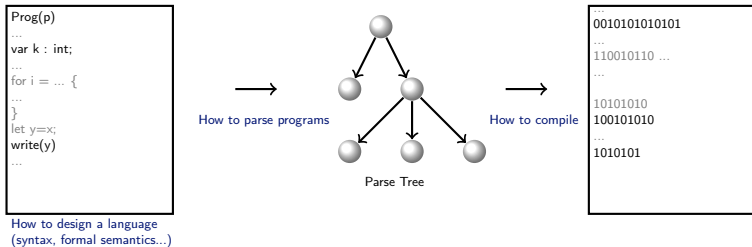


XML Introduction

Pierre Genevès
CNRS

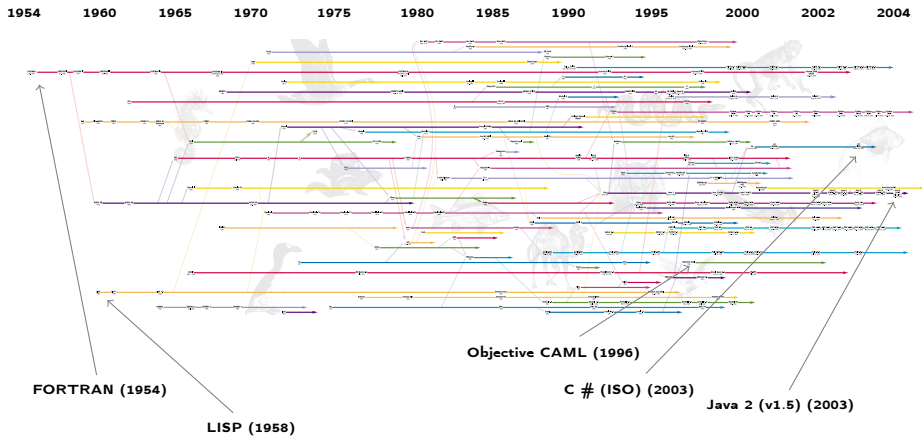
University of Grenoble, 2011–2012

You Know...



How to design and implement a **programming language**

History of Programming Languages

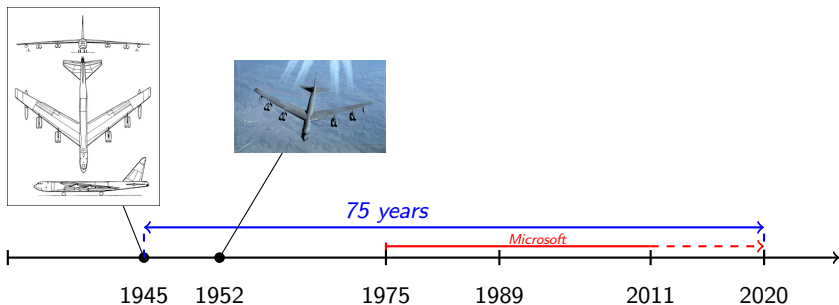


What about Data?

- Often, data is **more important** than programs (e.g. banks, aeronautical technical documentation, ...)
 - One reason for this is that data often have a much longer life cycle than programs
- **How to ensure long-term access to data?**
- How to design software systems such that manipulated data can still be accessed 15 or 50 years later?

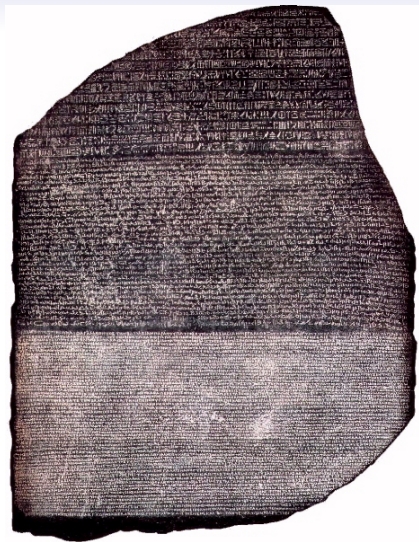
Example: Aeronautical Technical Documentation

In aeronautics, it is common to find products with life cycles that last for several decades, e.g. the B-52:



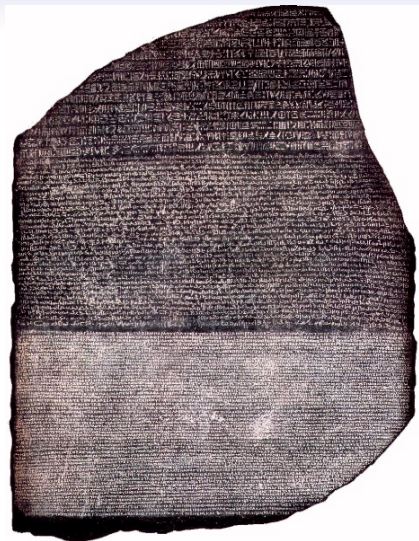
→ **How to ensure long-term
access to data?**

- How to ensure long-term access to data?
- An old concern...



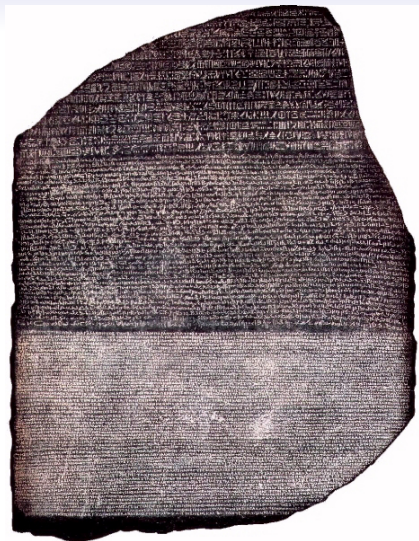
La pierre de Rosette.

- How to ensure long-term access to data?
- An old concern...
 - Can we really do better with computers?



La pierre de Rosette.

- **How to ensure long-term access to data?**
- An old concern...
 - Can we really do better with computers?
- A computer museum? 😊



La pierre de Rosette.

What has not changed for 50 years in Computer Science?

What has not changed for 50 years in Computer Science?



