XML Schemas

Problems with DTDs

- Written in a language other than XML; so need a separate parser.
- All definitions in a DTD are global, applying to the entire document. Cannot have two elements with the same name but with different content in separate contexts.
- The text content of an element can be PCDATA only; no finer typing for numbers or special string formats.
- Limited typing for attribute values.
- DTDs are not truly aware of namespaces; they recognize prefixes but not the underlying URI.

Form of an XML Schema Definition

An XML Schema *is* an XML document.

First item: xml declaration

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

XML comments and processing instructions are allowed.

Root element: *schema* with a namespace declaration.

```
<xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema">
        <!--  schema rules go here  -->
</xs:schema>
```

Possible namespace prefixes: *xs*, *xsd*, or none.

Element Specification

Elements are declared using an element named *xs:element* with an attribute that gives the name of the element being defined.

The type of the content of the new element can be specified by another attribute or by the content of the *xs:element* definition.

Element declarations can be one of two sorts.

Simple Type

Content of these elements can be text only.

Examples

```
<xs:element name="item" type="xs:string"/>
<xs:element name="price" type="xs:decimal"/>
```

The values *xs:string* and *xs:decimal* are two of the 44 simple types predefined in the XML Schema language.

Complex Type

Element content can contain other elements or the element can have attributes (or both).

Example

```
<xs:element name="location">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="city" type="xs:string"/>
      <xs:element name="state" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The element *xs:sequence* is one of several ways to combine elements in the content.

Corresponding DTD: `<!ELEMENT location (city, state)>`

Other *xs:element* Attributes

An *xs:element* element may also have attributes that specify the number of occurrences of the element at this position in the sequence.

```
minOccurs="0"             // default = 1

maxOccurs="5"             // default = maximum(1, minOccurs)
maxOccurs="unbounded"
```

Example: Translate phone.dtd into an XSD

File: phone.dtd

```
<!ELEMENT phoneNumbers (title, entries)>
<!ELEMENT title (#PCDATA)>
<!ELEMENT entries (entry*)>
<!ELEMENT entry (name, phone, city?)>
<!ELEMENT name (first, middle?, last)>
<!ELEMENT first (#PCDATA)>
<!ELEMENT middle (#PCDATA)>
<!ELEMENT last (#PCDATA)>
<!ELEMENT phone (#PCDATA)>
<!ELEMENT city (#PCDATA)>
```

We use *xs:string* in place of PCDATA.

Element sequencing is handled with *xs:sequence*.

The attributes *minOccurs* and *maxOccurs* take care of the * and ? in the DTD.

File: phone.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="phoneNumbers">
  <xs:complexType>
   <xs:sequence>
    <xs:element name="title" type="xs:string"/>
    <xs:element name="entries">
     <xs:complexType>
      <xs:sequence>
       <xs:element name="entry" minOccurs="0"
                            maxOccurs="unbounded">
        <xs:complexType>
         <xs:sequence>
          <xs:element name="name">
           <xs:complexType>
            <xs:sequence>
             <xs:element name="first" type="xs:string"/>
             <xs:element name="middle" type="xs:string"
                                   minOccurs="0"/>
             <xs:element name="last" type="xs:string"/>
            </xs:sequence>
           </xs:complexType>
          </xs:element>
          <xs:element name="phone" type="xs:string"/>
          <xs:element name="city" type="xs:string"
                                   minOccurs="0"/>
         </xs:sequence>
        </xs:complexType>
       </xs:element>
      </xs:sequence>
     </xs:complexType>
    </xs:element>
   </xs:sequence>
  </xs:complexType>
 </xs:element>
</xs:schema>
```

Notes on phone.xsd

Each element of a complex type is followed immediately by the definition of that type in the content of its *xs:element* element.

For example

```
<xs:element name="name">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string"
                                  minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

The type of the *name* element is a complex type without a name; it is an *anonymous* type.

Writing XML Schema following this strategy of using anonymous types leads to very deep indentation.

Alternative Strategy: Named Types

Define the complex types in the XML Schema definition and give them each a name. These definitions will lie at the top level of the *schema* element.

The scope of each complex type definition covers the entire *schema* element (order is of no consequence).

Naming Convention: Use the suffix "Type" when defining a new type in the XML Schema definition.

File: phoneT.xsd

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

  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string" minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="entryType">
    <xs:sequence>
      <xs:element name="name" type="nameType"/>
      <xs:element name="phone" type="xs:string"/>
      <xs:element name="city" type="xs:string" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="entriesType">
    <xs:sequence>
      <xs:element name="entry" type="entryType"
              minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>

  <xs:complexType name="phoneType">
    <xs:sequence>
      <xs:element name="title" type="xs:string"/>
      <xs:element name="entries" type="entriesType"/>
    </xs:sequence>
  </xs:complexType>

  <xs:element name="phoneNumbers" type="phoneType"/>

</xs:schema>
```

Validation

Many tools are available to validate an XML document against a XML Schema specification.

Most common XML parsers can be configured to perform the validation as a document is parsed.

Our first example uses the XML parser in the command-line tool *xmllint*.

To validate *phone2.xml* with the XML Schema in the file *phone.xsd*, enter:

```
% xmllint --schema phone.xsd phone2.xml
```

If the document is valid, it is parsed and printed, followed by a message "phone2.xml validates".

If you do not want to see the XML document printed, enter:

```
% xmllint --noout --schema phoneT.xsd phone2.xml
```

If invalid, all errors are reported, but the document is still parsed if it is well-formed.

