11. Data Models

Chapter 3: Document Type Definitions (DTDs)

- XML extremely flexible, but some programs reading XML may not be, may need specifically formatted XML → XML 1.0 provides solution: **document type definition (DTD)**
- DTD: written in formal syntax; defines which elements may appear where in a document, what the elements' content and attributes are
- Validation: a validating parser compares a document to its DTD; in cases of violations, programs can decide on actions (either reject whole document, try to fix document, accepts correct parts, etc.)

Validation

- A valid document includes a *document type **declaration*** that identifies the DTD that document satisfies.
- DTD lists all elements, attributes and, entities the document uses or may not use and the contexts in which it uses them.
- Validation follows principle: Everything not permitted is forbidden and everything in document must match a declaration in DTD.
- If a document has a document type declaration and the document satisfies the DTD, the document is valid, else it is invalid.
- DTDs do **not** say:
    - What the root element of a document is
    - How many instances of each kind of element appear in a document
    - What the character data inside the elements look like
    - The semantic meaning of an element (e.g. whether it contains a name of a date)
    - Anything about the length, structure, meaning, allowed values, or other aspects of the text content of an element of attribute
- Well-formedness is required of all XML document, validity is not. I.e. validity errors may be ignored or worked around in some cases or considered fatal in others. Programmer's choice.

Simple DTD Example

DTD:

```
<!ELEMENT person     (name, profession*)>
<!ELEMENT name       (first_name, last_name)>
<!ELEMENT first_name (#PCDATA)>
<!ELEMENT last_name  (#PCDATA)>
<!ELEMENT profession (#PCDATA)>
```

Each line of the DTD is an element declaration (first line declares the **person** element etc.). The breaks are just for readability (ignored by parser).

This DTD may be stored in a separate file from the documents it describes. Allows it to be easily referenced by multiple XML documents. Or can be included in the XML document using the document type declaration.

Valid **person** element:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
</person>
```

Invalid, since it omits required **name** child element:

```
<person>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
</person>
```

Invalid, since a **profession** element comes before the **name**:

```
<person>
  <profession>computer scientist</profession>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
</person>
```

Invalid, since it adds a **publication** element:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>mathematician</profession>
  <publication>On Computable Numbers...</publication>
</person>
```

Invalid, since it adds text outside the allowed children:

```
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  was a <profession>computer scientist</profession>,
  a <profession>mathematician</profession>, and a
  <profession>cryptographer</profession>.
</person>
```

The Document Type Declaration

A document's single document type declaration references a DTD to which it should be compared. Looks like this:

```
<!DOCTYPE person SYSTEM "http://www.cafeconleche.org/dtds/person.dtd">
```

Says the root of the document is **person** and where to find the DTD. This is included in the prolog of a document (prolog: everything in the XML document before the root element start-tag).

- URL: can be absolute (example above), relative (in case document resides at same base site as DTD, e.g. “/dtds/person.dtd”) or just the file name (if in same directory, e.g. “person.dtd”).
- Public IDs: uniquely identifies the XML application in use; local catalog server can convert the public ID into the most appropriate URLs for the local environment; in practice, hardly used; usually, validators rely on the URL to validate the document

Internal DTD subsets

- Internal DTD: Example of a valid person document with an internal DTD

```
<?xml version="1.0"?>
<!DOCTYPE person [
  <!ELEMENT first_name (#PCDATA)>
  <!ELEMENT last_name  (#PCDATA)>
  <!ELEMENT profession (#PCDATA)>
  <!ELEMENT name       (first_name, last_name)>
  <!ELEMENT person     (name, profession*)>
]>
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <profession>computer scientist</profession>
  <profession>mathematician</profession>
</person>
```

- Internal and external DTD: Some documents contain some declarations directly (i.e. internally) but link others using SYSTEM or PUBLIC. E.g. this declaration declares the **profession** and **person** elements itself but needs file *name.dtd* to contain the declaration of **name**.

```
<!DOCTYPE person SYSTEM "name.dtd" [
  <!ELEMENT profession (#PCDATA)>
  <!ELEMENT person (name, profession*)>
]>
```

