 portrays an `all` element with an associated `id` attribute.

11.11.1.2 The `maxOccurs` Attribute of an `all` Element

The `maxOccurs` attribute determines the maximum number of occurrences in the XML instance of elements whose element types are the set specified by the `all` element. Because the value of the `maxOccurs` attribute is fixed to '`1`', there can be at most one occurrence of the entire set of elements.

Attribute Overview

`all: maxOccurs`

Value:	'1'.
Default:	'1'.
Constraints:	Fixed; redundant if specified.
Required:	No.

11.11.1.3 The `minOccurs` Attribute of an `all` Element

The `minOccurs` attribute determines the minimum number of occurrences in the XML instance of elements whose element types are the entire set specified by the `all` Element. When the value of the `minOccurs` attribute is set to '0', the entire set of elements is optional.

Attribute Overview

`all: minOccurs`

Value:	'0' or '1'.
Default:	'1'.
Constraints:	None.
Required:	No.

11.11.2 Content Options for an `all` Element

An `all` element may contain an annotation and any number of elements. Table 11.12 details these content options.

Table 11.12. Content Options for an `all` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`element`	Each nested `element` describes an element type, zero or more instances of which might appear in an XML instance (as controlled by each element's `minOccur` and `maxOccur` attribute values). However, the elements may appear in any order.

The content pattern for an `all` element is:

```
annotation? element*
```

11.12 The choice Element

The choice element provides an XML representation for describing a selection from a set of element types. An XML instance contains elements whose element types are the set specified by the choice element in an XML schema document. The minOccurs and maxOccurs attributes may permit the XML instance to select several (for example, between two and four) occurrences of element types from the set. Listing 11.16 demonstrates the use of a choice element whose description is part of the global priceGroup. A XML instance must contain one of the following: a fullPrice element, a salePrice element, a clearancePrice element, or a freePrice element.

Listing 11.16 Selecting from a Set of Elements (pricing.xsd)

```
<xsd:group name="priceGroup" id="priceGroup.group">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A price is any one of the following:
                * Full Price (with amount)
                * Sale Price (with amount and authorization)
                * Clearance Price (with amount and
                  authorization)
                * Free (with authorization)
        </xsd:documentation>
    </xsd:annotation>
    <xsd:choice id="pg.choice">
        <xsd:element name="fullPrice"
                     type="fullPriceType"/>
        <xsd:element name="salePrice"
                     type="salePriceType"/>
        <xsd:element name="clearancePrice"
                     type="clearancePriceType"/>
        <xsd:element name="freePrice" type="freePriceType"/>
    </xsd:choice>
</xsd:group>
```

The XML representation of an element type that ultimately contains the priceGroup from Listing 11.16 might look like the following element:

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

```
        <xsd:group ref="priceGroup"/>
    </xsd:complexType>
</xsd:element>
```

Given the preceding element type, the following is valid in an XML instance:

```
<price>
    <freePrice authorization="CB"/>
</price>
```

Because of the `choice` element, an alternative to the previous element is to use `<fullPrice>` instead of `<freePrice>` in an XML instance:

```
<price>
    <fullPrice>32.00</fullPrice>
</price>
```

11.12.1 Attributes of a `choice` Element

Other than the ubiquitous `id` attribute, the attributes of the `choice` element describe the number of times each element type may occur. Table 11.13 details the attributes of the `choice` element.

Note

A `choice` element with no contents is invalid.

Table 11.13. Attribute Summary of a `choice` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`maxOccurs`	The value of the `maxOccurs` attribute determines the maximum number of occurrences of the element types represented in an XML instance. Each element in an XML instance must reference an element type that is a member of the set of element types defined by the `choice` element. Furthermore, the maximum number of occurrences of each element type is the product of the `maxOccurs` attribute of the `choice` element and the `maxOccurs` attribute of the element type.

`minOccurs`	The value of the `minOccurs` attribute determines the minimum number of occurrences of the element types represented in an XML instance. Each element in an XML instance must reference an element type that is a member of the set of element types defined by the `choice` element. Furthermore, the minimum number of occurrences of each element type is the product of the `minOccurs` attribute of the `choice` element and the `minOccurs` attribute of the element type.

Caution

The quantity of elements that may appear in an XML instance is constrained by the `minOccurs` and `maxOccurs` attributes of the `choice` element as well as the `minOccurs` and `maxOccurs` attributes of the particular element type.

11.12.1.1 The `id` Attribute of a `choice` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`choice: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Listing 11.16 demonstrates a `choice` element with an associated `id` attribute.

11.12.1.2 The `maxOccurs` Attribute of a `choice` Element

The `maxOccurs` attribute determines the maximum number of occurrences in the XML instance of elements whose element types are the set specified by the `choice` element. Note that—as described in Table 11.13—the maximum number of elements associated with each element type cannot be more than the product of the value of the `maxOccurs` attribute of the `choice` element and the value of the `maxOccurs` attribute on the element type.

Attribute Overview

choice: maxOccurs

Value:	A non-negative integer or 'unbounded'.
Default:	'1'.
Constraints:	None.
Required:	No.

11.12.1.3 The minOccurs Attribute of a choice Element

The minOccurs attribute determines the minimum number of occurrences in the XML instance of elements whose element types are the set specified by the choice element. Note that—as described in Table 11.13—the minimum number of elements associated with each element type cannot be more than the product of the value of the minOccurs attribute of the choice element and the value of the minOccurs attribute on the element type.

Attribute Overview

choice: minOccurs

Value:	A non-negative integer.
Default:	'1'.
Constraints:	None.
Required:	No.

11.12.2 Content Options for a choice Element

Excluding annotations, the content options for the choice element must ultimately resolve to a set of element types. Table 11.14 portrays the options available for a choice element.

Table 11.14. Content Options for a choice Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.

any	When a choice element contains an immediate any subelement, the XML instance might contain any set of global element types from the namespaces that the any element specifies. Section 8.4 covers the any element.
choice	When a choice element contains another immediate choice subelement, the XML instance might contain one of the elements that correspond to an appropriate "selection" from the enclosed choice.
element	Normally a choice element contains a number of element types, each of whose XML representation is an element.
group	When a choice element contains an immediate group subelement, the XML instance might contain a set of elements whose element types are the set constrained by the referenced named model group. The contents of the group determines the valid elements.
sequence	When a choice element contains an immediate sequence subelement, the XML instance might contain an ordered set of elements whose element types are the set specified by the enclosed sequence.

The content pattern for a choice element is:

```
annotation? (element | group | choice | sequence | any)*
```

Warning

No 'all' as child element

A choice element may not contain an immediate all subelement. However, the group element may contain an all subelement. The transitive relationship that might make a choice indirectly contain an all is invalid, an XML validator might not be able to determine veracity.

11.13 The sequence Element

The sequence element provides an XML representation of an ordered set of element types. For each element type associated with a sequence element in an XML schema document, there must be an element in the XML instance, in the same order. In fact, there may be zero or many elements for each type, depending on the values of the minOccurs and maxOccurs attributes associated with the element types.