Example: bp.xml

Remove a *last* element and a *phone* element from *phone2.xml*. In addition, provide a duplicate *first* element.

```
% xmllint --schema phone.xsd bp.xml
```

After the parsed XML document we get:

```
   bp.xml:6: element name: Schemas validity error : Element
      'name' [CT local]: The element content is not valid.
   bp.xml:21: element name: Schemas validity error : Element
      'name' [CT local]: The element content is not valid.
   bp.xml:29: element entry: Schemas validity error : Element
      'entry' [CT local]: The element content is not valid.
   bp.xml fails to validate
```

Combining Elements in Complex Types

We have already seen that a sequence of elements can be specified using *xs:sequence* inside a complex type.

Elements may be combined in other ways in an XML Schema.

Alternation: Use *xs:choice*

```
<xs:element name="color">
   <xs:complexType>
      <xs:choice>
         <xs:element name="rgb" type="xs:string"/>
         <xs:element name="hsv" type="xs:string:/>
         <xs:element name="cymk" type="xs:string:/>
      </xs:choice>
   </xs:complexType>
</xs:element>
```

The elements may be further modified using the attributes *minOccurs* and *maxOccurs*.

In our example we have the default value of "1" for all three choices.

Ignoring Order: Use *xs:all*

```
<xs:element name="rgbColor">
   <xs:complexType>
      <xs:all>
         <xs:element name="red" type="xs:unsignedbyte"/>
         <xs:element name="green" type="xs:unsignedbyte"/>
         <xs:element name="blue" type="xs:unsignedbyte"/>
      </xs:all>
   </xs:complexType>
</xs:element>
```

Any XML document will validate with this specification as long as the content of the element *rgbColor* contains exactly one occurrence of each of the three elements *red*, *green*, and *blue* in any order.

The *xs:all* element may not be placed inside other grouping elements and may not contain grouping of any kind.

The attribute *maxOccurs* and *minOccurs* may be attached to *sequence*, *choice*, and *all* definitions as well as element definitions.

For elements inside of an *xs:all* element, *minOccurs* and *maxOccurs* attributes can be no larger than "1".

Example: Using Element Combinations

Remember the DTD *elems.dtd*:

```
<!ELEMENT root (one+, (two | three)+,
                    four*, (five*, six)+, (one | two)?)>
<!ELEMENT one (EMPTY)>
<!ELEMENT two (EMPTY)>
<!ELEMENT three (EMPTY)>
<!ELEMENT four (EMPTY)>
<!ELEMENT five (EMPTY)>
<!ELEMENT six (EMPTY)>
```

In an XML Schema we specify an empty element by defining a complex type element with no content.

```
<complexType></complexType>
            or
      <complexType/>
```

In the XML Schema on the next page observe how *xs:sequence*, *xs:choice*, *minOccurs*, and *maxOccurs* are organized to mimic the DTD.

File: elems.xsd

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

  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="one" maxOccurs="unbounded">
          <xs:complexType/>
        </xs:element>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="two">
            <xs:complexType/>
          </xs:element>
          <xs:element name="three">
            <xs:complexType/>
          </xs:element>
        </xs:choice>
        <xs:element name="four" minOccurs="0"
                                  maxOccurs="unbounded">
          <xs:complexType/>
        </xs:element>
        <xs:sequence maxOccurs="unbounded">
          <xs:element name="five" minOccurs="0"
                                  maxOccurs="unbounded">
            <xs:complexType/>
          </xs:element>
          <xs:element name="six">
            <xs:complexType/>
          </xs:element>
        </xs:sequence>
        <xs:choice minOccurs="0">
          <xs:element name="one">
            <xs:complexType/>
          </xs:element>
          <xs:element name="two">
            <xs:complexType/>
          </xs:element>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

This XML schema contains a lot of redundancy.

By using a complex type to stand for the type of an empty element, we can reduce the clutter.

File: ele.xsd

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

  <xs:complexType name="emptyType"/>

  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="one" type="emptyType"
                              maxOccurs="unbounded"/>
        <xs:choice maxOccurs="unbounded">
          <xs:element name="two" type="emptyType"/>
          <xs:element name="three" type="emptyType"/>
        </xs:choice>
        <xs:element name="four" type="emptyType"
                    minOccurs="0" maxOccurs="unbounded"/>
        <xs:sequence maxOccurs="unbounded">
          <xs:element name="five" type="emptyType"
                  minOccurs="0" maxOccurs="unbounded"/>
          <xs:element name="six" type="emptyType"/>
        </xs:sequence>
        <xs:choice minOccurs="0">
          <xs:element name="one" type="emptyType" />
          <xs:element name="two" type="emptyType"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

The XML Schema still contains redundancy since multiple instances of the elements *one* and *two* are defined with the same types.

Referencing Elements

Elements that are defined at the top level inside of the *xs:schema* element are visible throughout the document and can be referenced inside any element definition by using the *ref* attribute in place of the *name* attribute.

This referencing mechanism can be used to reduce the kind of redundancy we have just seen, and in addition it provides a way to organize an XML Schema definition so that it is easier to read.

References can be made to simple type elements as well as complex type elements.

The example on the next page shows the use of references.

File: eleref.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:complexType name="emptyType"/>
  <xs:element name="one" type="emptyType"/>
  <xs:element name="two" type="emptyType"/>
  <xs:element name="three" type="emptyType"/>
  <xs:element name="four" type="emptyType"/>
  <xs:element name="five" type="emptyType"/>
  <xs:element name="six" type="emptyType"/>

  <xs:element name="root">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="one" maxOccurs="unbounded"/>
        <xs:choice maxOccurs="unbounded">
          <xs:element ref="two"/>
          <xs:element ref="three"/>
        </xs:choice>
        <xs:element ref="four" minOccurs="0"
                               maxOccurs="unbounded"/>
        <xs:sequence maxOccurs="unbounded">
          <xs:element ref="five" minOccurs="0"
                               maxOccurs="unbounded"/>
          <xs:element ref="six"/>
        </xs:sequence>
        <xs:choice minOccurs="0">
          <xs:element ref="one"/>
          <xs:element ref="two"/>
        </xs:choice>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Grouping Elements

References can be made to blocks of XSD code as well as to element definitions.

