Introduction to XML
Patryk Czarnik
XML and Applications 2014/2015
Lecture 1 - 6.10.2014

Text markup - fundamental distinction
Presentational markup
- Describes the appearance of text fragment
  - font, color, indentation...
- Procedural or structural
- Examples:
  - Postscript, PDF, TeX
  - HTML tags: &lt;B&gt; &lt;BR&gt;
  - direct formatting in word processors
  - XSL-FO (we will learn)
Semantic markup
- Describes the meaning (role) of a fragment
- Examples:
  - LaTeX (partially)
  - HTML tags: &lt;STRONG&gt; &lt;Q&gt; &lt;VAR&gt;
  - styles in word processors
  - most of SGML and XML applications

Documents in information systems
- Since the introduction of computers to administration, companies and homes plenty of digital documents have been written (or generated).
- Serious problem: number of formats, incompatibility.
- De facto standards in some areas (e.g. .doc, .pdf, .tex)
  - most of them proprietary
  - many of them binary and hard to use
  - some of them undocumented and closed for usage without a particular tool

Road to XML
- Late 1960s - IBM - SCRIPT project, INTIME experiment
  - Charles Goldfarb, Edward Mosher, Raymond Lorie
  - Generalized Markup Language (GML)
- 1974-1986 - Standard Generalized Markup Language (SGML)
  - ISO 8879:1986
- Late 1990s - Extensible Markup Language (XML)
  - W3C Recommendation 1998
  - Simplification(!) and subset of SGML

Why is XML a different approach?
- Common base
  - document model
  - syntax
  - technical support (parsers, libraries, supporting tools and standards)
- Different applications
  - varying set of tags
  - undetermined semantics
- Base to define formats rather than one format
- General and extensible!

What is XML?
- Standard - Extensible Markup Language
  - World Wide Web Consortium (W3C) Recommendation
    - version 1.0 - 1998
    - version 1.1 - 2004
- Language - a format for writing structural documents in text files
- Metalanguage - an extensible and growing family of concrete languages (XHTML, SVG, etc. . . )
- Means of:
  - two primary applications
  - document markup
  - carrying data (for storage or transmission)
What is XML not?
- Programming language
- Extension of HTML
- Means of presentation
  - You should say “data represented in XML format” rather than “presented”
- Web-only, WebServices-only, database-only, nor any other *-only technology – XML is general.
- Golden hammer
  - XML is not a solution for everything

XML components
Main logical structure
- Element (element)
  - start tag (znacznik otwierający)
  - end tag (znacznik zamkijący)
- Attribute (attribut)
  - Text content / text node (zawartość tekstowa / węzeł tekstowy)

XML components
Comments and PI's
- Comment (komentarz)
  - Processing instruction (instrukcja przetwarzania, ew. instrukcja sterująca, dyrektywa)
  - target (cel, podmiot)

XML components – CDATA
- CDATA section (sekcja CDATA)
  - Whole content treated as a text node, without any processing.
  - Allows to quote whole XML documents (not containing further CDATA sections).
  ```xml
  <![CDATA[The same text fragment written in 3 ways:
  <option x = '0' &amp; y = '1'; x &lt; 0]]>
  ]]>
  <![CDATA[The same text fragment written in 3 ways:
  &lt;option x = '0' &amp; y = '1'; x &lt; 0]]>
  ]]>
  <![CDATA[The same text fragment written in 3 ways:
  &lt;option x = '0' &amp; y = '1'; x &lt; 0]]>
  ]]>
  ```

XML declaration
- Looks like a PI, but formally it is not.
- May be omitted. Default values of properties:
  - version = 1.0
  - encoding = UTF-8 or UTF-16 (deducted algorithmically)
- standalone = no
- Document type declaration (DTD)
  - Optional

Unicode and character encoding
- Unicode – big table assigning characters to numbers.
  - Some characters behave in a special way, e.g. U+02DB . Ogonek
  - One-byte encodings (ISO-8859, DOS/Windows, etc.)
    - Usually map to Unicode, but not vice-versa
    - Mixing characters from different sets not possible
  - Unicode Transformation Formats:
    - UTF-8 – variable-width encoding, one byte for characters 0-127 (consistent with ASCII), 16 bits for most of usable characters, up to 32 bits for the rest
    - UTF-16 – variable-width, although 16 bits used for most usable characters; big-endian or little-endian
    - UTF-32 – fixed-length even for codes > 0xFFFF

Document Type Definition
- Specifies the “type” of this XML document.
- Not required and in fact not used in modern applications.
- Can be written in a separate file, inside the XML document, or using a mixed approach.
- Using a separate file gives some advantages and usually this is the choice.
- Apart from document structure definition, which we’ll learn in the next week, it allows to define entities and notations.

Where do entities come from?
- 5 predefined entities: lt gt amp apos quot
- Custom entities defined in DTD
  - simple (plain text) or complex (with XML elements)
  - internal or external
- `<ELEMENT doc ANY>`
- `<ENTITY %MyEntity>`
- `<ENTITY lect1 SYSTEM 'lect1.text'>`

We skip details of unparsed entities and notations.
Associating DTD to XML document (3 options)

- **Internal DTD**
  ```xml
  <!DOCTYPE doc [ 
  <!ELEMENT doc ANY>
  <!ENTITY title "XML and Advanced Applications">]
  <doc>
  </doc>
  </!DOCTYPE doc SYSTEM "entities.dtd">
  </!DOCTYPE doc SYSTEM "entities.dtd">
  <![ENTITY title "XML and Advanced Applications">]
  </doc>...
  </!DOCTYPE doc SYSTEM "entities.dtd">
  </!DOCTYPE doc SYSTEM "entities.dtd">
  ```

- **External DTD**
  ```xml
  <!DOCTYPE doc [ 
  <!ELEMENT doc ANY>
  <!ENTITY title "XML and Advanced Applications">]
  <doc>
  </doc>...
  </!DOCTYPE doc SYSTEM "entities.dtd">
  <![ENTITY title "XML and Advanced Applications">]
  </doc>...
  </!DOCTYPE doc SYSTEM "entities.dtd">
  ```

- **Mixed approach** – internal part processed first and has precedence for some kinds of definitions (including entities)
  ```xml
  <!DOCTYPE doc [ 
  <!ELEMENT doc ANY>
  <!ENTITY title "XML and Advanced Applications">]
  <doc>
  </doc>...
  </!DOCTYPE doc SYSTEM "entities.dtd">
  <![ENTITY title "XML and Advanced Applications">]
  </doc>...
  ```

Document as a tree

```
employee
  |  fname  |
  |  surname |
  |  tel     |
  |  Current |
  |  tel     |
  |
  |  Jan     |
  |  Kowalski|
  |  12323435 |
  |  intern  |
  |  1313    |
  |
  |  60506665 |
```

Language or metalanguage?

- **XML** is a language.
- Grammar, additional constraints expressed descriptively → one can determine whether a sequence of characters is well-formed XML.
- Better to think of as a **metalanguage**.
- Common base for defining particular languages
- Set of languages (open, unlimited)
- A particular language based on XML will be called an XML application.

XML vs (X)HTML

```
HTML
- Defined set of elements and attributes
- Their meaning established
- Defined (to some extent) way of presentation
- Although specification exists, tools accept (and often create) incorrect HTML.
```

```
XML
- All (syntactically correct) tag names allowed
- Undefined semantics
- <> is not necessarily a paragraph!
- Unspecified way of presentation
- Processors obliged to work with well-formed XML only
```

XML syntax – supplement

- Elements have to be closed (in stack-like order).
- Shorthand for empty elements: `<elem/>`
- Two possibilities of attribute value quotation: “ ” or ‘ ’
- Not every character is allowed in XML document, even by a character reference.
- Different sets in XML 1.0 and 1.1
- Surprising curiosities:
  - `>` is forbidden within comments
  - `]]>is forbidden anywhere in text content`
- therefore &gt; is ever needed

```
<?xml version="1.0"?>
<customer id="1313">
  <order date="2013-10-18" order-date="2013-10-18">123</order>
  <item good="561180" qty="10"/>
  <item good="561180" qty="10"/>
  <item good="565660" qty="10"/>
</customer>
```

Two faces of XML

```
"Text document"
- Flexible structure, mixed content
- Text (formatted or annotated with tags)
- Content created and used by humans
```

```
"Database"
- Strict structure
- Various datatypes
- Created and processed automatically
```

Applications of XML

- **Traditional (successor of SGML)** – content management
  - Source text markup – preferably semantic – to be used in various ways (publication, searching, analysis)
  - Combining documents (links, references, etc.)
- **Modern** – data serialisation, programming technologies
  - Saving structural data in files
  - Integration of distributed applications: “web services” (SOAP, REST, AJAX)
  - Databases (import/export, “XML databases”)
  - Format of configuration files for many technologies
- Somewhat between – IMO the best place for XML:
  - Structural documents (forms etc.) to be processed by IT systems
What can we do with XML?

- Define new XML-based formats using XML Schema or other standards
  - Validate documents against the definition
- Edit manually (e.g. Notepad) or using specialised tools
- Store in files or databases, transfer through network
- Process documents (read, use, modify or create, write) in custom applications
  - Use existing parsers and libraries
- Search and query for data using XQuery, XPath, XSLT, or custom applications
- Transform to other formats (for presentation, but not only) using XSLT, XQuery, or custom applications
- Format using stylesheets or specialised tools

Advantages of XML

- Compared to binary formats:
  - Readable (to some extent...) for humans, “self-descriptive”
    - Possibility to read or edit using simplest tools
    - Easier debugging
- Compared to ad-hoc designed formats:
  - Common syntax and document model
    - Common way of defining XML applications (XML Schema)
  - Existing tools, libraries, and supporting standards
- Interoperability
- Compared to WYSIWYG editors and their formats:
  - Semantic markup available, more advanced than flat styles
  - Relatively easy conversion to other formats (using transformations and stylesheets)

Drawbacks of XML

- Verbosity
  - Writing numbers, dates, images, etc. as text not efficient
  - Syntax of XML (e.g. element name repeated in closing tag)
  - Common use of whitespace for indentation (not obligatory, of course)
- Complexity
  - Inherited features of SGML (entities, notations, even whole DTD) which are rarely used in modern applications, but have to be supported by processors
  - Technical restrictions, e.g.:
    - Elements cannot overlap (trees, not DAGs)
    - Binary content not allowed (there are some solutions - we will learn)
    - Requirement of exactly one root element impractical

Alternatives to XML – text

For text-oriented applications of XML:

- TeX and LaTeX
- Direct tagging in graphical text editors
  - flat styles
  - more advanced solutions, e.g. Adobe FrameMaker
- “Lightweight markup”
  - MediaWiki
  - AsciiDoc, OrgMode, and others

Alternatives to XML – data

For “modern” applications of XML:

- JSON (JavaScript Object Notation)
  - more compact than XML
  - often used instead of XML in AJAX-like solutions
- YAML
  - similar to JSON, but more advanced
- CSV
  - simple and poor
- ASN.1, EDIFACT
  - different approach (not so generic)

Where does XML make sense?

- Text-oriented applications
  - As source format for further processing
  - To denote metadata, structural dependencies, links, etc.
- Data-oriented applications
  - When structural text or tree-like structure appears in a natural way; e.g. business documents interchange
  - When interoperability more important than efficiency
  - public administration services, external business partners, heterogeneous environment

But XML (read also “WebService”) is maybe not the best format to transfer arrays of numbers between nodes performing a physical process simulation.

Don’t force to use XML when there are better solutions.
Modelling XML Applications

Patryk Czarnik
XML and Applications 2013/2014
Lecture 2 - 14.10.2013

XML application (recall)

- XML application (zastosowanie XML)
  - A concrete language with XML syntax
- Typically defined as:
  - Fixed set of acceptable tag names (elements and attributes, sometimes also entities and notations)
  - Structure enforced on markup, e.g.: "<person> may contain one or more <first-name> and must contain exactly one <surname>"
  - Semantics of particular markups (at least informally)

Benefits of formal definition

- Tangible asset resulting from analysis & design
- Formal, unambiguous definition of language
- Reference for humans (document authors and readers, programmers and tool engineers)
- Ability to validate documents using tools or libraries
- Programs may assume correctness of the content of validated documents (less conditions to check!)
- Content assist in editors
- Autocomplete during typing, stub document generation

Modelling new XML application

- Analysis & design
  - analysis of existing documents, new requirements, etc.
  - identifying nouns, their role and dependencies
  - data types, constraints, limits
- Writing down
  - structure definition – "schema"
  - semantics description – usually in natural language; in schema (comments, annotations) or a separate document

Two levels of document correctness (recall)

- Document is well-formed (poprawny składniowo) if:
  - conforms to XML grammar,
  - and satisfies additional well-formedness constraints defined in XML recommendation.
  - Then it is accessible by XML processors (parsers).
- Document is valid (poprawny strukturalnie, "waliduje się") if additionally:
  - is consistent with specified document structure definition;
  - from context: DTD, XML Schema, or other;
  - in strict sense (DTD): satisfies validity constraints given in the recommendation.
  - Then it is an instance of a logical structure and makes sense in a particular context.

Element content - simple case

- Example content
  - <student>
    <first-name>Monika</first-name>
    <surname>Dąbrowska</surname>
    <birth-date>1998-03-13</birth-date>
  </student>

Document Type Definition (DTD)

- Defines structure of a class of XML documents ("XML application").
- Optional and not very popular in new applications.
  - Replaced by XML Schema and alternative standards.
  - It is worth to know it, though. Important for many technologies created 10-30 years ago and still in use.
- Contains declarations of:
  - elements ("element types" to be precise)
  - attributes ("attribute lists"...)
  - entities – described last week
  - notations – extremely rarely used, we’ll skip them

Example DTD (fragments)

- <student>
  <first-name>Henryk</first-name>
  <first-name>Walton</first-name>
  <last-name>Jones</last-name>
  <birth-date>1995-05-05</birth-date>
  <identification>
    <passport nr>3234567898</passport nr>
    <country>USA</country>
    <identification />
  </identification>
</student>

- <teacher>
  <first-name>Patryk</first-name>
  <last-name>Czarnik</last-name>
</teacher>

- <ELEMENT teacher (first-name, last-name)>
  <!ATLIST teacher degree (MSc | PhD | Prof) REQUIRED guest (yes | no) "no">
  <!ELEMENT student (first-name, surname, birth-date)>
  <!ELEMENT student (first-name, surname, birth-date, identification)>
  <!ELEMENT student (first-name, surname, birth-date, identification)>
  <!ELEMENT student (first-name, surname, birth-date, identification)>
</ELEMENT>

Standards for defining structure of XML documents

- DTD
  - origins from SGML (1974)
- XML Schema - W3C Recommendation(s)
  - version 1.0 - 2001
  - version 1.1 - 2012
- Relax NG
  - OASIS Committee Specification – 2001
  - ISO/IEC 19757-2 – 2003
- Schematron
  - alternative standard and alternative approach
  - several version since 1999
  - impact on XML Schema 1.1
**Element declaration in DTD**

- **Element name**
- **Element type**: one of:
  - **EMPTY**
  - **ANY**
  - (content specification)
- **Content specification** is built of:
  - element names
  - !PCDATA token
- Joint together using basic regular expression operators.
  - "1 !PCDATA is allowed only under special conditions

---

**Symbols in DTD element specifications**

- Parenthesis ( )
- Occurrence indicators (postfix operators)
  - ? - zero or one
  - * - zero or more
  - + - one or more
  - no symbol - exactly one
- Combination (infix associative operators)
  - , - sequence (in the given order)
  - | - choice (one of the given)

---

**Types in XML Schema**

**Concept of type** - one of basic distinctions wrt DTD

- Elements and attributes have specified types
- Type specify allowable content of an element / attribute
  - for elements - also their attributes
  - type spec. does not include identity constraints
- Type is independent of element (or attribute) name
  - many elements may have the same type
  - elements with the same name may have different types
  - “in different places”

---

**More details in examples!**

**Disclaimer**

Taking our experience and students’ opinions into account we will try not to copy standard specifications onto slides but rather to show by examples:

- some typical usage,
- different paths to do a thing - so you can choose your approach depending on needs,
- chosen cases of advanced usage and rarely used features - it is impossible to show all of them during a short lecture,
- some good and bad practices.

It also means, in particular, that slides are not a complete source of knowledge required to pass the exam.

**Basic things to look in the examples**

“students” - several ways to write a schema for the same document

- Structure of DTD, structure of XML Schema definition
- Typical element definition
- Controlling number of occurrences
- Sequence and choice
- Building complex models (nested groups)
- Defining attributes in schema and DTD

---

**Types - categorisation**

Types can be categorised with respect to:

- **complexity**
  - **complex types** define tree-level structure: subelements and attributes; they can be applied to elements only
  - **simple types** define text-level content; they can be applied to elements and attributes
- **scope**
  - **named types** are defined in global scope and can be used many times
  - **anonymous types** are defined in the place of use
- **origin**
  - predefined / built-in - provided by XML Schema
  - user-defined

---

**XML Schema**

- Replacement for DTD in new applications of XML
- Separate W3C standard
  - v 1.0 in 2001 - 3 recommendations
  - v 1.1 in 2012 - 2 recommendations
- “XML Schema definition” (*.xsd) is itself XML document
- Similar capabilities for tree-level structure specification
- Much more capabilities than in DTD for:
  - text-level content (“simple types”) “datatypes”
  - modularity of the definition (type inheritance, imports, namespace support)
  - identity constraints (keys and references)
  - in v 1.1 also more advanced constraints
- Much more verbose than DTD

---

**Element declaration**

```xml
<xs:schema xmlns="http://www.w3.org/1999/XMLSchema">
  <xs:element name="student" minOccurs="0" maxOccurs="unbounded">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first-name" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="last-name" type="xs:string" minOccurs="1" maxOccurs="1"/>
        <xs:element name="birth-date" type="xs:date" minOccurs="1" maxOccurs="1"/>
        <xs:element name="identification">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="PESEL" type="xs:string" minOccurs="1" maxOccurs="1"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

---

**More possibilities**

see lab classes

- Avoiding code duplication and different ways of writing definitions in schemas
  - Local definitions vs global definitions
  - Anonymous types vs named (global) types
  - Named groups
  - Extending complex types
  - Mixed content
    - DTD approach - (#PCDATA| a | b)*
    - Mixed content with controlled subelements - schema only
  - Any order (xs:all) - schema only
Element content defined with model groups:
- sequence – all in the given order
- choice – one of the given choices
- all – all given elements in any order
- sequence and choice may be nested, multiplied, etc.
- all – restricted
- may not be mixed with sequence and choice
- may not be nested
- can contain only elements with different names and occurrence number \( \leq 1 \)

### Namespaces – motivation

- Same names of tags may denote different things.
- Problematic especially when combining document fragments from different sources into one document.

### Namespaces – realisation

- **Namespace name** (identyfikator przestrzeni nazw) – globally unique identifier
  - Universal Resource Identifier (URI) in XML v1.0
  - Internationalized Resource Identifier (IRI) in XML v1.1
- **Namespace prefix** (prefiks przestrzeni nazw) – local, for convenient reference
  - Local for document or fragment
  - Processors should not depend on prefixes!
- Names resolved and interpreted as pairs: (namespace name, local name)
- To make things more complex:
  - scope and overriding
  - default namespace

### Usage of namespaces and prefixes

```xml
<art:article code="AI2509">
  <title>Assignment in Pascal and C</title>
  <author/>
  <fn:fn>
    <fn:name>fn</fn:name>
    <fn:surname>Hadrasiński</fn:surname>
    <fn:address>
      <fn:code>02-234</fn:code>
    </fn:address>
  </fn:fn>
  <body>
    <paragraph>
      Assignment is written as \$<code> = \$</code> in C and \$<code> := \$</code> in Pascal.
    </paragraph>
  </body>
</art:article>
```

### Namespaces – overriding and scopes

```xml
<pre:article code="AI2509" xmlns:pre="http://xml.nimw.edu.pl/">
  <pre:title>Assignment in Pascal and C</pre:title>
  <pre:author/>
  <fn:fn>
    <fn:name>fn</fn:name>
    <fn:surname>Hadrasiński</fn:surname>
    <fn:address>
      <fn:code>02-234</fn:code>
    </fn:address>
  </fn:fn>
  <pre:body>
    <pre:paragraph xmlns:pre="http://xml.nimw.edu.pl/">
      Assignment is written as \$<code> = \$</code> in C and \$<code> := \$</code> in Pascal.
    </pre:paragraph>
  </pre:body>
</pre:article>
```

### Namespaces – supplement

- **Qualified name** – name with non-empty ns.URI
- **Unqualified name** – name with null (not assigned) ns.
  - elements without prefixes when no default namespace
  - attributes without prefixes – always

### Default namespace

- Applies to element names which do not have a prefix.
- Does not apply to attributes.

```xml
<art:article code="AI2509" xmlns="http://xml.nimw.edu.pl/">
  <title>Assignment in Pascal and C</title>
  <author/>
  <fn:fn>
    <fn:name>fn</fn:name>
    <fn:surname>Hadrasiński</fn:surname>
    <fn:address>
      <fn:code>02-234</fn:code>
    </fn:address>
  </fn:fn>
  <body>
    <paragraph xmlns="http://xml.nimw.edu.pl/">
      Assignment is written as \$<code> = \$</code> in C and \$<code> := \$</code> in Pascal.
    </paragraph>
  </body>
</art:article>
```

### Module options

- Combining multiple files
  - DTD – external parameter entities
  - Schema – include, import, redefine
- Reusing fragments of model definition
  - DTD – parameter entities
  - Schema – groups and attribute groups
    - in practice equivalent to the above
- Schema – types, type derivation (no such feature in DTD)

### Global and local definitions

- In DTD all elements global, all attributes local
- In schema both can be global or local, depending on case

### Namespace awareness

- A document may be well-formed as XML while erroneous from the point of view of namespaces.
  - For some applications (usually old ones...) such document might be proper and usable.
- Modern parsers can be configured to process namespaces or not.
  - The mentioned document would be
    - parsed successfully by a parser which is not namespace-aware
    - revoked by a namespace-aware parser.

### XML namespaces – realisation

- **Namespace name** (identyfikator przestrzeni nazw)
  - globally unique identifier
  - Universal Resource Identifier (URI) in XML v1.0
  - Internationalized Resource Identifier (IRI) in XML v1.1
- **Namespace prefix** (prefiks przestrzeni nazw)
  - local, for convenient reference
  - Local for document or fragment
  - Processors should not depend on prefixes!
  - Names resolved and interpreted as pairs: (namespace name, local name)
  - To make things more complex:
    - scope and overriding
    - default namespace

### Usage of namespaces and prefixes

```xml
<art:article code="AI2509">
  <title>Assignment in Pascal and C</title>
  <author>
    <fn:fn>
      <fn:name>fn</fn:name>
      <fn:surname>Hadrasiński</fn:surname>
      <fn:address>
        <fn:code>02-234</fn:code>
      </fn:address>
    </fn:fn>
  </author>
  <body>
    <paragraph>
      Assignment is written as \$<code> = \$</code> in C and \$<code> := \$</code> in Pascal.
    </paragraph>
  </body>
</art:article>
```

### Namespaces – overriding and scopes

```xml
<pre:article code="AI2509" xmlns:pre="http://xml.nimw.edu.pl/">
  <pre:title>Assignment in Pascal and C</pre:title>
  <pre:author/>
  <fn:fn>
    <fn:name>fn</fn:name>
    <fn:surname>Hadrasiński</fn:surname>
    <fn:address>
      <fn:code>02-234</fn:code>
    </fn:address>
  </fn:fn>
  <pre:body>
    <pre:paragraph xmlns:pre="http://xml.nimw.edu.pl/">
      Assignment is written as \$<code> = \$</code> in C and \$<code> := \$</code> in Pascal.
    </pre:paragraph>
  </pre:body>
</pre:article>
```
Import or include?

- `xs:import`
  - Imports foreign definitions to refer to
- `xs:redefine`
  - Includes external definitions, but a local definition overrides external one if they share the same name
- `xs:include`
  - Basic command, almost like textual insertion
  - Imported module must have the same target namespace or no target namespace

A multi-module, namespace-aware project with overused `xs:include` leads to duplication of logic in the software that processes documents (or enforces meta-programming tricks to avoid it). /based on personal experience/

Schema and namespaces

- DTD is namespace-ignorant
- XML Schema conceptually and technically bound with XML namespaces
  - Basic approach: one schema (file) = one namespace
    - Splitting one ns into several files technically possible
  - Referring to components from other namespaces available
- Important attributes
  - `targetNamespace` - if given, all global definitions within a schema go into that namespace
  - `elementFormDefault`, `attributeFormDefault`
    - should local elements or attributes have qualified names?
      - default for both: unqualified
      - typical approach: elements qualified, attributes unqualified
      - setting may be changed for individual definitions

Using namespaces in XML Schema

Different technical approaches to handle namespaces in XML Schema

- XML Schema ns. bound to `xs` or `xsd`; no target namespace
- XML Schema ns. bound to `xs` or `xsd`; target namespace as default namespace
  - Convenient as long as we don't use keys and keyrefs
- Target namespace bound to a prefix (`tns`: by convention)
  - Then we can declare XML Schema as default namespace and avoid using `xs` or `xsd`:
  - See examples ns1.xsd - ns4.xsd
Modelling XML Applications (part 2)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 3 - 20.10.2014

Common design decisions
Element or attribute?
- Advantages of attributes:
  - more compact syntax
  - (only in DTD) some features available only for attributes
- Technical restrictions of attributes:
  - only text, without structure marked up
  - multiple attributes with the same forbidden
- General hints
  - Semantic hint: Use elements for data, attributes for metadata (whatever it means in your case ;)).
  - Presentational hint: If you had to print your document on paper, which parts of text would you print literally (they are elements) and which parts would only have some impact (or no impact) on the way things are presented (they should become attributes)?

Common design decisions
Names
- How descriptive (and long) should a name be?
- To use multipart names, or assume that the context is known?

Classification of types
Types by content model
- Simple type (value of a text node or an attribute; applicable to elements and attributes)
  - atomic type
  - list
  - union
- Complex type (structure model – subelements and attributes; applicable to elements)
  - empty content
  - element content
  - mixed content
  - simple content

Classification of types
Types by place of definition:
- anonymous – defined locally in place of use
- named – defined globally
  - built-in – defined in XML Schema specification
  - user-defined

Types by means of definition:
- primitive (simple types)
- defined directly (complex type as a sequence etc.)
- derived (some built-in types are defined by derivation!)
  - by extension (complex types only)
  - by restriction (complex and simple types)
  - as a list or union (simple types only)
Simple types
- Rich set of built-in types
  - decimal, integer, nonNegativeInteger, long, int, ...
  - boolean, float, double
  - date, time, dateTime, duration, ...
  - string, token, base64Binary, hexBinary, ...
  - See the recommendation for the complete hierarchy
- Defining custom types basing on built-in types
  - by restriction
  - as a list
  - as an union

Defining simple types by restriction
- Constraining facets – properties we can restrict
  - enumeration
  - pattern
  - length, minLength, maxLength
  - totalDigits, fractionDigits
  - maxInclusive, maxExclusive
  - minInclusive, minExclusive
  - whiteSpace

Value space vs lexical space
- A simple type specifies its
  - value space – set of abstract values
  - lexical space – set of valid text representations

<table>
<thead>
<tr>
<th>Type</th>
<th>Text representations</th>
<th>Abstract value</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:boolean</td>
<td>false</td>
<td>True</td>
</tr>
<tr>
<td>xs:decimal (and derivatives)</td>
<td>11.11</td>
<td>13</td>
</tr>
<tr>
<td>xs:string</td>
<td>'foo' 'bar'</td>
<td>'foo'</td>
</tr>
<tr>
<td>xs:token</td>
<td>'foo'</td>
<td>'bar'</td>
</tr>
</tbody>
</table>

Used directly in simple type definition:
```xml
<xs:simpleType name="lottoNumber"><
  <xs:restriction base="xs:integer">  
  <xs:minInclusive value="1"/>  
  <xs:maxInclusive value="49"/>  
  </xs:restriction>
</xs:simpleType>
```

Identity constraints
- Constraints on uniqueness and references
  - Two mechanisms:
    - DTD attribute types ID and IDREF
      - introduced in SGML DTD but still available in XML Schema
      - drawbacks:
        - one global scope, at most one ID per element
        - special form of values – only names allowed
        - IDs and references necessarily in attributes
    - XML Schema identity constraints
      - key, unique, and keyref definitions
      - more powerful and more flexible than ID/IDREF

List types
- List of values separated with whitespace.
- Not to confuse with sequences
  - list – simple type, no markup structure within
  - sequence – complex type, sequence of subelements
- Compact notation for lists of values
- Harder to process in XML processors (requires additional parsing using regexpr etc. - not available e.g. in XSLT 1.0)

Choosing the appropriate type
- Semantic meaning of a simple type:
  - not only a “set of allowed character strings”
  - also the way a value is interpreted!
- Types may affect the validation
  - e.g. leading zeros significant in strings, meaningless in numbers
- Processors may use the information about type, e.g.
  - schema-aware processing in XSLT 2.0 or XQuery
    - sorting, comparison, arithmetic operations
  - JAXB – generation of Java classes based on XSD
- Choosing the appropriate type sometimes not obvious
  - phone number, zip code, room number – number or string?

Global scope
- Identifiers unique within document scope
- References valid within document scope
- Expressible in DTD
  - using ID and IDREF, with some restrictions
  - and XML Schema
  - using key and keyref in the root element

Union types
- Union of sets of values
- Possibility to mix values of different primitive types
- Interpreting values as abstract values hard to perform
- Nevertheless, a usable feature (e.g. unbounded in XML Schema)

Global scope identifiers – DTD solution
- Restrictions:
  - one global scope for all elements
  - only names may become identifiers
  - (they have to start with a letter)
  - ids must be written in attributes, not elements

```
<database>
  <products>
    <product id="1">
      <nameComputer name="name"/>
      <price>250.00</price>
    </product>
    <product id="2">
      <nameTablet name="name"/>
      <price>1200.00</price>
    </product>
  </products>
</database>
```
XML Schema solution

```xml
<xs:element name="database">
    ...
    <xs:key name="Products"/>
    <xs:selector xpath="tns:products/tns:product"/>
    <xs:field xpath="@id"/>
    <xs:keyref name="ProductsInOrders" refer="tns:Products"/>
    <xs:field xpath="@ref"/>
</xs:element>
```

XML Schema keys and references

- Other possibilities: (see orders example)
  - local scope constraints
  - tuples as key values
- Pay attention:
  - only simple type values can become ids
  - type of reference source and target must be the same
  - default namespace does not work in XPath expressions so we have to use a prefix (usually tns:)
- Using references can help to avoid data duplication in documents
  - but overuse can make XML look like a flat relational database instead of a nice structural tree
  - find the golden mean!
XML Schema and alternatives

Patryk Czarnik
XML and Applications 2014/2015
Lecture 4 - 27.10.2014

Alternatives

- DTD - obviously
- RELAX NG (Regular Language for XML Next Generation)
  by James Clark and Murata Makoto
  OASIS (2001) and ISO (2003) standard
- Schematron
  by Rick Jelliffe (1999), developed at Academia Sinica
  (Taiwan), ISO standard (2006)
- Examplotron
  by Eric van der Vlist, project active in 2001-2003
- XML Schema 1.1
  W3C Recommendation, 2012
  borrows some ideas from the alternatives, mainly Schematron

Simple example in all standards

Schematron

