Introduction to W3C XML Schemas

Paul V. Biron, MLIS
Kaiser Permanente, So Cal Medical Group
Pasadena, CA

HL7 Spring Working Group Meetings
May 22, 2000

Intro

- Background
  - XML, DTDs, other schema languages, namespaces
- Goals of XML Schema
- Intro to XML Schema
  - Simple Types (Datatypes) and Complex Types (Structural Types)
  - Element and Attribute Declarations
- Advanced Topics
  - Restriction, extension, Namespaces, schema composition and access, local element declarations
- *Namespaces in XML (optional)*

XML

```
<?xml version="1.0"?>
<LevelOne><head>
<patient>
<patient.id>123456789</patient.id>
<patient.name>Jane Doe</patient.name>
<patient.date.of.birth>May 13,
    1923</patient.date.of.birth>
<patient.address>123 Main St., Anytown, USA
    (home)</patient.address>
<patient.phone>555-345-9876
    (home)</patient.phone>
</patient>
</head>
</LevelOne>
```

```
<?xml version="1.0" ?>
<PATIENT>
<PATIENT.NAME>
<GIVEN.NAME > Jane</ GIVEN.NAME >
<FAMILY.NAME >Doe</ FAMILY.NAME >
</PATIENT.NAME>
</PATIENT>
```

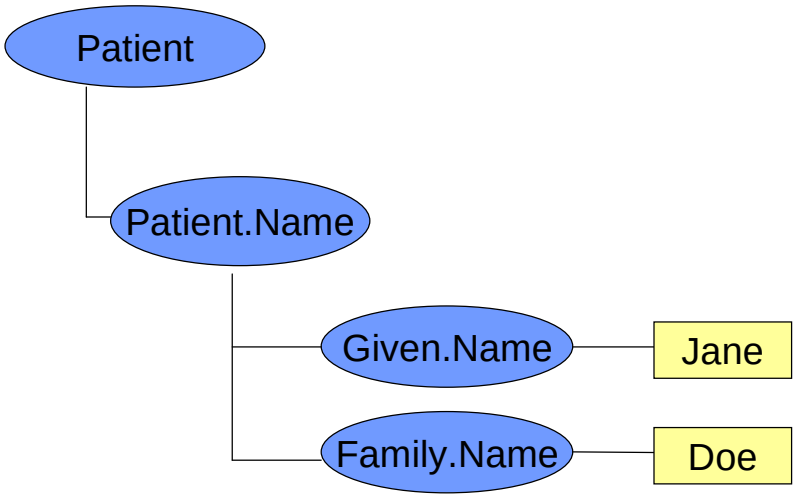
Prolog

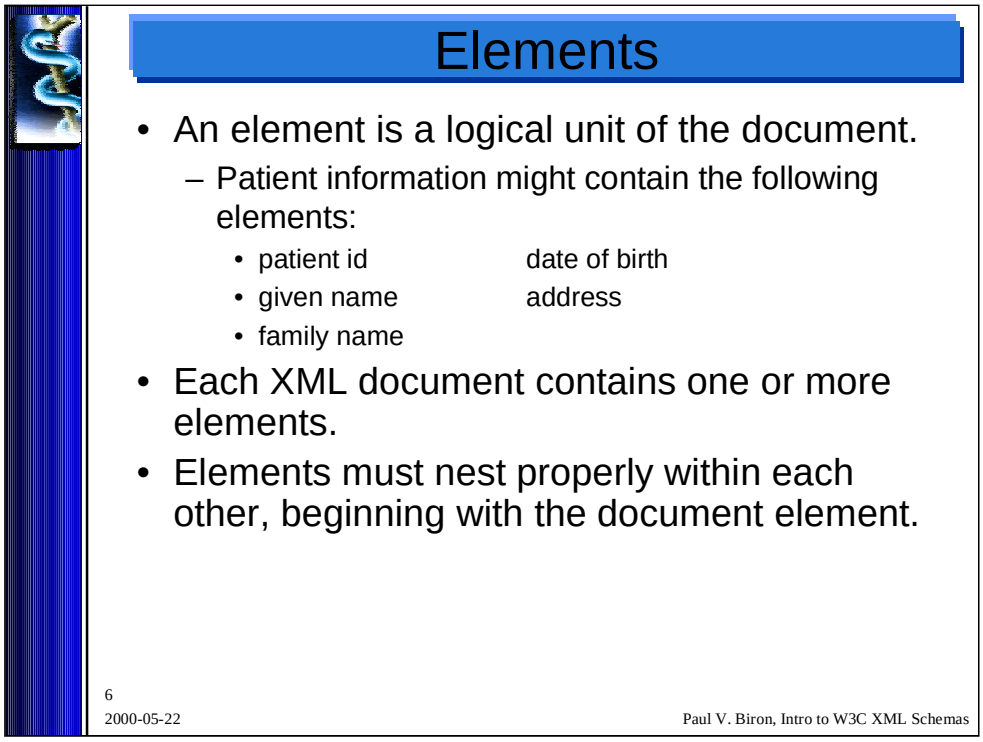
Start Tag

Character Data

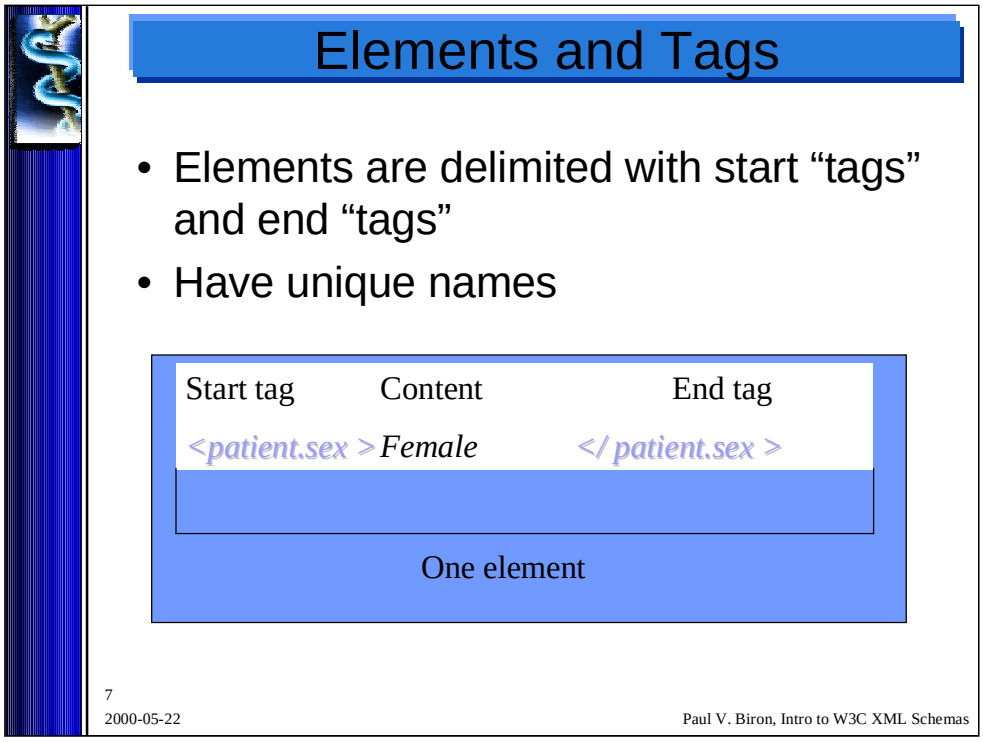
End Tag

Graphical XML Document View





Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value.



Tag names must begin with a letter or underscore. Characters allowed include letters, digits, underscores, hyphens, and periods. White space is not allowed. Tag names are case sensitive.

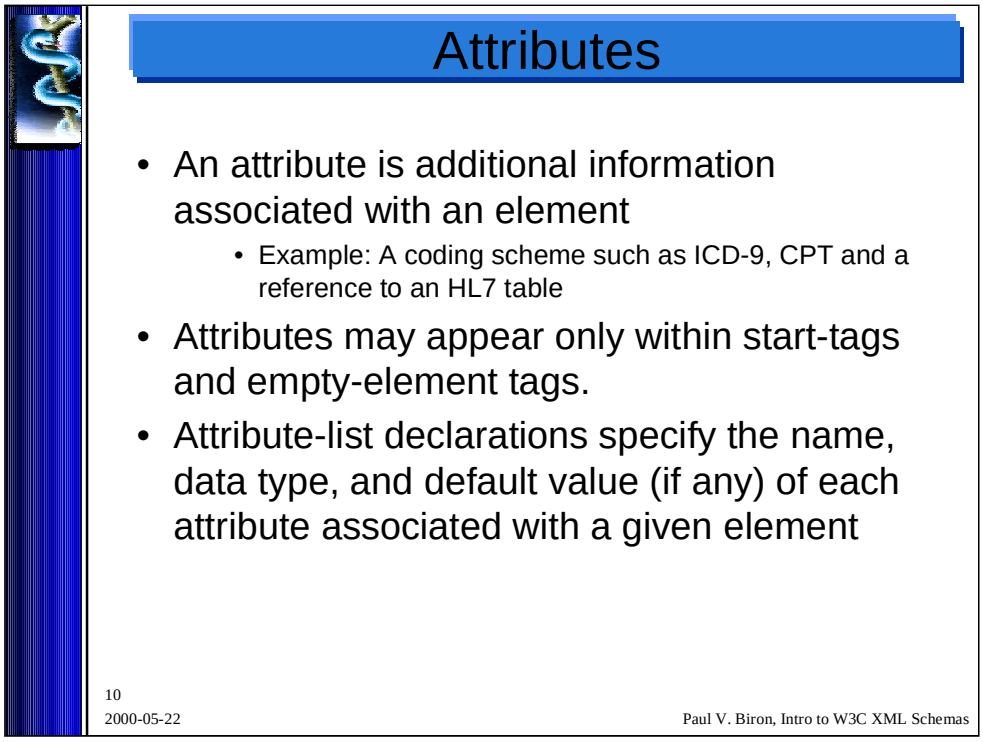
Tag begin with < and end with >. When surrounding content, the start tag name is surrounded by <> (e.g., <patient.sex>) and the end tag is surrounded by </> (e.g., </patient.sex>)

Element Content

- An element's content or the text between that start and end tag has the name #PCDATA.
- #PCDATA means parsed character data. Think of it as text.

Special Case

- The Empty Element
- Some information does not have content; in HTML a line break is an empty element
  -
</BR> does not make sense
- Empty elements in XML have the syntax:
  - <EmptyElement/>



Attributes are used to associate name-value pairs with elements

Attribute declarations begin with <!ATTLIST, followed by element name, attribute name, type, occurrence indicator and closing >, i.e.,

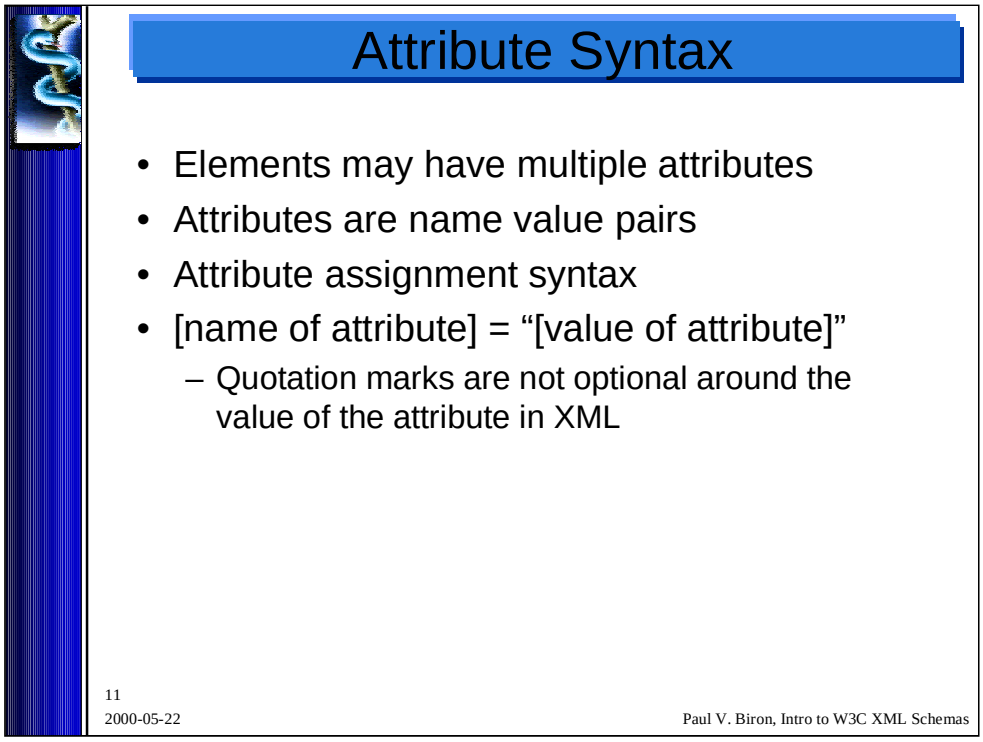
<!ATTLIST ELEMENT attr type occurrence_indicator>

Attribute-list declarations may be used:

To define the set of attributes pertaining to a given element type;

To establish type constraints for these attributes;

To provide default values for attributes



Attribute names must begin with a letter or underscore. Characters allowed include letters, digits, underscores, hyphens, and periods. White space is not allowed.

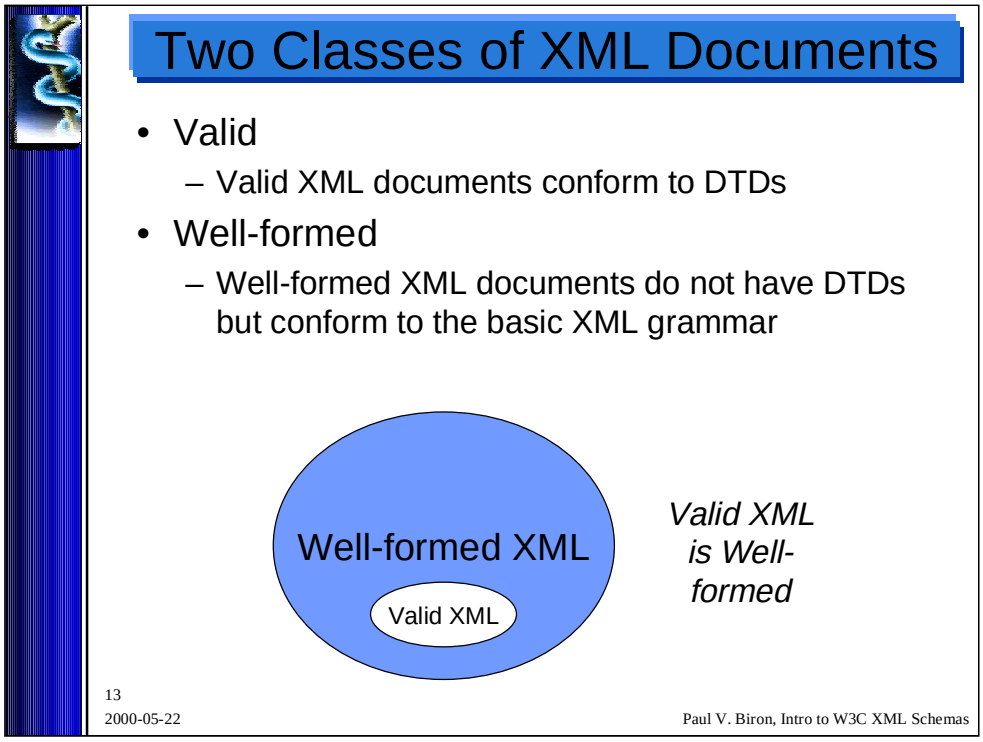
Attribute values can contain white space. Attribute values are delimited by and must be enclosed in quotation marks.