First we collect a section of well-formed code using the XML Schema element *xs:group* with an attribute that gives the group a name.

Then at positions in the definition where we wish to insert the code, we provide an *xs:group* element with a *ref* attribute that specifies the group definition.

File: products.xsd

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:group name="productGroup">
    <xs:sequence>
     <xs:choice>
        <xs:element name="productCode" type="xs:string"/>
        <xs:element name="stockNum" type="xs:string"/>
     </xs:choice>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:group>
          :
  <xs:complexType name="exportType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
     <xs:group ref="productGroup"/>
    </xs:sequence>
  </xs:complexType>
  <xs:complexType name="localGoodsType">
    <xs:sequence>
      <xs:element name="name" type="xs:string"/>
     <xs:group ref="productGroup"/>
    </xs:sequence>
  </xs:complexType>
          :
</xs:schema>
```

Summary of Strategies

We have three ways to organize an XML Schema definition.

- Define the type of each element (other than those with a predefined XSD type) using an anonymous type defined as the content of the element definition.
- Define a series of named complex types (later simple types also) at the top level of the XML Schema document and use those names to indicate the types to be used for the elements.
- Define a series of elements and groups of code at the top level of the Schema definition and then refer to those element definitions using the attribute *ref* when specifying the descendants of the root node of the XML document type being defined.

Many XML Schema definitions use a combination of these three techniques.

Mixed Content

Mixed content refers to the situation where an element has both text and elements in its content.

Because of the sub-elements, the element being defined must be a complex type.

To allow mixed content in an element definition, simply add an attribute *mixed* to the *xs:complexType* starting tag that asserts:

```
mixed="true"
```

This attribute has a default value of "false".

Example Fragment

DTD Specification

```
<!ELEMENT narrative (#PCDATA | bold | italics | underline)*>
<!ELEMENT bold (#PCDATA)>
<!ELEMENT italics (#PCDATA)>
<!ELEMENT underline (#PCDATA)>
```

XML Schema Specification

```
<xs:element name="narrative">
  <xs:complexType mixed="true">
    <xs:choice minOccurs="0" maxOccurrs="unbounded">
      <xs:element name="bold" type="xs:string"/>
      <xs:element name="italics" type="xs:string/>
      <xs:element name="underline" type="xs:string/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

The following element can be validated relative to the previous XSD fragment.

```
<narrative>
    Higher beings from <italics>outer space</italics> may
    not want to tell us the <underline>secrets of life
    </underline>because we're not ready. But maybe they'll
    change their tune after a little <bold>torture</bold>.
    <italics>Jack Handey</italics>
</narrative>
```

Attribute Specifications

Attributes are defined using the element *xs:attribute* with its own attributes, *name* and *type*.

The *xs:attribute* element must lie inside a complex type specification, but the type of the new attribute (its value) must be a simple type. Attribute values may not contain elements or other attributes.

Example: Specification of phoneA.xml

Add the *gender* attribute to the *name* element in the phone example.

In phone.xsd

```
<xs:element name="name">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string"
                                      minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="gender" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

In phoneT.xsd

```
<xs:complexType name="nameType">
  <xs:sequence>
    <xs:element name="first" type="xs:string"/>
    <xs:element name="middle" type="xs:string" minOccurs="0"/>
    <xs:element name="last" type="xs:string"/>
  </xs:sequence>
  <xs:attribute name="gender" type="xs:string"/>
</xs:complexType>
```

Later we show how the type of the *gender* attribute can be restricted to the values "male" or "female".

The attribute specification(s) must lie inside a complex type, but they must fall after the structure definition (*xs:sequence*, *xs:all*, or *xs:choice*) in the complex type.

Problem: What if the complex type has no structure definition?

Attributes with Empty Elements

These specifications are easy.

The only content inside the *xs:complexType* element will be the attribute definitions. This complex type definition can be an anonymous type or a named type.

```
<xs:complexType>
   <xs:attribute name="src" type="xs:anyURI"/>
   <xs:attribute name="width" type="xs:integer"/>
   <xs:attribute name="height" type="xs:integer"/>
</xs:complexType>
```

Attributes for Elements with Only Text Content

These specifications are a bit more complicated.

We need a complex type to allow for attribute definitions, but we need a simple type to specify the text content.

The solution is to place an *xs:simpleContent* element inside of the complex type and use its *xs:extension* element to specify the simple type.

```
<xs:element name="bookTitle">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="author" type="xs:string"/>
        <xs:attribute name="isbn" type="xs:string"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
<xs:element>
```

Other Attributes for *xs:attribute*

This table summarizes most of the possible attributes allowed on the element *xs:attribute*.

Attribute	Possible Attribute Values
name	Name of the attribute
type	Type of the value of the attribute
use	"required" "optional" (the default) "prohibited" (rarely used)
default	Default value for the attribute, which must conform to the type of the attribute value (*use* must be "optional").
fixed	Fixed value for the attribute, which must conform to the type of the attribute value.
ref	Name of an attribute defined globally; used in place of *name*.

Because the previous attribute definitions in these notes omitted the *use* attribute on *xs:attribute*, these attributes were optional in the XML documents that are instances of the XML Schemas.

Example: Attribute Definitions

```
<xs:attribute name="idNum" type="xs:integer"
                               use="required"/>
<xs:attribute name="security" type="xs:string"
                               default="high priority"/>
<xs:attribute name="likesXML" type="xs:string"
                                   fixed="yes"/>
<xs:attribute name="code" type="xs:integer"
                       use="optional" default="0"/>
<xs:attribute ref="security"/>

<xs:attribute ref="code" default="100"/>
```

Other Attributes for *xs:element*

In addition to *name*, *type*, *minOccurs*, and *maxOccurs*, *xs:element* can have the attribute *default* or *fixed*, but not both.

In both cases, if the element occurs in an XML document as an empty element, the *default* or *fixed* value is inserted as the content of the element.

If an element with a *fixed* value has content in an XML document, that content must match the *fixed* value.