```xml
/schema xmlns="http://purl.oclc.org/dtd/schematron">
  <pattern>
    <title>Person rules</title>
    <rule context="/person">
      <assert test="count(first-name) > 1">
        <message id="duplicate-first-name">
          Person should contain one or more first names.</message>
      </assert>
    </rule>
    <rule context="/person">
      <assert test="count(last-name) > 1">
        <message id="duplicate-last-name">
          Person should contain exactly one last name.</message>
      </assert>
    </rule>
    <rule context="/person">
      <assert test="not((local-name() = 'first-name' and local-name() = 'last-name'))">
        <message id="too-many-optional">
          Person should contain no other elements.</message>
      </assert>
    </rule>
  </pattern>
</schema>
```

Note: This is not the primary intended use of Schematron

Simple example in all standards

```xml
<fragment>
  <person>
    <first-name>Adam</first-name>
    <first-name>Maria</first-name>
    <last-name>Abacis</last-name>
  </person>
</fragment>
```

Simple example in all standards

```xml
<fragment>
  <person>
    <first-name>Adam</first-name>
    <second-name>Maria</second-name>
    <last-name>Abacis</last-name>
  </person>
</fragment>
```

Simple example in all standards

```xml
<fragment>
  <person>
    <first-name>Adam</first-name>
    <second-name>Maria</second-name>
    <last-name>Abacis</last-name>
  </person>
</fragment>
```

Simple example in all standards

```xml
<fragment>
  <person>
    <first-name>Adam</first-name>
    <second-name>Maria</second-name>
    <last-name>Abacis</last-name>
  </person>
</fragment>
```

Relax NG - basic ideas

- Clear theoretical basis: Tree automata with regular expressions specifying content in each node
  - The same model with appropriate restrictions is used by theoreticians to model DTD or XML Schema, but as a kind of "reverse engineering"
- Compact and readable XML syntax
  - Even more compact plain text syntax available
- Model components such as elements, attributes, or text nodes may be mixed together in definitions
- Modularity available through define / ref mechanism
  - Equivalent to DTD parameter entities or XSD groups, but with some additional operations to enhance convenience
- No direct support for simple types, but referring to XML Schema types is possible.
Schematron – basic ideas

- Approach different from grammar-based DTD, XSD, and RelaxNG:
  - XPath expressions specify assertions that must hold for instance documents (and elements within them)
  - High power of expression
  - Less convenient (than grammar rules) to write structural definitions
- Official implementation:
  - translation of Schematron scheme to XSLT
  - XSLT evaluates expressions and report errors
- Other implementations available, also for Schematron rules embedded in XML Schema definitions
- XML Schema 1.1 covers most typical Schematron use cases. Probably XSD 1.1 will replace Schematron at all.

Non-deterministic model

- Ambiguous model:
  - It is not possible to state which particle of the model definition is matched, even if whole document is known.
  - Example: (A,A,A)→(|A|A) for document AAAAAA
- Non-deterministic model
  - Similar to (non-JL1[1]) grammars, but extended to trees:
  - During one-pass parsing, it is not possible to determine the appropriate definition particle for a current element when only the start tag of the element is seen
  - Some models may be determined just by definition rearrangement, e.g.: A, A7, A7 → A, (A, A7)?
  - But some models may not; notable example: (A, B)*, A?
- XML Schema avoids non-deterministic models!

Model forbidden in XML Schema, valid in Relax NG

Value-aware constraints – Schematron solution

- We’d like to allow both formats:
  `<phone>1234567</phone>
  <main>123456</main>
  <main>int-113</main>
  <int-113</int-113></phone>`
- No possibility in XML Schema (other than mixed content)
- No problem in RelaxNG:
  ```xml
element name="phone">
  <choice>
    <text/>
  </choice>
  <group>
    <optional element name="cc="/optional>
    <optional element name="ac="/optional>
    <optional element name="main="/optional>
    <element name="int="/optional>
  </group>
  </element>
```

Choice between attribute and element

- Similarly as before, we want to allow both
  ```xml
  <section title="Introduction">
    ...,
  </section>
  <title>What does &lt;be or not to be?&gt; mean in fact?</title>
  ...,
  </section>
```
- Relax NG solution:
  ```xml
element name="section">
  <choice>
    <attribute name="title="/>
    <element name="text-model="/>
  </choice>
  <element>
    <ref name="text-model="/>
  </element>
```

Value-aware constraints – XSD 1.1 solution

- We want to choose one of models depending on the value of a field (attribute or element), e.g:
  ```xml
  <order>
    <order-date>2014-01-20</order-date>
    <delivery-date>2014-01-25</delivery-date>
    <item>
      ...,
      <value>199.99</value>
    </item>
    <item>
      ...,
      <value>29.99</value>
    </item>
    <item>
      ...,
      <value>219.99</value>
    </item>
  </order>
```

Value-aware constraints

- Relations and constraints other than equality
  ```xml
  <order>
    <order-date>2014-01-20</order-date>
    <delivery-date>2014-01-25</delivery-date>
    <item>
      ...,
      <value>199.99</value>
    </item>
    <item>
      ...,
      <value>29.99</value>
    </item>
    <item>
      ...,
      <value>219.99</value>
    </item>
  </order>
```

Content-aware model

- We want to choose one of models depending on the value of a field (attribute or element), e.g:
  ```xml
  <order>
    ...,
    <payment method="transfer"></payment>
  </order>
```

```xml
<order>
  ...,
  <payment method="transfer"></payment>
</order>
```

```xml
<order>
  ...,
  <payment method="card"></payment>
</order>
```
- It will work provided that type of payment will be defined generally and that all elements will have appropriate types
- But Schema 1.1 offers more natural solution...

So why do most people use XSD 1.0?

- Alternatives?
  - XML Schema is a W3C Recommendation
  - In this business it means even more than an ISO standard
  - Some popular technologies, with Web Services (WSDL) at the first place, use XML Schema

- XML Schema 1.1?
  - Still little support in generally available software, especially programming libraries
XML in Programming

Patryk Czarnik

XML and Applications 2014/2015
Lecture 5 – 3.11.2014

XML in programming – what for?

- To access data in XML format
- To use XML as data carrier (storage and transmission)
- To support XML applications (Web, content management)
- To make use of XML-related standards
  - (XML Schema, XInclude, XSLT, XQuery, XLink, …)
- To develop or make use of XML-based technology
  - XML RPC, Web Services (SOAP, WSDL)
  - REST, AJAX

XML in programming – how?

- Bad way
  - Treat XML as plain text and write low-level XML support from scratch
- Better approach
  - Use existing libraries and tools
- Even better
  - Use standardised interfaces independent of particular suppliers

XML and Java

- Propaganda
  - Java platform provides device-independent means of program distribution and execution.
  - XML is a platform-independent data carrier.

- Practice
  - Java - one of the most popular programming languages, open and portable.
  - Very good XML support in Java platform.
  - Many technologies use XML.

Of course you can find very good (or at least not bad) XML support on other programming platforms, but we have to choose one for presentation and exercises.

XML in Java – standards

Both included in Java Standard Edition since v.6

- Java API for XML Processing (JAXP 1.x – JSR-206)
  - many interfaces and few actual classes, “factories” and pluggability layer
  - support for XML parsing and serialisation (DOM, SAX, StAX)
  - support for XInclude, XML Schema, XPath, XSLT

- Java API for XML Binding (JAXB 2.x – JSR-222)
  - binding between Java objects and XML documents
  - annotation-driven
  - strict relation with XML Schema

XML in Java – standards

- W3C Recommendations
  - DOM Level 1 - 1998
  - DOM Level 3 - 2004
  - Several modules. We focus on DOM Core here

- Document model and universal API
  - independent of programming language (IDL)
  - independent of particular XML application

- Used in various environments
  - notable role in JavaScript / ECMA Script model
  - available (in some form) for all modern programming platforms

Classification of XML access models

And their “canonical” realisations in Java

- Document read into memory
  - generic interface: DOM
  - interface depending on document type/schema: JAXB

- Document processed node by node
  - event model (push parsing): SAX
  - streaming model (pull parsing): StAX

Primary DOM types

- Node
- NodeList
- NamedNodeMap
- Document
- Element
- Comment
- Attr
- Text
- Processing Instruction
- CDATA Section

DOM key ideas

- Whole document in memory
- Tree of objects
- Generic interface Node
- Specialised interfaces for particular kinds of nodes
- Available operations
  - reading document into memory
  - creating document from scratch
  - modifying content and structure of documents
  - writing documents to files / streams
Example: problem introduction

Count the number of seats in rooms equipped with a projector:

```xml
<rooms>
  <room>
    <number>2136</number>
    <floors>2</floors>
    <equipment>projector=false</equipment>
    <computers=false/>
    <seats=8</seats>
  </room>
  ...
</rooms>
```

Whole example in CountSeats.DOM_Generic.java

DOM in Java example

Parsing and basic processing

```java
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(fileName);
for(Node node = doc.getFirstChild();
    node != null;
    node = node.getNextSibling()) {
  if(node.getNodeType() == Node.ELEMENT_NODE
      && node.nodeName().equals(node.getNodeName())) {
    this.processRoom(node);
  }
}
```

Whole example in CountSeats_DOM_Generic.java

DOM in Java example

Visiting nodes in the tree

```java
private void processRooms(Node rootNode) {
  for(Node node = rootNode.getFirstChild();
      node != null;
      node = node.getNextSibling()) {
    if(node.getNodeType() == Node.ELEMENT_NODE
      && node.nodeName().equals(node.getNodeName())) {
      this.processRoom(node);
    }
  }
}
```

Whole example in CountSeats_DOM_Specialized.java

DOM in Java example

Access to attributes and text nodes

```java
if(equipmentNode != null) {
  NameNodeMap equipmentAttributes = equipmentNode.getAttributes();
  Node projectorNode = equipmentAttributes.getAttributeByName("projector");
  if(projectorNode != null) {
    String projector = projectorNode.getNodeValue();
    if("true".equals(projector) || "false".equals(projector)) {
      hasProjector = true;
    }
  }
}
```

Whole example in CountSeats_DOM_Generic.java

Approaches to using DOM

- Two approaches in DOM programming
  - Use only generic Node interface
  - Use specialised interfaces and convenient methods

Example features of specialised Element interface:
- searching the subtree for elements of the given name
  - `getElementsByTagName`
  - `getElementsByTagNameNS`
- direct access to attribute values
  - `getAttribute`
  - `getAttributeNS`
  - `setAttribute`
  - `setAttributeNS`

Whole example in CountSeats_DOM_Specialized.java

JAXB 2.0 architecture

- Application operates basing on (usually annotated) "JAXB classes"
  - generated from a schema
  - or written manually

Whole example in CountSeats_DOM_Specialized.java

JAXB example

- We generate java classes basing on our schema
  - `xjc -d src -p package_name school.xsd`

- One of generated classes:
  ```java
  @Named(type=PackageInfo.NAME)
  class Room {
    private Element equipmentElement = (Element) roomElement;
    @XmlElement(name = "equipment").item(0);
    public String getEquipmentElements() {
      String equipment = equipmentElement.getAttribute("equipment");
      return equipment;
    }
  }
  ```

Whole example in CountSeats_DOM_Specialized.java

XML binding and JAXB

- Mapping XML to Java
- High-level view on documents:
  - instead of `Document`:
    - `getElementsByTagName("seats")`
    - `item(0)`
    - `getTextContent()`
- we simply have:
  - `room.getSeats()`

Whole example in CountSeats_DOM_Specialized.java

JAXB example

- We generate java classes basing on our schema
  - `xjc -d src -p package_name school.xsd`

- One of generated classes:
  ```java
  @Named(type=PackageInfo.NAME)
  class Room {
    private Element equipmentElement = (Element) roomElement;
    @XmlElement(name = "equipment").item(0);
    public String getEquipmentElements() {
      String equipment = equipmentElement.getAttribute("equipment");
      return equipment;
    }
  }
  ```

All generated classes are in ..._jaxb_generated and the program in CountSeats_JAXB
JAXB example

```java
JAXBContext jaxbContext = JAXBContext.newInstance(Rooms.class);
Unmarshaller u = jaxbContext.createUnmarshaller();
Rooms rooms = (Rooms) u.unmarshal(new File(file));
if(rooms != null) {
    this.processRooms(rooms);
}
```

```java
private void processRooms(Rooms rooms) { 
    for(Room room : rooms.getRoom()) { 
        if(room.getEquipment().isProjector()) 
            room = room.getSeats(); 
    }
}
```

JAXB - applications and alternatives

- Primary applications:
  - high-level access to XML documents
  - serialisation of application data
  - automatic mapping of method invocations to SOAP messages in JAX-WS

- Many options to customise the mapping using Java or XML annotations

- Some alternatives:
  - Castor
  - Apache XML Beans
  - JIBX

SAX

- Simple API for XML - version 1.0 in 1998
- Original standard designed for Java
- Idea applicable for other programming languages

Typical usage:

- Programmer-provided class implementing `ContentHandler`
- Optionally classes implementing `ErrorHandle`, `DTDHandler`, or `EntityResolver`
- one class may implement all of them
- `DefaultHandler` - convenient base class to start with

SAX events in run

```xml
<?xml-stylesheet ...?>
<room>
    <equipment projector="true"/>
    <seats>
        60
    </seats>
</room>
```

SAX example (fragments)

```java
public class CHandler implements ContentHandler {
    ...
    public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException {
        switch(state){
            ...
            case IN_ROOM:
                if("equipment".equals(qName)) {
                    String projector = atts.getValue("projector");
                    if("true".equals(projector)) |
                        state = CHandler.States.IN_ROOM_WITH_PROJECTOR;
                }
        }
    }
    ...
}
```

Streaming (and event) processing

Motivation

- Whole document in memory (DOM, JAXB)
  - convenient
  - but expensive
    - memory for document
      (multiplied by an overhead for structure representation)
    - time for building the tree
    - reading always whole document, even if required data present at the beginning
    - sometimes not possible at all
      - more memory required than available
      - want to process document before it ends
  - Alternative: Reading documents node by node

SAX

Typical usage (ctnd):

- Obtain `XMLReader` (or `SAXParser`) from factory
- Create `ContentHandler` instance
- Register handler in reader
- Invoke `parse` method
  - Parser conducts processing and calls methods of our `ContentHandler`
- Use data collected by `ContentHandler`

SAX examples

- See whole example classes:
  - `CountSeats_SAX_Traditional` and `CHandler_Traditional` for traditional scenario of creating parses instance and registering a `ContentHandler`
  - `CountSeats_SAX_JAXP` and `CHandler_JAXP` for modern JAXP-conformant scenario of combining things together
SAX filters

- Motivation: Joining ContentHandler-like logic into chains
- Realisation:
  - interface XMLFilter
    (XMLReader having a parent XMLReader)
  - in practice filters implement also ContentHandler
  - convenient start-point: XMLFilterImpl
- Typical implementation of a filter:
  - handle incoming events like in a ContentHandler
  - pass events through by manual method calls on the next item in chain
- Filters can:
  - pass or halt an event
  - modify an event or a sequence of events!

Possible usage of SAX filters

SAX – typical problems

- To make implementations portable – we should manually join adjacent text nodes in an element
  - StringBuilder is a convenient class
- The same method called for different elements, in different contexts
  - Typical solution – remembering the state:
    - one boolean flag in simplest cases
    - enum is usually enough
    - elaborated structures may be required for complex logic
- It may become tedious in really complex cases.

StAX: Pull instead of being pushed

- Alternative for event model
  - application “pulls” events/nodes from parser
  - processing controlled by application, not parser
  - idea analogous to: iterator, cursor, etc.
- More intuitive control flow
  - reduced need of remembering the state etc.
- Advantages of SAX saved
  - high efficiency
  - possibility to process large documents

StAX Examples

- We’re not going to resolve our example program using filters, as it makes a little sense.
- An example filter can be found in more_sax/UpperCaseFilter

StAX – example?

- Streaming API for XML
  - Available in Java SE since version 6
  - Two levels of abstraction:
    - XMLStreamReader
      - one object for all purposes
      - most efficient approach
    - XMLReader
      - subsequent events (nodes) provided as separate objects
      - more convenient for high-level programming, especially when implementing modification of the document “on-the-fly”

StAX example with XMLStreamReader (fragments)

```java
XMLInputFactory xif = XMLInputFactory.newInstance();
reader = xif.createXMLStreamReader(new FileInputStream(fileName));
while (reader.hasNext()) {
  if (reader.isEndElement())
    return;
  else if (reader.isStartElement())
    if (element.getAttributeValue("type") == "string")
      String name = reader.getAttributeValue("name");
      System.out.println(name);
```
### Control flow in SAX

**Diagram**

- `program` int
- `setContentHandler`
- `parser`
- `startElement` characters, etc.
- `endDocument`
- `programmer's code`
- `parser internal code`

### Which model to choose? (1)

- **Document tree in memory:**
  - small documents (must fit in memory)
  - concurrent access to many nodes
  - creating new and editing existing documents "in place"
- **Generic document model (like DOM):**
  - not established or not known structure of documents
  - lower efficiency accepted
- **XML binding (like JAXB):**
  - established and known structure of documents
  - XML as a data serialisation method

### Control flow in StAX

**Diagram**

- `program`
- `createXMLEventReader`
- `nextEvent` `StartElement`
- `nextEvent` `Characters`
- `result`
- `close`
- `programmer's code`
- `parser internal code`

### Which model to choose? (2)

- **Processing node by node**
  - potentially large documents
  - relatively simple, local operations
  - efficiency is the key factor
- **Event model (SAX):**
  - using already written logic (SAX is more mature)
  - filtering events, asynchronous events
  - several aspects of processing during one reading of document (filters)
- **Streaming model (like StAX):**
  - processing depending on context; complex states
  - processing should stop after the item is found
  - reading several documents simultaneously

### StAX features

- **API for reading documents:**
  - `XMLStreamReader`, `XMLEventReader`
- **API for writing documents:**
  - `XMLStreamWriter`, `XMLEventWriter`
- **Filters**
  - simple definition of a filter: `accept(Event): boolean`
  - "filtered readers"
## Applications of Transformers

- **Simple:**
  -invoking XSLT transformations from Java
  -changing internal representation of XML in our program

- **Tricky:**
  - parsing and writing documents, e.g., serialization of a DOM tree
  - serialization of modified (or generated) sequences of SAX events
  - (together with SAX filters) enabling “on-the-fly” processing of large XML documents

## Handling errors

- Most JAXP components (specifically SAX and DOM parsers, Validators)
  - may throw SAXException
  - signal errors through ErrorHandler events
- Interface ErrorHandler
  - 3 methods (and severity levels): warning, error, fatalError
  - registering with setErrorHandler allows to override default error handling
- Required to manually handle validation errors

## Features of JAXP

- 3 models of XML documents in Java: DOM, SAX, StAX
- Formally JAXB is a separate specification
- Reading and writing documents
- Transformations of XML documents (Transformers)
  - applying XSLT in our programs
  - translating internal form of representation
- XPath support
- Validation
  - against DTD (only during parsing)
  - against XML Schema (during parsing or using Validators)
  - against XML Schema 1.1, Relax NG, or other alternative standards – when implementation supports

## Editing XML documents

- More natural when whole document present in memory
  - DOM - generic API
  - JAXB – deep embedding of XML in application model
- Harder, but possible, using node-by-node processing
  - required when processing big documents while having little memory
  - suggested for big (“long and flat”) documents and simple local operations – then we can save substantial resources
- StAX – possible using “writers”
  - IMO XMLStreamWriter more convenient than XMLStreamWriter
  - SAX
    - no direct support for editing/writing
    - available indirect solution: SAX filters and Transformer

## XPath support in Java

- DOM XPath module implementation
  - org.w3c.dom.xpath
  - officially not a part of Java SE, but available in practice (by inclusion of Xerces in Oracle Java SE runtime)
- JAXP XPath API
  - javax.xml.xpath
  - most efficient when applied for documents in memory (DOM trees)
  - our examples show this solution
- Note: using XPath may significantly reduce developer’s work, but the application may be less efficient (than if we used SAX, for example)

## Transformer: source and result

- Against DTD
  - setValidating(true) before parsing
- Against XML Schema (or other schema formats, if implementation supports)
  - setSchema(schema) before parsing
  - using Validator
- Validator API
  - validate(Source) – only checking of correctness
  - validate(Source, Result) – augmented document returned
    - not possible to use as Transformer – source and result must be of the same kind
    - (my private observation) – not always working as expected
Web services

Patryk Czarnik
XML and Applications 2014/2015
Lecture 7 - 24.11.2014

Pre-XML solutions

- ANSI Accredited Standards Committee X12 sub-group
  - USA national standard
  - used mainly in America
- EDIFACT
  - international standard (UN/CEFACT and ISO)
  - used mainly in Europe and Asia

EDIFACT structure

- EDIFACT characteristic
  - Format
    - text
    - hardly readable
    - tree structure
  - Predefined dictionaries
  - 193 message types
  - 279 segments
  - 186 elements
  (counted for version 08a, 2008)

XML EDI

- Idea: use XML as data format for EDI
- Traditional EDI
  - Documents unreadable without specification
  - Compact messages
  - Centralised standard maintenance
  - Changes in format requires software change
  - Specialised tools needed
- XML EDI
  - “Self-description” documents format
  - Verbos messages
  - “Plugable”, flexible standards
  - Well written software ready to extensions of format
  - XML-format layer handled by general XML libraries

Electronic data interchange (EDI) – motivation

- How to interchange data between companies / institutions (B2B)?
  - paper
  - electronic data interchange
- How to establish EDI protocol?
  - customer receives (or buys) a tool from provider
  - smaller partner complies to bigger partner
  - ad-hoc created conversion tools
- Standard
  - Standard deployment levels
  - software developed according to standard from beginning
  - interface added to legacy system

EDIFACT

- EDIFACT message example

XML EDI flexibility

- Format flexibility
  - Structures: choosing, repeating, nesting, optionality
  - Format extensions and mixing via namespaces
- Applications
  - Data interchange between partners’ systems
  - Web interface (with little help from XSLT)
  - Web Services integration
Web Services

- Idea: a website for programs (instead of people)
- General definition
  - communication based on high-level protocols
  - structural messages
  - services described
  - searching services
- Concrete definition: “Classical” Web-Services
  - HTTP or other protocols
  - SOAP
  - WSDL
  - UDDI
- Web Services Interoperability

Web Services standardisation

- SOAP (initially Simple Object Access Protocol):
  - beginnings: 1998
  - v1.1: W3C Note, 2001 (still in use)
  - v1.2: W3C Recommendation, June 2003 (also used)
- Web Services Description Language:
  - W3C Note, 2001 (most applications use this version!)
  - v2.0: W3C Recommendation, June 2007
- Universal Description Discovery and Integration:
  - OASIS project

Web Services standardisation (2)

- Web Services Interoperability - levels of WS compliance:
  - WS-I Basic Profile, Simple Soap Binding Profile, ...
  - WS-* standards: various standards, usually not W3C:
    - WS-Eventing, WS-Addressing, WS-Routing, WS-Security
- Business Process Execution Language (OASIS) - WS semantics description, programming using WS as building blocks

Classical vision of web services operation

- Service (and its provider)
- SOAP
- SOAP
- Service registry (UDDI)
- SOAP
- SOAP
- In fact, most deployed solutions don’t use the UDDI layer

SOAP 1.2 message

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Trans xmlns:t="http://www.w3schools.com/transaction/"
             soap:mustUnderstand="1">234</t:Trans>
  </soap:Header>
  <soap:Body>
    <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
      <m:ItemApplied>m:Item</m:ItemApplied>
      <m:Currency>PLN</m:Currency>
    </m:GetPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2 - communication protocol

- Built on top of existing transport protocol (HTTP or other)
- Message format
  - XML message with optional binary attachments
  - headers (optional XML elements) and body content
  - envelope and some special elements defined in standard
    - implementation-dependent content
- Differences to RPC, CORBA, DCOM etc.:
  - data represented in extensible, structural format (XML)
  - data types independent of platform (XML Schema)
  - lower efficiency

SOAP message – general form

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <soap:Fault>
      <soap:Code soap:ValueType="soap:Fault">
        <soap:Detail soap:Type="soap:Fault">
          <soap:actor>urn:exception</soap:actor>
          <soap:detail>Found no student identified with <soap:student>123</soap:student></soap:detail>
        </soap:Detail>
      </soap:Code>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2 - normal response

```xml
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
               soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Body>
    <m:GetPriceResponse xmlns:m="http://www.w3schools.com/prices">
      <m:Price>1.50</m:Price>
      <m:Currency>PLN</m:Currency>
    </m:GetPriceResponse>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2 - fault response
SOA and Web Services give an opportunity to build modular, flexible, and scalable solutions. Sometimes, by the cost of irrational inefficiency and complexity.

Web Service recommended when
- Many partners or public service (standardisation)
- Heterogeneous architecture
- Text and structural data already present in problem domain
- Interoperability and flexibility more important than efficiency

Web Service?... not necessarily
- Internal, homogeneous solution.
- Binary and flat data
- Efficiency more important than interoperability
Web services in Java

- Web services and web service clients can be built from scratch in any technology
  - but it would be the same mistake as reading XML documents char by char.
- Low-level technologies:
  - HTTP servlets and HTTP clients supported by XML processing APIs (DOM, SAX, StAX, JAXB, Transformers, ...)
  - SOAP with Attachments API for Java (SAAJ)
- High level approach (with low level hooks available):
  - Java API for XML Web Services (JAX-WS)

JAX-WS – introduction

- Annotation-driven
- Uses JAXB to translate Java objects to/from XML
- Central point: Service Endpoint Interface (SEI)
  - Java interface representing a WS port type
    - kalkulator.Kalkulator and pakiet.Service in our examples
- Translation between web services world (WSDL) and Java
  - top-down: from WSDL generate Java
  - bottom-up: from Java code generate WSDL (and treat the Java code as a WS implementation)

JAX-WS – low level hooks

- Providers – low level server side
  - Useful when
    - high efficiency required (e.g. streaming processing)
    - XML technology used in implementation
- Dispatch – low level client side
- One way methods
- Asynchronous client calls
- Handlers and handler chains
  - additional processing of messages between client and server logic
  - one place to perform common logic: logging, authentication, session binding

Web services in Java

- WS support (XML APIs, SAAJ, JAX-WS) present in Java SE
- JAX-WS and some of XML APIs since version 6.0
- Client side:
  - Possible to develop and run WS client in Java SE without any additional libraries!
- Server side:
  - Developing and compiling WS server (without any vendor-specific extensions) available in Java SE
  - Running a service requires an application server and a WS implementation
    - “Big” app servers (Glassfish, jBoss, WebSphere...) have preinstalled WS implementations
    - Lightweight servers (e.g. Tomcat) can be used by applications equipped with appropriate libraries and configuration

Advantages and risks of using JAX-WS

- High level view on web service
  - details of communication and SOAP/XML not (necessarily) visible to a programmer
  - proxy object on client side enables to transparently invoke methods on server-side just like on local objects
- Automatic generation/interpretation of WSDL
  - conformance to WSDL controlled by system
- Bottom-up scenario – easy introduction of WS interface to already existing systems
  - or for programmers not familiar with WSDL/XML details
- Risk of accidental service interface (WSDL)
  - (automatically generated, not elaborated enough)
  - inefficiency

JAX-WS examples

Details to note:

- top-down (Kalkulator):
  - (different) form of WSDL in RPC and Document styles
  - 3 ways WSDL can be translated to Java (and SOAP)
    - (RPC, document-wrapped, document-bare)
  - @WebService annotation in implementation class
- bottom-up (Hello)
  - how annotations affect SOAP messages (and WSDL)
  - how Java objects are represented in SOAP messages (JAXB)
  - high level proxy clients (Client_Weather_JAXWS)

SAAJ

- Package javax.xml.soap
- Main class – SOAPMessage
- Tree-like representation of SOAP messages
  - extension of DOM
  - easy access to existing and building fresh SOAP messages
  - support for HTTP headers, binary attachments, ...
- Easy sending of requests from client side
  - see example Client_Weather_SAAJ
- Possible implementation of server side as a servlet
  - see example Server_SAAJ

JAX-WS – main elements

- Class level annotations:
  - @WebService, @SOAPBinding
- Method-level annotations:
  - @Method, @OneWay, @SOAPBinding, @RequestWrapper, @ResponseWrapper
- Parameter-level annotations:
  - @WebParam
  - @Result (syntactically a method annotation, applies to what the method returns)
- Support for specific technologies
  - @MTOM – automatically created binary attachments
  - @Addressing – adds WS-Addressing headers

JAX-WS architecture

When both sides written in Java...

High level java clients available also for non-Java servers!
REST – motivation

- Complexity and inefficiency of SOAP-based services led designers/researchers to propose other solutions
  - service-oriented
  - but simpler (and less general) than classical WS
- The most popular alternative these days: Representational State Transfer (REST)
  - Very popular solution for integration of JavaScript clients (AJAX) with servers
  - And mobile clients as well...
  - In Java (EE) available through JAX-RS interface

REST – basic ideas

- Service = set of resources
  - resource identified by its URL
  - best practices: URLs unique, resources organised in collections
    - http://rest.example.org/service/orders/302312
- Resources
  - are representable
    - e.g. as XML
  - other formats available, a popular one is JSON
  - can be transferred through the net
- HTTP – protocol for remote access to the resources
  - HTTP methods (GET, PUT, etc) used directly

HTTP methods (in REST, but not only)

- GET - read the resource
  - no side effects
- PUT – write the resource
  - request body contains new contents
  - for writing new and overriding existing resources
- DELETE – deletes the resource
- POST – “take this piece of data and do something with it”
  - conceptually incompatible with REST ideas
  - used in practice to call remote logic more complex that reading or writing a resource
- OPTIONS, HEAD – no special meaning in REST
  - well, getting last modification time makes sense in REST...

JAX-RS – REST in Java

- Java API for RESTful Services (JAX-RS)
- Annotation driven API
- Support for different ways of passing arguments
- Content-type negotiation
  - the same resource may be available in different formats
- Easy to write HTTP servers
  - REST-specific logic has to be written manually
XPath (and XQuery)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 8 – 1.12.2014

XPath – status

- XPath 1.0
  - W3C Recommendation, XI 1999
  - used within XSLT 1.0, XML Schema, XPointer
- XPath 2.0
  - Several W3C Recommendations, 7 2007:
    - XML Path Language (XPath) 2.0
    - XQuery 1.0 and XPath 2.0 Data Model
    - XQuery 1.0 and XPath 2.0 Functions and Operators
    - XQuery 1.0 and XPath 2.0 Formal Semantics
  - Used within XSLT 2.0
  - Related to XQuery 1.0
- XPath 3.0
  - Several W3C Recommendations, IV 2014

XPath (and XQuery) Data Model

- Theoretical base of XPath, XSLT, and XQuery
- XML document tree
- Structures and simple data types
- Basic operations (type conversions etc.)
- Model different in different versions of XPath
  - 1.0 – 4 value types, sets of nodes
  - 2.0 & 3.0 – XML Schema types, sequences of nodes and other values

Models of XML processing

- Text level processing
  - possible but inconvenient and error-prone
- Custom applications using standardised API
  (DOM, SAX, JAXP, etc.)
  - flexible and (relatively) efficient
  - requires some work
- XML-related standards with high-level view on documents
  - XPath, XQuery, XSLT
  - XML-oriented and (usually) more convenient than above
  - sometimes not flexible enough
- “Off the shelf” tools and solutions

Version numbering

Subsequent generations of related standards.