Attribute = Modifier

- Attributes are immediately specified after the element they are associated with and provide further information

Start tag	Attribute	Content	End tag
<patient.sex table = "HL70001" >		*Female*	*</ patient.sex >*

One element with one attribute



Well-formed

Document must have at least one tag following the prolog — the root tag or document tag, which encloses the entire document

All elements must be nested and may not overlap

All elements must have balanced start and end tags

Empty tags must end with />

Attribute values must be enclosed by quotation marks

There must be declarations for any entities used

An entity reference always starts with &. Entity references pre-defined by XML:

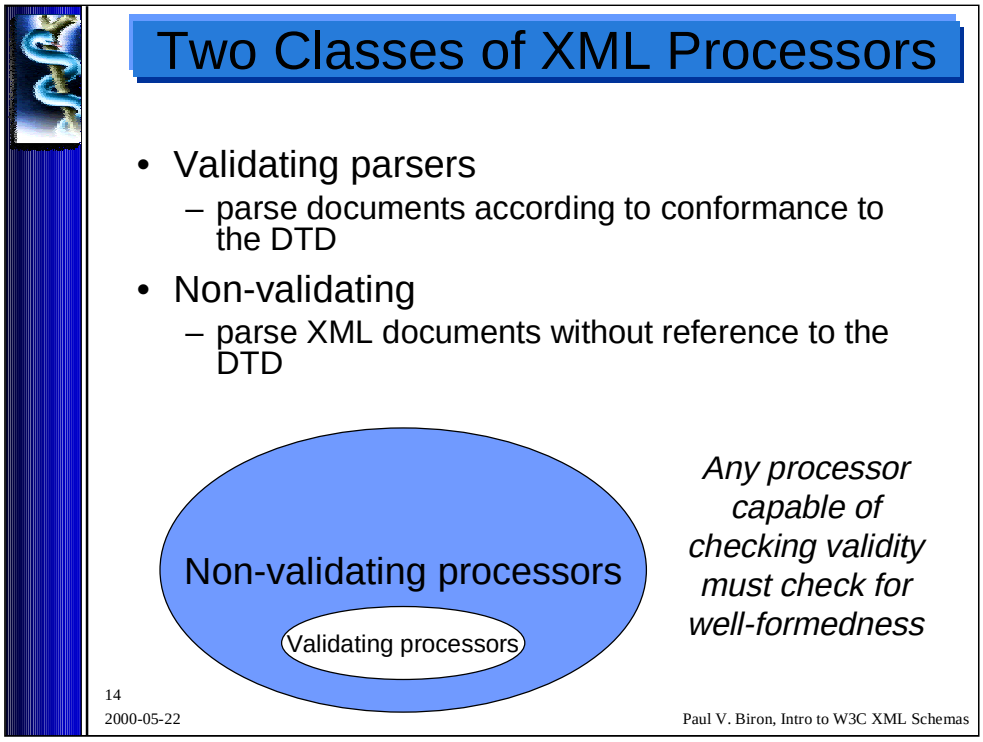
& = &

< = <

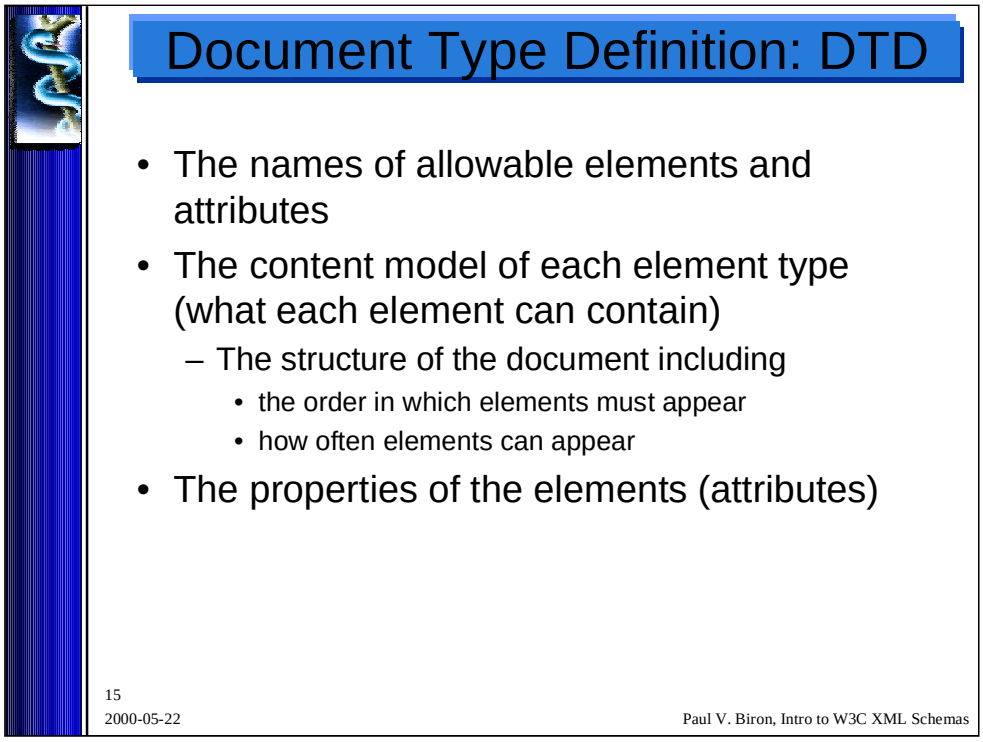
> = >

" = “

' = ‘



If the validating parser finds that the document conforms to the DTD, it passes the data along to the XML application (e.g., Web browser). If it finds a mistake, it reports an error.



The DTD is included in the prolog (which also contains the XML declaration)

Content of elements is PCDATA (text).

Structure of the document—XML differs from SGML in that it is not possible to require all elements but allow them to occur in any order; the order must be specified if all elements are required.

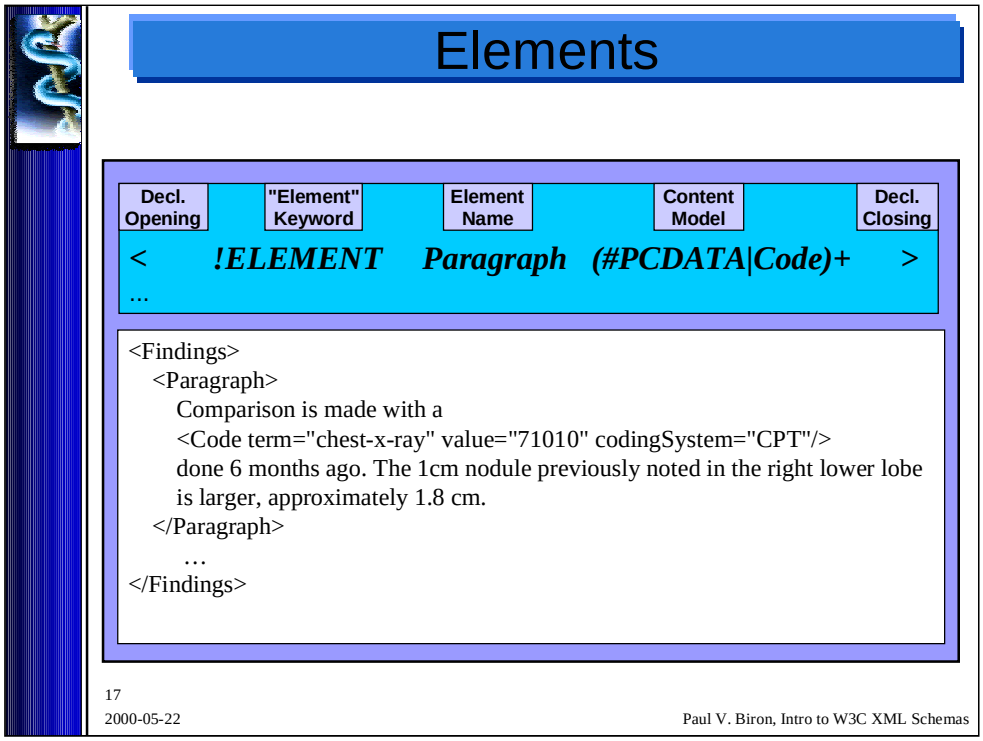
Cardinality of elements is indicated by the following symbols (occurrence indicators):

+ = 1 or more
* = 0 or more
? = optional (0 or 1)
no symbol = required

The DTD may be included in the document it describes or may be linked to from an external URL. External DTDs can be shared by different documents and Web sites.

XML is a self-describing format (the document can be read and displayed by any program that understands DTDs).

```
<?xml version="1.0" ?>
<PATIENT>
<PATIENT.NAME>
<GIVEN.NAME > Jane</ GIVEN.NAME >
<FAMILY.NAME >Doe</ FAMILY.NAME >
</PATIENT.NAME>
</PATIENT>
```



[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'
[46] contentspec ::= 'EMPTY' | 'ANY' | Mixed | children

[39] element ::= EmptyElemTag | STag content Etag
[40] Stag ::= '<' Name (S Attribute)* S? '>'
[41] Attribute ::= Name Eq AttValue
[43] content ::= (element | CharData | Reference | CDSect | PI | Comment)*
[42] ETag ::= '</' Name S? '>'

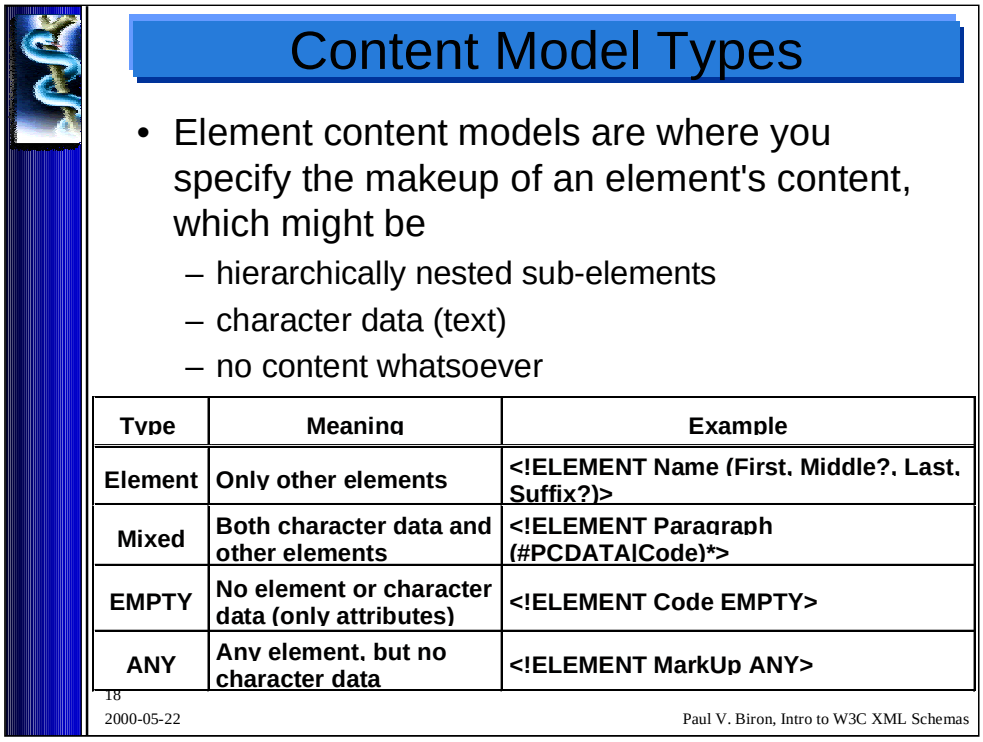
This slide shows both the productions from the XML Specification and an real-world example of the how those productions are used in the prolog and/or element content (in a DTD and/or instance document). Note, the terminology used in the *instance* cases differs from that used in the *productions* and elsewhere in the XML Specification.

Each XML document contains one or more elements, the boundaries of which are either delimited by start-tags and end-tags, or, for empty elements, by an empty-element tag. Each element has a type, identified by name, sometimes called its "generic identifier" (GI), and may have a set of attribute specifications. Each attribute specification has a name and a value.

The beginning of every non-empty XML element is marked by a start-tag. The Name in the start- and end-tags gives the element's type. The end of every element that begins with a start-tag must be marked by an end-tag containing a name that echoes the element's type as given in the start-tag.

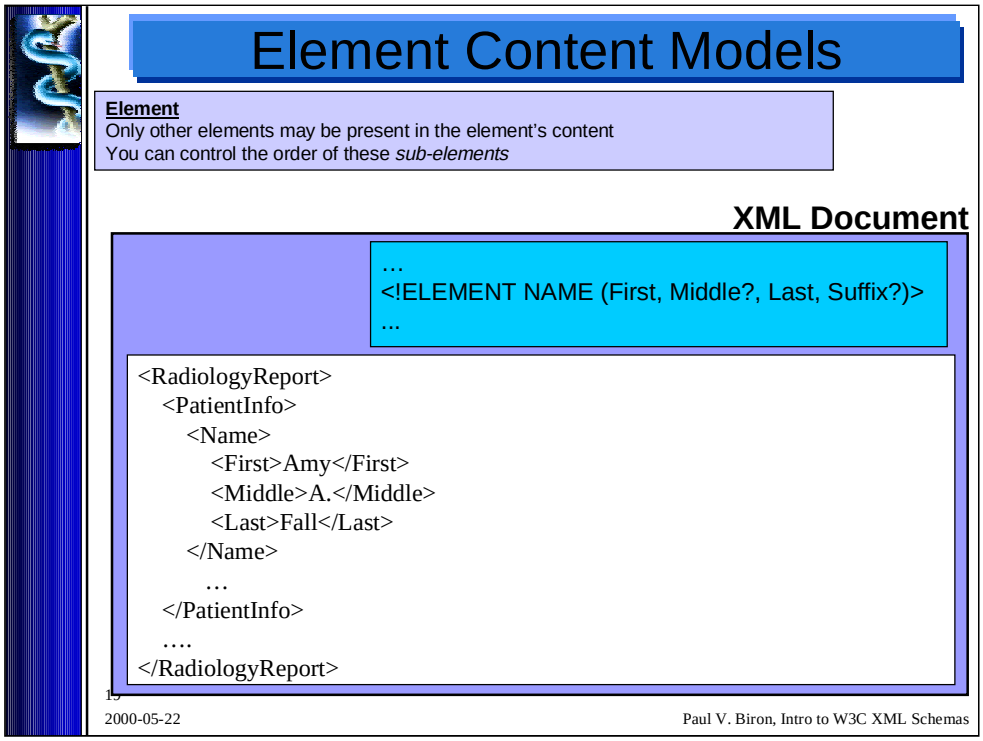
If an element is empty, it must be represented either by a start-tag immediately followed by an end-tag or by an empty-element tag. An empty-element tag takes a special form.

The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element's content. Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.



The element structure of an XML document may, for validation purposes, be constrained using element type and attribute-list declarations. An element type declaration constrains the element’s content.

Element type declarations often constrain which element types can appear as children of the element. At user option, an XML processor may issue a warning when a declaration mentions an element type for which no declaration is provided, but this is not an error.



An element type has ***element*** content when elements of that type may contain only child elements. The types, order, and number of occurrences of the child elements may all be constrained.

[45] elementdecl ::= '<!ELEMENT' S Name S contentspec S? '>'

[46] contentspec ::= 'EMPTY' | 'ANY' | Mixed | children

[47] children ::= (choice | seq) ('?' | '*' | '+')?

[48] cp ::= (Name | choice | seq) ('?' | '*' | '+')?

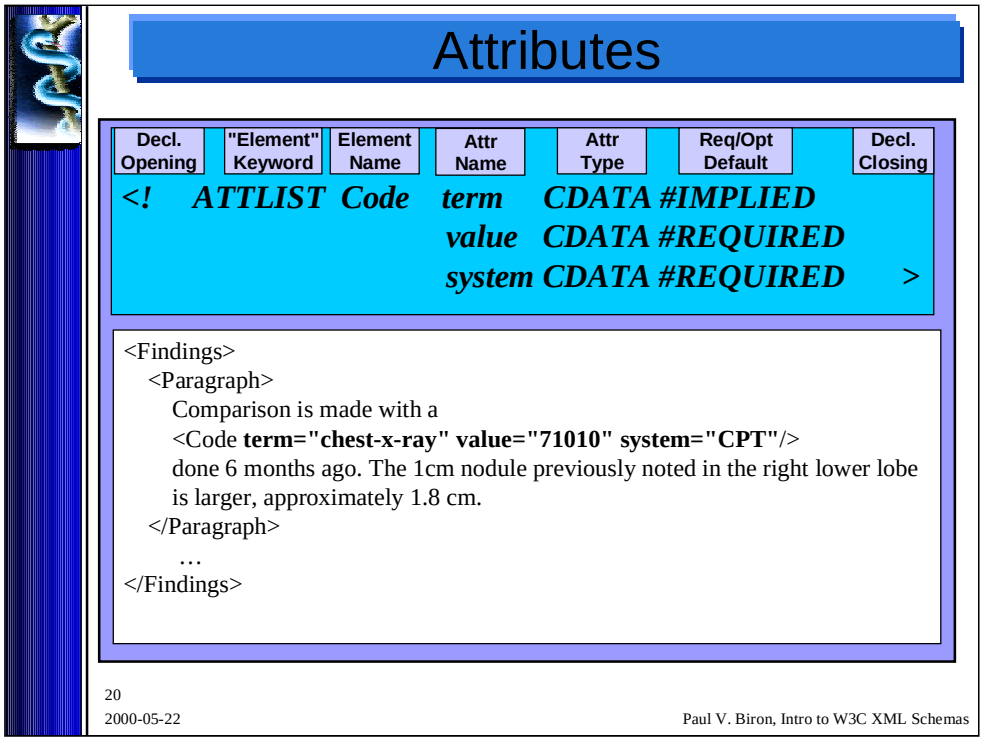
[49] choice ::= '(' S? cp (S? '|' S? cp)* S? ')'

[50] seq ::= '(' S? cp (S? ',' S? cp)* S? ')'

[4] NameChar ::= Letter | Digit | '.' | '-' | '_' | ':' | CombiningChar | Extender

[5] Name ::= (Letter | '_' | ':') (NameChar)*

where the Names give the types of elements that may appear as children.