Data Exchange – What Happened

- Often, data must be sent to a third-party program/person
- Data must be made explicit (e.g. storage in files)
- Widespread approach until the 1990's for defining a *file format*:
 - Define (binary?) representation for data + instructions, e.g. records
 - Write file format spec (v1.0?) + implement parser

Data Exchange – What Happened

- Often, data must be sent to a third-party program/person
- Data must be made explicit (e.g. storage in files)
- Widespread approach until the 1990's for defining a *file format*:
 - Define (binary?) representation for data + instructions, e.g. records
 - Write file format spec (v1.0?) + implement parser



MachinCompany
File format *X*
Parser *P_x*

Data Exchange – What Happened

- Often, data must be sent to a third-party program/person
- Data must be made explicit (e.g. storage in files)
- Widespread approach until the 1990's for defining a *file format*:
 - Define (binary?) representation for data + instructions, e.g. records
 - Write file format spec (v1.0?) + implement parser



MachinCompany
File format X
Parser P_X

Bidule Inc.
File format Y
Parser P_Y



Data Exchange – What Happened

- Often, data must be sent to a third-party program/person
- Data must be made explicit (e.g. storage in files)
- Widespread approach until the 1990's for defining a *file format*:
 - Define (binary?) representation for data + instructions, e.g. records
 - Write file format spec (v1.0?) + implement parser



Bidule Inc.
File format *Y*
Parser *P_Y*

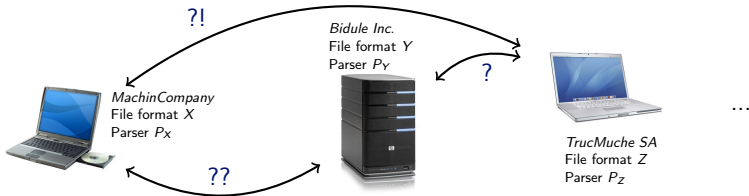


TrucMuche SA
File format *Z*
Parser *P_Z*

...

Data Exchange – What Happened

- Often, data must be sent to a third-party program/person
- Data must be made explicit (e.g. storage in files)
- Widespread approach until the 1990's for defining a *file format*:
 - Define (binary?) representation for data + instructions, e.g. records
 - Write file format spec (v1.0?) + implement parser



- **Problems:** - exchanging **data** → exchanging **programs!**
 - this approach **cannot scale** (and costs \$\$\$\$)

→ Need for **normalization** of data exchange

Motivation for XML:

To have **one** language to describe and exchange data

XML = Data

```
Pierre Genevès  
CNRS  
pierre.geneves@inria.fr
```

```
...
```

```
Nabil Layaïda  
INRIA  
nabil.layaida@inria.fr
```

Text file

XML = Data + Structure

```
Pierre Genevès  
CNRS  
pierre.geneves@inria.fr
```

...

```
Nabil Layaïda  
INRIA  
nabil.layaida@inria.fr
```

Text file

→
"Mark it up!"

```
<related>  
  <colleague>  
    <name>Pierre Genevès</name>  
    <affil>CNRS</affil>  
    <email>pierre.geneves@inria.fr</email>  
  </colleague>  
  ...  
  <colleague>  
    <name>Nabil Layaïda</name>  
    <affil>INRIA</affil>  
    <email>nabil.layaida@inria.fr</email>  
  </colleague>  
</related>
```

XML Document

XML = Data + Structure

```
Pierre Genevès  
CNRS  
pierre.geneves@inria.fr  
...  
Nabil Layaïda  
INRIA  
nabil.layaida@inria.fr
```

Text file

→
"Mark it up!"

```
<related>  
  <colleague>  
    <name>Pierre Genevès</name>  
    <affil>CNRS</affil>  
    <email>pierre.geneves@inria.fr</email>  
  </colleague>  
  ...  
  <colleague>  
    <name>Nabil Layaïda</name>  
    <affil>INRIA</affil>  
    <email>nabil.layaida@inria.fr</email>  
  </colleague>  
</related>
```

XML Document

Tags describe *structure*, independently from processors (tags are **not** implicit parameters for a given processor, e.g. tags are **not** intended for describing presentation)

XML = Data + Structure

```
Pierre Genevès  
CNRS  
pierre.geneves@inria.fr  
...  
Nabil Layaïda  
INRIA  
nabil.layaida@inria.fr
```

Text file

→
"Mark it up!"

```
<related>  
  <colleague>  
    <name>Pierre Genevès</name>  
    <affil>CNRS</affil>  
    <email>pierre.geneves@inria.fr</email>  
  </colleague>  
  ...  
  <colleague>  
    <name>Nabil Layaïda</name>  
    <affil>INRIA</affil>  
    <email>nabil.layaida@inria.fr</email>  
  </colleague>  
</related>
```

XML Document

Is this a good template? What about first/last name?
Several affil's? email's...?

XML Documents

- Ordinary text files (UTF8, UTF16, ...)
- Originates from typesetting/DocProcessing community
- Idea of labeled brackets (“mark up”) for structure is not new! (already used by Chomsky in the 1960's)
- Brackets describe a **tree structure**
- Allows applications from different vendors to exchange data!
- **Standardized, extremely widely accepted!**
- The Lingua franca for communicating on the web...

Standards for Data Exchange



- **Before:** file format tied to a processor (due to processor-specific instructions)
- **After:** markup language for describing (structured) data in itself (independently from processors)

XML History

Ancestors

1974 SGML (Charles Goldfarb at IBM Research)

1989 HTML (Tim Berners-Lee at CERN, Geneva)

1994 Berners-Lee founds World Wide Web Consortium (W3C)

1996 **XML** (W3C draft, v1.0 in 1998)

<http://www.w3.org/TR/REC-xml/>

Initial W3C Goals for XML¹

“The design goals for XML are:

- 1. XML shall be straightforwardly usable over the Internet.*
- 2. XML shall support a wide variety of applications.*
- 3. XML shall be compatible with SGML.*
- 4. It shall be easy to write programs which process XML documents.*
- 5. The number of optional features in XML is to be kept to the absolute minimum, ideally zero.*
- 6. XML documents should be human-legible and reasonably clear.*
- 7. The XML design should be prepared quickly.*
- 8. The design of XML shall be formal and concise.*
- 9. XML documents shall be easy to create.*
- 10. Terseness is of minimal importance.”*

¹<http://www.w3.org/TR/WD-xml-961114>

XML is a Data Exchange Format

- **Contra..** extremely verbose, lots of repetitive markup, large files
- **Pro..** answers much more ambitious goals:
 - **long-standing** (mark-up does not depend on the system where it was created nor on processings)
 - One of the **pillars of the web**
 - We have a **standard! a standard!... A STANDARD!**
 - 😊 **you will never need to write a parser again! Use XML!** 😊

XML is a Meta-Language for Creating Markup-Languages

... instead of writing a parser, **you simply fix your own “XML Dialect”**

by describing all “admissible structures” :

- allowed element names
- how they can be assembled together
- maybe even the specific data types that may appear inside

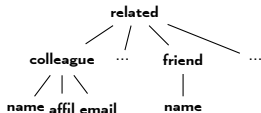
You do this using an **XML Type definition language** such as DTD or Relax NG (Oasis).