<table>
<thead>
<tr>
<th>When</th>
<th>XPath</th>
<th>XSLT</th>
<th>XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>1.0</td>
<td>1.0</td>
<td>-</td>
</tr>
<tr>
<td>2007</td>
<td>2.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2014</td>
<td>3.0</td>
<td>3.0( WD)</td>
<td>3.0</td>
</tr>
</tbody>
</table>

XPath and XQuery

Querying XML documents

Common properties
- Expression languages designed to query XML documents
- Convenient access to document nodes
- Intuitive syntax analogous to filesystem paths
- Comparison and arithmetic operators, functions, etc.

XPath

- Used within other standards:
  - XSLT
  - XML Schema
  - XPointer
  - DOM
- Standalone standard
- Extension of XPath
- Main applications:
  - XML data access and processing
  - XML databases

XQuery

Paths – typical XPath application

- /company/department/person
- //person
- /company/department[name = ‘accountancy’]
- /company/department[@id = ‘D07’]/person[3]
- /surname
- surname
- ../person[position = ‘manager’]/surname

But there is much more to learn here...

XML document in XPath model

- Document as a tree
- Physical representation level fully expanded
  - CDATA, references to characters and entities
  - No adjacent text nodes
- Namespaces resolved and accessible
- XML Schema applied and accessible
  - XPath 2.0 “schema aware” processors only
- Attribute nodes as element “properties”
  - formally, attribute is not child of element
  - however, element is parent of its attributes
- Root of tree – document node
  - main element (aka document element) is not the root

Document tree – example

```
<?xml version="1.0"?>
<sql-stylesheet href="style.css">
  <person id="77">
    <name>John</name>
    <surname>Smith</surname>
    <tel>123343456</tel>
    <tel>13113</tel>
    <id>77</id>
  </person>
</sql-stylesheet>
```
**XPath node kinds**
- Seven kinds of nodes:
  - document node (root)
  - element
  - attribute
  - text node
  - processing instruction
  - comment
  - namespace node
- Missing ones (e.g. when compared to DOM):
  - CDATA
  - entity
  - entity reference

**Data model in XPath 1.0**
- Four types:
  - boolean
  - string
  - number
  - node set
- No collections of simple values
- Sets (and not sequences) of nodes

**Literals and variables**
- **Literals**
  - strings:
    - value
    - "He said, "I don't like it.""
  - numbers:
    - 12
    - 12.5
    - 1.13e-8
- **Variables**
  - $x$ – reference to variable x
  - Variables introduced with:
    - XPath 2.0 constructs (for, some, every)
    - XQuery (FLWOR, some, every, function parameters)
    - XSLT 1.0 and 2.0 (variable, param)

**Sequences**
- Values in XPath 2.0 – sequences
- Sequence consists of zero or more items
  - nodes
  - atomic values
- Sequences properties
  - Items order and number of occurrence meaningful
  - Singleton sequence equivalent to its item
  - 3.14 = (3.14)
- Nested sequences implicitly flattened to canonical representation:
  - (3.14, (1, 2, 3), 'A[a]') = (3.14, 1, 2, 3, 'A[a]')

**Effective Boolean Value**
- Treating any value as boolean
- Motivation: convenience in condition writing, e.g. if (customer(@passport)) then
- Conversion rules
  - empty sequence → false
  - sequence starting with a node → true
  - single boolean value → that value
  - single empty string → false
  - single non-empty string → true
  - single number equal to 0 or NaN → false
  - other single number → true
  - other value → error

**Type casting**
- **Type constructors**
  - xs:date("2010-08-25")
  - xs:float("NaN")
  - address: kód-pocztowy("48-200") (schema aware processing)
  - string://objetk[4]) (valid in XPath 1.0 too)
- **Cast operator**
  - "2010-08-25" cast as xs:date

**Type system**
- Treating any sequence as a sequence of atomic values
  - often with an intention to get a singleton sequence
- Motivation: comparison, arithmetic, type casting
- Conversion rules (for each item)
  - atomic value → that value
  - node of declared atomic type → node value
  - node of list type → sequence of list elements
  - node of unknown simple type or one of xs:untypedAtomic, xs:atomicType → text content as single item
  - node with mixed content → text content as single item
  - node with element content → error

**Atomization**
- Function invocation:
  - concat('Mrs ', name, ' ', surname)
  - count(/person)
  - my:factorial(12)
- 150 built-in functions in XPath 2.0, 27 in XPath 1.0
- Abilities to define custom functions
  - XQuery
  - XSLT 2.0
  - execution environment
  - EXSLT - de-facto standard of additional XPath functions and extension mechanism for XSLT 1.0
Chosen built-in XPath functions

- Text:
  - concat($s1, s2, ...)
  - substring($s, pos, len)
  - starts-with($s1, $s2)
  - contains($s1, $s2)
  - string-length($s)
  - translate($s, $t1, $t2)

- Numbers:
  - floor($x)
  - ceiling($x)
  - round($x)

- Nodes:
  - name($n)
  - local-name($n)
  - namespace-uri($n)

Sequences (some only since XPath 2.0):
- count($S)
- sum($S)
- min($S)
- max($S)
- avg($S)
- empty($S)
- reverse($S)
- distinct-values($S)

Context:
- current()
- position()
- last()

General comparison – nonobvious behaviour

- Equality operator does not check the real equality
  - $(1, 2) != (1, 2)$ → true
  - $(1, 2)$ = $(2, 3)$ → true

- “Equality” is not transitive
  - $(1, 2)$ = $(2, 3)$ → true
  - $(2, 3)$ = $(3, 4)$ → true
  - $(1, 2)$ = $(3, 4)$ → false

- Inequality is not negation of equality
  - $(1, 2)$ = $(1, 2)$ → true
  - $(1, 2)$ != $(1, 2)$ → true
  - $(1, 2)$ != $(1, 2)$ → false

Sequence quantifiers (XPath 2.0)

- some $VAR$ in SEQUENCE satisfies CONDITION
- every $VAR$ in SEQUENCE satisfies CONDITION
  - Using Effective Boolean Value of CONDITION
  - Lazy evaluation allowed
  - Evaluation order not specified

Example
some $x$ in (1 to 10) satisfies $x > 7$
every $y$ in //person satisfies $ysurname$

Operators

- Arithmetic
  - + - *
  - div
  - idiv
  - mod

- Logical values
  - and
  - or
  - true(), false(), and not() are functions

- Node sets / sequences
  - union
  - intersect
  - except
  - not

- Node sets found - type error
- result without repeats, document order preserved

- Nodes
  - is
  - <=
  - >=

Conditional expression (XPath 2.0)

if ($CONDITION$) then $RESULT1$ else $RESULT2$

- Using Effective Boolean Value of $CONDITION$
- One branch evaluated

Example
if(details/price) then
  if(details/price == 1000) then 'Ensured mail'
  else 'Ordinary mail'
else 'No data'

Paths – more formally

- Absolute path: /step/step ...
- Relative path: step/step ...
- Step – full syntax:
  - axis::node-set [predicate] ...
  - axis – direction in document tree
  - node-test – selecting nodes by kind, name, or type
  - predicates – (0 or more) additional logical conditions for filtering

Comparison operators

- Atomic comparison (XPath 2.0 only)
  - eq ne le lt le ge ge
  - applied to singletons

- General comparison (XPath 1.0 and 2.0)
  - = != < <= > >=
  - applied to sequences

XPath 2.0 semantics:
There exists a pair of items, one from each argument sequence, for which the corresponding atomic comparison holds. (Argument sequences atomized on entry.)

Typical usage
books/price > 100
"At least one of the books has price greater than 100"

Iteration through sequence (XPath 2.0)

for $VAR$ in SEQUENCE
return $RESULT$

- VAR takes subsequent values from SEQUENCE
- $RESULT$ computed that many times
  - in context where $VAR$ is assigned the given value
  - overall result - (flattened) sequence of partial results

Example
for $i$ in (1 to 10)
  return $i$ * "$i$
for $x$ in //object
  return concat('Mazwa objektu: ', $x/mazwa')

Axis

- self
- child
- descendant
- parent
- ancestor
- following-sibling
- preceding-sibling
- following
- preceding
- attribute
- namespace
- descendant-or-self
- ancestor-or-self
**Axis**

- By kind of node:
  - `node()`
  - `text()`
  - `comment()`
  - `processing-instruction()`
- By name (examples):
  - `person`
  - `pre:person`
  - `pre:*`
  - `*:person` *(XPath 2.0 only)*
  - `*`
  - kind of node here: element or attribute, depending on axis

**Predicates**

- Evaluated for each node selected so far (node becomes the context node)
- Every predicate filters result sequence
- Depending on result type:
  - number – compared to item position (counted from 1)
  - not number – Effective Boolean Value used
- “Filter expressions” – predicates outside paths

**Examples**

```
[child::staff[child::person[child::name = 'Patryk']]
[child::person[child::name = 'Patryk']]/child::surname
//person[attribute::passport][3]
[1 to 10]::mod 2 = 0]
```

**Node test**

- child axis may be omitted
- `@` before name indicates attribute axis
- `.` stands for `self::node()`
- `..` stands for `parent::node()`
- `//` translated to `/descendant-or-self::node/`
  (textually, inside an expression)

**Abbreviated Syntax**

```
../object[uid = 'E4']
```

**Node test in XPath 2.0**

In XPath 2.0 more tests, basing on kinds of nodes, and schema-provided types of nodes (*schema aware* only).

- `document-node()`
- `processing-instruction(xml-stylesheet)`
- `element()`
- `element(person)`
- `element(?, personType)`
- `element(person, personType)`
- `attribute()`
- `attribute(id)`
- `attribute(*, xs:integer)`
- `attribute(id, xs:integer)`

**Evaluation order**

- From left to right
- Step by step (predicate applied to the last step)
  - `//department/person[1]`
  - `//department/person[1]`
- Predicate by predicate
  - `//person[@manages and position() = 5]`
  - `//person[@manages][position() = 5]`
XQuery, XML and databases

Patryk Czarnik
XML and Applications 2014/2015

XML support in databases - categorisation

- Classic (usually relational) database with XML support
  - logical structure - relations and references
  - additional XML-related features
  - used for application integration or storing XML data as part of larger data structures
- 'Native' XML database
  - logical structure - collection of XML document trees
  - XQuery (or XPath) as native query language
  - natural XML-related features
  - used for storing XML data (or structural data easily mapped to XML tree)

XML support in relational databases

- Possible functionalities
  - import and export of data in XML format
  - special treatment of XML data stored in fields
  - XML validation as part of integrity constraints checking
  - XPath or XQuery for querying field contents
  - XSLT applied to query result
- How to store XML data?
  - whole document (fragment) stored in single field
  - split into prima factors
    - each XML node in separate field
    - tables structure reflects tree structure of XML

XML in Oracle DB - XMLType

- XMLType - special datatype:
  - to be stored as LOB or used for columns, variables, etc.
  - indexing XML content
  - XPath expressions
  - validation against XML Schema
  - XSLT
- Available functions:
  - extract, extractValue, existsNode, transform, updateXML, XMLSequence

XML support in database engines

  - new data type XML:
    - only well-formed XML documents allowed
    - parsing and serialisation
    - implementation may add XML-specific operations
- Substantial support
  - IBM DB2 (since v.9 – pureXML)
  - Oracle (since 8i)
  - Microsoft SQL Server (since v.2000)
  - Sybase ASE (since v.12)
- Minimal support
  - MySQL - XPath queries over text fields containing XML
  - PostgreSQL – as above plus XML datatype but with no special operations

Native XML database

- Logical layer
  - XML document as basic data entity
  - collections of documents build a database
  - XML schema (or equivalent) as structure definition
  - XQuery (or XPath) as “native” query language
- Physical layer - not necessarily “files with XML text”
  - More than just a collection XML files:
    - transactions and concurrent access
    - security (access privileges etc.), versioning, replication, ...
    - API for data access and update
    - additional means of data access
      - e.g. REST-compliant HTTP server
    - indexing for efficient access to selected nodes

Example – XML support in Oracle database

http://www.oracle.com/xml
- Since Oracle 8i
  - details differ from version to version
- XML parsers
  - for database programming (PL/SQL)
  - or middleware programming (Java, C++)
- XML-SQL Utility
  - XML data import and export
- XMLType data type and XML-specific operations

XMLType applications – some examples

```
CREATE TABLE warehouses(
    warehouse_id NUMBER(4),
    warehouse_spec XMLTYPE,
    warehouse_name VARCHAR2(35),
    location_id NUMBER(4));

CREATE TABLE po_xtab of XMLType;

UPDATE po_xtab
SET po_doc = UPDATEXML(po_doc,
"/PO/CUSTNAME/text()", 'John');

INSERT INTO warehouses VALUES
(100, XMLType:
    "<Warehouse wNo="100">
    <Building><Name>Building</Building>
    </Warehouse>", 'Tower Records', 1003);

SELECT e.po_doc.getTextval() AS poXML
FROM po_xtab e
WHERE e.po_doc.existsNode('/PO/CUSTNAME = "po_2"') = 1;

CREATE INDEX city_index ON po_xtab
(po_xtab.extract("/PO/NAME/text()").getNumberVal());
```

Source: Oracle DB documentation
Standards for XML databases

- High level query languages:
  - XQuery – primary language for queries
  - versions 1.0 and 3.0 in use
  - XQL – former approach to make XML query language
  - XPath – poor stub for XQuery
- High level update languages:
  - XQuery Update Extension
  - XUpdate
- Programmer APIs
  - depend additionally on programming language
  - XML Database API (XIPI)
  - XQJ (for Java, expected to become XML equivalent of JDBC)
  - vendor-specific APIs...

XQuery – the query language for XML

- Status
  - XQuery 1.0 – W3C Recommendation, 1 2007
  - XQuery 3.0 – W3C Recommendation, IV 2014
  - Data model, functions and ops – shared with XPath 2.0 / 3.0
  - Syntax defined in a separate document
    - In practice: extension of XPath
- Main features:
  - Picking up data from XML documents
  - Constructing new result nodes
  - Sorting, grouping
  - Defining custom functions
  - Various output methods (XML, HTML, XHTML, text)
  - shared with XSLT

XQuery – query structure

- (Unexpectedly) XQuery is not an XML application
  - There exists a verbose XML syntax for XQuery, not intended to be written by hand
- Typical file extensions: .xquery, .xq, .xqm (for modules)
- Text format, header and body
  - header optional in normal queries
  - units declared as modules do not have body

XQuery headers

- Header part consists of declarations:
  - version declaration
  - import
  - flags and options
  - namespace declaration
  - global variable or query parameter
  - function

Example

```
xquery version '1.0' encoding 'utf-8';
declare namespace foo = 'http://example.org';
declare variable $id as xs:string external;
declare variable $doc := doc('example.xml');
$doc/foo/object[@id = $id]
```

Node constructors – computed

- The same example again to show the syntax

```
for $el in doc('example.xml')/" return
   element p {
     attribute style ('color: blue'),
     text ('I have found an element.'),
    } processing-instruction pi ('bla bla')
   comment ('Comments and PIs also taken to result')
}
```

Application example – dynamically computed name

```
for $el in doc('example.xml')/" return
   element (concat('elem-', name($el))
    attribute depth (count($el/ancestor::node())),
    text (name($el))
  )
</result>
```

Custom function definitions

- Simple example:

```
declare function local::factorial($n) {
  $n * local::factorial($n - 1)
};
```

Example using type declarations:

```
declare function local::factorial($n as xs:integer)
  as xs:integer {
    $n * local::factorial($n - 1)
};
```

Type constraints

- Type declarations possible (but not obligatory) for:
  - variables
  - function arguments and result
  - also in XSLT 2.0 (variables and parameters)
- Dynamic typing used in practical applications
  - 13 if (aCondition) then 'not a valid number' else 1 may fail or not depending on input data
  - some $x in (1+1, xs:math('long long time ago')) satisfies $x=2 fails or not depending on the processor
- Static typing discussed, but rarely deployed
  - "academic" solutions, for XQuery rather than XSLT
Type declarations

- Capabilities:
  - type name
  - built-in - always available
  - user-defined - schema aware processors only
  - kind of node | node() | item() | occurrence modifier (?, *, +, exactly one occurrence by default).

- Examples:
  - xs:double
  - element()
  - node()*
  - xs:integer?
  - item()+

XQuery 3.0 – selected improvements
(and XPath 3.0 to some extent)

- Minor features
  - string concatenation operator ||
    'Her name is ' || name
  - mapping operator ! (1, 2, 3) ! (4, 5) ~ (6, 7, 8)
  - switch
  - count clause in FLWOR

- More serious features
  - group-by clause and grouping "windows" in FLWOR
  - higher-level functions,
  - more predefined functions
  - in particular, more math available in separate namespace

XML database products – overview

<table>
<thead>
<tr>
<th>Product</th>
<th>Licence</th>
<th>Queries</th>
<th>XML:DB API</th>
</tr>
</thead>
<tbody>
<tr>
<td>eXist</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>BaseX</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>MarkLogic</td>
<td>commercial</td>
<td>XPath</td>
<td>yes</td>
</tr>
<tr>
<td>Apache Xindice</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>Sedna</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>Gemire Enterprise</td>
<td>commercial</td>
<td>XQuery, OQL</td>
<td>yes</td>
</tr>
<tr>
<td>Tamino</td>
<td>commercial</td>
<td>XQuery, XPath</td>
<td>part</td>
</tr>
</tbody>
</table>

Source: Wikipedia and providers' websites

In addition:
- Saxon – just a query processing engine, works on files (or other XML sources accessible in Java).

eXist – eXide

- XQuery programmer SDK running within a browser
- supports also (to some extent...)
  - XSLT, XML Schema, XHTML, XForms

eXist – template mechanism

- Easy integration of XQuery logic and HTML interface

XML:DB

- Initiative for XML database interfaces specification
- XML Database API (XAPI)
- accessing XML databases from programs
- resource collections (resource = XML document)
- reading and writing documents via DOM or SAX
- pluggable "services"; specified: XPath, transactions, operations on collections
- last version: 2001

- XML Update Language (XUpdate)
- XML application (format) for updating XML databases
- inserting, updating and removing nodes
- XPath used for node addressing
- last version: 2000

eXist DB

- One the most popular and elaborated XML database engines
- Open-source, but developed and supported by a (German) company; commercial support available

- Features include:
  - storage of XML and binary entities
  - various means of access, including: human-readable Web interface, direct HTTP access (REST-compliant), SOAP and XML-RPC, Java API (XQ), elements of XAPI
  - full XML model available in XPath, XQuery, and XSLT code
  - full XQuery support with majority of new 3.0 features
  - Update extension and some other non-standard extensions
  - XForms support using betterFORM or XSLTForms plugins
  - extensible with custom Java code

XUpdate – example

- Example (from XUpdate documentation)

```xml
<xupdate:modifications version="1.0">
  <xupdate:insert-after select="/addresses/address[1]">
    <xupdate:element name="address">
      <xupdate:attribute name="id">2</xupdate:attribute>
      <fullname>Lars Martin</fullname>
      <email>lars.martin@xyz.com</email>
    </xupdate:element>
  </xupdate:insert-after>
</xupdate:modifications>
```
XSLT

Patryk Czarnik
XML and Applications 2014/2015

XSLT - availability of tools

- XSLT 1.0 processors:
  - Internet browsers (IE, Mozilla/Firefox, Opera, Chrome)
  - Apache Xalan (for Java and C++)
  - xsltproc (Linux and related OSs)
  - XML extensions of database engines
- XSLT 2.0 processors:
  - Saxon (for Java and .NET)
    - basic version free (Open Source)
    - full (schema aware) version paid
  - commercial tools: XML Spy, oXygen
- Authoring tools:
  - Hundreds of plain text editors (with syntax highlighting etc.)
  - Advanced programmer environments (Eclipse, IntelliJ, ...)
  - Commercial XML-specialised tools (XML Spy, oXygen, ...)

XSLT - where does it come from?

- XSL - Extensible Stylesheet Language
  - Presentation of XML documents by transformation
- XSLT - XSL Transformations
  - Language (XML) to define transformations
  - Transformation of a source XML document tree to a result tree
  - Designed as one of XSL components
  - General enough to be used for other purposes

XSLT in run

- Processing based on tree nature of XML documents
  - Start: running template matching source document root (/)
    - such template exists even if not declared
  - Other templates run when applied with apply-templates instruction
    - and so on, recursively
    - usually following shape of source document
    - matching nodes to patterns and selecting appropriate templates
    - template result pasted into result tree
  - Other ways of flow control
    - call-template - invoking templates without pattern matching
    - for-each - iteration over sequences of nodes
    - if and choose - conditional processing

Pattern matching and template selection

- match attribute of template
  - pattern, restricted form of XPath expression
  - "What are the nodes this template applies to?"
- select attribute of apply-templates instruction
  - XPath expression
  - "Which nodes to process now?"
  - optional, children of current node selected if not given
- Matching
  - for each node to be processed, independently:
    - from all templates which patterns match the node
      - at least one always exists
    - select one with the highest priority
      - usually - one with the strictest pattern
    - in case of many templates with the same priority - conflict
      - error or latter template chosen, depending on implementation

Structure of XSLT stylesheets

- Stylesheet (arkusz) consists of templates
  - Template (szablon) - building block of XSLT
    - transformation of single source node to result tree fragment
    - may be called many times for different nodes
  - Within template:
    - text and elements out of XSLT namespace -> copied to result
    - XSLT instructions -> control flow, copying content from source, inserting computed content, and more features (in 2.0 even more)
    - XPath expressions in some instructions -> accessing source document, checking conditions, calculating numbers, etc.
  - XSLT can be seen as a programming language
    - with a bit non-standard syntax
    - convenient for processing and creating XML content

Structure of stylesheet - example (1)

```xml
<?xml version="1.0" encoding="utf-8"?>
<xsl:stylesheet version="1.0"
    xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="utf-8"/>
  <xsl:template match="/"/>
    <html>
      <head>
        <title>Kindergarten pupils</title>
      </head>
      <body>
        <h1>Kindergarten pupils</h1>
        <xsl:apply-templates/>
      </body>
    </html>
</xsl:stylesheet>
```

Structure of stylesheet - example (2)

```xml
<xsl:template match="kids">
  <ul>
    <xsl:apply-templates/>
  </ul>
</xsl:template>
```
### Sorting
- **sort** instruction ("declaration") in for-each, for-each-group and apply-templates
- Sorting options in attributes:
  - select - what values are compared - processed nodes by default
  - data-type, lang - data type (number | text) and natural language
  - order, case-order, stable

### Inserting content into template body
- Content from template body copied to result:
  - text nodes, except whitespace-only text nodes (indentation etc.)
  - elements, except elements in XSLT namespace
    - together with their attributes
    - elements content processed recursively
- Applies to any "sequence constructor" in general
  - content of block instructions (for-each, if, etc.)
  - function body (XSLT 2.0), parameter and variable body

### Shallow copying
- **copy** instruction
  - Creating node "the same as" the current one
    - same kind
    - same name and namespace (if applicable)
  - Content of current element not copied
  - Content of copy instruction processed as usual
    - may be used to copy / process source element content, if needed

### Grouping (XSLT 2.0)
- **for-each-group** instruction
- **current-group** and **current-grouping-key** functions

#### Example from Recommendation
```xml
<xsl:for-each-group select="cities/city" group-by="@country">
  <tr>
    <td><xsl:value-of select="position()"/></td>
    <td><xsl:value-of select="@country"/></td>
    <td><xsl:value-of select="current-group()/%name" separator=""," /></td>
  </tr>
</xsl:for-each-group>
```

### Constructor instructions
- Instructions creating result nodes:
  - element attribute text
  - comment processing-instruction

#### Typical applications of constructor instructions
- Inserting processing instructions and comments
- Inserting sole whitespace characters
- Inserting text fragments without unwanted whitespace characters
- Element or attribute name established at runtime
- Inserting attribute conditionally

### Inserting XPath expression result
- **copy-of** and **value-of** instructions
  - in XSLT 2.0 also **sequence**
  - XPath expression given in select attribute
    - Expression result inserted into transformation result, depending on instruction, as:
      - copy-of - deep copy of result
      - value-of - text node containing text representation of result
    - sequence - unprocessed sequence (matters only theoretically)

#### Examples
```xml
<xsl:copy-of select="//person[position()='chief']" />
<xsl:value-of select="//person[position()='chief']/surname" />
<xsl:value-of select="salary + sales * commission div 100" />
```

### Inserting XPath expression result
- **value-of** in XSLT 1.0 - details
  - If expression evaluates to simple value – that value printed
  - If expression evaluates to node set – only **first node** from set cast to string and printed
  - As slightly unintuitive – often leads to errors
value-of in XSLT 2.0 - details

- Expression evaluates to sequence (as always in XPath 2.0)
- Sequence flattened and atomized
- Result text:
  - string representations of all sequence items
  - separated with text provided in separator attribute
  - default separator - single space
- Incompatible with 1.0 behaviour

value-of etc. - best practices

- Instructions with select - all processing XPath result somehow...
- Which one to use - depending on what is processed
- Value-of
  - Value of simple type (number, string, date/time), in particular when computed by an expression or function
  - Value of single text node / simple-type element / attribute
  - Elements? - only when we want to flatten them to plain text
  - fully aware of uninitiative behaviour in XSLT 1.0
  - and incompatibilities between 1.0 and 2.0
  - Do not make your default manner of the last one, it is heavily overused in practice

Global variables and parameters

- Parameter value set from outside...
  - execution environment, command line, API call
- Variables computed once per transformation

local variables

- "Declarative" approach – variables cannot change their values
  - They may assume different values in subsequent invocations
  - Analogous to let in functional programming or final variables in java

Template parameters

- Declaration in template: param
- Value specification on invocation: with-param

Consequences of declarative approach

- name attribute of template
- call-template invokes
- Recursion permitted - high power of expression
Recursion “programming” in XSLT (even 1.0)

- Enable "programming" in XSLT (even 1.0)

```xml
<xsl:template match="factorial">  
  <xsl:param name="n"/>  
  <xsl:param name="res" select="1"/>  
  <xsl:choose>  
    <xsl:when test="$n > 1">  
      <xsl:call-template name="factorial">  
        <xsl:with-param name="m" select="$n - 1"/>  
        <xsl:with-param name="res" select="$n * $res"/>  
      </xsl:call-template>  
    </xsl:when>  
    <xsl:otherwise>  
      <xsl:value-of select="$res"/>  
    </xsl:otherwise>  
  </xsl:choose>  
</xsl:template>
```

Custom function recursive definitions (XSLT 2.0)

```xml
<xsl:template name="factorial">  
  <xsl:param name="n"/>  
  <xsl:param name="res" select="1"/>  
  <xsl:choose>  
    <xsl:when test="$n > 1">  
      <xsl:call-template name="factorial">  
        <xsl:with-param name="m" select="$n - 1"/>  
        <xsl:with-param name="res" select="$n * $res"/>  
      </xsl:call-template>  
    </xsl:when>  
    <xsl:otherwise>  
      <xsl:value-of select="$res"/>  
    </xsl:otherwise>  
  </xsl:choose>  
</xsl:template>
```

Serialisation of transformation result

- Transformation takes place at level of document tree
- Serialisation - converting result tree into sequence of bytes
  - Several formats ("methods") supported:
    - xml
      - ordinary XML format, “general parsed entity”
    - html
      - uses only starting tags for empty elements (\<br> rather than \<br/>)
      - automatically adds Content-Type metadata to HTML head tag
    - text
      - omits all tags and print only content of result text nodes
    - xhtml (XSLT 2.0)
      - much like xml, but allows transformers to control result structure

Temporary result tree fragments

- XSLT 1.0 - distinct types node set and result tree fragment
  - not allowed to mix
  - not allowed to process result tree fragment again
- XSLT 2.0 - no such restriction
  - In particular: result (e.g. of value-of or apply-templates) can be read and processed again
  - this is the real programming and full recursion!

```xml
<xsl:variable name="tmp">  
  <xsl:apply-templates select="document"/>  
</xsl:variable>  
<xsl:apply-templates select="step" mode="correct"/>
```

Some typical patterns

- Recommended structure of stylesheet depends on application
- Typical patterns and best practices for
  - text document presentation
  - data processing and presentation
  - copying and filtering

Text document presentation

- (this subject is going to be extended in the next lecture)
- Processing driven by source document structure (push)
- Many simple templates
  - one template for one type of input elements
  - translating source element into target element and processing its content with apply-templates
- More elaborated templates for elements requiring special treatment...
- Shape of result tree based on shape of source tree
- Works well with mixed content typical to text documents

Result of transformation – common applications

- XSL Formatting Objects:
  - according to original XSL approach
  - especially useful for printed publication
- HTML or XHTML
  - most popular
  - especially useful for Web publication
- Arbitrary XML, e.g.:
  - migration to new / other format
  - acquiring and processing data (alternative to XQuery)
  - generating XSL as result of XSL
- Plain text, e.g.:
  - CSV and other text data formats
  - scripts and configuration files
  - converting text documents to non-XML text formats

Method (xml | html | text | xhtml)

- encoding - character encoding
- indent (yes | no) - whether to automatically format output
- version - version of XML or HTML
- doctype - public, doctype - system - DOCTYPE declaration

```xml
<xsl:output method="xsl" version="1.0" encoding="utf-8" doctype-public=""" doctype-system="http://www.w3.org/Tr/xhtml1/DTD/xhtml1.dtd"/>
```

Begin of result document

- `<html version="1.0" encoding="utf-8">`
- `<!--DOCTYPE PUBLIC """">`
  - `http://www.w3.org/Tr/xhtml1/DTD/xhtml1.dtd"/>`

Text document presentation

- Root template
  - `<xsl:template match="/">`
    - `<html>`
      - `<head>`...
      - `<body>`
        - `<xsl:apply-templates />`
        - `</body>`
    - `</html>`
  - `<xsl:template>`

- Typical simple template
  - `<xsl:template match="item">`
    - `<li>`
      - `<xsl:apply-templates />`
      - `</li>`
    - `<xsl:template>`

- More elaborated template
  - `<xsl:template match="person">`
    - `<div class="person">`
      - `<xsl:apply-templates select="fname"/>`
      - `<xsl:template>`
    - `<xsl:template>`
Data processing and presentation

- Processing driven by intended result structure (pull)
- One (or a few) large template
  - fixes structure of result
  - inserts acquired / computed data into intended places using value-of
  - iterates over monomorphic sequences using for-each
- Complex XPath expressions
  - or even "XSLT programming" to compute more complex things

Copying and filtering - big step approach

Coarse-grained filtering template

```xml
<xsl:template match="/company">
  <div class="report">
    <p>Number of employees: <xsl:value-of select="count(.//person)" /></p>
    <p>Personal expenditures annually:
      <xsl:value-of select="sum(person/salary)" /></p>
    <xsl:for-each select="department">
      <dt><xsl:value-of select="name" /></dt>
      <dd><xsl:value-of select="12*sum(person/salary)" /></dd>
    </xsl:for-each>
  </div>
</xsl:template>
```

One big template

```xml
<xsl:template match="/company">
  <div class="report">
    <p>Number of employees: <xsl:value-of select="count(.//person)" /></p>
    <xsl:apply-templates select="department" />
  </div>
</xsl:template>
```

Data processing and presentation

Copying and filtering - small step approach

Standard copy template

```xml
<xsl:template match="node()[@xsl:ignore]=false">
  <xsl:copy>
    <xsl:apply-templates select="*" />
  </xsl:copy>
</xsl:template>
```

Simple template changing element name

```xml
<xsl:template match="collection/item">
  <xsl:apply-templates select="@*" />
  <xsl:attribute name="name">changed</xsl:attribute>
</xsl:template>
```

"Advanced" template

```xml
<xsl:template match="para">
  <p>Paragraph of size <xsl:value-of select="string-length(string())" /></p>
</xsl:template>
```

Copying and filtering

- Returning subset of source document

Filtering and copying

Two styles, depending on needs:

- Big step style (analogous to target-driven processing)
  - large fragments of input copied as whole
  - copy-of within big template

- Small step style (analogous to source-driven processing)
  - recursive processing with exclusion or specialised transformation of selected nodes
  - copy as default and specialized templates for elements to be processed
Presentation of XML

Patryk Czarnik
XML and Applications 2014/2015
Lecture 11 - 12.01.2015

Benefits of content and formatting separation

- With semantic tagging - source data analysis easier and more reliable (than reverse-engineering of formatted text)
- Ability to easily present:
  - the same document after modifications
  - other documents from the same class
- Changes in formatting applied easily:
  - modifications in one place - the stylesheet
  - whole class of documents formatted consistently
- Alternative styles for the same class of documents, depending on:
  - media type (screen, printout, voice)
  - details level
  - reader preferences (or disabilities...)

