Chapter 4: XML Schema

References:

- Meike Klettke, Holger Meyer: XML & Datenbanken. Abschnitt 5.1, 7.2.2 dpunkt.verlag, 2003, ISBN 3-89864-148-1.
- Harald Schöning, Walter Waterfeld: XML Schema. In: Erhard Rahm, Gottfried Vossen: Web & Datenbanken, Seiten 33-64. dpunkt.verlag, 2003, ISBN 3-89864-189-9.
- Elliotte Rusty Harold, W. Scott Means: XML in a Nutshell, A Desktop Quick Ref., 3rd Ed. O'Reilly, Okt. 2004, ISBN 0-596-00764-7, 689 Seiten, 37 Euro.
- Priscilla Walmsley: Definitive XML Schema. Prentice Hall, 2001, ISBN 0130655678, 560 pages.
- W3C Architecture Domain: XML Schema. [http://www.w3.org/XML/Schema]
- David C. Fallside, Priscilla Walmsley: XML Schema Part 0: Primer. W3C, 28. October 2004, Second Edition. [http://www.w3.org/TR/xmlschema-0/]
- Henry S. Thompson, David Beech, Murray Maloney, Noah Mendelsohn: XML Schema Part 1: Structures. W3C, 28. October 2004, Second Edition [http://www.w3.org/TR/xmlschema-1/]
- Paul V. Biron, Ashok Malhotra: XML Schema Part 2: Datatypes. W3C, 28. October 2004, Second Edition [http://www.w3.org/TR/xmlschema-2/]
- Matthias Hansch, Stefan Kulins, Martin Schrader: Aktuelles Schlagwort: XML Schema. In: Informatik Spektrum, Oktober 2002, 363–366. [http://www.wifo.uni-mannheim.de/xml-schema/]
- [http://www.w3schools.com/schema/]

Objectives

After completing this chapter, you should be able to:

- explain why DTDs are not sufficient for many applications.
- explain some XML schema concepts.
- write an XML schema.
- check given XML documents for validity according to a given XML schema.

Overview

Introduction (1)

Problems of DTDs:

- The type system is very restricted.

  E.g. one cannot specify that an element or an attribute must contain a number.

- Concepts like keys and foreign keys (known from the relational data model) cannot be specified.

  The scope of `ID` and `IDREF` attributes is global to the entire document. Furthermore, the syntax restrictions for `ID`s are quite severe.

- A DTD is not itself an XML document (i.e. it does not use the XML syntax for data).
- No support for namespaces.

Introduction (2)

- DTDs were probably sufficient for the needs of the document processing community, but do not satisfy the expectations of the database community.
- Therefore, a new way of describing the application-dependent syntax of an XML document was developed: XML Schema.
- In XML Schema, one can specify all syntax restrictions that can be specified in DTDs, and more (i.e. XML Schema is more expressive).

  Only entities cannot be defined in XML Schema.

Introduction (3)

- The W3C began work on XML Schema in 1998.
- XML Schema 1.0 was published as a W3C standard ("recommendation") on May 2, 2001.

  A second edition appeared October 28, 2004.
- XML Schema 1.1 became a W3C recommendation on April 5, 2012.
- The Standard consists of:
  - ◇ Part 0: Tutorial introduction (non-normative).
  - ◇ Part 1: Structures.
  - ◇ Part 2: Datatypes.

Introduction (4)

- A disadvantage of XML schema is that it is very complex, and XML schemas are quite long (much longer than the corresponding DTD).
- Quite a number of competitors were developed.

  E.g. XDR, SOX, Schematron, Relax NG. See: D. Lee, W. Chu: Comparative Analysis of Six XML Schema Languages. In ACM SIGMOD Record, Vol. 29, Nr. 3, Sept. 2000.
- Relax NG is a relatively well-known alternative.

  See: J. Clark, M. Makoto: RELAX NG Specification, OASIS Committee Specification, 3 Dec. 2001.
  [http://www.oasis-open.org/committees/relax-ng/spec-20011203.html]

Introduction (5)

Comparison with DBMS:

- In a (relational) DBMS, data cannot be stored without a schema.
- An XML document is self-describing: It can exist and can be processed without a schema.
- In part, the role of a schema in XML is more like integrity constraints in a relational DB.

  It helps to detect input errors. Programs become simpler if they do not have to handle the most general case.
- But in any case, programs must use knowledge about the names of at least certain elements.

Example Document (1)

STUDENTS

SID	FIRST	LAST	EMAIL
101	Ann	Smith	···
102	Michael	Jones	(null)
103	Richard	Turner	···
104	Maria	Brown	···

EXERCISES

CAT	ENO	TOPIC	MAXPT
H	1	Rel. Algeb.	10
H	2	SQL	10
M	1	SQL	14

RESULTS

SID	CAT	ENO	POINTS
101	H	1	10
101	H	2	8
101	M	1	12
102	H	1	9
102	H	2	9
102	M	1	10
103	H	1	5
103	M	1	7

Example Document (2)

- Translation to XML with data values in elements:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>101</SID>
      <FIRST>Ann</FIRST>
      <LAST>Smith</LAST>
    </STUDENT>
    ...
  </STUDENTS>
  ...
</GRADES-DB>
```

Example: First Schema (1)

- Part 1/4:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="GRADES-DB">
    <xs:complexType>
      <xs:sequence>
        <xs:element ref="STUDENTS"/>
        <xs:element ref="EXERCISES"/>
        <xs:element ref="RESULTS"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

Example: First Schema (2)

- Part 2/4:

```
<xs:element name="STUDENTS">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="STUDENT"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example: First Schema (3)

- Part 3/4:

```
<xs:element name="STUDENT">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="SID"/>
      <xs:element ref="FIRST"/>
      <xs:element ref="LAST"/>
      <xs:element ref="EMAIL" minOccurs="0"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example: First Schema (4)

- Part 4/4:

```
    <xs:element name="SID">
      <xs:simpleType>
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="100"/>
          <xs:maxInclusive value="999"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:element>
    <xs:element name="FIRST" type="xs:string"/>
    <xs:element name="LAST" type="xs:string"/>
    <xs:element name="EMAIL" type="xs:string"/>
    ...
  </xs:schema>
```

Example: First Schema (5)

Remarks:

- The prefix used for the namespace is not important. E.g. sometimes one sees "`xsd:`" instead of "`xs:`".
- A complex type is a type that contains elements and/or attributes.
- A simple type is something like a string or number.

  A simple type can be used as the type of an attribute, and as the data type of an element (content and attributes). A complex type can only be the data type of an element (attributes cannot contain elements or have themselves attributes). Instead of "element", I should really say "element type", but that might be confusing (it is not an XML Schema type).

Example: First Schema (6)

- In XML Schema, the sequence of declarations (and definitions, see below) is not important.

  The example contains many references to element types that are declared later. Actually, a schema can contain references to elements that are not declared at all, as long as these elements do not occur in the document, i.e. they are not needed for validation. Some validators even in this case print no error message: They use "lax validation" and check only for what they have declarations.

- It is necessary to use a one-element sequence (or choice) in the declaration of `STUDENTS`.

  One cannot use `xs:element` directly inside `xs:complexType`. This is similar to the content model in DTDs, which always needs "`(...)`".

Example: First Schema (7)

- The default for `minOccurs` and `maxOccurs` is `1`.
- In XML Schema, one cannot define what must be the root element type. E.g., a document consisting only of a `STUDENT`-element would validate.

  Every "globally" declared element type can be used. Global declarations are declarations that appear directly below `xs:schema`. As explained below, it is often possible to declare only the intended root element type globally, then there is no problem. Otherwise the application must check the root element type. Note that DTDs also do not define the root element type, this happens only in the `DOCTYPE`-declaration.

Validation (1)

Online Validators:

- Freeformatter

  [http://www.freeformatter.com/xml-validator-xsd.html]

- Decisisonsoft

  [http://tools.decisionsoft.com/schemaValidate/]

- XML Validation

  [http://www.xmlvalidation.com/?L=2]

Validation (2)

Validators for Local Installation:

- Altova XML Community Edition

  [http://www.softpedia.com/get/Internet/Other-Internet-Related/AltovaXML.shtml]

- XSV

  [http://www.ltg.ed.ac.uk/ ht/xsv-status.html]

Validation (3)

Validating parser libraries:

- Apache Xerces

  [http://xerces.apache.org/]

- Libxml2

  [http://xmlsoft.org/]

- Oracle XDK

  [http://www.oracle.com/technetwork/developer-tools/xmldevkit/]

- Microsoft MSXML

  [http://msdn2.microsoft.com/en-us/xml/default.aspx]

Validation (4)

- Depending on the validator used, it is not necessary that the XML data file (the instance of the schema) contains a reference to the schema.
- If one wants to refer to the schema, this can be done as follows:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB xmlns:xsi=
      "http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ex2.xsd">
  ...
</GRADES-DB>
```

Schema Styles (1)

- The same restrictions on XML documents can be specified in different ways in XML.

  I.e. there are equivalent, but very differently structured XML schemas.

- The above XML schema is structured very similar to a DTD: All element types are declared with global scope. No named types (see below) are used.

- This style is called "Salami Slice".

  The schema is constructed in small pieces on equal level.

  "'Salami slice' caputes both the disassembly process, the resulting flat look of the schema, and implies reassembly as well (into a sandwich)." [http://www.xfront.com/GlobalVersusLocal.html]

Schema Styles (2)

- One can also nest element declarations.
- Element declarations that are not defined as children of `xs:schema` cannot be referenced.

  They are local declarations in contrast to the global ones used above.
- In this way, one can have elements with the same name, but different content models in different contexts within one document.

  This is impossible with DTDs. It might be useful for complex documents, especially if the schema is composed out of independently developed parts. In relational DBs, different tables can have columns with the same name, but different types. Then the above XML translation of a relational schema cannot be done in "Salami Slice" style.

Schema Styles (3)

- XML Schema in “Russian Doll” style:

```
<xs:element name="GRADES-DB">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="STUDENTS">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="STUDENT"
                minOccurs="0"
                maxOccurs="unbounded">
              <xs:complexType>
                <xs:sequence>
                  <xs:element name="SID">
                        ...
```

Schema Styles (4)

- Advantages of “Russian Doll” style:
  - ◇ The structure of the schema is similar to the structure of the document.
  - ◇ In “Russian Doll” style, there is only one global element, thus the root element type is enforced.
- Disadvantages:
  - ◇ The declaration of equal subelements has to be duplicated.
  - ◇ Recursive element types are not possible.
  - ◇ No reuse of schema components.

Schema Styles (5)

- Actually, in XML schema, one
  - ◇ defines (data) types and
  - ◇ declares elements to have a (data) type.

    A declaration binds names that occur in the XML data file (the instance) to (data) types. A definition introduces names that can be used only in the schema.

- In the above examples, all types are anonymous.
  In "Venetian Blind" design, explicit types are used.

  At least for elements with similar content models. Elements are declared locally as in the "Russian Doll" style.
  "'Venetian Blind' captures the ability to expose or hide namespaces with a simple switch, and the assembly of slats captures reuse of components." [http://www.xfront.com/GlobalVersusLocal.html]

Schema Styles (6)

- XML Schema in “Venetian Blind” style, Part 1/4:

```
<xs:simpleType name="SIDType">
  <xs:restriction base="xs:integer">
    <xs:minInclusive value="100"/>
    <xs:maxInclusive value="999"/>
  </xs:restriction>
</xs:simpleType>
<!-- Continued on next three slides -->
```

Schema Styles (7)

- "Venetian Blind" Style, Part 2/4:

```
<xs:complexType name="StudentType">
  <xs:sequence>
    <xs:element name="SID" type="SIDType"/>
    <xs:element name="FIRST" type="xs:string"/>
    <xs:element name="LAST" type="xs:string"/>
    <xs:element name="EMAIL" type="xs:string"
        minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

Schema Styles (8)

- "Venetian Blind" Style, Part 3/4:

```
<xs:complexType name="StType">
  <xs:sequence>
    <xs:element name="STUDENT" type="studentType"
        minOccurs="0" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
```

Schema Styles (9)

- "Venetian Blind" Style, Part 4/4:

```
<xs:complexType name="GradesType">
  <xs:sequence>
    <xs:element name="STUDENTS" type="StType"/>
    <xs:element name="EXERCISES" type="ExType"/>
    <xs:element name="RESULTS" type="ResType"/>
  </xs:sequence>
</xs:complexType>

<xs:element name="GRADES-DB" type="GradesType">
```

Schema Styles (10)

- Remarks about “Venetian Blind” style:
  - ◇ There is only one global element declaration, thus the root element type is enforced.

    All other elements are known only locally within their type.
  - ◇ Probably, this is often the best style.

    The content model (and attributes) of equal subelements is specified only once (in the corresponding type). The components (types) are resuable. The reusability is even better than in the “Salami Slice” style, because the (data) types can be used with different element (type) names.
  - ◇ It is possible to define types and elements with the same name.

Schema Styles (11)

Summary:

Style	Element Decl.	Type Decl.
Salami Slice	Global	Anonymous, local (except predefined simple types)
Russian Doll	Local (except root)	Anonymous, local (except predefined simple types)
Venetian Blind	Local (except root)	Named, global

Example with Attributes (1)

- Document:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENT SID='101' FIRST='Ann' LAST='Smith'/>
  <STUDENT SID='102' FIRST='Michael' LAST='Jones'/>
  ...
  <EXERCISE CAT='H' ENO='1' TOPIC='Rel. Algeb.'/>
  ...
  <RESULT SID='101' CAT='H' ENO='1' POINTS='10'/>
  ...
</GRADES-DB>
```

Example with Attributes (2)

- Schema, Part 1/3:

```
<xs:element name="GRADES-DB">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="STUDENT"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="EXERCISE"
        minOccurs="0" maxOccurs="unbounded"/>
      <xs:element ref="RESULT"
        minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```

Example with Attributes (3)

- Schema, Part 2/3:

```
<xs:element name="STUDENT">
  <xs:complexType>
    <xs:attribute name="SID" use="required">
      <xs:simpleType>
        <xs:restriction base="xs:integer">
          <xs:minInclusive value="100"/>
          <xs:maxInclusive value="999"/>
        </xs:restriction>
      </xs:simpleType>
    </xs:attribute>
    <!--- declaration continued on next slide -->
```

Example with Attributes (4)

- Schema, Part 3/3:

```
        <xs:attribute name="FIRST"
                use="required"
                type="xs:string"/>
        <xs:attribute name="LAST"
                use="required"
                type="xs:string"/>
        <xs:attribute name="EMAIL"
                type="xs:string"/>
    </xs:complexType>
</xs:element> <!-- STUDENT -->
```

Example with Attributes (5)

- The same (simple) data type can be used for attributes and for element content.

  In contrast, DTDs had some data types for attributes, but basically no data types for element content (only strings) (and of course content models, but that is a separate issue).

- In the example, the elements have empty content (`xs:complexType` contained no content model).
- If an element type has element content and attributes, inside `xs:complexType`, one must specify
  - ◇ first the content model (e.g., with `xs:sequence`)
  - ◇ and then declare the attributes.

Example with Attributes (6)

- Element types with attributes and simple types as content, e.g.