Of course, such type definition languages are simple, because you want the parsers to be efficient!

XML Document Type Definition

- The XML Recommendation² includes an XML type definition language for specifying **document types**: DTD

related	→	(colleague friend)*
colleague	→	name, affil ⁺ , email
friend	→	name, affil*, phone*, email?
...		
Document Type		



- Each element is associated with its **content model**: a reg. expr. (, | ? * +)
- A document type (a set of such associations + a particular root element) describes a set of **valid documents** used by an organisation

²<http://www.w3.org/TR/REC-xml/>

XML: What Else?

- Attributes
- Comments
- Processing Instructions
- Entity References
- Namespaces
- Text (a specific node kind)

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

- Attributes
- Comments
- Processing Instructions
- Entity References
- Namespaces
- Text (a specific node kind)

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

```
<!-- a comment here -->
```

- **Attributes**
- **Comments**
- Processing Instructions
- Entity References
- Namespaces
- Text (a specific node kind)

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

```
<!-- a comment here -->
```

→ Attributes

→ Comments

→ Processing Instructions

```
<?php sql("SELECT * FROM .") ... ?>
```

→ Entity References

→ Namespaces

→ Text (a specific node kind)

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

```
<!-- a comment here -->
```

→ Attributes

→ Comments

→ Processing Instructions

```
<?php sql("SELECT * FROM .") ... ?>
```

→ **Entity References**

DTD: **<!ENTITY notice "All rights...">**

→ Namespaces

instance: **<p> Copyright: ¬ice; </p>**

→ Text (a specific node kind)

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

```
<!-- a comment here -->
```

→ Attributes

→ Comments

→ Processing Instructions

```
<?php sql("SELECT * FROM .") ... ?>
```

→ Entity References

```
DTD: <!ENTITY notice "All rights...">
```

→ Namespaces

```
instance: <p> Copyright: &notice; </p>
```

→ Text (a specific node kind)

```
<book xmlns="http://www.books.com/xml"
      xmlns:cars="http://www.cars.com/xml">
  <part><cars:part>Avoids collisions..</cars:part></part>
</book>
```

XML: What Else?

```
<friend surname="Pitou" birthday="08/08">..</friend>
```

```
<!-- a comment here -->
```

→ Attributes

→ Comments

→ Processing Instructions

```
<?php sql("SELECT * FROM .") ... ?>
```

→ Entity References

```
DTD: <!ENTITY notice "All rights...">
```

→ Namespaces

```
instance: <p> Copyright: &notice; </p>
```

→ Text (a specific node kind)

```
<book xmlns="http://www.books.com/xml"
```

```
xmlns:cars="http://www.cars.com/xml">
```

```
<part><cars:part>Avoids collisions..</cars:part></part>
```

```
</book>
```

Back in 2011: XML is 13

“There is essentially no computer in the world, desktop, handheld, or backroom, that doesn't process XML sometimes...”

T. Bray

Some Widespread XML Dialects...

- XHTML (W3C) – the neat version of HTML!
- SVG (W3C) – Animated Vector Graphics
- SMIL (W3C) – Synchronized Multimedia Documents, and MMS
- MathML (W3C) – Mathematical formulas
- XForms (W3C) – Web forms
- Fix, FPML – Financial structured products, transactions ...
- CML – Chemical molecules
- X3D (Web3D) - 3D Graphics
- XUL (Mozilla), MXML (Macromedia), XAML (Microsoft) – Interface Definition Languages
- SOAP (RPC using HTTP), WSDL (W3C), WADL (Sun) – Web Services
- RDF (W3C), OWL (W3C) – Metadata/Knowledge in the Semantic Web
- ...

Intermediate Outline

- Two notions of correctness:
 - Well-formedness
 - Validity
- Defining classes of documents
 - DTDs, XML Schemas
 - Modeling trees and graphs

XML Defines 2 Levels of Correctness

1. Well-formed XML (minimal requirement)

- The **flat text format** seen on the **physical** side, *i.e.* a set of (UTF8/16) character sequences being well-formed XML
- Ensures data correspond to **logical tree-like structures**
(applications that want to analyse and transform XML data in any meaningful manner will find processing flat character sequences hard and inefficient)

2. Valid XML (optional, stricter requirement)

- More often than not, applications require the XML input **trees** to conform to a **specific XML dialect**, defined by *e.g.* a DTD

Well-Formed XML

```
<a att="v">  
  <bar><e/></foo>  
  <c>...</c>  
</a>
```

```
<a att="v1" att="v2">  
  <b>...</b>  
  <c><e/></c></e>  
</a>
```

Not Well-formed XML

```
<a>  
  <b>...</b>  
  <c><d/><e/></c>  
</a>
```



Well-formed XML

- Proper nesting of opening/closing tags
 - Shortcut: `<e/>` for `<e></e>`
 - Every attribute must have a (unique) value
 - A document has one and only one root
 - No ambiguity between structure and data
- Any XML processor considers well-formed XML as a **logical tree structure** which is:
- ordered (except attributes!)
 - finite (leaves are empty elements or character data)

Characters	<	>	"	'	&
Entities	<	>	"	'	&

→ It **must** stop for not well-formed XML.