```
<xs:element name="happy" type="xs:string"
                                  fixed="yes"/>

<xs:element name="hungry" type="xs:string"
                                  default="unknown"/>
```

A general entity

```
<xs:element name="myEmail" type="xs:string"
                        fixed="slonnegr@cs.uiowa.edu"/>
```

Now use <myEmail/> in an XML document as a kind of entity reference.

Derived Complex Types

The *xs:complexContent* element is used within an *xs:complexType* element to indicate that the content will be a derivation of another complex type.

The derived type can be a restriction of the existing type or an extension of the existing type. If you want to do both, restrict first and extend second.

Restrictions

To restrict an existing complex type, first replicate the original definitions of elements and attributes, and then tighten relevant constraints in the copied model.

Kinds of Restrictions

- Change *minOccurs* or *maxOccurs* to be more restrictive.
- Omit an element (by setting maxOccurs="0") or an attribute (by setting use="prohibited") when the item was previously optional.
- Set a default value of an element or an attribute that previously had none.
- Set a fixed value for an element or attribute that previously had none.

Observe that any instance that conforms to the restricted schema will also conform to the original base model.

Example: Restriction

This XML Schema allows a sequence of *part* items followed by a sequence of *rpart* items.

Each *part* and *rpart* item consists of a sequence of elements, *partNum*, *partName*, *description*, *source*, and *use*, as well as two attributes.

The definition of the *rpart* elements has restrictions on the elements and attributes of the *part* elements.

partName may must occur exactly one time.

description may not occur at all.

source may occur at most once and has a default value.

use may occur at most twice.

date attribute may not occur at all.

File: restrict.xsd

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

<xs:element name="parts">
   <xs:complexType>
     <xs:sequence>
       <xs:element name="part" type="partType"
                                         maxOccurs="5"/>
       <xs:element name="rpart" type="restrictedPartType"
                                         maxOccurs="5"/>
     </xs:sequence>
   </xs:complexType>
 </xs:element>
```

```
  <xs:complexType name="partType">
    <xs:sequence>
      <xs:element name="partNum" type="xs:string"/>
      <xs:element name="partName" type="xs:string"
                                  minOccurs="0" maxOccurs="3"/>
      <xs:element name="description" type="xs:string"
                                               minOccurs="0"/>
      <xs:element name="source" type="xs:string" minOccurs="0"
                                  maxOccurs="unbounded"/>
      <xs:element name="use" type="xs:string" minOccurs="0"
                                  maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="productCode" type="xs:string"/>
    <xs:attribute name="date" type="xs:date"/>
  </xs:complexType>

  <xs:complexType name="restrictedPartType">
    <xs:complexContent>
      <xs:restriction base="partType">
        <xs:sequence>
          <xs:element name="partNum" type="xs:string"/>
          <xs:element name="partName" type="xs:string"
                             minOccurs="1" maxOccurs="1"/>
          <xs:element name="description" type="xs:string"
                             minOccurs="0" maxOccurs="0"/>
          <xs:element name="source" type="xs:string"
                             minOccurs="0" maxOccurs="1"
                                          default="Hills"/>
          <xs:element name="use" type="xs:string"
                             minOccurs="0" maxOccurs="2"/>
        </xs:sequence>
        <xs:attribute name="productCode" type="xs:string"/>
        <xs:attribute name="date" type="xs:date"
                                          use="prohibited"/>
      </xs:restriction>
    </xs:complexContent>
  </xs:complexType>
</xs:schema>
```

File: restrict.xml

```
<?xml version="1.0"?>
<parts>
  <part productCode="p454">
    <partNum>pq1245</partNum>
    <partName>Widget</partName>
    <description>A strange thing</description>
    <source>Hills</source>
    <source>Lone Tree</source>
    <use>extraction</use>
  </part>
  <part productCode="p184" date="2005-02-23">
    <partNum>ts8765</partNum>
    <partName>Dohickey</partName>
    <partName>Dingbat</partName>
    <description>A stranger thing</description>
    <source>Tiffin</source>
    <source>Iowa City</source>
    <source>Coralville</source>
    <use>production</use>
    <use>exclusion</use>
  </part>
  <rpart>
    <partNum>ak9255</partNum>
    <partName>Thingamabobber</partName>
    <source>Coralville</source>
    <use>deletion</use>
  </rpart>
  <rpart productCode="p095">
    <partNum>do7752</partNum>
    <partName>Whatsit</partName>
    <source>Iowa City</source>
  </rpart>
  <rpart productCode="p885">
    <partNum>yy4396</partNum>
    <partName>Wingding</partName>
    <use>insertion</use>
    <use>accumulation</use>
  </rpart>
</parts>
```

Validation

```
% xmllint --noout --schema restrict.xsd restrict.xml
restrict.xml validates
```

The first time I tried this validation, it returned incorrect error messages, which indicated that *xmllint* did not understand attributes in a restricted complex type.

Since then a new version of *xmllint* must have been installed because it works now.