The baseCatalogEntryType described in Listing 11.7 contains a sequence describing an ordered set of element types.

11.13.1 Attributes of a sequence Element

Other than the ubiquitous id attribute, the attributes of the sequence element describe the number of times

each element type must occur. Table 11.15 details the attributes of the `sequence` element.

Table 11.15. Attribute Summary of a sequence Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`maxOccurs`	The value of the `maxOccurs` attribute determines the maximum number of occurrences of the element types represented in an XML instance. Each element in an XML instance must reference an element type that is a member of the set of element types defined by the `sequence` element. Furthermore, the maximum number of occurrences of each element type is the product of the `maxOccurs` attribute of the `choice` element and the `maxOccurs` attribute of the element type.
`minOccurs`	The value of the `minOccurs` attribute determines the minimum number of occurrences of the element types represented in an XML instance. Each element in an XML instance must reference each element type, in order, that is a member of the set of element types defined by the `sequence` element. Furthermore, the minimum number of occurrences of each element type is the product of the `minOccurs` attribute of the `choice` element and the `minOccurs` attribute of the element type.

Caution

The quantity of elements that may appear in an XML instance is constrained by the `minOccurs` and `maxOccurs` attributes of the `sequence` element as well as the `minOccurs` and `maxOccurs` attributes of the particular element type.

11.13.1.1 The id Attribute of a sequence Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`sequence: id`

Value:	An ID.
Default:	None.

Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Listing 11.7 demonstrates a `sequence` element with an associated `id` attribute.

11.13.1.2 The `maxOccurs` Attribute of a `sequence` Element

The `maxOccurs` attribute determines the maximum number of times an XML instance may reference the entire set of element types.

Attribute Overview

`sequence: maxOccurs`

Value:	A non-negative integer or '`unbounded`'.
Default:	'`1`'.
Constraints:	None.
Required:	No.

11.13.1.3 The `minOccurs` Attribute of a `sequence` Element

The `minOccurs` attribute determines the minimum number of times an XML instance may reference the entire set of element types.

Attribute Overview

`sequence: minOccurs`

Value:	A non-negative integer or '`unbounded`'.
Default:	'`1`'.
Constraints:	None.
Required:	No.

11.13.2 Content Options for a `sequence` Element

Excluding annotations, the content options for the `sequence` element must ultimately resolve to a set of

element types. Table 11.16 portrays the options available for a `sequence` element.

The content pattern for a sequence element is:

all is not here

```
annotation? (element | group | choice | sequence | any)*
```

Table 11.16. Content Options for a `sequence` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`any`	When a `sequence` element contains an immediate `any` subelement, the XML instance might contain any set of global element types from the namespaces that the any element specifies. Section 8.4 covers the `any` element.
`choice`	When a `sequence` element contains an immediate `choice` subelement, the XML instance might contain the sequence of elements that correspond to an appropriate "selection" from the enclosed `choice`.
`element`	Normally a `sequence` element contains a number of element types, each of whose XML representation is an `element`.
`group`	When a `sequence` element contains an immediate `group` subelement, the XML instance might contain a set of elements whose element types are the set specified by the referenced named model group. The contents of the `group` determines the valid elements.
`sequence`	When a `sequence` element contains another immediate `sequence` sub-element, the XML instance might contain an ordered set of elements whose element types are the set specified by the enclosed `sequence`.

Warning

A `sequence` element may not contain an immediate `all` subelement. However, the `sequence` element may contain an `all` subelement. The transitive relationship that might make a `sequence` indirectly contain an `all` is invalid, an XML validator might not be able to determine veracity.

11.14 The `group` Element

The `group` element describes a named model group. A named model group encapsulates an `all` element, a

choice element, or a sequence element. When a complex type incorporates a named model group (via the ref attribute), the scope of the enclosed components change: The namespace for these components becomes local to the parent element type. Note that there might be many different parent element types when the complex type is global, and the complex type is the structure type of multiple element types.

Listing 11.17 repeats the XML representation of priceGroup. In addition, Listing 11.17 portrays the utility of this group by repeatedly including priceGroup in various catalog entry types.

Listing 11.17 Reuse of a group Element (pricing.xsd and catalog.xsd)

```
<xsd:group name="priceGroup" id="priceGroup.group">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A price is any one of the following:
                * Full Price (with amount)
                * Sale Price (with amount and authorization)
                * Clearance Price (with amount and
                  authorization)
                * Free (with authorization)
        </xsd:documentation>
    </xsd:annotation>
    <xsd:choice>
        <xsd:element name="fullPrice"
                     type="fullPriceType"/>
        <xsd:element name="salePrice"
                     type="salePriceType"/>
        <xsd:element name="clearancePrice"
                     type="clearancePriceType"/>
        <xsd:element name="freePrice" type="freePriceType"/>
    </xsd:choice>
</xsd:group>

<xsd:complexType name="baseCatalogEntryType"
                 abstract="true"
                 id="baseCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A catalog entry must have:
                * A database ID
                * Part Name
                * Part Number
                * Options available
```

```
                * Description
                * Price
                * Included Quantity when ordering
                  one item.
            The "baseCatalogEntryType" is
            non-instantiable:  a derived type must
            be created before a catalog
            entry can be instantiated.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:sequence id="bacet-seq">
        <xsd:element ref="sequenceID"/>
        <xsd:element name="partName" type="partNameType"/>
        <xsd:element name="partNumber" type="partNumberType"/>
        <xsd:element name="partOption" type="partOptionType"/>
        <xsd:element name="description"
                     type="catalogEntryDescriptionType"/>
        <xsd:group ref="priceGroup"/>
        <xsd:element name="includedQuantity"
                     type="xsd:positiveInteger"/>
        <xsd:element name="customerReview"
                     type="customerReviewType"
                            minOccurs="0"
                            maxOccurs="unbounded"/>
    </xsd:sequence>
    <xsd:attribute name="category"
                   type="categoryType"
                   use="required"/>
</xsd:complexType>

<xsd:complexType name="unitCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="unitCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A unit item contains nothing more
            or less than a basic catalog entry ID:
                * A database ID
                * Part Name
                * Part Number
                * Options available
```

```
                * Price
                * Included Quantity when ordering
                  one item (always one for unit items).
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType">
            <xsd:sequence>
                <xsd:element ref="unitID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="unitPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"/>
                <xsd:element name="description"
                           type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             fixed="1"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                           type="categoryType"
                           fixed="unit"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="bulkCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="bulkCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            A bulk item is just like any
            other, except that the part
            number is restricted to a
            bulk part number.
```

```
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType">
            <xsd:sequence>
                <xsd:element ref="bulkID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="bulkPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"/>
                <xsd:element name="description"
                             type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                           type="categoryType"
                           fixed="bulk"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType
        name="baseAssemblyCatalogEntryType"
        abstract="true"
        block="#all"
        final="restriction"
        id="baseAssemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            An assembled item is similar to the
            other catalog entries.  The part number
            is restricted to an assembly number.
            In addition, there may be no options.
            Finally, a part list is also needed.
            Note that the "includedQuantity" has
```

```
            a default of one, but can be overridden
            in instances.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:restriction base="baseCatalogEntryType"
                         id="bacet.rst">
            <xsd:sequence>
                <xsd:element ref="assemblyID"/>
                <xsd:element name="partName"
                             type="partNameType"/>
                <xsd:element name="partNumber"
                             type="assemblyPartNumberType"/>
                <xsd:element name="partOption"
                             type="partOptionType"
                             minOccurs="0"
                             maxOccurs="0"/>
                <xsd:element name="description"
                            type="catalogEntryDescriptionType"/>
                <xsd:group ref="priceGroup"/>
                <xsd:element name="includedQuantity"
                             type="xsd:positiveInteger"
                             default="1"/>
                <xsd:element name="customerReview"
                             type="customerReviewType"
                             minOccurs="0"
                             maxOccurs="unbounded"/>
            </xsd:sequence>
            <xsd:attribute name="category"
                           type="categoryType"
                           fixed="assembly"/>
        </xsd:restriction>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="assemblyCatalogEntryType"
                 block="#all"
                 final="#all"
                 id="assemblyCatalogEntryType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The actual definition of an assembly,
            including the contained parts.
```

```
        </xsd:documentation>
    </xsd:annotation>
    <xsd:complexContent>
        <xsd:extension base="baseAssemblyCatalogEntryType"
                       id="acet.ext">
            <xsd:sequence>
                <xsd:element name="partList"
                             type="partNumberListType"/>
            </xsd:sequence>
        </xsd:extension>
    </xsd:complexContent>
</xsd:complexType>

<xsd:complexType name="catalogType"
                 id="catalogType.catalog.cType">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            This catalog type must be altered
            every time a new catalog entry
            type is created.  The
            "catalogType2" complex type refers
            only instantiable derived classes.
                -- Shorthand Notation --
        </xsd:documentation>
    </xsd:annotation>
    <xsd:choice minOccurs="1"
                maxOccurs="unbounded">
        <xsd:element name="unitPart"
                     type="unitCatalogEntryType"
                     block="restricion extension"/>
        <xsd:element name="bulkPart"
                     type="bulkCatalogEntryType"
                     block="restricion extension"/>
        <xsd:element name="assemblyPart"
                     type="assemblyCatalogEntryType"
                     block="restricion extension"/>
    </xsd:choice>
</xsd:complexType>
```

