# EDA095
# eXtensible Markup Language

Pierre Nugues

Lund University
http://cs.lth.se/pierre_nugues/

April 7, 2011

# Character Sets

Codes are used to represent characters.
ASCII has 0 to 127 code points and is only for English
Latin-1 extends it to 256 code points. It can be used for most Western European languages but forgot many characters, like the French Œ, œ, the German quote ,, or the Dutch IJ, ij.
It is not adopted by all the operating systems, MacOS for instance

# Unicode

Unicode is an attempt to represent most alphabets.
From *Programming Perl* by Larry Wall,Tom Christiansen,Jon Orwant, O'Reilly, 2000:

> *If you don't know yet what Unicode is, you will soon—even if you skip reading this chapter—because working with Unicode is becoming a necessity.*

It started with 16 bits and has now uses 32 bits.
The standard character representation in many OSes and programming languages, including Java
Characters have a code point and a name as

```
U+0042 LATIN CAPITAL LETTER B
U+0391 GREEK CAPITAL LETTER ALPHA
U+00C5 LATIN CAPITAL LETTER A WITH RING ABOVE
```

# Unicode Blocks (Simplified)

| Code | Name | Code | Name |
|---|---|---|---|
| U+0000 | Basic Latin | U+1400 | Unified Canadian Aboriginal Syllab |
| U+0080 | Latin-1 Supplement | U+1680 | Ogham, Runic |
| U+0100 | Latin Extended-A | U+1780 | Khmer |
| U+0180 | Latin Extended-B | U+1800 | Mongolian |
| U+0250 | IPA Extensions | U+1E00 | Latin Extended Additional |
| U+02B0 | Spacing Modifier Letters | U+1F00 | Extended Greek |
| U+0300 | Combining Diacritical Marks | U+2000 | Symbols |
| U+0370 | Greek | U+2800 | Braille Patterns |
| U+0400 | Cyrillic | U+2E80 | CJK Radicals Supplement |
| U+0530 | Armenian | U+2F80 | KangXi Radicals |
| U+0590 | Hebrew | U+3000 | CJK Symbols and Punctuation |
| U+0600 | Arabic | U+3040 | Hiragana, Katakana |
| U+0700 | Syriac | U+3100 | Bopomofo |
| U+0780 | Thaana | U+3130 | Hangul Compatibility Jamo |

# Unicode Blocks (Simplified) (II)

| Code | Name | Code | Name |
|---|---|---|---|
| U+0900 | Devanagari, Bengali | U+3190 | Kanbun |
| U+0A00 | Gurmukhi, Gujarati | U+31A0 | Bopomofo Extended |
| U+0B00 | Oriya, Tamil | U+3200 | Enclosed CJK Letters and Months |
| U+0C00 | Telugu, Kannada | U+3300 | CJK Compatibility |
| U+0D00 | Malayalam, Sinhala | U+3400 | CJK Unified Ideographs Extension A |
| U+0E00 | Thai, Lao | U+4E00 | CJK Unified Ideographs |
| U+0F00 | Tibetan | U+A000 | Yi Syllables |
| U+1000 | Myanmar | U+A490 | Yi Radicals |
| U+10A0 | Georgian | U+AC00 | Hangul Syllables |
| U+1100 | Hangul Jamo | U+D800 | Surrogates |
| U+1200 | Ethiopic | U+E000 | Private Use |
| U+13A0 | Cherokee | U+F900 | Others |

# The Unicode Encoding Schemes

Unicode offers three different encoding schemes: UTF-8, UTF-16, and UTF-32.
UTF-16 was the standard encoding scheme.
It uses fixed units of 16 bits – 2 bytes –
*FÊTE* `0046 00CA 0054 0045`
UTF-8 is a variable length encoding.
It maps the ASCII code characters U+0000 to U+007F to their byte values 0x00 to 0x7F.
All the other characters in the range U+007F to U+FFFF are encoded as a sequence of two or more bytes.