Alternate: Schema Validator

Go to the web page:

www.xmlforasp.net/SchemaValidator.aspx

At this site:

1. Copy and paste an XML document in one text area.
2. Copy and paste an XML Schema definition in another text area.
3. Press the Validate button.

For the two files *restrict.xsd* and *restrict.xml*, the Schema Validator returns:

```
Validation of XML was SUCCESSFUL!
```

Extensions

An extension of a complex type always takes the form of new components appended to the end of the existing model.

An implied sequence element encloses both models so as to enforce the rule that all models have a single topmost grouping construct.

Neither the original data type nor the new one may include the *xs:all* element because it must always be at the top of a content model.

Both parts must agree on allowing mixed content.

File: expand.xsd

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

  <xs:complexType name="partType">
      <!-- Same as in restrict.xsd -->
  </xs:complexType>

  <xs:complexType name="expandedPartType">
    <xs:complexContent>
      <xs:extension base="partType">
        <xs:sequence>
          <xs:element name="color" type="xs:string"/>
          <xs:element name="supplier" type="xs:string"
                             minOccurs="0" maxOccurs="5"/>
        </xs:sequence>
        <xs:attribute name="newAtt" type="xs:string/>
      </xs:extension>
    </xs:complexContent>
  </xs:complexType>
  <xs:element name="parts">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="part" type="partType"
                                                maxOccurs="5"/>
        <xs:element name="epart" type="expandedPartType"
                                                maxOccurs="5"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

File: expand.xml

```
<?xml version="1.0"?>
<parts>
    <part productCode="p454">
      <partNum>pq1245</partNum>
      <partName>Widget</partName>
      <description>A strange thing</description>
      <source>Hills</source>
      <source>Lone Tree</source>
      <use>extraction</use>
    </part>
    <epart newAtt="att value">
      <partNum>ak9255</partNum>
      <partName>Thingamabobber</partName>
      <partName>Dingbat</partName>
      <description>A stranger thing</description>
      <source>Tiffin</source>
      <source>Iowa City</source>
      <source>Coralville</source>
      <use>deletion</use>
      <color>puce</color>
    </epart>
```

```
  <epart productCode="p095" date="2005-01-01">
    <partNum>do7752</partNum>
    <partName>Whatsit</partName>
    <source>Iowa City</source>
    <color>taupe</color>
    <supplier>Smith Brothers</supplier>
    <supplier>Crate and Barrel</supplier>
  </epart>
</parts>
```

Both *xmllint* and Schema Validator report a successful validation with these two files.

Simple Types

XML Schema Definitions provide a rich set of predefined primitive types along with a mechanism to customize these types to create an accurate specification of XML documents.

The predefined types can be classified into several groups.

- Numeric
- Date and time
- XML types
- String
- Boolean
- URIs
- Binary data

Numeric Types

The number types are listed in the table below, showing the range of values for each of the integer types.

Type	Range of Value
xs:byte	-128 to 127
xs:short	-32,768 to 32,767
xs:int	-2,147,483,648 to 2,147,483,647
xs:long	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
xs:unsignedByte	0 to 255
xs:unsignedShort	0 to 65,535
xs:unsignedInt	0 to 4,294,967,295
xs:unsignedLong	0 to 18,446,744,073,709,551,615
xs:integer	-∞ to +∞
xs:positiveInteger	1 to +∞
xs:negativeInteger	-∞ to -1
xs:nonNegativeInteger	0 to +∞
xs:nonPositiveInteger	-∞ to 0
xs:decimal	Optional decimal point with ≤18 digits
xs:float	Standard 32-bit floating-point
xs:double	Standard 64-bit floating-point

Date, Time, and Duration Types

These data types are described by giving the patterns their values follow. Uppercase letters, hyphens, and colons are required. Lowercase letters stand for decimal digits or numerals as appropriate.

Type	Value Format
xs:date	ccyy-mm-dd
xs:time	hh:mm:ss.ssss
xs:datetime	ccyy-mm-ddThh:mm:ss.s
xs:gYear	ccyy
xs:gYearMonth	ccyy-mm
xs:gMonth	--mm
xs:gMonthDay	--mm-dd
xs:gDay	---dd
xs:duration	PnYnMnDTnHnMn.nS

The time types allow a specification of time zone (Z or -6:00).

Any of the year, month, day, hours, minutes, and seconds of the *xs:duration* type may be omitted, but not all of them.

Example durations

P2Y5M27DT8H35M

P88D

PT17M6.75S

XML Types

To maintain backward compatibility with DTDs, XML Schemas allow the following XML types.

Type	Description
xs:Name	A string that represents a valid XML identifier
xs:NCName	A string that represents a local name with no colons
xs:QName	A string that represents a qualified name (optional prefix)
xs:NMTOKEN	Same as DTD NMTOKEN
xs:NMTOKENS	A list of NMTOKEN names
xs:ID	Same as xs:NCName
xs:IDREF	Same as xs:NCName
xs:IDREFS	A list of IDREF names
xs:ENTITY	A name declared in a DTD
xs:ENTITIES	A list of ENTITY names
xs:language	A recognized language identifier
xs:NOTATION	"Something" identified by an identifier

String Types

In addition to the basic string type that corresponds to PCDATA, XML Schemas allow two other types that instruct the XML parser to modify the white space in the string.

Type	Description
xs:string	Any string of characters
xs:normalizedString	A string that the parser will normalize, replacing each white space character by a space
xs:token	A string that the parser will normalize and collapse, changing each sequence of white space into a single space and trimming leading and trailing spaces

Miscellaneous Types

Here we have the boolean type, a URI type, and two binary types.

Type	Sample Values
xs:boolean	false, true, 0, 1
xs:anyURI	http://www.cs.uiowa.edu/ myDirectory/files/info
xs:hexBinary	87B3EA93C5
xs:base64Binary	jdU7+3hfu/Sm

Base 64 Encoding

A 65-character subset of ascii is used, enabling 6 bits to be represented per printable character.