[52] AttlistDecl ::= '<!ATTLIST' S Name AttDef* S? '>'
[53] AttDef ::= S Name S AttType S DefaultDecl

[41] Attribute ::= Name Eq AttValue

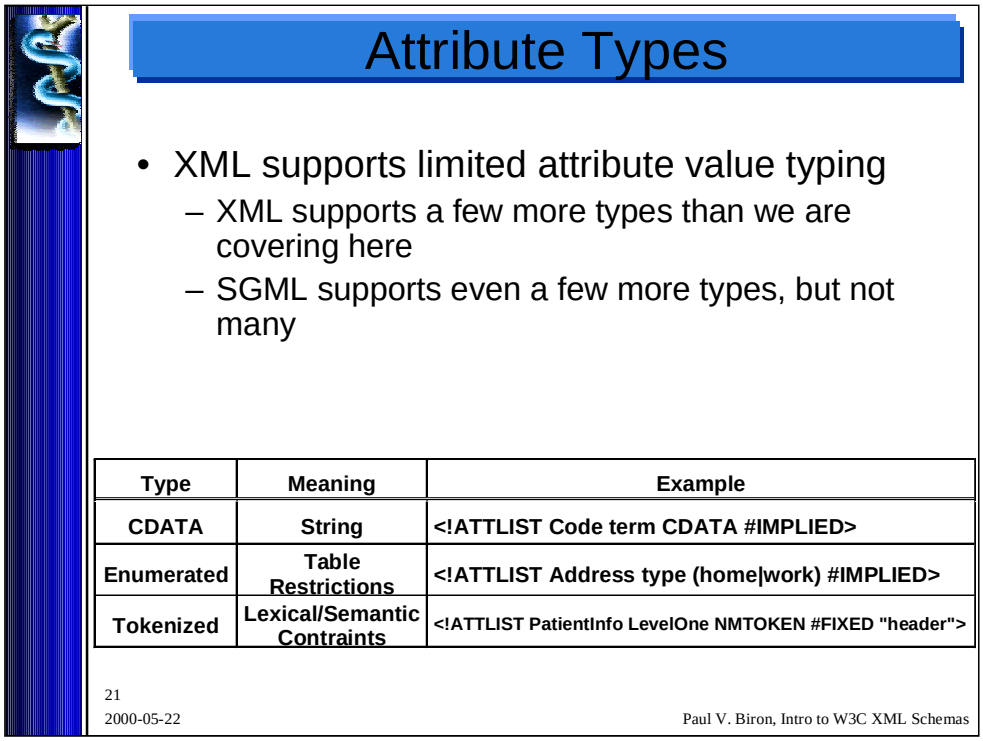
This slide shows both the productions from the XML Specification and a real-world example of the how those productions are used in the prolog and/or element content (in a DTD and/or instance document). Note, the terminology used in the *instance* cases differs from that used in the *productions* and elsewhere in the XML Specification.

Attributes are used to associate name-value pairs with elements. The Name-AttValue pairs are referred to as the attribute specifications of the element, with the Name in each pair referred to as the attribute name and the content of the AttValue (the text between the ' or " delimiters) as the attribute value. Attribute specifications may appear only within start-tags and empty-element tags.

Attribute-list declarations may be used:

- To define the set of attributes pertaining to a given element type.
- To establish type constraints for these attributes.
- To provide default values for attributes.

Attribute-list declarations specify the name, data type, and default value (if any) of each attribute associated with a given element type.



XML attribute types are of three kinds: a string type, a set of tokenized types, and enumerated types. The string type may take any literal string as a value; the tokenized types have varying lexical and semantic constraints, as noted:

[54] AttType ::= StringType | TokenizedType | EnumeratedType

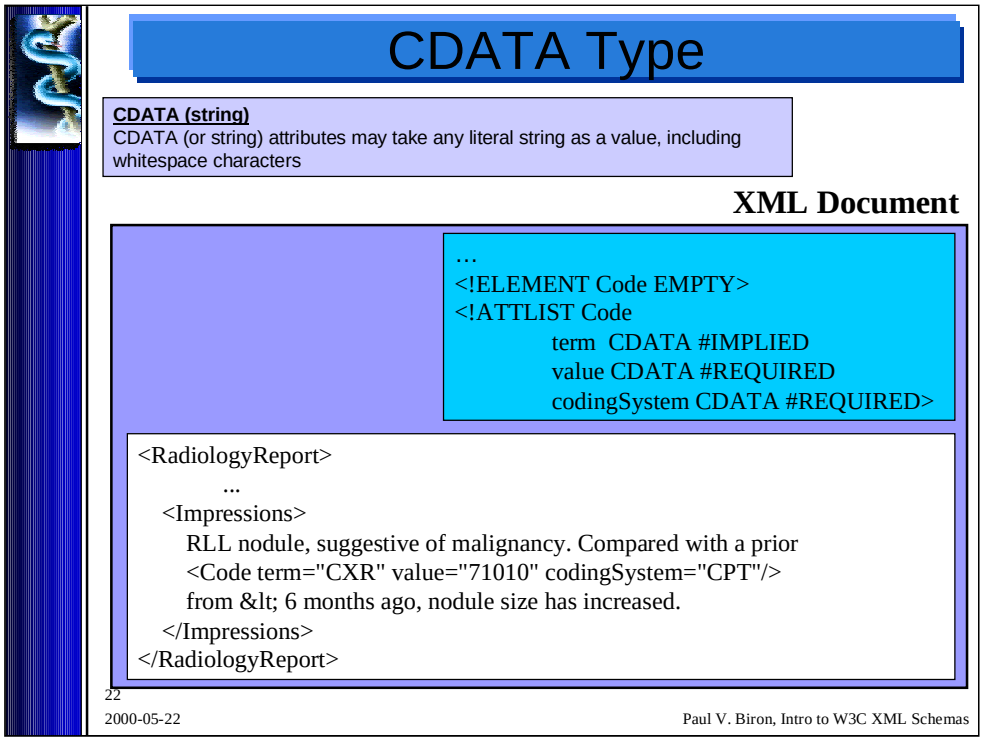
[55] StringType ::= 'CDATA'

[56] TokenizedType ::= 'ID' | 'IDREF' | 'IDREFS' | 'ENTITY' | 'ENTITIES' | 'NMTOKEN' | 'NMTOKENS'

[57] EnumeratedType ::= NotationType | Enumeration

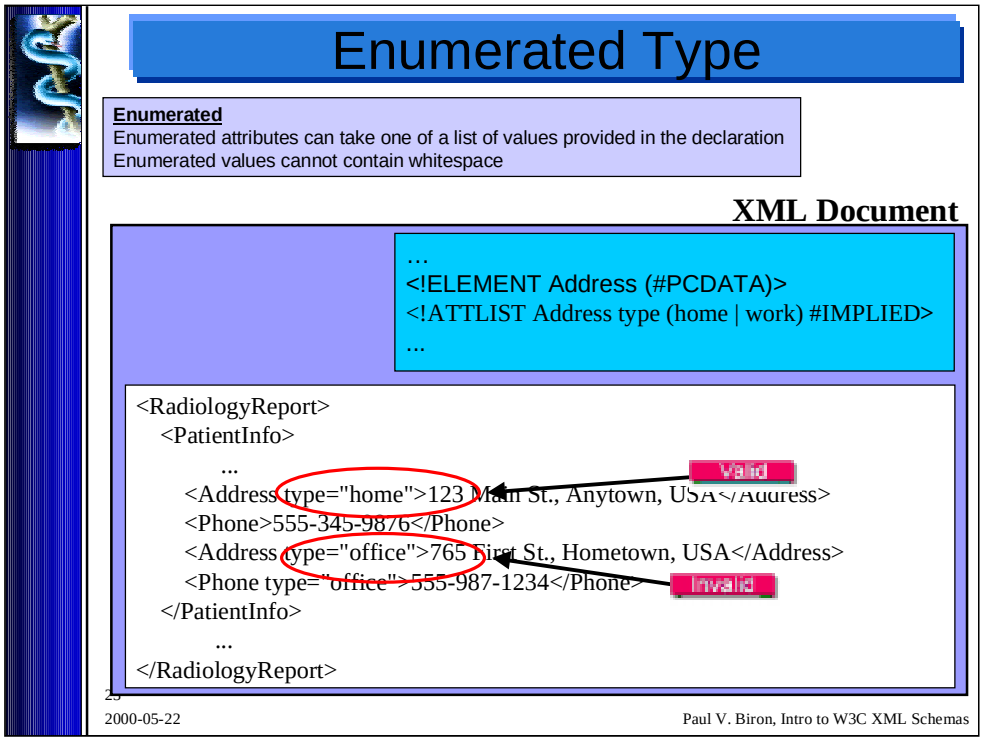
[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'

[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'



CDATA stands for ***character data***.

As an attribute type, it is analogous to the #PCDATA element type.



Enumerated attributes can take one of a list of values provided in the declaration. There are two kinds of enumerated types:

[57] EnumeratedType ::= NotationType | Enumeration
[58] NotationType ::= 'NOTATION' S '(' S? Name (S? '|' S? Name)* S? ')'
[59] Enumeration ::= '(' S? Nmtoken (S? '|' S? Nmtoken)* S? ')'

A NOTATION attribute identifies a notation, declared in the DTD with associated system and/or public identifiers, to be used in interpreting the element to which the attribute is attached.

[Note, we will not be discussing NOTATION attributes during this tutorial]

Intro

- Background
  - XML, DTDs, other schema languages, namespaces
- Goals of XML Schema
- Intro to XML Schema
  - Simple Types (Datatypes) and Complex Types (Structural Types)
  - Element and Attribute Declarations
- Advanced Topics
  - Restriction, extension, Namespaces, schema composition and access, local element declarations

What's Wrong with DTDs

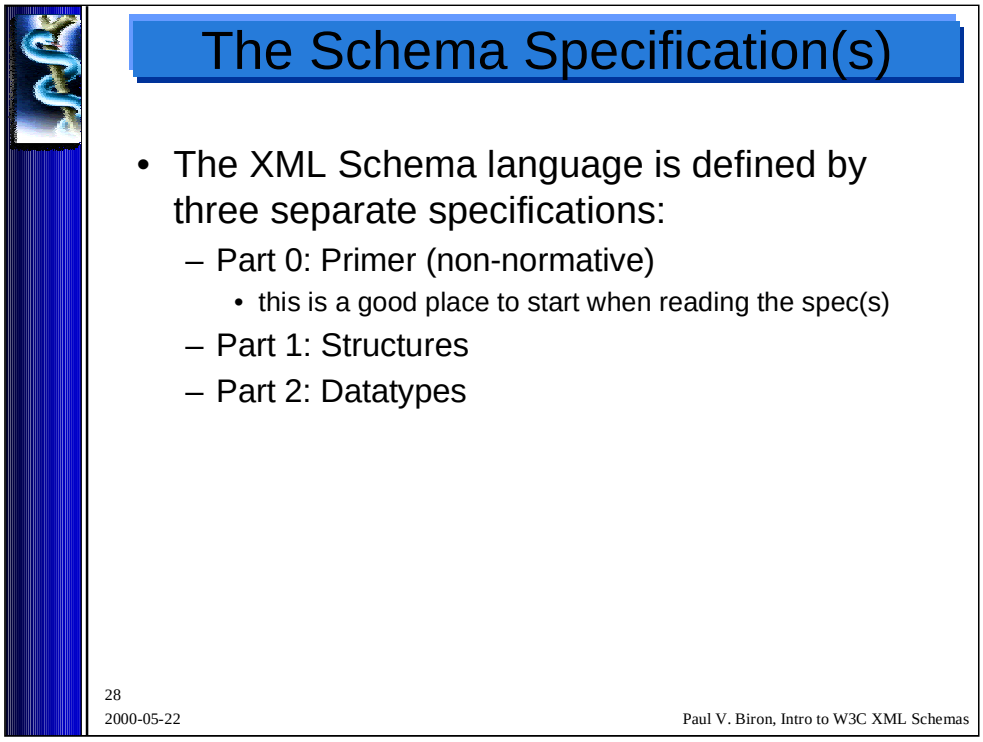
- Special ad hoc notation
- no datatypes (well, hardly)
- Complex to extend (risk of rigidity)
- Do not play well with namespaces
- No formal role for documentation

Goals of XML Schema

- Specific goals beyond DTD functionality are
  - integration with namespaces
  - incomplete constraints on the content of an element
  - integration with primitive data types
  - inheritance: existing mechanisms use content models to specify part-of relations and only specify kind-of relations implicitly or informally. Making kind-of relations explicit would make both understanding and maintenance easier.

Intro

- Background
  - XML, DTDs, other schema languages, namespaces
- Goals of XML Schema
- Intro to XML Schema
  - Simple Types (Datatypes) and Complex Types (Structural Types)
  - Element and Attribute Declarations
- Advanced Topics
  - Restriction, extension, Namespaces, schema composition and access, local element declarations



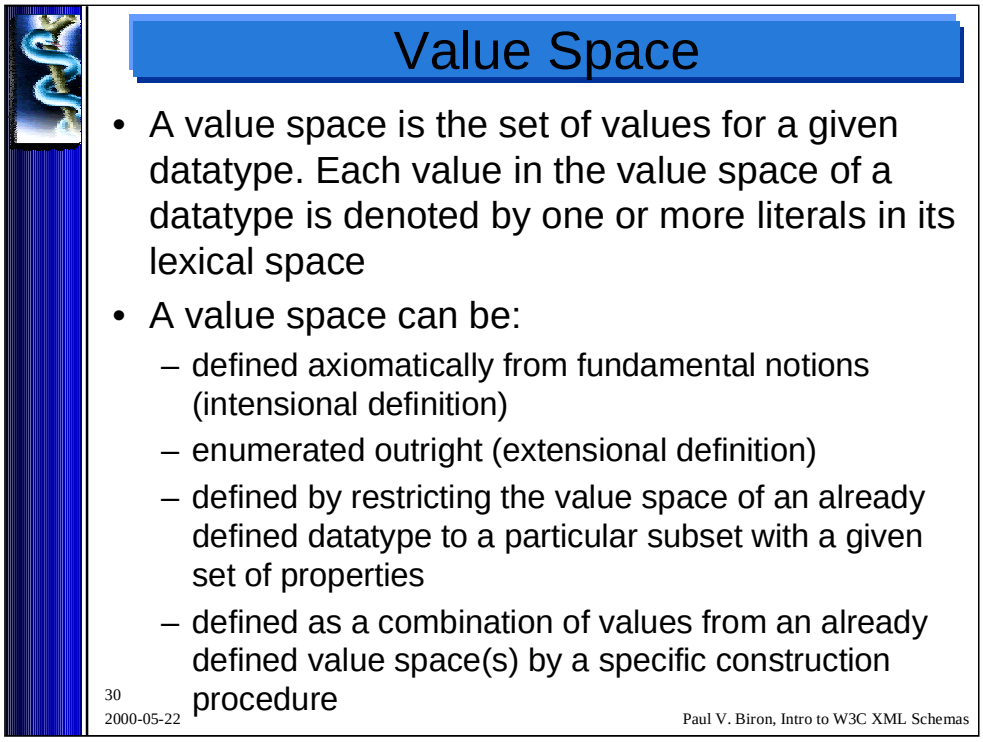
"XML Schema Part 0: Primer" is a non-normative document intended to provide an easily readable description of the XML Schema facilities and is oriented towards quickly understanding how to create schemas using the XML Schema language. XML Schema Part 1: Structures and XML Schema Part 2: Datatypes provide the complete normative description of the XML Schema definition language, and the primer describes the language features through numerous examples which are complemented by extensive references to the normative texts.

The purpose of "XML Schema Part 1: Structures" is to provide an inventory of XML markup constructs with which to write schemas, that is, to define and describe a class of XML documents by using these constructs to constrain and document the meaning, usage and relationships of their constituent parts: datatypes, elements and their content, attributes and their values, entities and their contents and notations.

"XML Schema Part 2: Datatypes" discusses datatypes that can be used in an XML Schema. These datatypes can be specified for element content that would be specified as #PCDATA and attribute values of various types in a DTD. It is the intension of the specification that it be usable outside of the context of XML Schemas for a wide range of other XML-related activities such as XSL and RDF Schema.

Datatypes

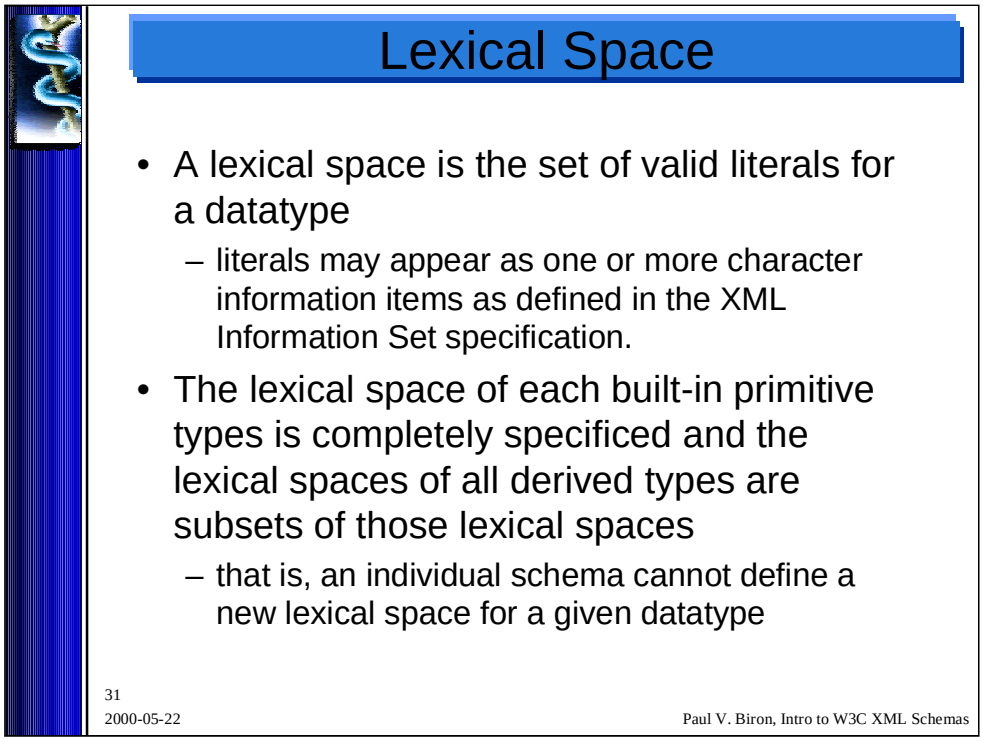
- A datatype is a 3-tuple, consisting of
  - a set of distinct values, called its **value space**
  - a set of lexical representations, called its **lexical space**
  - a set of **facets** that characterize properties of the value space, individual values or lexical items.
- Used to type attribute values and PCDATA content of elements



Value spaces defined "axiomatically" are those of the primitive, built-in datatypes.

