Describing Document Types: The Schema Languages of XML Part 2

John Cowan

Copyright

- Copyright © 2005 John Cowan
- Licensed under the GNU General Public License
- ABSOLUTELY NO WARRANTIES; USE AT YOUR OWN RISK
- Black and white for readability
- The Gentium font available at

  http://www.sil.org/~gaultney/gentium

Part 2 Abstract

This part of the tutorial is entirely about W3C XML Schema Definition Language (XSD), also called XML Schema. XSD is a very complex standard with lots of details and special rules. This presentation attempts you to familiarize you with most of the concepts and terminology of XSD so that you can understand what people are talking about when they talk about it. It's not intended to teach you to write new schemas from scratch, although it probably makes it easier to learn to do that.

INTRODUCTION

Why W3C XML Schema?

- It's a W3C Recommendation
- More tools (and tool vendors) support it than support RELAX NG
- There are many worthy schemas already written in it
- It's *there*

Problems of XSD

- It has a large variety of arbitrary and hard-to-remember limitations
- It was designed by a committee
- It has no compact syntax

But wait!

- This presentation will use the compact syntax designed by Kilian Stillhard
- There is nothing official about it
- It isn't even widely used
- But:
  - It's related to the RELAX NG compact syntax
  - It lets me fit examples on a slide
  - Open-source conversion software is available

Types

- The notion of *pattern* in RELAX NG is replaced in XSD by the similar but not identical notion of *type*
- Every element and attribute has a type, either *simple* or *complex*
- Types can be given names which are then accessible from other XSD schemas

Simple types

- A *simple type* corresponds to a RELAX NG datatype
  - Attribute values are simple types
  - Element content can be simple types
- Elements with simple types must have only character content, no child elements or attributes

Complex types

- An element with *complex type with simple content* has character data that represents a simple type, plus attributes
- An element with *complex type with complex content* has child elements (with or without character data), or attributes, or both

THE INVOICE EXAMPLE, AGAIN

An Invoice in XML

```
<invoice number="640959-0" date="2002-03-12">
  <soldTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </soldTo>
  <shipTo>
    <name>Reuters Health Information</name>
    <address>45 West 36th St. New York NY 10018</address>
  </shipTo>
  <terms>Net 10 days</terms>
  <item ordered="6" shipped="6" unitPrice="7.812">
     Binder, D-ring, 1.5"</item>
  <item ordered="4" shipped="2" backOrdered="2"
     unitPrice="3.44">Fork, Plastic, Heavy, Medium</item>
</invoice>
```

The Russian Doll schema (1)

```
element invoice {
   (soldTo, shipTo, terms, item*)
   element soldTo {
      (name, address)
      element name { xs:string }
      element address { xs:string }
   }
   element shipTo {
      (name, address)
      element name { xs:string }
      element address  { xs:string }
   }
   element terms { xs:string }
```

The Russian Doll schema (2)

```
      element item {
         xs:string
         required attribute unitPrice { xs:decimal }
         required attribute ordered { xs:integer }
         required attribute shipped { xs:integer }
         attribute backOrdered { xs:integer }
      }
      attribute number { xs:string }
      attribute date { xs:date }
   }
```

Things to note

- Most of this looks pretty familiar!
- The content model *must* be in parentheses, as in DTDs, and is separate from the element sub-declarations (declarations aren't patterns)
- We use `xs:string` instead of `text` for elements with unconstrained simple content

Declaring attributes

- Attribute declarations are inside an element declaration but aren't part of the content model
- By convention, attribute declarations appear at the end (in the XML syntax, they are required to)
- Attributes are optional unless marked `required`

The Salami Slice schema (1)

```
element invoice {
  (soldTo, shipTo, terms, item*)
  required attribute number { xs:string }
  required attribute date { xs:date }
  }
element soldTo {
  (name, address)
  }
element shipTo {
  (name, address)
  }
element terms { xs:string }
```

The Salami Slice schema (2)

```
element item {
  xs:string
  required attribute unitPrice { xs:decimal }
  required attribute ordered { xs:integer }
  required attribute shipped { xs:integer }
  attribute backOrdered { xs:integer }
  }

element name { xs:string }

element address { xs:string }
```

Local vs. global declarations

- Declarations at the top level are *global*
- Declarations embedded in other declarations are *local* to those declarations
- Names in a content model match a local declaration if there is one, a global declaration if not
- The document element must have a global declaration