```
<length unit="cm">12</length>
```

can be defined by extension of the simple type:

```
<xs:complexType name="lengthType">
  <xs:simpleContent>
    <xs:extension base="xs:integer">
      <xs:attribute name="unit" type="xs:string">
    </xs:extension>
  </xs:simpleContent>
<xs:complexType>
```

Overview

1. Introduction, Examples
2. Simple Types
3. Complex Types, Elements, Attributes
4. Integrity Constraints
5. Advanced Constructs

Data Types: Introduction (1)

- The second part of the XML schema standard defines
  - ◇ a set of 44 built-in simple types,

    In addition, there are two "ur types": `anyType` and `anySimpleType`.
  - ◇ possibilities for defining new simple types by restriction (similar to `CHECK` constraints in SQL), and the type constructors `union` and `list`.
- Many of the built-in types are not primitive, but defined by restriction of other built-in types.

  19 types are primitive.

Data Types: Introduction (2)

- These definitions were put into a separate standard document because it is possible that other (XML) standards (besides XML schema) might use them in future.
- The requirements for this standard include
  - ◇ It must be possible to represent the primitive types of SQL and Java as XML Schema types.
  - ◇ The type system should be adequate for import/export from database systems (e.g., relational, object-oriented, OLAP).

Data Types: Introduction (3)

- Datatypes are seen as triples consisting of:
  - ◇ a value space (the set of possible values of the type),
  - ◇ a lexical space (the set of constants/literals of the type),

    Every element of the value space has one or more representations in the lexical space (exactly one canonical representation).
  - ◇ a set of "facets", which are properties of the type, distinguished into "fundamental facets" that describe the type (e.g. `ordered`), and "constraining facets" that can be used to restrict the type.

Data Types: Introduction (4)

- The standard does not define data type operations besides equality (=) and order (<, >).

  E.g., the standard does not talk about +, string concatenation, etc. (But Appendix E explains how durations are added to dateTimes.).

- One should define application-specific data types, even if they are equal to a built-in type:
  - ◇ This makes the semantics and comparability of attributes and element contents clearer.
  - ◇ If one later has to change/extend a data type, this is automatically applied to all attributes/ elements that contain values of the type.

Built-in Simple Types (1)

- Strings and Names

  `string`, `normalizedString`, `token`, `Name`, `NCName`, `QName`, `language`

- Numbers

  `float`, `double`, `decimal`, `integer`, `positiveInteger`, `nonPositiveInteger`, `negativeInteger`, `nonNegativeInteger`, `int`, `long`, `short`, `byte`, `unsignedInt`, `unsignedLong`, `unsignedShort`, `unsignedByte`

- Date and Time

  `duration`, `dateTime`, `date`, `time`, `gYear`, `gYearMonth`, `gMonth`, `gMonthDay`, `gDay`

- Boolean

  `boolean`

Built-in Simple Types (2)

- Legacy Types

  `ID`, `IDREF`, `IDREFS`, `ENTITY`, `ENTITIES`, `NMTOKEN`, `NMTOKENS`, `NOTATION`

- Character Encodings for Binary Data

  `hexBinary`, `base64Binary`

- URIs

  `anyURI`

- “Ur-types”

  `anyType`, `anySimpleType`

Facets (1)

Constraining Facets:

- Bounds: `minInclusive`, `maxInclusive`,
  `minExclusive`, `maxExclusive`
- Length: `length`, `minLength`, `maxLength`
- Precision: `totalDigits`, `FractionDigits`
- Enumerated Values: `enumeration`
- Pattern matching: `pattern`
- Whitespace processing: `whiteSpace`

Facets (2)

Fundamental Facets:

- `ordered`: `false`, `partial`, `total`

  The specification defines the order between data type values. Sometimes, values are incomparable, which means that the order relation is a partial order. Some types are not ordered at all.
  Note that every value space supports the notion of equality. The value spaces of all primitive data types are disjoint.

- `bounded`: `true`, `false`
- `cardinality`: `finite`, `countably infinite`
- `numeric`: `true`, `false`

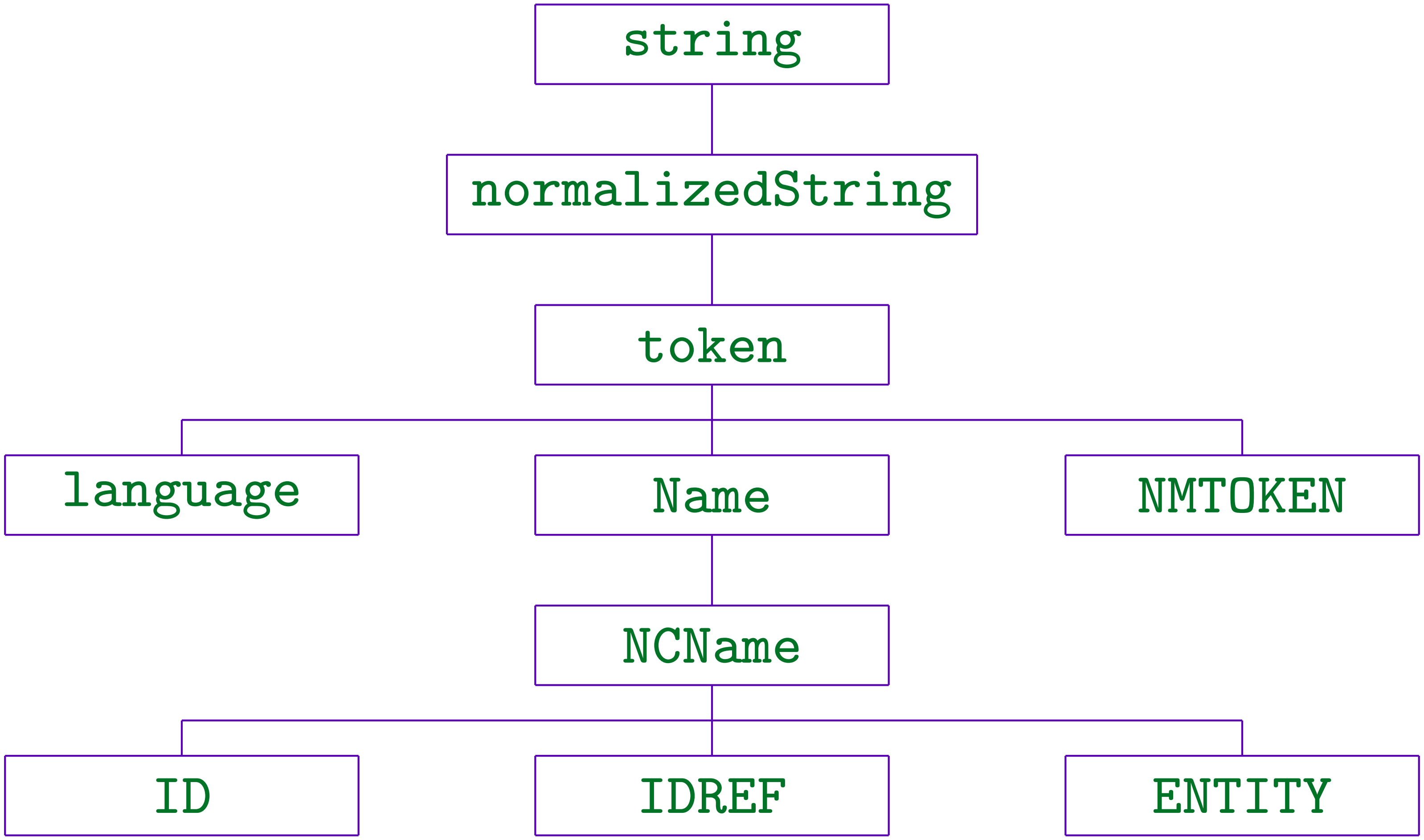
Strings and Names (1)

- A string is a finite-length sequence of characters as defined in the XML standard.

  The XML standard in turn refers to the Unicode standard, and excludes control characters (except tab, carriage return, linefeed), "surrogate blocks", `FFFE`, `FFFF`.

- In XML Schema, `string` values are not ordered.
- The following (constraining) facets can be applied to `string` and its subtypes: `length`, `minLength`, `maxLength`, `pattern`, `enumeration`, `whitespace`.
- The hierarchy of types derived from `string` by restriction is shown on the next slide.

Strings and Names (2)



Strings and Names (3)

- `normalizedString` are strings that do not contain the characters carriage return, line feed, and tab.
- The XML processor will replace line ends and tabs by spaces.

  The combination "carriage return, linefeed" is replaced by a single space. The XML Schema Standard says that even the lexical space does not contain carriage return, linefeed, tab. If I understand correctly, that would mean that they are forbidden in the input. However, the book "Definite XML Schema" states that the processor does this replacement. This seems plausible, because even in the original XML standard, `CDATA` attributes were normalized in this way. By the way, this gives an apparent incompatibility with the original XML standard, when one defines an attribute of type `string`: Does normalization occur anyway, because it is built into XML?

Strings and Names (4)

- `token` is a string without
  - ◇ carriage return, linefeed, tab,
  - ◇ sequences of two or more spaces,
  - ◇ leading or trailing spaces.
- The name "token" is misleading: It is not a single "word symbol", but a sequence of such "tokens".

  Again, I and the book "Definite XML Schema" believe that the XML processor normalizes input strings in this way, whereas the standard seems to say that the external representation must already fulfill the above requirements. In the XML standard, this normalization is required for all attribute types except `CDATA`.

Strings and Names (5)

- normalizedString and token can be derived from string by using the facet whiteSpace, which has three possible values:
  - ◇ preserve: the input is not changed.

    The XML standard requires that any XML processor replaces the sequence "carriage return, linefeed" by a single linefeed.
  - ◇ replace: carriage return, linefeed, and tab are replaced by space.
  - ◇ collapse: Sequences of spaces are reduced to a single one, leading/trailing spaces are removed.

Strings and Names (6)

- `Name`: An XML name.

  I.e. a sequence of characters that starts with a letter, an underscore "`_`", or a colon "`:`", and otherwise contains only letters, digits, and the special characters underscore "`_`", colon "`:`", hyphen "`-`", and period "`.`". Letter means an Unicode letter, not only an ASCII letter (actually, there are also more digits in Unicode than in ASCII).

- `NMTOKEN`: Any sequence of XML name characters.

  This is like `Name`, but without the requirement that it must start with a letter etc. E.g., a sequence of digits would be valid. For compatibility, `NMTOKEN` should be used only for attributes (not element content).

- `NCName`: "Non-colonized name", i.e. like `Name`, but without colon "`:`".

  Important because the colon has a special meaning for namespaces.

Strings and Names (7)

- `ID`: Syntax like `NCName`, but the XML processor enforces uniqueness in the document.

  Actually, the XML Schema standard (Part 2) does not mention the uniqueness requirement, but the book "Definite XML Schema" does mention it (it is probably inherited from the XML standard). As all legacy types, `ID` should be used only for attributes. The XML standard forbids that an element type has two or more attributes of type `ID`. Furthermore, `ID`-attributes cannot have default or fixed values specified.

- `IDREF`: Syntax like `NCName`, value must appear as value of an `ID`-attribute in the document.

Strings and Names (8)

- `ENTITY`: Syntax like `NCName`, must be declared as an unparsed entity in a DTD.

  It is interesting that the XML Schema standard does mention the restriction with the DTD.

- `language`: Language identifier, see RFC 3066.

  E.g. `en`, `en-US`, `de`. These are language identifiers according to the ISO standard ISO 639, optionally with a country code as defined in ISO 3166. However, also the IANA (Internet Assigned Numbers Authority) registers languages, their names start with "`i-`". Unofficial languages start with "`x-`". The pattern given in the XML Schema standard permits an arbitrary number of pieces (at least one), separated by hyphens, each consisting of 1 to 8 letters and digits (the first piece must be only letters).

Strings and Names (9)

- The preceding types are derived from `string` directly or indirectly by restriction.

  With the facets `whiteSpace` and `pattern` (see below).

- However, there are also built-in types that are derived using the type constructor `list`. The result is a space-separated list of values of the base type.
- The following legacy types are defined as lists:
  - ◇ `IDREFS`: list of `IDREF` values.
  - ◇ `NMTOKENS`: list of `NMTOKEN` values.
  - ◇ `ENTITIES`: list of `ENTITY` values.

Strings and Names (10)

- `QName` is the type for qualified names, i.e. names that can contain a namespace prefix.

  The prefix is not required, either because there is a default namespace declaration, or because the name belongs to no namespace.

- `QName` is not derived from `string`, since it is not a simple string, but contains two parts:
  - ◇ The local name, and
  - ◇ the namespace URI.

    Note the distinction between lexical space and value space: The lexical space contains the prefix (like `xs:`), the value space the corresponding URI.

Length Restrictions (1)

- One can define a type by constraining the length (measured in characters) of a string type, e.g.