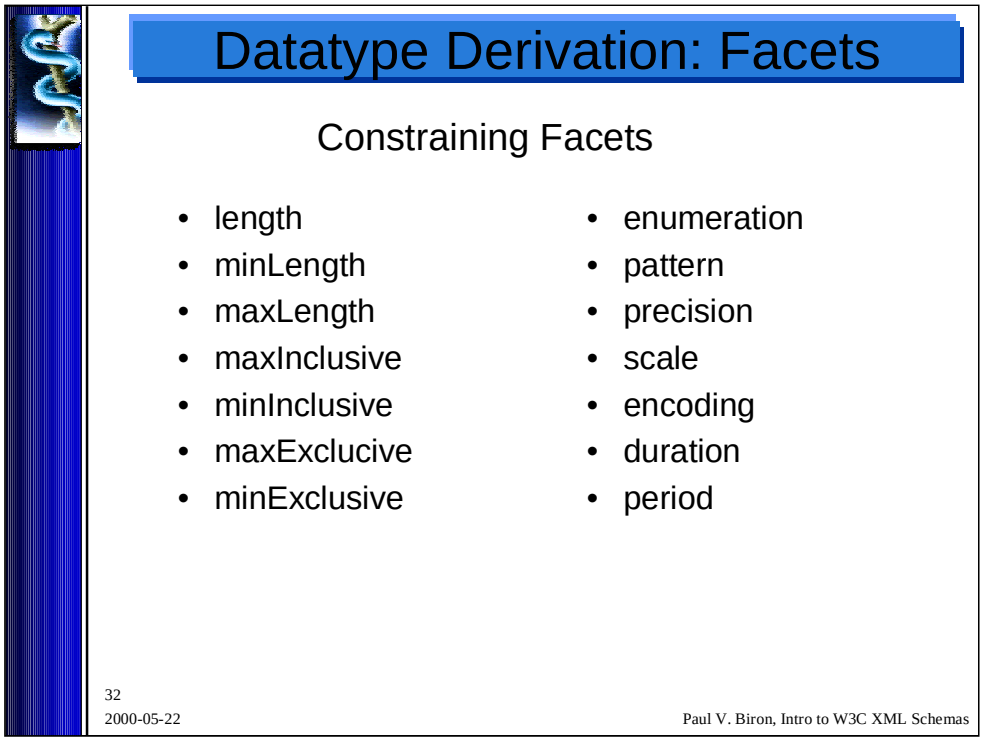
Value spaces defined by restricting the value space of another datatype are those of the derived types (both built-in and user-derived types).

Value spaces defined by combining values are those of the "list" types (list is a form of derivation, more later). Note: hopefully, a future version of the specification will allow the explicit definition of other forms of "combination" (or aggregate) datatypes, such as matricies (lists of lists), sets and records.



While it would be extremely useful to allow schema authors to define their own lexical spaces, it was felt that for the first version of the schema language we would not allow this for (at least) the following reasons:

1. To keep the specification as simple as possible
2. To insure interoperability
   a. The best method of associating individual literals in the "user-defined" lexical space has yet to be determined
   b. Until such time, it is unclear how a general purpose schema processor would know how to datatype validate an instance governed by such a schema.

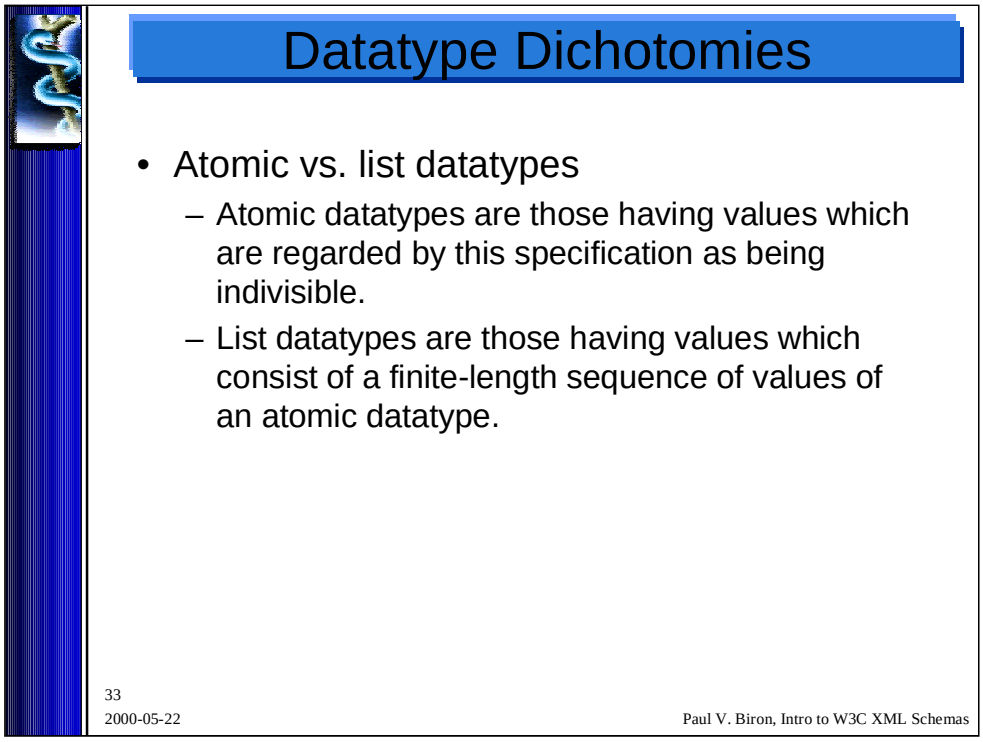


The constraining facets of a datatype serve to distinguish those aspects of one datatype which differ from other datatypes. Rather than being defined solely in terms of a prose description the datatypes in this specification are defined in terms of the synthesis of facet values which together determine the value space and properties of the datatype. A constraining facet is an optional property that can be applied to a datatype to constrain its value space.

Each constraining facet (hereafter, referred to simply as facets) is applicable to a particular primitive datatype; some facets are applicable to multiple datatypes. Derived datatypes "inherit" all of the facets of their primitive ancestor.

There is not enough time/space in this tutorial to individually cover all of the facets in detail; we'll cover some of the more useful ones in due course.

Note: enumeration is not really a facet and *should* be considered a form of derivation, but I lost that editorial battle within the Schema WG.

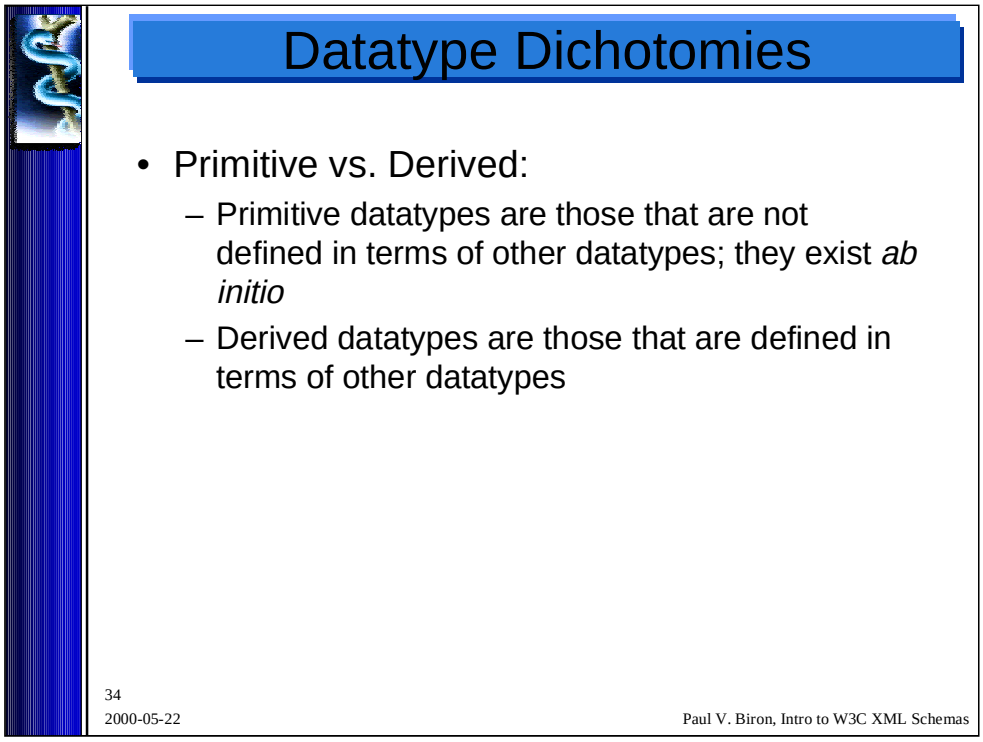


atomic datatypes may be either primitive or derived. The value space of an atomic datatype is a set of "atomic" values, which for the purposes of this specification, are not further decomposable. The lexical space of an atomic datatype is a set of literals whose internal structure is specific to the datatype in question and completely defined in the schema language.

list datatypes are always derived. The value space of a list datatype is a set of finite-length sequences of atomic values. The lexical space of a list datatype is a set of literals whose internal structure is a whitespace separated sequence of literals of the atomic datatype of the items in the list.

.

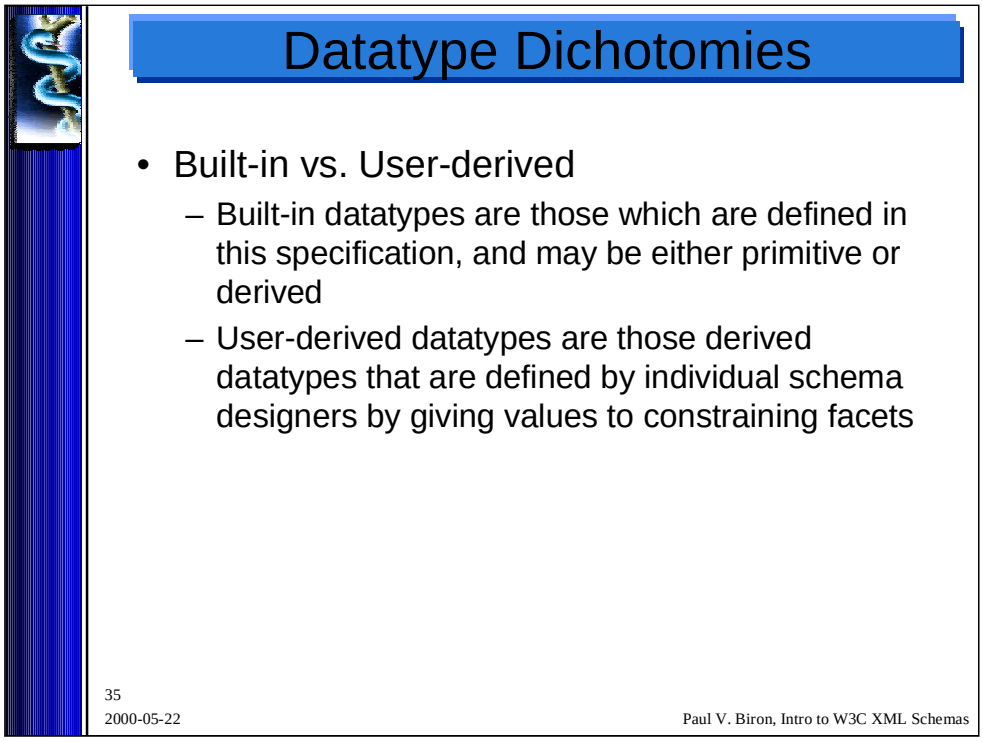
Note: Other type systems (including V3 Datatypes) treat list datatypes as special cases of the more general notions of aggregate or collection datatypes.



For example, a float is a well-defined mathematical concept that cannot be defined in terms of other datatypes, while a date is a special case of the more general datatype recurringDuration.

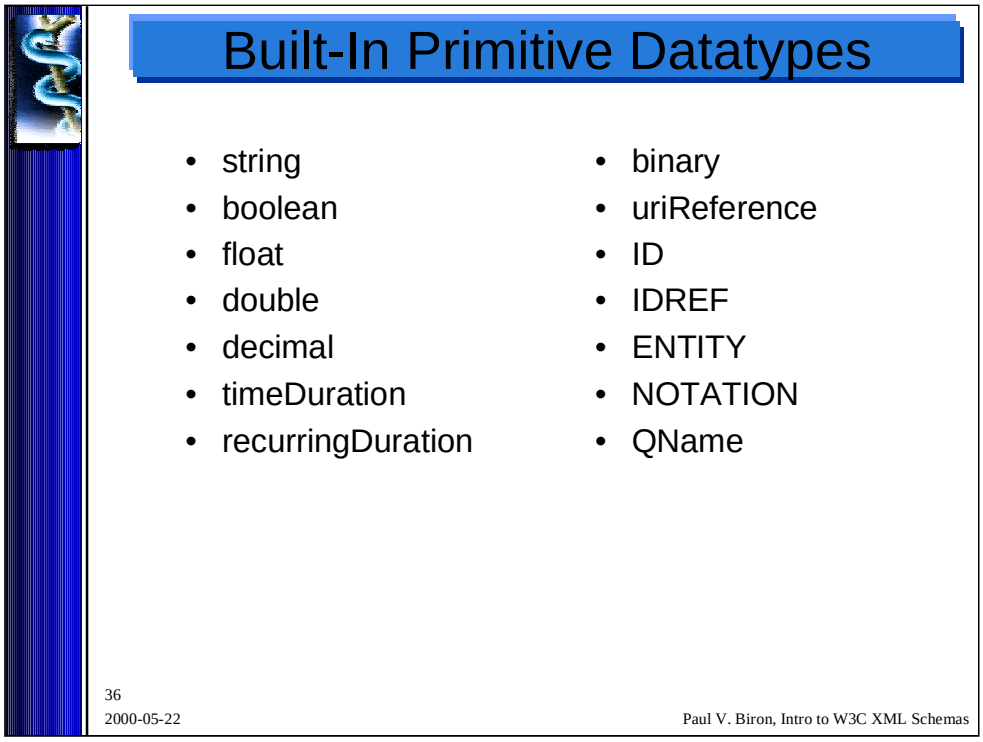
The datatypes defined by the schema language fall into both the primitive and derived categories. It is felt that a judiciously chosen set of primitive datatypes will serve the widest possible audience by providing a set of convenient datatypes that can be used as is, as well as providing a rich enough base from which the variety of datatypes needed by schema designers can be derived.

Every derived datatype is defined in terms of an existing datatype, referred to as the base type. base types may be either primitive or derived.



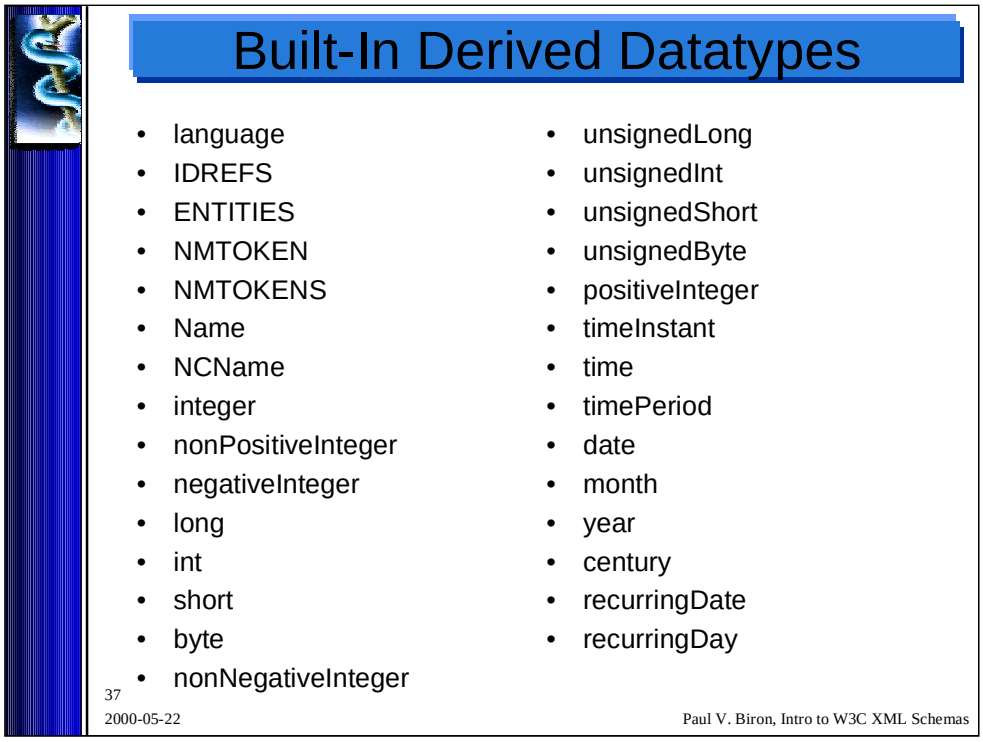
Conceptually there is no difference between the built-in derived datatypes included in the schema language and the user-derived datatypes which will be created by individual schema authors. The built-in derived datatypes are those which are believed to be so common that if they were not defined in this specification many schema authors would end up "reinventing" them.