# UTF-8

| Range | Encoding |
|---|---|
| U-0000 – U-007F | 0xxxxxxx |
| U-0080 – U-07FF | 110xxxxx 10xxxxxx |
| U-0800 – U-FFFF | 1110xxxx 10xxxxxx 10xxxxxx |
| U-010000 – U-10FFFF | 11110xxx 10xxxxxx 10xxxxxx 10xxxxxx |

# Encoding FÊTE in UTF-8

The letters F, T, and E are in the range U-00000000..U-0000007F.
Ê is `U+00CA` and is in the range U-00000080..U-000007FF.
Its binary representation is 0000 0000 1100 1010.
UTF-8 uses the eleven rightmost bits of 00CA.
The first five underlined bits together with the prefix 110 form the octet 1100 0011 that corresponds to `C3` in hexadecimal.
The seven next boldface bits with the prefix 10 form the octet 1000 1010 or `8A` in hexadecimal.
The letter Ê is encoded or `C3 8A` in UTF-8.
FÊTE and the code points U+0046 U+00CA U+0054 U+0045 are encoded as `46 C3 8A 54 45`

# Locales and Word Order

Depending on the language, dates, numbers, time is represented differently:
Numbers: 3.14 or 3,14?
Time: 01/02/03

- 3 februari 2001?
- January 2, 2003?
- 1 February 2003?

Collating strings: is Andersson before or after Åkesson?

# The Unicode Collation Algorithm

The Unicode consortium has defined a collation algorithm that takes into account the different practices and cultures in lexical ordering. It has three levels for Latin scripts:

- The primary level considers differences between base characters, for instance between A and B.
- If there are no differences at the first level, the secondary level considers the accents on the characters.
- And finally, the third level considers the case differences between the characters.

# Differences

These level features are general, but not universal.
Accents are a secondary difference in many languages but Swedish sorts accented letters as individual ones and hence sets a primary difference between A and Å or O and Ö.

1. First level: {a, A, á, Á, à, À, etc.} < {b, B} < {c, C, ć, Ć, ĉ, Ĉ, ç, Ç, etc.} < {e, E, é, É, è, È, ê, Ê, ë, Ë, etc.} < ...
2. Second level: {e, E} << {é, É} << {è, È} << {ê, Ê} << {ë, Ë}
3. Third level: {a} <<< {A}

The comparison at the second level is done from the left to the right of a word in English, the reverse in French.

# Sorting Words in French and English

| English | French |
|---|---|
| *Péché* | *pèche* |
| *PÉCHÉ* | *pêche* |
| *pèche* | *Pêche* |
| *pêche* | *Péché* |
| *Pêche* | *PÉCHÉ* |
| *pêché* | *pêché* |
| *Pêché* | *Pêché* |
| *pécher* | *pécher* |
| *pêcher* | *pêcher* |

# Collation Demonstration

IBM has implemented open source classes to handle locales
It provides a collation demonstration here:
http://www.ibm.com/software/globalization/icu/
The Java package can be downloaded from
http://site.icu-project.org/

# Markup Languages

Markup languages are used to annotate texts with a structure and a presentation

Annotation schemes used by word processors include LaTex, RTF, etc.

XML, which resembles HTML, is now a standard annotation and exchange language

XML is a coding framework: a language to define ways of structuring documents.

XML is also used to create tabulated data (database-compatible data)

# XML

XML uses plain text and not binary codes.
It separates the definition of structure instructions from the content – the data.
Structure instructions are described in a document type definition (DTD) that models a class of XML documents.
Document type definitions contain the specific tagsets to mark up texts.
A DTD lists the legal tags and their relationships with other tags.
XML has APIs available in many programming languages: Java, Perl, SWI Prolog, etc.

# XML Elements

A DTD is composed of three kinds of components: elements, attributes, and entities.
The elements are the logical units of an XML document.
A DocBook-like description (http://www.docbook.org/)

```
<!-- My first XML document -->
<book>
  <title>Network Processing Cookbook</title>
  <author>Pierre Cagné</author>

  <!-- The image to show on the cover -->
  <img></img>
  <text>Here comes the text!</text>
</book>
```

# Differences with HTML

XML tags must be balanced, which means that an end tag must follow each start tag.
Empty elements `<img></img>` can be abridged as `<img/>`.
XML tags are case sensitive: `<TITLE>` and `<title>` define different elements.
An XML document defines one single root element that spans the document, here `<book>`

# XML Attributes

An element can have attributes, i.e. a set of properties.
A `<title>` element can have an alignment: flush left, right, or center, and a character style: underlined, bold, or italics.
Attributes are inserted as name-value pairs in the start tag

```
<title align="center" style="bold">
  Network Processing Cookbook
</title>
```