- the *internal DTD subset*: the part between brackets is called
- the *external DTD subset*: all parts coming from outside the document
- compatibility: The two subsets must be compatible, i.d. neither can override element declarations the other make (thus name.dtd cannot declare the **person** element). However, entity declarations can be overridden with some important consequences for DTD structure and design. (see below “Parameter Entities”)
- standalone: If external subset is used, you should give the standalone attribute the value “no”. (TIP: Almost all XML documents that use external DTD subsets require standalone to have the value no. Since setting standalone to no is always permitted, even when it’s not required, it’s the safest thing to do.)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
```

- A validating processor is required to read the external DTD subset, a nonvalidating processor may do so, but is not required to (even if standalone=”no”). May lead to confusion.

Validation a document

- General rule: web browsers do not validate documents, only check for well-formedness.

- Two good online validators:
    - The Brown University Scholarly Technology Group's XML Validation Form at http://www.stg.brown.edu/service/xmlvalid/
    - Richard Tobin's XML well-formedness checker and validator at http://www.cogsci.ed.ac.uk/~richard/xml-check.html
- (Detailed description on how to use them)

Element Declaration

- Every element used in a valid document must be declared.
- Form: `<!ELEMENT name content_specification>`
- *Name*: can be any legal XML name.
- *Content specification*: what children, and in which order; can be quite complex (e.g. that an element must have three child elements of a given type, or two children of one type followed by another element of a second type)
- Elements can contain (see Book for more examples to each):
    - #PCDATA: parsed character data; e.g. this declaration says that a phone_number element may contain text but not any elements:
      `<!ELEMENT phone_number (#PCDATA)>`
    - Child elements: e.g. `<!ELEMENT fax (phone_number)>`
    - Sequences: of child elements and parsed character data in specific order
- The number of children:
    - ? Zero or one of the element is allowed
    - * Zero or more of the element is allowed
    - + One or more of the element is allowed

  e.g. `<!ELEMENT name (first_name, middle_name?, last_name?)>`

- Choices: e.g. `<!ELEMENT methodResponse (params | fault)>`
  Here **methodResponse** element contains either a **params** child or a **fault** child, but not both

- Parentheses: choices and sequences can be enclosed in parentheses, and then suffixed with a ?, *, or +. These parenthesized items can be nested to form complex elements.

- Mixed content: if a single element contains both child elements and un-marked up, non-whitespace character data. For example:
  `<!ELEMENT definition (#PCDATA | term)*>`
  Valid element:
  `<definition>A <term>Turing Machine</term> refers to an abstract finite state automaton...</definition>`
  You can add any number of other child elements to the list of mixed content, but #PCDATA must always be the first child in the list.
  This is the *only* way to indicate that an element contains mixed content. You **cannot** say:
    - there must be exactly one term child, as well as parsed character data
    - the parsed character data must all come after the term child
    - You cannot use parentheses around a mixed-content declaration to make it part of a larger grouping

  You can only say that the element contains any number of any elements from a particular list in any order, as well as undifferentiated parsed character data.

- Empty elements: declared using the keyword EMPTY: `<!ELEMENT image EMPTY>`
  An empty element cannot contain anything, not even whitepace.
  Valid: `<image source="bus.jpg" width="152" height="345" alt="Alan Turing standing in front of a bus"></image>`

Invalid:
```
<image source="bus.jpg" width="152" height="345"
       alt="Alan Turing standing in front of a bus"> </image>
```

- ANY: an element that makes no assertions about what it may or may not contain (very bad form to use in finished DTDs. Only time you'll see it used is when external DTDs subsets and entities may change uncontrollably, actually quite rare)

Attribute Declarations

A valid document must declare all its elements' attributes. Done with ATTLIST declarations. Single ATTLIST can declare multiple attributes for a single element type. If same attribute repeated for different elements, must be declared separately for each element where it appears.
Example:
```
<!ATTLIST image source CDATA #REQUIRED
                alt    CDATA #IMPLIED
>
```
Here, the **source** attributes are required, and **alt** is optional and may be omitted. All four attributes are declared to contain character data (the most generic attribute type).