```
<xs:simpleType name="varchar20">
  <xs:restriction base="xs:string">
    <xs:maxLength value="20"/>
  </xs:restriction>
</xs:simpleType>
```

- There are three length constraining facets:
  - ◇ `maxLength`: String length must be $\leq$ `value`.
  - ◇ `minLength`: String length must be $\geq$ `value`.
  - ◇ `length`: String length must be $=$ `value`.

  Using the length restrictions for `QName` is deprecated.

Length Restrictions (2)

- One can use `minLength` and `maxLength` together, but not together with `length`.
- For example, strings with 3 to 10 characters:

```
<xs:simpleType name="From3To10Chars">
  <xs:restriction base="xs:string">
    <xs:minLength value="3"/>
    <xs:maxLength value="10"/>
  </xs:restriction>
</xs:simpleType>
```

- One cannot specify any of the three facets more than once in the same restriction.

Length Restrictions (3)

- One can further constrain a defined type, but one cannot extend it, e.g. the following is invalid:

```
<xs:simpleType name="varchar40">
  <xs:restriction base="xs:varchar20">
    <xs:maxLength value="40"/> <!-- ERROR -->
  </xs:restriction>
</xs:simpleType>
```

  Actually, one can extend a type, but not in `xs:restriction`. E.g., one can add values with `union` (see below).

- It would, however, be possible to define strings of maximal length 10 in this way.

Enumeration Types

- Example:

```
<xs:simpleType name="weekday">
  <xs:restriction base="xs:token">
    <xs:enumeration value="Sun"/>
    <xs:enumeration value="Mon"/>
    <xs:enumeration value="Tue"/>
    ...
  </xs:restriction>
</xs:simpleType>
```

  By using `xs:token` as base type, leading and trailing white space is accepted and automatically removed.

- If one wants to restrict an enumeration type further, one must again list all possible values.

Regular Expressions (1)

- The facet "`pattern`" can be used to derive a new (restricted) type from the above string types by requiring that the values match a regular expression.

  The facet `pattern` can also be applied to some other types, see below.

- The regular expressions in XML Schema are inspired by the regular expressions in Perl.

  However, XML schema requires that the regular expressions matches the complete string, not only some part inside the string (i.e. there is an implicit `^` at the beginning and `$` at the end: If necessary, use `.*` to allow an arbitrary prefix of suffix).

Regular Expressions (2)

- E.g., a type for product codes that consist of an uppercase letter and four digits (e.g., A1234) could be defined as follows:

```
<xs:simpleType name="prodCode">
  <xs:restriction base="xs:token">
    <xs:pattern value="[A-Z][0-9]{4}"/>
  </xs:restriction>
</xs:simpleType>
```

- One can specify more than one pattern, then it suffices if one of the pattern matches.

Regular Expressions (3)

- A regular expression is composed from zero or more branches, separated by "|" characters.

  As usual, "|" indicates an alternative: The language defined by the regular expression $b_1|\ldots|b_n$ is the union of the languages defined by the branches b_i (see below).

- A branch consists of zero or more pieces, concatenated together.

  The language defined by the regular expression $p_1 \ldots p_n$ consists of all words w that can be constructed by concatenating words w_i of the languages defined by the pieces p_i, i.e. $w = w_1 \ldots w_n$.

Regular Expressions (4)

- A piece consists of an atom and an optional quantifier: `?`, `*`, `+`, `{`n`,`m`}`, `{`n`}`, `{`n`,}`.

  The quantifier permits repetition of the piece, see below. If the quantifier is missing, the language defined by the piece is of course equal to the language defined by the atom. Otherwise, the language defined by the piece consists of all words of the form $w_1 \ldots w_k$, where each w_i is from the language defined by the atom, and k satisfies the requirements of the quantifier (see next slide).

- An atom is
  - ◇ a character (except metacharacters, see below)
  - ◇ a character class (see below),
  - ◇ or a regular expression in parentheses "`(...)`".

Regular Expressions (5)

- Meaning of quantifiers (permitted repetitions k):
  - ◇ (No quantifier): exactly once ($k = 1$).
  - ◇ ?: optional ($k = 0$ or $k = 1$).
  - ◇ *: arbitrarily often (no restriction on k).
  - ◇ +: once or more ($k \geq 1$).
  - ◇ {n,m}: between n and m times ($n \leq k \leq m$).
  - ◇ {n}: exactly n times ($k = n$).
  - ◇ {n,}: at least n times ($k \geq n$).

Regular Expressions (6)

- Metacharacters are characters that have a special meaning in regular expressions. One needs a character class escape (see below) for a regular expression that matches them.

  Metacharacters are: `.`, `\`, `?`, `*`, `+`, `|`, `{`, `}`, `(`, `)`, `[`, `]`.

- Character classes are:
  - ◇ Character class escape: `\...` (see below)
  - ◇ Character class expressions: `[...]` (see below)
  - ◇ The wildcard “.”

    Matches any character except carriage return and newline.

Regular Expressions (7)

- Character class escapes (slide 1/2):
  - ◇ `\`x for every metacharacter x: matches x.
  - ◇ `\n`: newline
  - ◇ `\r`: carriage return
  - ◇ `\t`: tab
  - ◇ `\d`: any decimal digit
  - ◇ `\s`: any whitespace character
  - ◇ `\i`: any character allowed first in XML name

    I.e. a letter, underscore "_", or colon ":".
  - ◇ `\c`: any character allowed inside XML name

Regular Expressions (8)

- Character class escapes (slide 2/2):
  - ◇ `\w`: any character not in categories “punctation”, “separator”, “other” in the Unicode standard.

    In Perl, this is simply an alphanumeric “word character”, i.e. a letter, a digit, or the underscore “_”.
  - ◇ `\p{`x`}`: Any character in Unicode category x.

    E.g.: `\p{L}`: all letters, `\p{Lu}`: all uppercase letters, `\p{Ll}`: all lowercase letters, `\p{Sc}`: all currency symbols, `\p{isBasicLatin}`: all ASCII characters (codes `#x0000` to `#x007F`), `\isCyrillic{Sc}`: all cyrillic characters (codes `#x0400` to `#x04FF`).

    [www.unicode.org/Public/3.1-Update/UnicodeCharacterDatabase-3.1.0.html].
  - ◇ `\D`, `\S`, `\I`, `\C`, `\W`, `\P{`x`}`: complement of corresponding lowercase escape.

Regular Expressions (9)

- A character class expression has one of the forms (slide 1/2):
  - ◇ [$c_1 \ldots c_n$] with characters, character ranges, or character escapes c_i.

    It matches every character matched by one of the c_i, i.e. it is basically an abbreviation for (c_1|...|c_n), where character ranges x-y are replaced by all characters with Unicode value between the Unicode values of x and y. E.g. `[a-d]` is equivalent to `[abcd]`. Because of the special meaning of `-`, it must be the first or last character if it is meant literally. In a character range x-y, one cannot use multi character escapes (character escapes that match more than one character, e.g. `\d`) as x and y.

Regular Expressions (10)

- Character class expressions (slide 2/2):
  - [^$c_1 \ldots c_n$]: complement of [$c_1 \ldots c_n$].

    Because of the special meaning of ^, it must be not the first character if it is meant literally.
  - [$c_1 \ldots c_n$-E]: Characters matched by [$c_1 \ldots c_n$], but not matched by the character class expression E.

    E.g. `[a-z-[aeiou]]` is equivalent to `[bcdfghjklmnpqrstvwxyz]`.
  - [^$c_1 \ldots c_n$-E]: Characters matched by [^$c_1 \ldots c_n$], but not matched by E.

Floating Point Numbers

- `float`: 32-bit floating point type

  It can be represented as $m * 2^e$, where m is an integer whose absolute value is less than 2^{24}, and e is an integer with $-149 \leq e \leq 104$. In addition, it contains the three special values `-INF` (negative infinity), `+INF` (positive infinity), and `NaN` ("not a number": error value). `NaN` is incomparable with all other values. This type is very similar to the one defined in IEEE 754-1985, but has only one zero and one `NaN`. Furthermore, `NaN=NaN` in XML Schema. Constants (literals) are, e.g., `-1E2`, `+1.2e-3`, `1.23`, `-0`.

- `double`: 64-bit floating point type

  It can be represented as $m * 2^e$, where m is an integer whose absolute value is less than 2^{53}, and e is an integer with $-1075 \leq e \leq 970$.

- The two are distinct primitive types.

Fixed Point Numbers (1)

- decimal: primitive type for fixed point numbers.

  Exact numeric types in contrast to float/double, which are approximate numeric types, because the rounding errors are not really forseeable. E.g., one should not use float for amounts of money.

- Value space: numbers of the form $i * 10^{-n}$, where i and n are integers and $n \geq 0$ (e.g. 1.23).
- Lexical space: finite-length sequences of decimal digits with at most one decimal point in the sequence, optionally preceded by a sign (+, -).

  The book "Definitive XML Schema" states that the sequence may start of end with a period (e.g. ".123"), the standard does not clearly specify this.

Fixed Point Numbers (2)

- Leading zeros and trailing zeros after the decimal point are not significant, i.e. `3` and `003.000` are the same decimal number.

  The option `+` sign is also not significant.

- Every XML Schema processor must support at least 18 digits.

  E.g. it could use 64 bit binary integer numbers, plus an indication of where the decimal point is. However, also using strings or a BCD encoding (4 bit per digit) would be possible internal representations.

- All integer types are derived from `decimal` by restriction (see below).

Fixed Point Numbers (3)

- By using the facets `totalDigits` and `fractionDigits`, one can get the SQL data type `NUMERIC(`p`,`s`)`.

  p is the precision (`totalDigits`), s is the scale (`fractionDigits`).

- E.g. `NUMERIC(5,2)` permits numbers with 5 digits in total, of which two are after the decimal point (like `123.45`):

```
<xs:simpleType name="NUMERIC_5_2">
  <xs:restriction base="xs:decimal">
    <xs:totalDigits value="5"/>
    <xs:fractionDigits value="2"/>
  </xs:restriction>
</xs:simpleType>
```

Fixed Point Numbers (4)

- One can specify bounds b for the data value d with the facets
  - ◇ `minExclusive`: $d > b$.

    `b` is the contents of the `value`-Attribute of the `xs:minExclusive` element. The same for the other facets.
  - ◇ `minInclusive`: $d \geq b$.
  - ◇ `maxInclusive`: $d \leq b$.
  - ◇ `maxExclusive`: $d < b$.
- The factes `length`, `minLength`, `maxLength` are not applicable for numeric types.

  If necessary, one can use a pattern.

Fixed Point Numbers (5)

- The facet `whiteSpace` has the fixed value `collapse` for the numeric types: leading and trailing spaces are automatically skipped.

  Because the facet value is fixed, one cannot change this behaviour.

- The facet `pattern` is applicable, e.g. one could exclude or require leading zeros.

  `pattern` applies to the lexical representation of the value.

- The facet `enumeration` is applicable.

  E.g. one could list the valid grades in the German system: `1.0`, `1.3`, `1.7`, `2.0`, ..., `4.3`, `5.0`.

Fixed Point Numbers (6)

- Integer types are derived from `decimal` by setting `fractionDigits` to 0 and selecting the bounds shown on the next slide.
- There are four classes of integer types:
  - ◇ `integer`: no restrictions
  - ◇ `positiveInteger` etc.: restrictions at -1, 0, 1.
  - ◇ `long`, `int`, `short`, `byte`: restriction given by representability in 64, 32, 16, 8 Bit.
  - ◇ `unsigned long` etc.: minimum 0, maximum x Bit.

    Using a sign is invalid for these types, even -0 is not allowed.

Fixed Point Numbers (7)

Type	minInclusive	maxInclusive
integer		
positiveInteger	1	
nonPositiveInteger		0
negativeInteger		-1
nonNegativeInteger	0	
long (64 Bit)	-9223372036854775808	9223372036854775807
int (32 Bit)	-2147483648	2147483647
short (16 Bit)	-32768	32767
byte (8 Bit)	-128	127
unsigned long	0	18446744073709551615
unsigned int	0	4294967295
unsigned short	0	65535
unsigned byte	0	255

Boolean

- The value space consists of the truth values `true`, `false`.
- The lexical space consists of `true`, `false`, `1`, `0`.

  As one would expect, `1` represents the value `true`, and `0` represents the value `false`.

Date and Time Types (1)

- A `dateTime`-value has the form (similar to ISO 8601)

  $$yyyy\texttt{-}mm\texttt{-}dd\texttt{T}hh\texttt{:}mi\texttt{:}ss\texttt{.}xxx\,zzzzzz$$

  where (continued on next slide)

  ◇ $yyyy$ is the year,

  It is possible to use negative years for the time Before Christ ("Before Common Era"), but the meaning might change: Currently, there is no year `0000`, the year before `0001` is `-0001`. This was changed in the corresponding ISO standard, `0000` is now 1 BC. More than four digits are permitted (then leading zeros are disallowed).

  ◇ mm is the month (1 to 12)

  ◇ dd is the day (1 to max. 31, restricted by month)

  E.g., February 30 is impossible, and February 29 only in leap years.

Date and Time Types (2)

- Components of `dateTime` values, continued:
  - ◇ *hh* is the hour (0 to 23)

    The value `24` is permitted if minute and second is `0`, it is the same as `00:00:00` on the following day.
  - ◇ *mi* is the minute (0 to 59)
  - ◇ *ss* is the second (0 to 59)

    When a leap second is inserted, also `23:59:60` is possible. From 1972 to 2005, this has happend 23 times. Note that the seconds part of `dateTime`-values cannot be left out.
  - ◇ *xxx* is an optional fractional part of a second

    It can have arbitrary length, not only milliseconds.
  - ◇ *zzzzzz* is optional timezone information

Date and Time Types (3)

- The suffix `Z`, e.g. `2007-05-14T15:30:00Z` marks a value as UTC ("Universal Coordinated Time").

  This is May 14, 2007, 3:30pm, in Greenwich, UK.

- `2007-05-14T15:30:00Z` is the same as
  - ◇ `2007-05-14T16:30:00+01:00`

    CET: Central European Time, e.g. in Germany ("MEZ")
  - ◇ `2007-05-14T17:30:00+02:00`

    CEDT/CEST: Central European Daylight savings/Summer Time
  - ◇ `2007-05-14T10:30:00-05:00`

    EST: Eastern Standard Time, e.g. New York, Pittsburgh.

Date and Time Types (4)

- If timezone information is not specified, as e.g. in

  `2007-05-14T16:00:00`

  the time is considered to be local time in some (unknown) timezone.

- One should avoid comparing local time and time with timezone information (UTC).

  E.g., `2007-05-14T15:30:00Z` and `2007-05-14T16:00:00` are uncomparable (e.g. in Germany, `2007-05-14T16:00:00` would actually be before `2007-05-14T15:30:00`). If, however, the time difference is greater than 14 hours (maximal zone difference), local time and UTC are comparable. Note that all `dateTime`-values without timezone are considered comparable, i.e. it is assumed that they are all in the same timezone.

Date and Time Types (5)

date:

- Value space: top-open intervals of dateTime-values (they include 00:00:00, but not 24:00:00).

  This means that 2007-05-14+13:00 is actually the same date-value as 2007-05-13-11:00. In general, values are not necessarily printed on output in the same way as they are read from input. This also applies to dateTime values: The actual timezone is lost, values are stored internally as UTC. The application program could know the timezone.

- Lexical space: date-values are written in the form

  yyyy-mm-dd

  with optional timezone information as before.

- E.g.: 2007-05-14 (local time), 2007-05-14+01:00.

Date and Time Types (6)

`gYearMonth`:

- Value space: Intervals of `dateTime`-values from the beginning of the month (inclusive) to the beginning of the next month (exclusive).

  The "g" indicates that this depends on the Gregorian calendar (this is the ususal calender e.g. in Germany and the US). Whereas also `dateTime`-literals are written using the Gregorian Calender, they can easily be converted into other calendars. For year/month combinations, this conversion is usually not possible.

- Lexical space: Constants are written in the form *yyyy-mm*, with optional time zone information.
- E.g.: `2007-05` (local time), `2007-05+01:00`.

Date and Time Types (7)

`gYear`:

- Value space: years (intervals of `dateTime` values corresponding to one year in the Gregorian calendar).
- Lexical space: Constants are written in the form *yyyy*, with optional time zone information.
- E.g. `2007` (local time), `-0001` (local time, 1/2 BC), `2007+01:00`, `2007Z` (UTC).
- `dateTime`, `date`, `gYearMonth`, `gYear` form a hierarchy of larger and larger timeline intervals.

  Actually, `dateTime` values are points on the timeline (zero duration).

Date and Time Types (8)

`time`:

- An instant of time that recurs every day.
- Constants are written in the form *hh*:*mi*:*ss*, with optional fractional seconds and timezone.

  This is simply the suffix of `dateTime` literals after the `T`. This especially means that the seconds cannot be left out (`15:30` is invalid).

- E.g. `15:30:00`, `15:30:00.123+01:00`, `15:30:00Z`.
- `time`-values are ordered, with the usual problem that local time and timezoned time can can be compared only if the difference is large enough.

Date and Time Types (9)

`gDay`:

- A day that recurs every month, e.g. the 15th.

  More precisely, it is a recurring time interval of length one day.

- Lexical representation: ---dd (plus opt. timezone).

`gMonth`:

- A month that recurs every year, e.g. May.
- Lexical representation: --mm (plus opt. timezone).

`gMonthDay`:

- A day that recurs every year, e.g. December 24.
- Lexical representation: --mm-dd (opt. timezone).

Date and Time Types (10)

`duration`:

- A duration of time, consisting of seven components: sign, and number of years, months, days, hours, minutes, seconds.

  Seconds can have a fractional part, the other numbers are integers.

- The constants are written as optional sign, then the letter "`P`", then one or more numbers with unit (`Y`, `M`, `D`, `H`, `M`, `S` — in this order), with the letter `T` used as separator in front of the time-related values.

  E.g. `P2M` is two months, and `PT2M` is two minutes. The letter "`T`" must be written if and only if hours, minutes, or seconds are specified.

Date and Time Types (11)

- Examples:
  - ◇ `P1Y2M3D` is a duration of one year, two months, and three days.
  - ◇ `P2DT12H` is a duration of two days and twelve hours.
  - ◇ `-P1D` is the duration that gives yesterday if added to today's date.
- The values of the components are not restricted by the size of the next larger component, e.g. `PT36H` is possible (36 hours).

Date and Time Types (12)

- duration values are ordered only partially, e.g. P30D and P1M are not comparable.

  P1M is larger than P27D, it is uncomparable to P28D, P29D, P30D, P31D, and it is smaller than P32D. However, P5M not simply multiplies these values by 5, but looks at an arbitrary sequence of five consecutive months. Thus, p5M is larger than P149D, smaller than P154D, and uncomparable to the values in between.

- One can use the pattern facet to enforce that durations are specified in a single unit, P\d+D permits only days.

Date and Time Types (13)

- Applicable constraining facets:
  - ◇ `pattern`
  - ◇ `enumeration`
  - ◇ `minExclusive`, `minInclusive`,
    `maxExclusive`, `maxInclusive`

    As explained above, mixing UTC and local time should be avoided. If a value is not comparable with the bound, it is considered illegal.
  - ◇ `whiteSpace` has the fixed value `collapse`.

Binary Data

- Values of the types `hexBinary` and `base64Binary` are finite-length sequences of bytes ("binary octets").
- The lexical space of `hexBinary` is the set of even-length strings of decimal digits and letters `a-f`/`A-F`.

  Every hexadecimal digit represents 4 bits of the binary string.
- The Base64-encoding packs 6 Bits into every character by using the characters `A-Z`, `a-z`, `0-9`, "`+`", "`/`" (and "`=`" at the end to mark characters derived from fill bytes).

  See RFC 2045. The string length is always a multiple of four (4 characters from the encoding are mapped to 3 bytes of binary data).

URIs

- The value space of the built-in type `anyURI` is the set of (absolute or relative) URIs, optionally with a fragment identifier (i.e., "`#`").

  See RFC 2396 and RFC 2732.

- Some international characters are allowed directly in constants of type `anyURI` that would normally have to be escaped with "%xy".

  See the XLink specification, Section 5.4 "Locator Attribute", and Section 8 "Character Encoding in URI References".

- It is not required that the URI can be dereferenced (accessed).

NOTATION

- One can declare notations (non-XML data formats) in XML schema:

```
<xs:notation name="gif" public=
  "-//IETF//NOTATION Media Type image/gif//EN"/>
```

- Values of the built-in data type `NOTATION` are the qualified names of the declared notations.
- One cannot use this type directly for elements and attributes, but must declare an enumeration:

```
<xs:simpleType name="imageFormat">
  <xs:restriction base="xs:NOTATION">
    <xs:enumeration value="gif"/>
    <xs:enumeration value="jpeg"/>
    ...
```

Union Types (1)

- One can define a new simple type by constructing the union of two or more simple types.

  One can construct the union of a union, but this is equivalent to a "flat" union. One cannot take the union of complex types.

- Example: The attribute `maxOccurs` permits integers (≥ 0) and the special value "`unbounded`" (a string).

- The components of a union type can be specified by the attribute "`memberTypes`" or by `simpleType`-children, or a mixture of both.

  The order of the menber types is insofar significant, as the value will count as a value of the first member type for which it is a legal value.

Union Types (2)

```
<!-- Enumeration type with only value "unbounded" -->
<xs:simpleType name="uType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="unbounded"/>
  </xs:restriction>
</xs:simpleType>

<!-- Defining a union with attribute memberTypes: -->
<xs:simpleType name="cardinality">
  <xs:union memberTypes="nonNegativeInteger uType"/>
</xs:simpleType>
```

Union Types (3)

```
<!-- Defining a union with simpleType children: -->
<xs:simpleType name="cardinality">
  <xs:union>
    <xs:simpleType>
      <xs:restriction base="xs:integer">
        <xs:minInclusive value="0">
      </xs:restriction>
    </xs:simpleType>
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="unbounded"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Union Types (4)

```
<!-- Using a mixture of both: -->
<xs:simpleType name="cardinality">
  <xs:union memberTypes="nonNegativeInteger">
    <xs:simpleType>
      <xs:restriction base="xs:string">
        <xs:enumeration value="unbounded"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:union>
</xs:simpleType>
```

Union Types (5)

`<union>`:

- Possible attributes:
  - ◇ `id`: Unique ID

    All XML Schema elements have attribute `id` of type `ID`. It will not be explicitly mentioned for the other element types.
  - ◇ `memberTypes`: component types of the union

    This is a list of `QName` values. The attribute or a `simpleType`-child (or both) must be present (empty unions are not reasonable).
- Content model:

```
annotation?, simpleType*
```

Union Types (6)

`<union>`, continued:

- Possible parent element types: `simpleType`.

  Normally, it is not really necessary to specify the possible parent element types, since this information can be derived from the content model of the other element types. However, this is at least useful cross-reference information: It simplifies the understanding where the current element type can be used. Furthermore, sometimes an element type has different syntactic variants depending on the context in which it appears (remember that this is a feature of XML Schema that goes beyond the possibilities of DTDs). Then the parent type really gives important information.

- Union types can be restricted by facets `pattern` and `enumeration`.

List Types (1)

- A simple type can be constructed as list of values of another simple type.

  The component type cannot be itself a list type, not a union that contains a list, and so on. Because of the lexical representation, nested lists could not be distinguished from the corresponding flat list. List types can be defined only for simple types, not for complex types.

- The lexical represenation of a list value is a string that consists of the lexical representation of the single values, separated by whitespace.

  Whitespace is one or more spaces, tabs, and line breaks. This is the same representation that is used in classical SGML/XML e.g. for `IDREFS`: This type is defined in XML Schema as list of `IDREF` values.

List Types (2)

- Suppose we want to define a list of weekdays when a museum is open:

```
<museum name="Big Art Gallery"
    open="Tue Wed Thu Fri Sat Sun"
    from="09:00:00" to="17:00:00"/>
<museum name="Private Computer Collection"
    open="Sat Sun"
    from="15:00:00" to="18:00:00"/>
```

- This can be done as follows:

```
<xs:simpleType name="weekdayList">
  <xs:list itemType="weekday"/>
</xs:simpleType>
```

List Types (3)

- Instead of specifying a named component type in the `itemType` attribute, one can also define a type in a `simpleType` child element:

```
<xs:simpleType name="weekdayList">
  <xs:list>
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="Sun"/>
        ...
      </xs:restriction>
    </xs:simpleType>
  </xs:list>
</xs:simpleType>
```

List Types (4)

- The constants of the list item type must not contain whitespace.

  The input string is split into list elements at whitespace before the single list elements are validated.

- Instead of a list type, one could also use a sequence of elements:
  - ◇ Advantage of list type: shorter.
  - ◇ Advantage of element list: List items can be structured (e.g. attributes can be added).

    Furthermore, currently XPath and XSLT do not permit access to the single items in a list type, but one can of course select single elements in a sequence.

List Types (5)

`<list>`:

- Possible attributes:
  - ◇ `itemType`: Type of list elements (a `QName`).

    One must use either this attribute or a `simpleType` child element. One cannot use both.
- Content model:

  `annotation?, simpleType?`
- Possible parent element types: `simpleType`.

List Types (6)

- List types can be restricted by facets:
  - ◇ `length`, `minLength`, `maxLength`,

    The length is the number of list items, not the string length of the lexical representation. If necessary, the string length can be restricted with `pattern`. Note that empty lists are possible. If necessary, use `minLength` with a value of `1`.
  - ◇ `pattern`,

    This is a pattern for the entire list, not for the list items. A pattern for the list items can be specified in the definition of the item type.
  - ◇ `enumeration`.

Restrictions: Summary (1)

`<restriction>` (for simple types):

- Possible attributes:
  - ◇ `base`: Name of the type to be restricted (a `QName`).

    Either this attribute or a `simpleType` child must be used.
- Content model:

```
annotation?, simpleType?,
     ( minExclusive | minInclusive
     | maxExclusive | maxInclusive
     | length | minLength | maxLength
     | totalDigits | fractionDigits
     | enumeration | pattern | whiteSpace)*
```

- Possible parent element types: `simpleType`.

Restrictions: Summary (2)

`<restriction>`, continued:

- The above content model is a little too generous:
  - ◇ `length` cannot be used together with `minLength` or with `maxLength`.
  - ◇ Also `minExclusive` and `minInclusive` cannot be used together.
  - ◇ The same for `maxExclusive` and `maxInclusive`.
  - ◇ Except `enumeration` and `pattern`, one cannot use the same facet twice.
- And there are restrictions given by the base type.

Restrictions: Summary (3)

`<minInclusive>`, `<minExclusive>`, ... (facets):

- Possible attributes:
  - ◇ `value`: The parameter of the restriction.

    This attribute is required. Its type depends on the facet.
  - ◇ `fixed`: A boolean value that indicates wether this facet can be further restricted in derived types.

    The default value is `false`. Note that this attribute is not applicable for `pattern` and `enumeration`.
- Content model:

  `annotation?`
- Possible parent element types: `restriction`.

Simple Types: Declaration (1)

`<simpleType>` (with name):

- Possible attributes:
  - ◇ `name`: Name of the type (an `NCName`).
  - ◇ `final`: Restrictions for the derivation of other types from this one (see below).
- Content model:

```
annotation?, (restriction | list | union)
```

- Possible parent element types: `schema`, `redefine`.

Simple Types: Declaration (2)

`<simpleType>` (without name):

- Possible attributes:
  - ◇ (only `id`)
- Content model:

  ```
  annotation?, (restriction | list | union)
  ```

- Possible parent element types: `element`, `attribute`, `restriction`, `list`, `union`.

Simple Types: Declaration (3)

Attribute `final`:

- One can forbid that a type is used for deriving other types (inspired by object-oriented languages).
- Possible values of the attribute are:
  - ◇ `#all`: There cannot be any derived type.
  - ◇ Lists of `restriction`, `list`, `union`: Only the explicitly listed type derivations are forbidden.
- If `final` is not specified, the value of the attribute `finalDefault` of the `schema`-element is used (which in turn defaults to `""`, i.e. no restrictions).

Overview

Content Models (1)

- Content models are used to describe the sequence of elements that are nested inside an element (child elements).
- Content models in XML Schema offer basically the same possibilities as content models in DTDs:
  - ◇ `sequence`: Corresponds to "`,`" in DTDs.
  - ◇ `choice`: Corresponds to "`|`" in DTDs.
  - ◇ `all`: Corresponds to "`&`" in (SGML) DTDs.
- The attributes `minOccurs` and `maxOccurs` take the place of "`?`", "`*`", "`+`" in DTDs.

Content Models (2)

- `all` means that the elements in the group must occur (unless `minOccurs=0` for that element), but the order is arbitrary (any permutation is permitted).
- In XML Schema, `all` groups are very restricted:
  - ◇ They must appear on the outermost level, and they cannot contain other model groups, only elements.

    I.e. `all` cannot be used together with `choice` and `sequence`.
  - ◇ For every element it contains, `maxOccurs` must be `1` (`minOccurs` may be `0` or `1`).

Content Models (3)

- Like XML DTDs, XML Schema requires deterministic content models, e.g. this is not permitted:

```
<!-- Invalid! Corresponds to (A | (A, B)) -->
<xs:complexType name="nondeterministic">
  <xs:choice>
    <xs:element name="A" type="xs:string"/>
    <xs:sequence>
      <xs:element name="A" type="xs:string"/>
      <xs:element name="B" type="xs:string"/>
    </xs:sequence>
  </xs:choice>
</xs:complexType>
```

Content Models (4)

`<sequence>`, `<choice>`:

- Possible attributes:
  - ◇ `minOccurs`: Minimum number of times the group must occur (`nonNegativeInteger`)

    The default value of both, `minOccurs` and `maxOccurs`, is 1.
  - ◇ `maxOccurs`: Maximum number of times the group may occur (`nonNegativeInteger` or "`unbounded`")
- Content model:

  `annotation?, (element|group|choice|sequence|any)*`
- Possible parent elements: `complexType`, `restriction`, `extension`, `group`, `choice`, `sequence`.