Furthermore, including these derived datatypes in language serves to demonstrate the mechanics and utility of the datatype generation facilities of this specification, as well as providing a "santity check" that the language includes those features necessary to define the datatypes which are most useful to the widest possible audience.



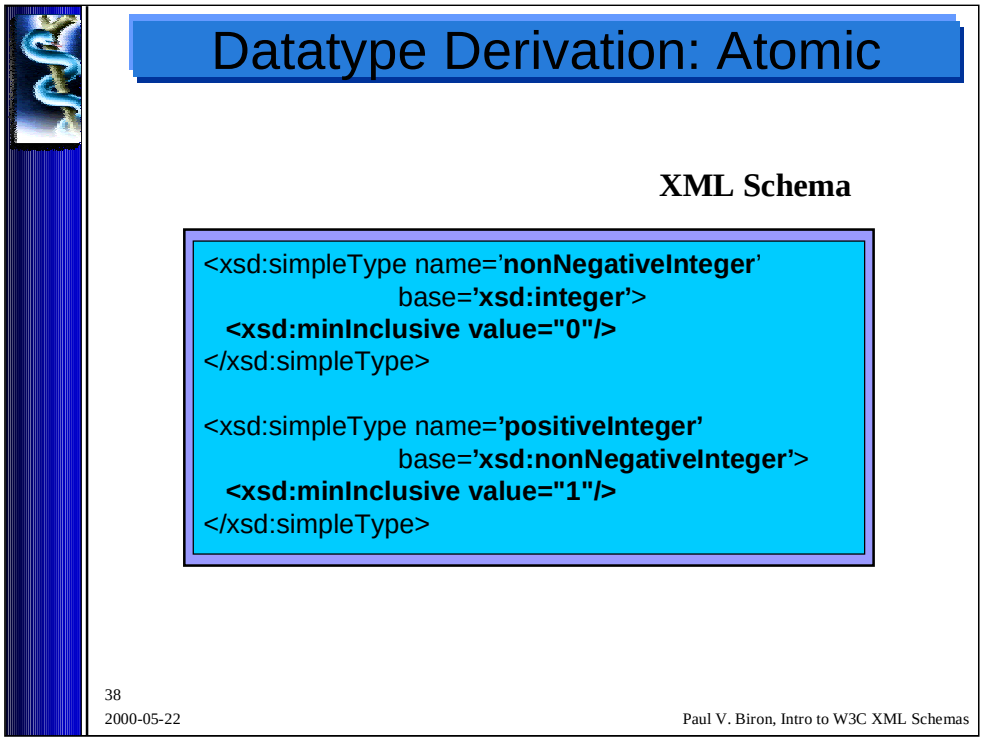
• Primitive datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*.

• Derived datatypes are those that are defined in terms of other datatypes.



• Primitive datatypes are those that are not defined in terms of other datatypes; they exist *ab initio*.

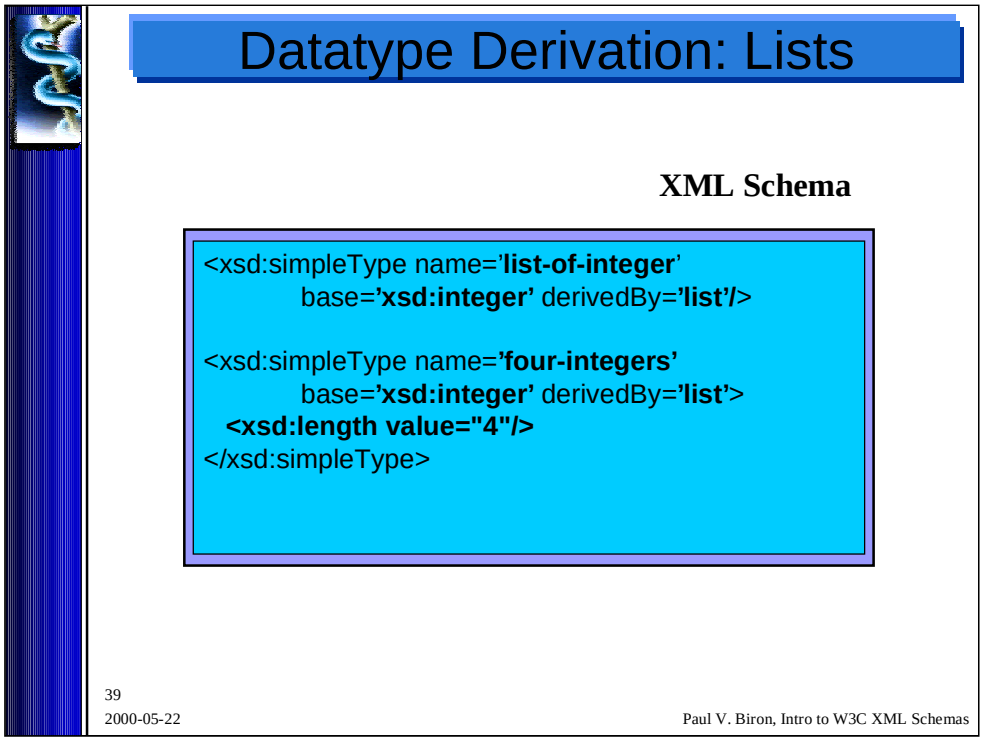
• Derived datatypes are those that are defined in terms of other datatypes.



Alternatively, the above types could have been defined using the minExclusive facet, as follows:

```
<xsd:simpleType name='nonNegativeInteger' base='xsd:integer'>
  <xsd:minExclusive value="-1"/>
</xsd:simpleType>

<xsd:simpleType name='positiveInteger' base='xsd:nonNegativeInteger'>
  <xsd:minExclusive value="0"/>
</xsd:simpleType>
```

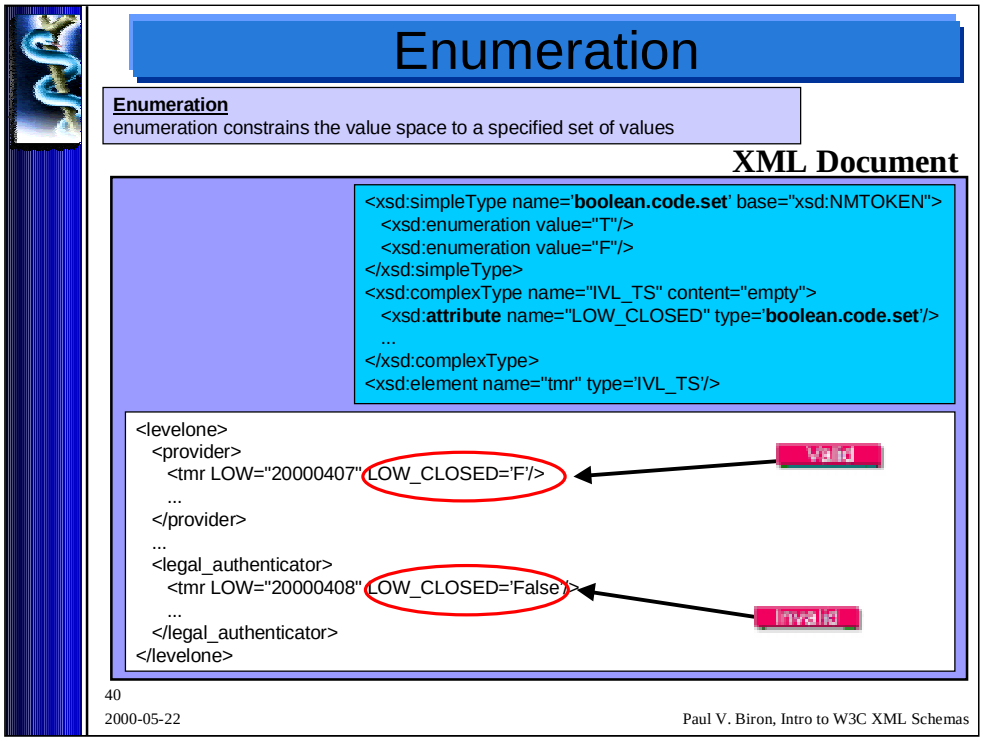


A list dataype must be derived from an atomic datatype. This yields a list datatype that can contain whitespace separated lists of values of the base type.

When a datatype is derived by list, the following constraining facets may be used:

- length
- maxLength
- minLength
- enumeration

For length, maxLenth and minLength, the unit of length is measured in number of list items.



The simpleType defn is from line 64 of v3dt.xsd

The complexType defn is from line 829 of v3dt.xsd

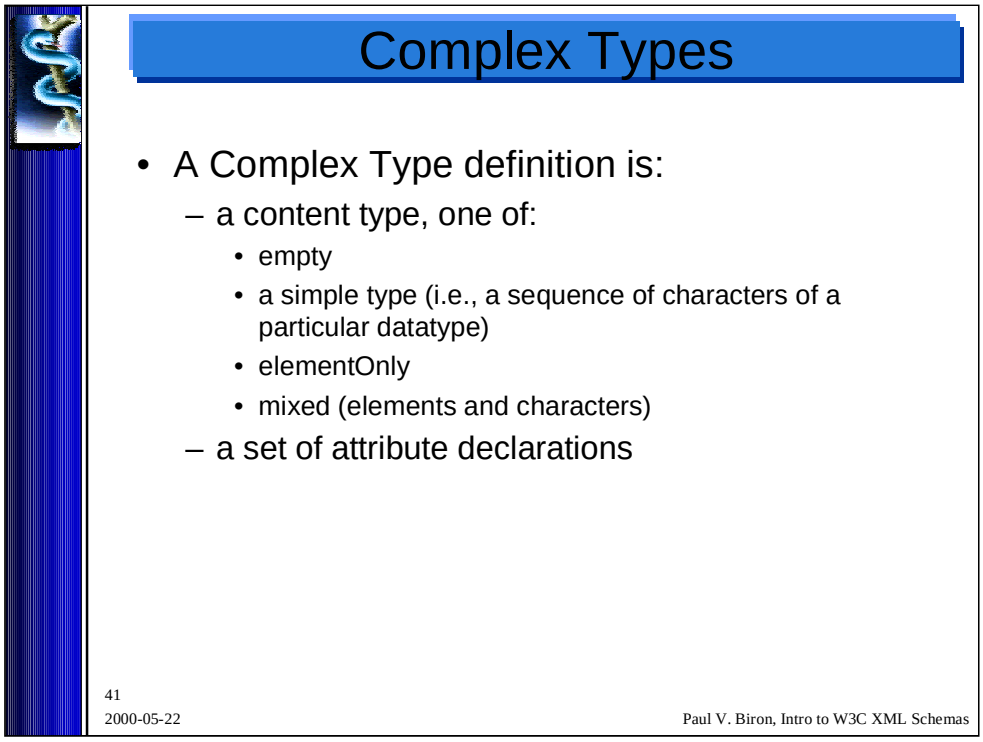
The element decl is from line 662 (modified slightly)

The valid occurrence is from line 97 of levelone.xml

The invalid occurrence is from line 49 of levelone.xml

The above example is equivalent to the following DTD fragment:

```
<!ELEMENT tmr EMPTY>
<!ATTLIST tmr
    LOW_CLOSED (T | F) #IMPLIED
    ...>
```



There is a basic difference between complex types which allow elements in their content and may carry attributes, and simple types (or datatypes) which cannot have element content and cannot carry attributes.

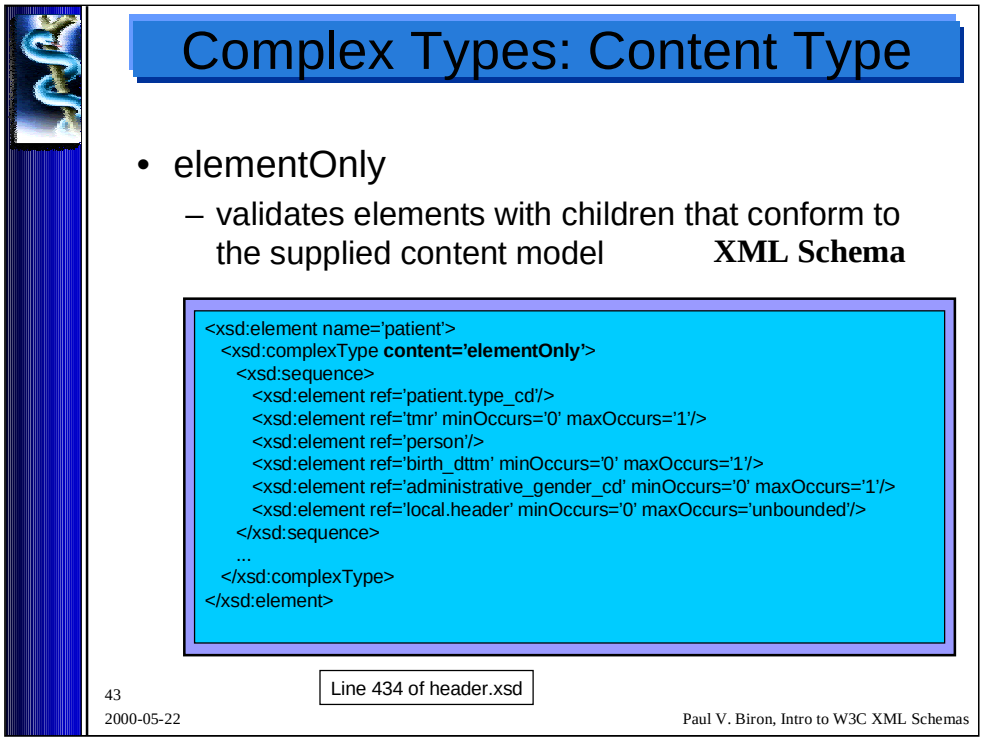
Complex Types: Content Type

- empty
  - validates elements with no character or element information item children

XML Schema

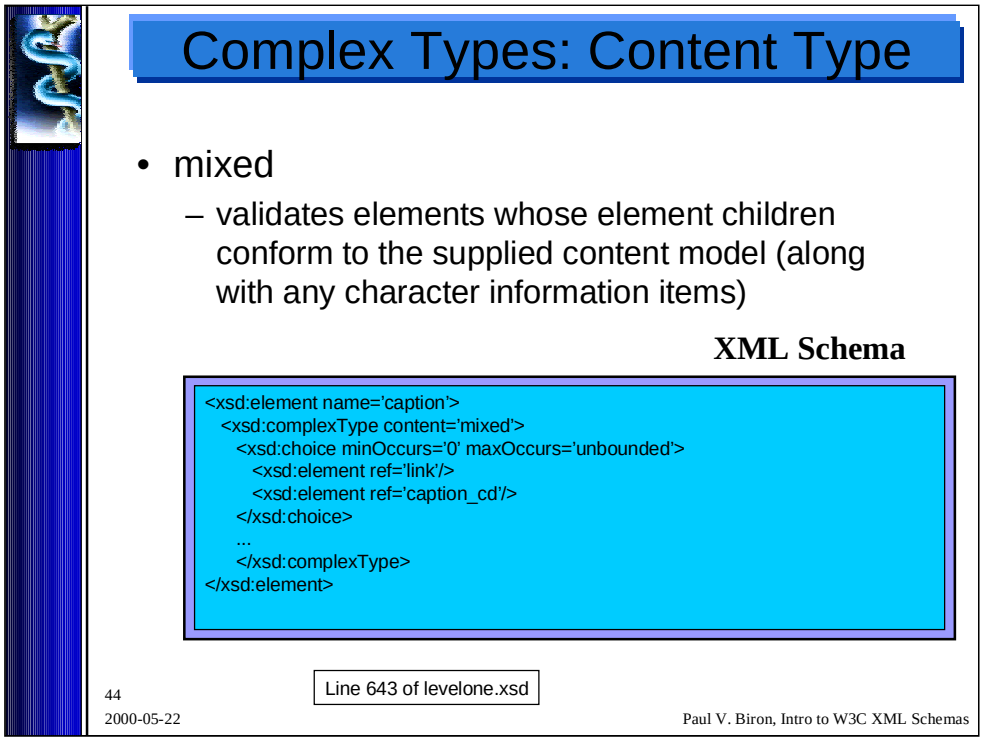
```
<xsd:complexType name='INT' content="empty">
  <xsd:attribute name="V" type="xsd:integer"/>
  <xsd:attribute name="V-T" type="xsd:string"
                                use="fixed" value="xsd:integer"/>
  ...
</xsd:complexType>
<xsd:element name="version_nbr" type='INT'/>
```