Cascading Style Sheets – history

- Roots of stylesheet idea – 1970s:
  - translation of markup documents to (different) printer languages
- Beginning of CSS: 1994
- CSS Level 2: May 1998
- CSS 2.1: June 2011
- restricts CSS 2 and makes it more precise
- CSS Level 3: split into modules, some of them are final recommendations, some are not

Separation of content and formatting

- According to best XML practices:
  - Documents consist of content / data.
  - Tags are for structure and meaning (semantic tagging).
  - e.g. <count value="2">95</count> rather than <count 2.95/>
  - There is no direct formatting information.
- How to present documents:
  - Generic (and poor) XML presentation methods
  - XML source
  - document tree
  - unformatted text content
  - Custom application handling a particular known class of documents
  - Importing XML to text editors or DTP tools
  - External style sheets

Standards related to XML presentation

- Assigning style to document:
  - Associating Style Sheets with XML documents
- Stylesheet languages:
  - DSSSL (historical, used for SGML)
  - Document Style Semantics and Specification Language
  - CSS
  - Cascading Style Sheets
  - XSL
  - Extensible Stylesheet Language

Applications of CSS

- First and major one: style for Web sites
- Separation of content and style for HTML
- (Simple) stylesheet for XML
- CSS 2 and the idea of “accessibility”:
  - support for different media
  - support for alternative presentation means (e.g. voice generation)
  - enabling reader to override style proposed by author (reader rules)

Idea of stylesheet

- Using xml - stylesheet processing instruction
- Defined in W3C recommendation
  - Associating Style Sheets with XML documents

Example stylesheet (fragment)

```xml
<company>
  <name>Extremely professional staff</name>
  <department id="acc">
    <name>Accountancy</name>
    <person position="expert" id="102013">
      <fn name="Dawid" surName="Fiszewicz"/></fn>
      <phone type="office">+4823213293</phone>
      <phone type="mobile">+4855102585</phone>
      <email>office@example.com</email>
    </person>
    <person position="chief" id="102104">
      <fn name="Monika" surName="Bochnakowicz"/></fn>
      <phone type="office">+4823221294</phone>
      <email>edoffice@example.com</email>
    </person>
  </department>
  <main-office>...<main-office>
</company>
```
Rich visual formatting features
- Selecting elements by
  - name
  - location in document tree
  - attribute existence
  - attribute values
- Good support
  - internet browsers
  - authoring tools
- Easy to write simple stylesheets ;)

Only visualisation, not translation to different formats
- Selectors relatively weak. Conditions not expressible in CSS:
  - checking content of element, e.g.:
    - element A that contains element B
    - element A that contains text abc
  - logical composition of many conditions
    (available to some extent, but inconvenient)
  - value comparison (e.g. show negative amounts in red)
- Structure of blocks directly based on structure of source elts
  - reordering of elements hard (and not possible in general way)
  - not possible to show one element several times on page
- No data processing. Not available for example:
  - number calculations (summing etc.)
  - operations on text (shortening, regexp matching, etc.)

Extremely professional staff

Accountancy

Dawid Paczkiewicz
id: 48b2321323
mab: 48b5150150
paczkiewicz@example.com

Monika Dmochowska
id: 48b2313209
mob: 48b5102513
mdm@example.com

CSS selectors (representative examples)
- surname - element of the given name
- fnames, surname - both elements
- company name - name being descendant of company
- company > name - name being direct child of company
- surname + phone - phone being first child of its parent
- phone: first-child - phone being first child of its parent
- person|position| - person with position attribute
- person|position|managers - person with position attribute equal to manager
- person [roles=|managers|] - person with attribute role containing word manager (attribute as space-separated list)
- ol|staff| - equivalent to ol|class=|staff| (HTML only)
- person|k12 - person with ID (in DTD meaning) equal to k12

Transformation to HTML - example (1)

Exx:stylesheet version="1.0"
xmns:xsl="http://www.w3.org/1999/XSL/Transform"
xsl:output method="html" encoding="utf-8" /
xsl:template match="/"
  <div>
    <h1>
      Employees of <xsl:value-of select="/company/"/>
    </h1>
    <style type="text/css">
      body { background-color: #FFDF0D; }
      div, person { margin: 10px auto 10px 10px; }
    </style>
    <div>
      <h1>
      </h1>
    </body>
  </div>
</xsl:template>

Presentation by transformation

XML ("pure data")

Definition of transformation (in role of stylesheet)

Document for direct presentation

Transformation to HTML - example (2)

<xsl:template match="person">
  <xsl:variable name="mgr">
    <xsl:if test="position()='manager'">manager</xsl:if>
  </xsl:variable>
  <div class=""manager">
    <xsl:variable name="name"/>
    <xsl:apply-template select="name"/>
  </div>
  <div class=""phone">
    <xsl:apply-template select="phone"/>
  </div>
  <div class=""email">
    <xsl:apply-template select="email"/>
  </div>
</xsl:template>
Transformation to HTML – example (3)

Original idea of XSL

Transformation to XSL-FO – example (3)

Resulting HTML code (fragments)

Transformation to XSL-FO – example (1)

Resulting XSL-FO code (fragments)

HTML – resulting formatting

Transformation to XSL-FO – example (2)

XSL-FO – resulting formatting
XSL-FO – basic facts

- Presentation-oriented XML application
- Elements for different kinds of visual objects (block, inline, table, and so on)
- Attributes for formatting, based on CSS properties
- Especially useful for printed publications
- Focused on paged media type:
  - master pages (templates), page areas (header, footer, etc.)
  - automatic text flow and repeated (“static”) content
- Practice: intermediate format in XML = XSL-FO → PDF transformation
- Not supported by web browsers
- Designed as part of XSL framework
- result of XSL transformation
- not intended to be used standalone

Basic structure of XSL-FO document

```
<root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <layout-master-set>
    <fo:simple-page-master master-name="my-page">
      <fo:region-body/>
    </fo:simple-page-master>
  </layout-master-set>

  <fo:page-sequence master-reference="my-page">
    <fo:flow name="xsl-region-body">
      <block>Hello World</block>
    </fo:flow>
  </fo:page-sequence>
</root>
```

Formatting objects

XSL-FO elements relate to resulting formatting objects.

- Block level
  - block
  - list-block, list-item, list-item-label
  - table, table-row, table-cell,...
- Inline level
  - inline, character
  - external-graphics
- Special features
  - basic-link, bookmark, marker
  - footnote
  - page-number

Formatting properties

Most of XSL-FO attributes relate to style properties analogue to CSS properties.

- margin, padding, border-style
- background-color, background-image
- font-family, font-weight, font-style, font-size
- text-align, text-align-last, text-indent, start-indent, end-indent, wrap-option,
- break-before
- and much more (almost 300 properties in XSL 1.1)

“Page master” – page template

- Single page layout
- A document may be split in many such pages
- One body may be split in many such pages
- Four predefined (but optional to use) edge regions

```
<fo:simple-page-master master-name="A4">
  <fo:page-sequence master-reference="A4">
    <fo:flow name="xsl-region-body"/>
  </fo:page-sequence>
</fo:simple-page-master>
```

Distributing content to pages

- page-sequence – results in a number of pages
- flow – content split into pages
- static-content – content repeated on all pages
- flow-name – page region reference

```
<fo:page-sequence master-reference="A4">
  <fo:flow name="xsl-region-body"/>
</fo:page-sequence>
```

Page sequence master

- Using different page layouts within one page-sequence
- Simple page masters referred to be used in order (repetitions available)

```
<fo:layout-master-set>
  <fo:simple-page-master master-name="first">
    <fo:flows>
      <fo:flow name="xsl-region-body"/>
    </fo:flows>
  </fo:simple-page-master>
</fo:layout-master-set>
```
Tools mentioned today

- Adobe FrameMaker
  - especially useful for large and complex text documents
  - advanced support for XML and structured documents
  - constructs analogous to DTD and stylesheets
- Adobe InDesign
  - especially useful for documents that have to look perfectly
  - basic support for XML be means of filling a template with XML content

Adobe FrameMaker

- Word processor / desktop publishing tool
  - One of first that advanced tools
  - Acquired by Adobe in 1995
  - Especially popular for:
    - complex documents, where structure important
    - large documents, e.g. technical documentation
- Two kinds of documents (and 2 ways of authoring):
  - unstructured - flat, paragraph-based structure, similar to styles in popular word processors
  - structured - tree-like structure, based on SGML and XML

FrameMaker augments a structural approach to the content with a WYSIWYG editor convenience.

Structured application

- EDD - Element Definition Document (or Elements Catalogue)
  - document structure definition (elements, attributes)
  - formatting and other rules
- structured template - FM document
  - pagination, layout, header and footer, ...
  - styles ("paragraph/character format tags"), variables, markers, cross-reference formats, ...
- Elements Catalogue imported from EDD

Optional components:

- DTD – may also be generated from EDD
- Read/write rules – extra translations between XML and FM
- XSLT pre- and post-processing
- API client – custom executable application

Structured application dependencies

Content model (General rule)

- Expression built from element names, <TEXT> token, parentheses, and:
  - + – sequence of subelements
  - & – subelements in any order
  - | – choice
  - occurrence indicators (after element name or | group):
    - ? – optional element (0-1 occurrence)
    - * – any number of occurrence (0-unbounded)
    - + – at least one occurrence (1-unbounded)
    - no indicator - exactly one occurrence

Examples:

- imiq+, nazwisko
- Title, Abstract?, Section*

Structured documents in FM

- FM concept analogous to XML application in XML world
- FM manages a set of registered structured applications
- XML documents opened / saved directly:
  - Template and formatting rules from EDD define the formatting.
  - When an XML file is opened, the XML content fills the template and formatting rules define the style applied to the content...
  - Manual formatting available in FM, but lost when document saved as XML.

Element Definition Document

- FM document defining other documents structure
- EDD role corresponding to (in general XML applications):
  - DTD or XML Schema – structure definition
  - CSS or XSL (to some extent) - formatting rules
- Structure definition
  - available elements, their type and acceptable content
  - attributes, their type and optionality
  - Particular elements marked as FM special objects (tables and table components, variables, markers, cross-references, ...)
- Rules for elements:
  - formatting
  - initial value or structure
  - prefix and suffix

Kinds of elements (examples)

- Container
  - element with no special meaning
  - may contain elements or text (or both → mixed model)
- CrossRefReference – FM cross-reference
- Footnote – FM footnote
- Equation, Graphic – anchored objects;
  - XML would contain references to external entities
- Marker – FM marker
- SystemVariable – FM system variable reference
- Table, TableBody, TableHeading, TableRow, TableCell
  - table components
EDD and DTD – similarities

- Document structure definition
- Container elements
- Content model specification ([], *, ? +)
- Optional and required attributes
- Unique ID, ID Reference – ID, IDREF in DTD

EDD and DTD – differences

**EDD**
- FrameMaker-special element kinds (tables, variables, etc.)
- Numeric attribute types
- Multi-value attributes
- & - elements in any order
- Formatting rules
- No means for structure modularisation
  - style modularisation available through format change lists

**DTD**
- General-purpose elements (like EDD Container)
- No numeric types (for XML)
- Space-separated NMTOKENS and IDREFS
- Only choice and sequence
- No formatting specification
- Parameter entities as means for DTD modularisation

Formatting rules

- Appearance of particular elements described in EDD
- In element definition (e.g. Container) rules grouped by scope of effect:
  - TextFormatRules - formatting of whole element, inherited by descendants
  - FirstParagraphRules, LastParagraphRules - formatting of first / last paragraph only
  - PrefixRules, SuffixRules - content generated in front / at end of element and its formatting
- Some more features analogous to CSS selectors:
  - context rules
  - level rules
- We omit the rest of details here...

Element (Container): Mead

General rule: <TEXT>
Text format rules
1. In all contexts
   - Default font properties
   - Size: 16
   - Numbering properties
     - Autonumber format: <n>.

Adobe InDesign

- Advanced desktop publishing tool
  - part of Adobe Creative Suite
- Especially helpful when:
  - text and graphics mixed together
  - advanced, non-standard page layouts used
  - precise positioning and typesetting required
  - high-quality printout planned
- Compromise between word processor and graphic design tool
- Less structure support when compared to Adobe FrameMaker or XML-based solutions
- harder to automatize publication process
- easier to make ad-hoc formatting enhancements

Context rules – example

Text format rules
1. If context is: List [Type = “Bulleted”]
   - Numbering properties
     - Autonumber format: \#lt
     - Character format: bulletsymbol
   Else, if context is: List [Type = “Numbered”]
   1.1 If context is: (first)
      - Numbering properties
        - Autonumber format: n=1\lt
   Else
      - Numbering properties
        - Autonumber format: n=\lt

EDD and XML Schema

**EDD**
- FM-tied (special element kinds, formatting)
- No constraints for simple values, except lists of choice for attributes
- General ID/IDREF mechanism
- No means for structure modularisation
- Format specific for FM

**XML Schema**
- General-purpose technology (like DTD)
- Simple types and precise control of simple values (text, numbers, etc.)
- Advanced key/keyref mechanism
- Modularisation though types, type inheritance, groups
- Understandable and usable outside FM world (e.g. for WebServices)

XML in InDesign

- Parts of document may be annotated with XML-like structure
- Exporting tagged content as XML
- Importing XML, and then:
  - manually distributing XML fragments among text frames
  - automatic layout of imported content if placeholders were prepared
- Relating InDesign styles and XML tags:
  - applying styles to tags automatically
  - tagging content based on styles
  - XSLT pre- and post-processing
Manual formatting and style application

- XML content formatted as any text content
  - manual formatting
  - formatting with styles
- Recommended for:
  - one-time project (formatting not intended to be used again)
  - short text
  - non-repeating XML elements or formatting unrelated to XML structure

Mapping tags to styles – example

Sample document before and after applying styles

<table>
<thead>
<tr>
<th>Alice Blonde</th>
</tr>
</thead>
<tbody>
<tr>
<td>United Kingdom</td>
</tr>
<tr>
<td>Cardiff</td>
</tr>
<tr>
<td>Code 5813</td>
</tr>
<tr>
<td>33135888</td>
</tr>
<tr>
<td><a href="mailto:alice@example.com">alice@example.com</a></td>
</tr>
</tbody>
</table>

Import XML – two main approaches

- Content-first approach:
  - import content
  - then take care of it:
    - distribute content to text frames
    - format (by e.g. mapping tags to styles)
- Placeholders approach:
  - prepare document with stub content distributed and formatted as desired
  - then import (merge) XML and get it distributed and formatted automatically

Mapping tags to styles

- Automatic application of styles to elements
  - paragraph, character, table, and table cell styles applicable
  - Styles reapplied in all tagged stories of document
  - Elements with the same name receive the same chosen style
  - Styles have to be already defined

Mapping tags to styles – remarks

- Benefits:
  - fast formatting of large documents
  - consistent formatting
  - easy style enhancements in future
- Difficulties and discomforts:
  - styles have to be defined manually before mapping
  - special characters – paragraph breaks, spaces etc. – have to exist in structured content before formatting
  - one paragraph style used for many XML elements in case that those elements reside in the same source line
  - unneeded indents and line breaks from XML cannot be eliminated easily

Importing XML – content first approach

- Benefits:
  - real document visible while preparing layout and formatting
  - fast final result
- Disadvantages:
  - manual work to do each time document is imported
- Reasonable usage:
  - one-time process
  - e.g. importing structured content into larger, unstructured document, produced on special demand...
  - preliminary step while preparing template in placeholder approach

Style attributes – more flexible, but less convenient approach

```xml
<card
 xmlns:aid="http://ms.adobe.com/AdobeIndesign/4.0/"
 >
 <person aid:style="Person">
   <first-name>Alice</first-name>
   <surname aid:csstyle="5Name">Blonde</surname>
 </person>
 ...
</card>
```

Importing XML – placeholders approach

- Benefits:
  - cheap application to arbitrary many documents
  - repeatable, predictable results
- Disadvantages:
  - more preliminary work required
- Reasonable usage:
  - repeatable tasks
  - part of (semi-)automatised publication process when (part of) data comes from external XML
  - InDesign document as (very advanced...) stylesheets
  - saved as template allows for easy fresh documents creation
XPointer – xpointer scheme

- xpointer scheme allows to address elements using XPath:
- xmlns scheme adds namespace declarations to the above:
  - ustawa.xml#xmlns(pr=http://www.sejm.gov.pl/prawo)
  - xpointer(pr:art[5]/pr:par[2])

XInclude – example

```xml
<recipe>
  <xinclude xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
            xsi:schemaLocation="salad.xmp#xpointer"
            xmlns="http://www.w3.org/2001/XMLSchema-instance">
    <xi:include href="/salad.xml#xpointer(recipe/title)"/>
    <xi:fallback>No such recipe.</xi:fallback>
  </xinclude>
</recipe>
```

Standards for inter-document relations

- XPointer – addressing documents and their fragments
- XInclude – logical inclusion of documents within other documents
- XLink – declarative relations between documents and their fragments

XLink

- HTML links (<a>, <img>):
  - link two documents: link source and target
  - link source is always in the linking element
- XLink — an extended idea of linking:
  - link information represented in any element:
    - element name is not important
    - attributes coming from XLink namespace are
    - more than two ends of link (hyperlink → relation)
    - possibility to represent link outside linked resources
- Status: historical roots: HyTime,
  - XLink 1.0 – W3C recommendation: 2001,
  - XLink 1.1 – current version (made official TR: May 2010).

XInclude

- Including external XML documents (or their fragments) in another XML document.
- Similar to entities, but:
  - normal element markup, no special syntax,
  - no need to declare anything in DTD, nor to have DTD at all
- Main capabilities:
  - including complete documents (identified by URL) or their fragments (pointed by XPointer)
  - including XML tree (default) or raw text
  - defining content to be used in case of an error
- Supported by many parsers, including Java (JAXP).

XPointer

- The standard defines addressing XML documents and their fragments using standard URI syntax:
- 3 W3C recommendations dated 2002-2003:
  - XPointer Framework
    - http://www.w3.org/2001/Xlocator/
  - XPointer element() Scheme
    - http://www.w3.org/2001/Xlocator/
  - XPointer xmlns() Scheme
    - http://www.w3.org/2001/Xlocator/
  - (neverending?) Working Draft

Terminology

- Resource – any addressable unit of information or a service (file, program, query result).
- Link – a relation between participating resources, expressed explicitly with a linking element.
- Arc – information about traversal between labelled resources (in defined direction):
  - outbound arc – from a local resource to some external resource
  - inbound arc – from an external resource to some local resource
  - third party – between two external resources
- Note: a resource is regarded as remote when addressed by URI (even though it resides in the same document or linking element as the link which uses it).
Types of links

- **Simple link:**
  - is outbound
  - binds exactly two resources: a local one with an external one
  - contains exactly one arc between resources

- **Extended link:**
  - binds arbitrary number of local and external resources,
  - uses arcs to define methods of traversal between resources,
  - defines roles of participating resources,
  - defines roles of arcs.

Attributes in extended links

- **type** - role of the element in a link
  - simple | extended | locator | arc | resource | title | none

- **href** - URI of the external resource

- **role** - abstract identifier of the resource role (URI)

- **arcrole** - as above, but for an arc

- **title** - text label of the resource or arc

- **show** - presentation info: new | replace | embed | other | none

- **actuate** - activation info: onLoad | onRequest | other | none

- **label** - label used as identifier in from and to, not necessarily unique

- **from, to** - pointer (in an arc) for a certain resource label

XForms – document structure

- Forms are embedded in a host document, usually XHTML

- **Data model**
  - **xf:model** element
    - anywhere in host document
    - header more elegant
    - but body more practical for dynamic documents
    - more than one model available; in such case they must have identifiers

- **Form controls (in XForms namespace)**
  - placed within normal XHTML tags
  - (some of them) may contain further XHTML fragments

- **Action specifications and constraints tied with XForm elements**
  - by inserting them inside model fragments or control tags
  - using general *xf:bind* elements

Simple example

```xml
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:xm="http://www.w3.org/2002/xm"="">
  <head>
    <xm:model>
      <xm:instance>
        <xm:person sex="M">
          <xm:first-name>Michael</xm:first-name>
          <xm:last-name>Johnson</xm:last-name>
          <xm:birth-date>1965-06-23</xm:birth-date>
        </xm:person>
        <xm:person sex="F">
          <xm:first-name>Martha</xm:first-name>
          <xm:last-name>White</xm:last-name>
        </xm:person>
        <xm:person sex="M">
          <xm:first-name>Joseph</xm:first-name>
          <xm:last-name>Green</xm:last-name>
          <xm:birth-date>1970-01-01</xm:birth-date>
        </xm:person>
        <xm:person sex="M">
          <xm:first-name>James</xm:first-name>
          <xm:last-name>Wilson</xm:last-name>
          <xm:birth-date>1975-03-15</xm:birth-date>
        </xm:person>
      </xm:instance>
    </xm:model>
  </head>
</html>
```

Future of XLink

- **Applications:**
  - organization and association of resources even when no writing permission is granted
  - a new type of added value – link sets

- **Scope:**
  - local – link servers, link databases
  - Internet?

- **Problems:**
  - visualization of extended links
  - synchronization of links and resources (Internet)

Extended link – an example

```xml
<family xlink:type="extended" xmlns:link="http://www.w3.org/1999/xlink">
  <link xlink:parent xlink:href="http://example.com/loc1">
    <link xlink:title="Joseph"/>
  </link>
  <link xlink:parent xlink:href="http://example.com/loc2">
    <link xlink:title="Katherine"/>
  </link>
  <link xlink:parent xlink:href="http://example.com/loc3">
    <link xlink:title="Jane"/>
  </link>
  <link xlink:arc xlink:from="parent" xlink:to="child"/>
</family>
```

XForms

- **XML application for specification of interactive forms**

- **Versions:**
  - 1.0 – 2003
  - 1.1 – 2009 (currently most commonly used)
  - 2.0 – WD

- **More than HTML forms:**
  - data model defined separately from UI
  - by example or using XML Schema
  - processing model specified with events and actions
  - various data access modes given in submission module
  - including REST-compliant HTTP access
  - more UI controls, interactive switch, automatic repeat

Simple example

```xml
<body>
  <div xhtml:ref="first-name">First Name</div>
  <xm:input xhtml:ref="first-label" xhtml:type="text" />

  ...<xm:select xhtml:ref="gender"
    xhtml:label="Select Gender"
    xhtml:options="<xm:option xhtml:value="M">Male</xm:option>
    <xm:option xhtml:value="F">Female</xm:option>
">Gender</xm:select>

  <xm:submit xhtml:ref="submit">
    Submit</xm:submit>
</body>
```

Simple example – an example
REST services – recall

- REST for Representational State Transfer
  - Principles:
    - Service = collection of resources
    - URL identifies a resource
    - Resource has a normalised representation and can be transferred through the network
      - XML for structural data
      - Binary and other structural formats (JSON) also permitted
    - HTTP methods directly used to manipulate resources
      - GET, PUT, DELETE – obvious semantics
      - Other HTTP methods, HTTP authentication, cookies, additional headers and arguments – all may be used to implement additional features

Some popular XML applications

- Multimedia
  - Scalable Vector Graphics (SVG)
  - Mathematical Markup Language (MathML)
- Security
  - XML Signature
  - XML Encryption

REST for XML database

- REST - remote access to a repository
  - Can it be an XML database? Why not...
  - Possible applications:
    - Access API independent of particular platform or pr.lang.
    - Easy and efficient remote access from
      - Javascript clients (AJAX)
      - Mobile clients
    - Integration with XML-related standards
      - XSLT, XQuery - documents available through HTTP URLs
      - XForms - acquiring and modifying documents directly form XForms
    - HTTP interface available also to call server-side XQuery scripts
    - XRX architecture: XForms + REST + XQuery

Some popular XML applications

- Documents / text processing / publications:
  - DocBook
  - Text Encoding Initiative (TEI)
  - Darwin Information Typing Architecture (DITA)
  - Open Document (ODF, OASIS standard, ISO/IEC 26300)
  - Office Open XML (OOXML / OpenXML) (Ecma standard, ISO/IEC 29500)
- Metadata and knowledge representation:
  - Dublin Core
  - RDF
  - Topic Maps
Text markup – fundamental distinction

**Presentational markup**
- Describes the appearance of text fragment
  - font, color, indentation,...
- Procedural or structural
- Examples:
  - Postscript, PDF, TeX
  - HTML tags: `<B>` `<BR>`
  - direct formatting in word processors
  - XSL-FO (we will learn)

**Semantic markup**
- Describes the meaning (role) of a fragment
- Examples:
  - LaTeX (partially)
  - HTML tags: `<STRONG>` `<Q>` `<CITE>` `<VAR>`
  - styles in word processors (if used in that way)
  - most of SGML and XML applications

Documents in information systems

- Since the introduction of computers to administration, companies and homes plenty of digital documents have been written (or generated).
- Serious problem: number of formats, incompatibility.
- De facto standards in some areas (e.g. .doc, .pdf, .tex)
  - most of them proprietary
  - many of them binary and hard to use
  - some of them undocumented and closed for usage without a particular tool

Road to XML

- Late 1960s - IBM - SCRIPT project, INTIME experiment
  - Charles Goldfarb, Edward Mosher, Raymond Lorie
  - Generalized Markup Language (GML)
  - ISO 8879:1986
- Late 1990s - Extensible Markup Language (XML)
  - W3C Recommendation 1998
  - Simplification(!) and subset of SGML

Why is XML a different approach?

- **Common base**
  - document model
  - syntax
  - technical support (parsers, libraries, supporting tools and standards)
- **Different applications**
  - varying set of tags
  - undetermined semantics
  - Base to define formats rather than one format
- **General and extensible!**

What is XML?

- **Standard** - Extensible Markup Language
  - World Wide Web Consortium (W3C) Recommendation
    - version 1.0 - 1998
    - version 1.1 - 2004
- **Language** - a format for writing structural documents in text files
- **Metalanguage** - an extensible and growing family of concrete languages (XHTML, SVG, etc...)
- **Means of:**
  - document markup
  - carrying data (for storage or transmission)

Text markup – roots

The term *markup* origins from hints in manuscript to be printed in press.

Po polsku *znakowanie tekstu*

And she went on planning to herself how she would manage it. ‘They must go by the carrier,’ she thought; ‘and how funny it’ll look! And how odd the directions will look! ALICE’S RIGHT FOOT, ESQ.

And now we have 1000+1 formats to handle...

Text markup – roots

In fact people have marked up text since the beginning of writing.
- Marking up things in hand-written text:
  - punctuation, indentation, spaces, underlines, capital letters.
- Structural documents:
  - layout of letter - implicit meaning, tables, enumeration, lists.
- Today informal markup used in computer-edited plain text:
  - email, forum, blog (FB etc.), SMS, chat, instant messaging.

And she went on planning to herself how she would manage it. ‘They must go by the carrier,’ she thought; ‘and how funny it’ll look, sending presents to one’s own feet! And how odd the directions will look! ALICE’S RIGHT FOOT, ESQ.

**Let’s design another format replacing all existing!**

And now we have 1000+1 formats to handle...
What is XML not?

- Programming language
- Extension of HTML
- Means of presentation
  - You should say “data represented in XML format” rather than “presented”
- Web-only, WebServices-only, database-only, nor any other *-only technology – XML is general.
- Golden hammer
  - XML is not a solution for everything

XML components

Main logical structure

- Element (element)
  - start tag (znacznik otwierający)
  - end tag (znacznik zamkujący)
- Attribute (attribut)
  - Text content
    - / text node (zawartość tekstowa / węzeł tekstowy)

Comments and PIs

- Comment (komentarz)
- Processing instruction (instrukcja przetwarzania, ew. instrukcja sterująca, dyrektywa)
  - target (cel, podmiot)

Where do entities come from?

5 predefined entities: \lt \gt \& \# amp apos quot
- Custom entities defined in DTD
  - simple (plain text) or complex (with XML elements)
  - internal or external
- We skip details of unparsed entities and notations.

Unicode and character encoding

- Unicode – big table assigning characters to numbers.
  - Some characters behave in a special way, e.g. U+0D2B . Ogonek
  - One-byte encodings (ISO-8859, DOS/Windows, etc.)
    - Usually map to Unicode, but not vice-versa
    - Mixing characters from different sets not possible
- Unicode Transformation Formats:
  - UTF-8 – variable-width encoding, one byte for characters 0-127 (consistent with ASCII), 16 bits for most of usable characters, up to 32 bits for the rest
  - UTF-16 – variable-width, although 16 bits used for most usable characters; big-endian or little-endian
  - UTF-32 – fixed-length even for codes > 0xFFFF

Document Type Definition

- Specifies the “type” of this XML document.
  - Not required and in fact not used in modern applications.
  - Can be written in a separate file, inside the XML document, or using a mixed approach.
  - Using a separate file gives some advantages and usually this is the choice.
- Apart from document structure definition, which we’ll learn in the next week, it allows to define entities and notations.