Content Models (5)

`<all>`:

- Possible attributes:
  - ◇ `minOccurs`: Minimum number of times the group must occur (0 or 1)

    The default value of both, `minOccurs` and `maxOccurs`, is 1.
  - ◇ `maxOccurs`: Maximum number of times the group may occur (1 is the only legal value)
- Content model:

  ```
  annotation?, element*
  ```
- Possible parent elements: `complexType`, `restriction`, `extension`, `group`.

Content Models (6)

`<element>` (element reference):

- Possible attributes:
  - ◇ `ref`: Name of the element being referenced (a `QName`, the element must be declared globally).

    If `element` is used as element reference, this attribute is required. The element may be declared later or (if it does not occur in the data) may be declared not at all.
  - ◇ `minOccurs`, `maxOccurs`: (see above)
- Content model:

  `annotation?`
- Possible parent elements: `all`, `choice`, `sequence`.

Content Models (7)

Named Model Groups:

- It is possible to introduce a name for a model group, and to use this “named model group” as part of other model groups (like macro/parameter entity).
- Thus, if one must declare several element types that have in part equal content models, it suffices to define the common part only once.

  If one wants to define a common part only once without named model groups, one needs an element as a container for this part. This makes the instance (data file) more complicated (additional level of nesting).

Content Models (8)

Named Model Groups, continued:

- Advantages:
  - ◇ This helps to ensure the consistency of similar content models.

    This especially holds also for later changes: The common part has to be changed only in a single place.
  - ◇ Makes equal parts obvious in the schema.
  - ◇ The schema becomes shorter.

    If the common part is sufficently large.
  - ◇ Permits reusable components below the element / complex type level.

Content Models (9)

`<group>` (named model group definition):

- Possible attributes:
  - ◇ `name`: Name of the model group being defined (an `NCName`).

    If `group` is used for defining a named model group, this attribute is required.
- Content model:

  `annotation?, (all | choice | sequence)`
- Possible parent element types: `schema`, `redefine`.

Content Models (10)

`<group>` (named model group reference):

- Possible attributes:
  - ◇ `ref`: Name of the model group being referenced (a `QName`).

    If `group` is used to refer to a named model group, this attribute is required.
  - ◇ `minOccurs`, `maxOccurs`: (see above)
- Content model:

  `annotation?`
- Possible parent elements: `complexType`, `restriction`, `extension`, `choice`, `sequence`.

Content Models (11)

Wildcard:

- With "<any>" it is possible to allow arbitrary elements (one can restrict the namespace).
- E.g., to permit arbitrary XHTML in a product description (without explicitly listing elements):

```
<xs:complexType name="Description" mixed="true">
  <xs:sequence>
    <xs:any minOccurs="0" maxOccurs="unbounded"
        namespace="http://www.w3.org/1999/xhtml"
        processContents="skip"/>
  </xs:sequence>
</xs:complexType>
```

Content Models (12)

Wildcard, continued:

- With the attribute `processContents`, one can select whether the contents of elements inserted for the wildcard should be checked:
  - ◇ `skip`: Only the well-formedness is checked.
  - ◇ `lax`: If the XML Schema processor can find declarations of the elements, it will validate their contents. Otherwise no warning is printed.

    For instance, there might be a configuration file that contains a mapping from namespaces to schemas. Or an RDDL description is stored under the namespace URI, with a link to the schema.
  - ◇ `strict`: Full validation.

Content Models (13)

Wildcard, continued:

- A wildcard is a "quick&dirty" solution. There are safer ways to use elements from another schema (see below).

  In this case, `processContents` was set to "`skip`". But even if it were set to "`strict`", this would not prevent XHTML elements like `meta` (intended for the head). Thus, even then it is not guaranteed that a product catalog generated in XHTML will be valid XHTML. Furthermore, one could also use `h1` (biggest headline) and other elements that will not look nice if they appear in a product description. The only safe solution is probably to explicitly list the allowed XHTML elements. With a bit of luck, the schema for XHTML contains a named model group that can be used.

Content Models (14)

Wildcard, continued:

- With the attribute `namespace`, one can restrict the namespace of the elements matched with `any`:
  - ◇ `##any`: no restriction (this is the default).
  - ◇ `##other`: any namespace except the target namespace of this schema.

    In this case, it is required that the elements have a namespace.
  - ◇ List of URIs, "`##local`", "`##targetNamespace`": Only these namespaces are permitted.

    "`##local`" allows elements without a namespace,
    "`##targetNamespace`" allows elements from this schema.

Content Models (15)

`<any>`:

- Possible attributes:
  - ◇ `namespace`: Restrictions for the namespace of the elements inserted for the wildcard.

    See Slide 4-129. The default is no restriction (“`##any`”).
  - ◇ `processContents`: Defines whether the contents of elements matched with “`any`” is checked.

    See Slide 4-127. The default is “`strict`”.
  - ◇ `minOccurs`, `maxOccurs`: (see above)
- Content model:

  ```
  annotation?
  ```
- Possible parent element types: `choice`, `sequence`.

Complex Types (1)

- Complex types are used to define the characteristics of elements (content model and attributes).
- However, if an element has no attributes and no element content (only a string, number, etc.), a simple type suffices.
- There are two ways to use complex types:
  - ◇ Define a named complex type and reference it in the `type`-attribute of `element`.
  - ◇ Define an anonymous complex type as a child of `element`.

Complex Types (2)

- The possibilities for defining a complex type are:
  - ◇ List the content model (see above), followed by the attributes (see below).

    If the content model is missing (empty), elements of this type have empty content. If the attribute part is not used, elements of this type have no attributes.
  - ◇ Use a `simpleContent` child: For type derivation.

    This is used for deriving a complex type from a simple type (by adding attributes), or from another complex type with simple content (by restriction or extension). See last section of this chapter.
  - ◇ Use a `complexContent` child: For type derivation.

    This is used for restricting or extending a complex type with element content. See last section of this chapter.

Complex Types (3)

`<complexType>` (anonymous type):

- Possible attributes:
  - ◇ `mixed`: Can additional character data appear between the elements of the content model?

    Used for specifying mixed content models. The default is `false`.

- Content model:

```
annotation?, (simpleContent | complexContent |
                      ((all|choice|sequence|group)?,
                       (attribute|attributeGroup)*,
                       anyAttribute?))
```

- Possible parent element types: `element`.

Complex Types (4)

`<complexType>` (for defining a named type):

- Possible attributes:
  - ◇ `name`: Name of the type to be defined (`NCName`).
  - ◇ `mixed`: For mixed content models, see above.
  - ◇ `abstract`, `block`, `final`: See following slides.
- Content model:

```
annotation?, (simpleContent | complexContent |
               ((all|choice|sequence|group)?,
                (attribute|attributeGroup)*,
                anyAttribute?))
```

- Possible parent element types: `schema`, `redefine`.

Complex Types (5)

Attribute `final` (forbids type derivation):

- One can forbid that other types are derived from this type. Possible values of the attribute `final` are:
  - ◇ `"#all"`: There cannot be any derived type.

    `"extension restriction"` (in either order) is equivalent.
  - ◇ `"extension"`: Type derivation by extension is excluded, type derivation by restriction is possible.
  - ◇ `"restriction"`: Conversely.
  - ◇ `""`: Both forms of type derivation are possible.

    If `final` is not specified, the value of the attribute `finalDefault` of the `schema`-element is used (which in turn defaults to `""`).

Complex Types (6)

Attribute `block` (forbids type substitution):

- If an element type E is declared with a complex type C, and C' is derived from C, elements of type E can state that they are really of type C' (with `xsi:type=`C'), and e.g. use the additional attributes or child elements of type C'.
- The attribute `block` can be used to prevent this.

  Possible values are: `"#all"` (i.e. type substitution is not permitted), `""` (i.e. type substitution is possible), `"restriction"` (i.e. only types defined by extension can be used), `"extension"` (i.e. only types defined by restriction can be used), `"extension restriction"` (in either order: same as `"#all"`). The default is `blockDefault` in the `schema`-element, which in turn defaults to `""` (no restriction).

Complex Types (7)

Attribute `abstract` (forbids instantiation):

- If `abstract` is `"true"`, no elements can have this complex type.

  The default value is `"false"`.

- Thus the type is defined only as a basis for type derivation.

  Actually, one can define element types of an abstract complex type, but then type substitution must be used for all elements of this type.

- This corresponds to the notion of abstract superclasses in object-oriented programming.

Attributes (1)

- Elements can have attributes, therefore complex types must specify which attributes are allowed or required, and which data types the attribute values must have.
- Attributes can be declared
  - ◇ globally, and then referenced in complex types,
  - ◇ locally within a complex type (immediately used, never referenced).

    This is a counterpart to "anonymous types" which are defined when they used (and cannot be reused). However, attributes always have a name.

Attributes (2)

- If a target namespace is declared for the schema, globally declared attributes are in this namespace.
- Thus, they need an explicit namespace prefix in each occurrence in the data file.

  Default namespaces do not apply to attributes.
- For locally declared attributes, one can choose whether they must be qualified with a namespace.

  This is done with the `form` attribute (`"qualified"` or `"unqualified"`). A default can be set with the `attributeFormDefault`-attribute of the `schema`-element. If this is not set, the default is `"unqualified"`, i.e. the attribute is used without namespace prefix.

Attributes (3)

- Since one usually does not want to specify a namespace prefix, global attribute declarations are seldom used.

  Global attributes with a namespace prefix are typically used when many or all elements can have this attribute.

- If several elements/complex types have the same attribute, one can define an attribute group (see below), in order to specify the characteristics of the attribute only once.

  When the attribute group is used, it becomes a local declaration (it works like a parameter entity/macro).

Attributes (4)

- As in DTDs, one can specify a default or fixed value for an attribute.

  Fixed values are mainly interesting for global attributes, see Chapter 1.

- If the attribute does not occur in the start tag of an element, the XML Schema processor automatically adds it with the default/fixed value.

  Thus the application gets this value. Attributes with fixed value can have only this single value and usually do not appear in the data file.

- In XML Schema, default/fixed values are specified with the attributes `default`/`fixed` of `attribute` elements. These attributes are mutually exclusive.

Attributes (5)

- As in DTDs, one can specify whether an attribute value must be given in every start tag or not.

  In XML DTDs, the alternatives are: (1) a default value, (2) `#REQUIRED`, (3) `#IMPLIED` (meaning optional), and (4) `#FIXED` with a value.

- In XML Schema, this is done with the attribute "`use`". It can have three possible values:
  - ◇ `"optional"`: Attribute can be left out.
  - ◇ `"required"`: Attribute value must be given.

    This cannot be used together with a default value.
  - ◇ `"prohibited"`: Attribute value cannot be specified.

    This is only used for restricting complex types, see below.

Attributes (6)

`<attribute>` (attribute reference):

- Possible attributes:
  - ◇ `ref`: Name of the attribute (`QName`, required).
  - ◇ `use`: `"optional"`, `"required"`, or `"prohibited"`.

    The default is `"optional"`, i.e. the attribute can be left out.
  - ◇ `default`: Default value for the attribute.
  - ◇ `fixed`: Fixed value for the attribute.
- Content model: `annotation?`
- Possible parent element types:

  `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (7)

`<attribute>` (global attribute declaration):

- Possible attributes:
  - ◇ `name`: Name of the declared attribute (`NCName`).

    This attribute is required.
  - ◇ `type`: Data type of the attribute (`QName`).

    This attribute is mutually exclusive with the `simpleType` child. If neither is used, the default is `anySimpleType` (no restriction).
  - ◇ `default`: Default value for the attribute.
  - ◇ `fixed`: Fixed value for the attribute.
- Content model: `annotation?, simpleType?`
- Possible parent element types: `schema`.

Attributes (8)

`<attribute>` (local attribute declaration):

- Possible attributes:
  - ◇ `name`: Name of the attribute (`NCName`, required).
  - ◇ `type`: Data type of the attribute (`QName`).
  - ◇ `form`: `"qualified"` or `"unqualified"` (→ 4-139).
  - ◇ `use`: `"optional"`, `"required"`, or `"prohibited"`.
  - ◇ `default`, `fixed`: see above.
- Content model: `annotation?, simpleType?`
- Possible parent element types:

  `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (9)

Constraint on Attributes within a Complex Type:

- A complex type cannot have more than one attribute with the same name.

  This is not surprising, because the XML standard requires this already for well-formed XML. Note that the qualified name counts: One could have attributes with the same name in different namespaces.

- A complex type cannot have more than one attribute of type `ID`.

  Also this is a restriction given by the XML standard (although only for DTDs, maybe one could have removed it in XML Schema, but XML Schema anyway has more powerful identification mechanisms). Note also that attributes of type `ID` cannot have default or fixed values.

Attributes (10)

Attribute Wildcard:

- One can permit that the start tags of an element type can contain additional attributes besides the attributes declared for that element type.

  Actually, certain attributes such as namespace declarations, and `xsi:*` are always allowed, and do not have to be explicitly declared.

- This is done by including the attribute wildcard "`<anyAttribute>`" in the complex type definition.
- The wildcard matches any number of attributes.

  This is a difference to the element wildcard `<any>`. Thus, it makes no sense to specify `<anyAttribute>` more than once in a complex type.

Attributes (11)

`<anyAttribute>`:

- Possible attributes:
  - ◇ `namespace`: Restrictions for the namespace of the attributes inserted for the wildcard.

    See Slide 4-129. The default is no restriction (“`##any`”).
  - ◇ `processContents`: Defines whether the value of the additional attributes is type-checked.

    See Slide 4-127. The default is “`strict`”.
- Content model: `annotation?`
- Possible parent element types:

  `complexType`, `restriction`, `extension`, `attributeGroup`.

Attributes (12)

Attribute Groups:

- If several complex types have attributes in common, one can define these attributes only once in an attribute group (example see next slide).

  Since elements / complex types cannot have two attributes with the same name, also attribute groups cannot contain attributes with the same name. In the same way, multiple `ID`-attributes are forbidden.

- This attribute group can then be referenced in a complex type, or in other attribute groups.
- Like model groups, attribute groups are similar to a special kind of parameter entity.