The encoding process translates 24-bit groups of input bits into output strings of four encoded characters.

Proceeding from left to right, a 24-bit input group is formed by concatenating three 8-bit input groups. These 24 bits are then treated as four 6-bit groups, each of which is translated into a single character in the base 64 alphabet.

Each 6-bit group can be used as an index into an array of 64 printable characters. The character referenced by the index is placed in the output string.

Base 64 Alphabet

Value	Code	Value	Code	Value	Code	Value	Code
0	A	17	R	34	i	51	z
1	B	18	S	35	j	52	0
2	C	19	T	36	k	53	1
3	D	20	U	37	l	54	2
4	E	21	V	38	m	55	3
5	F	22	W	39	n	56	4
6	G	23	X	40	o	57	5
7	H	24	Y	41	p	58	6
8	I	25	Z	42	q	59	7
9	J	26	a	43	r	60	8
10	K	27	b	44	s	61	9
11	L	28	c	45	t	62	+
12	M	29	d	46	u	63	/
13	N	30	e	47	v		
14	O	31	f	48	w	(pad)	=
15	P	32	g	49	x		
16	Q	33	h	50	y		

Special processing is performed if fewer than 24 bits are available at the end of the data being encoded.

A fully encoded chunk of four characters is always completed at the end of a translation.

When fewer than 24 input bits are available in an input group, zero bits are added (on the right) to form a whole number of 6-bit groups.

Padding at the end of the data is performed using the '=' character.

Since all base 64 input is an integral number of bytes, only the following cases can arise:

1) If the final chunk of input to be encoded is exactly 24 bits, the final unit of encoded output will four characters with no "=" padding.
2) If the final piece of input is exactly 8 bits, the final unit of encoded output will be two characters followed by two "=" padding characters.
3) If the final chunk of input is exactly 16 bits, the final unit of encoded output will be three characters followed by one "=" padding character.

Simple Type Restrictions

The power of predefined types can be enhanced by defining new types (named or anonymous) that derive from the simple types by restricting their values in one way or another.

A restriction has two basic aspects,

- Naming the base type, an existing simple type.
- Describing the restrictions using facets.

Facets

Facets are XML Schema elements whose *value* attribute defines the restriction indicated by the facet.

XML Schema has twelve facets for restricting simple types.

Facet	Applicable To
xs:minInclusive	numbers and dates
xs:maxInclusive	numbers and dates
xs:minExclusive	numbers and dates
xs:maxExclusive	numbers and dates
xs:totalDigits	numbers only
xs:fractionDigits	numbers only
xs:length	string types
xs:minLength	string types
xs:maxLength	string types
xs:enumeraton	most types
xs:pattern	most types
xs:whitespace	most types

Restrictions on Range

A named type for ages

```
<xs:simpleType name="ageType">
   <xs:restriction base="xs:integer">
      <xs:minInclusive value="0"/>
      <xs:maxInclusive value="120"/>
   </xs:restriction>
</xs:simpleType>
```

An anonymous type for an element

```
<xs:element name="temperature">
   <xs:simpleType>
      <xs:restriction base="xs:integer">
         <xs:minInclusive value="-40"/>
         <xs:maxInclusive value="130"/>
      </xs:restriction>
   </xs:simpleType>
</xs:element>
```

A type for afternoon

```
<xs:simpleType name="pmType">
   <xs:restriction base="xs:time">
      <xs:minInclusive value="12:00:00"/>
      <xs:maxInclusive value="23:59:59"/>
   </xs:restriction>
</xs:simpleType>
```

Restrictions on Digits

A currency type

```
<xs:simpleType name="currencyType">
   <xs:restriction base="xs:decimal">
      <xs:fractionDigits value="2"/>
   </xs:restriction>
</xs:simpleType>
```

A counting type

The *totalDigits* facet specifies the maximum number of digits.

```
<xs:simpleType name="countingType">
   <xs:restriction base="xs:nonNegativeInteger">
      <xs:totalDigits value="6"/>
   </xs:restriction>
</xs:simpleType>
```

Restrictions on Length

A password type

```
<xs:simpleType name="passwordType">
   <xs:restriction base="xs:string">
      <xs:minLength value="8"/>
      <xs:maxLength value="12"/>
   </xs:restriction>
</xs:simpleType>
```

A zipcode type

```
<xs:simpleType name="zipType">
   <xs:restriction base="xs:string">
      <xs:length value="5"/>
   </xs:restriction>
</xs:simpleType>
```

Restrictions to a Set of Values

A city type

```
<xs:simpleType name="cityType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="Iowa City"/>
      <xs:enumeration value="Coralville"/>
      <xs:enumeration value="North Liberty"/>
      <xs:enumeration value="Hills"/>
      <xs:enumeration value="Solon"/>
   </xs:restriction>
</xs:simpleType>
```

A gender type

```
<xs:simpleType name="genderType">
   <xs:restriction base="xs:string">
      <xs:enumeration value="male"/>
      <xs:enumeration value="female"/>
   </xs:restriction>
</xs:simpleType>
```

A year type

```
<xs:simpleType name="yearType">
   <xs:restriction base="xs:gYear">
      <xs:enumeration value="2002"/>
      <xs:enumeration value="2003"/>
      <xs:enumeration value="2004"/>
      <xs:enumeration value="2005"/>
      <xs:enumeration value="2006"/>
   </xs:restriction>
</xs:simpleType>
```

Restrictions Using a Pattern

A TLA (Three Letter Acronym) type

```
<xs:simpleType name="tlaType">
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z][A-Z][A-Z]"/>
   </xs:restriction>
</xs:simpleType>
```

Patterns

A pattern is a string from the language of regular expressions that defines a set of strings that are said to match the pattern.