The Venetian Blind schema (1)

```
element invoice {invoice}
complexType invoice {
  (soldTo {name-addr},
   shipTo {name-addr},
   terms {xs:string},
   item {item}*)
  required attribute number {xs:string}
  required attribute date {xs:date}
  }
```

The Venetian Blind schema (2)

```
complexType item {
  xs:string
  attribute unitPrice {xs:decimal}
  required attribute ordered {xs:integer}
  required attribute shipped {xs:integer}
  attribute backOrdered {xs:integer}
  }

complexType name-addr {
  (name {xs:string}, address {xs:string})
  }
```

Defining Types

- Define a global name for a type
- Local type definitions are not allowed
- Type names don't conflict with element or attribute names
- An element name in a content model can be followed by a type name in braces to specify the element's type

Naming conflicts

- Global element, attribute, and type names do not conflict
- Complex and simple type names do conflict
- Simple type names you specify are in a different namespace from the built-in type names, and so do not conflict
- A local element or attribute shadows a global element or attribute with the same name

CONTENT MODELS

Content models

- Elements with complex content have a content model specifying the child elements (not also the attributes, as in RELAX NG)
- A content model contains one or more element names wrapped in parentheses and separated by `,` or `|` or `&`

Content models

- The element names can be followed by a type name in braces
- Content models can be nested
- XSD content models provide only modest advances over DTD content models:
  - occurrence constraints other than `?`, `*`, and `+`
  - the `&` connector (not as powerful as RELAX NG's)
  - mixed content models aren't so limited

Element content

- Specify a content model in parentheses
- Add local element declarations as needed
  - elements referenced in the content model are assumed to be global if undeclared
- Add local attribute declarations or references to global attribute declarations as needed

Element content

- Example:
```
element timestamp {
    (date, time)
    element date { xs:date }
    element time { xs:time }
    attribute verified
       { xs:boolean}
    }
```

Complex types with simple content

- Specify a simple type
- Add local attribute declarations or references to global attribute declarations as needed
- Example:

```
element title {
 xs:string
 attribute xml:lang
 attribute sub { xs:boolean }
 }
```

Mixed content

- Add the keyword `mixed` in front of the content model outside the parentheses
- Example:

```
element p {
  mixed (emph)
  optional attribute class
    { xs:Name }
  }
```

- `optional` is optional on attributes

Empty content

- Omit the content model and use the keyword `empty`
- Example:
```
element marker { empty }
```
- Attributes are still allowed

Nested element declarations

- You can write a local element declaration nested directly inside a content model (as opposed to inside the element or complex type declaration that contains the content model)
- It must be wrapped in braces
- Not clear why you'd do this, but it's possible
- The XML syntax does not distinguish

Choice and sequence

- Choice and sequence are exactly the same as in DTDs
- Conceptually more limited than RELAX NG, where they can be used in any pattern, not just a content model
- Complex example: `((a | b, c | d) | e)` means either a choice of `a` or `b` elements followed by a choice of `c` or `d` elements, or else an `e` element by itself.

Occurrence constraints

- Specified in a content model to give the number of times an element can appear
- The familiar `?`, `*`, and `+` are used in the compact syntax
- XSD also allows specifying the number of permitted repetitions
- Attributes do not use occurrence constraints; they are either `required` or `optional` (which is the default)

Occurrence constraints

- You can specify a fixed number of repetitions using `[5]`.
- For a variable number, use `[3,7]`.
- For a lower limit only, use `[3,]`.
- In the XML syntax, the attributes `minOccurs` and `maxOccurs` are used for all occurrence constraints
  - Both attributes default to 1
  - Use "`unbounded`" to specify no upper limit

All

- The `&` connector (pronounced “all”) is severely restricted in XSD
- It can only connect single element names with either:
  - no occurrence constraint or
  - `?` occurrence constraint
- It can only appear at the top level of an element declaration or complex type definition
- No interleaving is allowed

Named model groups

- You can give a name to a content model (with or without an occurrence constraint) using a `group` definition
- Example:
  `group abc { (a | b | c)* }`
- To refer to a named content model from another content model, use `@abc`
- Group definitions must be global

Named attribute groups

- Attribute groups are the attribute-related analogue of content model groups
- They name a related set of attributes used by more than one element or complex type
- Example:

```
attributeGroup i18n {
  attribute dir { xs:string }
  attribute xml:lang
    { xs:language }
  }
```