# Entities

Entities are data stored somewhere in a computer that have a name.
They can be accented characters, symbols, strings as well as text or image files.
An entity reference is the entity name enclosed by a start delimiter & and an end delimiter ; such as `&EntityName;`
The entity reference will be replaced by the entity.
Useful entities are the predefined entities and the character entities

# Entities (II)

There are five predefined entities recognized by XML.
They correspond to characters used by the XML standard, which can't be used as is in a document.

| Symbol | Entity encoding | Meaning |
|---|---|---|
| < | < | Less than |
| > | > | Greater than |
| & | & | Ampersand |
| " | " | Quotation mark |
| ' | ' | Apostrophe |

A character reference is the Unicode value for a single character such as Ê for Ê (or Ê)

# Writing a DTD: Elements

A DTD specifies the formal structure of a document type.
A DTD file contains the description of all the legal elements, attributes, and entities.
The description of the elements is enclosed between the delimiters `<!ELEMENT` and `>`.

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
```

# Character Types

| Character type | Description |
|---|---|
| PCDATA | Parsed character data. This data will be parsed and must only be text, punctuation, and special characters; no embedded elements |
| ANY | PCDATA or any DTD element |
| EMPTY | No content – just a placeholder |

# Writing a DTD: Attributes

Attributes are the possible properties of the elements.
Their description is enclosed between the delimiters <!ATTLIST and >.

```
<!ATTLIST title
  style (underlined | bold | italics) "bold"
  align (left | center | right) "left">
```

# Some XML Attribute Types

| Attribute types | Description |
|---|---|
| CDATA | The string type: any character except <, >, &, ', and " |
| ID | An identifier of the element unique in the document; ID must begin with a letter, an underscore, or a colon |
| IDREF | A reference to an identifier |
| NMTOKEN | String of letters, digits, periods, underscores, hyphens, and colons. It is more restrictive than CDATA, for instance, spaces are not allowed |

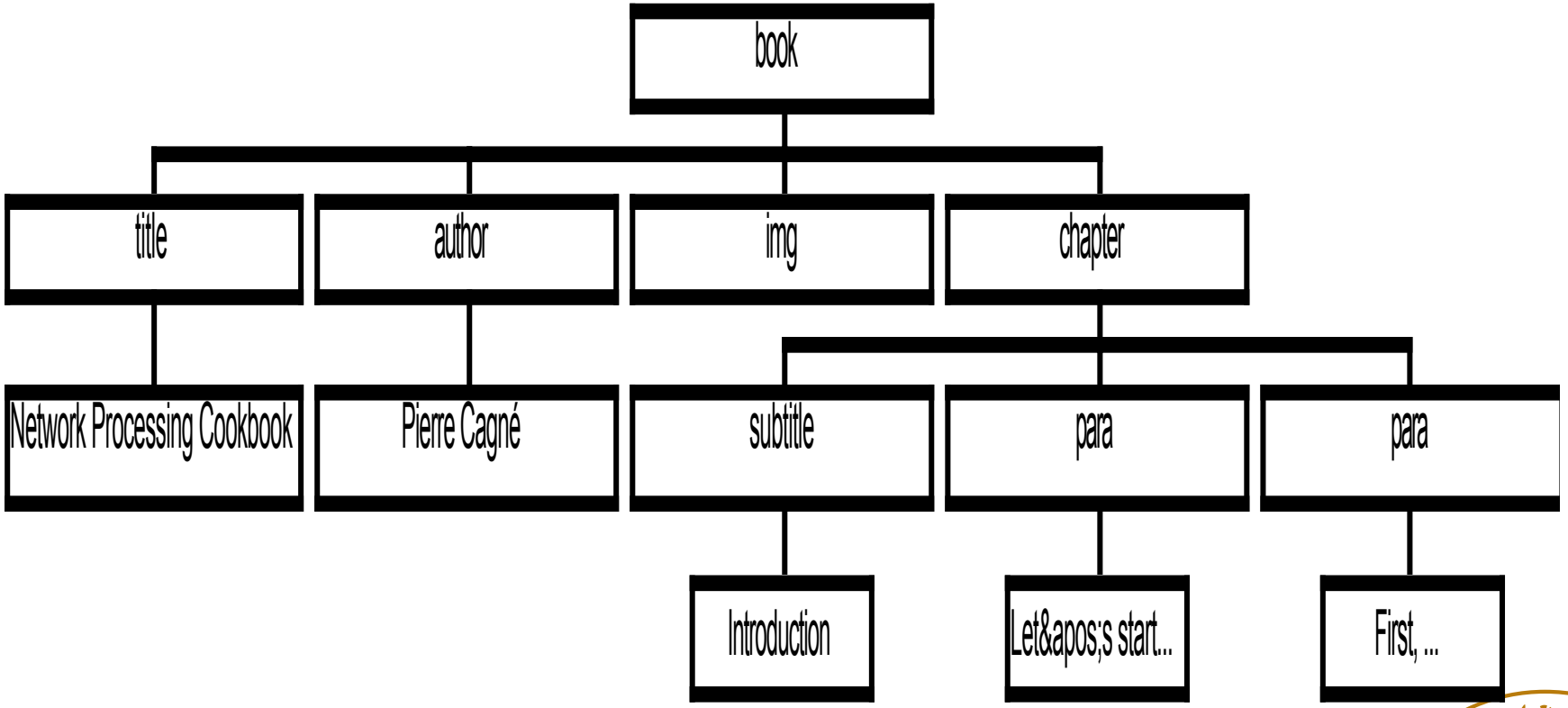
# Some Default Value Keywords

| Predefined default values | Description |
|---|---|
| `#REQUIRED` | A value must be supplied |
| `#FIXED` | The attribute value is constant and must be equal to the default value |
| `#IMPLIED` | If no value is supplied, the processing system will define the value |

# Writing an XML Document

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE book [
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
...
]>
<book>
  <title style="i">Network Processing Cookbook</title>
  <author style="b">Pierre Cagné</author>
  <img src="pierre.jpg"/>
  <chapter number="c1">
    <subtitle>Introduction</subtitle>
    <para>Let&apos;s start doing simple things: collect texts.
    </para>
    <para>First, choose a site you like</para>
  </chapter>
</book>
```