Valid XML

- The header of a document **may** include a reference to a DTD:

```
<!DOCTYPE root PUBLIC "public-identifier" "uri.dtd">
```

- A document with such a declaration must be **valid** wrt the declared type
- The parser will **validate** it

Example

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN"  
  "http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">  
<html>  
  ...  
</html>
```

Why Validate?

- A document type is a **contract** between data producers and consumers
- Validation allows:
 - a producer to check that he produces what he promised
 - a consumer to check what the producer sends
 - a consumer to protect his application
 - **leaving error detection up to the parser**
 - simplifying applications (we know where to find relevant information in valid documents)
 - delivering high-speed XML throughput (once the input is validated, a lot of runtime checks can be avoided)

Document Type Definition (DTD)

- Any element *e* to be used in the XML dialect needs to be introduced via

`<!ELEMENT e cm>`

Content model <i>cm</i>	Valid content
ANY	arbitrary well-formed XML content
EMPTY	no child elements allowed (attributes OK)
Reg. exp. over tag names, #PCDATA, and constructors , + * ?	order and occurrence of child elements and text content must match the regular expression

- Example (XHTML 1.0 Strict DTD): `<!ELEMENT img EMPTY>`

Reg. Exp. in DTD Content Models

Reg. Exp.	Semantics
tagname	element named tagname
#PCDATA	text content (parsed character data)
c_1, c_2	c_1 directly followed by c_2
$c_1 \mid c_2$	c_1 or, alternatively, c_2
c^+	c , one or more times
$c?$	optional c

Example: recipes.xml (fragment)

```
1 <!ELEMENT recipe (title,ingredient*,preparation,comment?,nutrition)>
2 <!ELEMENT title (#PCDATA)>
3 <!ELEMENT ingredient (ingredient*,preparation)?>
4 <!ELEMENT preparation (step*)>
```

Declaring Attributes

- Using the DTD ATTLIST declaration, validation of XML documents is extended to attributes
- The ATTLIST declaration associates a list of **attribute names** a_i with their owning element e :

```
<!ATTLIST e
```

```
     $a_1$   $\tau_1$   $d_1$ 
```

```
    ...
```

```
     $a_n$   $\tau_n$   $d_n$ 
```

```
>
```

- The **attribute types** τ_i define which values are valid for attributes a_i .
- The **defaults** d_i indicate if a_i is required or optional (and, if absent, if a default value should be assumed for a_i).
- In XML, attributes of an element are **unordered**. The ATTLIST declaration prescribes no order of attribute usage.

- Via **attribute types**, control over the valid attribute values can be exercised:

Attribute Type τ_i	Semantics
CDATA	character data (no < but < ;, ...)
$(v_1 \mid v_2 \mid \dots \mid v_n)$	enumerated literal values
ID	value is document-wide unique identifier for owner element
IDREF	references an element via its ID attribute

Example: academic.xml

```
1 <!ELEMENT academic (Firstname, Middlename*, Lastname)>
2 <!ATTLIST academic
3   title (Prof|Dr) #REQUIRED
4   team CDATA #IMPLIED
5 >
6
7 <academic title="Dr" team="WAM"> ... </academic>
```

- Attribute defaulting in DTDs:

Attribute Default d_i	Semantics
#REQUIRED	element must have attribute a_i
#IMPLIED	attribute a_i is optional
v (a value)	attribute a_i is optional, if absent, default value v for a_i is assumed
#FIXED v	attribute a_i is optional, if present, must have value v

Example: contacts.xml

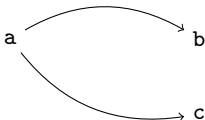
```
1 <!ELEMENT contact (name, email+, phone*)>
2 <!ATTLIST contact
3   emailMode (text|xhtml) "text"   <!--send safely-->
4 >
```

Crossreferencing via ID and IDREF

- Well-formed XML documents essentially describe tree-structured data
- Attributes of type ID and IDREF may be used to encode **graph structures** in XML. A validating XML parser can check such a graph encoding for consistent connectivity.
- To establish a directed edge between two XML nodes a and b:



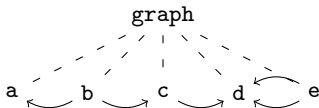
1. attach a unique **identifier** to node b (using an ID attribute), then
2. **refer** to b from a via this identifier (using an IDREF attribute).
3. For an outdegree > 1 (see below), use an IDREFS attribute.



Graphs in XML – An Example

Graph.xml

```
1 <?xml version="1.0"?>
2 <!DOCTYPE graph [
3   <!ELEMENT graph (node+) >
4   <!ELEMENT node ANY > <!-- attach arbitrary data to a node -->
5   <!ATTLIST node
6     id ID #REQUIRED
7     edges IDREFS #IMPLIED > <!-- we may have nodes with outdegree 0 -->
8 ]>
9
10 <graph>
11   <node id="A">a</node>
12   <node id="B" edges="A C">b</node>
13   <node id="C" edges="D">c</node>
14   <node id="D">d</node>
15   <node id="E" edges="D D">e</node>
16 </graph>
```



Example (Character references in “ComicsML”)

```
ComicsML.dtd (fragment)
1 <!DOCTYPE strip [
2   ...
3   <!ELEMENT character (#PCDATA) >
4   <!ATTLIST character
5     id      ID              #REQUIRED >
6   <!ELEMENT bubble (#PCDATA) >
7   <!ATTLIST bubble
8     speaker IDREF          #REQUIRED
9     to      IDREFS         #IMPLIED
10    tone    (angry|question|...) #IMPLIED >
11 ]>
```

Validation results (message generated by Apache's Xerces):

- Setting attribute to some random non-existent character identifier:
ID attribute 'yoda' was referenced but never declared
- Using a non-enumerated value for attribute tone:
Attribute 'tone' does not match its defined enumeration list

Other DTD Features

- **User-defined entities** via `<!ENTITY e d>` declarations (usage: `&e;`)

```
<!ENTITY pam "Pierre-Antoine-Marie">
```

- **Parameter Entities** ("DTD macros") via `<!ENTITY % e d >` (usage: `;%e;`)

```
<!ENTITY ident "ID #REQUIRED">
...
<!ATTLIST character
  id %ident; >
```

- **Conditional sections** in DTDs via `<![INCLUDE[...]]>` and `<![IGNORE[...]]>`

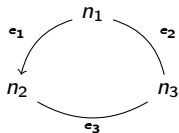
```
<!ENTITY % draft 'INCLUDE' >
<!ENTITY % final 'IGNORE' >

<![%draft;[
<!ELEMENT book (comments*, title, body, supplements?)>
]]>
<![%final;[
<!ELEMENT book (title, body, supplements?)>
]]>
```

A "Real Life" DTD – GraphML

- GraphML³ has been designed to provide a convenient file format to represent arbitrary graphs
- Graphs (element graph) are specified as lists of nodes and edges
- Edges point from source to target
- Nodes and edges may be annotated using arbitrary description and data
- Edges may be directed (+ attribute edgedefault of graph)