XML components – CDATA

- CDATA section (sekcja CDATA)
  - Whole content treated as a text node, without any processing.
  - Allows to quote whole XML documents (not containing further CDATA sections).

XML components – entity references

- Character reference decimally: \&#252;
  - (referencja do znaku)
- Character reference hexademically: \&#xFC;
  - Relate to character numbers in Unicode table.
  - Allow to insert any acceptable character even if out of current file encoding or hard to type from keyboard.
  - Not available within element names etc.
- Entity reference: \&lt; \&amp; \&apos; \&quot;
  - (referencja do encji)
    - Easy inserting of special characters.
    - Repeating or parametrised content.
    - Inserting content from external file or resource
      - addressable by URL.

Document prolog

```xml
<?xml version='1.0' encoding='iso-8859-2' standalone='no'?>
<!DOCTYPE article SYSTEM "article.dtd">
<article>
  ...
</article>
```

XML declaration

- Looks like a PI, but formally it is not.
- May be omitted. Default values of properties:
  - version = 1.0
  - encoding = UTF-8 or UTF-16 (deducted algorithmically)
  - standalone = no
- Document type declaration (DTD)
  - Optional

Example

The same text fragment written in 3 ways:
```xml
<option>0 &lt;&amp; \&amp; \&apos; \&quot;
</example>
```
### Document as a tree

```xml
<employee
  id="77"
  fname="Jan"
  surname="Kowalski"
  tel="123234345">
  <tel>605066665</tel>
  <intern>1313</intern>
</employee>
```

### Language or metalanguage?

- **XML** is a language.
  - Grammar, additional constraints expressed descriptively
  - One can determine whether a sequence of characters is
    well-formed XML.
  - Better to think as of a **metalanguage**.
    - Common base for defining particular languages
    - Set of languages (open, unlimited)
    - A particular language based on XML will be called an
      **XML application**.

### XML vs (X)HTML

- **HTML**
  - Defined set of elements and attributes
  - Their meaning established
  - Defined (to some extent) way of presentation
  - Although specification exists, tools accept (and often create) incorrect
    HTML.

- **XML**
  - All (syntactically correct) tag names allowed
  - Undefined semantics
  - `<p>` is not necessarily a paragraph!
  - Unspecified way of presentation
  - Processors obliged to work with well-formed XML only

### XML vs SGML

- **SGML**
  - “Convenient for author”
    - Some ambiguity allowed when supported by DTD,
    - e.g. in HTML `<p>` or `<li>` may stay not closed
  - More datatypes for attributes in DTD
  - More DTD structuralisation capabilities
  - DTD required

- **XML**
  - “Convenient for processor”
    - Strict unambiguous syntax
    - Less options, simpler DTD
    - Unified with modern internet standards (URI, Unicode)
    - DTD optional
**What can we do with XML?**

- Define new XML-based formats using XML Schema or other standards
- Validate documents against the definition
- Edit manually (e.g., Notepad) or using specialised tools
- Store in files or databases, transfer through network
- Process documents (read, use, modify or create, write) in custom applications
- Use existing parsers and libraries
- Search and query for data using XQuery, XPath, XSLT, or custom applications
- Transform to other formats (for presentation, but not only) using XSLT, XQuery, or custom applications
- Format using stylesheets or specialised tools

**Advantages of XML**

- Compared to binary formats:
  - Readable (to some extent...) for humans, "self-descriptive"
  - Possibility to read or edit using simplest tools
  - Easier debugging
- Compared to ad-hoc designed formats:
  - Common syntax and document model
  - Common way of defining XML applications (XML Schema)
  - Existing tools, libraries, and supporting standards
  - Interoperability
- Compared to WYSIWYG editors and their formats:
  - Semantic markup available, more advanced than flat styles
  - Relatively easy conversion to other formats (using transformations and stylesheets)

**Drawbacks of XML**

- Verbosity
  - Writing numbers, dates, images, etc. as text not efficient
  - Syntax of XML (e.g., element name repeated in closing tag)
  - Common use of whitespace for indentation (not obligatory, of course)
- Complexity
  - Inherited features of SGML (entities, notations, even whole DTD) which are rarely used in modern applications, but have to be supported by processors
  - Technical restrictions, e.g.:
    - Elements can not overlap (trees, not DAGs)
    - Binary content not allowed (there are some solutions - we will learn)
    - Requirement of exactly one root element impractical

**Alternatives to XML**

**Alternatives to XML – text**

- For text-oriented applications of XML:
  - TeX and LaTeX
  - Direct tagging in graphical text editors
    - flat styles
    - more advanced solutions, e.g., Adobe FrameMaker
  - "Lightweight markup"
    - MediaWiki
    - AsciiDoc, OrgMode, and others

---

**Alternatives to XML – data**

- For "modern" applications of XML:
  - JSON (JavaScript Object Notation)
    - more compact than XML
    - often used instead of XML in AJAX-like solutions
  - YAML
    - similar to JSON, but more advanced
  - CSV
    - simple and poor
  - ASN.1, EDIFACT
    - different approach (not so generic)

---

**Where does XML make sense?**

- Text-oriented applications
  - As source format for further processing
  - To denote metadata, structural dependencies, links, etc.
- Data-oriented applications
  - When structural text or tree-like structure appears in a natural way; e.g., business documents interchange
  - When interoperability more important than efficiency
    - public administration services, external business partners, heterogeneous environment

But XML (read also "WebService") is maybe not the best format to transfer arrays of numbers between nodes performing a physical process simulation.

---

Don’t force to use XML when there are better solutions.
Modelling XML Applications

Patryk Czarnik
XML and Applications 2013/2014
Lecture 2 - 14.10.2013

Standards for defining structure of XML documents

- **DTD**
  - origins from SGML (1974)

- **XML Schema** - W3C Recommendation(s)
  - version 1.0 - 2001
  - version 1.1 - 2012

- **Relax NG**
  - OASIS Committee Specification – 2001
  - ISO/IEC 19757-2 – 2003

- **Schematron**
  - alternative standard and alternative approach
  - several version since 1999
  - impact on XML Schema 1.1

XML application (recall)

- **XML application (zastosowanie XML)**
  - A concrete language with XML syntax

  Typically defined as:
  - Fixed set of acceptable tag names (elements and attributes, sometimes also entities and notations)
  - Structure enforced on markup, e.g.: "<person> may contain one or more <first-name> and must contain exactly one <surname>"

  Semantics of particular markups (at least informally)

Benefits of formal definition

- Tangible asset resulting from analysis & design
  - Formal, unambiguous definition of language
  - Reference for humans (document authors and readers, programmers and tool engineers)

  Ability to validate documents using tools or libraries
  - Programs may assume correctness of the content of validated documents (less conditions to check!)

  Content assist in editors
  - autocomplete during typing, stub document generation

Document Type Definition (DTD)

- Defines structure of a class of XML documents ("XML application").

  Optional and not very popular in new applications.
  - Replaced by XML Schema and alternative standards.
  - It is worth to know it, though. Important for many technologies created 10-30 years ago and still in use.

  Contains declarations of:
  - elements ('element types' to be precise)
  - attributes ('attribute lists'...)
  - entities - described last week
  - notations - extremely rarely used, we'll skip them

Two levels of document correctness (recall)

- Document is well-formed (poprawny składniowo) if:
  - conforms to XML grammar,
  - and satisfies additional well-formedness constraints defined in XML recommendation.
  - Then it is accessible by XML processors (parsers).

- Document is valid (poprawny strukturalnie, "waliduje się") if additionally:
  - is consistent with specified document structure definition;
  - from context: DTD, XML Schema, or other;
  - in strict sense (DTD): satisfies validity constraints given in the recommendation.

  Then it is an instance of a logical structure and makes sense in a particular context.

Example DTD (fragments)

```
<ELEMENT student (first-name, surname, birth-date)>  
  <ELEMENT first-name (#PCDATA)>  
  <ELEMENT surname (#PCDATA)>  
  <ELEMENT birth-date (#PCDATA)>  
</ELEMENT>
```
Element declaration in DTD

- Element name
- Element type: one of:
  - EMPTY
  - ANY
  - (content specification)
- Content specification is built of:
  - element names
  - #PCDATA token
- joint together using basic regular expression operators.
  - "'" #PCDATA is allowed only under special conditions

Types in XML Schema

- Concept of type – one of basic distinctions wrt DTD
  - Elements and attributes have specified types
  - Type specify allowable content of an element / attribute
  - for elements – also their attributes
  - type spec. does not include identity constraints
  - Type independent of element (or attribute) name
  - many elements may have the same type
  - elements with the same name may have different types
  - "in different places"

Symbols in DTD element specifications

- Parenthesis ( )
- Occurrence indiectors (postfix operators)
  - ? - zero or one
  - * - zero or more
  - + - one or more
  - no symbol - exactly one
- Combination (infix associative operators)
  - , - sequence (all in the given order)
  - | - choice (one of the given)

Types - categorisation

- Types can be categorised with respect to:
  - complexity
    - complex types define tree-level structure: subelements and attributes; can be applied to elements only
    - simple types define text-level content; they can be applied to elements and attributes
  - scope
    - named types are defined in global scope and can be used many times
    - anonymous types are defined in the place of use
  - origin
    - predefined / built-in – provided by XML Schema
    - user-defined

XML Schema

- Replacement for DTD in new applications of XML
- Separate W3C standard
  - v 1.0 in 2001 - 3 recommendations
  - v 1.1 in 2012 - 2 recommendations
- "XML Schema definition" (*.xsd) is itself XML document
- Similar capabilities for tree-level structure specification
- Much more capabilities than in DTD for:
  - text-level content ("simple types") "datatypes"
  - modularisation of the definition
    (type inference, imports, namespace support)
  - identity constraints (keys and references)
    - in v 1.1 also more advanced constraints
  - Much more verbose than DTD

Element declaration

```xml
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="student">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="first-name" type="xs:string" minOccurs="1" maxOccurs="3"/>
        <xs:element name="last-name" type="xs:string" minOccurs="1" maxOccurs="3"/>
        <xs:element name="birth-date" type="xs:date"/>
        <xs:element name="identification">
          <xs:complexType>
            <xs:choice>
              <xs:element name="PESEL" type="xs:string"/>
              <xs:element name="passport-nr" type="xs:string"/>
              <xs:element name="country" type="xs:string"/>
            </xs:choice>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

More possibilities

- see lab classes

Basic things to look in the examples

- "students" - several ways to write a schema for the same document
  - Structure of DTD, structure of XML Schema definition
  - Typical element definition
  - Controlling number of occurrences
  - Sequence and choice
  - Building complex models (nested groups)
  - Defining attributes in schema and DTD

More details in examples!

- Taking our experience and students' opinions into account we will try not to copy standard specifications onto slides but rather to show by examples:
  - some typical usage,
  - different paths to do a thing – so you can choose your approach depending on needs,
  - chosen cases of advanced usage and rarely used features – it is impossible to show all of them during a short lecture,
  - some good and bad practices.
- It also means, in particular, that slides are not a complete source of knowledge required to pass the exam.

Disclaimer

- Avoiding code duplication and different ways of writing definitions in schemas
  - Local definitions vs global definitions
  - Anonymous types vs named (global) types
  - Named groups
  - Extending complex types
  - Mixed content
    - DTD approach – (#PCDATA | a | b)*
    - Mixed content with controlled subelements – schema only
  - Any order (xs:all) – schema only
Model groups

- Element content defined with model groups:
  - sequence – all in the given order
  - choice – one of the given choices
  - all – all given elements in any order
- sequence and choice – may be nested, multiplied, etc.
- all – restricted
  - may not be mixed with sequence and choice
  - may not be nested
- can contain only elements with different names and occurrence number ≤ 1

Namespaces – motivation

- Same names of tags may denote different things.
- Problematic especially when combining document fragments from different sources into one document.

XML namespaces – realisation

- **Namespace name** (identyfikator przestrzeni nazw)
  - globally unique identifier
  - Universal Resource Identifier (URI) in XML v1.0
  - Internationalized Resource Identifier (IRI) in XML v1.1
- **Namespace prefix** (prefixs przestrzeni nazw)
  - local, for convenient reference
  - Local for document or fragment
  - Processors should not depend on prefixes!
- Names resolved and interpreted as pairs: (namespace name, local name)
- To make things more complex:
  - scope and overriding
  - default namespace

Usage of namespaces and prefixes

```xml
<article code="A1259">  
title: Assignment in Pascal and C</title>  
<author>    
  <name>/firstName surname (surname)/</name>  
  <address>    
    <code>0.234</code>  
  </address>  
</author>  
<body>    
  <paragraph>    
    Assignment is written as <$code xx = \$xx $code> in C    
    and <$code xx = \$xx $code> in Pascal.    
  </paragraph>  
</body>  
</article>
```

Namespaces – overriding and scopes

```xml
<pre code="A1259">  
<code pre="http://xml.mimuw.edu.pl/>article"  
<code pre="http://xml.mimuw.edu.pl/>text-document"  
<author>    
  <name>/firstName surname (surname)/</name>  
  <address>    
    <code>0.234</code>  
  </address>  
</author>  
<body>    
  <paragraph>    
    Assignment is written as <$code xx = \$xx $code> in C    
    and <$code xx = \$xx $code> in Pascal.    
  </paragraph>  
</body>  
</pre>
```

Namespaces – supplement

- **Qualified name** – name with non-empty ns.URI
- **Unqualified name** – name with null (not assigned) ns.
- elements without prefixes when no default namespace
- attributes without prefixes – always

Namespace name

- Only identifier, even if in form of an address!
- Should be in form of URI /IRI; some processors do not check it, though
- Pay attention to every character (uppercase/lowercase, etc.) – most processors simply compare strings

XML namespaces may be used not only for element and attribute names – e.g. type names in XML Schema

Namespace awareness

- A document may be well-formed as XML while erroneous from the point of view of namespaces.
  - For some applications (usually old ones...) such document might be proper and usable.
- Modern parsers can be configured to process namespaces or not.
  - The mentioned document would be
  - parsed successfully by a parser which is not namespace-aware,
  - revoked by a namespace-aware parser.

Default namespace

- Applies to element names which do not have a prefix.
- Does not apply to attributes.

Modularisation options

- Combining multiple files
  - DTD – external parameter entities
  - Schema – include, import, redefine
- Reusing fragments of model definition
  - DTD – parameter entities
  - Schema – groups and attribute groups (in practice equivalent to the above)
  - Schema – types, type derivation (no such feature in DTD)
- Global and local definitions
  - In DTD all elements global, all attributes local
  - In schema both can be global or local, depending on case
**Import or include?**

- **xs:import**
  - Imports foreign definitions to refer to
- **xs:redefine**
  - Includes external definitions, but a local definition overrides external one if they share the same name
- **xs:include**
  - Basic command, almost like textual insertion
  - Imported module must have the same target namespace or no target namespace

A multi-module, namespace-aware project with overused `xs:include` leads to duplication of logic in the software that processes documents (or enforces meta-programming tricks to avoid it). *Based on personal experience*

---

**Schema and namespaces**

- DTD is namespace-ignorant
- XML Schema conceptually and technically bound with XML namespaces
  - Basic approach: one schema (file) = one namespace
    - Splitting one ns into several files technically possible
  - Referring to components from other namespaces available
- Important attributes
  - `targetNamespace` - if given, all `global` definitions within a schema go into that namespace
  - `elementFormDefault`, `attributeFormDefault`
    - should `local` elements or attributes have qualified names?
    - default for both: unqualified
    - typical approach: elements qualified, attributes unqualified
    - setting may be changed for individual definitions

---

**Using namespaces in XML Schema**

Different technical approaches to handle namespaces in XML Schema

- XML Schema ns. bound to `xs` or `xsd`; no target namespace
- XML Schema ns. bound to `xs` or `xsd`; target namespace as default namespace
  - Convenient as long as we don't use keys and keyrefs
- Target namespace bound to a prefix (`tns` by convention)
  - Then we can declare XML Schema as default namespace and avoid using `xs` or `xsd`:
- See examples ns1.xsd - ns4.xsd
Modelling XML Applications (part 2)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 3 - 20.10.2014

Common design decisions

Element or attribute?

- Advantages of attributes:
  - more compact syntax
  - (only in DTD) some features available only for attributes

- Technical restrictions of attributes:
  - only text, without structure marked up
  - multiple attributes with the same forbidden

- General hints
  - Semantic hint: Use elements for data, attributes for metadata (whatever it means in your case :));
  - Presentational hint: If you had to print your document on paper, which parts of text would you print literally (they are elements) and which parts would only have some impact (or no impact) on the way things are presented (they should become attributes)?

Classification of types

- Every element and attribute has a type
  - If not specified: xs:anyType or xs:anySimpleType, resp.
  - “What an element/attribute may contain” but also “How to interpret a value”

Classification of types

- Types by content model
  - Simple type (value of a text node or an attribute; applicable to elements and attributes)
    - atomic type
    - list
    - union
  - Complex type (structure model - subelements and attributes; applicable to elements)
    - empty content
    - element content
    - mixed content
    - simple content

Types by place of definition:

- anonymous - defined locally in place of use
- named - defined globally
  - built-in - defined in XML Schema specification
  - user-defined

Types by means of definition:

- primitive (simple types)
  - defined directly (complex type as a sequence etc.)
  - derived (some built-in types are defined by derivation!)
    - by extension (complex types only)
    - by restriction (complex and simple types)
    - as a list or union (simple types only)
Simple types
- Rich set of built-in types
  - decimal, integer, nonNegativeInteger, long, int, ...
  - boolean, float, double
  - date, time, dateTime, duration, ...
  - string, token, base64Binary, hexBinary, ...
  - See the recommendation for the complete hierarchy
- Defining custom types basing on built-in types
  - by restriction
  - as a list
  - as an union

Defining simple types by restriction
- Constraining facets – properties we can restrict
  - enumeration
  - pattern
  - length, minLength, maxLength
  - totalDigits, fractionDigits
  - maxInclusive, maxExclusive
  - minInclusive, minExclusive
  - whiteSpace
- Used directly in simple type definition:

  ```xml
  <xs:simpleType name="lottonumber">
    <xs:restriction base="xs:integer">
      <xs:minInclusive value="1"/>
      <xs:maxInclusive value="49"/>
    </xs:restriction>
  </xs:simpleType>
  ```

Identity constraints
- Constraints on uniqueness and references
  - Two mechanisms:
    - DTD attribute types ID and IDREF
      - introduced in SGML DTD but still available in XML Schema
      - drawbacks:
        - one global scope, at most one ID per element
        - special form of values – only names allowed
        - IDs and references necessarily in attributes
  - XML Schema identity constraints
    - key, unique, and keyref definitions
    - more powerful and more flexible than ID/IDREF

Value space vs lexical space
- A simple type specifies its
  - value space – set of abstract values
  - lexical space – set of valid text representations

<table>
<thead>
<tr>
<th>Type</th>
<th>Text representations</th>
<th>Abstract value</th>
</tr>
</thead>
<tbody>
<tr>
<td>xs:boolean</td>
<td>false</td>
<td>False</td>
</tr>
<tr>
<td></td>
<td>true</td>
<td>True</td>
</tr>
<tr>
<td>xs:decimal</td>
<td>0.13</td>
<td>0.13</td>
</tr>
<tr>
<td></td>
<td>13.00</td>
<td>13</td>
</tr>
<tr>
<td>xs:string</td>
<td>foo</td>
<td>foo</td>
</tr>
<tr>
<td></td>
<td>bar</td>
<td>bar</td>
</tr>
<tr>
<td>xs:token</td>
<td>foo</td>
<td>foo</td>
</tr>
</tbody>
</table>

List types
- List of values separated with whitespace.
- Not to confuse with sequences
  - list – simple type, no markup structure within
  - sequence – complex type, sequence of subelements
- Compact notation for lists of values
- but
- Harder to process in XML processors (requires additional parsing using regexp etc. - not available e.g. in XSLT 1.0)

```
<xs:simpleType name="lottonumberList"/>
```

Choosing the appropriate type
- Semantic meaning of a simple type:
  - not only a “set of allowed character strings”
  - also the way a value is interpreted!
- Types may affect the validation
  - e.g. leading zeros significant in strings, meaningless in numbers
- Processors may use the information about type, e.g.
  - schema-aware processing in XSLT 2.0 or XQuery
  - sorting, comparison, arithmetic operations
  - JAXB – generation of java classes based on XSD
- Choosing the appropriate type sometimes not obvious
  - phone number, zip code, room number – number or string?

Global scope
- Identifiers unique within document scope
- References valid within document scope
- Expressible in DTD
  - using ID and IDREF, with some restrictions
  - and XML Schema
  - using key and keyref in the root element

Union types
- Union of sets of values
- Possibility to mix values of different primitive types
  - Interpreting values as abstract values hard to perform
  - Nevertheless, a usable feature (e.g. unbounded in XML Schema)

```
<xs:simpleType name="clothingsizeletter">
  <xs:restriction base="xs:token">
    <xs:enumeration value="XS"/>
    <xs:enumeration value="S"/>
    <xs:enumeration value="M"/>
    <xs:enumeration value="L"/>
    <xs:enumeration value="XL"/>
    <xs:enumeration value="XXL"/>
  </xs:restriction>
</xs:simpleType>
```

Global scope identifiers – DTD solution
- DTD solution
  - ```
    <!ATTLIST product id ID #REQUIRED>
    <!ATTLIST product-ref ref IDREF #REQUIRED>
  ```
  - Restrictions:
    - one global scope for all elements
    - only names may become identifiers
    - (they have to start with a letter)
    - ids must be written in attributes, not elements
XML Schema solution:

```xml
<xs:element name="database">
  ...
  <xs:key name="Products">
    <xs:selector xpath="tns:products/tns:product"/>
    <xs:field xpath="?id"/>
  </xs:key>
  <xs:keyref name="ProductsInOrders" refer="tns:Products">
    <xs:field xpath="@ref"/>
  </xs:keyref>
</xs:element>
```

XML Schema keys and references:

- Other possibilities: (see orders example)
  - local scope constraints
  - tuples as key values
- Pay attention:
  - only simple type values can become ids
  - type of reference source and target must be the same
  - default namespace does not work in XPath expressions
    so we have to use a prefix (usually `tns:`)
- Using references can help to avoid data duplication in documents
  - but overuse can make XML look like a flat relational database instead of a nice structural tree
  - find the golden mean!
XML Schema and alternatives

Patryk Czarnik
XML and Applications 2014/2015
Lecture 4 - 27.10.2014

Alternatives

- DTD - obviously
- RELAX NG (Regular Language for XML Next Generation)
  - by James Clark and Murata Makoto
  - OASIS (2001) and ISO (2003) standard
- Schematron
  - by Rick Jelliffe (1999), developed at Academia Sinica (Taiwan), ISO standard (2006)
- Examplotron
  - by Eric van der Vliet, project active in 2001-2003
- XML Schema 1.1
  - W3C Recommendation, 2012
  - borrows some ideas from the alternatives, mainly Schematron

Some possibilities of XML Schema we have not learnt too much

- Deriving complex types by restriction
  - restriction of values set, not necessarily of structure
  - super type more general, subtype more specific
- Substitution groups
  - virtual elements that can be substituted with other ones
- Wildcards (any and anyAttribute)
  - allow to insert any tags or tags from given namespaces
- Nullable elements
  - marking elements as “nonexisting” with xsi:nil attribute
  - personally, I don’t like this idea...
- Specifying actual type with xsi:type in documents

Some drawbacks of XML Schema

- Verbose syntax, specification hard to understand
- Limited power of expression
  - deterministic model required
- no choice between:
  - text content and element content
  - attribute and element
  - content-aware model not available (in 1.0)
- no value-aware constraints other than simple identity constraints (in 1.0)
- Extending types only by appending model fragments at the end of a sequence
  - no direct support for adding new elements to a choice
  - available with other techniques: element groups or substitution groups

Simple example in all standards

DTD
<s:element name="person"> <s:attribute name="first-name" /> <s:attribute name="last-name" type="#PCDATA" /> </s:element>
</s:complexType>
<s:element name="first-name" type="s:string" maxOccurs="unbounded" />
<s:element name="last-name" type="s:string" />
</s:complexType>
</s:schema>

Relax NG - basic ideas

- Clear theoretical basis: Tree automata with regular expressions specifying content in each node
- The same model with appropriate restrictions is used by theoreticians to model DTD or XML Schema, but as a kind of “reverse engineering”
- Compact and readable XML syntax
  - Even more compact plain text syntax available
- Model components such as elements, attributes, or text nodes may be mixed together in definitions
- Modularisation available through define / ref mechanism
  - Equivalent to DTD parameter entities or XSD groups, but with some additional operations to enhance convenience
- No direct support for simple types, but referring to XML Schema types is possible.
Schematron – basic ideas

- Approach different from grammar-based DTD, XSD, and RelaxNG:
  - XPath expressions specify assertions that must hold for instance documents (and elements within them)
  - High power of expression
  - Less convenient (than grammar rules) to write structural definitions
- Official implementation:
  - translation of Schematron scheme to XSLT
  - XSLT evaluates expressions and report errors
- Other implementations available, also for Schematron rules embedded in XML Schema definitions
- XML Schema 1.1 covers most typical Schematron use cases. Probably XSD 1.1 will replace Schematron at all.

Non-deterministic model

- Ambiguous model:
  - It is not possible to state which particle of the model definition is matched, even if whole document is known.
  - Example: (A,A,A) = (A,A) for document AAAAA
- Non-deterministic model
  - Similar to (non-LL(1)) grammars, but extended to trees:
  - During one-pass parsing, it is not possible to determine the appropriate definition particle for a current element when only the start tag of the element is seen
  - Some models may be determined just by definition rearrangement, e.g.: A, A?, A?? = A, (A, A?)?
  - But some models may not:
    - Notable example: (A, B)*, A?
  - XML Schema avoids non-deterministic models!

Model forbidden in XML Schema, valid in Relax NG

Value-aware constraints – Schematron solution

- We’d like to allow both formats:
  - <phone>1234567</phone>
  - <phone>cc:48H</phone>

- No possibility in XML Schema (other than mixed content)
- No problem in RelaxNG:

```
<element name="phone">
  <choice>
    <text>
    </text>
  </choice>
  <group>
    <optional element name="cc" />
    <optional element name="ac" />
    <element name="main" />
    <element name="int" />
    <optional>
    </optional>
  </group>
</element>
```

Value-aware constraints – XSD 1.1 solution

- Similarly as before, we want to allow both
  - <section title="Introduction"></section>
  - and
  - <section title="What does &lt;p&gt;be or not to be&lt;/p&gt; mean in fact?" title="..."

```
<element name="section">
  <choice>
    <attribute name="title" />
    <ref name="text-model" />
  </choice>
  <choice>
    <ref name="text-model" />
  </choice>
</element>
```

Value-aware constraints

- Relations and constraints other than equality
  - <order>
    <order-date>2014-01-20</order-date>
    <delivery-date>2014-01-25</delivery-date>
    <item>
      <value>199.90</value>
    </item>
    <item>
      <value>20</value>
    </item>
  </order>

Content-aware model

- We want to choose one of models depending on the value of a field (attribute or element), e.g.
Content aware model – Schematron solution

```xml
<pattern>
<title>Payment method</title>
<rule context="payment">
  <assert test="method = 'transfer' and due-date or method = 'card' and card-no and holder and exp-date">
    Payment with a Transfer or a Card.
  </assert>
</rule>
</pattern>
```

- It will work provided that type of payment will be defined generally and that all elements will have appropriate types
- But Schema 1.1 offers more natural solution...

---

Content aware model – XSD 1.1 solution

```xml
<xs:element name="payment" type="Payment">
  <xs:alternative test="method = 'transfer'" type="TransferPayment" />
  <xs:alternative test="method = 'card'" type="CardPayment" />
  <xs:alternative test="not(method = ['transfer', 'card', 'cash'])" type="x:s:error" />
</xs:element>
```

```xml
<xs:complexType name="Payment">
  <xs:alternative test="method = 'transfer'" type="TransferPayment" />
  <xs:alternative test="method = 'card'" type="CardPayment" />
  <xs:complexType name="TransferPayment">
    <xs:complexContent base="Payment">
      <xs:sequence>
        <xs:element name="due-date" type="xs:date" />
      </xs:sequence>
    </xs:complexContent>
  </xs:complexType>
  <xs:complexType name="CardPayment">
    <xs:complexContent base="Payment">
      <xs:sequence>
        <xs:element name="card-no" type="CardNumber" />
        <xs:element name="holder" type="xs:string" />
      </xs:sequence>
    </xs:complexContent>
  </xs:complexType>
</xs:complexType>
```

---

So why do most people use XSD 1.0?

- Alternatives?
  - XML Schema is a W3C Recommendation
  - In this business it means even more than an ISO standard
  - Some popular technologies, with Web Services (WSDL) at the first place, use XML Schema
- XML Schema 1.1?
  - Still little support in generally available software, especially programming libraries
XML in Programming

Patryk Czarnik
XML and Applications 2014/2015
Lecture 5 – 3.11.2014

XML and Java

- Propaganda
  - Java platform provides device-independent means of program distribution and execution.
  - XML is a platform-independent data carrier.
- Practice
  - Java - one of the most popular programming languages, open and portable.
  - Very good XML support in Java platform.
  - Many technologies use XML.
  - Of course you can find very good (or at least not bad) XML support on other programming platforms, but we have to choose one for presentation and exercises.

XML in programming – what for?

- To access data in XML format
- To use XML as data carrier (storage and transmission)
- To support XML applications (Web, content management)
- To make use of XML-related standards
  - (XML Schema, XInclude, XSLT, XQuery, XLINK, ...)
- To develop or make use of XML-based technology
  - XML RPC, Web Services (SOAP, WSDL)
  - REST, AJAX

XML in Java – standards

Both included in Java Standard Edition since v.6
- Java API for XML Processing (JAXP 1.x – JSR-206)
  - many interfaces and few actual classes, “factories” and pluggability layer
  - support for XML parsing and serialisation (DOM, SAX, StAX)
  - support for XInclude, XML Schema, XPath, XSLT
- Java API for XML Binding (JAXB 2.x – JSR-222)
  - binding between Java objects and XML documents
  - annotation-driven
  - strict relation with XML Schema

XML in programming – how?

- Bad way
  - Treat XML as plain text and write low-level XML support from scratch
- Better approach
  - Use existing libraries and tools
- Even better
  - Use standardised interfaces independent of particular suppliers

Classification of XML access models

And their “canonical” realisations in Java

- Document read into memory
  - generic interface: DOM
  - interface depending on document type/schema: JAXB
- Document processed node by node
  - event model (push parsing): SAX
  - streaming model (pull parsing): StAX

Document Object Model

- W3C Recommendations
  - DOM Level 1 – 1998
  - DOM Level 3 – 2004
  - Several modules. We focus on DOM Core here
- Document model and universal API
  - independent of programming language (IDL)
  - independent of particular XML application
- Used in various environments
  - notable role in JavaScript / ECMA Script model
  - available (in some form) for all modern programming platforms

Primary DOM types

DOM key ideas

- Whole document in memory
- Tree of objects
- Generic interface Node
- Specialised interfaces for particular kinds of nodes
- Available operations
  - reading document into memory
  - creating document from scratch
  - modifying content and structure of documents
  - writing documents to files / streams
### Example: problem introduction

Count the number of seats in rooms equipped with a projector.