# Tree Representation



# The DTD

```
<!ELEMENT book (title, (author | editor)?, img, chapter+)>
<!ELEMENT title (#PCDATA)>
<!ATTLIST title style (u | b | i) "b">
<!ELEMENT author (#PCDATA)>
<!ATTLIST author style (u | b | i) "i">
<!ELEMENT editor (#PCDATA)>
<!ATTLIST editor style (u | b | i) "i">
<!ELEMENT img EMPTY>
<!ATTLIST img src CDATA #REQUIRED>
<!ELEMENT chapter (subtitle, para+)>
<!ATTLIST chapter number ID #REQUIRED>
<!ATTLIST chapter numberStyle (Arabic | Roman) "Roman">
<!ELEMENT subtitle (#PCDATA)>
<!ELEMENT para (#PCDATA)>
```

# DTD Address

The document declares the DTD it uses. It can be inside the XML document and enclosed between the delimiters `<!DOCTYPE [` and `]>`:

```
<!DOCTYPE book [
...
]>
```

or outside

```
<!DOCTYPE book SYSTEM
  "file://localhost/home/pierre/xml/book_definition.dtd">
<!DOCTYPE book SYSTEM
  "file:///home/pierre/xml/book_definition.dtd">
```

# Validation

A document is **well formed** when it has no syntax errors: the brackets are balanced; the encoding is correct; etc.

A document is **valid** when it conforms to the DTD

There are many validation services, use for example the W3C: `http://validator.w3.org/`.

Netbeans can also check the validity of documents

Examples with: `MyBook.xml`, `MyBook2.xml`, `book_definition.dtd`

# Name Spaces

The element names we have used can conflict with names in other documents, title for instance, that is used by HTML
XML namespaces are a naming scheme that makes names local.
They use the `xmlns` attribute
We apply it to an element and all its children:

```
<title xmlns="http://cs.lth.se/pierre">
   Network Processing Cookbook
</title>
```

The URI is just a unique name. It is not accessed.

