XML and Databases

Chapter 6: XML Schema II: Simple Types

Prof. Dr. Stefan Brass

Martin-Luther-Universität Halle-Wittenberg

Winter 2019/20

http://www.informatik.uni-halle.de/~brass/xml19/

Objectives

After completing this chapter, you should be able to:

- select or define simple types for an application.
- explain union an list types in XML schema.
- check given XML documents for validity according to a given XML schema, in particular with respect to simple types.

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

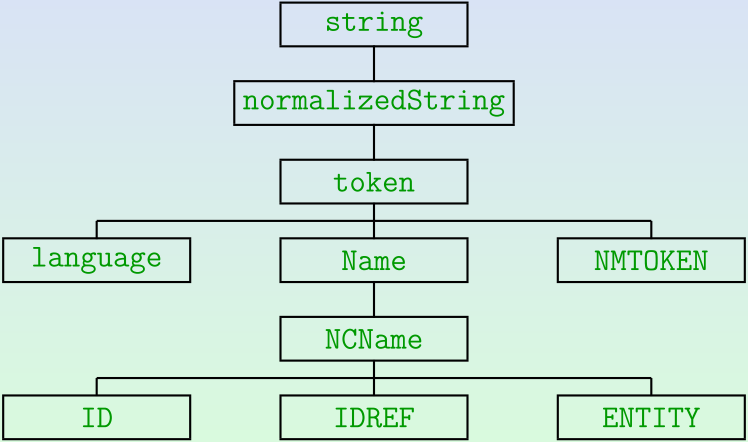
Strings and Names (1)

- 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 ≤ `value`.
  - `minLength`: String length must be ≥ `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 $w_1 \dots w_k$, 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`|`$\ldots$`|`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):
  - [ˆ$c_1 \ldots c_n$]: complement of [$c_1 \ldots c_n$].

    Because of the special meaning of ˆ, 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]`.

  - [ˆ$c_1 \ldots c_n$−E]: Characters matched by [ˆ$c_1 \ldots c_n$], but not matched by E.

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

Date and Time Types (1)

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

  yyyy-*mm*-*dd* T *hh*:*mi*:*ss*.*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.

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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.

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"/>
    ...
```

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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.

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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.

Inhalt

1. Introduction
2. Strings, Names
3. Numbers
4. Date/Time
5. Other
6. UNION
7. LIST
8. Reference

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).

References

- Harald Schöning, Walter Waterfeld: XML Schema.
  In: Erhard Rahm, Gottfried Vossen: Web & Datenbanken, Seiten 33-64.
  dpunkt.verlag, 2003, ISBN 3-89864-189-9.
- 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/]
- [http://www.w3schools.com/schema/]