Named attribute groups

- To refer to an attribute group from another declaration or definition, use simply `attributeGroup i18n`, with no braces following
- Named attribute group definitions must be global
- Attribute groups can contain references to other attribute groups

SIMPLE TYPES

Simple types

- Simple types are used for two purposes:
  - The type of an attribute
  - The type of an element with no child elements
- So far we have only seen simple types that are built in to XSD: `xs:string`, `xs:integer`, `xs:date`, etc.
- Support for the `xs:` prefix is hard-coded in the compact syntax

Restricted simple types

- Each built-in type has *facets* that specify how it can be constrained: for example,
  - Numeric types can be limited in range
  - String types can be limited in length
- To specify a restricted simple type, follow the type name with facets wrapped in braces
- The complete list of facets and their specific syntax will be discussed later

Examples of restricted types

- `xs:integer {[-10, +10]}` is an integer between -10 and +10 inclusive
- `xs:string {"foo", "bar", "baz"}` is a string that is either "foo", "bar", or "baz"
- `xs:string {length = 5}` is a string with exactly five characters

List and union types

- `list {xs:integer}` specifies a simple type that consists of whitespace-separated integers
- Lists must specify just one simple type
- `union {xs:positiveInteger, xs:negativeInteger}` represents an integer that is either negative or positive (i.e. a non-zero integer)

Named simple type definitions

- A simple type definition lets you assign a name to a (generalized) simple type: a restricted type, a list, or a union
- Example:

```
simpleType codeword {
   xs:string {length = 5}
   }
```

- Simple type definitions must be global

Named simple type definitions

- The names of any simple types you define can be used just like the names of built-in simple types
- In particular, they can be restricted further
- If you want to prevent a particular facet of your type from being further restricted, precede the facet specification with the keywords `fixed` or `fixed-maximum` or `fixed-minimum`

Recursive simple types

- A simple type definition with no name can be used recursively inside a simple type specification
- Example:

```
list {
   simpleType xs:string
     { length = 5 }
   }
```

Fixed and default values

- A fixed or default value can be specified for an attribute (as in DTDs) or for an element with a simple type (which DTDs cannot do)
- To specify a value, append `= "value"` (for a fixed value) or `<= "value"` (for a default value) to:
  - a local element declaration
  - an attribute declaration (local or global)

Fixed and default values

- Examples:

```
element edit {
   xs:string <= "no"}
attribute country {
   xs:string = "US"}
```

- A default attribute will be supplied only if the attribute is not present
- Default element content will be supplied only if the element is present but empty

NAMESPACES

The target namespace

- The global declarations and definitions in a single XSD schema must define names that all belong to the same namespace, or else belong to no namespace
- To declare this namespace (called the *target namespace*), use a `targetNamespace` declaration at the top of the schema specifying the namespace URI

Importing

- It's possible for elements and attributes to have types or child elements or both that belong to namespaces other than the target namespace
- Such namespaces must first be declared and then their schemas must be imported
- Then anything declared using a global declaration or definition in the imported schema may be used in the current schema with a proper prefix

Importing

- Example:
```
namespace html
"http://www.w3.org/1999/html"

import "xhtml11.xsd"
namespace
"http://www.w3.org/1999/html"
```
- `namespace` declarations turn into `xmlns:foo` declarations in the XML syntax

Unqualified local declarations

- A local element declaration without a prefix declares an element in the target namespace
- A local attribute declaration without a prefix declares an attribute in no namespace
- To override these rules, use the keywords `qualified` and `unqualified` before the name being declared

Changing the qualification rules

- The qualification rules can be changed by specifying the following options at the top of the schema:
  - `elementDefault unqualified`
  - `attributeDefault qualified`

Changing the qualification rules

- **Warning:** in the XML syntax, local element declarations are unqualified by default!
- To override this rule (which the compact syntax automatically does), add the attribute `elementDefault = "unqualified"` to the `schema` (document) element

Multi-file schemas

- The `include` declaration can be used to incorporate definitions from another file
- `include` declarations must appear at the top of the schema but after any namespace declarations

Multi-file schemas

- Included definitions must specify one of:
  - the same target namespace as the base schema
  - no target namespace when the base schema has no target namespace
  - no target namespace when the base schema specifies a target namespace
- In the third case (a "chameleon schema") the included names take on the target namespace of the base