# Name Spaces (II)

We can use name spaces as tag prefixes
We define the prefix as `<element xmlns:prefix="URI">`
The prefix enables to mix names without collision

```
<book xmlns:pierre="http://cs.lth.se/pierre/"
    xmlns:mathias="http://www.svt.se/">
  <pierre:title style="i">
    Language Processing Cookbook
  </pierre:title>

  <mathias:title style="i">
    A Swedish Cookbook
  </mathias:title>
```

# XHTML

XHTML is a clean, XML compatible HTML. It reformulates HTML 4 (http://www.w3.org/TR/xhtml1/)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html
     PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN"
    "http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xml:lang="en" lang="en">
  <head>
    <title>Virtual Library</title>
  </head>
  <body>
    <p>Moved to <a href="http://example.org/">example.org</a>.
    </p>
  </body>
</html>
```

# XML Schemas

The combination of two different grammar styles to describe a DTD and an XML document is not flexible
In addition, DTDs have a weak expressive power.
XML Schemas are an alternative choice to DTDs (`http://www.w3.org/XML/Schema.html`). The main differences are:

- They use an XML-based syntax
- They use data types for elements and attributes
- They can specify the occurrence numbers instead of just +, *, or ?
- Schemas are increasingly replacing DTDs
- Schemas are more complex and verbose

# Schema Example

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs='http://www.w3.org/2001/XMLSchema'>
  <xs:element name='book'>
    <xs:complexType>
      <xs:sequence>
        <xs:element ref='title'/>
        <xs:choice minOccurs='0' maxOccurs='1'>
          <xs:element ref='author'/>
          <xs:element ref='editor'/>
        </xs:choice>
        <xs:element ref='img'/>
        <xs:element ref='chapter' maxOccurs='unbounded'/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
```

# Schema Example (II)

```
<xs:element name='title'>
  <xs:complexType>
    <xs:simpleContent>
      <xs:extension base='xs:string'>
        <xs:attribute name='style' use='optional' default='b'>
          <xs:simpleType>
            <xs:restriction base='xs:string'>
              <xs:enumeration value='u'/>
              <xs:enumeration value='b'/>
            </xs:restriction>
          </xs:simpleType>
        </xs:attribute>
      </xs:extension>
    </xs:simpleContent>
  </xs:complexType>
</xs:element>
...
<xs:element name='subtitle' type='xs:string'/>
<xs:element name='para' type='xs:string'/>
```

# Text Annotation

Many text and language processing tools use XML:

- Documents from `http://xml.openoffice.org/` are interesting to read;
- See also `http://www.ecma-international.org/publications/standards/Ecma-376.htm`;
- See also EPUB: `http://idpf.org/epub`.

# Generic Parsers for XML

There is large set of tools to parse XML documents
Many of them are available from Java.
SAX and DOM are the two main parser families:

- DOM, the *document object model*, is a full-fledged parsing interface that enables the programmer to interact with a hierarchical parse tree obtained from a document.
- DOM is the official recommendation of W3C
- SAX, Simple API for XML, is a low level parsing mechanism.
- SAX is fast

# XML DOM

DOM parsers are available in many languages (including Java).
A DOM parser builds a complete parse tree that resides in memory.
You can then easily traverse the tree, access the nodes, and modify them.
Specifications are available here:
`http://www.w3.org/TR/DOM-Level-3-Core/`
Many tools enable a programmer to generate programs automatically to visit a tree

# XML Sax

SAX is an event-oriented parser.
It scans the text and once it encounters an element, it executes something.
It resembles the built-in Java HTML parser `HTMLEditorKit.Parser`
It does not build a tree and hence is less convenient (to me).
SAX is useful if you have strict speed and memory requirements
As for DOM, many tools enable to generate automatically programs to visit a tree

# The DOM API in Java

The DOM package is `org.w3c.dom.*`
The reference on Java bindings is available here:
`http://www.w3.org/TR/DOM-Level-3-Core/java-binding.html`
Key interfaces are:

- `Document`: a hierarchical tree,
- `Node`: the nodes of the tree.
- `NodeList`: a list of `Nodes`
- `Element`, `Text`, `Attr`: subclasses derived from `Node`

# Building a Tree

```
DocumentBuilderFactory factory =
  DocumentBuilderFactory.newInstance();
factory.setValidating(true);
Document document = null;
try {
  DocumentBuilder parser = factory.newDocumentBuilder();
  document =
    parser.parse(new File("src/data/MyBook3.xml"));
} catch (Exception e) {
  e.printStackTrace();
}
```

`document` is a DOM `Document` object (a parse tree)

# Traversing a Tree

We traverse the tree using the `Document` or `Node` methods

We get the root with

```
Element root = document.getDocumentElement();
```

We obtain the nodes using `getChildNodes()`, `getFirstChild()`, and `getLastChild()`.