A `catalog` element whose structure type is the preceding element type might look like the following:

```
<xsd:element name="catalog" type="catalogType"/>
```

Given the preceding element, the following is a valid `catalog` XML instance:

```
<catalog>
    <unitPart>
        <unitID>10327</unitID>
        <partName>Unit 1</partName>
        <partNumber>UX001</partNumber>
        <partOption>
            <color>magenta</color>
        </partOption>
        <description>Unit 1 is the thing to buy</description>
        <fullPrice>28.00</fullPrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <unitPart>
        <unitID>10329</unitID>
        <partName>Unit 2</partName>
        <partNumber>UX002</partNumber>
        <partOption>
            <color>Pink Grapefruit</color>
        </partOption>
        <description>Unit 2 lasts longer than Unit 1</description>
        <fullPrice>28.01</fullPrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <bulkPart>
        <bulkID>40000397</bulkID>
        <partName>Bulk 1</partName>
        <partNumber>BLK2088</partNumber>
        <partOption />
        <description>
            This includes 1,000
            <partList>UX002</partList>
            pieces.
        </description>
        <fullPrice>1700.00</fullPrice>
        <includedQuantity>200</includedQuantity>
    </bulkPart>
    <unitPart>
        <unitID>40004787</unitID>
        <partName>Unit 3</partName>
        <partNumber>UX003</partNumber>
        <partOption />
```

```
        <description>
            This is obsolete ... on sale
        </description>
        <salePrice employeeAuthorization="MWS"
                   managerAuthorization="ALB">28.03</salePrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <unitPart>
        <unitID>40004787</unitID>
        <partName>Unit 4</partName>
        <partNumber>UX004</partNumber>
        <partOption />
        <description>
            This is also obsolete ... on clearance
        </description>
        <clearancePrice employeeAuthorization="MWS"
                   managerAuthorization="ALB">8.57</clearancePrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <assemblyPart>
        <assemblyID>40004788</assemblyID>
        <partName>Assembly 1</partName>
        <partNumber>ASM9001</partNumber>
        <description>
            This is made up of both
            <partList>UX002 UX003</partList>
            pieces which makes assembly 1
            better than the competition.
        </description>
        <fullPrice>3200.00</fullPrice>
        <includedQuantity>1</includedQuantity>
        <customerReview>
            <customerName>I. M. Happy</customerName>
            <customerFeedback>This is the most
awesome phat product in the entire
    world.</customerFeedback>
        </customerReview>
        <partList>UX002 UX003</partList>
    </assemblyPart>
</catalog>
```

11.14.1 Attributes of a group Element

Named model groups have a name. A local `group` element must contain a reference to a global named model group. The other attributes of a named model group are similar to other tyes of groups. Table 11.17 describes the attributes for a named model group.

Table 11.17. Attribute Summary of a `group` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`maxOccurs`	The value of the `maxOccurs` attribute determines the maximum number of occurrences of the model group represented in an XML instance. The XML instance must reference element types in accordance with the rules of the enclosed model group. Furthermore, the maximum number of occurrences of each element type at most is the product of the `maxOccurs` attribute of the `group` element and the `maxOccurs` attribute of the element type.
`minOccurs`	The value of the `minOccurs` attribute determines the minimum number of occurrences of the model group represented in an XML instance. The XML instance must reference element types in accordance with the rules of the enclosed model group. Furthermore, the minimum number of occurrences of each element type is the product of the `minOccurs` attribute of the `group` element and the `minOccurs` attribute of the element type, assuming that the element is selected at all.
`name`	The value of the `name` attribute is the name of the named model group. One or more complex types might reference this named model group.
`ref`	The value of a `ref` attribute is a reference to a named model group.

11.14.1.1 The `id` Attribute of a `group` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`group: id`

Value:	An ID.
Default:	None.
Constraints:	An `id` must be unique within an XML schema.
Required:	No.

Listing 11.16 demonstrates a `group` element with an associated `id` attribute.

11.14.1.2 The `maxOccurs` Attribute of a `group` Element

The `maxOccurs` attribute determines the maximum number of times an XML instance may reference the entire set of element types. This attribute is not valid for the global `group`:This attribute is only valid when the `group` element is enclosed within a `complexType`.

Attribute Overview

`group: maxOccurs`

Value:	A non-negative integer or '`unbounded`'.
Default:	'`1`'.
Constraints:	None.
Required:	No.