Redefinition

- The `redefine` declaration is like the `include` declaration, but allows replacement of selected declarations
- Put the new declarations in braces after the URI of the schema to be included
- Only simple and complex types, content model groups, and attribute groups can be redefined

Global attributes gotcha

- If a schema has a target namespace, then by default global attributes are in that namespace
- This means that they must be prefix-qualified in the instance document
- Global attributes therefore are not equivalent to local attribute declarations in different elements (unlike in RNG)

WILDCARDS

Element wildcards

- An element wildcard marks a point in a content model where anything may appear
- Element wildcards must always be enclosed in braces to delineate them from the rest of the content model
- The simplest element wildcard is `{skip any}`, which means that any element can appear here with no constraints

Element wildcards

- An alternative is `{strict any}`, which also allows any element but requires that it has a defined type and is valid against it
- `{lax any}` is a compromise:
  - if the element has a schema definition, treat it as `{strict any}`
  - otherwise as `{skip any}`
- A wildcard can be followed by an occurrence constraint, most often `*`

Attribute wildcards

- Attribute wildcards can be used in an element, complex type, or attribute group declaration
- They allow any number of attributes with any name to appear in an element
- No braces or occurrence constraints required or allowed

Attribute wildcards

- The keyword is `anyAttribute` rather than `any`
- `skip`, `strict`, and `lax` are all available, with the same meanings as in `any`, but `skip` probably makes the most sense

Namespace-specific wildcards

- An element or attribute wildcard can be made namespace-specific by following it with the keyword `namespace` and one or more URIs (in quotes, comma-separated)
- The URIs specify the possible valid namespaces for the element or attributes, as the case may be, that can appear

Namespace-specific wildcards

- You can also use the following keywords (without quotes) in place of URIs:
  - `##targetNS` means the target namespace
  - `##local` means elements or attributes without a namespace
  - `##other` means all namespaces except the target namespace
- Note that these keywords are not URIs, which is necessary for the XML syntax

Wildcard examples

- `(start, {skip any}*)` allows a `start` element followed by any number of other unvalidated elements
- `{strict any namespace "http://www.w3.org/1999/xhtml" }*` allows any valid elements from the XHTML namespace

Wildcard examples

- `lax anyAttribute namespace "http://www.w3.org/1999/xlink/"` allows any attributes from the XLink namespace
- The validator will validate the attributes if it has declarations for them, but will remain silent if it does not

DERIVED TYPES

Type derivation

- In XSD, type derivation is a relationship between named types
- Complex types can be derived by extension or by restriction
- Simple types can be derived only by restriction

Extending complex types

- A derived type *extends* its base type if it has all the same elements and attributes plus some new ones
- Syntax: `complexType derived extends base { ... }`
- Only the new elements and attributes appear inside the braces
- New elements are required to follow old ones in the instance

Restricting complex types

- A derived type *restricts* its base if it has fewer child elements (or occurrences of them), or fewer attributes, or both, than its base type
- Syntax:
```
complexType derived
         restricts base { ... }
```
- All the child elements valid in the restricted type must appear within the braces

Restricting complex types

- An optional child element or attribute in the base type may become required in the derived type
- In general, an occurrence constraint can be tightened in the derived type by having:
  - a greater number of minimum repetitions, and/or
  - a lesser number of maximum repetitions

Special cases

- A complex type with simple content is actually an extension of a simple type
- An empty complex type is actually a restriction of the built-in complex type `anyType` to have no child elements
- Simple types can be restricted by adding new facets or narrowing existing ones; they cannot be extended

`xsi:type`

- Attributes in the `xsi:` namespace are placed in the instance to give additional information to schema validators processing that instance
- The value of `xsi:type` is the type of the element in which it appears
- `xsi:type` is useful in certain special cases

Abstract types

- An abstract type (marked by the word `abstract` before the name of the type) is one that can't have any elements or attributes declared to use it
- They correspond roughly to abstract classes in object-oriented programming
- Abstract types are useful as base types for non-abstract derived types

Controls on derivation

- A final type is one which cannot have any derived types
- Syntax: use the keyword `final` before the type name
- The schema option `default final` (at the top of the schema) makes all types in the schema final

Controls on derivation

- To control derivation by extension and restriction separately, use the keywords `final-extension` and `final-restriction` instead
- These keywords can be used on individual type declarations or in `default` options at the top of the schema

