Week 11 – Data Models

Document Type Definitions

Many programs can work with only some XML applications but not others. XML 1.0 provides a solution to this dilemma: a document type definition (DTD). DTDs are written in a formal syntax that explains precisely which elements may appear where in the document and what the elements’ contents and attributes are. A DTD can make statements such as “A ul element only contains li elements” or “Every employee element must have a social_security_number attribute.” Different XML applications can use different DTDs to specify what they do and do not allow.

**Validation:** A valid document includes a document type declaration that identifies the DTD that the document satisfies. The DTD lists all the elements, attributes, and entities the document uses and the contexts in which it uses them. Validity operates on the principle that everything not permitted is forbidden.

If a document has a document type declaration and the document satisfies the DTD that the document type declaration indicates, then the document is said to be valid. If it does not, it is said to be invalid. **Well-formedness is required of all XML documents; validity is not.**

**The Document Type Declaration:** A valid document includes a reference to the DTD to which it should be compared. This is given in the document’s single document type declaration. A document type declaration looks like this:

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

This says that the root element of the document is person and that the DTD for this document can be found at http://www.cafeconleche.org/dtds/person.dtd.

**Internal DTD Subsets:** When you’re first developing a DTD, it’s often useful to keep the DTD and the canonical example document in the same file so you can modify and check them simultaneously. Therefore, the document type declaration may contain the DTD between square brackets rather than referencing it at an external URL. Example 3-4 demonstrates.
Some document type declarations contain some declarations directly but link in others using a SYSTEM or PUBLIC identifier. For example, this document type declaration declares the profession and person elements itself but relies on the file name.dtd to contain the declaration of the name element:

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

**Element declarations**

Every element used in a valid document must be declared in the document’s DTD with an element declaration. Element declarations have this basic form:

```
<!ELEMENT name content_specification>
```

The name of the element can be any legal XML name. The content specification indicates what children the element may or must have and in what order. Content specifications can be quite complex. They can say, for example, 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, or any elements chosen from seven different types interspersed with text.

**#PCDATA**

The simplest content specification is one that says an element may only contain parsed character data, but may not contain any child elements of any type. In this case the content specification consists of the keyword #PCDATA inside parentheses:
Another simple content specification is one that says the element must have exactly one child of a given type. In this case, the content specification consists of the name of the child element inside parentheses:

```xml
<!ELEMENT fax (phone_number)>
```

**Sequences**

In practice, a content specification that lists exactly one child element is rare. Most elements contain either parsed character data or (at least potentially) multiple child elements. The simplest way to indicate multiple child elements is to separate them with commas. This is called a sequence. It indicates that the named elements must appear in the specified order:

```xml
<!ELEMENT name (first_name, last_name)>
```

**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 required.

For example, this declaration says that a name element must contain exactly one first_name, may or may not contain a middle_name, and may or may not contain a last_name:

```xml
<!ELEMENT name (first_name, middle_name?, last_name?)>
```

**Choices**

Sometimes one instance of an element may contain one kind of child, and another instance may contain a different child. This can be indicated with a choice. A choice is a list of element names separated by vertical bars. For example, this declaration says that a methodResponse element contains either a params child or a fault child:

```xml
<!ELEMENT methodResponse (params | fault)>
```

Choices can be extended to an indefinite number of possible elements:

```xml
<!ELEMENT digit
     (zero | one | two | three | four | five | six | seven | eight | nine)>
```
Parentheses

Individually, choices, sequences, and suffixes are fairly limited. However, they can be combined in arbitrarily complex fashions to describe most reasonable content models:

```xml
<!ELEMENT circle (center, (radius | diameter))>
```

Suffixes can be applied to parenthesized elements, too. For instance, let's suppose that a polygon is defined by individual coordinates for each vertex, given in order:

```xml
<!ELEMENT polygon
  (((x, y) | (r, θ)), ((x, y) | (r, θ)), ((x, y) | (r, θ))*)
```

The plus sign is applied to `((x, y) | (r, θ))`.