Basic Regular Expressions

Let "a" represent an arbitrary symbol chosen from the underlying alphabet and let E and F stand for arbitrary regular expressions

Expression	**Matches**
a	a
E | F	E or F
EF	A string matching E followed by a string matching F
E^*	Zero or more occurrences of strings that match E
E+	One or more occurrences of strings that match E
E?	A string that matches E or an empty string

These operations follow precedence rules with the unary operators having the strongest precedence and alternation the weakest.

Parentheses may be used to override precedence.

Regular Expression Extensions

Construct	Matches
Character classes (match one character only)	
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (negation)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p: [a-dm-p] (union)
[a-z&&[def]]	d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^k-p]]	a through z, and not k through p: [a-jq-z] (subtraction)
Quantifiers	
E{n}	E, exactly n times
E{n,}	E, at least n times
E{n,m}	E, at least n but not more than m times

Predefined character classes

.	Any character except new line
\d	A digit: [0-9]
\D	A non-digit: [^0-9]
\s	A white space character: [\t\n\v\f\r]
\S	[^\s]
\i	First character in an XML identifier
\I	[^\i]
\c	Any character legal in an NMTOKEN
\C	[^\c]
\w	A "word" character (not punctuation or separator)
\W	[^\w]

Examples

A phone number type

```
<xs:simpleType name="phoneType">
   <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}-\d{4}"/>
   </xs:restriction>
</xs:simpleType>
```

A gender type

```
<xs:simpleType name="genderType">
   <xs:restriction base="xs:string">
      <xs:pattern value="male|female"/>
   </xs:restriction>
</xs:simpleType>
```

or

```
<xs:simpleType name="genderType">
   <xs:restriction base="xs:string">
      <xs:pattern value="male"/>
      <xs:pattern value="female"/>
   </xs:restriction>
</xs:simpleType>
```

A zipcode type

```
<xs:simpleType name="zipType">
   <xs:restriction base="xs:string">
      <xs:pattern value="[0-9]{5}-[0-9]{4}"/>
   </xs:restriction>
</xs:simpleType>
```

An auto license type

```
<xs:simpleType name="licenseType">
   <xs:restriction base="xs:string">
      <xs:pattern value="[A-Z]{3} \d{3}"/>
   </xs:restriction>
</xs:simpleType>
```

Restrictions on White Space

These restrictions provide information to the parser about how to handle white space characters.

```
<xs:simpleType>
   <xs:restriction base="xs:string">
      <xs:whitespace value="preserve"/>
   </xs:restriction>
</xs:simpleType>
```

The *value* attribute for the *xs:whitespace* element may take one of three values.

preserve	the default
replace	change each white space character into a space
collapse	change each sequence of white space into a single space and trim the string

Lists of Items

A simple type can be defined as a list of items of some other simple type.

The element *xs:list* has an attribute *itemType* for specifying the kind of components in the list.

Example: A List of Exam Scores

```
<xs:simpleType name="scoreListType">
   <xs:list itemType="xs:nonNegativeInteger"/>
</xs:simpleType>

<xs:element name="scores" type="scoreListType"/>
```

XML element

```
    <scores>82 96 74 68 80</scores>
```

Lists of *xs:string* items are dangerous since a string may have spaces, and spaces are used to delimit the list.

A list type can be restricted using *xs:length*, *xs:maxLength*, *xs:minLength*, or *xs:emuneration*.

```
  <xs:simpleType name="shortScoreListType">
    <xs:restriction base=" scoreListType ">
      <xs:maxLength value="3"/>
    </xs:restriction>
  </xs:simpleType>
```

Note that these simple types being defined, including list types, can be used as an element type, as an attribute type, or can be created as a named type.

Unions

A simple type can be defined as the (disjoint) union of two existing simple types.

The element *xs:union* has an attribute *memberTypes* whose value is a space-separated list of simple types that have already been defined.

Example

Suppose we want to store exam scores, but in some instances, the grade may not be available.

1. Define a type representing a missing score.

```
<xs:simpleType name="noScoreType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="none"/>
  </xs:restriction>
</xs:simpleType>
```

2. Define a union type of integer scores and missing scores.

```
<xs:simpleType name="scoreOrNoType">
  <xs:union memberTypes ="xs:integer noScoreType"/>
</xs:simpleType>
```

3. Define a list of the union type.

```
<xs:simpleType name="scoreOrNoList">
  <xs:list itemType ="scoreOrNoType "/>
</xs:simpleType>
```

4. Define a type whose values can be a list of scores (or none) or can be a date on which we can expect the grades to be made available.

```
<xs:simpleType name="scoresOrDateType">
  <xs:union memberTypes ="xs:date scoreOrNoList"/>
</xs:simpleType>
```

XML Schema Validation

We want to use a Java XML parser to validate an XML document relative to an XML Schema definition.

First we need to tie the XML Schema document to the XML document similar to what we did with external DTDs.

The testing will be done using a version of the phone XML document.

Note that the XML Schema definition is specified using an attribute *xsi:noNamespaceSchemaLocation*, which requires the definition of a namespace for the prefix *xsi*, which is performed by the attribute definition

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

This XML Schema definition uses many of the techniques that we have covered in this chapter.