```
1 <graphml>
2   <graph edgedefault="undirected">
3     <node id="n1"/>
4     <node id="n2"/>
5     <node id="n3"/>
6     <edge id="e1" source="n1" target="n2" directed="true"/>
7     <edge id="e2" source="n2" target="n3" directed="false"/>
8     <edge id="e3" source="n3" target="n1"/>
9   </graph>
10 </graphml>
```



³<http://graphml.graphdrawing.org/>

```

1 <!-- GRAPHML DTD (flat version) ===== -->
2 <!ELEMENT graphml ((desc)?,(key)*,((data)|(graph))*>
3
4 <!ELEMENT locator EMPTY>
5 <!ATTLIST locator
6     xmlns:xlink CDATA #FIXED "http://www.w3.org/TR/2000/PR-xlink-20001220/"
7     xlink:href CDATA #REQUIRED
8     xlink:type (simple) #FIXED "simple">
9
10 <!ELEMENT desc (#PCDATA)>
11
12 <!ELEMENT graph ((desc)?,(((data)|(node)|(edge)|(hyperedge))*|(locator)))>
13 <!ATTLIST graph
14     id ID #IMPLIED
15     edgedefault (directed|undirected) #REQUIRED>
16
17 <!ELEMENT node (desc?,(data|port)*,(graph?)|locator)>
18 <!ATTLIST node
19     id ID #REQUIRED>
20
21 <!ELEMENT port ((desc)?,(data)|(port))*>
22 <!ATTLIST port
23     name NMTOKEN #REQUIRED>
24
25 <!ELEMENT edge ((desc)?,(data)*,(graph?)>
26 <!ATTLIST edge
27     id ID #IMPLIED
28     source IDREF #REQUIRED
29     sourceport NMTOKEN #IMPLIED
30     target IDREF #REQUIRED
31     targetport NMTOKEN #IMPLIED
32     directed (true|false) #IMPLIED>
33
34 <!ELEMENT key (#PCDATA)>
35 <!ATTLIST key
36     id ID #REQUIRED
37     for (graph|node|edge|hyperedge|port|endpoint|all) "all">
38
39 <!ELEMENT data (#PCDATA)>
40 <!ATTLIST data
41     key IDREF #REQUIRED
42     id ID #IMPLIED>

```

Concluding Remarks

- DTD syntax:
 - ✓ **Pro:** compact, easy to understand
 - ✗ **Con:** ?

Concluding Remarks

- DTD syntax:
 - ✓ **Pro:** compact, easy to understand
 - ✗ **Con:** not in XML!
 - DTD functionality:
 - ✗ no fine-grained types (everything is character data; what about, e.g. integers?)
 - ✗ no further occurrence constraints (e.g. cardinality of sequences)
- DTD is a quite limited type definition language (but: see XML Schema...)

XML Schema

- With XML Schema⁴, W3C provides an XML type definition language that goes beyond the capabilities of the “native” DTD concept:
 - XML Schema descriptions are valid XML documents themselves
 - XML Schema provides a rich set of built-in data types
 - Users can extend this type system via user-defined types
 - XML element (and attribute) types may even be derived by inheritance

XML Schema vs. DTDs

→ Why would you consider this an advantage?

⁴<http://www.w3.org/TR/xmlschema-0/>

Some XML Schema Constructs

Declaring an element

```
<xsd:element name="author"/>
```

No further typing specified: the author element may contain string values only.

Declaring an element with bounded occurrence

```
<xsd:element name="character" minOccurs="0" maxOccurs="unbounded"/>
```

Absence of minOccurs/maxOccurs implies exactly once.

Declaring a typed element

```
<xsd:element name="year" type="xsd:date"/>
```

Content of year takes the format YYYY-MM-DD. Other **simple types**: string, boolean, number, float, duration, time, AnyURI, ...

- **Simple types** are considered **atomic** with respect to XML Schema (e.g., the YYYY part of an xsd:date value has to be extracted by the XML application itself).

- Non-atomic **complex types** are built from simple types using **type constructors**.

Declaring sequenced content

```
1 <xsd:complexType name="Characters">
2   <xsd:sequence>
3     <xsd:element name="character" minOccurs="1"
4                           maxOccurs="unbounded"/>
5   </xsd:sequence>
6 </xsd:complexType>
7 <xsd:complexType name="Prolog">
8   <xsd:sequence>
9     <xsd:element name="series"/>
10    <xsd:element name="author"/>
11    <xsd:element name="characters" type="Characters"/>
12  </xsd:sequence>
13 </xsd:complexType>
14 <xsd:element name="prolog" type="Prolog"/>
```

An `xsd:complexType` may be used anonymously (no name attribute).

- With attribute `mixed="true"`, an `xsd:complexType` admits **mixed content**.

- New complex types may be **derived** from an existing (base) type.

Deriving a new complex type

```
1 <xsd:element name="newprolog">
2   <xsd:complexType>
3     <xsd:complexContent>
4       <xsd:extension base="Prolog">
5         <xsd:element name="colored" type="xsd:boolean"/>
6       </xsd:extension>
7     </xsd:complexContent>
8   </xsd:complexType>
9 </xsd:element>
```

- **Attributes** are declared within their owner element.

Declaring attributes

```
1 <xsd:element name="strip">
2   <xsd:attribute name="copyright"/>
3   <xsd:attribute name="year" type="xsd:gYear"/> ...
4 </xsd:element>
```

Other `xsd:attribute` modifiers: use (required, optional, prohibited), fixed, default.

- The validation of an XML document against an XML Schema goes as far as peeking into the **lexical representation** of simple typed values.

Restricting the value space of a simple type (enumeration)

```
1 <xsd:simpleType name="Car">
2   <xsd:restriction base="xsd:string">
3     <xsd:enumeration value="Audi"/>
4     <xsd:enumeration value="BMW"/>
5     <xsd:enumeration value="VW"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

Restricting the value space of a simple type (regular expression)

```
1 <xsd:simpleType name="AreaCode">
2   <xsd:restriction base="xsd:string">
3     <xsd:pattern value="0[0-9]+"/>
4     <xsd:minLength value="3"/>
5     <xsd:maxLength value="5"/>
6   </xsd:restriction>
7 </xsd:simpleType>
```

- Other **facets**: length, maxInclusive (upper bound for numeric values)...