Attributes (13)

- Example for attribute group definition (`CAT`, `ENO`):

```
<xs:attributeGroup name="exIdent">
  <xs:attribute name="CAT" use="required">
    <xs:simpleType>
      <xs:restriction base="xs:token">
        <xs:enumeration value="H"/>
        <xs:enumeration value="M"/>
        <xs:enumeration value="F"/>
      </xs:restriction>
    </xs:simpleType>
  </xs:attribute>
  <xs:attribute name="ENO" use="required"
      type="xs:positiveInteger"/>
</xs:attributeGroup>
```

Attributes (14)

- A reference to the attribute group "`exIdent`" (see previous slide) looks as follows:

```
<attributeGroup ref="exIdent"/>
```

- The attributes of the attribute group (e.g., `CAT` and `ENO`) are inserted in place of the group reference.

  This is basically done like the expansion of a macro/entity. However, a complex type can contain only one attribute wildcard. In XML Schema, it was decided that referencing two attribute groups that both contain wildcards in the same complex type is no error. In this case, the namespace constraints are intersected, and the `processContents`-value of the first group is chosen (a wildcard directly in the complex type counts as first).

Attributes (15)

`<attributeGroup>` (attribute group definition):

- Possible attributes:
  - ◇ `name`: Name of the attribute group (`NCName`).

    This attribute is required.
- Content model:

```
annotation?,
(attribute|attributeGroup)*, anyAttribute?
```

- Possible parent element types: `schema`, `redefine`.

Attributes (16)

`<attributeGroup>` (attribute group reference):

- Possible attributes:
  - ◇ `ref`: Name of the attribute group (`QName`).

    This attribute is required.
- Content model:

  `annotation?`
- Possible parent element types:

  `complexType`, `restriction`, `extension`, `attributeGroup`.

Elements (1)

- The main purpose of an element declaration is to introduce an element type name and associate it with a (simple or) complex type.

  In addition, they can define a default/fixed value for the content, permit or forbid a nil value, define keys or foreign keys, block type substitution, and define substitution groups. See below.

- Simple and complex types together are called data types (to distinguish them from "element types").

  At least in the book "Definitive XML Schema". The Standard uses simply "type" (for simple and complex type) and avoids the word "element type". On my slides, I sometimes incorrectly use "element" instead of "element type". Maybe, "element name" would be good.

Elements (2)

- The association of the declared element type with the simple/complex type can be done in two ways:
  - ◇ By including a `simpleType` or `complexType` child element (anonymous type definition).
  - ◇ By referencing a named (globally declared) simple or complex type with the `type`-attribute.

    The two possibilities are mutually exclusive.
- If none of the two is used, the element type is associated with `anyType`, and permits arbitrary (well-formed) content and arbitrary attributes.

  Unless the element type is part of a substitution group, see below.

Elements (3)

- Element declarations can be
  - ◇ global (later referenced by the element name),

    For element references, see above ("Content Models": 4-121).
  - ◇ local inside a complex type declaration (immediately used and never referenced again).
- As with attributes,
  - ◇ globally declared element types always belong to the target namespace of the schema,
  - ◇ whereas one can choose whether locally declared element types belong to the target namespace or remain unqualified (no namespace).

Elements (4)

- The namespace decision for local element declarations is done with the attribute `form`. It can be
  - ◇ `"qualified"`: The element type name belongs to the target namespace of the schema.
  - ◇ `"unqualified"`: The element type name has no namespace.

  If a local element type declaration does not contain the `form`-attribute, the default is defined with `elementFormDefault` in the `schema`-element. This in turn defaults to `"unqualified"`. The possibility to switch nearly all element types between unqualified and qualified form with a single attribute setting is one aspect of the "Venetian Blind" design.

Elements (5)

- The namespace of elements can be defined implicitly with a default namespace declaration.

  Important difference to attributes: For elements, it is no problem if every element belongs to a namespace (if it is the same namespace).

- However, the user of a schema must know which elements belong to a namespace and which not. One should use a simple rule, e.g.
  - ◇ The root element belongs to the target namespace of the schema, the others not.
  - ◇ All elements belong to the target namespace.
  - ◇ The schema has no target namespace.

Elements (6)

- Global declarations must be used:
  - ◇ for the possible root element type(s),
  - ◇ for element types that participate in substitution groups (see below).
- Local declarations must be used:
  - ◇ if the same element type has different attributes or content models depending on the context.

    It might be better to say if there are different element types with the same name.
  - ◇ if the element type name should be unqualified.

    And at least one name in the schema needs a namespace.

Elements (7)

Default and Fixed Values:

- Whereas in DTDs, one can specify default and fixed values only for attributes, in XML Schema, this is possible for attributes and elements.
- However,
  - ◇ for an attribute, the default/fixed value is automatically added if the attribute is missing,
  - ◇ for an element, the element must still be present, but with empty content.

    In both cases, the validation adds data to the data explicitly given in the input document. This might simplify the application.

Elements (8)

- Only values of simple types can be specified as default/fixed values.

  This is a technical restriction, because default/fixed values are specified in an attribute. But probably default/fixed values for elements were mainly added to make attributes and elements with simple content more similar/interchangable.

- Of course, the default/fixed value must be legal for the declared element content.

  Thus, default/fixed values can be used only for elements with simple content, or mixed content when all child elements are optional.

Elements (9)

- If a default value is declared, there is no way to enter the empty string as element content.

  Then the element is empty, and the default value is added. If the `whitespace`-facet is `collapse`, the default value is added even if there are spaces between start and end tag. But see `xsi:nil` below.

- Note that empty elements can have attributes.

  The default value added as long as the contents is empty.

- A fixed value is very similar to a default value, with the additional constraint that if a value is explicitly specified, it can be only this value.

  Possibly a different lexical representation of the same value.

Elements (10)

Nil:

- Also "nil values" are possible for element content if the element type declaration contains

  `nillable="true"`

  The default value is false.

- This is probably similar to a null value in databases.

  The specific meaning of the nil value depends on the application (i.e. is not defined by XML Schema). The nil value is different from the empty string (and from the missing element).

- Fixed values cannot be combined with `nillable`.

Elements (11)

- In the input document, elements with nil content are marked with `xsi:nil="true"`.

  Where `xsi` is mapped to `http://www.w3.org/2001/XMLSchema-instance`. Note that the attribute `xsi:nil` can be used even if it is not declared for the element type (if the element type is `nillable`).

- In this case, the element content must be empty (but the element can still have attributes).
- It is not required that the element type permits an empty content (but it must be `nillable`).
- If an element is nil, a default value is not added, although the contents looks empty (it is nil).

Elements (12)

`<element>` (global element type declaration):

- Possible attributes:
  - ◇ `name`: Element type name (`NCName`, required).
  - ◇ `type`: Name of simple or complex type (`QName`).
  - ◇ `default`, `fixed`, `nillable`: see above.
  - ◇ `abstract`, `substitutionGroup`, `block`, `final`: see below.
- Content model:

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

- Possible parent element types: `schema`.

Elements (13)

`<element>` (local element type declaration):

- Possible attributes:
  - ◇ `name`: Element type name (`NCName`, required).
  - ◇ `form`: `"qualified"` or `"unqualified"` (see above).
  - ◇ `type`: Name of simple or complex type (`QName`).
  - ◇ `minOccurs`, `maxOccurs`: see above.
  - ◇ `default`, `fixed`, `nillable`: see above.
  - ◇ `block`: see below.
- Content model:

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

- Possible parent elements: `all`, `choice`, `sequence`.

Elements (14)

- The scope of a local element type declaration is the enclosing complex type definition.

  One can have two completely different local element type declarations inside different complex types.

- Within the same complex type, one can declare the same element type more than once, if the associated data type is identical.

  Only the types must be identical. Other properties (like default values) can be different. Anonymous types are never identical, even if they have the same content model and attributes.

  This double declaration might be necessary if the element type appears more than once in a content model and one wants a local declaration.

Elements (15)

Attribute `substitutionGroup`:

- It is possible to define a hierarchy on element types, again similar to subclasses.
- The name of the "superclass" (called the "head of the substitution group" in XML Schema) is defined in the attribute `substitutionGroup` (a `QName`).
- If the declaration of element type `E` contains

  `substitutionGroup="S"`

  then `E` is permitted everywhere where `S` is permitted, i.e. `E` can be substituted for `S`.

Elements (16)

- This is also possible over several levels (if `X` defines `E` as the head of its substitution group, `X` can be substituted for `E` and for `S`).
- Of course, the data types of these element types must be compatible, e.g. the data type of `E` must be derived from the data type of `S` (maybe indirectly) (it can also be the same).
- Alternatives to substitution groups are:
  - ◇ choice model group with all “subclass elements”,
  - ◇ “superclass element” with type substitution.

Elements (17)

Attribute `abstract`:

- If this is `"true"`, the element type cannot be used in input documents (i.e. it cannot be instantiated).
- It can only be used as head of a substitution group ("superclass").

  It appears of course in model groups of the schema, but only as placeholder for one of the element types that can be substituted for this element type. The element type substitution is required in this case.

- The default is `"false"`.

Elements (18)

Attribute `final`:

- With `final="#all"`, one can prevent that the current type can be used as head of a substitution group.

  The default is the value of the `finalDefault`-attribute of the `schema`-element, which defaults to `""`, i.e. no restriction.

- One can also specify restrictions on the data types of the element types that can be substituted for the current element type.

  E.g. `final="restriction"` means that the current element type can be head of a substitution group, but the data type of the substituted element type must be derived by restriction.

Elements (19)

Attribute `block`:

- The attribute `block` can be used to forbid type substitution or usage of substitution groups in the instance (input document, data file).

  As mentioned on Slide 4-136, one can use `xsi:type` in the input document (data file) to state that an element type E has not its normal data type C, but a data type C' that is derived from C.

  With the attribute `block`, certain forms of type derivation (`restriction` or `extension`) can be excluded from this possibility.
  `block="restriction extension"` completely excludes type substitution.

  The list can also contain `substitution`, which forbids element type substitution (via substitution groups). This is basically the same as `final="#all"`, but now only the concrete occurrence in the input document is false, not the schema.

Exercise

- Please define complex types that correspond to the following part of a DTD:

```
<!ELEMENT EXERCISES (EXERCISE)*>
<!ELEMENT EXERCISE (ENO, TOPIC, MAXPT, RESULT*)>
<!ELEMENT ENO (#PCDATA)>
    <!-- Should be positive integer -->
<!ELEMENT TOPIC (#PCDATA)>
<!ELEMENT MAXPT (#PCDATA)>
    <!-- Should be non-negative integer -->
<!ELEMENT RESULT (SID, POINTS)>
<!ELEMENT SID (#PCDATA)>
<!ELEMENT POINTS (#PCDATA)>
    <!-- number with one digit after '.' -->
```

Overview

Integrity Constraints (1)

- DTDs have `ID/IDREF` to permit a unique identification of nodes and links between elements.
- This mechanism is quite restricted:
  - ◇ The identification must be a single XML name.

    A number cannot be used as identification. Composed keys are not supported. On the other hand, DTDs do not allow further restrictions of the possible values (certain format of the names).
  - ◇ The scope is global for the entire document.

    One cannot state that the uniqueness only has to hold within an element (e.g., representing a relation). One cannot specify any constraints of the element type that is referenced with `IDREF`.
  - ◇ This works only for attributes, not for elements.

Integrity Constraints (2)

- XML Schema has mechanisms corresponding to keys and foreign keys in relational databases that solve the problems of `ID/IDREF`.

  They are more complex than the relational counterparts, because the hierarchical structure of XML is more complex than the flat tables of the relational model. The simplicity of the relational model was one of its big achievements. This is given up in XML databases.

- The facets correspond to `CHECK`-constraints that restrict the value set of a single column.

  Not all SQL conditions that refer to only one column can be expressed with facets. On the other hand, patterns in XML Schema are much more powerful than SQL's `LIKE`-conditions. It is strange that patterns refer to the external representation.

Integrity Constraints (3)

- Otherwise, XML Schema 1.0 is not very powerful with respect to constraints (changed in Ver. 1.1).

  E.g., `CHECK`-constraints in relational databases can state that if one column has a certain value then another column must be not null, or that two columns exclude each other. Such constraints are not possible in XML Schema (certain cases can be specified with content models).

- For example, XML Schema itself requires that the `type`-attribute of `element` is mutually exclusive with `simpleType`/`complexType`-child elements. This constraint cannot be specified in XML Schema.

Integrity Constraints (4)

- XML Schema 1.1 (released April 5, 2012) introduced an Element `assert` that permits to specify arbitrary conditions in XPath 2.0.
- For instance, one can compare two attribute values of an element (attribute `min` must be $\leq$ `max`):

```
<xs:complexType name="intRange">
    <xs:attribute name="min" type="xs:int"/>
    <xs:attribute name="max" type="xs:int"/>
    <xs:assert test="@min le @max"/>
</xs:complexType>
```

[https://www.w3.org/TR/2012/REC-xmlschema11-1-20120405/]

Unique/Key Constraints (1)

- Consider again the example:

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>101</SID>
      <FIRST>Ann</FIRST>
      <LAST>Smith</LAST>
    </STUDENT>
    ...
  </STUDENTS>
  ...
</GRADES-DB>
```

Unique/Key Constraints (2)

- SID-values uniquely identify the children of STUDENTS:

```
<xs:element name="STUDENTS">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="STUDENT"
          minOccurs="0" maxOccurs="unbounded"/>
    </xs:sequence>
  </xs:complexType>
  <xs:unique name="STUDENTS_KEY">
    <xs:selector xpath="*"/>
    <xs:field xpath="SID"/>
  </xs:unique>
</xs:element>
```

Unique/Key Constraints (3)

- There are three components to a `unique`-constraint (basically corresponds to relation, row, column(s)):
  - ◇ The scope, which delimits the part of the XML document, in which the uniqueness must hold.

    Every element of the type in which the `unique`-constraint is defined is one such scope.
  - ◇ The elements which are identified.

    The XPath-expression in `selector` specifies how to get from a scope-element to these elements ("target node set").
  - ◇ The values which identify these elements.

    The XPath-expressions in one or more `field`-elements specify how to get from the identified elements to the identifying values.

Unique/Key Constraints (4)

- In the example:
  - ◇ The scope is the `STUDENTS`-element.

    In the example, there is only one `STUDENTS`-element. If there were more than one, the uniqueness has to hold only within each single element.
  - ◇ The elements that are identified are the children of `STUDENTS` (the `STUDENT`-elements).

    One could also write `xpath="STUDENT"`.
  - ◇ The value that identifies the elements is the value of the `SID`-child.

Unique/Key Constraints (5)

- The correspondence of the scope to a relation is not exact:
  - ◇ In the example, it is also possible to define the entire document as scope, but to select only `STUDENT`-elements (see next slide).
  - ◇ In contrast to the `ID`-type, it is no problem if other keys contain the same values.

    Even if the scope is global, the uniqueness of values must hold only within a key (i.e. one could say that the scope is the key).
- Only values of simple types can be used for unique identification.

Unique/Key Constraints (6)

- SID-values uniquely identify STUDENT-elements:

```
<xs:element name="GRADES-DB">
  <xs:complexType>
    <xs:sequence>
      <xs:element ref="STUDENTS"/>
      <xs:element ref="EXERCISES"/>
      <xs:element ref="RESULTS"/>
    </xs:sequence>
  </xs:complexType>
  <xs:unique name="STUDENTS_KEY">
    <xs:selector xpath="STUDENTS/STUDENT"/>
    <xs:field xpath="SID"/>
  </xs:unique>
</xs:element>
```

Unique/Key Constraints (7)

- Example with composed key:

```
<xs:element name="GRADES-DB">
  <xs:complexType>
    ...
  </xs:complexType>
  <xs:unique name="EXERCISES_KEY">
    <xs:selector xpath="EXERCISES/*"/>
    <xs:field xpath="CAT"/>
    <xs:field xpath="ENO"/>
  </xs:unique>
</xs:element>
```

Unique/Key Constraints (8)

- Suppose we store the data in attributes:

```
<EXERCISE CAT='H' ENO='1'
    TOPIC='Rel. Algeb.' MAXPT='10'/>
```

- Attributes as fields are marked with "@":

```
<xs:element name="GRADES-DB">
  ...
  <xs:unique name="EXERCISES_KEY">
    <xs:selector xpath="EXERCISES/*"/>
    <xs:field xpath="@CAT"/>
    <xs:field xpath="@ENO"/>
  </xs:unique>
</xs:element>
```

Unique/Key Constraints (9)

- Example with exercise info nested in categories:

```
<EXERCISES>
  <CATEGORY CAT="H">
    <EX ENO="1" TOPIC="Rel. Algeb." MAXPT="10"/>
    <EX ENO="2" TOPIC="SQL" MAXPT="10"/>
  </CATEGORY>
  <CATEGORY CAT="M">
    <EX ENO="1" TOPIC="SQL" MAXPT="14"/>
  </CATEGORY>
</EXERCISES>
```

- XML Schema supports only a subset of XPath. In particular, one cannot access ancestors in `xs:field`. But the unique identification of `EX` needs `CAT`.

Unique/Key Constraints (10)

- The problem is solved by defining two keys:
  - ◇ One key ensures that the `CAT`-value uniquely identifies `CATEGORY`-elements.
  - ◇ The other key is defined within the `CATEGORY` element type (thus, there is one instance of the key, i.e. scope, for every category element). This key ensures the unique identification of `EX`-elements by the `ENO` within each `CATEGORY` element.
- However, in this way no foreign keys can be specified that reference `EX`-elements by `CAT` and `ENO`.

Unique/Key Constraints (11)

- Key on CATEGORY:

```
<xs:element name="GRADES-DB">
  ...
  <xs:unique name="CATEGORY_KEY">
    <xs:selector xpath="EXERCISES/CATEGORY"/>
    <xs:field xpath="@CAT"/>
  </xs:unique>
</xs:element>
```

The XPath-expression in `selector` could also be `EXERCISES/*` (because `EXERCISES` has only `CATEGORY`-elements as children).

One could define the key also under `EXERCISES` (instead of `GRADES-DB`) since the document contains only one element of type `EXERCISES`, and all elements to be identified are nested within this element.

Unique/Key Constraints (12)

- Key on EX-elements within CATEGORY:

```
<xs:element name="CATEGORY">
  ...
  <xs:unique name="EX_KEY">
    <xs:selector xpath="*"/>
    <xs:field xpath="@ENO"/>
  </xs:unique>
</xs:element>
```

- It is no problem that there are two EX-elements with the same ENO (e.g., 1) as long as they are nested within different CATEGORY-elements.
- This is similar to a weak entity.

Unique/Key Constraints (13)

- For a given “context node” (in which the key is defined), the selector defines a “target node set”.
- For each node in the target node set, the XPath-expression in each field must return 0 or 1 values. It is an error if more than one value is returned.
- The target nodes, for which each field has a value (that is not nil), form the “qualified node set”.
- The unique identification is required only for the qualified node set. Multiple elements with undefined or partially defined key values can exist.

Unique/Key Constraints (14)

- If one writes `xs:key` instead of `xs:unique`, the fields must exist. In this case, it is an error if the XPath-expression in `xs:field` returns no values (and it it always an error if it returns more than one value).

  Furthermore, neither the identified nodes nor the identifying fields may be nillable.

- Note that value equality respects the type:
  - ◇ For a field of type `integer`, `"03"` and `"3"` are the same (so the uniqueness would be violated).
  - ◇ For a field of type `string`, they are different.

Unique/Key Constraints (15)

<unique>/<key>:

- Possible attributes:
  - ◇ name: Name of the key constraint (NCName).

    This attribute is required. The value must be unique in the schema among all unique, key, and keyref-constraints.
- Content model:

```
annotation?, selector, field+
```

- Possible parent element types: element.

Unique/Key Constraints (16)

`<selector>`:

- Possible attributes:
  - ◇ `xpath`: Defines the nodes that are to be identified by the key (restricted XPath expression).

    It is required. The XPath subset is explained below.
- Content model:

  `annotation?`
- Possible parent element types: `unique`, `key`, `keyref`.

Unique/Key Constraints (17)

`<field>`:

- Possible attributes:
  - ◇ `xpath`: Defines a component of the tuple of values that uniquely identifies the nodes.

    This attribute is required. The value must again be a restricted XPath expression, see below.
- Content model:

  `annotation?`
- Possible parent element types: `unique`, `key`, `keyref`.

XPath Subset (1)

- The standard states: "In order to reduce the burden on implementers, in particular implementers of streaming processors, only restricted subsets of XPath expressions are allowed in {selector} and {fields}."
- Indeed, the subset of XPath that can be used to define the components of keys, is quite simple.
- The purpose of XPath is to select a set of nodes in the XML tree, given a context node as a starting point. In the XPath subset, one can navigate only downward in the tree (in full XPath, also upward).

XPath Subset (2)

- The XPath subset that can be used in `selector` and the subset that can be used in `field` differ slightly.
- A selector XPath expression consists of one or more "Paths", separated by "|":

```
Selector ::= Path ('|' Path)*
```

  The set of nodes that are selected by this expression is the union of the nodes selected by the single paths (as usual, "|" means disjunction).
- Between any two tokens, whitespace is allowed.

XPath Subset (3)

- A Path
  - ◇ can optionally start with ".//".
  - ◇ After that, it is a sequence of steps, separated with "/":

```
Path ::= ('.//')? Step ('/' Step)*
```

- Let the start node set be:
  - ◇ If ".//" is present:
    The context node and all its descendants.
  - ◇ Otherwise: Only the context node.

XPath Subset (4)

- Each step defines a new set of nodes, given the resulting nodes from the previous step (initialized with the start node set).

  Formally, a step defines a set of selected nodes for a single given node. If the current node set consists of several nodes, take the union of the selected nodes given each element in the current node set.

- A step can be: `Step ::= '.' | NameTest`
  - ◇ ".": Selects the current node (nothing changed).
  - ◇ A "name test": This selects those children of the current node that are element nodes with an element type name satisfying the "name test".

XPath Subset (5)

- A “name test” is:
  - ◇ An element type name (a `QName`).

    Default namespace declarations do not affect XPath expressions. If the element type is in a namespace, one must use the prefix.
  - ◇ A wildcard “*” (satisfied by all element nodes).
  - ◇ A namespace with a wildcard (satisfied by all element nodes that belong to that namespace).

```
NameTest ::= QName | '*' | NCName ':' '*'
```

  A name test can also be used for attribute nodes (see below).

- That completes the definition of XPath expressions that can be used in the attribute `xpath` of `selector`.

XPath Subset (6)

- The XPath expressions in `field` permit in addition to select an attribute node as last step in `Path`:

  `Path ::= ('.//')? (Step '/')* (Step | '@' NameTest)`

  Although one can use "|" (disjunction) and wildcards, this is probably seldom applied because the XPath expression in `field` must select a single node. The node contents/value is taken implicitly at the end.

- A name test for attributes offers the same three possibilities as explained for element nodes above:
  - ◇ "`Name`": Attribute with that qualified name.
  - ◇ "`*`": Any attribute.
  - ◇ "`Prefix:*`": All attributes in that namespace.

XPath Subset (7)

Exercise:

- Consider again the example:

```
<GRADES-DB>
  <STUDENTS>
    <STUDENT>
      <SID>
```

- The key is defined in the `GRADES-DB` element.
- Above, the following XPath expression was used to select the nodes to be identified: `STUDENTS/STUDENT`.
- Give three alternatives.

Key References (1)

- A “key reference” identity constraint corresponds to a foreign key in relational databases.
- It demands that certain (tuples of) values must appear as identifying values in a key constraint.

  “Key constraint” means `key` or `unique`.
- Example: For each `SID`-value in a `RESULT` element, there must be a `STUDENT`-element with the same `SID` (one can store points only for known students).

  As in relational databases, it is not required that the two fields have the same name.

Key References (2)

- SID-values in RESULT reference SID-values in STUDENT:

```
<xs:element name="GRADES-DB">
  ...
  <xs:key name="STUDENT_KEY">
    <xs:selector xpath="STUDENTS/STUDENT"/>
    <xs:field xpath="SID"/>
  </xs:key>
  <xs:keyref name="RESULT_REF_STUDENT"
      refer="STUDENT_KEY">
    <xs:selector xpath="RESULTS/RESULT"/>
    <xs:field xpath="SID"/>
  </xs:keyref>
</xs:element>
```

Key References (3)

`<keyref>`:

- Possible attributes:
  - ◇ `name`: Name of the foreign key constraint (`NCName`).

    This attribute is required. The value must be unique in the schema among all `unique`, `key`, and `keyref`-constraints.
  - ◇ `refer`: Name of a `unique`/`key`-constraint (`NCName`).

    This attribute is required: By linking the foreign key to the referenced key, it defines which values are possible.
- Content model:

```
annotation?, selector, field+
```

- Possible parent element types: `element`.

Key References (4)

- The referenced key must be defined in the same node or in a descendant node (i.e. “below”) the node in which the foreign key constraint is defined.

  I would have required the opposite direction, because on the way up, there could be only one instance of the referenced key, on the way down, there can be several (see below). But the committee certainly had reasons, probably related to the parsing/checking algorithms.

- The standard explains that “node tables” which map key values to the identified nodes are computed bottom-up.

  The standard talks of “key sequence” instead of “key values” to include also composed keys (with more than one field).

Key References (5)

- It is possible that several instances of the referenced key exist below the foreign key.
- In that case, the union of the node tables is taken, with conflicting entries removed.

  I.e. if two instances of the referenced key contain the same key value with different identified nodes, that key value is removed from the table: It cannot be referenced (the reference would not be unique).

  The situation is even more complicated, if the key is defined in an element type that has descendants of the same type. Then key value-node pairs originating in the current node take precedence over pairs that come from below. Values that come from below are only entered in the node table if they do not cause a conflict.

Key References (6)

- Fields of key and foreign key are matched by position in the identity constraint definition, not by name (as in relational databases).
- Normally, the types of corresponding fields (of the key and the foreign key) should be the same.
- However, if the types of both columns are derived from the same primitive type, it might still work (for values in the intersection of both types).
- But values of unrelated types are never identical: E.g. the string "1" is different from the number "1".

Overview

Derived Complex Types (1)

- There are two ways to derive complex types:
  - ◇ by extension, e.g. adding new elements at the end of the content model, or adding attributes,
  - ◇ by restriction, e.g. removing optional elements or attributes, or restricting the data type of attributes, etc.
- Derived simple types are always restrictions.

  One can extend a simple type by adding attributes, but then it becomes a complex type.

Derived Complex Types (2)

- Extension looks very similar to subclass definitions in object-oriented languages.

  There all attributes from the superclass are inherited to the subclass, and additional attributes can be added.

- However, a basic principle in object-oriented languages is that a value of a subclass can be used wherever a value of the superclass is needed.

- In XML, it depends on the application, whether it breaks if there are additional elements/attributes.

  Since XML Schema has this feature, future applications should be developed in a way that tolerates possible extensions.

Derived Complex Types (3)

- Additional attributes are probably seldom a problem, since attributes are typically accessed by name (not in a loop).
- It was tried to minimize the problems of additional child elements by allowing them only at the end of the content model.
- Formally, the content model of the extended type is always a `sequence` consisting of
  - ◇ the content model of the base type,
  - ◇ the added content model (new child elements).

Derived Complex Types (4)

- Consider a type for STUDENT-elements:

```
<xs:complexType name="STUDENT_TYPE">
  <xs:sequence>
    <xs:element name="SID"   type="SID_TYPE"/>
    <xs:element name="FIRST" type="xs:string"/>
    <xs:element name="LAST"  type="xs:string"/>
    <xs:element name="EMAIL" type="xs:string"
        minOccurs="0"/>
  </xs:sequence>
</xs:complexType>
```

- Suppose that exchange students must in addition contain the name of the partner university.

Derived Complex Types (5)

- Example for type extension:

```
<xs:complexType name="EXCHANGE_STUDENT_TYPE">
  <xs:complexContent>
    <xs:extension base="STUDENT_TYPE">
      <xs:sequence>
        <xs:element name="PARTNER_UNIV"
            type="UNIV_TYPE"/>
      </xs:sequence>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

- The effective content model is now:

```
((SID, FIRST, LAST, EMAIL?), (PARTNER_UNIV))
```

Derived Complex Types (6)

- In the same way, one can add attributes. Suppose that STUDENT_TYPE2 has attributes SID, FIRST, LAST, EMAIL (and empty content).
- Then a new attribute is added as follows:

```
<xs:complexType name="EXCHANGE_STUDENT_TYPE2">
  <xs:complexContent>
    <xs:extension base="STUDENT_TYPE2">
      <xs:attribute name="PARTNER_UNIV"
              type="UNIV_TYPE" use="required"/>
    </xs:extension>
  </xs:complexContent>
</xs:complexType>
```

Derived Complex Types (7)

- Let us return to the case where STUDENT has child elements SID, FIRST, LAST, EMAIL.
- The type of EMAIL might be a simple type:

```
<xs:simpleType name="EMAIL_TYPE">
  <xs:restriction base="xs:string">
    <xs:maxLength value="80"/>
  </xs:restriction>
</xs:simpleType>
```

- Suppose that an attribute must be added that indicates whether emails can be formatted in HTML or must be plain text.