11.14.1.3 The `minOccurs` Attribute of a `group` Element

The `minOccurs` attribute determines the minimum number of times an XML instance may reference the entire set of element types. This attribute is not valid for a global `group`:This attribute is only valid when the `group` element is enclosed within a `complexType`.

Attribute Overview

`group: minOccurs`

Value:	A non-negative integer.
Default:	'`1`'.
Constraints:	None.
Required:	No.

11.14.1.4 The `name` Attribute of a `group` Element

The value of the `name` attribute is the name of a global model group. A complex type specifies a global named model group by referring to this name.

Attribute Overview

group: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	No.

11.14.1.5 The ref Attribute of a group Element

The ref attribute references a global group.

Attribute Overview

group: ref

Value:	A QName.
Default:	None.
Constraints:	The name must refer to a named model group (that is, a group element).
Required:	No.

11.14.2 Content Options for a group Element

Excluding annotations, the content options for the group element must ultimately resolve to a set of element types. Table 11.18 contains the options available for a group element.

Table 11.18. Content Options for a group Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.
all	An all element is the representation of an all model group.
choice	A choice element is the XML representation of a choice model group Schema component.

`sequence`	A `sequence` element is the XML representation of a sequence model group.

The content pattern for a `group` element is:

```
annotation? (all | choice | sequence)
```

CONTENTS

Chapter 12. Built-in Datatypes

no need to second time revision for this chapter. While coding simply refer data type hierarchy diagram and my excel sheet for applicable constraining facets

IN THIS CHAPTER

A datatype has two primary components, plus "facets" that are used to describe derived datatypes. The two primary components are a value space and a lexical space. The character strings in the lexical space serve as names for the abstract objects in the value space. There must, of course be a prescribed mapping from lexical-space character strings to value-space values. It's the lexical-space strings that actually occur in XML documents.

On the other hand, most restriction derivations involve restrictions on permitted values in the value space, so when defining restrictions (via an appropriate `simpleType` element in a schema document), you should be aware of the interrelationship between the two. Apart from this, *values* occur at only one point in schema processing. When a default value is prescribed in the schema, the abstract value is recorded in the appropriate place; when the default is placed into an information item, it is converted back to one of the strings in the lexical space that "names" that value.

infoset

Specifically, the name to be selected—if there is more than one candidate—is also prescribed as part of the datatype's definition. Every value in the value space is expected to have a distinguished name in the lexical space; that name is the *canonical representation*. The canonical representation is sometimes also used by applications that deal directly with the values when they need to create an XML representation of a result.

abstract representation --> infoset --> lexical space --> value space
|
canonical rep.

Warning

Derived datatypes normally use the same canonical representation as the base type. However, it is possible, by using a *pattern* facet, to create a derived type that does not have canonical representations from the base type remaining in the lexical space. No way is provided for users to define new canonical representations. Therefore, users should be very careful when using pattern facets; carelessness could produce a datatype with no canonical representations for some or all values, and the defaulting mechanism of schemas is broken for such a datatype. This is a known problem in the current Schema Recommendation, and will undoubtedly be addressed in a subsequent revision.

Also, note that the `integer` datatype is derived by restriction from `decimal`; it too suffers from this problem. The Schema Recommendation solves the problem for `integer` "by fiat"—

CONTENTS

Chapter 13. Identity Constraints

12 - June - 08

18 - Sep - 08

25 Page

IN THIS CHAPTER

Chapter 1 defines an XML validator as an XML parser layer that validates an XML instance against an XML schema. An XML validator is effectively the front end of a compiler. Specifically, a "vanilla" XML parser understands XML lexical tokens and parses against a simple XML grammar. Most of the features of a DTD place further constraints on the grammar. The newer XML Schema provides an even more complete grammar definition.

When a compiler has finished parsing, the next step is typically semantic validation. Semantic validation assures the veracity of the *meaning* of the data, whereas the aforementioned parsing assures the veracity of the *structure* of the data.

Identity constraints provide for some limited semantic validation. In particular, the `unique` and `key` elements enforce data uniqueness; the `keyref` element enforces referential integrity.

The last step in a compiler (barring optional steps like optimization) is code generation. XSLT arguably provides a rudimentary mechanism for specifying code generation.

13.1 Identity Constraint Example

The examples in this section provide a fairly condensed, but complete XML schema and a corresponding XML instance. Listing 13.1 demonstrates more than one example of each type of identity constraint (`unique`, `key`, and `keyref`). The remaining sections of this chapter reiterate excerpts from this schema to emphasize specific characteristics of the identity constraints.

Listing 13.1 Compressed Order Data Schema (`idConstraintDemo.xsd`)

```
<xsd:complexType name="compressedOrderType">
    <xsd:sequence>
        <xsd:element name="order"
                     minOccurs="1"
```

```
                maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:sequence>
                    <xsd:element name="customerID"
                                 type="xsd:token"/>
                    <xsd:element name="item"
                                 minOccurs="1"
                                 maxOccurs="unbounded">
                        <xsd:complexType>
                            <xsd:sequence minOccurs="1"
                                          maxOccurs="unbounded">
                                <xsd:element
                                     name="partNumber"
                                     type="partNumberType"/>
                                <xsd:element name="quantity"
                                             type="xsd:positiveInteger"/
>
                            </xsd:sequence>
                        </xsd:complexType>
                    </xsd:element>
                </xsd:sequence>
                <xsd:attribute name="totalQuantity"
                               type="xsd:positiveInteger"
                               use="required"/>
                <xsd:attribute name="orderID"
                               type="xsd:token"
                               use="required"/>
                <xsd:attribute name="shipmentID"
                               type="xsd:token"
                               use="optional"/>
            </xsd:complexType>

            <xsd:key name="orderPartNumberKey">
                <xsd:annotation>
                    <xsd:documentation xml:lang="en">
                        The part number must be unique
                        within the order, but NOT
                        across orders.
                    </xsd:documentation>
                </xsd:annotation>
                <xsd:selector xpath="item"/>
                <xsd:field xpath="partNumber"/>
            </xsd:key>
```

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

<xsd:complexType name="shippedOrderSummaryType">
    <xsd:sequence>
        <xsd:element name="shipped"
                     minOccurs="1"
                     maxOccurs="unbounded">
            <xsd:complexType>
                <xsd:attribute name="shipmentID"
                               type="xsd:token"/>
                <xsd:attribute name="custID"
                               type="xsd:token"/>
                <xsd:attribute name="qty"
                               type="xsd:positiveInteger"/>
            </xsd:complexType>
        </xsd:element>
    </xsd:sequence>
</xsd:complexType>

<xsd:element name="idConstraintDemo">
    <xsd:complexType>
        <xsd:sequence>

            <xsd:element name="compressedOrder"
                         type="compressedOrderType">