Other XML Schema Concepts

- **Fixed** and **default** element content,
- support for **null values**,
- uniqueness constraints, arbitrary **keys** (specified via XPath)
- ...

Intermediate Outline

- Processing XML Documents
- Parsing
 - Two radically different approaches: DOM and SAX
 - Advantages and drawbacks

XML Processing Model

Validation is good

- Validation is better than writing code
- Remember the promise:
 - “**you will never have to write a parser again during your lifetime**”
 - instead, you will spend your lifetime trying to encode the right grammar ☺...
- Virtually all XML applications operate on the logical tree view which is provided to them by an [XML parser](#)
- An XML parser can be validating or non-validating
- XML parsers are widely available (e.g. Apache's Xerces).
- How is the XML parser supposed to communicate the XML tree structure to the application?

XML Parsers

- Two different approaches:
 1. Parser **stores** document into a fixed (standard) data structure (e.g. DOM)

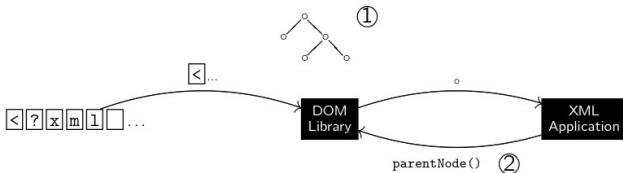
```
parser.parse("foo.xml");  
doc = parser.getDocument();
```

2. Parser **triggers events**. Does not store! User has to write own code on how to store / process the events triggered by the parser.

Next slides on DOM & SAX by Marc H. Scholl (Uni KN)...

DOM—Document Object Model

- With **DOM**, W3C has defined a **language-** and **platform-neutral** view of XML documents.
- DOM APIs exist for a wide variety of—predominantly object-oriented—programming languages (Java, C++, C, Perl, Python, ...).
- The DOM design rests on two major concepts:
 - ① An **XML Processor** offering a DOM interface parses the XML input document, and constructs the **complete XML document tree** (in-memory).
 - ② The **XML application** then issues DOM library calls to **explore** and **manipulate** the XML document, or **generate** new XML documents.



- The DOM approach has some obvious advantages:
 - ▶ Once DOM has build the XML tree structure, (tricky) issues of XML grammar and syntactical specifics are void.
 - ▶ **Constructing** an XML document using the DOM instead of serializing an XML document manually (using some variation of `print`), ensures **correctness** and **well-formedness**.
 - ★ No missing/non-matching tags, attributes never owned by attributes, ...
 - ▶ The DOM can simplify document **manipulation** considerably.
 - ★ Consider transforming

Weather forecast (English)

```
1 <?xml version="1.0"?>
2 <forecast date="Thu, May 16">
3   <condition>sonny</condition>
4   <temperature unit="Celsius">23</temperature>
5 </forecast>
```

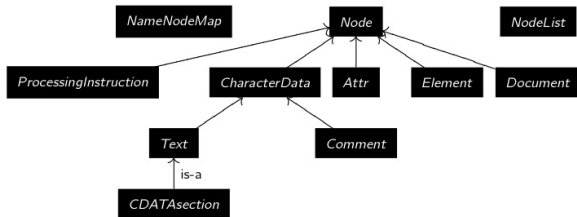
into

Weather forecast (German)

```
1 <?xml version="1.0"?>
2 <vorhersage datum="Do, 16. Mai">
3   <wetterlage>sonnig</wetterlage>
4   <temperatur skala="Celsius">23</temperatur>
5 </vorhersage>
```

DOM Level 1 (Core)

- To operate on XML document trees, DOM Level 1⁴ defines an inheritance hierarchy of node objects—and methods to operate on these—as follows (excerpt):



- Character strings (DOM type *DOMString*) are defined to be encoded using UTF-16 (e.g., Java DOM represents type *DOMString* using its *String* type).

⁴<http://www.w3.org/TR/REC-DOM-Level-1/>

- (The complete DOM interface is too large to list here.) Some methods of the principal DOM types *Node* and *Document*:

DOM Type	Method	Comment
<i>Node</i>	<i>nodeName</i> :: <i>DOMString</i>	redefined in subclasses, e.g., tag name for <i>Element</i> , "#text" for <i>Text</i> nodes, ...
	<i>parentNode</i> :: <i>Node</i>	
	<i>firstChild</i> :: <i>Node</i>	leftmost child node
	<i>nextSibling</i> :: <i>Node</i>	returns NULL for root element or last child or attributes
	<i>childNodes</i> :: <i>NodeList</i>	see below
	<i>attributes</i> :: <i>NameNodeMap</i>	see below
	<i>ownerDocument</i> :: <i>Document</i>	
	<i>replaceChild</i> :: <i>Node</i>	replace new for old node, returns old
<i>Document</i>	<i>createElement</i> :: <i>Element</i>	creates element with given tag name
	<i>createComment</i> :: <i>Comment</i>	creates comment with given content
	<i>getElementsByTagName</i> :: <i>NodeList</i>	list of all <i>Elem</i> nodes in document order

Some DOM Details

- Creating an element (or attribute) using *createElement* (*createAttribute*) does *not* wire the new node with the XML tree structure yet.
Call *insertBefore*, *replaceChild*, ... to wire a node at an explicit position.
- DOM type *NodeList* (node sequence) makes up for the lack of collection datatypes in most programming languages.
Methods: *length*, *item* (node at specific index position).
- DOM type *NameNodeMap* represents an *association table* (nodes may be accessed by name).

Example:

bubble



Apply method *attributes* to
Element object ν_0 to obtain
this *NameNodeMap*:

<u>name</u>		<u>node</u>
"speaker"	↦	α_1
"to"	↦	α_2

speaker to

Methods: *getNamedItem*, *setNamedItem*, ...

DOM Example Code

- The following slide shows C++ code written against the Xerces C++ DOM API⁵.
- The code implements a variant of the *content :: Doc* \rightarrow (*Char*):
 - ▶ Function `collect ()` decodes the UTF-16 text content returned by the DOM and prints it to standard output directly (`transcode ()`, `cout`).

N.B.