Derived Complex Types (8)

- When an attribute is added to a simple type, one gets a complex type:

```
<xs:complexType name="EMAIL_TYPE2">
  <xs:simpleContent>
    <xs:extension base="EMAIL_TYPE">
      <xs:attribute name="HTML_OK"
          type="xs:boolean" use="optional"/>
    </xs:extension>
  </xs:simpleContent>
</xs:complexType>
```

- Example (element `EMAIL` of type `EMAIL_TYPE2`):

```
<EMAIL HTML_OK="false">brass@acm.org</EMAIL>
```

Derived Complex Types (9)

<simpleContent>/<complexContent>:

- Possible attributes:
  - ◇ mixed (only for complexContent): Is character data is allowed between child elements?

    Possible values are true (for mixed content models) and false (else). The default value is the value in the enclosing complexType element, which defaults to false. This attribute in complexContent is simply an alternative to specifying it in complexType.

- Content model:

  ```
  annotation?, (extension | restriction)
  ```

- Possible parent element types: complexType.

Derived Complex Types (10)

`<extension>` (inside `<simpleContent>`):

- Possible attributes:
  - ◇ `base`: The base type that is extended to define a new type (`QName`, required).

    For an extension inside `simpleContent`, the base type must be a simple type, or a complex type derived from a simple type (i.e. with a simple type as content).
- Content model:
  ```
  annotation?,
  (attribute | attributeGroup)*, anyAttribute?
  ```
- Possible parent element types: `simpleContent`.

Derived Complex Types (11)

`<extension>` (inside `<complexContent>`):

- Possible attributes:
  - ◇ `base`: The base type that is extended to define a new type (`QName`, required).

    For an extension inside `complexContent`, the base type must be a complex type, i.e. it must have element, mixed, or empty content.
- Content model:

```
annotation?,
(group | all | choice | sequence)?,
(attribute | attributeGroup)*, anyAttribute?
```

- Possible parent element types: `complexContent`.

Derived Complex Types (12)

- If one uses restriction to define a derived type, it is guaranteed that every value of the derived type is also a valid value of the original type.
- If one wants to restrict a content model, one must repeat the complete content model.

  I.e. also the unmodified parts must be listed. The restricted content model does not have to be structurally identical. E.g. groups with only a single element can be eliminated (if `minOccurs` and `maxOccurs` are both 1), a `sequence` group with `minOccurs="1"` and `maxOccurs="1"` can be merged with an enclosing `sequence` group, the same for `choice`-groups. However, for `all` and `choice` groups, subgroups must be listed in the same order, although the sequence is semantically not important.

Derived Complex Types (13)

- If one wants to restrict an attribute, it suffices to repeat only this attribute.
- Consider again STUDENT_TYPE2 with attributes SID, FIRST, LAST, EMAIL. The optional attribute EMAIL can be removed as follows:

```
<xs:complexType name="STUDENT_TYPE3">
  <xs:complexContent>
    <xs:restriction base="STUDENT_TYPE2">
      <xs:attribute name="EMAIL"
          use="prohibited"/>
    </xs:restriction>
  </xs:complexContent>
</xs:complexType>
```

Derived Complex Types (14)

- The same change for the type STUDENT with child elements SID, FIRST, LAST, EMAIL (minOccurs="0"):

```
<xs:complexType name="STUDENT_TYPE4">
 <xs:complexContent>
  <xs:restriction base="STUDENT_TYPE">
    <xs:sequence>
      <xs:element name="SID"   type="SID_TYPE"/>
      <xs:element name="FIRST" type="xs:string"/>
      <xs:element name="LAST"  type="xs:string"/>
    </xs:sequence>
  </xs:restriction>
 </xs:complexContent>
</xs:complexType>
```

Derived Complex Types (15)

- Possible restrictions for complex types:
  - ◇ Optional attribute becomes required/prohibited.
  - ◇ The cardinality of elements or model groups becomes more restricted (`minOccurs` ↑, `maxOccurs` ↓).
  - ◇ Alternatives in `choice`-groups are reduced.
  - ◇ A restricted type can be chosen for an attribute or a child element.
  - ◇ A default value can be changed.
  - ◇ An attribute or element can get a fixed value.
  - ◇ Mixed content can be forbidden.

Documentation, App. Info (1)

- Documentation about the schema can be stored within the XML Schema definition.

  And not only as XML comments: Many XML tools suppress comments, and very little formatting can be done there.

- This is one purpose of the `annotation` element type, which is allowed
  - ◇ as first child of every XML Schema element type

    But it cannot be nested, i.e. it cannot be used within `annotation` or its children `documentation` and `appinfo`.
  - ◇ anywhere as child of `schema` and `redefine`.

    There, multiple `annotation` elements are allowed. Inside all other element types, only one `annotation` element is permitted.

Documentation, App. Info (2)

- Many relational databases also have the possibility to store comments about tables and columns in the data dictionary.

  Of course, this is usually pure text, quite short and without formatting.

- The other purpose of the annotation element is to store information for tools (programs) that process XML Schema information within the schema.

  E.g. tools that compute a relational schema from an XML schema, and map data between the two, or tools that generate form-based data entry programs out of the schema data.

- This makes XML Schema extensible.

Documentation, App. Info (3)

`<annotation>`:

- Possible attributes: (only `id`)
- Content model:

  `(documentation | appinfo)*`

- Possible parent element types:

  `all`, `any`, `anyAttribute`, `attribute`, `attributeGroup`, `choice`, `complexContent`, `complexType`, `element`, `enumeration`, `extension`, `field`, `fractionDigits`, `group`, `import`, `include`, `key`, `keyref`, `length`, `list`, `maxExclusive`, `maxInclusive`, `maxLength`, `minExclusive`, `minInclusive`, `minLength`, `notation`, `pattern`, `redefine`, `restriction`, `schema`, `selector`, `sequence`, `simpleContent`, `simpleType`, `totalDigits`, `union`, `unique`, `whitespace`.

Documentation, App. Info (4)

`<documentation>`:

- Possible attributes:
  - ◇ `source`: URI pointing to further documentation
  - ◇ `xml:lang`: natural language of the documentation

    E.g. `de`, `en`, `en-US` (it has type `xs:language`).

- Content model: `ANY`

  In XML schema, this is the `any` wildcard, together with `mixed="true"`. It is processed using lax validation, i.e. one can specify a schema location with `xsi:schemaLocation` (e.g. in the root `xs:schema` element of the schema). Otherwise only the well-formedness is checked.

- Possible parent element types: `annotation`

Documentation, App. Info (5)

<appinfo>:

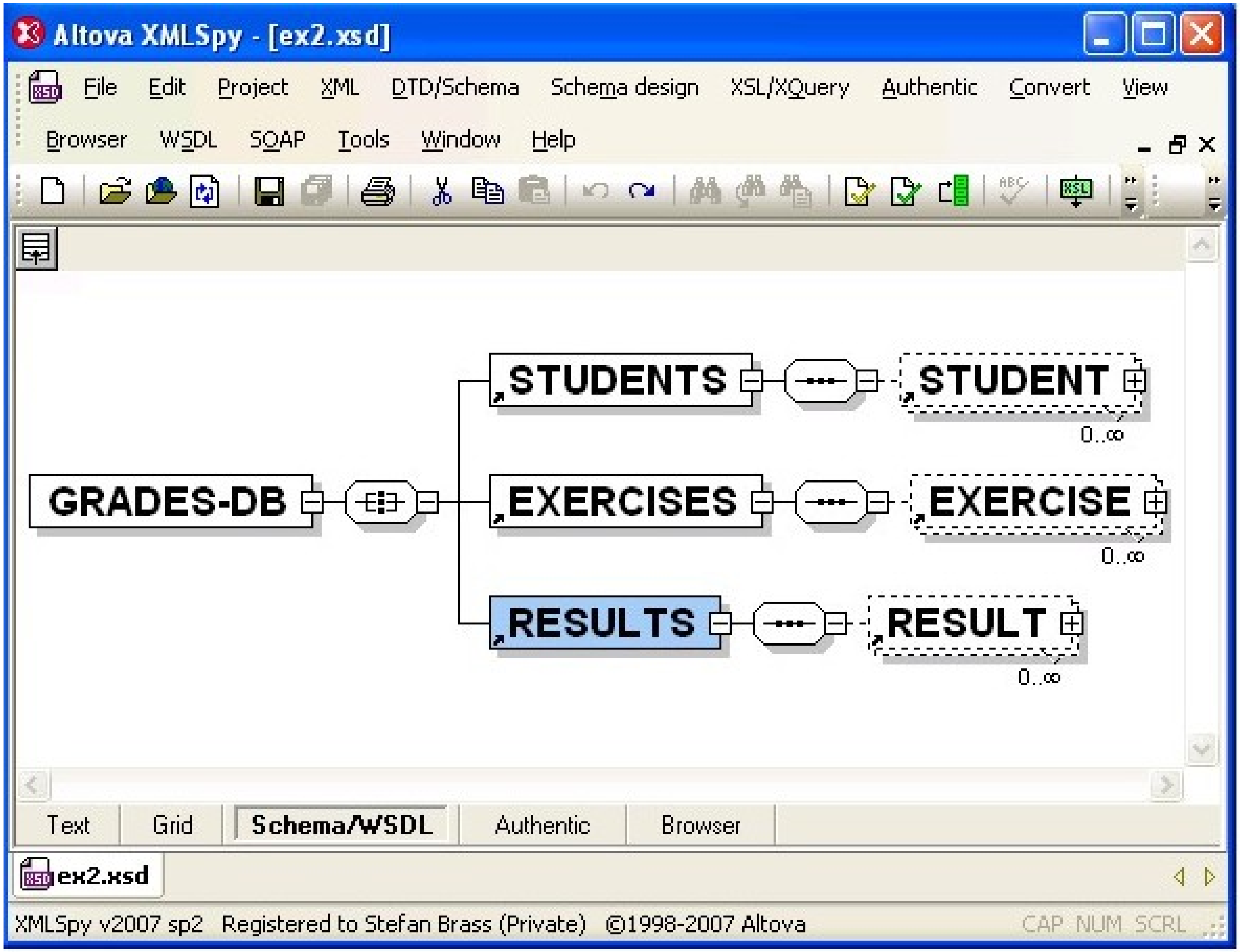
- Possible attributes:
  - ◇ source: URI pointing to further documentation
- Content model: ANY

  I.e. any wildcard with mixed content. Processed using lax validation. So appinfo has the same declaration as documentation, only without the xml:lang attribute.
- Possible parent element types: annotation

Documentation, App. Info (6)

- Example:

```
<xs:schema
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:doc="http://doc.org/d1"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://doc.org/d1 doc.xsd">
  <xs:element name="GRADES-DB">
    <xs:annotation>
      <xs:documentation xml:lang="en">
        <doc:title>Grades Database</doc:title>
        This is the root element.
        ...
    <xs:complexType>
        ...
```


Altova XMLSpy - [ex2.xsd]
File Edit Project XML DTD/Schema Schema design XSL/XQuery Authentic Convert View
Browser WSDL SOAP Tools Window Help
GRADES-DB
STUDENTS
STUDENT
0..∞
EXERCISES
EXERCISE
0..∞
RESULTS
RESULT
0..∞
Text Grid Schema/WSDL Authentic Browser
ex2.xsd
XMLSpy v2007 sp2 Registered to Stefan Brass (Private) ©1998-2007 Altova
CAP NUM SCRL

The Schema Element (1)

`<schema>`:

- Possible attributes:
  - ◇ `targetNamespace`: Namespace for defined schema components.

    At least the global types, elements, and attributes belong to this namespace. Whether local components belong to it depends on the `elementFormDefault`, `attributeFormDefault` and the `form` attribute of the element or attribute declaration.
  - ◇ `elementFormDefault`: `qualified` or `unqualified`.

    Default is `unqualified`. I.e. unless something else is specified in the local element declaration, it has no namespace.
  - ◇ `attributeFormDefault`: `qualified` or `unqualified`.

    Default is `unqualified`.

The Schema Element (2)

`<schema>`, continued:

- Possible attributes, continued:
  - ◇ `blockDefault`: Permission of type substitution (with `xsi:type`) and substitution groups.

    Possible values of this attribute are: `#all` or a list of `substitution`, `extension`, `restriction`. Default is the empty list, i.e. no restriction.
  - ◇ `finalDefault`: Permission of type derivation.

    Possible values are `#all` or a list of `extension`, `restriction`, `list`, `union`. Default is the empty list, i.e. no restriction.
  - ◇ `version`: Version of the schema (type `token`)
  - ◇ `xml:lang`: Language of the schema document.

The Schema Element (3)

`<schema>`, continued:

- Content model:

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

  I.e. if `include`, `import` and `redefine` are used, this must be done before defining the schema components of the current schema document. Note that if one would simply add `annotation` to the `choice` starting with `simpleType`, the content model would not be deterministic.

- Possible parent element types: None.

Including Schema Files

`<include>`:

- Possible attributes:
  - ◇ `schemaLocation`: The URI of the schema document to include.

    The included schema document cannot have a different target namespace than the including schema document. It is ok if it has no namespace.

- Content model:

```
annotation?
```

- Possible parent element types: `schema`.

Importing Schema Files

`<import>`:

- Possible attributes:
  - ◇ `namespace`: The namespace of types, elements, etc. that will be used in this schema.

    This cannot be the same as the target namespace of the current schema document.
  - ◇ `schemaLocation`: URI of a schema document that defines the schema components of `namespace`.
- Content model:

  `annotation?`
- Possible parent element types: `schema`.

Include with Redefinition (1)

`<redefine>`:

- Possible attributes:
  - ◇ `schemaLocation`: The URI of the schema document to include.

    The included schema document must have the same target namespace as the current schema.
- Content model:

  ```
  (annotation | simpleType | complexType |
   attributeGroup | group)*
  ```
- Possible parent element types: `schema`.

Include with Redefinition (2)

- `redefine` automatically includes all schema components from the referenced schema, not only the redefined ones.
- Not arbitrary redefinitions are possible: Types or groups must restrict or extend the original version, it cannot be something entirely new.
- The new, redefined versions also apply to all elements (subtypes etc.) in the included schema.