                <xsd:key name="orderKey"
                         id="orderKey.key">
                    <xsd:annotation>
                        <xsd:documentation xml:lang="en">
                            The customerID uniquely identifies
                            each customer.
                        </xsd:documentation>
                    </xsd:annotation>
                    <xsd:selector xpath="order"
                                  id="orderKey.key.selector"/>
                    <xsd:field xpath="@orderID"
                               id="orderKey.key.field"/>
                </xsd:key>

                <xsd:unique name="orderShippedUnique"
                            id="orderShippedUnique.unique">
```

same structure type, has two key

what is the diff between key and unique ??

key - force the element to HAVE (existence) of unique value. but UNIQUE give assurance for uniqueness only if there value existence

```
                <xsd:annotation>
                    <xsd:documentation xml:lang="en">
                        The customerID uniquely identifies
                        each customer.
                    </xsd:documentation>
                </xsd:annotation>
                <xsd:selector xpath="order"/>
                <xsd:field xpath="@shipmentID"/>
            </xsd:unique>

        </xsd:element>

        <xsd:element name="shippedOrderSummary"
                     type="shippedOrderSummaryType"/>

        <xsd:element ref="customerList"/>
        <xsd:element name="catalog" type="catalogType"/>

    </xsd:sequence>

</xsd:complexType>

<xsd:key name="customerKey">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The customerID uniquely identifies
            each customer.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="customerList/*"/>
    <xsd:field xpath="@customerID"/>
</xsd:key>

<xsd:keyref name="customerRef" refer="customerKey">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Each order must refer to a known customer.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="compessedOrder/order"/>
    <xsd:field xpath="customerID"/>
</xsd:keyref>

<xsd:key name="partNumberKey"
```

```
            id="partNumberKey.key">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                The part number uniquely identifies
                each orderable item.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="catalog/*"/>
        <xsd:field xpath="partNumber"/>
    </xsd:key>

    <xsd:keyref name="partNumberRef"
                id="partNumberRef.keyref"
                refer="partNumberKey">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                Each order must refer to a known part number.
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="compressedOrder/order/item"/>
        <xsd:field xpath="partNumber"/>
    </xsd:keyref>

    <xsd:key name="orderedItemsKey"
             id="orderedItemsKey.key">
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                Identify ordered customers and
                shipping quantities.  Note that
                while schema already enforces
                the following constraint, the
                'key' also assures that there *is*
                a customerID and a shippingQuantity
                identified for each order
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="compressedOrder/order"/>
        <xsd:field xpath="@totalQuantity"/>
        <xsd:field xpath="customerID"/>
    </xsd:key>

    <xsd:keyref name="shippedItemsRef"
                id="shippedItems.keyref"
                refer="orderedItemsKey">
```

```
        <xsd:annotation>
            <xsd:documentation xml:lang="en">
                Ensure a shipped customer/qty
                for each ordered customer/qty
            </xsd:documentation>
        </xsd:annotation>
        <xsd:selector xpath="shippedOrderSummary/shipped"/>
        <xsd:field xpath="@qty"/>
        <xsd:field xpath="@custID"/>
    </xsd:keyref>

</xsd:element>
```

Listing 13.2 provides an XML instance that corresponds to the XML schema introduced in Listing 13.1.

Listing 13.2 Compressed Order Data Instance (`idConstraintDemo.xml`)

```
<idConstraintDemo>

<compressedOrder>
    <order orderID="X012532"
           shipmentID="X012432-S"
           totalQuantity="3">
        <customerID>AW132E57</customerID>
        <item>
            <partNumber>ASM9001</partNumber>
            <quantity>3</quantity>
        </item>
    </order>
    <order orderID="X012533"
           totalQuantity="2">
        <customerID>AW220A38</customerID>
        <item>
            <partNumber>UX001</partNumber>
            <quantity>1</quantity>
        </item>
        <item>
            <partNumber>UX002</partNumber>
            <quantity>1</quantity>
        </item>
    </order>
</compressedOrder>

<shippedOrderSummary>
```

```
    <shipped shipmentID="X012432-S" custID="AW132E57" qty="3"/>
</shippedOrderSummary>

<customerList>
    <businessCustomer customerID="SAM132E57">
        <name>Cliff Binstock</name>
        <phoneNumber>503-555-0000</phoneNumber>
        <address>
            <street>123 Gravel Road</street>
            <city>Nowheresville</city>
            <state>OR</state>
            <country>US</country>
            <zip>97000</zip>
            <effectiveDate>2001-11-08</effectiveDate>
        </address>
    </businessCustomer>
</customerList>

<catalog>
    <unitPart>
        <unitID>10327</unitID>
        <partName>Unit 1</partName>
        <partNumber>UX001</partNumber>
        <partOption>
            <color>magenta</color>
        </partOption>
        <description>Unit 1 is the thing to buy</description>
        <fullPrice>28.00</fullPrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <unitPart>
        <unitID>10329</unitID>
        <partName>Unit 2</partName>
        <partNumber>UX002</partNumber>
        <partOption>
            <color>Pink Grapefruit</color>
        </partOption>
    <description>Unit 2 lasts longer than Unit 1</description>
        <fullPrice>28.01</fullPrice>
        <includedQuantity>1</includedQuantity>
    </unitPart>
    <assemblyPart>
        <assemblyID>40004788</assemblyID>
        <partName>Assembly 1</partName>
        <partNumber>ASM9001</partNumber>
```

```
        <description>
            This is made up of both
            <partList>UX002 UX003</partList>
            pieces which makes this assembly
            better than the competition.
        </description>
        <fullPrice>3200.00</fullPrice>
        <includedQuantity>1</includedQuantity>
        <customerReview>
            <customerName>I. M. Happy</customerName>
            <customerFeedback>This is the most
awesome phat product in the entire.
    world.</customerFeedback>
        </customerReview>
        <partList>UX002 UX003</partList>
        <status>3</status>
    </assemblyPart>
</catalog>