Derived types in instances

- If the schema says that a particular element must be of a certain type, the instance may in fact contain an element that matches a derived type instead
- However, the actual type of the element must be marked using the `xsi:type` attribute in the instance
- This is the only way to use an element declaration involving an abstract type

Blocking the use of derived types

- You can use the keywords `block`, `block-extension`, and `block-restriction` to prevent elements of derived types from being used in the instance
- These keywords are exactly analogous to `final`, `final-extension`, and `final-restriction`

Substitution groups

- An element can be declared as substitutable for another element, known as its *head element*
- All elements substitutable for the same head constitute a *substitution group*
- The relationship of an element in a substitution group to its head element is analogous to the relationship of a derived type to its base type

Substitution groups

- Syntax:
```
element s substitutes h
    { ... }
```
- The type of an element in a substitution group must be:
  – the same as the type of the head, or
  – derived from the type of the head

Abstract elements

- Abstract elements are the substitution-group analogue of abstract types
- Abstract elements can be declared in the schema but can't appear in the instance
- Other members of the substitution group can appear in the instance in place of the abstract element
- Syntax: use `abstract` before the element name

Substitutions in instances

- If the schema calls for a particular element, any other element in the substitution group may appear instead
- No magic attribute marker is required in this case
- The keyword `block-substitution` prevents the substitution of a particular element in the instance, analogous to the keyword `block` for derivation

IDENTITY CONSTRAINTS

Uniqueness

- A `unique` declaration is placed inside an element declaration
- It specifies a constraint that certain descendant elements of the declared elements or their attributes must all have distinct values in the document
- The descendant elements or attributes are specified by simplified XPaths.

Uniqueness

- Example declaration:
```
unique uniqueNames
   field "name" in "person"
```
- This constraint requires:
  - that the values of the `name` child elements (the *fields*) ...
  - of any `person` elements (the *selection*) ...
  - which are children of a specific `persons` element (the *constraint root*) ...
  - are all different.

Uniqueness

- Selector and field elements that are not descendants of a constraint root are not involved in the constraint
- Each distinct constraint root element in the document has its own space of unique values
- Uniqueness declarations require you to specify a name, which is used only for documentation purposes

Uniqueness example

```
<persons>
  <person>
    <name>John Cowan</name>
    ...
  </person>
  <person>
    <name>George Bush</name>
    ...
  </person>
```

Multiple-field uniqueness

- You can specify multiple fields whose values must be jointly unique
- Example:

```
unique uniqueNames
   field "firstname",
     "lastname" in "person"
```

- The combined first name and last name values must be unique for each person

XPath limitations

- The basic selector XPath is a chain of simple name tests, like `"foo"`, `"foo/bar"`, `"foo/*/bar"`
- You can also have `".//foo/bar"`, allowing the selection elements to be an arbitrary descendant of the constraint root
- You can specify multiple selector XPaths by combining them with `|` inside the quotes

XPath limitations

- Field XPaths have much the same restrictions as selector XPaths
- The last step may be an attribute, prefixed with `@` as usual
- A field XPath may not contain `|`

Keys and references

- Keys and key references are a generalized form of IDs and ID references from DTDs
- A `key` declaration has the same syntax and semantics as a `unique` declaration, except that the fields must exist in each selection (not just be unique when they do exist)
- A `keyref` declaration says that the values of specified elements and attributes must be equal to the values specified by a specified `key` or `unique`

Keys and references

- A `keyref` declaration matches a `key` or `unique` declaration with the same constraint root (or a descendant of it) and a matching name
- Example: To make sure every person's manager is also a person, use

```
key nameKey field
   "name" in "person"
keyref managerRef field
   "manager" in "person"
```

Keys and references

- The name of a `keyref` declaration is for documentation purposes only
- Attributes with the simple types `xs:ID` and `xs:IDREF` are treated like keys and key references, but obey DTD rules:
  - all IDs must be unique in the document
  - an IDREF can refer to any ID in the document

OTHER DETAILS

A few lexical details

- Strings, URIs, and XPaths must be wrapped in double quotation marks
- `\"`, `\n`, `\r`, `\t` in strings mean what you expect
- Names that are the same as compact-syntax keywords must be preceded by `\`

Annotations

- XML Schema allows two kinds of annotations, documentation and application information
- Both can contain arbitrary content
- A comment wrapped in `/*` and `*/` becomes a documentation annotation for the next declaration
- The compact syntax can't represent complex documentation or application information