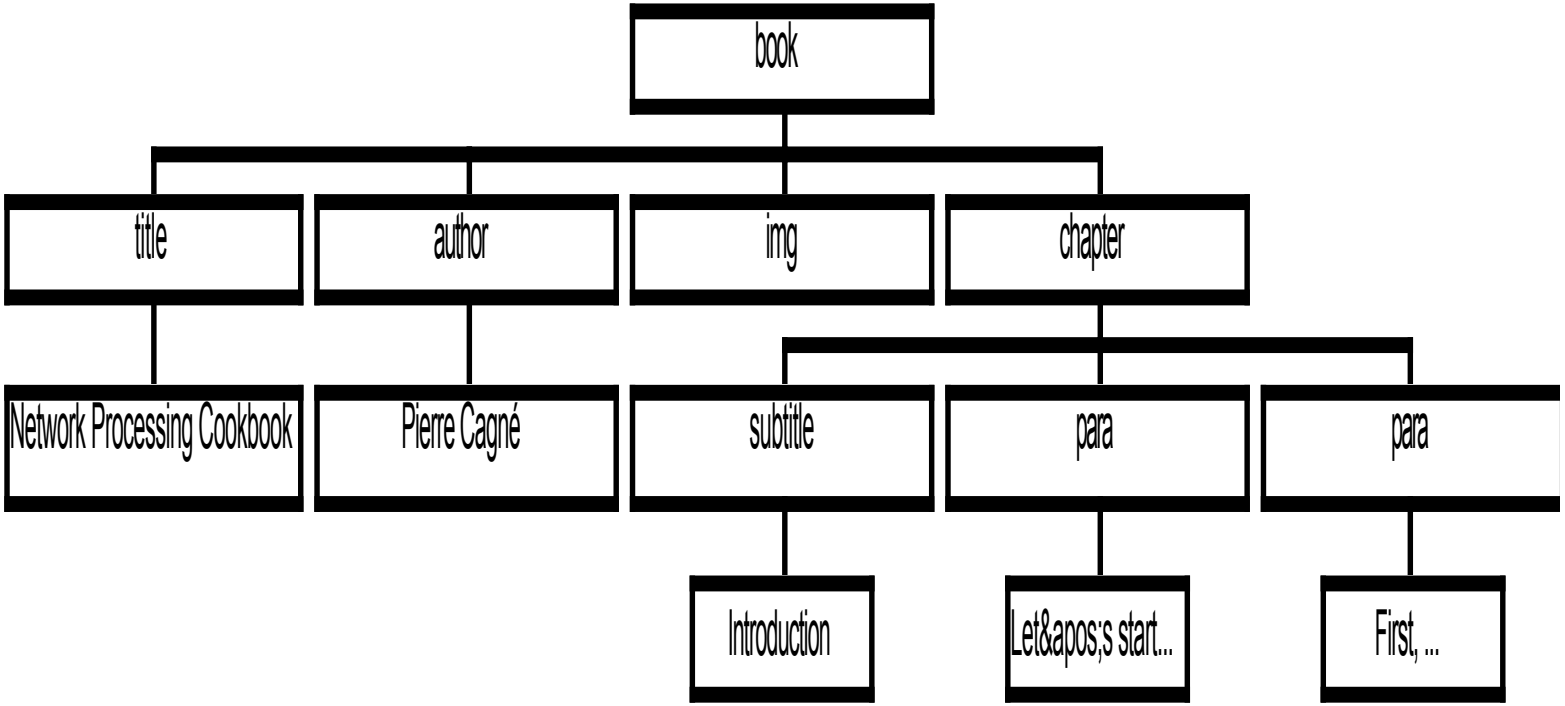
Or `getElementsByTagName(String name)`

We use `item(int number)` to access a `NodeList`

We access and set them using `getNodeValue()` and `setNodeValue()`

We obtain the attributes using `getAttributes()` or `getAttribute(String name)`

# Tree


book
title
author
img
chapter
Network Processing Cookbook
Pierre Cagné
subtitle
para
para
Introduction
Let's start...
First, ...