```java
private void processRoom( Node node ) {
    for( Node child = node.getFirstChild(); child != null; child = child.getNextSibling() ) {
        if( child.getNodeType() == Node.ELEMENT_NODE ) {
            String name = child.getNodeName();
            if( name.equals( "room" ) ) {
                NodeList list = node.getChildNodes();
                for( int i = 0; i < list.getLength(); ++i ) {
                    Node node = list.item( i );
                    if( node.getNodeType() == Node.ELEMENT_NODE ) {
                        String type = node.getNodeName();
                        if( type.equals( "equipment" ) ) {
                            NodeList equipmentList = node.getChildNodes();
                            for( int j = 0; j < equipmentList.getLength(); ++j ) {
                                Node equipmentNode = equipmentList.item( j );
                                if( equipmentNode.getNodeType() == Node.ELEMENT_NODE ) {
                                    String name = equipmentNode.getNodeName();
                                    if( name.equals( "projector" ) ) {
                                        ++seats;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Whole example in `CountSeats_DOM_Generic.java`

### Approaches to using DOM

- Two approaches in DOM programming
  - Use only generic `Node` interface
  - Use specialised interfaces and convenient methods

### Using specialised interfaces (fragments)

```java
private void processRoom( Node node ) {
    for( Node child = node.getFirstChild(); child != null; child = child.getNextSibling() ) {
        if( child.getNodeType() == Node.ELEMENT_NODE ) {
            String name = child.getNodeName();
            if( name.equals( "room" ) ) {
                NodeList list = node.getChildNodes();
                for( int i = 0; i < list.getLength(); ++i ) {
                    Node node = list.item( i );
                    if( node.getNodeType() == Node.ELEMENT_NODE ) {
                        String type = node.getNodeName();
                        if( type.equals( "equipment" ) ) {
                            NodeList equipmentList = node.getChildNodes();
                            for( int j = 0; j < equipmentList.getLength(); ++j ) {
                                Node equipmentNode = equipmentList.item( j );
                                if( equipmentNode.getNodeType() == Node.ELEMENT_NODE ) {
                                    String name = equipmentNode.getNodeName();
                                    if( name.equals( "projector" ) ) {
                                        ++seats;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Whole example in `CountSeats_DOM_Specialized.java`

### DOM in Java example

- Parsing and basic processing

```java
private void processRoom( Node node ) {
    for( Node child = node.getFirstChild(); child != null; child = child.getNextSibling() ) {
        if( child.getNodeType() == Node.ELEMENT_NODE ) {
            String name = child.getNodeName();
            if( name.equals( "room" ) ) {
                NodeList list = node.getChildNodes();
                for( int i = 0; i < list.getLength(); ++i ) {
                    Node node = list.item( i );
                    if( node.getNodeType() == Node.ELEMENT_NODE ) {
                        String type = node.getNodeName();
                        if( type.equals( "equipment" ) ) {
                            NodeList equipmentList = node.getChildNodes();
                            for( int j = 0; j < equipmentList.getLength(); ++j ) {
                                Node equipmentNode = equipmentList.item( j );
                                if( equipmentNode.getNodeType() == Node.ELEMENT_NODE ) {
                                    String name = equipmentNode.getNodeName();
                                    if( name.equals( "projector" ) ) {
                                        ++seats;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Whole example in `CountSeats_DOM_Generic.java`

### DOM in Java example

- Visiting nodes in the tree

```java
private void processRoom( Node node ) {
    for( Node child = node.getFirstChild(); child != null; child = child.getNextSibling() ) {
        if( child.getNodeType() == Node.ELEMENT_NODE ) {
            String name = child.getNodeName();
            if( name.equals( "room" ) ) {
                NodeList list = node.getChildNodes();
                for( int i = 0; i < list.getLength(); ++i ) {
                    Node node = list.item( i );
                    if( node.getNodeType() == Node.ELEMENT_NODE ) {
                        String type = node.getNodeName();
                        if( type.equals( "equipment" ) ) {
                            NodeList equipmentList = node.getChildNodes();
                            for( int j = 0; j < equipmentList.getLength(); ++j ) {
                                Node equipmentNode = equipmentList.item( j );
                                if( equipmentNode.getNodeType() == Node.ELEMENT_NODE ) {
                                    String name = equipmentNode.getNodeName();
                                    if( name.equals( "projector" ) ) {
                                        ++seats;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Whole example in `CountSeats_DOM_Specialized.java`

### Approaches to using DOM

- Two approaches in DOM programming
  - Use only generic `Node` interface
  - Use specialised interfaces and convenient methods

### Using specialised interfaces (fragments)

```java
private void processRoom( Node node ) {
    for( Node child = node.getFirstChild(); child != null; child = child.getNextSibling() ) {
        if( child.getNodeType() == Node.ELEMENT_NODE ) {
            String name = child.getNodeName();
            if( name.equals( "room" ) ) {
                NodeList list = node.getChildNodes();
                for( int i = 0; i < list.getLength(); ++i ) {
                    Node node = list.item( i );
                    if( node.getNodeType() == Node.ELEMENT_NODE ) {
                        String type = node.getNodeName();
                        if( type.equals( "equipment" ) ) {
                            NodeList equipmentList = node.getChildNodes();
                            for( int j = 0; j < equipmentList.getLength(); ++j ) {
                                Node equipmentNode = equipmentList.item( j );
                                if( equipmentNode.getNodeType() == Node.ELEMENT_NODE ) {
                                    String name = equipmentNode.getNodeName();
                                    if( name.equals( "projector" ) ) {
                                        ++seats;
                                    }
                                }
                            }
                        }
                    }
                }
            }
        }
    }
}
```

Whole example in `CountSeats_DOM_Specialized.java`

### Mapping XML to Java

- High-level view on documents:
  - instead of `Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().parse( fileName );`
  - we simply have `Document doc = DocumentFactory.newInstance().newDocument();`

### JAXB 2.x architecture

- Application operates basing on (usually annotated) "JAXB classes"
  - generated from a schema
  - or written manually

### JAXB example

- We generate java classes basing on our schema:
  - `xjc -d src -p package_name school.xsd`

### One of generated classes:

```java
@Generated( "com.sun.tools.xjc.Driver", "Bot", "" )
public class Room {

    private String _name = "Room", _propOrder = { 
        "name", "floor", "equipment", "seats" };
    public Room( String name ) {
        this._name = name;
    }
    
    public String getName() {
        return _name;
    }
    public void setName( String name ) {
        _name = name;
    }
    
    public int getFloor() {
        return _floor;
    }
    public void setFloor( int _floor ) {
        _floor = _floor;
    }
    
    public String getEquipment() {
        return _equipment;
    }
    public void setEquipment( String _equipment ) {
        _equipment = _equipment;
    }
    
    public int getSeats() {
        return _seats;
    }
    public void setSeats( int _seats ) {
        _seats = _seats;
    }
    
    // more getters and setters...
}
```

All generated classes are in `..._jAXB_generated and the program in CountSeats_JAXB`
### JAXB example

```java
JAXBContext jaxbContext = JAXBContext.newInstance(Rooms.class);
Unmarshaller u = jaxbContext.createUnmarshaller();
Rooms rooms = (Rooms) u.unmarshal(new File(fileName));
if(rooms != null) {
    processRooms(rooms);
}
```

```java
private void processRooms(Rooms rooms) {
    for(Room room : rooms.getRooms()) {
        if(room.getEquipment().isProjector())
            & & room.getSeats() != null {
                sun += room.getSeats();
            } } }
```

### Event model

- Document seen as a sequence of events
  - "an element is starting"
  - "a text node appears", etc.
- Programmer provides code fragments - "event handlers"
- Parser reads a document and
  - controls basic syntax correctness
  - calls programmer's code relevant to actual events
- Separation of responsibility:
  - Parser responsible for physical-level processing
  - Programmer responsible for logical-level processing

### SAX events in run

```xml
<xml-stylesheet .../>
<room>
  <equipment projector="true"/>
</room>
<seats>
  68
</seats>
</room>
```

### SAX example (fragments)

```java
public class CHandler implements ContentHandler {
    ...
    public void startElement(String uri, String localName, String qName, Attributes atts) throws SAXException {
        switch(state){
        case In_ROOM:
            if("equipment".equals(qName)) {
                String projector = atts.getValue("projector");
                if("true".equals(projector))
                    ...
            case CHandler_States.IN_ROOM_WITH_PROJECTOR:
            ...
        ...
    ...
```
**SAX filters**

- Motivation: joining ContentHandler-like logic into chains
- Realisation:
  - interface XMLFilter
    (XMLReader having a parent XMLReader)
  - in practice filters implements also ContentHandler
  - convenient start-point: XMLFilterImpl
- Typical implementation of a filter:
  - handle incoming events like in a ContentHandler
  - pass events through by manual method calls on the next item in chain
- Filters can:
  - pass or halt an event
  - modify an event or a sequence of events!

**SAX – typical problems**

- To make implementations portable – we should manually join adjacent text nodes in an element
  - StringBuilder is a convenient class
- The same method called for different elements, in different contexts
  - Typical solution – remembering the state:
    - one boolean flag in simplest cases
    - enum is usually enough
    - elaborated structures may be required for complex logic
    - It may become tedious in really complex cases.

**Possible usage of SAX filters**

![Diagram showing possible usage of SAX filters](image)

**StAX example with XMLStreamReader (fragments)**

```java
import XMLStreamReader;
reader = x relaxationXMLStreamReader(new FileReader(file));
while (reader.hasNext()) {
  if (reader.isStartElement()) {
    if (reader.getName().equals("rooms") {
      this.processRooms();
    }
  }
  reader.next();
}

while (reader.hasNext()) {
  if (reader.isStartElement()) {
    if (reader.getName().equals("rooms") {
      this.processRooms();
    }
  }
  reader.next();
}
```

**StAX Example**

- Whole programs:
  - CountSeats_Stax_Stream presents the usage of the low-level XMLStreamReader
  - CountSeats_Stax_Event presents the usage of XMLEventReader

**StAX**

- Streaming API for XML
- Available in Java SE since version 6
- Two levels of abstraction:
  - XMLStreamReader
    - one object for all purposes
    - most efficient approach
  - XMLEventReader
    - subsequent events (nodes) provided as separate objects
    - more convenient for high-level programming,
      especially when implementing modification of the document “on-the-fly”

**StAX: Pull instead of being pushed**

- Alternative for event model
  - application “pulls” events/nodes from parser
  - processing controlled by application, not parser
  - idea analogous to: iterator, cursor, etc.
- More intuitive control flow
  - reduced need of remembering the state etc.
- Advantages of SAX saved
  - high efficiency
  - possibility to process large documents

**SAX Filters – example?**

- We’re not going to resolve our example program using filters, as it makes a little sense.
- An example filter can be found in more_sax/lowerCaseFilter

**StAX example with XMLEventReader (fragments)**

```java
import XMLEventReader;
reader = x relaxationXMLEventReader(new FileReader(file));
while (reader.hasNext()) {
  if (reader.isStartElement()) {
    if (reader.getName().equals("rooms") {
      this.processRooms();
    }
  }
  reader.next();
}
```

**SAX – typical problems**

- To make implementations portable – we should manually join adjacent text nodes in an element
  - StringBuilder is a convenient class
- The same method called for different elements, in different contexts
  - Typical solution – remembering the state:
    - one boolean flag in simplest cases
    - enum is usually enough
    - elaborated structures may be required for complex logic
    - It may become tedious in really complex cases.

**Possible usage of SAX filters**

![Diagram showing possible usage of SAX filters](image)

**StAX example with XMLStreamReader (fragments)**

```java
import XMLStreamReader;
reader = x relaxationXMLStreamReader(new FileReader(file));
while (reader.hasNext()) {
  if (reader.isStartElement()) {
    if (reader.getName().equals("rooms") {
      this.processRooms();
    }
  }
  reader.next();
}
```

**StAX Example**

- Whole programs:
  - CountSeats_Stax_Stream presents the usage of the low-level XMLStreamReader
  - CountSeats_Stax_Event presents the usage of XMLEventReader

**StAX**

- Streaming API for XML
- Available in Java SE since version 6
- Two levels of abstraction:
  - XMLStreamReader
    - one object for all purposes
    - most efficient approach
  - XMLEventReader
    - subsequent events (nodes) provided as separate objects
    - more convenient for high-level programming,
      especially when implementing modification of the document “on-the-fly”

**StAX: Pull instead of being pushed**

- Alternative for event model
  - application “pulls” events/nodes from parser
  - processing controlled by application, not parser
  - idea analogous to: iterator, cursor, etc.
- More intuitive control flow
  - reduced need of remembering the state etc.
- Advantages of SAX saved
  - high efficiency
  - possibility to process large documents

**SAX Filters – example?**

- We’re not going to resolve our example program using filters, as it makes a little sense.
- An example filter can be found in more_sax/lowerCaseFilter

**SAX – typical problems**

- To make implementations portable – we should manually join adjacent text nodes in an element
  - StringBuilder is a convenient class
- The same method called for different elements, in different contexts
  - Typical solution – remembering the state:
    - one boolean flag in simplest cases
    - enum is usually enough
    - elaborated structures may be required for complex logic
    - It may become tedious in really complex cases.
Control flow in SAX

Which model to choose? (1)
- Document tree in memory:
  - small documents (must fit in memory)
  - concurrent access to many nodes
  - creating new and editing existing documents “in place”
- Generic document model (like DOM):
  - not established or not known structure of documents
  - lower efficiency accepted
- XML binding (like JAXB):
  - established and known structure of documents
  - XML as a data serialisation method

Control flow in StAX

Which model to choose? (2)
- Processing node by node
  - potentially large documents
  - relatively simple, local operations
  - efficiency is the key factor
- Event model (SAX):
  - using already written logic (SAX is more mature)
  - filtering events, asynchronous events
  - several aspects of processing during one reading of document (filters)
- Streaming model (like StAX):
  - processing depending on context; complex states
  - processing should stop after the item is found
  - reading several documents simultaneously

StAX features
- API for reading documents:
  XMLStreamReader, XMLEventReader
- API for writing documents:
  XMLStreamWriter, XMLEventWriter
- Filters
  - simple definition of a filter: accept(Event): boolean
  - “filtered readers”
XML in Programming 2
(just a small supplement)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 6 - 17.11.2014

Applications of Transformers

- Simple:
  - invoking XSLT transformations from Java
  - changing internal representation of XML in our program

- Tricky:
  - parsing and writing documents, e.g. serialisation of a DOM tree
  - serialisation of modified (or generated) sequences of SAX events
  - (together with SAX filters) enabling “on-the-fly” processing of large XML documents

Features of JAXP

- 3 models of XML documents in Java: DOM, SAX, StAX
  - Formally JAXB is a separate specification
- Reading and writing documents
- Transformations of XML documents (Transformers)
  - applying XSLT in our programs
  - translating internal form of representation
- XPath support
- Validation
  - against DTD (only during parsing)
  - against XML Schema (during parsing or using Validators)
  - against XML Schema 1.1, Relax NG, or other alternative standards – when implementation supports

Editing XML documents

- More natural when whole document present in memory
  - DOM – generic API
  - JAXB – deep embedding of XML in application model
- Harder, but possible, using node-by-node processing
  - required when processing big documents while having little memory
  - suggested for big (“long and flat”) documents and simple local operations – then we can save substantial resources
- StAX – possible using “writers”
  - IMX XMLEventWriter more convenient than XMLStreamWriter
  - SAX
    - no direct support for editing/writing
    - available indirect solution: SAX filters and Transformer

XPath support in Java

- DOM XPath module implementation
  - org.w3c.dom.xpath
- officially not a part of Java SE, but available in practice (by inclusion of Xerces in Oracle Java SE runtime)
- JAXP XPath API
  - javax.xml.xpath
  - most efficient when applied for documents in memory (DOM trees)
  - our examples show this solution
- Note: using XPath may significantly reduce developer’s work, but the application may be less efficient (than if we used SAX, for example)

Validator

- Against DTD
  - setValidating(true) before parsing
- Against XML Schema (or other schema formats, if implementation supports)
  - setSchema(schema) before parsing
  - using Validator

Validator API

- validate(Source) – only checking of correctness
- validate(Source, Result) – augmented document returned
  - not possible to use as Transformer – source and result must be of the same kind
  - (my private observation) – not always working as expected
Web services

Pre-XML solutions
- ANSI Accredited Standards Committee X12 sub-group
  - USA national standard
  - used mainly in America
- EDIFACT
  - international standard (UN/CEFACT and ISO)
  - used mainly in Europe and Asia

Evolution of internet applications
- human → human
  - email
  - WWW sites written manually
- application → human
  - web applications (e.g. an internet shop)
- application → application
  - “electronic data interchange”
    - low-level technologies and ad-hoc solutions
    - pre-XML standards (e.g. EDIFACT)
    - “web services”
    - REST, AJAX, etc.

EDIFACT characteristic
- Format
  - text
  - hardly readable
  - tree structure
- Predefined dictionaries
  - 193 message types
  - 279 segments
  - 186 elements
  (counted for version 08a, 2008)

Electronic data interchange (EDI) – motivation
- How to interchange data between companies / institutions (B2B)?
  - paper
  - electronic data interchange
- How to establish EDI protocol?
  - customer receives (or buys) a tool from provider
  - smaller partner complies to bigger partner
  - ad-hoc created conversion tools
- Standard
  - Standard deployment levels
  - software developed according to standard from beginning
  - interface added to legacy system

EDIFACT
- EDIFACT message example

XML EDI
- Idea: use XML as data format for EDI
- Traditional EDI
  - Documents unreadable without specification
  - Compact messages
  - Centralised standard maintenance
  - Changes in format requires software change
  - Specialised tools needed
- XML EDI
  - “Self-descriptionning” documents format
  - Verbose messages
  - “Pluggable”, flexible standards
  - Well written software ready to extensions of format
  - XML-format layer handled by general XML libraries

XML EDI flexibility
- Format flexibility
  - Structures: choosing, repeating, nesting, optionality
  - Format extensions and mixing via namespaces
- Applications
  - Data interchange between partners’ systems
  - Web interface (with little help from XSLT)
- Web Services integration
Web Services

- Idea: a website for programs (instead of people)
- General definition
  - communication based on high-level protocols
  - structural messages
  - services described
  - searching services
- Concrete definition: “Classical” Web-Services
  - HTTP or other protocols
  - SOAP
  - WSDL
  - UDDI
  - Web Services Interoperability

Web Services standardisation

- SOAP (initially Simple Object Access Protocol):
  - beginnings: 1998
  - v1.1: W3C Note, 2001 (still in use)
  - v1.2: W3C Recommendation, June 2003 (also used)
- Web Services Description Language:
  - W3C Note, 2001 (most applications use this version!)
  - v2.0: W3C Recommendation, June 2007
- Universal Description Discovery and Integration:
  - OASIS project

Web Services standardisation (2)

- Web Services Interoperability - levels of WS compliance:
  - WS-I Basic Profile, Simple Soap Binding Profile, ...
  - WS-* standards: various standards, usually not W3C:
    - WS-Eventing, WS-Addressing, WS-Routing, WS-Security
- Business Process Execution Language (OASIS) - WS semantics description, programming using WS as building blocks

Classical vision of web services operation

- Service (and its provider)
- SOAP
- SOAP
- published WSDL
- reads WSDL
- Service registry (UDDI)
- In fact, most of deployed solutions don’t use the UDDI layer

SOAP 1.2 message

```xml
<soap:Envelope
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <t:Trans xmlns:t="http://www.w3schools.com/transaction/" soap:mustUnderstand="1">234</t:Trans>
  </soap:Header>
  <soap:Body>
    <m:GetPrice xmlns:m="http://www.w3schools.com/prices">
      <m:Item>Apples</m:Item>
      <m:Currency>PLN</m:Currency>
    </m:GetPrice>
  </soap:Body>
</soap:Envelope>
```

SOAP 1.2 communication protocol

- Built on top of existing transport protocol (HTTP or other)
- Message format
  - XML message with optional binary attachments
  - headers (optional XML elements) and body content
  - envelope and some special elements defined in standard
  - implementation-dependent content
- Differences to RPC, CORBA, DCOM etc.:
  - data represented in extensible, structural format (XML)
  - data types independent of platform (XML Schema)
  - lower efficiency

SOAP message – general form

```xml
<soap:Envelope
  xmlns:soap="urn:SOAP"
  xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
  soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
  <soap:Header>
    <soap:Fault>
      <soap:Receiver>soap:Fault</soap:Receiver>
      <soap:faultstring>Data missing</soap:faultstring>
      <soap:faultdetail>
        <soap:exception>Found no student identified</soap:exception>
        <soap:ind>123</soap:ind>
      </soap:faultdetail>
    </soap:Fault>
  </soap:Body>
</soap:Envelope>
```
***SOAP – more info***

- Request and response have the same structure.
  - In fact, we can think of SOAP as a document transport protocol, not necessarily in client-server architecture.
- Header part optional, Body part required.
- Restrictions on XML part:
  - no DTD (and external entity references),
  - no processing instructions.
- Although SOAP allows many body elements (elements within soap:Body), WS-I BP requires exactly one.
- To make applications portable we should follow this restriction.

---

***WSDL and SOAP interaction***

- Basically – specified through binding element in WSDL
  - not so simple, because of many possibilities
- **RPC style**
  - SOAP XML structure derived basing on operation name and message parts
- **Document style**
  - theoretically designed to allow sending arbitrary content enclosed in XML documents
  - in practice - also used for RPC realisation, but the author of WSDL has to define an appropriate document structure
  - (some tools may be helpful, e.g. bottom-up service generation in Java JAX-WS)
- **Message use:** literal or encoded.
  - We should use literal in modern applications.

---

***WSDL – service description***

- XML document describing a service
- Interface ("visit card") of a service (or set of services)
- Specifies (from abstract to concrete things)
  - XML types and elements (using XML Schema)
  - types of messages
  - port types – available operations, their input and output
  - details of binding abstract operations to a concrete protocol (SOAP in case of "classical" services)
  - ports – concrete instances of services, with their URL
- Splitting definitions into several files and using external schema definitions available

---

***Service registration and discovery***

- **Idea**
  - service provider registers service
  - user searches for service and finds it in registry
- **Universal Description Discovery and Integration (UDDI)**
  - available as service (SOAP)
  - business category-based directory ("yellow pages")
  - searching basing on service name, description ("white pages")
  - registration and updates for service providers

---

***UDDI – issues***

- Main issue – who can register?
  - anybody – chaos and low reliability
  - accepted partners – an institution responsible for access policy needed, no such (widely accepted) institution exists
- **Reality**
  - UDDI rarely used
  - if ever – for "local" SOA-based solutions (intranets)

---

***Web Services advantages and problems***

- **Advantages:**
  - Standardised, platform-independent technology
  - Interoperability
  - Existing tools and libraries
- **Main drawbacks:**
  - **Inefficiency**
    - size of messages → transfer, memory usage
    - data representation translated many times on the road from client to server (and vice versa) → processor usage / time
  - Complex standards, especially when using something more than raw WSDL+SOAP

---

***Service Oriented Architecture***

- Old-school approach for software building (when we have some logic already developed and we want to use it again):
  - link and compile static components – code, libraries, etc.
- **SOA approach:**
  - use working services to obtain existing logic
  - (to make it possible) build your pieces of software as services
  - Result: services built basing on other services
- Main differences (advantages?)
  - we don't have to include a component to use it
  - we avoid not only code duplication, but also a duplication of working logic

---

***Are Web Services good or bad?***

- **SOA and Web Services** give an opportunity to build
  - modular, flexible, and scalable solutions
  - (sometimes) by the cost of irrational inefficiency and complexity
- **Web Service recommended when**
  - Many partners or public service (standardisation)
  - Heterogeneous architecture
  - Text and structural data already present in problem domain
  - Interoperability and flexibility more important than efficiency
- **Web Service?... not necessarily**
  - Internal, homogeneous solution.
  - Binary and flat data
  - Efficiency more important than interoperability
Web services in Java

- Web services and web service clients can be built from scratch in any technology
  - but it would be the same mistake as reading XML documents char by char.
- Low-level technologies:
  - HTTP and HTTP clients supported by XML processing APIs (DOM, SAX, StAX, JAXB, Transformers, ...)
  - SOAP with Attachments API for Java (SAAJ)
    - extension of DOM directly supporting SOAP
  - High level approach (with low level hooks available):
    - Java API for XML Web Services (JAX-WS)

Web services in Java

- WS support (XML APIs, SAAJ, JAX-WS) present in Java SE
  - JAX-WS and some of XML APIs since version 6.0
- Client side:
  - Possible to develop and run WS client in Java SE without any additional libraries!
- Server side:
  - Developing and compiling WS server (without any vendor-specific extensions) available in Java SE
  - Running a service requires an application server and a WS implementation
    - “Big” app servers (Glassfish, JBoss, WebSphere...) have preinstalled WS implementations
    - Lightweight servers (e.g. Tomcat) can be used by applications equipped with appropriate libraries and configuration

Advantages and risks of using JAX-WS

- High level view on web service
  - details of communication and SOAP/XML not (necessarily) visible to a programmer
  - proxy object on client side enables to transparently invoke methods on server-side just like on local objects
- Automatic generation/interpretation of WSDL
  - conformance to WSDL controlled by system
- Bottom-up scenario – easy introduction of WS interface to already existing systems
  - or for programmers not familiar with WSDL/XML details
- Risk of
  - accidental service interface (WSDL)
    - automatically generated, not elaborated enough
    - inefficiency

JAX-WS – introduction

- Annotation-driven
- Uses JAXB to translate Java objects to/from XML
- Central point: Service Endpoint Interface (SEI)
  - Java interface representing a WS port type
    - `kalkulator.Kalkulator` and `paket.Service` in our examples
- Translation between web services world (WSDL) and Java
  - top-down: from WSDL generate Java
    - server side – service interface and implementation skeleton
    - client side – proxy class enabling easy remote invocations
  - both sides – auxiliary classes, usually JAXB counterparts of XML elements appearing in messages
  - bottom-up: from Java code generate WSDL (and treat the Java code as a WS implementation)
    - usually done automatically during application deployment

JAX-WS – main elements

- Class level annotations:
  - `@WebService`, `@SOAPBinding`
- Method-level annotations:
  - `@WebMethod`, `@OneWay`, `@SOAPBinding`, `@RequestWrapper`, `@ResponseWrapper`
- Parameter-level annotations:
  - `@WebParam`
  - `@WebResponse (syntactically a method annotation, applies to what the method returns)`
- Support for specific technologies
  - `@MTOM` - automatically created binary attachments
  - `@Addressing` – adds WS-Addressing headers

JAX-WS – low level hooks

- Providers – low level server side
  - Useful when
    - high efficiency required (e.g. streaming processing)
    - XML technology used in implementation
  - Dispatch – low level client side
  - One way methods
  - Asynchronous client calls
- Handlers and handler chains
  - additional processing of messages between client and server logic
  - one place to perform common logic: logging, authentication, session binding

JAX-WS – examples

Details to note:

- top-down (`Kalkulator`):
  - (different) form of WSDL in RPC and Document styles
  - 3 ways WSDL can be translated to Java (and SOAP)
    - (RPC, document-wrapped, document-bare)
  - `@WebService` annotation in implementation class
- bottom-up (Hello)
  - how annotations affect SOAP messages (and WSDL)
  - how Java objects are represented in SOAP messages (JAXB)
  - high level proxy clients (`Client_Weather_JAXWS`)

SAAJ

- Package `javax.xml.soap`
- Main class – SOAPMessage
- Tree-like representation of SOAP messages
  - extension of DOM
  - easy access to existing and building fresh SOAP messages
  - support for HTTP headers, binary attachments, ...
- Easy sending of requests from client side
  - see example `Client_Weather_SAAJ`
- Possible implementation of server side as a servlet
  - see example `Server_SAAJ`

SAJ

- Package `javax.xml.soap`
- Main class – SOAPMessage
- Tree-like representation of SOAP messages
  - extension of DOM
  - easy access to existing and building fresh SOAP messages
  - support for HTTP headers, binary attachments, ...
- Easy sending of requests from client side
  - see example `Client_Weather_SAAJ`
- Possible implementation of server side as a servlet
  - see example `Server_SAAJ`

JAX-WS architecture

When both sides written in Java...
**REST - motivation**

- Complexity and inefficiency of SOAP-based services led designers/researchers to propose other solutions
  - service-oriented
  - but simpler (and less general) than classical WS
- The most popular alternative these days: Representational State Transfer (REST)
  - Very popular solution for integration of JavaScript clients (AJAX) with servers
  - And mobile clients as well...
  - In Java (EE) available through JAX-RS interface

**JAX-RS - REST in Java**

- Java API for RESTful Services (JAX-RS)
- Annotation driven API
- Support for different ways of passing arguments
- Content-type negotiation
  - the same resource may be available in different formats
- Easy to write HTTP servers
  - REST-specific logic has to be written manually

**REST - basic ideas**

- Service = set of resources
  - resource identified by its URL
  - best practices: URLs unique, resources organised in collections

  http://rest.example.org/service/orders/302312

- Resources
  - are representable
    - e.g. as XML
  - other formats available, a popular one is JSON
  - can be transferred through the net
- HTTP - protocol for remote access to the resources
  - HTTP methods (GET, PUT, etc) used directly

**HTTP methods (in REST, but not only)**

- GET - read the resource
  - no side effects
- PUT - write the resource
  - request body contains new contents
  - for writing new and overriding existing resources
- DELETE - deletes the resource
- POST - "take this piece of data and do something with it"
  - conceptually incompatible with REST ideas
  - used in practice to call remote logic more complex that reading or writing a resource
- OPTIONS, HEAD - no special meaning in REST
  - well, getting last modification time makes sense in REST...
XPath (and XQuery)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 8 - 1.12.2014

XPath - status
- XPath 1.0
  - W3C Recommendation, XI 1999
  - used within XSLT 1.0, XML Schema, XPointer
- XPath 2.0
  - Several W3C Recommendations, I 2007:
    - XML Path Language (XPath) 2.0
    - XQuery 1.0 and XPath 2.0 Data Model
    - XQuery 1.0 and XPath 2.0 Functions and Operators
    - XML Schema applied and accessible
  - Related to XQuery 1.0
- XPath 3.0
  - Several W3C Recommendations, IV 2014

XPath and XQuery
Querying XML documents
- Common properties
  - Expression languages designed to query XML documents
  - Convenient access to document nodes
  - Intuitive syntax analogous to filesystem paths
  - Comparison and arithmetic operators, functions, etc.
- XPath
  - Standalone standard
  - Extension of XPath
  - Main applications:
    - XML data access and processing
    - XML databases
- XQuery
  - Used within other standards:
    - XSLT
    - XML Schema
    - XPointer
    - DOM
  - Several W3C Recommendations, IV 2014

XQuery Data Model
- Theoretical base of XPath, XSLT, and XQuery
- XML document tree
- Structures and simple data types
- Basic operations (type conversions etc.)
- Model different in different versions of XPath
  - 1.0 – 4 value types, sets of nodes
  - 2.0 & 3.0 – XML Schema types, sequences of nodes and other values

Models of XML processing
- Text level processing
  - possible but inconvenient and error-prone
- Custom applications using standardised API
  - DOM, SAX, JAXP, etc.
  - requires some work
- XML-related standards with high-level view on documents
  - XPath, XQuery, XSLT
  - XML-oriented and (usually) more convenient than above
  - sometimes not flexible enough
- “Off the shelf” tools and solutions

Version numbering
- Subsequent generations of related standards.

<table>
<thead>
<tr>
<th></th>
<th>XPath</th>
<th>XSLT</th>
<th>XQuery</th>
</tr>
</thead>
<tbody>
<tr>
<td>1999</td>
<td>1.0</td>
<td>1.0</td>
<td>-</td>
</tr>
<tr>
<td>2007</td>
<td>2.0</td>
<td>2.0</td>
<td>1.0</td>
</tr>
<tr>
<td>2014</td>
<td>3.0</td>
<td>3.0</td>
<td>(WD)</td>
</tr>
</tbody>
</table>

Paths - typical XPath application
- /company/department/person
- //person
- /company/department[name = 'accountancy']
- /company/department[@id = 'D07']/person[3]
- /surname
- surname
- ../person[position = 'manager']/surname

But there is much more to learn here...

Document tree – example
**XPath node kinds**
- Seven kinds of nodes:
  - document node (root)
  - element
  - attribute
  - text node
  - processing instruction
  - comment
  - namespace node
- Missing ones (e.g. when compared to DOM):
  - CDATA
  - entity
  - entity reference

**Data model in XPath 1.0**
- Four types:
  - boolean
  - string
  - number
  - node set
- No collections of simple values
- Sets (and not sequences) of nodes

**Effective Boolean Value**
- Treating any value as boolean
- Motivation: convenience in condition writing, e.g. if (customer[@passport]) then
- Conversion rules
  - empty sequence → false
  - sequence starting with a node → true
  - single boolean value → that value
  - single empty string → false
  - single non-empty string → true
  - single number equal to 0 or NaN → false
  - other single number → true
  - other value → error

**Type system**

**Sequences**
- Values in XPath 2.0 - sequences
- Sequence consists of zero or more items
  - nodes
  - atomic values
- Sequences properties
  - Items order and number of occurrence meaningful
  - Singleton sequence equivalent to its item 3.14 = (3.14)
- Nested sequences implicitly flattened to canonical representation:
  - (3.14, (1, 2, 3), 'A[a]' = (3.14, 1, 2, 3, 'A[a]')

**Atomization**
- Treating any sequence as a sequence of atomic values
  - often with an intention to get a singleton sequence
- Motivation: comparison, arithmetic, type casting
- Conversion rules (for each item)
  - atomic value → that value
  - node of declared atomic type → node value
  - node of list type → sequence of list elements
  - node of unknown simple type or one of xs:untypedAtomic, xs:anySimpleType
  - node with mixed content → text content as single item
  - node with element content → error

**Functions**
- Function invocation:
  - concat('Mrs ', name, ' ', surname)
  - count(/person)
  - my:factorial(12)
- 150 built-in functions in XPath 2.0, 27 in XPath 1.0
- Abilities to define custom functions
  - XQuery
  - XSLT 2.0
  - execution environment
  - EXSLT - de-facto standard of additional XPath functions and extension mechanism for XSLT 1.0

**Type casting**
- Type constructors
  - xs:date("2010-08-25")
  - xs:float("NaN")
  - adresy:kod-pocztyw("48-200") (schema aware processing)
  - string://obiekt[4]) (valid in XPath 1.0 too)
- Cast operator
  - "2010-08-25" cast as xs:date

**Literals and variables**
- Literals
  - strings:
    - "12.5"
    - "He said, "I don't like it.""
  - numbers:
    - 12
    - 12.5
    - 1.13e-8
- Variables
  - $x$ – reference to variable x
  - Variables introduced with:
    - XPath 2.0 constructs (for, some, every)
    - XQuery (FLWOR, some, every, function parameters)
    - XSLT 1.0 and 2.0 (variable, param)
**Chosen built-in XPath functions**

- Text:
  - `concat(s1, s2, ...)` substring(s, pos, len)
  - `starts-with(s1, s2)` contains(s1, s2)
  - `string-length(s)` translate(s, t1, t2)

- Numbers:
  - `floor(x)` `ceiling(x)` `round(x)`

- Nodes:
  - `name(n)` `local-name(n)` `namespace-uri(n)`

- Sequences (some only since XPath 2.0):
  - `count(S)` `sum(S)` `min(S)` `max(S)` `avg(S)`
  - `empty(S)` `reverse(S)` `distinct-values(S)`

- Context:
  - `current()` `position()` `last()`

**Operators**

- Arithmetic
  - `+` `-` `*` `div` `idiv` `mod`
  - `+` `-` also on date/time and duration

- Logical values
  - `and` `or`
  - `true()`, `false()`, and `not()` are functions

- Node sets / sequences
  - `union` `intersect` `except` `not` nodes found - type error
  - `result without repeats, document order preserved`

- Nodes
  - `is` `<>` `>`

**Comparison operators**

- Atomic comparison (XPath 2.0 only)
  - `eq` `ne` `lt` `le` `gt` `ge`
  - applied to singletons

- General comparison (XPath 1.0 and 2.0)
  - `=` `!=` `<` `<=` `>` `>=`
  - applied to sequences

- XPath 2.0 semantics:
  - There exists a pair of items, one from each argument sequence, for which the corresponding atomic comparison holds. (Argument sequences atomized on entry.)

**General comparison – nonobvious behaviour**

- Equality operator does not check the real equality
  - `(1,2) != (1,2) \rightarrow true`
  - `(1,2) = (2,3) \rightarrow true`

- “Equality” is not transitive
  - `(1,2) = (2,3) \rightarrow true`
  - `(2,3) = (3,4) \rightarrow true`
  - `(1,2) = (3,4) \rightarrow false`

- Inequality is not negation of equality
  - `(1,2) != (1,2) \rightarrow true`
  - `(1,2) != (1,2) \rightarrow true`
  - `(1) != () \rightarrow false`
  - `() != () \rightarrow false`

**Sequence quantifiers (XPath 2.0)**

- Some `$VAR` in `SEQUENCE` satisfies `CONDITION`
- Every `$VAR` in `SEQUENCE` satisfies `CONDITION`

**Conditional expression (XPath 2.0)**

- if `(CONDITION)`
  - then `RESULT1` else `RESULT2`

- Using Effective Boolean Value of `CONDITION`

- One branch evaluated

**Paths – more formally**

- Absolute path: `/step/step ...`
- Relative path: `step/step ...`
- Step – full syntax:
  - `axis::node-set [predicate1] [predicate2] ...`
  - `axis` – direction in document tree
  - `node-test` – selecting nodes by kind, name, or type
  - `predicates` – (0 or more) additional logical conditions for filtering

**Iteration through sequence (XPath 2.0)**

- for `$VAR` in `SEQUENCE`
  - return `RESULT`

- `VAR` takes subsequent values from `SEQUENCE`
- `RESULT` computed that many times
  - in context where `VAR` is assigned the given value
  - overall result = (flattened) sequence of partial results

**Axis**

- `self`
- `child`
- `descendant`
- `parent`
- `ancestor`
- `following-sibling`
- `preceding-sibling`
- `following`
- `preceding`
- `attribute`
- `namespace`
- `descendant-or-self`
- `ancestor-or-self`

**Typical usage**

- `books/price > 100`
  - At least one of the books has price greater than 100
Axis

Node test

By kind of node:
- node()
- text()
- comment()
- processing-instruction()

By name (examples):
- person
- pre:person
- pre:*  (XPath 2.0 only)
- *

kind of node here: element or attribute, depending on axis

Predicates

- Evaluated for each node selected so far (node becomes the context node)
- Every predicate filters result sequence
- Depending on result type:
  - number – compared to item position (counted from 1)
  - not number – Effective Boolean Value used
- “Filter expressions” – predicates outside paths

Examples
- /child::staff/child::person[child::name = 'Patryk']
- child::person[child::name = 'Patryk']/child::surname
- /person[attribute::passport[3]]
- [1 to 10]. mod 2 = 0

Abbreviated Syntax

- child axis may be omitted
- @ before name indicates attribute axis
- . stands for self::node()
- .. stands for parent::node()

// translated to /descendant-or-self::node/
(textually, inside an expression)

Example
- ../../../object[ gid = 'E4' ]

expands to
- self::node()[/descendant-or-self::node[
- child::object[attribute::id = 'E4']]}

Node test in XPath 2.0

In XPath 2.0 more tests, basing on kinds of nodes, and schema-provided types of nodes (“schema aware” only). Examples:
- document-node()
- processing-instruction(xmlstylesheet)
- element()
- element(person)
- element(*, personType)
- element(person, personType)
- attribute()
- attribute(id)
- attribute(*, xs:integer)
- attribute(id, xs:integer)

Evaluation order

- From left to right
- Step by step (predicate applied to the last step)
  - //department/person[1]
  - (/department/person)[1]
- Predicate by predicate
  - //person[@manages and position() = 5]
  - //person[@manages][position() = 5]
XQuery, XML and databases

Patryk Czarnik
XML and Applications 2014/2015

Example – XML support in Oracle database
http://www.oracle.com/xml
- Since Oracle 8i
  - details differ from version to version
- XML parsers
  - for database programming (PL/SQL)
  - or middleware programming (Java, C++)
- XML-SQL Utility
  - XML data import and export
- XMLType data type and XML-specific operations

XML support in databases – categorisation
- Classic (usually relational) database with XML support
- additional XML-related features
- used for application integration or storing XML data as part of larger data structures
- `Native` XML database
- logical structure – collection of XML document trees
- XPath (or XPath) as native query language
- natural XML-related features
- used for storing XML data (or structural data easily mapped to XML tree)

XML-SQL Utility
- getXML() function – XML data export