Version specification

- You can specify the version of a particular schema using a `version` declaration at the top of the schema
- Example:
  ```
  version "1.2b"
  ```
- This is for documentation purposes only

The `schemaLocation` hint

- The attribute `xsi:schemaLocation` may be used in the document element of an instance
- Its value is one or more pairs of URIs separated by spaces
- Each pair consists of a namespace URI followed by the URI where a schema for that namespace may be found

The `schemaLocation` hint

- The location of a schema for elements in no namespace is specified by the analogous attribute `xsi:noNamespaceSchemaLocation`
- Both attributes are merely hints which a schema validator may ignore
  - Jing ignores them, for example
- Attributes in the `xsi:` namespace are not themselves validated against a schema

Nil values

- *Nil* is the XSD analog of the database null value
- It can be important to distinguish between:
  - a missing element
  - an empty element
  - a nil-value element
- Elements that can be nilled are marked by preceding the declaration with `nillable`

Nil values

- Nil elements are marked in the instance with the attribute `xsi:nil="true"`
- A nil element must be empty no matter what its declaration says
- Nil elements can have attributes, however
- (Note: RNG can handle nils by providing a choice between the real content model and `attribute xsi:nil { "true" }`)

SUMMARIES

Summary of schema options

- All these must appear (if at all) at the top of the schema before anything else:
  - `targetNamespace`
  - `namespace`
  - `default`
  - `elementDefault`
  - `attributeDefault`
  - `version`

Summary of schema inclusions

- All these must appear (if at all) after any schema options but before anything else:
  - `include` (incorporate text directly; must share target namespace)
  - `redefine` (like `include`, but allows overriding of type and group declarations)
  - `import` (bring in global declarations from other target namespaces)

Summary of top-level declarations

- All these must appear after any schema options or inclusions:
  - `element`
  - `attribute`
  - `complexType`
  - `simpleType`
  - `group`
  - `attributeGroup`

Summary of element declarations

- A name
- Optional substitution and derivation
- Then in braces, one of:
  - A content model, with or without `mixed`
  - `empty`
  - A simple type

Summary of element declarations

- Plus any of:
  - local element declarations
  - local attribute or attribute group declarations
  - attribute wildcards
  - identity constraints
- Plus an optional fixed or default value

Summary of attribute declarations

- A name, plus:
  - a simple type reference
  - an optional fixed or default value

Summary of complex type definitions

- A name plus any of the things that an element declaration can have:
  - except a head element for a substitution group

Summary of simple type definitions

- A name plus one of:
  - a simple type name with optional facets
  - a simple type specification with optional facets
  - a union simple type
  - a list simple type

Summary of group definitions

- Model group definitions:
  - A name plus a content model
- Attribute group definitions:
  - A name plus any of:
    - attribute declarations
    - attribute wildcards
    - references to other attribute groups

Summary of notation declarations

- A name, an optional system identifier (or URI), and an optional public identifier
- Notation declarations appear in XSD only for backward compatibility with DTDs
- Notation declarations in DTDs are obsolescent anyhow

DATATYPES

XML Schema Datatypes

- A type is a named set of values
- An XML Schema datatype provides a standardized, machine-checkable representation of a type
- XML Schema types can be grouped:
  - numeric, date, boolean, string, miscellaneous

Decimal Types

- `decimal`
  - `integer`
    - `nonPositiveInteger`
      - `negativeInteger`
    - `nonNegativeInteger`
      - `positiveInteger`
      - `unsigned{Long, Int, Short, Byte}`
    - `long, int, short, byte`

Decimal Types

- `long`, `short`, `int`, and `byte` are the same as in Java: 64, 32, 16, 8 bits
- `unsignedLong`, `unsignedShort`, `unsignedInt`, and `unsignedByte` are the obvious unsigned analogues
- All other numeric types are unbounded

Floating-point Types

- Only two floating-point types
  - `float`
  - `double`
- IEEE ranges and precisions (same as Java, all modern hardware)

Date Types

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

Date Types

- Duration
  - `duration`
- Single Time Interval
  - `dateTime, date, gYear, gYearMonth`
- Recurring Time Interval
  - `time, gMonth, gDay, gMonthDay`

Date Type Examples

`duration`	`P1D    PT30M    P2M`
`dateTime`	`2002-06-17T13:45:00`
`Date`	`1776-07-04`
`Time`	`17:05:00-05:00`
`gYear`	`1984`
`gMonth`	`--12`
`gDay`	`---29`
`gYearMonth`	`1917-11`
`gMonthDay`	`--09-11`