Line 786 of v3dt.xsd
Line 691 of header.xsd



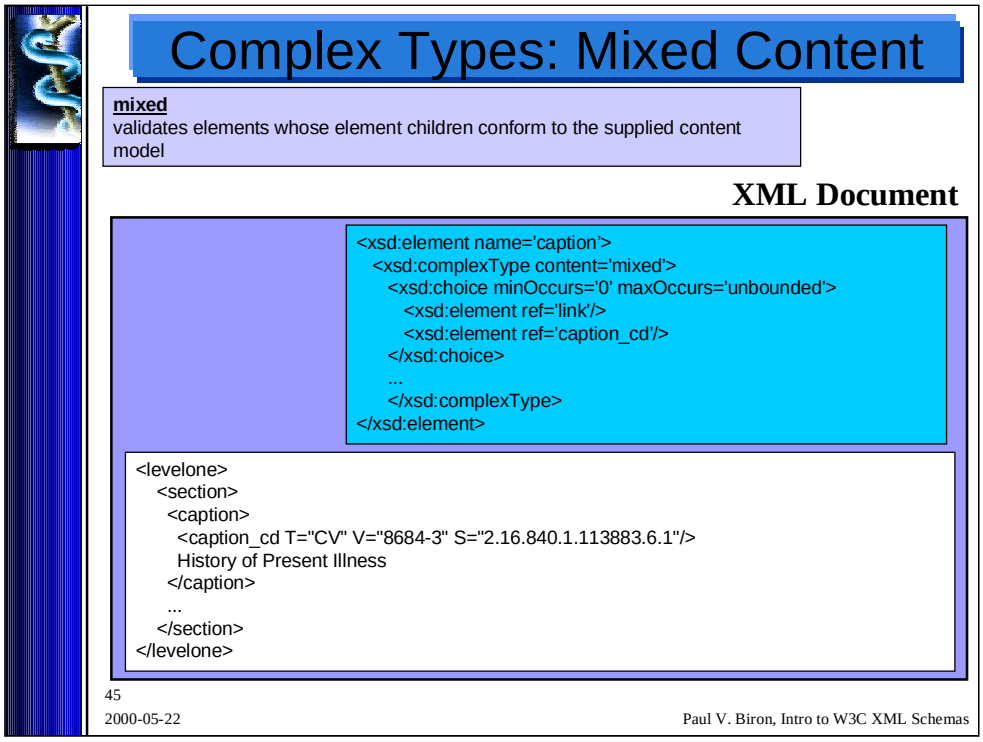
The above element declaration is equivalent to the following declaration in a DTD:

```
<!ELEMENT patient (patient.type_cd, tmr?, person, birth_dttm?,
    administrative_gender_cd?, local.header*)>
```



The above element declaration is equivalent to the following declaration in a DTD:

<!ELEMENT caption (#PCDATA | link | caption_cd)*>



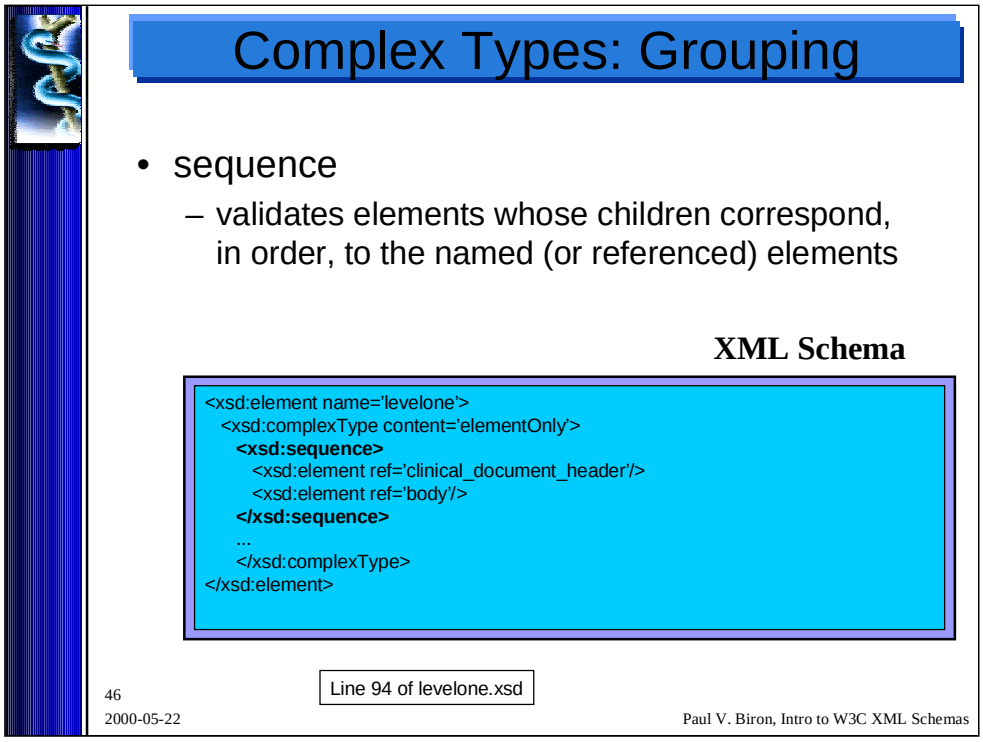
The valid instance is from line 141 of levelone.xml.

Unlike mixed content models in DTDs, the order and cardinality of element children in mixed content models in XML Schema is significant. That is, that in order to imitate the DTD mixed declaration from the previous slide that we need to explicitly note that there is an optionally repeating <xsd:choice> surrounding the two element references in the complex type.

If the complexType had looked like:

```
<xsd:element name='caption'>
    <xsd:complexType content='mixed'>
    <xsd:element ref='link'/>
    <xsd:element ref='caption_cd'/>
        ...
    </xsd:complexType>
</xsd:element>
```

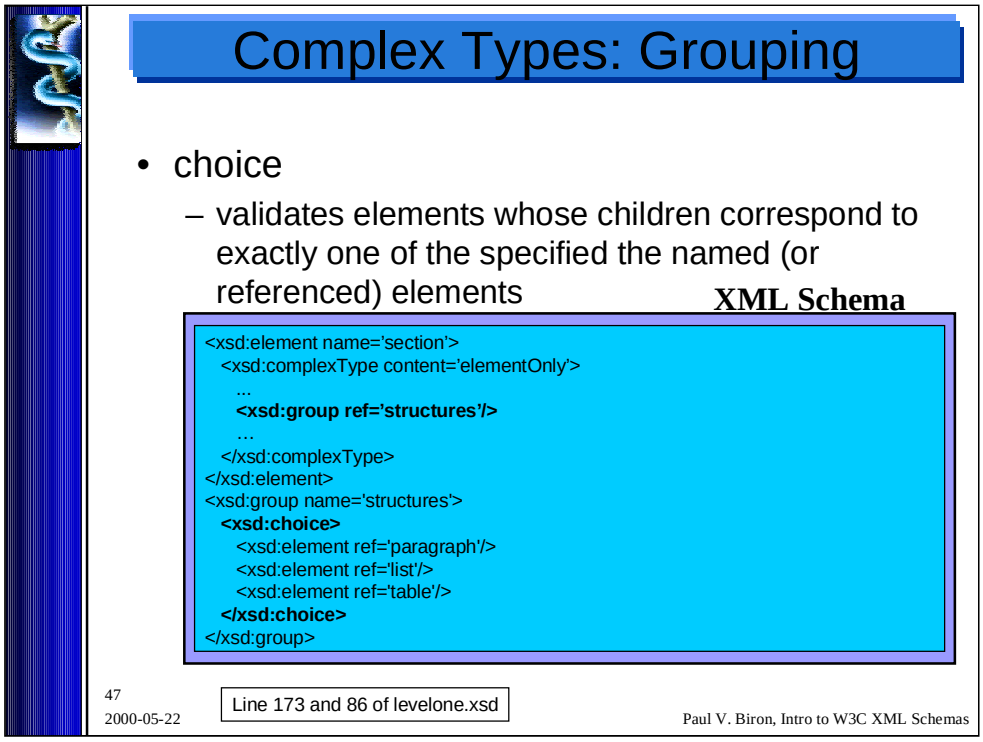
we would still have a mixed content model (that is, <caption> could contain text mixed in with <link> and <caption_cd> elements), however the element children would have to match (in order and cardinality) exactly with what is specified in the content model.



XML Schema enables unnamed groups of elements to be defined and constrained to appear in the same order (sequence) as they are declared.

The above element declaration is equivalent to the following declaration in a DTD:

```
<!ELEMENT levelone (clinical_document_header, body)>
```



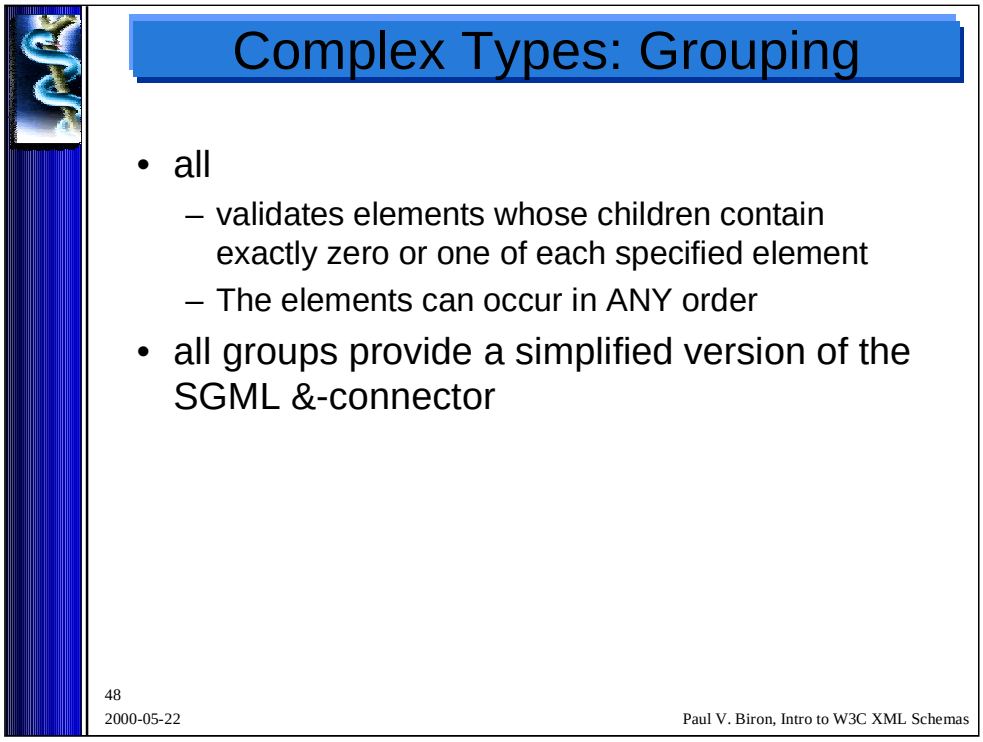
A choice group element allows only one of its children to appear in an instance.

The above element declaration is equivalent to the following declaration in a DTD:

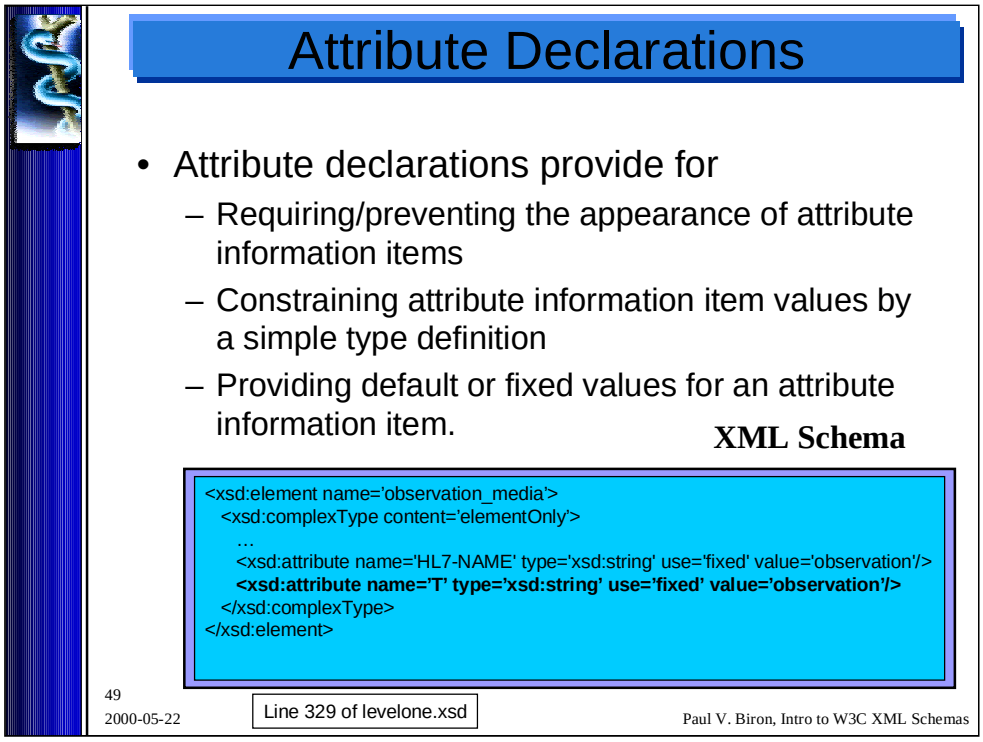
```
<!ELEMENT section (… (list | paragraph | table) …)>
```

Named groups of elements can also defined and later referenced from a content model. Named groups can be used to build up the content models of complex types (thus mimicking common usage of parameter entities in XML 1.0). The way the above is actually specified in the PRA DTDs is as follows:

```
<!ENTITY % structures "paragraph | list | table">
<!ELEMENT section (caption?,(%structures; | section)*)>
```



Since **all** groups are not expressible in DTDs, there are no simple examples in the PRA schemas of this construct and I have not found the time to sufficiently examine the inherent structure of these DTDs enough to know whether we *should* be using **all** groups anywhere.

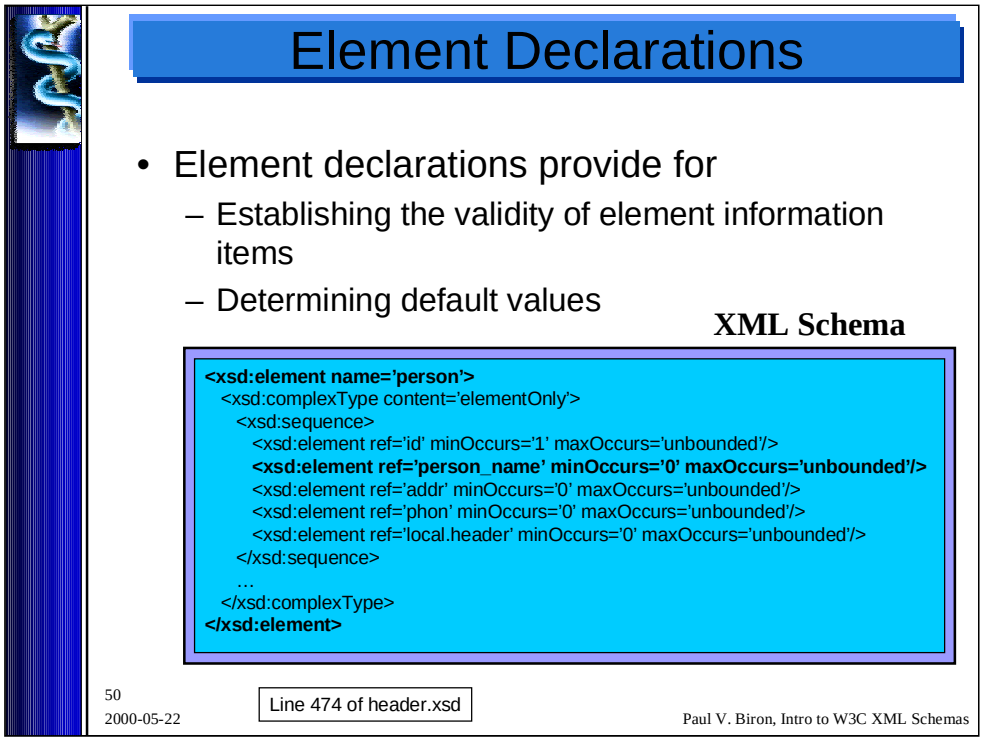


An attribute declaration is an association between a name and a simple type (datatype) definition, together with occurrence information and (optionally) a default value. The association is either global, or local to its containing complex type definition. Attribute declarations contribute to schema-validity as part of complex type definition validation, when their occurrence, defaults and type components are checked against an attribute information item with a matching name and namespace.

The occurrence of an attribute is controlled by the **use** attribute, whose value may be one of: fixed | default | required | optional. If not specified, optional is assumed. If either fixed or default is specified, then the **value** attribute must also be present and its value specifies the fixed or default attribute value.

The above attribute declaration is equivalent to the following declaration in a DTD:

```
<!ELEMENT observation_media (...)>
<!ATTLIST observation_media
    ...
    HL7-NAME CDATA #FIXED 'observation'
    T CDATA #FIXED 'observation'>
```



An element declaration is an association of a name with a type definition, either simple or complex and an (optional) default value. The association is either global or scoped to a containing complex type definition. A global element declaration with name 'A' is broadly comparable to a pair of DTD declarations as follows, where the associated type definition fills in the ellipsis:

<!ELEMENT A . . .>

<!ATTLIST A . . .>

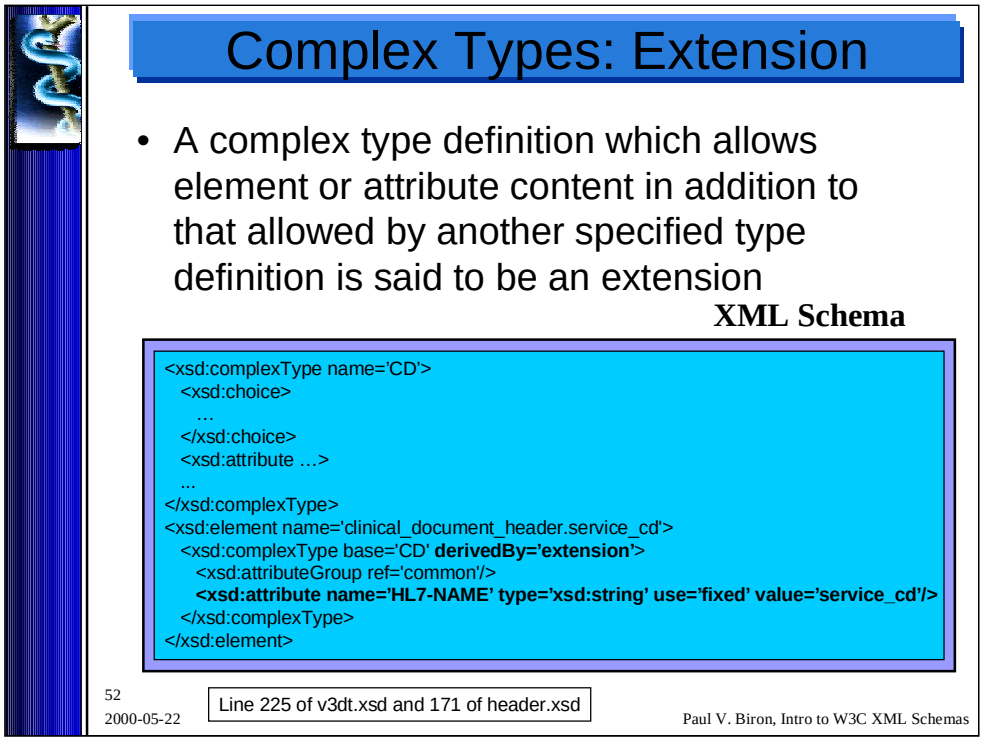
(We'll talk about locally-scoped element declarations later).