Mixed content

In narrative documents, it's common for a single element to contain both child elements and un-marked up, non-whitespace character data. The definition element contains some non-whitespace text and a term child. This is called mixed content. An element that contains mixed content is declared like this:

```xml
<!ELEMENT definition (#PCDATA | term)*)
```

Empty Element

Some elements do not have any content at all. These are called empty elements and are sometimes written with a closing `/`. For example:

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

These elements are declared by using the keyword EMPTY for the content specification. For example:

```xml
<!ELEMENT image EMPTY>
```

Any

Very loose DTDs occasionally want to say that an element exists without making any assertions about what it may or may not contain. In this case, you can specify the keyword ANY as the content specification:

```xml
<!ELEMENT page ANY>
```
### Attribute Declarations

In addition to declaring its elements, a valid document must declare all the elements’ attributes. This is done with ATTLIST declarations. A single ATTLIST can declare multiple attributes for a single element type. However, if the same attribute is repeated on multiple elements, then it must be declared separately for each element where it appears.

For example, ATTLIST declares the source attribute of the image element:

```xml
<!ATTLIST image source CDATA #REQUIRED>
```

It says that the image element has an attribute named source. The value of the source attribute is character data, and instances of the image element in the document are required to provide a value for the source attribute.

A single ATTLIST declaration can declare multiple attributes for the same element. For example, this ATTLIST declaration not only declares the source attribute of the image element, but also the width, height, and alt attributes:

```xml
<!ATTLIST image source CDATA #REQUIRED
  width CDATA #REQUIRED
  height CDATA #REQUIRED
  alt   CDATA #IMPLIED>
```

### Attribute Types

A DTD allows you to make somewhat stronger statements about the content of an attribute value. Indeed, these are stronger statements than can be made about the contents of an element. For instance, you can say that an attribute value must be unique within the document, that it must be a legal XML name token, or that it must be chosen from a fixed list of values.

There are 10 attribute types in XML. They are:

- CDATA
- NMTOKEN
- NMTOKENS
- Enumeration
- ENTITY
- ENTITIES
- ID
- IDREF
- IDREFS
- NOTATION

These are the only attribute types allowed. A DTD cannot say that an attribute value must be an integer or a date between 1966 and 2004, for example.
CDATA

A CDATA attribute value can contain any string of text acceptable in a well-formed XML attribute value. This is the most general attribute type.

NM_TOKEN

An XML name token is very close to an XML name. However, a name token differs from an XML name in that any of the allowed characters can be the first character in a name token, while only letters, ideographs, and the underscore can be the first character of an XML name.

NM_TOKENS

A NM_TOKENS type attribute contains one or more XML name tokens separated by whitespace.

Enumeration

An enumeration is the only attribute type that is not an XML keyword. Rather, it is a list of all possible values for the attribute, separated by vertical bars. Each possible value must be an XML name token.

ID

An ID type attribute must contain an XML name (not a name token but a name) that is unique within the XML document. More precisely, no other ID type attribute in the document can have the same value.

IDREF

An IDREF type attribute refers to the ID type attribute of some element in the document. Thus, it must be an XML name. IDREF attributes are commonly used to establish relationships between elements when simple containment won’t suffice.

IDREFS

An IDREFS type attribute contains a whitespace-separated list of XML names, each of which must be the ID of an element in the document. This is used when one element needs to refer to multiple other elements.
General Entity Declarations

XML predefines five entities:

\&lt;
  The less-than sign, a.k.a. the opening angle bracket (\textlangle)
\&amp;
  The ampersand (\&)
\&gt;
  The greater-than sign, a.k.a. the closing angle bracket (\textgreater)
\&quot;
  The straight, double quotation marks (""
\&apos;
  The apostrophe, a.k.a. the straight single quote (')

It is also possible to define entities for our convenience. For example, we can use the following entity by using \&super:

\&lt;!ENTITY super "supercalifragilisticexpialidocious"\&gt;

Keep in mind that the text we use to replace the entity declaration must be well-formed.

External Parsed General Entities

XML allows us to define entities that reference an external file, known as External Parsed General Entities. For example, whenever the following entity is used, the replacement will be done by the content of the URL:

\&lt;!ENTITY footer SYSTEM "http://www.oreilly.com/boilerplate/footer.xml"\&gt;

Similarly, it is also possible to reference entities that will be included without any parsing:

\&lt;!ENTITY turing_getting_off_bus
  SYSTEM "http://www.turing.org.uk/turing/pil/busgroup.jpg"
  NDATA jpeg\&gt;
XML Schemas

Although DTDs can enforce basic structural rules, many applications need a more powerful and expressive validation method. The W3C developed XML schemas to address these needs.