</idConstraintDemo>
```

13.2 Concepts and Observations

This section covers terminology unique to identity constraints. Furthermore, this section expounds upon two of the new terms: *selector* and *field*. Every identity constraint specifies a selector and one or more fields.

13.2.1 Identity Constraint Terminology

Identity constraints are a very small part of the Schema Recommendation. There is some terminology specific to identity constraints. The following list provides the terminology that is the foundation of the text in this chapter:

Node: The PSVI describes a tree that represents an XML instance. The tree is made up of a set of nodes (See Chapter 2). Some of these nodes, particularly those associated with elements and attributes, provide the infrastructure for the validation provided by identity constraints.

Selector: In an XML schema, each identity constraint specifies a selector. Technically, the value of a selector is an XPath that identifies an element. From the perspective of writing the schema, the XPath identifies an element type. The XPath must specify an element type that is a descendant of the element type enclosing the identity constraint.

Field: In an XML schema, each identity constraint specifies one or more fields. Technically, the value of each field is an XPath that identifies a child element or an attribute of the element identified by the XPath of the corresponding selector. From the perspective of writing the schema, the XPath identifies an element type or attribute type. The XPath must specify an element type or attribute type that is the element type—or is a descendant of the element type—that encloses the identity constraint.

Target node set: During XML instance validation, the target node set is the set of nodes identified by the selector of an identity constraint.

Key sequence: During XML instance validation, the key sequence is the set of values associated with the elements and attributes identified by the fields of an identity constraint. There is one key sequence for each node in the target node set.

13.2.2 Selectors and Fields

Each identity constraint identifies a set of nodes that must be unique or that require referential integrity. The grammar for locating these nodes is a subset of XPath (see Chapter 4). Each identity constraint has a specific scope, which is the enclosing element type: Any element type may provide any number of identity constraints.

Each identity constraint specifies a selector and one or more fields. The value of the XPath of each selector ultimately identifies a target node set in a corresponding XML instance. Each field, which is slightly more complicated, identifies one of the following:

- The value of an element corresponding to a node in the target node set
- The value of an attribute of the element corresponding to a node in the target node set
- The value of a descendant of the element corresponding to a node in the target node set
- The value of an attribute of a descendant of the element corresponding to a node in the target node set

Like a selector, a field ultimately identifies an element or an attribute. The XPath specified by the field is relative to the XPath specified by the selector. A key sequence is the set of values for a specific node in the target set that correspond to the set of fields specified by an identity constraint.

13.2.3 Limited XPath Support

Element types provide the context for identity constraints. The XPath for each selector is relative to the enclosing element type. Furthermore, the XPath for each field is relative to the XPath for each selector.

Listing 13.3 is the grammar for the XPath expressions for selectors and fields.

Listing 13.3 Selector and Field XPath Grammar

```
<selector> ::= <path> |
               <selector> '|' <path>

<field> ::= <path> |
            <attributeStep> |
            <path> <attributeStep>

<path>     ::= './/' |
               <path> |
               './/' <path>
```

```
<path> ::= <elementStep> |
           <path> '/' <elementStep>

<elementStep> ::= '.' | <nameTest>

<attributeStep> ::= '@' <nameTest>

<nameTest> ::= <qName> | '*' | <ncName> ':*'
```

Note that a selector may specify multiple XPaths.

Table 13.1 provides some sample XPath values. Note that each XPath for a selector locates a particular element. Each field locates the value of a particular element or attribute.

Table 13.1. Sample Selector and Field XPaths

XPath	*Description*	*Structure*
`.`	The current element	<current>foo</current>
`xyz`	The `xyz` child element of the current element	<current> <xyz>foo</xyz> <xyz>bar</xyz> </current>
`abc/xyz`	The `xyz` child elements of the `abc` child elements of the current element	<current> <abc> <xyz>foo</xyz> </abc> <abc> <xyz>bar</xyz> </abc> </current>
`.//xyz`	Any `xyz` descendant element of the current element	<current> <xyz>foo</xyz> <abc> <xyz>bar</xyz> <abc/> <current>
`*/xyz`	The `xyz` child elements of all child elements of the current element	<current> <abc> <xyz>foo</xyz> </abc> <def> <xyz>bar</xyz> </def> </current>
`@attr`	The `attr` attribute of the current element	<current attr="1"/>

`xyz/@attr`	The `attr` attribute of the `xyz` child element	<current> <xyz @attr="1"> <xyz @attr="2"/> </ current>
`.//@attr`	The `attr` attributes of any descendant element of the current element	<current> <xyz @attr="1"/> <abc @attr="2"> <xyz @attr="3"/> <abc/> <current>
`*/@attr`	The `attr` attributes of all immediate child elements of the current element	<current> <abc @attr="1"> <def @attr="2"/> </ current>

13.2.4 Value Equality

The foundation of identity constraints is a test for equality. Each key sequence specified by a `unique` or `key` identity constraint must be distinct: The test is equality (or more correctly, lack of equality). Additionally, each key sequence specified by a `keyref` identity constraint must exist as a key sequence in the target node set to which the `keyref` refers: Again, the test is equality. In an XML schema, two (simple type) values are *equal* when

- The simple types for the two values are identical, or they both have a common derivation ancestor. The derived simple types may be built-in derived datatypes or user-derived. (The idea is that value spaces in primitive datatypes are disjoint "by fiat" even if, for example, they both are numeric and have value spaces one of which is a nominal subset of the other.)
- The values in the value space are equal (identical).

For example, none of the `string` '3', the `integer` 3, the `float` 3, or the `double` 3 are equal. Conversely, any of the following with the value 3 are equal: an `unsignedInt`, an `unsignedLong`, a `nonNegativeInteger`, an `integer`, or a `decimal`. Logically, the representations in the lexical space do not have to be identical. For example, the `float` value represented by '3.0' is equal to (the same as) the `float` value represented by '3'.

13.2.5 Enforcing Uniqueness

Both the `unique` and `key` identity constraints assure the uniqueness of each set of values—the key sequence—in a corresponding target node set. The `key` identity constraint adds an additional constraint: For each node in the target node set, there *must be* a corresponding key. The `unique` identity constraint, however, assures uniqueness *only when the keys exist*.

This section demonstrates the difference between `unique` and `key`. In Listing 13.2, each order has a required order identifier and an optional shipment identifier. Each order identifier is unique. Each shipment identifier is also unique, although the company has not yet shipped all orders. The `key` element is appropriate for enforcing uniqueness on the order identifier, because each order has a *required* unique order identifier. On the other hand, the

`unique` element is appropriate for enforcing uniqueness on the shipment identifier, because each `order` has an *optional*, yet unique shipment identifier.

Note

The uniqueness of the key sequence specified by the set of `field` elements is relative to the element specified by the `selector` element. For example, in Listing 13.1, the `compressedOrder` contains an `order`, which contains (ultimately) a `partNumber`. Note that the `orderPartNumberKey` ensures that an order has only one reference to any part number. On the other hand, many customers, with distinct orders, might order the same part number. Therefore, the selector specifies the `orderPartNumberKey`, not `compressedOrder`.

13.2.6 Enforcing Referential Integrity

Enforcing referential integrity is one easy step beyond creating a key, which was portrayed in the previous section. The `keyref` element specifies a target node set and a corresponding key sequence in the same manner as the `unique` and `key` elements. In addition, the `keyref` element specifies a reference identity constraint, which is the name of a `unique` or `key` identity constraint. The values for each key sequence specified by the `keyref` element must "equal" the corresponding set of values in any key sequence specified by the reference key.

Tip

An XML validator can test the values of a set of elements or attributes identified by a `keyref` element against a set of presumably corresponding values specified by either a `key` identity constraint or a `unique` identity constraint.

unique - no need to be exist on xml instance
key - must be exist in xml instance.

this the difference between these two.

13.3 The `unique` Element

The `unique` identity constraint assures that each key sequence—the set of values specified by the XPath expressions of the nested `field` elements—is unique. A key sequence is unique if no two key sequences have "equal" values for all keys. Unlike the `key` element, the key sequence does not have to exist for each node in the target node set. The `unique` identity constraint is appropriate, for example, for optional elements or attributes.

Note

The uniqueness of the key sequence specified by the set of `field` elements is relative to the element specified by the `selector` element.

The uniqueness of the keys is dependent on the selector. In the example for this chapter, an identity constraint confines an order to have no more than one part number. However, the XML instance that contains a set of orders

might have multiple part numbers; different customers might order the same part.

Listing 13.4 is the XML representation of the `orderShippedUnique` identity constraint. Each order (a node in the target node set) must have a unique shipping ID (the key sequence). Because this is a `unique` element, the order is not required to have a shipping ID.

since it is optional

Listing 13.4 Example of `unique`