- A W3C DOM node type named τ is referred to as `DOM_` τ in the Xerces C++ DOM API.
- A W3C DOM property named *foo* is—in line with common object-oriented programming practice—called `getFoo ()` here.

⁵<http://xml.apache.org/>

Example: C++/DOM Code

```
1 // Xerces C++ DOM API support
2 #include <dom/DOM.hpp>
3 #include <parsers/DOMParser.hpp>
4
5 void collect (DOM_NodeList ns)
6 {
7     DOM_Node n;
8
9     for ( unsigned long i = 0;
10          i < ns.getLength ();
11          i++){
12         n = ns.item (i);
13
14         switch (n.getNodeType ()) {
15             case DOM_Node::TEXT_NODE:
16                 cout << n.getNodeValue ().transcode ();
17                 break;
18             case DOM_Node::ELEMENT_NODE:
19                 collect (n.getChildNodes ());
20         }
21     }
22 }
```

```
23
24 void content (DOM_Document d)
25 {
26     collect (d.getChildNodes ());
27 }
28
29 int main (void)
30 {
31     XMLPlatformUtils::Initialize ();
32
33     DOMParser parser;
34     DOM_Document doc;
35
36     parser.parse ("foo.xml");
37     doc = parser.getDocument ();
38
39     content (doc);
40
41     return 0;
42 }
```

Now: Find all occurrences of Dogbert speaking (attribute speaker of element bubble) ...

dogbert.cc (1)

```
1 // Xerces C++ DOM API support
2 #include <dom/DOM.hpp>
3 #include <parsers/DOMParser.hpp>
4
5 void dogbert (DOM_Document d)
6 {
7     DOM_NodeList    bubbles;
8     DOM_Node        bubble, speaker;
9     DOM_NamedNodeMap attrs;
10
11     bubbles = d.getElementsByTagName ("bubble");
12
13     for (unsigned long i = 0; i < bubbles.getLength (); i++) {
14         bubble = bubbles.item (i);
15
16         attrs = bubble.getAttributes ();
17         if (attrs != 0)
18             if ((speaker = attrs.getNamedItem ("speaker")) != 0)
19                 if (speaker.getNodeValue ().
20                     compareString (DOMString ("Dogbert")) == 0)
21                     cout << "Found Dogbert speaking." << endl;
22     }
23 }
```

dogbert.cc (2)

```
24
25 int main (void)
26 {
27     XMLPlatformUtils::Initialize ();
28
29     DOMParser parser;
30     DOM_Document doc;
31
32     parser.parse ("foo.xml");
33     doc = parser.getDocument ();
34
35     dogbert (doc);
36
37     return 0;
38 }
```

DOM—A Memory Bottleneck

- The two-step processing approach (① parse and construct XML tree, ② respond to DOM property function calls) enables the DOM to be “**random access**”:

The XML application may explore and update any portion of the XML tree at any time.

- The inherent memory hunger of the DOM may lead to
 - ① heavy **swapping** activity
(partly due to unpredictable memory access patterns, `madvise()` less helpful)
or
 - ② even “out-of-memory” failures.
(The application has to be extremely careful with its own memory management, the very least.)

Numbers





DOM and random node access

Even if the application touches a single element node only, the DOM API has to maintain a data structure that represents the **whole XML input document** (all sizes in kB):⁶

XML size	DOM process size DSIZ	$\frac{\text{DSIZ}}{\text{XML size}}$	Comment
7480	47476	6.3	(Shakespeare's works) many elements containing small text fragments
113904	552104	4.8	(Synthetic eBay data) elements containing relatively large text fragments

⁶The random access nature of the DOM makes it hard to provide a truly “lazy” API implementation.

To remedy the memory hunger of DOM-based processing ...

- Try to **preprocess** (*i.e.*, filter) the input XML document to reduce its overall size.
 - ▶ Use an XPath/XSLT processor to preselect *interesting* document regions,
 - ▶  *no updates* to the input XML document are possible then,
 - ▶  make sure the XPath/XSLT processor is *not* implemented on top of the DOM.

Or

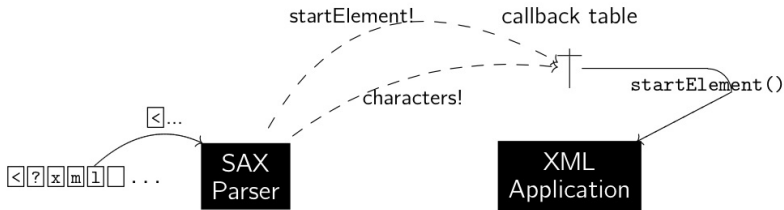
- Use a **completely different** approach to XML processing (→ **SAX**).


SAX—Simple API for XML

- **SAX⁷ (Simple API for XML)** is, unlike DOM, *not* a W3C standard, but has been developed jointly by members of the XML-DEV mailing list (*ca.* 1998).
- SAX processors use **constant space**, regardless of the XML input document size.
 - ▶ Communication between the SAX processor and the backend XML application does *not* involve an intermediate tree data structure.
 - ▶ Instead, the **SAX parser sends events** to the application whenever a certain piece of XML text has been recognized (*i.e.*, parsed).
 - ▶ The **backend acts on/ignores events** by populating a **callback function table**.

⁷<http://www.saxproject.org/>

Sketch of SAX's mode of operations



- A SAX processor reads its input document **sequentially** and **once** only.
- No memory of what the parser has seen so far is retained while parsing. As soon as a  *significant bit of XML text* has been recognized, an **event** is sent.
- The application is able to act on events **in parallel** with the parsing progress.