An XML Schema is an XML document containing a formal description of what comprises a valid XML document. If a document satisfies all the constraints specified by the schema, it is said to be schema-valid. Schemas allow placing constraints in the format and data types of element and attribute values. Besides, schemas provide a framework for declaring new types, deriving new types and reusing types from other schemas.

Schema Basics

Every schema consists of a single root `xs:schema` element. This element contains declarations for all elements and attributes that may appear in a valid instance document. This is an example of a very simple schema:

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

Instance elements declared using top-level `xs:element` elements in the schema (immediate child elements of the `xs:schema` element) are considered global elements. This means that they can appear as root element of any document associated with the schema where they are declared.

Element Declarations

XML documents are composed primarily of nested elements, and `xs:element` is one of the most used declarations in a typical schema. The following example specifies that documents must consist of a single element, `fullName`:

```xml
<xs:element name="fullName" type="xs:string"/>
```

Notice that the declaration specifies the element type.
**Simple types**

Schemas support two different types of content, simple and complex. Simple content consists of pure text that does not contain nested elements. These are the built-in simple types:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>anyURI</td>
<td>A Uniform Resource Identifier</td>
</tr>
<tr>
<td>base64Binary</td>
<td>Base64-encoded binary data</td>
</tr>
<tr>
<td>boolean</td>
<td>May contain either true or false, 0 or 1</td>
</tr>
<tr>
<td>byte</td>
<td>A signed byte quantity $\geq -128$ and $\leq 127$</td>
</tr>
<tr>
<td>dateTime</td>
<td>An absolute date and time</td>
</tr>
<tr>
<td>duration</td>
<td>A length of time, expressed in units of years, months, days, hours, etc.</td>
</tr>
<tr>
<td>ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS</td>
<td>Same values as defined in the attribute declaration section of the XML 1.0 Recommendation</td>
</tr>
<tr>
<td>integer</td>
<td>Any positive or negative integer</td>
</tr>
<tr>
<td>language</td>
<td>May contain same values as xml:lang attribute from the XML 1.0 Recommendation</td>
</tr>
<tr>
<td>Name</td>
<td>An XML name</td>
</tr>
<tr>
<td>string</td>
<td>Unicode string</td>
</tr>
</tbody>
</table>

Attributes must always be declared with simple types since they cannot contain elements. Elements declared as simple types cannot have attributes.

**Attribute Declarations**

Adding attributes to simple types is actually complicated since in principle they should not have any. A new complex type based on the simple type must be declared. Attributes are declared using the `xs:attribute` element. This is an example:

```xml
<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>
```
Working with Namespaces

Schemas can support and describe XML namespaces. To associate a schema with a particular XML namespace, we add the targetNamespace attribute to the root xs:schema element:

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://namespaces.oreilly.com/xmlnut/address">
```

Now, attributes must be updated to point to an attribute group that belongs to the target namespace. The following example shows how global declarations should be prefixed:

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    targetNamespace="http://namespaces.oreilly.com/xmlnut/address"
    xmlns:addr="http://namespaces.oreilly.com/xmlnut/address">
    . . .
    <xs:attributeGroup ref="addr:nationality"/>
    . . .
```

Keep in mind that attributes, unlike elements, don’t inherit the default namespace from the xmlns="..." attribute. They must always be explicitly prefixed if they need to belong to a particular namespace.

Complex Types

A schema assigns a type to each element and attribute it declares. Elements with complex types may contain nested elements and have attributes. Only elements can contain complex types. Attributes always have simple types. New types are defined using xs:complexType or xs:simpleType elements. If a new type is declared globally, it needs to be given a name so that it can be referenced from element and attribute declarations within the schema.

Occurrence constraints

One very useful feature of schemas is the ability to explicitly set the min and max times an element may occur at a particular point in a document. This can be done by using minOccurs and maxOccurs. If they are not provided explicitly, the default value for minOccurs and maxOccurs is 1.
Types of Element Content

So far we have only seen elements that contain character data and elements that contain other elements. However, there are more possible element contents: empty elements, elements with simple content, elements with mixed content and elements with any content.

Empty Elements

An empty element is an element that cannot contain anything. Empty elements convey all their information via attributes or by their position in relation to other elements. To signal that an element is empty it is enough to define it as only being able to contain complex content and not provide any declaration for its nested elements (hence forcing it to remain empty).

Complex Content

Sometimes it is useful to specify that a complex type can only contain complex content (elements). This can be done via de xs:complexType element.