Within a complexType definition, **global** element declarations are referred to by naming them with the **ref** attribute of <xsd:element>

Unlike with DTDs, XML Schema provides for the ability to give a default for elements whose content model contains only PCDATA (i.e., those whose type is a datatype). As with **all** groups, there are no simple examples in the PRA schemas of this construct and I have not found the time to sufficiently examine the inherent structure of these DTDs enough to know whether we *should* be using element default values anywhere.

Intro

- Background
  - XML, DTDs, other schema languages, namespaces
- Goals of XML Schema
- Intro to XML Schema
  - Simple Types (Datatypes) and Complex Types (Structural Types)
  - Element and Attribute Declarations
- Advanced Topics
  - Restriction, extension, Namespaces, schema composition and access, local element declarations

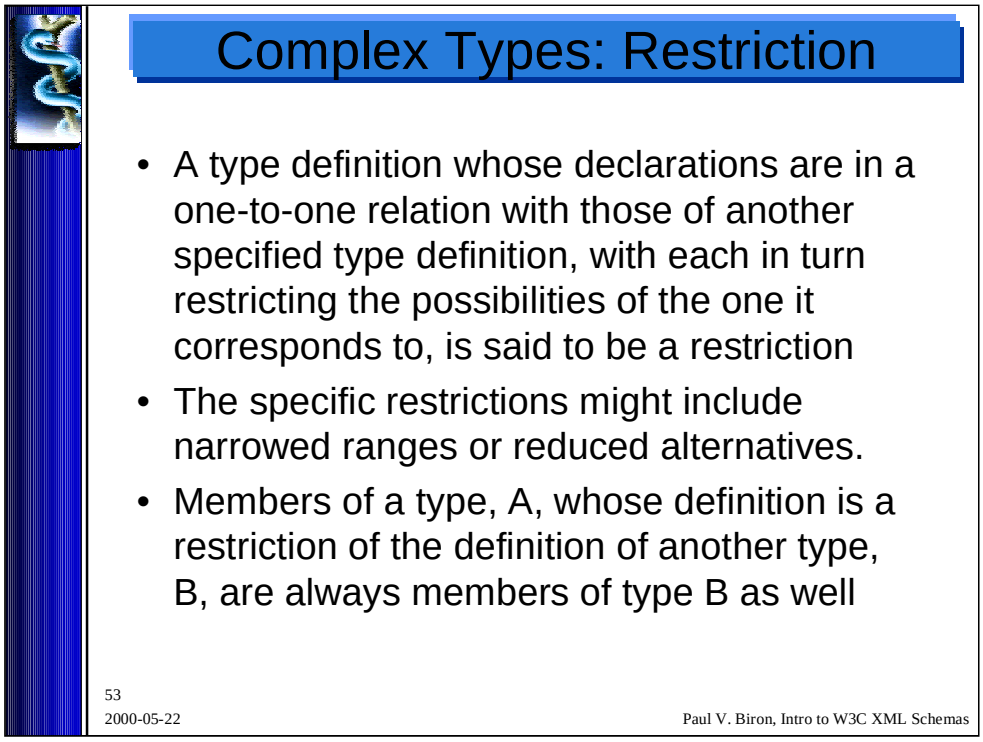


A complex type which **extends** another does so by having additional content model particles at the end of the other definition's content model, or by having additional attribute declarations, or both.

Note, XML Schema allows only appending, and not other kinds of extensions. This decision simplifies application processing required to cast instances from derived to base type. Future versions may allow more kinds of extension, requiring more complex transformations to effect casting.

A type definition used as the basis for an extension or restriction is known as the **base type** definition of that definition.

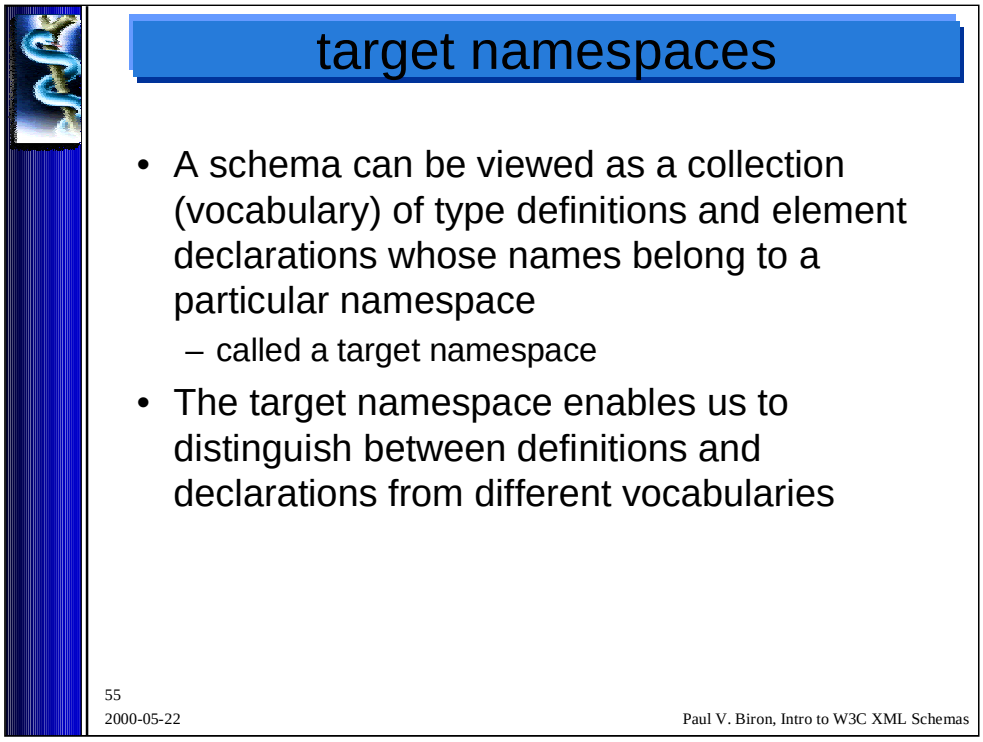
A distinguished **ur-type** definition is present in each XML Schema, serving as the root of the type definition hierarchy for that schema. The ur-type definition has the unique characteristic that it can function as a complex or a simple type definition, according to context.



In addition to deriving new complex types by extending content models, it is also possible to derive new types by restricting the content models of existing types. Restriction of complex types is conceptually the same as restriction of simple types, except that the restriction of complex types involves a type's declarations rather than the acceptable range of a simple type's values. A complex type derived by restriction is very similar to its base type, except that its declarations are more limited than the corresponding declarations in the base type. In fact, the values represented by the new type are a subset of the values represented by the base type (as is also the case with restriction of simple types). In other words, an application prepared for the values of the base type would not be surprised by the values of the restricted type.

Schema Access & Composition

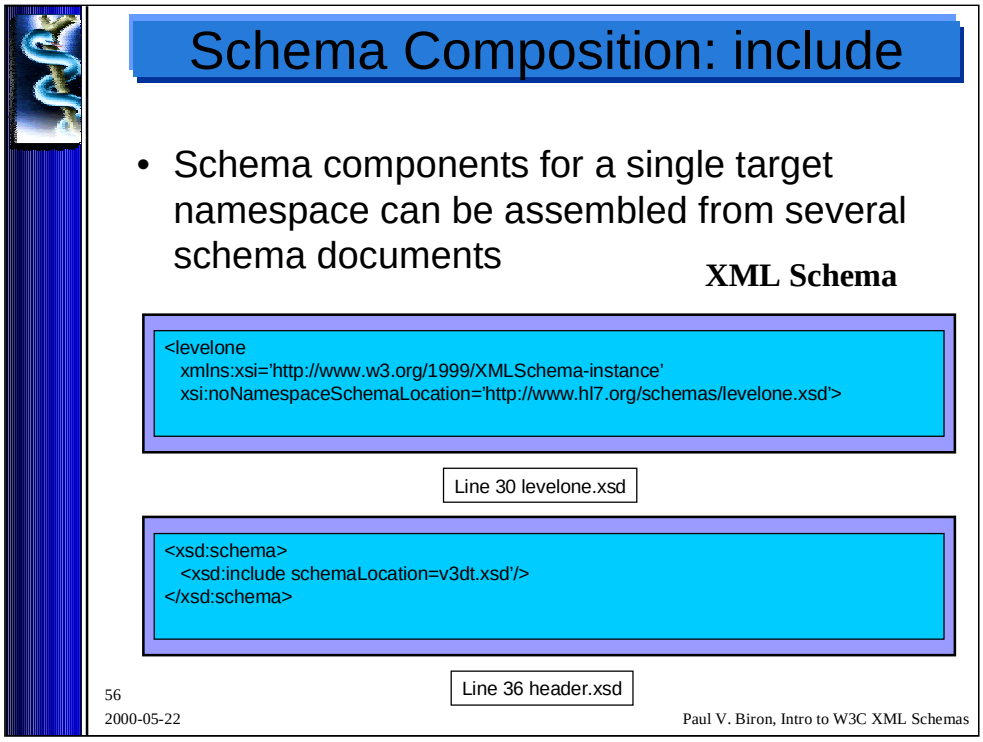
- target namespaces
- include
  - composing schemas for a single target namespace
- import
  - composing schemas for multiple target namespaces



When we want to check that an instance document conforms to one or more schemas (through a process called schema validation), we need to identify which element and attribute declarations and type definitions in the schemas should be used to check which elements and attributes in the instance document. The target namespace plays an important role in the identification process.

Thus far, there has been no requirement (or utility) to make use of namespace qualified elements in the development of either the PRA or V3 messages. This is partly due to the fact that DTD validation of instance documents which use multiple namespaces is not cleanly defined.

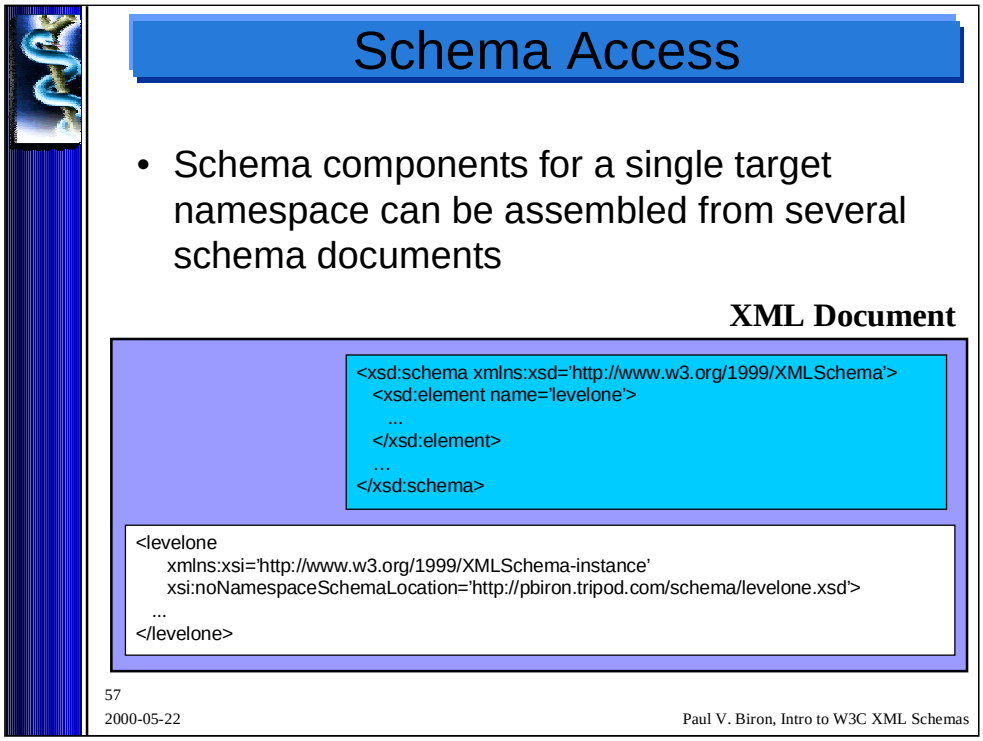
As we experiment with using XML Schemas to define the PRA and V3 messages we may find that taking advantage of namespace qualified elements/attribute declarations provides some benefits.



As schemas become larger, it is often desirable to divide their content among several schema documents for purposes such as ease of maintenance, access control, and readability.

The effect of this **include** element is to bring in the definitions and declarations contained in v3dt.xsd, and make them available as part of the header.xsd schema target namespace (as well as to bring in the definitions and declarations contained in header.xsd, and make them available as part of the levelone.xsd schema target namespace). The one important caveat to using include is that the target namespace of the **included** constructions must be the same as the target namespace of the including schema.

The required **schemaLocation** attribute, which is a URI, provides hints from the schema author to a processor regarding the location of schema documents. The presence of these hints does not require the processor to obtain or use the cited schema documents, and the processor is free to use other schemas obtained by any suitable means, or to use no schema at all.

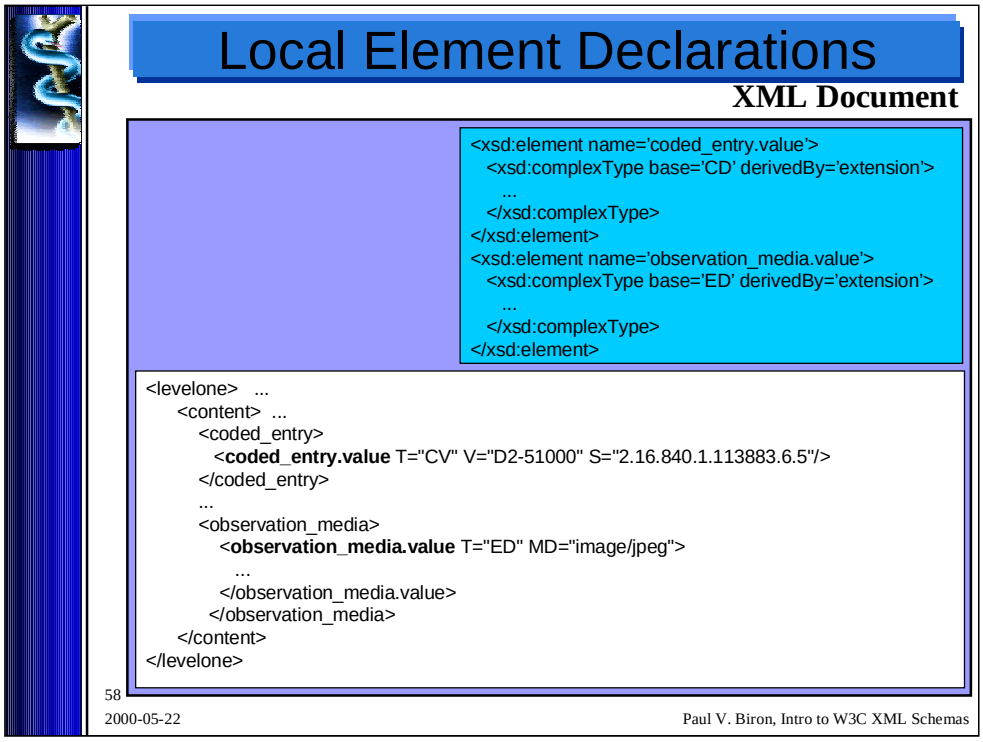


The **xsi:noNamespaceSchemaLocation** attribute on an element in an instance document provides hints to a processor regarding the location of schema documents that govern elements/attributes which are not namespace qualified.

As with the **schemaLocation** attribute on <xsd:include>, the presence of these hints does not require the processor to obtain or use the cited schema documents, and the processor is free to use other schemas obtained by any suitable means, or to use no schema at all.