```
SELECT xmlgen.getXML('select * from emp') FROM dual;
```

XML support in database engines
  - only well-formed XML documents allowed
  - parsing and serialisation
  - implementation may add XML-specific operations
- Substantial support
  - IBM DB2 (since v.9 – pureXML)
  - Oracle (since 8i)
  - Microsoft SQL Server (since v.2000)
  - Sybase ASE (since v.12)
- Minimal support
  - MySQL - XPath queries over text fields containing XML
  - PostgreSQL – as above plus XML datatype but with no special operations

XML in Oracle DB – XMLType
- XMLType – special datatype:
  - to be stored as LOB or used for columns, variables, etc.
  - indexing XML content
  - XPath expressions
  - validation against XML Schema
  - XSLT
- Available functions:
  - extract, extractValue, existsNode, transform, updateXML, XMLSequence

XMLType applications – some examples
CREATE TABLE warehouses(
  warehouse_id NUMBER(4),
  warehouse_spec XMLType,
  warehouse_name VARCHAR2(35),
  location_id NUMBER(4));

UPDATE po_xml_tab
SET poDoc = UPDATEXML(poDoc,
  '/PO/CUSTNAME/text()', 'John');

INSERT INTO warehouses VALUES (100, XMLType:
  '<Warehouse wno="100">'
  '  <Building.Owner>Building</Building>
  </Warehouse>'), 'Tower Records', 1003);

SELECT e.poDoc.getTextNodeVal() AS poXML
  FROM po_xml_tab e
  WHERE e.poDoc.existsNode('/PO/NAME = "po_2"') = 1;

CREATE INDEX city_index ON po_xml_tab
  (poDoc.extract('/PO/NAME/text()').getNumberVal());

Source: Oracle DB documentation

Native XML database
- Logical layer
  - XML document as basic data entity
  - collections of documents build a database
  - XML schema (or equivalent) as structure definition
  - XQuery (or XPath) as “native” query language
- Physical layer – not necessarily “files with XML text”
  - More than just a collection XML files:
    - transactions and concurrent access
    - security (access privileges etc.), versioning, replication, ...
    - API for data access and update
    - additional means of data access
    - e.g. REST-compliant HTTP server
    - indexing for efficient access to selected nodes
Standards for XML databases

- High level query languages:
  - XQuery – primary language for queries
  - versions 1.0 and 3.0 in use
  - XQL – former approach to make XML query language
  - XPath – poor stub for XQuery
- High level update languages:
  - XQuery Update Extension
  - XUpdate
- Programmer APIs
  - depend additionally on programming language
  - XML Database API (XAPI)
  - XQ (for Java, expected to become XML equivalent of JDBC)
  - vendor-specific APIs...

XQuery – the query language for XML

- Status
  - XQuery 1.0 – W3C Recommendation, 1 2007
  - XQuery 3.0 – W3C Recommendation, IV 2014
  - Data model, functions and ops – shared with XPath 2.0 / 3.0
  - Syntax defined in a separate document
    - In practice: extension of XPath
- Main features:
  - Picking up data from XML documents
  - Constructing new result nodes
  - Sorting, grouping
  - Defining custom functions
  - Various output methods (XML, HTML, XHTML, text)
  - shared with XSLT

XQuery – query structure

- (Unexpectedly) XQuery is not an XML application
  - There exists a verbose XML syntax for XQuery, not intended to be written by hand
- Typical file extensions: .xqery, .xq, .xqm (for modules)
- Text format, header and body
  - header optional in normal queries
  - units declared as modules do not have body

XQuery headers

- Header part consists of declarations:
  - version declaration
  - import
  - flags and options
  - namespace declaration
  - global variable or query parameter
  - function

Example

```xml
<xquery version="1.0" encoding="utf-8">
  declare namespace foo = "http://example.org";
  declare variable $id as xs:string external;
  declare variable $doc := doc("example.xml");
  $doc/foo/object[@id = $id]
</xquery>
```

FLWOR expression

- For, Let, Where, Order by, Return
  - More clauses in XQuery 3.0, but the acronym remains.
- Replaces for from XPath
- Obvious influence of SQL SELECT

Example

```xml
for $obj in doc("example.xml")/list/object
let $prev := $obj/preceding-sibling:element()
let $prev-name := $prev[1]/name
where $obj/name
order by $id/$name
return
<di class="result">
  Object named (xs:string($obj/name))
  has count($prev) predecessors.
  The nearest predecessor name is
  (xs:string($prev-name)).
</di>
```

Node constructors – direct

- XML document fragment within query

Example

```xml
for $el in doc("example.xml")/"/* return
  <p style="color: blue">I have found an element...
  <a href="#">Hia Hia Hia</a>
</p>
```

Expressions nested within constructors – braces

Example

```xml
for $el in doc("example.xml")/"/* return
  <elem depth="count($el/ancestor::node())">
    (name($el))
  </elem>
</result>
```

Node constructors – computed

- The same example again to show the syntax

Example

```xml
for $el in doc("example.xml")/* return
  element (concat("elem-", name($el)))
  attribute depth (count($el/ancestor::node())),
  text (name($el))
</result>
```

Custom function definitions

- Simple example:

Example

```xml
declare function local::factorial($n)
  {$n * local::factorial($n - 1)}
```

Example using type declarations:

Example

```xml
declare function local::factorial($n as xs:integer)
  as xs:integer{
    $n * local::factorial($n - 1)
  }
```

Type constraints

- Type declarations possible (but not obligatory) for:
  - variables
  - function arguments and result
  - also in XSLT 2.0 (variables and parameters)
- Dynamic typing used in practical applications
  - 13 + if (aCondition) then ‘not a valid number’ else 1
  - may fail or not depending on input data
  - some $x in (1+1, xs:dateTime(‘long long time ago’))
  - satisfies $x=2 fails or not depending on the processor
- Static typing discussed, but rarely deployed
  (“academic” solutions, for XQuery rather than XSLT)
**Type declarations**

- **Capabilities:**
  - type name
    - built-in - always available
    - user-defined - schema aware processors only
  - kind of node | node() | item()
  - occurrence modifier (?, +, *, exactly one occurrence by default).
- **Examples:**
  - `xs:double`
  - element()
  - node()
  - `xs:integer`?
  - item()+

**XQuery 3.0 - selected improvements**

**XML database products - overview**

<table>
<thead>
<tr>
<th>Product</th>
<th>Licence</th>
<th>Queries</th>
<th>XML:DB API</th>
</tr>
</thead>
<tbody>
<tr>
<td>eXist</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>BaseX</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>MarkLogic</td>
<td>commercial</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Xindice</td>
<td>open source</td>
<td>XPath</td>
<td>yes</td>
</tr>
<tr>
<td>Sedna</td>
<td>open source</td>
<td>XPath, XQuery</td>
<td>yes</td>
</tr>
<tr>
<td>Gemfire Enterprise</td>
<td>commercial</td>
<td>XQuery, OQL</td>
<td>yes</td>
</tr>
<tr>
<td>Tamino</td>
<td>commercial</td>
<td>XQuery, XPath</td>
<td>part</td>
</tr>
</tbody>
</table>

Source: Wikipedia and providers' websites

In addition:
- Saxon - just a query processing engine, works on files (or other XML sources accessible in Java).

**eXist - eXide**

- **XQuery programmer SDK running within a browser**
  - supports also (to some extent...)
  - XSLT, XML Schema, XHTML, XForms

**eXist - template mechanism**

- Easy integration of XQuery logic and HTML interface

**XML:DB**

- Initiative for XML database interfaces specification
  - XML Database API (XAPI)
  - accessing XML databases from programs
  - resource collections (resource = XML document)
  - reading and writing documents via DOM or SAX
  - pluggable "services": specified: XPath, transactions, operations on collections
  - last version: 2001
- XML Update Language (XUpdate)
  - XML application (format) for updating XML databases
  - inserting, updating and removing nodes
  - XPath used for node addressing
  - last version: 2000

**eXist DB**

- One the most popular and elaborated XML database engines
- Open-source, but developed and supported by a (German) company; commercial support available
- Features include:
  - storage of XML and binary entities
  - various means of access, including: human-readable Web interface, direct HTTP access (REST-compliant), SOAP and XML-RPC, Java API (XQ), elements of XAPI
  - full XML model available in XPath, XQuery, and XSLT code
  - full XQuery support with majority of new 3.0 features, Update extension and some other non-standard extensions
  - XForms support using betterFORM or XSLTForms plugins
  - extensible with custom Java code
**XSLT**

XSLT is an Extensible Stylesheet Language. It is a transformation language for XML documents. XSLT can be used for a variety of purposes, including:

- Converting XML data into other formats (e.g., HTML, CSV)
- Creating dynamic interfaces and forms
- Displaying records from databases
- Generating reports and other documents
- Creating personalized output for users

**XSLT - availability of tools**

- XSLT 1.0 processors:
  - Internet browsers (IE, Mozilla/Firefox, Opera, Chrome)
  - Apache Xalan (for Java and C++)
  - xsltproc (Linux and related OSs)
  - XML extensions of database engines
  - DTP tools

- XSLT 2.0 processors:
  - Saxon (for Java and .NET)
    - basic version free (Open Source)
    - full (schema aware) version paid
  - commercial tools: XML Spy, oXygen

- Authoring tools:
  - Hundreds of plain text editors (with syntax highlighting etc.)
  - Advanced programmer environments (Eclipse, IDEA, ...)
  - Commercial XML-specialised tools (XML Spy, oXygen, ...)

**XSLT - where does it come from?**

- XSL - Extensible StyleSheet Language
  - Presentation of XML documents by transformation
- XSLT - XSL Transformations
  - Language (XML) to define transformations
  - Transformation of a source XML document tree to a result tree
  - Designed as one of XSL components
  - General enough to be used for other purposes

**XSLT - status**

- Version 1.0
  - October 1999
  - Makes use of XPath 1.0
  - Popular and widely supported by tools
- Version 2.0
  - January 2007
  - Makes use of XPath 2.0, related to XQuery 1.0
  - More general (and specified in more consistent way) data model
  - More features
  - Less popular, little (but existing) support
- Version 3.0
  - Work in progress, almost finished; motivated mainly by development of XQuery 3.0

**Structure of XSLT stylesheet**

- Stylesheet (arkusz) consists of templates
  - Template (szablon) – building block of XSLT
  - transformation of a single source node to result tree fragment
  - may be called many times for different nodes
- Within template:
  - text and elements out of XSLT namespace – copied to result
  - XSLT instructions – control flow, copying content from source, inserting computed content, and more features (in 2.0 even more)
  - XPath expressions in some instructions – accessing source document, checking conditions, calculating numbers, etc.
- XSLT can be seen as a programming language
  - with a bit non-standard syntax
  - convenient for processing and creating XML content

**Structure of stylesheet – example (1)**

```xml
<xml version="1.0" encoding="utf-8"?>
<xml:stylesheet version="1.0" xmlns:xml="http://www.w3.org/1999/XSL/Transform">
  <xsl:output method="html" encoding="utf-8"/>
  <xsl:template match="/"/>
  <html>
    <head>
      <title>Kindergarten pupils</title>
    </head>
    <body>
      <h1>Kindergarten pupils</h1>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

**Structure of stylesheet – example (2)**

```xml
<xsl:template match="kids">
  <xsl>xml:apply-templates />
</xsl:template>
<xsl:template match="girl">
  <li color="pink">
    <xsl:apply-templates />
  </li>
</xsl:template>
<xsl:template match="boy">
  <li color="blue">
    <xsl:apply-templates />
  </li>
</xsl:template>
</stylesheet>
```

**XSLT in run**

- Processing based on tree nature of XML documents
- Start: running template matching source document root (/)
  - such template exists even if not declared
- Other templates run when applied with apply-templates instruction
  - and so on, recursively
  - usually following shape of source document
  - matching nodes to patterns and selecting appropriate templates
  - template result pasted into result tree
- Other ways of flow control
  - call-template – invoking templates without pattern matching
  - for-each - iteration over sequences of nodes
  - if and choose – conditional processing

**Pattern matching and template selection**

- match attribute of template
  - pattern, restricted form of XPath expression
  - "What are the nodes this template applies to?"
- select attribute of apply-templates instruction
  - XPath expression
  - "Which nodes to process now?"
  - optional, children of current node selected if not given
- Matching
  - for each node to be processed, independently:
    - from all templates which patterns match the node
      - at least one always exists
      - select one with the highest priority
      - usually – one with the strictest pattern
      - in case of many templates with the same priority – conflict
        - error or latter template chosen, depending on implementation
Pattern matching and template selection – example 1

```xml
<xsl:template match="Kids">
  <ul>
    <li><xsl:apply-templates /></li>
  </ul>
</xsl:template>

<xsl:template match="girl">
  <li color="pink">Alice</li>
</xsl:template>

<xsl:template match="boy">
  <li color="blue">Bob</li>
</xsl:template>

<xsl:template match="Dorothy">
  <li color="pink">Dorothy</li>
</xsl:template>
```

Pattern matching and template selection – example 2

```xml
<xsl:template match="Cars">
  <ul>
    <li><xsl:apply-templates select="car" /></li>
  </ul>
</xsl:template>

<xsl:template match="car[max > 200]">
  <li><xsl:value-of select="name" /></li>
</xsl:template>

<xsl:template match="car">
  <li><xsl:value-of select="name" />/li>
</xsl:template>
```

Built-in templates

- Templates applied if no user-provided template matches node
- lowest priority
- for each node there exists a template matching the node
- For document root and elements:
  - apply templates to children
  - preserve current mode
  - passing down all parameters passed to the current template
  - does not process attributes
- For text nodes and attributes:
  - copy text value to result
- For comments and processing instructions:
  - do (and return) nothing

Processing modes

- mode attribute in template and apply-templates
- mode identified with an author-defined name
- in XSLT 2.0 additional keywords:
  #default, #any, #current
- Alternative templates for the same nodes
- Possible applications:
  - processing the same nodes differently during the same transformation, e.g. details and summary
  - preparing library of different templates in advance and using selected subset in particular transformation

Conditional processing – if

- Logical condition test checked (Effective Boolean Value)
- If true → body processed and result inserted into result tree
- No else

Conditional processing – choose

- Many branches
- Conditions (test in when) checked in order of occurrence
- (Only) first branch with satisfied condition processed
- Optional otherwise branch – used if no condition satisfied

Built-in templates

- Templates applied if no user-provided template matches node
- lowest priority
- for each node there exists a template matching the node
- For document root and elements:
  - apply templates to children
  - preserve current mode
  - passing down all parameters passed to the current template
  - does not process attributes
- For text nodes and attributes:
  - copy text value to result
- For comments and processing instructions:
  - do (and return) nothing

Processing modes – example

```xml
<xsl:template match="person">
  <li><xsl:apply-templates /></li>
</xsl:template>

<xsl:template match="name">
  <li><xsl:value-of select="name" /></li>
</xsl:template>

<xsl:template match="department">
  <table>
    <xsl:apply-templates select="person" mode="table" />
  </table>
</xsl:template>
```

Combining stylesheets

Stylesheet files can be combined using top-level elements:

- `<xsl:include>`
  - simple inclusion of external stylesheet elements into current one
- `<xsl:import>`
  - importing templates from external stylesheet with lower precedence
  - acceptable only at the beginning of stylesheet
  - current and imported stylesheets form a tree - the further a template is defined, the lower precedence it has

Iteration over sequence – for-each

- For each node from select-ed set
  - node becomes the “current node”
  - body processed, result inserted into result tree
- Alternative way of walking through source document tree, but:
  - no pattern matching, each node processed in the same way
  - no separation of concerns, one blob of XSLT code in extreme case

```xml
<xsl:template match="department">
  <ul>
    <xsl:for-each select="person">
      <li><xsl:value-of select="name" /></li>
    </xsl:for-each>
  </ul>
</xsl:template>
```
Sorting
- sort instruction ("declaration") in for-each, for-each-group and apply-templates
- Sorting options in attributes:
  - select - what values are compared - processed nodes by default
  - data-type, lang - data type (number | text) and natural language
  - order, case-order, stable

Inserting content into template body
- Content from template body copied to result:
  - text nodes, except whitespace-only text nodes (indention etc.)
  - elements, except elements in XSLT namespace
  - together with their attributes
  - elements content processed recursively
- Applies to any "sequence constructor" in general
  - content of block instructions (for-each, if, etc.)
  - function body (XSLT 2.0), parameter and variable body

Shallow copying
- copy instruction
  - Creating node "the same as" the current one
  - same kind
  - same name and namespace (if applicable)
  - Content of current element not copied
- Content of copy instruction processed as usual
  - may be used to copy / process source element content, if needed

Grouping (XSLT 2.0)
- for-each-group instruction
- current-group and current-grouping-key functions

Example from Recommendation
```xml
<xsl:for-each-group select="cities/city" group-by="country">
  <tr>
    <td><xsl:value-of select="position()"/></td>
    <td><xsl:value-of select="@country"/></td>
    <tr>
      <td><xsl:value-of select="current-group()'/name" separator=''/></td>
    </tr>
  </xsl:for-each-group>
</tr>
```

Constructor instructions
- Instructions creating result nodes:
  - element attribute text
  - comment processing-instruction

Example from Recommendation
```xml
<xsl:template match="person">
  <xsl:element name="if">
    <xsl:attribute name="position">manager</xsl:attribute>
    <xsl:attribute name="boss">xsl:attribute</xsl:if>
  </xsl:element>
  <xsl:variable name="title" select="text()"/>
  <xsl:variable name="date" select="modified-date"/>
  <xsl:variable name="name" select="current-group()'/name" separator=''/>
  <xsl:processing-instruction target="xsi:stylesheet" type="text/css">
    href="/styles.css"
  </xsl:processing-instruction>
</xsl:template>
```

Inserting XPath expression result
- copy-of and value-of instructions
  - in XSLT 2.0 also sequence
- XPath expression given in select attribute
- Expression result inserted into transformation result, depending on instruction, as:
  - copy-of - deep copy of result
  - value-of - text node containing text representation of result
  - sequence - unprocessed sequence (matters only theoretically)

Examples
```xml
<xsl:copy-of select="/person[position()='chief']"/>
<xsl:value-of select="/person[position()='chief']/surname"/>
<xsl:value-of select="salary + sales * commision div 100"/>
```

Creating result tree
- Inserting result content into template body
  - convenient (hence "templates")
- Creating nodes by constructor instructions
  - general
- Copying nodes from source document or computing values using XPath expressions
  - expressible

Typical applications of constructor instructions
- Inserting processing instructions and comments
- Inserting sole whitespace characters
- Inserting text fragments without unwanted whitespace characters
- Element or attribute name established at runtime
- Inserting attribute conditionally

value-of in XSLT 1.0 – details
- If expression evaluates to simple value – that value printed
- If expression evaluates to node set – only first node from set cast to string and printed
- As slightly unintuitive – often leads to errors
value-of in XSLT 2.0 - details

- Expression evaluates to sequence (as always in XPath 2.0)
- Sequence flattened and atomized
- Result text:
  - string representations of all sequence items
  - separated with text provided in separator attribute
  - default separator - single space
- Incompatible with 1.0 behaviour

```xml
<person fn="Patryk" sn="Czarnecki"/>
<person fn="Radek" sn="Bartosiak"/>
<person fn="PatrykCzarneckiRadekBartosiak"/>
</result>
```

value-of etc. - best practices

- Instruction with `select` - all processing XPath result somehow...
- Which one to use - depending on what is processed

value-of
- Value of simple type (number, string, date/time), in particular when computed by an expression or function
- Value of single text node / simple-type element / attribute
- Elements? - only when we want to flatten them to plain text
  - fully aware of uninitiative behaviour in XSLT 1.0
  - and incompatibilities between 1.0 and 2.0
- Do not make your default manner of the last one, it is heavily overused in practice

apply-templates

- Source document nodes
- using apply-templates we enable them to be processed
- by default text content printed
- built-in templates used
- similar to value-of, but more intuitive (all nodes printed)
- templates - allow us to impact the treatment of nodes

```xml
<xsl:template match="*">
  <value-of select="."/>
</xsl:template>
```

copy-of

- Nodes copied directly from source
- Nodes created in advance, e.g. variable values
- Worth to use for variables/parameters even if they are text-only now (value-of would also work), preparing ourselves to enrich their structure in the future (then value-of would not work any more)

Global variables and parameters

- Parameter value set from outside...
- execution environment, command line, API call
- Variables computed once per transformation

```xml
<xsl:variable name="elements-count" select="count(/element[])">
  <xsl:variable name="message">
    The document contains <xsl:value-of select="elements-count"/>
  </xsl:variable>
</xsl:variable>
```

Inserting XPath result into attribute

- "Attribute value template"
- Given within braces
- Result of XPath expressions processed in the same way as with value-of

```xml
<xsl:element name="b" select="($count/ancestor-or-self::section)"><xsl:value-of select="."/></xsl:element>
```

Local variables

- variable instruction
- "Declarative" approach - variables cannot change their values
- They may assume different values in subsequent invocations
- Analogous to let in functional programming or final variables in Java

```xml
<xsl:template match="account">
  <xsl:variable name="sign">
    <xsl:when test="@deb = 0">positive</xsl:when>
    <xsl:when test="@deb &lt; 0">negative</xsl:when>
  </xsl:choose>
  <xsl:element name="$sign"/>
</xsl:variable>
```