SAX Events

- To meet the constant memory space requirement, SAX reports **fine-grained parsing events** for a document:

Event	... reported when seen	Parameters sent
<i>startDocument</i>	<code><?xml...?></code> ⁸	
<i>endDocument</i>	<code><EOF></code>	
<i>startElement</i>	<code><t a₁=v₁ ... a_n=v_n></code>	$t, (a_1, v_1), \dots, (a_n, v_n)$
<i>endElement</i>	<code></t></code>	t
<i>characters</i>	<i>text content</i>	Unicode buffer ptr, length
<i>comment</i>	<code><!--c--></code>	c
<i>processingInstruction</i>	<code><?t pi?></code>	t, pi
	⋮	

⁸**N.B.:** Event *startDocument* is sent even if the optional XML text declaration should be missing.

dilbert.xml

```
1 <?xml encoding="utf-8"?> *1
2 <bubbles> *2
3   <!-- Dilbert looks stunned --> *3
4   <bubble speaker="phb" to="dilbert"> *4
5     Tell the truth, but do it in your usual engineering way
6     so that no one understands you. *5
7   </bubble> *6
8 </bubbles> *7 *8
```

Event ⁹ ¹⁰	Parameters sent
*1	<i>startDocument</i>
*2	<i>startElement</i> <i>t</i> = "bubbles"
*3	<i>comment</i> <i>c</i> = "_Dilbert looks stunned_"
*4	<i>startElement</i> <i>t</i> = "bubble", ("speaker","phb"), ("to","dilbert")
*5	<i>characters</i> <i>buf</i> = "Tell the...understands you.", <i>len</i> = 99
*6	<i>endElement</i> <i>t</i> = "bubble"
*7	<i>endElement</i> <i>t</i> = "bubbles"
*8	<i>endDocument</i>

⁹Events are reported in **document reading order** *1, *2, ..., *8.

¹⁰**N.B.:** Some events suppressed (white space).

SAX Callbacks

- To provide an efficient and tight **coupling** between the SAX **frontend** and the application **backend**, the SAX API employs **function callbacks**.¹¹
 - 1 Before parsing starts, the application **registers function references** in a table in which each event has its own slot:

Event	Callback		Event	Callback
⋮			⋮	
<i>startElement</i>	?	— →	<i>startElement</i>	<i>startElement ()</i>
<i>endElement</i>	?	<i>SAX register(startElement,</i>	<i>endElement</i>	<i>endElement ()</i>
⋮		<i>startElement ())</i>	⋮	
⋮		<i>SAX register(endElement,</i>	⋮	
⋮		<i>endElement ())</i>	⋮	

- 2 The application alone decides on the implementation of the functions it registers with the SAX parser.
- 3 **Reporting an event** \star_i ; then amounts to call the function (with parameters) registered in the appropriate table slot.

¹¹Much like in event-based GUI libraries.



Java SAX API

In Java, populating the callback table is done via implementation of the SAX `ContentHandler` interface: a `ContentHandler` object represents the callback table, its methods (e.g., `public void endDocument ()`) represent the table slots.

Example: Reimplement `content.cc` shown earlier for DOM (find all XML text nodes and print their content) using SAX (pseudo code):

`content (File f)`

```
// register the callback,  
// we ignore all other events  
SAXregister (characters, printText);  
SAXparse (f);  
return;
```

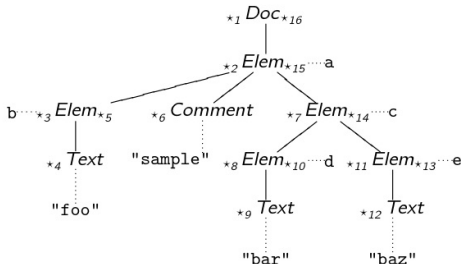
`printText ((Unicode) buf, Int len)`

```
Int i;  
foreach  $i \in 1 \dots len$  do  
   $\lfloor$  print (buf[i]);  
return;
```

SAX and the XML Tree Structure

- Looking closer, the **order** of SAX events reported for a document is determined by a **preorder traversal** of its document tree¹²:

```
Sample XML document
1  *1
2  <a>*2
3    <b>*3 foo*4 </b>*5
4    <!--sample-->*6
5    <c>*7
6      <d>*8 bar*9 </d>*10
7      <e>*11 baz*12 </e>*13
8    </c>*14
9  </a>*15 *16
```



N.B.: An *Elem* [*Doc*] node is associated with two SAX events, namely *startElement* and *endElement* [*startDocument*, *endDocument*].

¹²Sequences of sibling *Char* nodes have been collapsed into a single *Text* node.

Challenge

- This **left-first depth-first** order of SAX events is well-defined, but appears to make it hard to answer certain queries about an XML document tree.

 Collect all direct children nodes of an *Elem* node.

In the example on the previous slide, suppose your application has just received the *startElement*(*t* = "a") event \star_2 (*i.e.*, the parser has just parsed the opening element tag <a>).

With the remaining events $\star_3 \dots \star_{16}$ still to arrive, can your code detect all the immediate children of *Elem* node a (*i.e.*, *Elem* nodes b and c as well as the *Comment* node)?

The previous question can be answered more generally:

*SAX events are sufficient to **rebuild the complete XML document tree** inside the application. (Even if we most likely don't want to.)*

SAX-based tree rebuilding strategy (sketch):

- 1 `[startDocument]`
Initialize a **stack** S of **node IDs** (e.g. $\in \mathbb{Z}$). **Push** first ID for this node.
- 2 `[startElement]`
Assign a **new ID** for this node. **Push** the ID onto S .¹³
- 3 `[characters, comment, ...]`
Simply assign a new node ID.
- 4 `[endElement, endDocument]`
Pop S (no new node created).



Invariant: The **top of** S holds the identifier of the current **parent node**.

¹³In callbacks ② and ③ we might wish to store further node details in a table or similar summary data structure.

Final Remarks on SAX

- For an XML document fragment shown on the left, SAX might actually report the events indicated on the right:

XML fragment	XML + SAX events
1 <affiliation>	1 <affiliation>* ₁
2 AT&T Labs	2 AT* ₂ &* ₃ T Labs
3 </affiliation>	3 * ₄ </affiliation>* ₅

* ₁	<i>startElement</i> (affiliation)
* ₂	<i>characters</i> ("\\n\\tAT", 5)
* ₃	<i>characters</i> ("&", 1)
* ₄	<i>characters</i> ("T\\tLabs\\n", 7)
* ₅	<i>endElement</i> (affiliation)



White space is reported.

Multiple *characters* events may be sent for text content (although adjacent).

(Often SAX parsers break text on entities, but may even report each character on its own.)

Summary

We have seen:

- Motivation for XML (where XML originates from and what it is aimed for)
 - How the XML **meta-language** works
 - What is new/important with XML (standard, independence from processors, well-formed data can be processed, users can agree on a type and save coding effort, they can exchange valid data + the type(s)...)
 - ... and what is less (markup syntax details, verbosity)
 - How to **define your own XML dialect** (using DTD or XML Schema)
 - How/when to use the 2 different kinds of XML parsers (DOM, SAX)
- Welcome to the world of XML!...