# Printing the Content of the Chapters

(FirstDOM.java)

```
public class FirstDOM {
public static void main(String[] args) {
  DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
  factory.setValidating(true);
  Document document = null;
  try {
    DocumentBuilder parser = factory.newDocumentBuilder();
    document = parser.parse(new File("src/xml/MyBook3.xml"));
  } catch (Exception e) {
    e.printStackTrace();
  }
  Element root = document.getDocumentElement();
  NodeList chapters = root.getElementsByTagName("chapter");
```

# Printing the Content of the Chapters (II)

(FirstDOM.java)

```
  for( int i = 0; i < chapters.getLength(); i++ ) {
    System.out.println("Chapter: " +
      ((Element) chapters.item(i)).getAttribute("number"));
    NodeList content = (chapters.item(i)).getChildNodes();
    System.out.println("Number of nodes: " +
      content.getLength());
      for (int j = 0; j < content.getLength(); j++) {
        System.out.println(j + ": " +
          (content.item(j)).getChildNodes());
      }
    }
  }
}
```

# Node Types

The Node class gives the list of possible node types.
We extract them using getNodeType()

| | | |
|---|---|---|
| ELEMENT_NODE | Element | 1 |
| ATTRIBUTE_NODE | Attr | 2 |
| TEXT_NODE | Text node | 3 |
| CDATA_SECTION_NODE | CDATASection | 4 |
| ENTITY_REFERENCE_NODE | EntityReference | 5 |
| ENTITY_NODE | Entity | 6 |
| PROCESSING_INSTRUCTION_NODE | ProcessingInstruction | 7 |
| COMMENT_NODE | Comment | 8 |
| DOCUMENT_NODE | Document | 9 |
| DOCUMENT_TYPE_NODE | DocumentType | 10 |
| DOCUMENT_FRAGMENT_NODE | DocumentFragment | 11 |
| NOTATION_NODE | Notation | 12 |

# Printing the Text Only

We can use the type for a very simple nonrecursive program: (SecondDOM.java)

```
switch ((content.item(j)).getNodeType()) {
  case org.w3c.dom.Node.CDATA_SECTION_NODE:
    System.out.println("CDATA");
    System.out.println(((org.w3c.dom.CDATASection)
      (content.item(j))).getData());
    break;
  case org.w3c.dom.Node.ELEMENT_NODE:
    System.out.println("ELEMENT");
    org.w3c.dom.Element nodeElement =
      (org.w3c.dom.Element)(content.item(j));
    NodeList t = nodeElement.getChildNodes();
    for (int k = 0; k < t.getLength(); k++) {
      System.out.println((t.item(k)).getNodeValue());
    }
    break;
```

# Printing the Text Only (II)

```
case org.w3c.dom.Node.PROCESSING_INSTRUCTION_NODE:
    System.out.println("INSTR");
...
}
```

# Generating a Program to Traverse the Tree

Writing program to traverse a tree can be tedious and repetitive
In Java, tools such as NetBeans, will generate a template program from the DTD to visit the tree and access the nodes.
You just have to fill in your code in the template methods
Use the menu item: "Select Generate DOM Tree Scanner."
The program traverses the whole document...
(Main.java)
(Book_definitionScanner.java)

# Transformations

Parsers can be used to transform and generate documents
A basic generation is just to read the document, parse it, and generate it from the tree.

```
DocumentBuilder parser = factory.newDocumentBuilder();
document = parser.parse(new File("src/xml/MyBook3.xml"));
TransformerFactory tFactory =
  TransformerFactory.newInstance();
Transformer transformer = tFactory.newTransformer();
Source source = new DOMSource(document);
Result result = new StreamResult(System.out);
transformer.transform(source, result);
```

# Extensible Stylesheet Language (XSL)

We can apply transformations using the extensible stylesheet language (XSL).
XSLT traverses recursively an XML tree from the root to the leaves and visits all the nodes
XSL is a full-fledged language with loops, conditionals, etc.
XSLT has directives to match nodes of an XML document and generates an output using:

```
<xsl:template match="...">
```