File: phoneX.xml

```
<?xml version="1.0"? standalone="no">
<phoneNumbers
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="phoneX.xsd">
  <title>Phone Numbers</title>
  <entries>
   <entry>
     <name>
       <first>Rusty</first>
       <last>Nail</last>
     </name>
      <phone areaCode="319">335-0055</phone>
     <city>Iowa City</city>
   </entry>
```

```
      <entry>
        <name gender="male">
          <first>Justin</first>
          <last>Case</last>
        </name>
        <phone>354-9876</phone>
        <city>Coralville</city>
      </entry>

      <entry>
        <name gender="female">
          <first>Pearl</first>
          <middle>E.</middle>
          <last>Gates</last>
        </name>
        <phone areaCode="319">335-4582</phone>
        <city>North Liberty</city>
      </entry>

      <entry>
        <name gender="female">
          <first>Helen</first>
          <last>Back</last>
        </name>
        <phone>337-5967</phone>
      </entry>
    </entries>
  </phoneNumbers>
```

File: phoneX.xsd

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

  <xs:simpleType name="cityType">
    <xs:restriction base="xs:string">
      <xs:enumeration value="Iowa City"/>
      <xs:enumeration value="Coralville"/>
      <xs:enumeration value="North Liberty"/>
      <xs:enumeration value="Hills"/>
      <xs:enumeration value="Solon"/>
    </xs:restriction>
  </xs:simpleType>

  <xs:complexType name="nameType">
    <xs:sequence>
      <xs:element name="first" type="xs:string"/>
      <xs:element name="middle" type="xs:string"
                                          minOccurs="0"/>
      <xs:element name="last" type="xs:string"/>
    </xs:sequence>
    <xs:attribute name="gender">
      <xs:simpleType>
        <xs:restriction base="xs:string">
          <xs:pattern value="male|female"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
  </xs:complexType>

  <xs:simpleType name="phoneType">
    <xs:restriction base="xs:string">
      <xs:pattern value="\d{3}-\d{4}"/>
    </xs:restriction>
  </xs:simpleType>
  <xs:complexType name="entryType">
    <xs:sequence>
      <xs:element name="name" type="nameType"/>
      <xs:element name="phone">
        <xs:complexType>
          <xs:simpleContent>
            <xs:extension base="phoneType">
              <xs:attribute name="areaCode">
                <xs:simpleType>
                  <xs:restriction base="xs:string">
                    <xs:pattern value="\d{3}"/>
                  </xs:restriction>
                </xs:simpleType>
              </xs:attribute>
            </xs:extension>
          </xs:simpleContent>
        </xs:complexType>
      </xs:element>
      <xs:element name="city" type="cityType"
                                        minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
<xs:element name="phoneNumbers">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="title" type="xs:string"/>
        <xs:element name="entries">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="entry" type="entryType"
                      minOccurs="0" maxOccurs="unbounded"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Validating with SAX

A SAX (Simple API for XML) parser can be configured to perform XML Schema validation.

After we have obtained a SAXParser object from the factory, we set a property for the parser that indicates that the schema method to be used is the XML Schema.

But first we must turn on the validation and namespace awareness in the factory. Namespaces are need because of the "http://www.w3.org/2001/XMLSchema-instance" namespace.

Because of changes made between SAX1 and SAX2, the SAXParser is used to create an XMLReader object that does the actual parsing.

When something goes wrong, the errors are reported by the methods in an instance of MyErrorHandler, which is the same class we used in DOM.

File: SaxCheck.java

```
import javax.xml.parsers.SAXParserFactory;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.ParserConfigurationException;
import org.xml.sax.XMLReader;
import org.xml.sax.SAXException;
import java.io.IOException;

public class SaxCheck
{
  static public void main(String [] args)
  {
    SAXParserFactory factory =
                    SAXParserFactory.newInstance();
    factory.setValidating(true);
    factory.setNamespaceAware(true);
```

```
        try
        {  SAXParser saxParser = factory.newSAXParser();

           saxParser.setProperty(
      "http://java.sun.com/xml/jaxp/properties/schemaLanguage",
      "http://www.w3.org/2001/XMLSchema");

           XMLReader xmlReader = saxParser.getXMLReader();
           xmlReader.setErrorHandler(new MyErrorHandler());
           xmlReader.parse("" + new File(args[0]).toURL());
           System.out.println("The XML document is valid.");
        }
        catch (ParserConfigurationException e)
        {   System.out.println("Parser configuration error");   }
        catch (SAXException e)
        {   System.out.println("Parsing error.");   }
        catch (IOException e)
        {   System.out.println("IO error.");   }
      }
   }
```

Execution

```
% java SaxCheck phoneX.xml
The XML document is valid.
```

Now change the first *areaCode* attribute value to "xyz" and try again.

```
% java SaxCheck phoneB.xml
Error:
cvc-pattern-valid: Value 'xyz' is not facet-valid with respect
                              to pattern '\d{3}' for type 'null'.
  Line 11
  Column 31
  Document file:///mnt/nfs/fileserv/fs3/slonnegr/SaxCheck/phoneB.xml

Parsing error.
```

Type Hierarchy

```
anyType
   l— all complex types
   l— anySimpleType
            l— duration
            l— dateTime
            l— time
            l— date
            l— gYearMonth
            l— gYear
            l— gMonthDay
            l— gDay
            l— gMonth
            l— boolean
            l— base64Binary
            l— hexBinary
            l— float
            l— double
            l— anyURI
            l— QName
            l— NOTATION
            l— decimal
            l      l— integer
            l             l— nonPositiveInteger
            l             l      l— negativeInteger
            l             l— long
            l             l      l— int
            l             l            l— short
            l             l                  l— byte
            l             l— nonNegativeInteger
            l                    l— positiveInteger
            l— string
                   l— normalizedString
                          l— token
                                 l— language
                                 l— NMTOKEN
                                 l— Name
                                        l-- NCName
                                               l-- ID
                                               l-- IDREF
                                               l-- ENTITY
```