```
<!ATTLIST image source CDATA #REQUIRED>
<!ATTLIST image alt    CDATA #IMPLIED>
```
This has the same effect as the declaration above.

- In only well-formed XML attributes can be any string of text with the following restrictions:
  - Any occurrences of < or & must be escaped as < and &
  - Whichever kind of quotation mark (‘ or “), is used to delimit the value must be escaped
- With DTD there are 10 attribute types in XML:
  - CDATA: any string of text acceptable in well-formed XML, most general; used for datas such as prices, URLs, email, citations etc.
  - NMTOKEN: name token, very close to XML name, contains alphanumeric and/or ideographic characters and punctuation marks (_,-,.,:), no whitespace. Difference to XML name: can start with any legal character (.cshrc is valid name token but invalid XML name)
  - NMTOKENS: contains one or more XML name tokens separated by whitespace
  - Enumeration: only attribute type, that is not a key word; list of all possible values for the attribute, separated by vertical bars
    ```
    <!ATTLIST date month (January | February | March | April | May
    | June |July | August | September | October | November |
    December) #REQUIRED>
    ```
  - ID: must contain an XML name that is unique within the document; no other ID type attribute in the document can have the same value (attributes of non-ID type not considered). Each element can have at most one ID type attribute. ID numbers are tricky since a number is not an XML name. Normal solution: prefix with underscore
    ```
    <!ATTLIST employee social_security_number ID #REQUIRED>
    ```
    Valid: `<employee social_security_number="_078-05-1120"/>`
  - IDREF: refers to the ID attribute of some element in the document (thus must be XML name)
    ```
    <!ATTLIST employee social_security_number ID    #REQUIRED>
    <!ATTLIST project  project_id             ID    #REQUIRED>
    <!ATTLIST team_member person              IDREF #REQUIRED>
    ```
    Valid example:

    ```
    <project id="p1">
      <goal>Develop Strategic Plan</goal>
    ```

```
  <team_member person="ss078-05-1120"/>
  <team_member person="ss987-65-4320"/>
</project>
<employee social_security_number="ss078-05-1120">
  <name>Fred Smith</name>
</employee>
<employee social_security_number="ss987-65-4320">
  <name>Jill Jones</name>
</employee>
```

- IDREFS: contains a whitespace-separated list of XML names, each of which must be the ID of an element in the document.

```
<!ATTLIST employee social_security_number ID     #REQUIRED
                          fsteam          IDREFS #REQUIRED>
<!ATTLIST project  project_id             ID     #REQUIRED>
```

Valid example:

```
<project project_id="p1" team="ss078-05-1120 ss987-65-4320">
  <goal>Develop Strategic Plan</goal>
</project>
<employee social_security_number="ss078-05-1120">
  <name>Fred Smith</name>
</employee>
<employee social_security_number="ss987-65-4320" >
  <name>Jill Jones</name>
</employee>
```

- ENTITY: contains the name of an unparsed entity declared elsewhere in the DTD
- ENTITIES: whitespace-separated list of entities
- NOTATION: contains the name of a notation declared in the document's DTD

```
<!NOTATION gif  SYSTEM "image/gif">
<!NOTATION jpeg SYSTEM "image/jpeg">
<!ATTLIST  image type NOTATION (gif | jpeg) #REQUIRED>
```

The type attribute of each **image** element can have one of two values **gif** and **jpeg**.

- Attribute defaults: default declaration for a attribute; four possibilities for the default:
  - #IMPLIED optional, no default value provided
  - #REQUIRED required, no default value provided
  - #FIXED attribute value is constant and immutable; the attribute has the specified value whether the attribute is explicitly noted on an individual instance of an element. If included, must have the specified value.
  - Literal the actual default value is given as a quoted string

- General Entity Declarations
  There are 5 predefined entities: < (<), & (&), > (>), " (“), ' (‘)
  You can define your own entities: for example this defines &super; as an abbreviation for supercalifragilisticexpialidocious:

```
<!ENTITY super "supercalifragilisticexpialidocious">
```

  The entity replacement text must be well-formed. (e.g. you cannot put start-tag in one entity and end-tag in another)

External Entities

- External Parsed General Entities: here again a validating parser must retrieve the external entity, but a nonvalidating parser may or may not do it.

```
<!ENTITY footer SYSTEM
"http://www.oreilly.com/boilerplate/footer.xml">
```

- External Unparsed Entities and Notations: entities containing non-XML data, used for embedding e.g. JPEG photographs, MIDI sound files etc.
```
<!ENTITY turing_getting_off_bus
         SYSTEM "http://www.turing.org.uk/turing/pi1/busgroup.jpg"
         NDATA jpeg>
```
- Notations: since the data in the code above is not in XML format, the NDATA declaration specifies the type of data; here, jpeg is used. XML does not recognize this as an image type, but rather as a notation declared elsewhere using NOTATION like this:
```
<!NOTATION jpeg SYSTEM "image/jpeg">
```

- Embedding unparsed Entities in Documents:
  The DTD only declares the existence, location, and type of the unparsed entity. To actually include it you insert an element with ENTITY type attribute whose value is the name of an unparsed entity declared in the DTD.