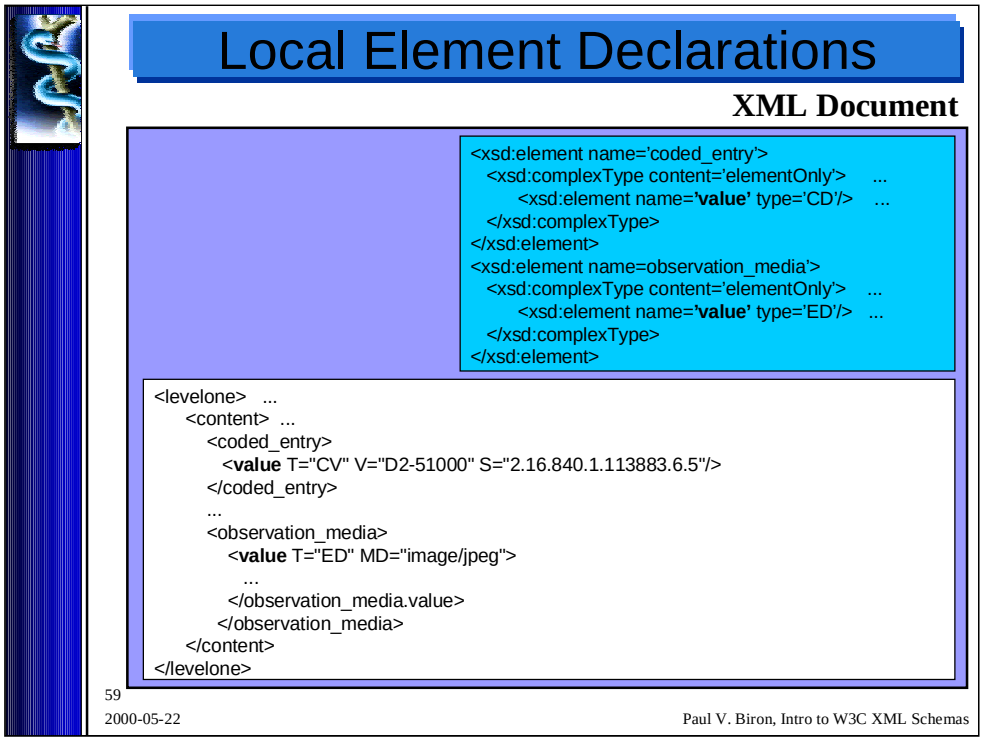
There is also an **xsi:schemaLocation** attribute for use in instance documents which use qualified element/attributes.

xsi:schemaLocation and **xsi:noNamespaceSchemaLocation** attributes can occur on any element, and all such attributes must be processed as if they had occurred on the element information item initially assessed for schema-validity (e.g., the root element of the instance document).



With DTDs, all element declarations are "global," that is, their content model/attribute list is context independent which can cause problems.

This example from the PRA (line 322 of levelone.xsd) demonstrates a common solution to a common problem. The problem is this: there may be many different elements which we'd like to name **value** but they may have different content models and/or attribute lists. With DTDs, the only reasonable solution is to develop some form of *naming convention* for the different elements (in this case, we prepend the name of the parent element followed by ".").



Using local element declarations, we are able to tell the schema processor (validating parser) to treat the **value** element differently when it appears as a child of code_entry vs. observation_media.

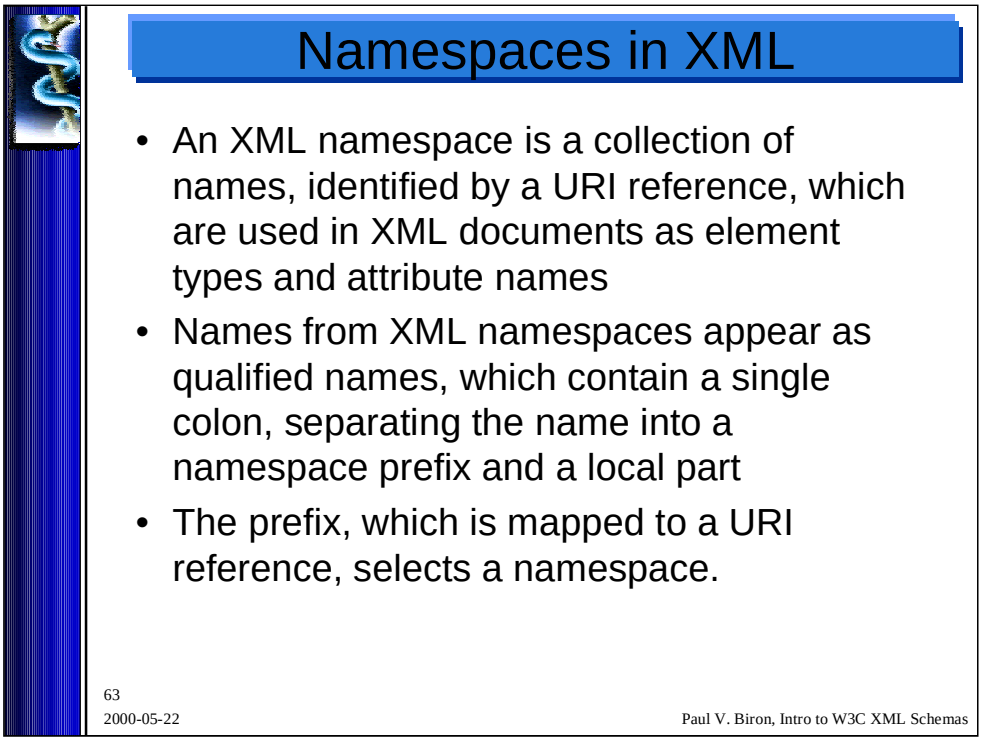
Schema Tools

- Validators
  - XSV
    - basic support for 2000-04-07 WD
    - http://cgi.w3.org/cgi-bin/xmlschema-check
  - Oracle XML Schema Processor
    - basic support for 2000-02-25 WD
    - http://technet.oracle.com/tech/xml/schema_java/index.htm
  - Xerces (Apache, IBM AlphaWorks)
    - basic support for 1999-12-17 WD
    - http://xml.apache.org/xerces-j/
  - XML Spy
    - basic support for 2000-04-07 WD
    - http://new.xmlspy.com

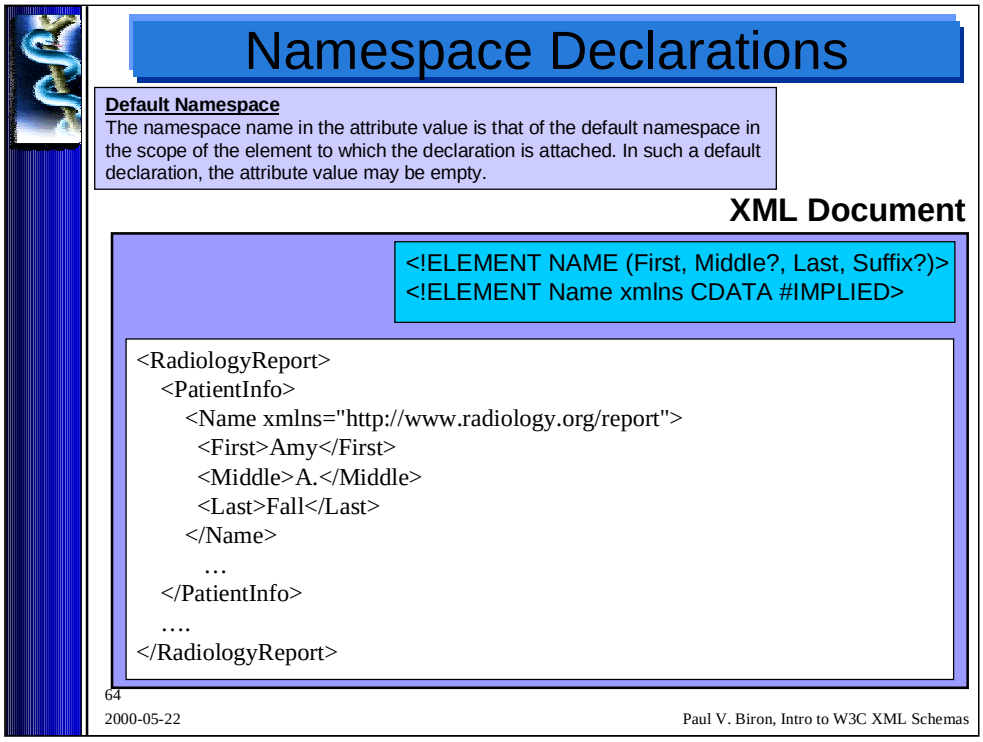
Schema Tools

- DTD to Schema Converters
  - XML Authority
    - basic support for 2000-04-07 WD
    - http://www.extensibility.com/products/xml_authority.htm
  - XML Spy
    - basic support for 2000-04-07 WD
    - http://new.xmlspy.com

Namespaces in XML



Why do we need namespace prefixes? URI references can contain characters not allowed in names, so cannot be used directly as namespace prefixes. Therefore, the namespace prefix serves as a proxy for a URI reference. An attribute-based syntax described below is used to declare the association of the namespace prefix with a URI reference.



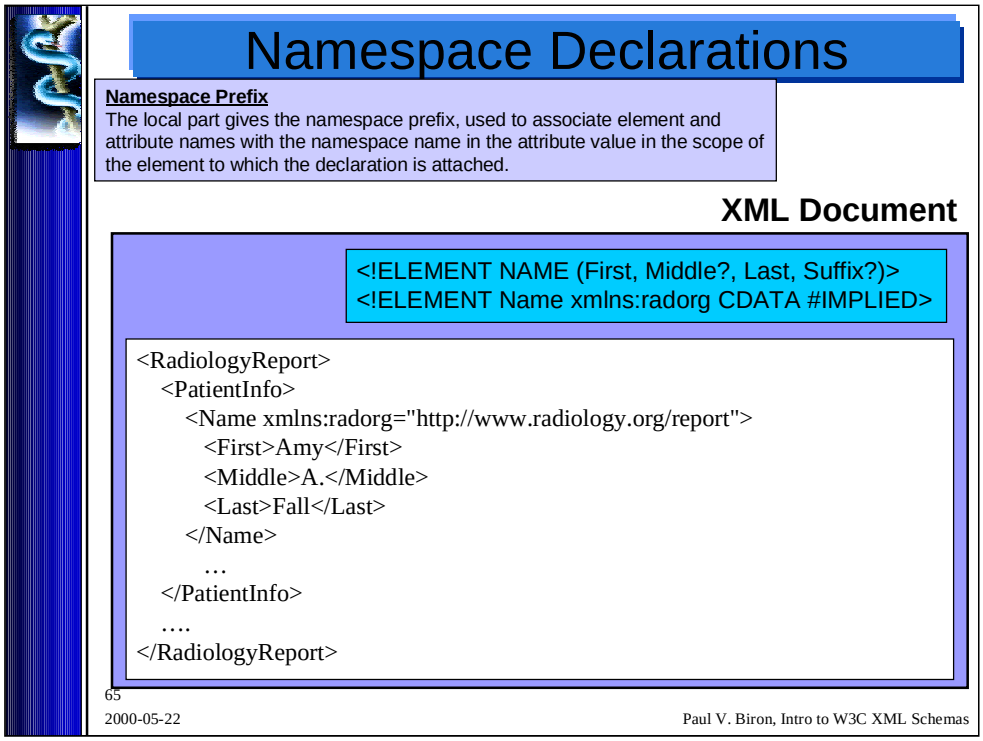
[1] NSAttName ::= PrefixedAttName | DefaultAttName

[2] PrefixedAttName ::= 'xmlns:' NCName

[3] DefaultAttName ::= 'xmlns'

[4] NCName ::= (Letter | '_') (NCNameChar)*/*An XML Name, minus the ":" */

[5] NCNameChar ::= Letter | Digit | '.'| '-' | '_' | CombiningChar | Extender



[1] NSAttName ::= PrefixedAttName | DefaultAttName

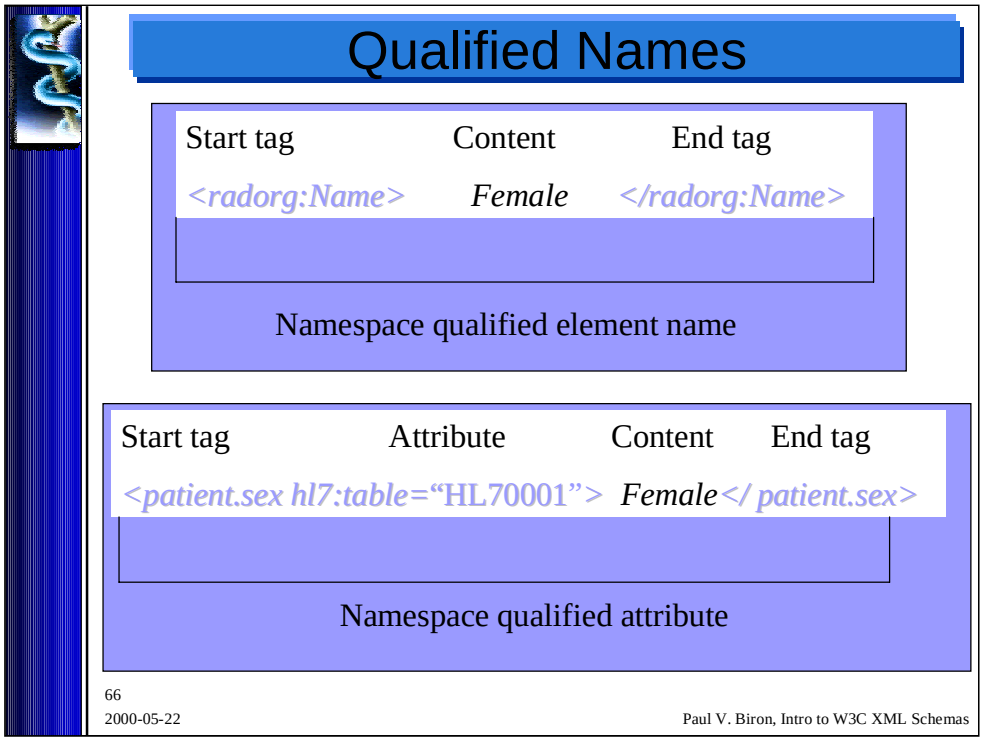
[2] PrefixedAttName ::= 'xmlns:' NCName

[3] DefaultAttName ::= 'xmlns'

[4] NCName ::= (Letter | '_') (NCNameChar)*/*An XML Name, minus the ":" */

[5] NCNameChar ::= Letter | Digit | '.'| '-' | '_' | CombiningChar | Extender

Prefixes beginning with the three-letter sequence x, m, l, in any case combination, are reserved for use by XML and XML-related specifications.



[6] Qname ::= (Prefix ':')? LocalPart

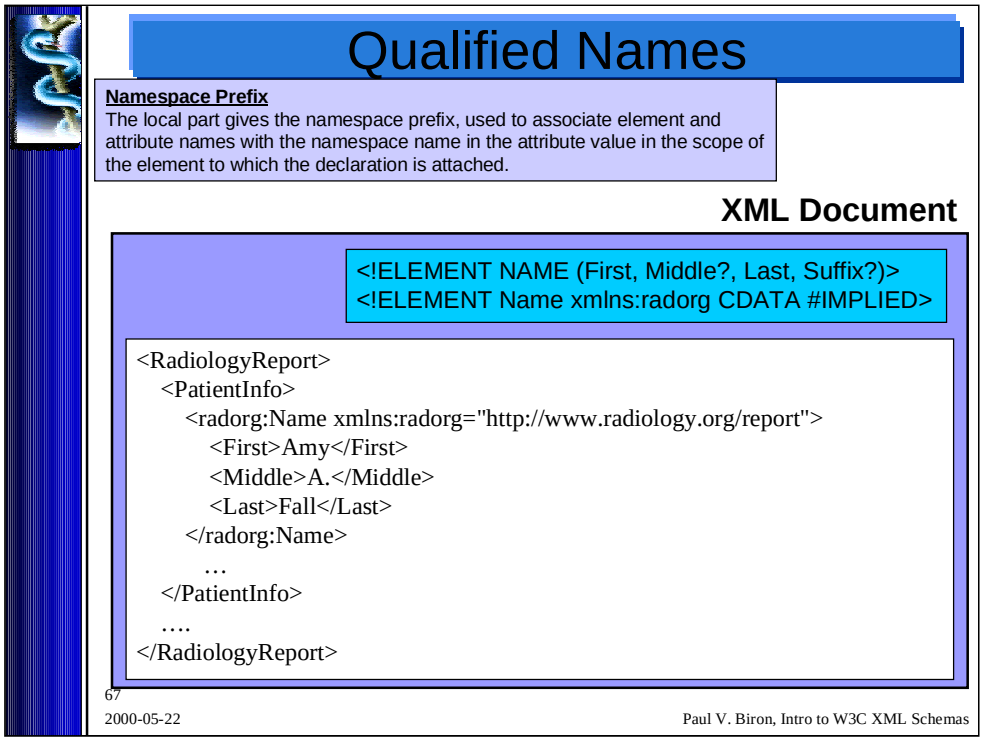
[7] Prefix ::= NCName

[8] LocalPart ::= NCName

[9] STag ::= '<' QName (S Attribute)* S? '>'

[10] ETag ::= '</' QName S? '>'

[11] EmptyElemTag ::= '<' QName (S Attribute)* S? '/>'



[6] Qname ::= (Prefix ':')? LocalPart

[7] Prefix ::= NCName

[8] LocalPart ::= NCName

The namespace prefix, unless it is xml or xmlns, must have been declared in a namespace declaration attribute in either the start-tag of the element where the prefix is used or in an an ancestor element (i.e. an element in whose content the prefixed markup occurs). The prefix xml is by definition bound to the namespace name http://www.w3.org/XML/1998/namespace. The prefix xmlns is used only for namespace bindings and is not itself bound to any namespace name.