```
<xsd:unique name="orderShippedUnique"
            id="orderShippedUnique.unique">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The customerID uniquely identifies
            each customer.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="order"/>
    <xsd:field xpath="@shipmentID"/>
</xsd:unique>
```

unique, key, keyref must be nested in a element

it cannot be as global one as shown in example

13.3.1 The Attributes of a `unique` Element

The `unique` element has only two attributes: the ubiquitous `id`, and the `name` of the identity constraint. Table 13.2 itemizes the attributes of a `unique` element.

Table 13.2. Attribute Summary for a `unique` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`name`	The value of the `name` attribute uniquely identifies an identity constraint. A `keyref` may optionally reference this name to enforce referential integrity against the set of keys specified by this identity constraint.

13.3.1.1 The `id` Attribute of a `unique` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

unique: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 13.3 is a unique element that has an id attribute.

13.3.1.2 The name Attribute of a unique Element

The value of a name uniquely identifies an identity constraint. A keyref may optionally reference this name to enforce referential integrity against the set of keys specified by this identity constraint.

Attribute Overview

unique: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	Yes.

Listing 13.3 is a unique element that has a name attribute.

13.3.2 Content Options for a unique Element

All of the identity constraints have the same attributes. Other than the annotation, the attributes locate a set of nodes. Table 13.3 itemizes these attributes.

Table 13.3. Content Options for a unique Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.
selector	The value of the selector attribute is an abbreviated XPath that determines the target node set in a corresponding XML instance.

field	The value of the field attribute is an abbreviated XPath that determines the key sequence for each node in the target node set.

The content pattern for the unique element is:

```
annotation? selector field+
```

13.4 The key Element

The key identity constraint assures that each key sequence—the set of values specified by the XPath expressions of the nested field elements—is unique. A key sequence is unique if no two key sequences have "equal" values for all keys. Unlike the unique element, a key sequence *must* exist for each node in the target node set. The key identity constraint is appropriate, for example, for required elements or attributes.

Note

yes. i tested.

target node must be in xml instance document (for key tag. but no need to there for unique tag)

The uniqueness of the key sequence specified by the set of field elements is relative to the element specified by the selector element.

The uniqueness of the keys is dependent on the selector. In the example for this chapter, an identity constraint confines an order to no more than one part number. However, the XML instance that contains a set of orders might have multiple part numbers; different customers might order the same part.

Listing 13.5 is the XML representation of the orderKey identity constraint. Each order (a node in the target node set) must have a unique order ID (the key sequence). Because this is a key element, the order is required to have an order ID.

Listing 13.5 Example of key

```
<xsd:key name="orderKey"
         id="orderKey.key">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The customerID uniquely identifies
            each customer.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="order"
                  id="orderKey.key.selector"/>
    <xsd:field xpath="@orderID"
               id="orderKey.key.field"/>
</xsd:key>
```

13.4.1 The Attributes of a key Element

The key element has only two attributes: the ubiquitous id, and the name of the identity constraint. Table 13.4 itemizes the attributes of a key element.

Table 13.4. Attribute Summary for a key Element

Attribute	*Description*
id	The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
name	Uniquely identifies the identity constraint. A keyref may optionally reference this name to enforce referential integrity against the set of keys specified by this identity constraint.

13.4.1.1 The id Attribute of a key Element

The value of an id attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

key: id

Value:	An ID.
Default:	None.
Constraints:	An id must be unique within an XML schema.
Required:	No.

Listing 13.5 is a key element that has an id attribute.

13.4.1.2 The name Attribute of a key Element

The value of a name uniquely identifies an identity constraint. A keyref may optionally reference this name to enforce referential integrity against the set of keys specified by this identity constraint.

Attribute Overview

key: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	Yes.

Listing 13.5 is a `key` element that has a `name` attribute.

13.4.2 Content Options for a `key` Element

All of the identity constraints have the same elements. Other than the `annotation`, the elements locate a set of nodes. Table 13.5 itemizes these elements.

The content pattern for the `key` element is:

```
annotation? selector field+
```

Table 13.5. Content Options for a `key` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`selector`	The value of the `selector` attribute is an abbreviated XPath that determines the target node set in a corresponding XML instance.
`field`	The value of the `field` attribute is an abbreviated XPath that determines the key sequence for each node in the target node set.

13.5 The `keyref` Element

The `keyref` identity constraint assures that each key sequence specified by the `keyref` exists, by virtue of equality, as a key sequence in the reference identity constraint. The reference identity constraint is always a `unique` or `key` identity constraint. Because the `keyref` identity constraints are slightly more complicated, this section includes two examples. Note that Listing 13.6 is not complicated: It is one key comprised of an element instance value. Listing 13.7 demonstrates the use of two keys. Listing 13.7 also demonstrates mixing key values comprised of element and attribute instance values.

Listing 13.6 portrays both the `partNumberKey` and a corresponding `partNumberRef`. The `partNumberKey` enforces unique part numbers in a catalog. The `partNumberRef` ensures that orders contain only part numbers in the catalog.

Listing 13.6 Example of `keyref`

```
<xsd:key name="partNumberKey"
         id="partNumberKey.key">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            The part number uniquely identifies
            each orderable item.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="catalog/*"/>
    <xsd:field xpath="partNumber"/>
</xsd:key>

<xsd:keyref name="partNumberRef"
            id="partNumberRef.keyref"
            refer="partNumberKey">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Each order must refer to a known part number.
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="compressedOrder/order/item"/>
    <xsd:field xpath="partNumber"/>
</xsd:keyref>
```

Listing 13.7 portrays both the `orderItemsKey` and a corresponding `shippedItemsRef`. The `orderItemsKey` specifies a unique customer ID and the total quantity of parts ordered. The `shippedItemsRef` ensures that each customer receives the appropriate number of parts. This example has a two-part key (customer ID and quantity). In addition, this example demonstrates the use of attributes as keys.

Listing 13.7 Multi-part Key Sequences

```
<xsd:key name="orderedItemsKey"
         id="orderedItemsKey.key">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Identify ordered customers and
            shipping quantities.  Note that
            while schema already enforces
            the following constraint, the
            'key' also assures that there *is*
            a customerID and a shippingQuantity
            identified for each order
```