  ```
  <!ELEMENT image EMPTY>
  <!ATTLIST image source ENTITY #REQUIRED>
  ```

  Then, this image element would refer to the photograph at http://www.turing.org.uk/turing/pi1/busgroup.jpg:

  ```
  <image source="turing_getting_off_bus"/>
  ```

  XML doesn't guarantee any particular behaviour from an application that encounters this type of unparsed entity. Behaviour really depends on the used parser.

  (TIP: Many developers and the author of the book feel that unparsed entities are complicated and should not be used. In this example, including all necessary information in a single empty element such as
  `<image source=" http://www.turing.org.uk/turing/pi1/busgroup.jpg "/>`
  is arguably preferable to splitting the information between the element where it is used and the DTD of the document.

Parameter Entities

- useful in cases where multiple elements share all or part of the same attribute lists and content specifications; preferable to define a constant that can hold all common parts (changes to the common part needs to be done in one place instead of each element declaration)
- syntax: parameter entity reference declared much like general entity reference, but with an extra percent sign between <!ENTITY and the name of the entity:
```
<!ENTITY % residential_content "address, footage, rooms, baths">
<!ENTITY % rental_content      "rent">
<!ENTITY % purchase_content    "price">
```

  Also dereferenced the same way as general entity, but with percent instead of ampersand:
```
<!ELEMENT apartment (%residential_content;, %rental_content;)>
<!ELEMENT sublet    (%residential_content;, %rental_content;)>
<!ELEMENT house     (%residential_content;, %purchase_content;)>
```

  The parser substitutes the entity's replacement text for the entity reference. The same technique works equally well for attribute types and element names. But works only for external DTDs. Internal DTD subsets do not allow parameter entity references to be only part

of a markup declaration. However, parameter entity references can be used in internal DTD subsets to insert one or more entire markup declarations, typically through external parameter entities.

- Redefining parameter entities: if document uses internal and external DTD subsets, the internal can specify new replacement texts for the entities.
  If ELEMENT and ATTLIST declarations in the external DTD subset are written indirectly with parameter entity references (instead of directly with literal element names) the internal DTD subset can change the DTD for the document. E.g., a single document could add a **bedrooms** child element to the listings by redefining the residential_content entity like this:

```
<!ENTITY % residential_content "address, footage, rooms,
                                bedrooms, baths, available_date">
```

  In case of conflicting entity declarations, first encountered one has precedence. The parser reads the internal DTD first.

- Conditional inclusion: the keywords IGNORE and INCLUDE can be used to "comment out" resp. include a section of declarations.

```
<![IGNORE[ <!ELEMENT production_note (#PCDATA)> ]]>
```

  By defining a parameter entity like this:

```
<!ENTITY % notes_allowed "INCLUDE">
```

  And using a parameter entity reference like this:

```
<![%notes_allowed;[ <!ELEMENT production_note (#PCDATA)> ]]>
```

  We have conditional inclusion, by redefining the **notes_allowed** parameter entity from outside the DTD.

Section 3.9 holds two DTD examples.

Locating Standard DTDs

There are a lot of standard DTDs for each kind of discipline and profession. Some examples:
- http://www.oasis-open.org/cover/siteIndex.html#toc-applications
- http://www.w3.org/TR/

Chapter 17 : XML Schemas

- XML schema: XML document containing a formal description of what comprises a valid XML document. A W3C XML Schema Language schema is an XML Schema written in the particular syntax recommended by the W3C (will be further called as "schema").

- *Instance Document*: an XML document described by a schema. If a document satisfies all the constraints specifies by the schema, it is called *schema-valid*.
- A schema doc is associated with an instance doc through one of the following methods:
  - An xsi:schemaLocation attribute on an element contains a list of namespaces used within the element and the URLs of the schemas used to validate elements and attributes in those namespaces
  - An xsi:noNamespaceSchemaLocation attribute contains a URL for the schema used to validate elements that are not in any namespace

    - A validating parser might be instructed to validate a given document using a provided schema, ignoring any hints inside the document itself
- Schemas vs DTDs: DTDs provide coarse control over element nesting, element occurrence constraints, permitted attributes, and attribute types and default values. Schemas, however, include following features:
  Simple and complex data types, type derivation and inheritance, element occurrence constraints, and namespace-aware element and attribute declarations
  also possible to define simple types more fine-grained
- !! schemas cannot define general entities (XML inclusions may replace some uses of general entities). DTDs remain extremely convenient for short entities.

Schema Basics

- Document organization: every schema document consists of a singe root xs:schema element. Instance elements declared using top-level xs:element elements are considered global elements. In the same scope, you cannot have two global elements with the same name. Declaring an element and an attribute with the same name is no problem.

- Annotation: good practice to include some documentary material about who authored the schema, what is it for, copy right restrictions etc. Use xs:annotation element which may contain xs:documentation and xs:appinfo, used to provide human- resp. machine-readable information.

- Element declarations: `<xs:element name="fullName" type="xs:string">`

- Built-in simple types:
    - anyURI — Uniform resource identifier
    - base64Binary — Base64-encoded binary data
    - boolean — true or false, 0 or
    - byte — a signed byte quantity >=-228 and <=227
    - dateTime — an absolute data and time
    - duration — a length of time, expressed in units of years, months, …
    - ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS — Same values as defined in the attribute declaration section of the XML 1.0 Recommendation
    - integer — any positive or negative number
    - language — may contain same values as xml:lang
    - Name — an XML name
    - string — Unicode string

- Attribute declarations: e.g. add a **language** attribute of a new complex type based on the built-in type xs:string to **fullName**

```
<xs:element name="fullName">
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base="xs:string">
        <xs:attribute name="language" type="xs:language"/>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
```

- Target namespaces:
  `<xs:schema xmlns:xsi="http://www.w3.org/2001/XMLSchema"`

```
targetNamespace="http://namespaces.oreilly.com/xmlnut/address">
```
defines the namespace of the schema, i.e. all elements, attributes declared in the schema are in the specified target namespace

- Elements inherit the default namespace from the xmlns:"…" attribute, that means an unqualified element name is considered to be in the default namespace

- However, this is **not** the case for attributes. An unqualified attribute doesn't belong to any namespace.

- Controlling qualifications: the **elementFormDefault** and **attributeFormDefault** attributes of the xs:schema element control whether locally declared elements and attributes must be namespace-qualified within instance documents.

Complex Types

- May contain nested elements and have attributes; only elements can contain complex types, attributes always have simple types
  Example:

- Occurrence constraints: **minOccurs** and **maxOccurs** can be used to set the minimum and maximum number of times an element may occur at a particular point in the document. The default value for both (in case not explicitly set) is 1. By setting **maxOccurs** to **unbounded** and **minOccurs** to 0 or 1, one can emulate * and +.

- Types of element content:
  - empty: elements that cannot contain anything. Usually information provided entirely via attributes or their position in relation to other elements (e.g.
)

  - simple content: element only containing simple types, such as the built-in types. Also possible to define new simple types, by using facets (see below for more details). Cannot contain nested elements.

  - complex content: the complexContent element defines extensions or restrictions on a complex type that contains mixed content or elements only.

  - Mixed content: the mixed attribute of the complexType element controls whether character data may appear within the body of the element with which it is associated
    ```
    <xs:complexType mixed="true"/>
    ```

  - Any type: xs:any; any type of markup content, accepts attributes that indicate what level of validation should be performed. Also, it accepts a target namespace that can be used to limit the vocabulary of included content.
    Example:
    ```
    <xs:element name="notes" minOccurs="0">
      <xs:complexType>
        <xs:sequence>
          <xs:any namespace="http://www.w3.org/1999/xhtml"
               minOccurs="0" maxOccurs="unbounded"
               processContents="skip"/>
        </xs:sequence>
      </xs:complexType>
    </xs:element>
    ```
    The attributes of the xs:any element tells the processor that zero or more elements belonging to the XHTML namespace may occur at this location. It also states that the

elements should be skipped, which means no validation will be performed against the actual XHTML namespace by the parser. Other possible values for processContents are **lax** (try to validate any element for which a declaration can be found and silently ignore any unrecognized elements) and **strict** (requires every element to be declared and valid per schema associated with the give namespace). xs:anyAttribute does the same to attributes as xs:any to elements.

Facets

An aspect of a possible value for a **simple data type**; depending on base type some facets make more sense than others (e.g. min & max values make sense for numeric data types but not for boolean). They are applied using the xs:restriction element.

Supported facet types:

- whitespace: 3 possible values
  - preserve: keeps all whitespace exactly as it was
  - replace: replaces all occurrences of tab, line feed, and carriage return with space characters
  - collapse: performs the replace step first and collapses multiple space characters into a single space
- length (or minLength and maxLength): enforces an exact length or a range of length
- Enumeration: restricts possible values to a member of a predefined list
- numeric facets
  - minInclusive and minExclusive/maxInclusive and maxExclusive
  - totalDigits and fractionDigits
- pattern: e.g. `<xs:pattern value="\d\d -\d\d\d -\d\d\d"/>` enforcing a Legi-format
- lists: lists of arbitrary types
- union: in cases where an attribute can have any of several types (e.g. a string from a predefined list, or a new different string → union of list type and string type)

Controlling Element Placement

- The order in which the children elements occur can be fixed using different keywords. In the following example the marked keyword can be replaced by 3 keywords each having a different effect:

```
<xs:element name="letter">
  <xs:complexType mixed="true">
    <xs:keyword >
      <xs:element name="greeting"/>
      <xs:element name="body"/>
      <xs:element name="closing"/>
    </xs:keyword>
  </xs:complexType>
</xs:element>
```

  - xs:sequence: a letter must include a greeting element, a body element and a closing element in that order
  - xs:choice: a letter must include exactly one of the three elements
  - xs:all: a letter must include all three elements but in any order (xs:all can only contain elements that are optional or appear only once)

- xs:group element allows sequences, choices, and model groups of individual element declarations to be grouped together and given a unique name. These groups can then be included in another element-content model using an xs:group element with the **ref** attribute set to the same value as the name attribute of the source group

Using multiple documents

- xs:include: `<xs:include schemaLocation="physical-address.xsd"/>`
  Content included like this is treated as though it were actually a part of the schema document. But unlike external entities, the included document must be a valid schema on its own (i.e. well-formed, xs:schema as root, target namespace must match that of including document)
- xs:redefine: much like include, but types from included schema can be extended, redefined in the scope of the xs:redefine element without changing the original declaration
- xs:import: makes possible to make the global types and elements that are described by a schema belonging to another namespace accessible from within an arbitrary schema (mainly used to use type libraries of e.g. the W3C)

```
<xs:import
namespace="http://www.w3.org/2001/03/XMLSchema/TypeLibrary"
schemaLocation="http://www.w3.org/2001/03/XMLSchema/TypeLibrary.xsd"/
>
```

Derived Complex Types

- xs :extension:

```
<xs:extension base="addr:physicalAddressType">
      <xs:sequence>
        <xs:element name="zipCode" type="xs:string"/>
      </xs:sequence>
    </xs:extension>
```

  This declaration appends the **zipCode** element to **physicalAddressType**. The newly derived type will inherit all declaration added to the underlying type.

- xs:restriction: useful if a new type is a logical subset of an existing type (usually when declaring the new type it is necessary to completely reproduce the parent type definition and omit the parts which are not required)
- Like in OO-programming the substitution principle holds: the derived type may appear in place of the parent type within an instance document
- *Substitution group*: collection of elements that are interchangeable with a particular element (called *head element*) within an instance document. Create a substitution group by adding the attribute substitutionGroup that names the head element for that group to an element declaration.

Controlling Type Derivations

- **abstract**: applies to type and element declarations; when set to **true** the element or attribute cannot appear directly in an instance document. If a type declared **abstract**, no element declared with that type may appear in an instance document
- **final**: can be added to a complex type definition and set to either **#all**, **extension**, or **restriction**. Prevents the complex type from being derived.
- **fixed**: marks facets of simple types as immutable; facets marked as **fixed="true"** cannot be overridden in derived types
- **xs:unique**: enforces element and attribute value uniqueness for a specified set of elements in a schema document; needs to define set of all elements to be evaluated (using a restricted XPath expression), and the precise element and attribute values that must be unique

```
<xs:element name="contacts" type="addr:contactsType" minOccurs="0">
  <xs:unique name="phoneNums">
    <xs:selector xpath="addr:phone"/>
    <xs:field xpath="@addr:number"/>
  </xs:unique>
</xs:element>
```

The example above prevents the same phone number from appearing several times within a given **contact** element

- **xs:key**: almost the same as **xs:unique** but with the difference that every selected element *must* have a value for each the specified fields
- **xs:keyref**: the attribute must match the specified attribute of another element