Simple Content

As its own name indicates, the xs:simpleContent can be used to declare an element that can only contain simple content.
Facets

A facet is an aspect of a possible value for a simple data type. For example, a numeric data type can be restricted by the minimum and maximum. The following list covers the facet types supported by a schema processor:

- length (or minLength and maxLength)
- pattern
- enumeration
- whitespace
- maxInclusive and maxExclusive
- minInclusive and minExclusive
- totalDigits
- fractionDigits

One of the more useful types of restriction is the simple enumeration, that allows restricting possible values for an element to a predefined list:

```xml
<x:simpleType name="locationType">
  <xs:restriction base="xs:string">
    <xs:enumeration value="work"/>
    <xs:enumeration value="home"/>
    <xs:enumeration value="mobile"/>
  </xs:restriction>
</xs:simpleType>
```

The xs:pattern facet can be used to place very sophisticated restrictions on the format of string values. The facet compares the values against a regular expression:

```xml
<x:simpleType name="ssn">
  <xs:restriction base="xs:string">
    <xs:pattern value="\d\d-\d\d-\d\d\d\d\d"/>
  </xs:restriction>
</xs:simpleType>
```

Mixed Content

To declare an element that can contain parsed data and elements drawn from a list, we can use mixed content. For this, we use the mixed attribute, which controls whether character data may appear within the body of the element.

```xml
<x:element name="letter">
  <xs:complexType mixed="true"/>
</x:element>
```
Allowing Any Content

It is often necessary to allow users to include any type of markup content they see fit. This can be supported by the xs:any element. A namespace can also be included to limit the vocabulary of included content:

```xml
<x:s:element name="notes" minOccurs="0">
  <xs:complexType>
    <xs:sequence>
      <xs:element any namespace="http://www.w3.org/1999/xhtml"
                  minOccurs="0" maxOccurs="unbounded"
                  processContents="skip"/>
    </xs:sequence>
  </xs:complexType>
</xs:element>
```
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:
  o What the root element of a document is
  o How many instances of each kind of element appear in a document
  o What the character data inside the elements look like
  o The semantic meaning of an element (e.g. whether it contains a name of a date)
  o 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:

```xml
<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:

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

Invalid, since a profession element comes before the name:

```xml
<person>
  <professions>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:

```xml
<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:

```xml
<person>
  <name>
    <first_name>Alan</first_name>
    <last_name>Turing</last_name>
  </name>
  <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:

```xml
<!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
<?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`.

```xml
<!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.e. 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
<?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/](http://www.stg.brown.edu/service/xmlvalid/)
- (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 whitespace.
  - **Valid:** `<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:

```xml
<!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).

```xml
<!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 &lt; and &amp;
  - 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
    ```xml
    <!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.
    ```xml
    <!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)
    ```xml
    <!ATTLIST employee social_security_number ID     #REQUIRED>
    <!ATTLIST project project_id           ID     #REQUIRED>
    <!ATTLIST team_member person           IDREF #REQUIRED>
    ```
    Valid example:
    ```xml
    <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.

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

Valid example:

```xml
<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

```xml
<!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: &lt; (<), &amp; (&), &gt; (>), &quot; ("), &apos; (’)
  You can define your own entities: for example this defines &super; as an abbreviation for supercalifragilisticexpialidocious:
  ```xml
  <!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.
- External Unparsed Entities and Notations: entities containing non-XML data, used for embedding e.g. JPEG photographs, MIDI sound files etc.

```xml
<!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:

```xml
<!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.

```xml
<!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:

```xml
<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

```xml
<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:

```xml
<!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:

```xml
<!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:

  ```xml
  <!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.

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

  By defining a parameter entity like this:

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

  And using a parameter entity reference like this:

  ```xml
  <![%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.oasis-open.org/cover/siteIndex.html#toc-applications)
- [http://www.w3.org/TR/](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 specified 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 single 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 it is for, copyright 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
  - base64Binary
  - boolean: true or false, 0 or 1
  - byte
  - dateTime
  - duration
  - ID, IDREF, IDREFS, ENTITY, ENTITIES, NOTATION, NMTOKEN, NMTOKENS: Same values as defined in the attribute declaration section of the XML 1.0 Recommendation
  - integer
  - language
  - Name
  - 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"/>
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:
    o empty: elements that cannot contain anything. Usually information provided entirely via attributes or their position in relation to other elements (e.g. <br>)

    o 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.

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

    o 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"/>

    o 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"/> 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:

```xml
<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