When traversing the tree, at a certain depth, if there is no rule that match the node, XSLT just outputs the text of the corresponding subtree
XSLT uses the XML path language – XPath – to name nodes and attributes in a document

# XPath

XPath is a language to name and access nodes in an XML tree:
`http://www.w3.org/TR/xpath20/`
Inspired by Unix paths: reuses the `/`, `.` and `..` symbols
Naming nodes in MyBook3.xml

| | |
|---|---|
| `title` | All the title elements children of the context node |
| `*/title` | All the title elements grandchildren of the context node |
| `/book/title` | All the title elements children of the book root node |
| `//para` | All the `para` elements at any depth |
| `chapter[para]` | All the chapter elements children of the context node that have a `para` child (Boolean predicate) |
| `chapter[@number="c1"]` | All the chapter elements children of the context node that have a `number` attribute with value c |

# XPath: A First Example

MyBook1.xsl

The processing stops after it has reached the book element

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/book">
  <html>
    <head>
      <title>Pierre's Book</title>
    </head>
    <body>
      <h1>Book</h1>
    </body>
  </html>
</xsl:template>
```

# XPath: A Second Example

MyBook2.xsl The processing continues and outputs the text

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/book">
  <html>
    <head>
      <title>Pierre's Book</title>
    </head>
    <body>
      <h1>Book</h1>
      <xsl:apply-templates/>
    </body>
  </html>
</xsl:template>
```

# Extensible Stylesheet Language (XSL)

Other instructions:

- Apply templates to a subset of children of the current node using `<xsl:apply-templates select="..."/>`
- Print the content of an element using `<xsl:value-of select="..."/>`
- The current element is denoted `<xsl:value-of select="."/>`
- Loops using `<xsl:for-each select="..."> ... </xsl:for-each>`

# XSL: Traversing a Subtree

MyBook3.xsl The processing continues and outputs the text

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/book">
  <html>
    <head>
      <title>Pierre's Book</title>
    </head>
    <body>
      <h1>Book</h1>
      <xsl:apply-templates select="chapter"/>
    </body>
  </html>
</xsl:template>
```

# XSL: Printing Attribute Values

MyBook4.xsl The processing outputs the chapter number and the text

```
<xsl:template match="chapter">
  <p>
    <xsl:value-of select="@number"/>
  </p>
  <xsl:apply-templates/>
</xsl:template>
```

# XSL: Printing Text

MyBook5.xsl The processing outputs the para content

```
<xsl:template match="chapter">
  <xsl:for-each select="para">
    <p>
      <xsl:value-of select="."/>
    </p>
  </xsl:for-each>
</xsl:template>
```

# Extensible Stylesheet Language (XSL)

```
<xsl:stylesheet
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  version="1.0">
<xsl:template match="/book">
  <html>
    <head><title>Pierre's Book</title></head>
    <body><h1>Book</h1>
      <table border="1">
        <tr>
          <td><b>Title</b></td><td><b>Author</b></td>
        </tr>
        <tr>
          <td>
            <xsl:value-of select="title"/>
          </td>
          <td><xsl:value-of select="author"/></td>
        </tr>
```

# Extensible Stylesheet Language (XSL)

```
...
      </table>
      <h1>Content</h1>
      <table border="1">
        <tr>
          <td><b>Chapter</b></td><td><b>Text</b></td>
        </tr>
      </table>
    </body>
  </html>
</xsl:template>
```

# Extensible Stylesheet Language (XSL)

Other instructions: `apply-templates` and `for-each`

```
<xsl:apply-templates select="chapter"/>
...
<xsl:template match="chapter">
  <tr>
    <td><xsl:value-of select="subtitle"/></td>
    <td>
      <xsl:for-each select="para">
        <xsl:value-of select="."/>
      </xsl:for-each>
    </td>
  </tr>
</xsl:template>
```

(MyBook.xsl)

# XSLT and the Web

Most browsers can understand XSLT instructions and transform XML documents.
This is done by an `xml-stylesheet` processing instruction in the beginning of the XML document
The instruction also specifies two attributes:

- The MIME type: `type="application/xml"`
- The XSL file address: `href="program.xsl"`

For example:

```
<?xml-stylesheet type="application/xml"
  href="MyBook.xsl" charset="UTF-8"?>
```