Template parameters

- Declaration in template: param
- Value specification on invocation: with-param

```xml
<xsl:template match="department">
  <xsl:apply-templates select="person">
    <xsl:with-param name="prefix">Employee</xsl:with-param>
  </xsl:apply-templates>
</xsl:template>
```

Consequences of declarative approach

- name attribute of template
- call-template invokes
- Recursion permitted - high power of expression

```xml
<xsl:template match="descendant-or-self::*">
  <xsl:call-template name="descendant-element"/>
</xsl:template>
```

Named templates

```xml
<xsl:stylesheet version="2.0" xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template match="/">
    <html>
      <head><title>My Document</title></head>
      <body>
        <xsl:for-each select="/">
          <xsl:call-template name="descendant-element"/>
        </xsl:for-each>
      </body>
    </html>
  </xsl:template>
</xsl:stylesheet>
```
Recursion “programming” in XSLT (even 1.0)

- Enable “programming” in XSLT (even 1.0)

```
<xsl:template name="factorial">
  <xsl:param name="n"/>
  <xsl:param name="res" select="1"/>
  <xsl:choose>
    <xsl:when test="$n = 1" />
    <xsl:call-template name="factorial">
      <xsl:with-param name="m" select="$n - 1"/>
      <xsl:with-param name="res" select="$n * $res"/>
    </xsl:call-template>
  </xsl:choose>
  <xsl:otherwise>
    <xsl:call-template name="factorial">
      <xsl:with-param name="m" select="$n - 1"/>
      <xsl:with-param name="res" select="$n * $res"/>
    </xsl:call-template>
  </xsl:otherwise>
</xsl:template>
```

Serialisation parameters

- method (xml | html | text | xhtml)
- encoding – character encoding
- indent (yes | no) – whether to automatically format output
- version – version of XML or HTML
- doctype - public, doctype - system - DOCTYPE declaration

Example: XHTML 1.0 declaration in XSLT 1.0

```
<xsl:output method="xsl" version="1.0" encoding="utf-8" doctype-public="-//W3C//DTD XHTML 1.1//EN" doctype-system="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

Beginning of result document

```
<html xmlns="http://www.w3.org/1999/xhtml"
      xml:lang="en"
      xml:stylesheet href="http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd"/>
```

Some typical patterns

- Recommended structure of stylesheet depends on application
- Typical patterns and best practices for
  - text document presentation
  - data processing and presentation
  - copying and filtering

Custom function recursive definitions (XSLT 2.0)

- Factorial - naive recursive definition

```
<xsl:template name="factorial">
  <xsl:param name="n"/>
  <xsl:choose>
    <xsl:when test="$n = 1" />
    <xsl:call-template name="factorial">
      <xsl:with-param name="m" select="$n - 1"/>
      <xsl:with-param name="res" select="$n * $res"/>
    </xsl:call-template>
  </xsl:choose>
  <xsl:otherwise>
    <xsl:call-template name="factorial">
      <xsl:with-param name="m" select="$n - 1"/>
      <xsl:with-param name="res" select="$n * $res"/>
    </xsl:call-template>
  </xsl:otherwise>
</xsl:template>
```

Place of use

- <xsl:value-of select="loc:factorial(5)"/>

Result of transformation - common applications

- XSL Formatting Objects:
  - according to original XSL approach
  - especially useful for printed publication
- HTML or XHTML
  - most popular
  - especially useful for Web publication
- Arbitrary XML, e.g.:
  - migration to new / other format
  - acquiring and processing data (alternative to XQuery)
  - generating XSLT as result of XSLT
- Plain text, e.g.:
  - CSV and other text data formats
  - scripts and configuration files
  - converting text documents to non-XML text formats

Text document presentation

- (this subject is going to be extended in the next lecture)
- Processing driven by source document structure (push)
- Many simple templates
  - one template for one type of input elements
  - translating source element into target element and processing its content with apply-templates
- More elaborated templates for elements requiring special treatment...
- Shape of result tree based on shape of source tree
- Works well with mixed content typical to text documents

Serialisation of transformation result

- Transformation takes place at level of document tree

Serialisation – converting result tree into sequence of bytes

- Several formats ("methods") supported:
  - xml
    - ordinary XML format, “general parsed entity”
    - proper XML document only if exactly one top-level element exists
  - html
    - uses only starting tags for empty elements (<br> rather than<br/>)
    - automatically adds Content-Type metadata to HTML head tag
  - text
    - omits all tags and print only content of result text nodes
  - xhtml (XSLT 2.0)
    - much like xml, but allows transformers to control result structure

Temporary result tree fragments

- XSLT 1.0 – distinct types node set and result tree fragment
  - not allowed to mix
  - not allowed to process result tree fragment again
- XSLT 2.0 – no such restriction
  - In particular: result (e.g. of value-of or apply-templates) can be read and processed again
  - this is the real programming and full recursion!

XSLT 2.0 but not XSLT 1.0

```
<xsl:variable name="tmp">
  <xsl:apply-templates select="document"/>
</xsl:variable>
<xsl:apply-templates select="step" mode="correct"/>
```

Text document presentation

```
<html>
  <head>
    ...</head>
  <body>
    ...</body>
</html>
```

Typical simple template

```
<xsl:template match="item"/>
```

```
<body>
  <div>
    <xsl:apply-templates/>
  </div>
</body>
```

More elaborated template

```
<xsl:template match="person">
  <div class="person">
    <xsl:apply-templates select="fname"/>
  </div>
  <xsl:apply-templates select="surname"/>
  <br/>
  <xsl:apply-templates select="contact"/>
</div>
```
### Data processing and presentation

- Processing driven by intended result structure (pull)
- One (or a few) large template
  - fixes structure of result
  - inserts acquired / computed data into intended places using value-of
  - iterates over monomorphic sequences using for-each
- Complex XPath expressions
  - or even "XSLT programming" to compute more complex things

### Copying and filtering

#### Big step approach

**Coarse-grained filtering template**

```xml
<xsl:template match="/company">
  <company>
    <xsl:copy-of select="name"/>
    <xsl:for-each select="department">
      <dt><xsl:value-of select="name" /></dt>
      <dd><xsl:value-of select="12*sum(person/salary)" /></dd>
    </xsl:for-each>
  </company>
</xsl:template>
```

#### Small step approach

**Standard copy template**

```xml
<xsl:template match="/*[node()]">
  <xsl:copy>
    <xsl:copy-of select=".0"/>
    <xsl:apply-templates select="."/>
  </xsl:copy>
</xsl:template>
```

**Simple template changing element name**

```xml
<xsl:template match="collection/item">
  <collection-item>
    <xsl:apply-templates select="."/>
  </collection-item>
</xsl:template>
```

**"Advanced" template**

```xml
<xsl:template match="para" name="string-length">
  <p><xsl:value-of select="string-length($string)" /></p>
</xsl:template>
```

### Filtering and copying

- Returning subset of source document

Two styles, depending on needs:

- Big step style (analogous to target-driven processing)
  - large fragments of input copied as whole
  - copy-of within big template

- Small step style (analogous to source-driven processing)
  - recursive processing with exclusion or specialised transformation of selected nodes
  - copy as default and specialized templates for elements to be processed
Presentation of XML

Patryk Czarnik
XML and Applications 2014/2015
Lecture 11 – 12.01.2015

Benefits of content and formatting separation
- With semantic tagging – source data analysis easier and more reliable (than reverse-engineering of formatted text)
- Ability to easily present:
  - the same document after modifications
  - other documents from the same class
- Changes in formatting applied easily
- Modifications in one place – the stylesheet
- Whole class of documents formatted consistently
- Alternative styles for the same class of documents, depending on
  - media type (screen, printout, voice)
  - details level
  - reader preferences (or disabilities...)

Separation of content and formatting
- According to best XML practices:
  - Documents consist of content / data.
  - Tags are for structure and meaning (semantic tagging).
  - e.g. <count>2, 99</count> rather than <i>2, 99</i>
- There is no direct formatting information.
- How to present documents?
  - Generic (and poor) XML presentation methods
  - XML source
  - document tree
  - unformatted text content
- Custom application handling a particular known class of documents
- Importing XML to text editors or DTP tools
- External style sheets

Standards related to XML presentation
- Assigning style to document:
  - Associating Style Sheets with XML documents
- Stylesheet languages:
  - DSSSL (historical, used for SGML)
  - Document Style Semantics and Specification Language
  - CSS
  - Cascading Style Sheets
  - XSL
  - Extensible Stylesheet Language

Applications of CSS
- First and major one: style for Web sites
- Separation of content and style for HTML
- (Simple) style sheets for XML
- CSS 2 and the idea of “accessibility”:
  - support for different media
  - support for alternative presentation means (e.g. voice generation)
  - enabling reader to override style proposed by author (reader rules)

Example stylesheet (fragment)

```xml
<?xml version="1.0" encoding="utf-8"?>
<xml-stylesheet type="text/css" href="blue.css"/>
<xml-stylesheet title="Yellow" alternate="yes" type="text/css" href="yellow.css"/>
<xml-stylesheet title="Pink" alternate="yes" type="text/css" href="pink.css"/>
```

Idea of stylesheet

```xml
<people position="expert" id="102105">
  <name>David Paskiewicz</name>
  <phone type="office">+48223120203</phone>
  <phone type="home">+48801502505</phone>
  <email>paskiewicz@example.com</email>
</people>
```

Associating style with document

```xml
<person>
  <name>Irene Paskiewicz</name>
  <phone type="office">+48801502505</phone>
  <email>irene@example.com</email>
</person>
```
Example stylesheet (fragment)

```css
.person {
  display: block;
  margin: 10px auto 10px 30px;
  padding: 0.75em 1em;
  width: 200px;
  border: solid 2px #000000;
  background-color: #FFFFFF;
}
.person[position='chief'] {
  background-color: #00FF00;
}
.fname, .surname {
  display: inline;
  font-size: larger;
}
.person[position='chief'] .surname {
  font-weight: bold;
}
```

CSS capabilities and advantages

- Rich visual formatting features
- Selecting elements by:
  - name
  - location in document tree
  - attribute existence
  - attribute values
- Good support:
  - internet browsers
  - authoring tools
- Easy to write simple stylesheets :)

Extensible Stylesheet Language (XSL)

- Defined in W3C recommendations (v1.0 in 1999 and 2001):
  - XSLF - general framework and XSL Formatting Objects
  - XSLT - language for defining XML transformations
  - XPath - expression language, including paths for document fragments addressing

Original approach:
- Transformation definition (XSLT), in the role of stylesheet, specifies how a source document is translated into FO document.
- Presentation of result FO is specified by XSL-FO standard and available through a rendering engine.

Practice:
- HTML result format used more often (although FO also used)
- XSLT and XPath also used for purposes other than presentation

Resulting visualisation

Extremely professional staff

Accountancy

David Puczkowski
id: 482153293
mab: 480650059
puczkowski@example.com

Monika Domińska-Woźna
id: 482153293
mab: 480650059
puczkowskiew@examplo.com

CSS shortcomings

- Only visualisation, not translation to different formats
- Selectors relatively weak. Conditions not expressible in CSS:
  - checking content of element, e.g.:
    - element A that contains element B
    - element A that contains text abc
  - logical composition of many conditions
    (available to some extent, but inconvenient)
  - value comparison (e.g. show negative amounts in red)
- Structure of blocks directly based on structure of source elt
- Reordering of elements hard (and not possible in general way)
- Not possible to show one element several times on page
- No data processing. Not available for example:
  - number calculations (summing etc.)
  - operations on text (shortening, regexp matching, etc.)

CSS selectors (representative examples)

- `surname` - element of the given name
- `fname`, `surname` - both elements
- `company`, `name` - name being descendant of company
- `company` > `name` - name being direct child of company
- `surname` + `phone` - phone directly succeeding surname
- `phone: first-child` - phone being first child of its parent
- `person[position]` - person owning position attribute
- `person[position='manager']` - person with position attribute equal to manager
- `person [roles='manager']` - person with attribute role containing word manager (attribute as space-separated list)
- `ol. staff` - equivalent to `ol{class='staff'}` (HTML only)
- `person#k12` - person with ID (in DTD meaning) equal to `k12`

Presentation by transformation

Transformation to HTML - example (1)

`<xsl:template match="person">`<br>`<xsl:if test="@position='manager'">manager</xsl:if>"`<br>`</xsl:template>`

```
<example style="color: red">Good support
internet browsers
authoring tools
</example>
```

Transformation to HTML - example (2)

```
<example style="color: blue">Easy to write simple stylesheets :)
</example>
```
**XSL-FO – basic facts**

- Presentation-oriented XML application
- Elements for different kinds of visual objects (block, inline, table, and so on)
- Attributes for formatting, based on CSS properties
- Especially useful for printed publications
- Focused on paged media type:
  - master pages (templates), page areas (header, footer, etc.)
  - automatic text flow and repeated (“static”) content
- Practice: intermediate format in
  - XML → XSL-FO → PDF transformation
- Not supported by web browsers
- Designed as part of XSL framework
  - result of XSL transformation
  - not intended to be used standalone

**Basic structure of XSL-FO document**

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
  <fo:layout-master-set>
    <fo:layout-master-reference master-name="my-page"/>
  </fo:layout-master-set>
  <fo:flow name="xsl-region-body">
    <fo:block>Hello World</fo:block>
  </fo:flow>
</fo:root>
```

**Formatting objects**

XSL-FO elements relate to resulting formatting objects.

- Block level
  - block
  - list-block, list-item, list-item-label
  - table, table-row, table-cell,...
- Inline level
  - inline, character
  - external-graphics
- Special features
  - basic-link, bookmark, marker
  - footnote
  - page-number

**List – example**

```
<fo:list-block>
  <fo:list-item>
    <fo:list-item-label>
      <fo:block>First name: </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>John</fo:block>
    </fo:list-item-body>
  </fo:list-item>
  <fo:list-item>
    <fo:list-item-label>
      <fo:block>Last name: </fo:block>
    </fo:list-item-label>
    <fo:list-item-body>
      <fo:block>Doe</fo:block>
    </fo:list-item-body>
  </fo:list-item>
</fo:list-block>
```

**Table – example**

```
<fo:table border="solid 2pt black">
  <fo:table-header>
    <fo:table-row>
      <fo:table-cell>First Name</fo:table-cell>
      <fo:table-cell>Last Name</fo:table-cell>
    </fo:table-row>
  </fo:table-header>
  <fo:table-body>
    <fo:table-row>
      <fo:table-cell>John</fo:table-cell>
      <fo:table-cell>Doe</fo:table-cell>
    </fo:table-row>
  </fo:table-body>
</fo:table>
```

**Formatting properties**

Most of XSL-FO attributes relate to style properties analogue to CSS properties.

- margin, padding, border-style
- background-color, background-image
- font-family, font-weight, font-style, font-size
- text-align, text-align-last, text-indent, start-indent, end-indent, wrap-option,
- break-before
- and much more (almost 300 properties in XSL 1.1)

**“Page master” – page template**

- Single page layout
- A document may be split in many such pages
- One body may be split in many such pages
- Four predefined (but optional to use) edge regions

```
<fo:page-master master-name="A4">
  <fo:region-body width="297mm" height="210mm" margin="3cm">
    <fo:block>
      <fo:block>
        Employees of <xsl:value-of select="company/name"/>
      </fo:block>
    </fo:block>
  </fo:region-body>
</fo:page-master>
```

**Distributing content to pages**

- page-sequence – results in a number of pages
- flow – content split into pages
- static-content – content repeated on all pages
- flow-name – page region reference

```
<fo:page-sequence master-reference="A4">
  <fo:repeatable-page-master-reference master-reference="first"/>
  <fo:repeatable-page-master-reference master-reference="default"/>
</fo:page-sequence>
```

**Page sequence master**

- Using different page layouts within one page-sequence
- Simple page masters referred to be used in order (repetitions available)

```
<fo:layout-master-set>
  <fo:page-sequence-master master-name="first"/>
  <fo:page-sequence-master master-name="second"/>
</fo:layout-master-set>
```

```
<fo:page flow-name="xsl-region-body"/>
```
Tools mentioned today
- Adobe FrameMaker
  - especially useful for large and complex text documents
  - advanced support for XML and structured documents
  - constructs analogous to DTD and stylesheets
- Adobe InDesign
  - especially useful for documents that have to look perfectly
  - basic support for XML be means of filling a template with XML content

Adobe FrameMaker
- Word processor / desktop publishing tool
  - One of first that advanced tools
  - Acquired by Adobe in 1995
- Especially popular for:
  - complex documents, where structure important
  - large documents, e.g. technical documentation
- Two kinds of documents (2 ways of authoring):
  - unstructured - flat, paragraph-based structure, similar to styles in popular word processors
  - structured - tree-like structure, based on SGML and XML

FrameMaker augments a structural approach to the content with a WYSIWYG editor convenience.

Structured application
- EDD - Element Definition Document (or Elements Catalogue)
  - document structure definition (elements, attributes)
  - formatting and other rules
- structured template - FM document
  - pagination, layout, header and footer, ...
  - styles ("paragraph/character format tags"), variables, markers, cross-reference formats, ...
  - Elements Catalogue imported from EDD

Optional components:
- DTD - may also be generated from EDD
- Read/write rules - extra translations between XML and FM
- XSLT pre- and post-processing
- API client - custom executable application

Structured application dependencies

Element definition examples
- EDD shown in document view

Content model (General rule)
- Expression built from element names, <TEXT> token, parentheses, and:
  - grouping symbols (between element names or ( ) groups)
  - - sequence of subelements
  - & - subelements in any order
  - | - choice
  - occurrence indicators (after element name or ( ) group):
    - ? - optional element (0-1 occurrence)
    - * - any number of occurrence (0-unbounded)
    - + - at least one occurrence (1-unbounded)
    - # - non indicator - exactly one occurrence

Examples:
- imi+,
- nazwisko
- Title, Abstract?, Section*

Kinds of elements (examples)
- Container
  - element with no special meaning
  - may contain elements or text (or both → mixed model)
- CrossReference - FM cross-reference
- Footnote - FM footnote
- Equation, Graphic - anchored objects; XML would contain references to external entities
- Marker - FM marker
- SystemVariable - FM system variable reference
- Table, TableBody, TableHeading, TableRow, TableCell - table components

Structured documents in FM
- Structured application
  - FM concept analogous to XML application in XML world
  - FM manages a set of registered structured applications
- XML documents opened / saved directly:
  - Template and formatting rules from EDD define the formatting.
  - When an XML file is opened, the XML content fills the template and formatting rules define the style applied to the content...
  - Manual formatting available in FM, but lost when document saved as XML.

Element Definition Document
- FM document defining other documents structure
  - EDD role corresponding to (in general XML applications):
    - DTD or XML Schema – structure definition
    - CSS or XSL (to some extent) - formatting rules
  - Structure definition
    - available elements, their type and acceptable content
    - attributes, their type and optionality
  - Particular elements marked as FM special objects (tables and table components, variables, markers, cross-references, ...)
  - Rules for elements:
    - formatting
    - initial value or structure
    - prefix and suffix
EDD and DTD – similarities

- Document structure definition
- Container elements
- Content model specification (, *, ?, +)
- Optional and required attributes
- Unique ID, ID Reference - ID, IDREF in DTD

EDD and DTD – differences

EDD
- FrameMaker-special element kinds (tables, variables, etc.)
- Numeric attribute types
- Multi-value attributes
- & - elements in any order
- Formatting rules
- No means for structure modularisation
  - style modularisation available through format change lists

DTD
- General-purpose elements
  - like EDD Container
- No numeric types (for XML)
- Space-separated NMTOKENS and IDREFS
- Only choice and sequence
- No formatting specification
- Parameter entities as means for DTD modularisation

Formatting rules

- Appearance of particular elements described in EDD
- In element definition (e.g., Container) rules grouped by scope of effect:
- TextFormatRules - formatting of whole element, inherited by descendants
- FirstParagraphRules, LastParagraphRules - formatting of first / last paragraph only
- PrefixRules, SuffixRules - context generated in front / at end of element and its formatting
- Some more features analogous to CSS selectors:
  - context rules
  - level rules
- We omit the rest of the details here...

Editing rules – example

- Text Format rules
  1. Count ancestors named: Section
     - If level is: 1
     - Default font properties
     - Font size: 18
     - If level is: 2
     - Default font properties
     - Font size: 14
     - If level is: 3
     - Default font properties
     - Font size: 10

Adobe InDesign

- Advanced desktop publishing tool
  - part of Adobe Creative Suite
- Especially helpful when:
  - text and graphics mixed together
  - advanced, non-standard page layouts used
  - precise positioning and typsetting required
  - high-quality printout planned
- Compromise between word processor and graphic design tool
- Less structure support when compared to Adobe FrameMaker or XML-based solutions
  - harder to automatize publication process
  - easier to make ad-hoc formatting enhancements

Context rules – example

- Text Format rules
  1. If context is: List [Type = "Bulleted"]
     - Numbering properties
     - Autonumber format: <li>
     - Character format: bulletsymbol
     - Else, if context is: List [Type = "Numbered"]
       1.1 If context is: (first)
          - Numbering properties
          - Autonumber format: n=1<tt>n</tt>
          - Else
          - Numbering properties
          - Autonumber format: <tt>n</tt>

XML in InDesign

- Parts of document may be annotated with XML-like structure
- Exporting tagged content as XML
- Importing XML, and then:
  - manually distributing XML fragments among text frames
  - automatic layout of imported content if placeholders were prepared
- Relating InDesign styles and XML tags:
  - applying styles to tags automatically
  - tagging content based on styles
  - XSLT pre- and post-processing
Manual formatting and style application
- XML content formatted as any text content
  - manual formatting
  - formatting with styles
- Recommended for:
  - one-time project (formatting not intended to be used again)
  - short text
  - non-repeating XML elements or formatting unrelated to XML structure

Mapping tags to styles - example
- Sample document before and after applying styles

<table>
<thead>
<tr>
<th>Alice Blonde</th>
</tr>
</thead>
<tbody>
<tr>
<td>United Kingdom</td>
</tr>
<tr>
<td>Cardiff</td>
</tr>
<tr>
<td>City</td>
</tr>
<tr>
<td>3131101</td>
</tr>
<tr>
<td><a href="mailto:alice@example.co.uk">alice@example.co.uk</a></td>
</tr>
</tbody>
</table>

Import XML - two main approaches
- Content-first approach:
  - import content
  - then take care of it:
    - distribute content to text frames
    - format (by e.g. mapping tags to styles)
- Placeholders approach:
  - prepare document with stub content distributed and formatted as desired
  - then import (merge) XML and get it distributed and formatted automatically

Mapping tags to styles
- Automatic application of styles to elements
  - paragraph, character, table, and table cell styles applicable
  - Styles reapplied in all tagged stories of document
  - Elements with the same name receive the same chosen style
  - Styles have to be already defined

Mapping tags to styles - remarks
- Benefits:
  - fast formatting of large documents
  - consistent formatting
  - easy style enhancements in future
- Difficulties and discomforts:
  - styles have to be defined manually before mapping
  - special characters – paragraph breaks, spaces etc. – have to exist in structured content before formatting
  - one paragraph style used for many XML elements in case that those elements reside in the same source line
  - unneeded indents and line breaks from XML cannot be eliminated easily

Style attributes – more flexible, but less convenient approach

Importing XML – content first approach
- Benefits:
  - real document visible while preparing layout and formatting
  - fast final result
- Disadvantages:
  - manual work to do each time document is imported
  - Reasonable usage:
  - one-time process
  - e.g. importing structured content into larger, unstructured document, produced on special demand...
  - preliminary step while preparing template in placeholder approach

Importing XML – placeholders approach
- Benefits:
  - cheap application to arbitrary many documents
  - repeatable, predictable results
- Disadvantages:
  - more preliminary work required
  - Reasonable usage:
  - repeatable tasks
  - part of (semi-)automatised publication process when (part of) data comes from external XML
  - InDesign document as (very advanced...) stylesheet
  - saved as template allows for easy fresh documents creation
### Placeholders prepared - example

<table>
<thead>
<tr>
<th>First</th>
<th>Last</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel.</td>
<td>123456789</td>
</tr>
<tr>
<td>Email</td>
<td><a href="mailto:alice@example.com">alice@example.com</a></td>
</tr>
</tbody>
</table>

### XML imported into placeholders

<table>
<thead>
<tr>
<th>Alice Blonde</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>Tel.</td>
</tr>
<tr>
<td>Email</td>
</tr>
</tbody>
</table>
Some more XML applications and XML-related standards (XLink, XPointer, XForms)

Patryk Czarnik
XML and Applications 2014/2015
Lecture 12 – 19.01.2015

Standards for inter-document relations
- XPointer – addressing documents and their fragments
- XInclude – logical inclusion of documents within other documents
- XLink – declarative relations between documents and their fragments

XPointer
- The standard defines addressing XML documents and their fragments using standard URI syntax:
- 3 W3C recommendations dated 2002-2003:
  - XPointer Framework
    - http://www.w3.org/TR/xptr-framework/
  - XPointer element() Scheme
    - http://www.w3.org/TR/xptr-element/
  - XPointer xmlns() Scheme
    - http://www.w3.org/TR/xptr-xmlns/
  - XPointer xpointer() Scheme
    - http://www.w3.org/TR/xptr-xpointer/
    - (neverending?) Working Draft

XInclude
- Including external XML documents (or their fragments) in another XML document.
- Similar to entities, but:
  - normal element markup, no special syntax,
  - no need to declare anything in DTD, nor to have DTD at all
  - Main capabilities:
    - including complete documents (identified by URL) or their fragments (pointed by XPointer)
    - including XML tree (default) or raw text
    - defining content to be used in case of an error
  - Supported by many parsers, including Java (JAXP).

XLink
- HTML links (<a>, <img>):
  - link two documents: link source and target
  - link source is always in the linking element
- XLink — an extended idea of linking:
  - link information represented in any element:
    - element name is not important
    - attributes coming from XLink namespace are
    - more than two ends of link (hyperlink = relation)
    - possibility to represent link outside linked resources
- Status:
  - historical roots: HyTime,
  - XLink 1.0 – W3C recommendation: 2001,
  - XLink 1.1 – current version (made official TR: May 2010).

XPointer – xpointer scheme
- xpointer scheme allows to address elements using XPath:
  - xmlns scheme adds namespace declarations to the above:
    - ustawa.xml#xmins(pr=http://www.sejm.gov.pl/prawo)
    - xpointer(pr:art[5]/pr:par[2])

XInclude – example
- <recipe>
  - xmlns:xmins:x="http://www.w3.org/2001/XInclude"
    - href="sallad.xmlexpnter"/recipe title=""
    - <x:failback>
      - <error>No such recipe.</error>
    - </x:failback>
    </x:include>
  </recipe>

Terminology
- Resource – any addressable unit of information or a service (file, program, query result).
- Link – a relation between participating resources, expressed explicitly with a linking element.
- Arc – information about traversal between labelled resources (in defined direction):
  - outbound arc – from a local resource to some external resource
  - inbound arc – from an external resource to some local resource
  - third party – between two external resources
- Note: a resource is regarded as remote when addressed by URI (even though it resides in the same document or linking element as the link which uses it).
Types of links

- Simple link:
  - is outbound
  - binds exactly two resources: a local one with an external one
  - contains exactly one arc between resources
- Extended link:
  - binds arbitrary number of local and external resources, uses arcs to define methods of traversal between resources, defines roles of participating resources, defines roles of arcs.

Attributes in extended links

- type - role of the element in a link
  - simple | extended | locator | arc | resource | title | none
- href - URI of the external resource
- role - abstract identifier of the resource role (URI)
- arcrole - as above, but for an arc
- title - text label of the resource or arc
- show - presentation info: new | replace | embed | other | none
- actuate - activation info: onLoad | onRequest | other | none
- label - label used as identifier in from and to, not necessarily unique
- from, to - pointer (in an arc) for a certain resource label

XForms - document structure

- Forms are embedded in a host document, usually XHTML
- Data model - xf:model element
  - anywhere in host document
  - header more elegant
  - but body more practical for dynamic documents
  - more than one model available; in such case they must have identifiers
- Form controls (in XForms namespace)
  - placed within normal XHTML tags
  - (some of them) may contain further XHTML fragments
- Action specifications and constraints tied with XForm elements
  - by inserting them inside model fragments or control tags
  - using general xf:bind elements

Simple example

```xml
<html xmlns="http://www.w3.org/1999/xhtml" xmlns:xf="http://www.w3.org/2002/xforms">
  <head>
    <xf:model>
      <xf:instance>
        <xf:person>
          <xf:firstName>John</xf:firstName>
          <xf:lastName>Smith</xf:lastName>
        </xf:person>
      </xf:instance>
    </xf:model>
  </head>
</html>
```

Future of XLink

- Applications:
  - organization and association of resources even when no writing permission is granted
  - a new type of added value – link sets
- Scope:
  - local – link servers, link databases
  - Internet?
- Problems:
  - visualization of extended links
  - synchronization of links and resources (Internet)

XForms

- XML application for specification of interactive forms
- Versions:
  - 1.0 – 2003
  - 1.1 – 2009 (currently most commonly used)
  - 2.0 – WD
- More than HTML forms:
  - data model defined separately from UI
    - by example or using XML Schema
  - processing model specified with events and actions
  - various data access modes given in submission module
    - including REST-compliant HTTP access
  - more UI controls, interactive switch, automatic repeat

Simple example

```xml
<body>
  <div>
    <xf:input ref="first-name"/>
    <xf:label for="first-name"/>
    <xf:hint>Your first (given) name</xf:hint>
  </div>
</body>
```
REST services – recall

- REST for Representational State Transfer
- Principles:
  - Service = collection of resources
  - URL identifies a resource
  - Resource has a normalised representation and can be transferred through the network
  - XML for structural data
  - Binary and other structural formats (JSON) also permitted
- HTTP methods directly used to manipulate resources
  - GET, PUT, DELETE – obvious semantics
  - Other HTTP methods, HTTP authentication, cookies, additional headers and arguments – all may be used to implement additional features

Some popular XML applications

- Multimedia
  - Scalable Vector Graphics (SVG)
  - Mathematical Markup Language (MathML)
- Security
  - XML Signature
  - XML Encryption

REST for XML database

- REST – remote access to a repository
  - Can it be an XML database? Why not...
- Possible applications:
  - Access API independent of particular platform or pr.lang.
  - Easy and efficient remote access from
    - Javascript clients (AJAX)
    - Mobile clients
  - Integration with XML-related standards
    - XSLT, XQuery - documents available through HTTP URLs
    - XForms - acquiring and modifying documents directly form XForms
- HTTP interface available also to call server-side XQuery scripts
- XRX architecture: XForms + REST + XQuery

Some popular XML applications

- Documents / text processing / publications:
  - DocBook
  - Text Encoding Initiative (TEI)
  - Darwin Information Typing Architecture (DITA)
  - Open Document (ODF, OASIS standard, ISO/IEC 26300)
  - Office Open XML (OOXML / OpenXML) (Ecmma standard, ISO/IEC 29500)
- Metadata and knowledge representation:
  - Dublin Core
  - RDF
  - Topic Maps