```
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="compressedOrder/order"/>
    <xsd:field xpath="@totalQuantity"/>
    <xsd:field xpath="customerID"/>
</xsd:key>

<xsd:keyref name="shippedItemsRef"
            id="shippedItems.keyref"
            refer="orderedItemsKey">
    <xsd:annotation>
        <xsd:documentation xml:lang="en">
            Ensure a shipped customer/qty
            for each ordered customer/qty
        </xsd:documentation>
    </xsd:annotation>
    <xsd:selector xpath="shippedOrderSummary/shipped"/>
    <xsd:field xpath="@qty"/>
    <xsd:field xpath="@custID"/>

</xsd:keyref>
```

13.5.1 The Attributes of a `keyref` Element

In addition to the `id` and `name` attributes which apply to all identity constraints, the `keyref` element has the `refer` attribute, which points to the reference against which to validate. Table 13.6 itemizes the attributes of a `keyref` element.

Table 13.6. Attribute Summary for a `keyref` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`name`	The value of the `name` attribute uniquely identifies the identity constraint.
`refer`	Each key sequence specified by this identity constraint must equal any key sequence in the target node set specified by the reference identity constraint.

13.5.1.1 The `id` Attribute of a `keyref` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

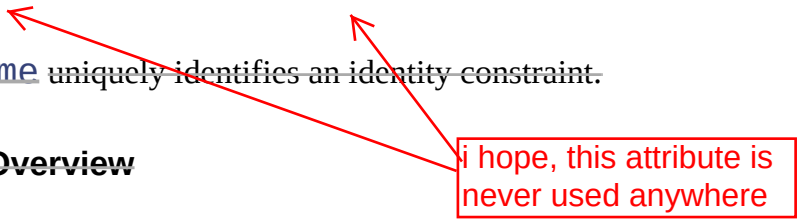
keyref: id

Value:	An ID.
Default:	None.
Constraints:	The value of an `id` must be unique within an XML schema.
Required:	No.

Listing 13.6 portrays a `keyref` element that has an `id` attribute.

13.5.1.2 The name Attribute of a keyref Element

The value of a `name` uniquely identifies an identity constraint.



Attribute Overview

keyref: name

Value:	An NCName.
Default:	None.
Constraints:	None.
Required:	Yes.

Listing 13.6 portrays a `keyref` element that has a `name` attribute.

13.5.1.3 The refer Attribute of a keyref Element

The value of a `refer` attribute is the name of a `unique` or `key` identity constraint. During XML validation, each key sequence specified by the `keyref` element must equal a key sequence specified by the reference identity constraint.

Attribute Overview

keyref: refer

Value:	A QName.
Default:	None.

Constraints:	The value of the `refer` attribute must be the name of a `unique` or `key` identity constraint. Additionally, the number of `field` elements in the reference identity constraint must match the number of fields in the enclosing `keyref`.
Required:	Yes.

Listing 13.6 portrays a `keyref` element that has a `refer` attribute.

13.5.2 Content Options for a `keyref` Element

All of the identity constraints have the same attributes. Other than the `annotation`, the attributes locate a set of nodes. Table 13.7 itemizes these attributes.

Table 13.7. Content Options for a `keyref` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.
`selector`	The value of the `selector` attribute is an abbreviated XPath that determines the target node set in a corresponding XML instance.
`field`	The value of the `field` attribute is an abbreviated XPath that determines the key sequence for each node in the target node set.

The content pattern for the `keyref` element is:

```
annotation? selector field+
```

13.6 The `selector` Element

The value of the `xpath` attribute of a `selector` element specifies a target node set for any type of identity constraint. See any listing in this chapter for an example of the `selector` element.

13.6.1 The Attributes of a `selector` Element

In addition to the ubiquitous `id` attribute, the `selector` element must specify an `xpath`, which selects a set of nodes. Table 13.8 itemizes the attributes of a `selector` element.

Table 13.8. Attribute Summary for a `selector` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`xpath`	The value of the `xpath` attribute determines the target node set of a corresponding XML instance.

13.6.1.1 The `id` Attribute of a `selector` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`selector: id`

Value:	An ID.
Default:	None.
Constraints:	The value of an `id` must be unique within an XML schema.
Required:	No.

Listing 13.5 portrays a `selector` element that has an `id` attribute.

13.6.1.2 The `xpath` Attribute of a `selector` Element

The value of the `xpath` attribute determines the target node set of a corresponding XML instance. Although there are no specific constraints on the value of the `xpath` attribute, the appropriate element type containment should mirror the XPath.

Attribute Overview

`selector: xpath`

Value:	An appropriate subset of an XPath expression.
Default:	None.
Constraints:	None.
Required:	Yes.

All listings in this chapter portray `selector` elements that have an `xpath` attribute.

13.6.2 Content Options for a `selector` Element

Table 13.9 shows that an `annotation` is the only content option available for the `selector` element.

Table 13.9. Content Options for a `selector` Element

Element	*Description*
`annotation`	The `annotation` element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the `selector` element is:

```
annotation?
```

13.7 The `field` Element

The value of the `xpath` attribute of a `field` element specifies a key for any type of identity constraint. In an XML instance, a key sequence is the set of keys that correspond to the set of fields for an identity constraint. See any listing in this chapter for an example of the `field` element.

13.7.1 The Attributes of a `field` Element

The `field` element has the same attributes as the `selector` element. Table 13.10 itemizes these attributes.

Table 13.10. Attribute Summary for a `field` Element

Attribute	*Description*
`id`	The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.
`xpath`	The value of the `xpath` attribute determines one key value from the target node set of a corresponding XML instance. The set of `xpath` attributes corresponding to the set of `field` elements associated with an identity constraint determines the entire key sequence.

13.7.1.1 The `id` Attribute of a `field` Element

The value of an `id` attribute uniquely identifies an element within the set of schema documents that comprise an XML schema.

Attribute Overview

`field: id`

Value:	An ID.
Default:	None.
Constraints:	The value of an `id` must be unique within an XML schema.
Required:	No.

Listing 13.5 portrays a `field` element that has an `id` attribute.

13.7.1.2 The `xpath` Attribute of a `field` Element

The value of the `xpath` attribute determines a specific key. The key may be one of many that comprise a key sequence. Although there are no specific constraints on the value of the `xpath` attribute, the appropriate element type (or attribute type) containment should mirror the XPath.

Attribute Overview

`field: xpath`

Value:	An appropriate subset of an XPath expression.
Default:	None.
Constraints:	None.
Required:	Yes.

All listings in this chapter portray `field` elements that have an `xpath` attribute.

13.7.2 Content Options for a `field` Element

Table 13.11 shows that an `annotation` is the only content option available for the `selector` element.

Table 13.11. Content Options for a `field` Element

Element	*Description*
annotation	The annotation element, discussed in Section 7.5, provides a way to document schema elements.

The content pattern for the field element is:

annotation?

CONTENTS