Boolean Type

- Only two values are legal:
  - `true` (which can also be written `1`)
  - `false` (which can also be written `0`)

String Types

- `string`
  - `normalizedString`
    - `token`
      - `language`
      - `NMTOKEN(S)`
      - `Name`
        - `NCName`
          - `ID, IDREF(S), ENTITY(IES)`

Miscellaneous Types

- Raw octet types
  - `hexBinary`
  - `base64Binary`
- `anyURI`
- `QName`
- `NOTATION`

Syntax of datatype names

- Datatypes are always namespace-qualified
- Default prefixes:
  - `xsd:` in the RELAX NG compact syntax
  - `xs:` in the W3C compact syntax
- XML syntaxes do not have default prefixes, but the above forms should be used by convention

FACETS

Facets

- Allow the creation of new datatypes by restricting the existing ones in one or more ways
- Called parameters in RELAX NG
- Facets can be grouped into families applicable to datatype families:
  - length, value, pattern
  - enumeration, whiteSpace

General Facet Syntax

- In the RELAX NG compact syntax, all facets are specified as `facetname = "facetvalue"`, with explicit quotation marks
- In the XSD compact syntax:
  - Some facets have their own idiosyncratic ultra-compact syntax
  - All other facets are specified as `facetname = facetvalue`, with no quotation marks

Length Facets

- Applicable to string and miscellaneous types
- `length` facet specifies the exact length
- `minLength` and `maxLength` facets set limits; either or both may be used
- lengths of `hexBinary` and `base64Binary` types are measured in octets, not characters

Length Facets

- The XSD compact syntax has a special representation of length facets based on its representation of occurrence indicators
  - `length = [3]` represents a `length` of 3
  - `length = [3,]` represents a `minLength` of 3
  - `length = [,5]` represents a `maxLength` of 5

Length Facets

- The XSD compact syntax has a special representation of length facets based on its representation of occurrence indicators
  - `length = [3,5]` represents simultaneously:
    - a `minLength` of 3
    - a `maxLength` of 5

Bounds Value Facets

- Applicable to numeric and date types
- `minExclusive` and `minInclusive` specify a lower bound; either but not both may be used
- `maxExclusive` and `maxInclusive` specify an upper bound; either but not both may be used

Bounds Value Facets

- The XSD compact syntax also has a special representation of bounds facets based on the mathematical representation of intervals
- Square brackets represent inclusive bounds (as with lengths), parentheses represent exclusive ones
- So `[-100,+100]` bounds the value between -100 inclusive and +100 inclusive

Bounds Value Facets

- You can mix bracket types: `[0,100)` represents at least 0 but not more than or equal to 100
- You can also omit either side of the interval:
  - `[10.5,]` means at least 10.5
  - `(,10.5)` means less than 10.5

Decimal Value Facets

- `totalDigits` specifies the total number of significant digits in a `decimal`, `integer`, `(non)PositiveInteger`, or `(non)NegativeInteger` value
- `fractionDigits` specifies the number of fractional digits in a `decimal` value

Pattern Facet

- Applicable to any type
- Specifies a regular expression that the data must match
- XML Schema: If multiple `pattern` facets are present, the data must match at least one of them
- RELAX NG: If multiple `pattern` facets are present, the data must match all of them
- In the XSD compact syntax, patterns are enclosed in slashes

`whiteSpace` facet

- Not applicable in RELAX NG
- Specifies how to handle whitespace in a `string`
- Three cases:
  - Preserve (use the value as-is)
  - Replace (change TAB, CR, LF to a space)
  - Collapse (remove leading and trailing whitespace and collapse runs of whitespace)

`enumeration` facet

- Not applicable to RELAX NG (use a choice between values instead)
- Specifies a set of values
- The data must match exactly one of them
- In the XSD compact syntax, represented by comma-separated strings in quotation marks

Facets for non-atomic types

- In XSD, you can apply the `length`, `maxLength`, and `minLength` facets to a list type, in which case they refer to the number of items in the list
- In XSD, you can apply the `pattern` and `enumeration` facets to both list types and union types

MORE INFORMATION

http://www.w3.org/TR/xmlschema-0/
http://dret.net/projects/xscs/
http://www.ccil.org/~cowan/