Java XML Processing

Text: EN

Software: EN

REALDOLMEN
to get there, together

Table of contents

1. Java XML Processing

1.1. Course Objectives

Getting to know the Java APIs for XML processing: SAX, DOM, JDOM, StAX, JAXB, TrAX, Java XML Web Services Less theory, more practice

1.2. Agenda

- Day one
  - Introduction
  - Summary of basic XML concepts
  - Overview of Java APIs for XML
  - XML uses and best practices
  - SAX
- Day two
  - DOM
  - StAX
  - JDOM
- Day three
  - TrAX
  - JAXB
  - Java XML Web Services

2. Summary of XML Concepts

2.1. Seven XML statements

1. XML is a method for putting structured data into a text file
2. XML resembles HTML
3. XML is machine readable / human intelligible
4. XML is a family of technologies
5. XML is verbose (but that is OK)
6. XML is based on solid technologies
7. XML is license-free, platform independent and well-supported

2.2. A brief history of XML

YEAR	TECHNOLOGIES
1969	ARPAnet (first 'Internet') developed by the Advanced Research Projects Agency of the Department of Defense
1979	In 1979, a Committee was formed to develop a generic EDI standard (Electronic Data Interchange X12), the predecessor to XML
1986	SGML: Standard Generalized Markup Language proposed. The mother of all markup languages
1989	Tim Berners-Lee (CERN) proposes the World Wide Web project (original name was "Mesh")
1990	Tim Berners-Lee developed the first beta version of HTML to enable scientists to easily exchange and share research
1993	Berners-Lee used Connolly's work as a basis for a RFC (Request For Comments): HTML 1.0. Described "content, not format", just like SGML (13 tags!)

YEAR	TECHNOLOGIES
1995	HotJava, a transportable language, is introduced at SunWorld '95
1996	Jon Bosak at Sun and small group of SGML experts proposed new W3C working group to devise modified SGML. The purpose was to reinject SGML into the Web by concentrating on SGML's extensible tags. There would be 3 parts: XML, XLL and XSL
1998	RECOMMENDATION: XML 1.0 (first edition) – simplified SGML without HTML's problems
2000	RECOMMENDATION: XHTML 1.0 - The eXtensible HyperText Markup Language will slowly replace the last version of HTML, being 4.01. It respects XML syntax so it can also be validated using Schemas and transformed and presented using XSL

2.3. XML vs. SGML

- Standard Generalized Markup Language
  - Based on IBM's GML (Goldfarb, et al.)
  - ISO standard since 1989
  - Used for large-scale document management (Boeing 747 user's manual)
  - Expensive, complex to implement
  - Not Web-friendly (e.g., no "well-formed" SGML)
  - Too many options (e.g., tag minimization)
- XML comes from SGML

2.4. XML vs. HTML

HTML	XML
Predefined tags define how to present data	Defines its own tags to identify data
Missing end tags are permitted, ` `, `<p>`	Requires matching end tags, ` `, `</br>`, or ` `
Attributes do not require quotes, `<img src=bedfont.jpg>`	Attributes do require a quotes `<img src="bedfont.jpg"/>`
Attributes do not require a value, `<input type=radio checked>`	Attributes do require a value, `<input type="radio" checked="checked/">`
Tolerates non-nested tags, `<h1><centre>Hello</h1></centre>`	Requires property nested tags, `<h1><centre>Hello</centre></h1>`
Browser may do a best guess on non well-formed HTML	Well-formedness violation causes parsers to generate fatal error
Is not case sensitive `<table>...</Table>` is valid	Case sensitive, `<table>` and `<Table>` are different tags
No support for empty element	Supports empty elements `<Radio/>`

2.5. XML vs. SGML vs. HTML vs. XHTML

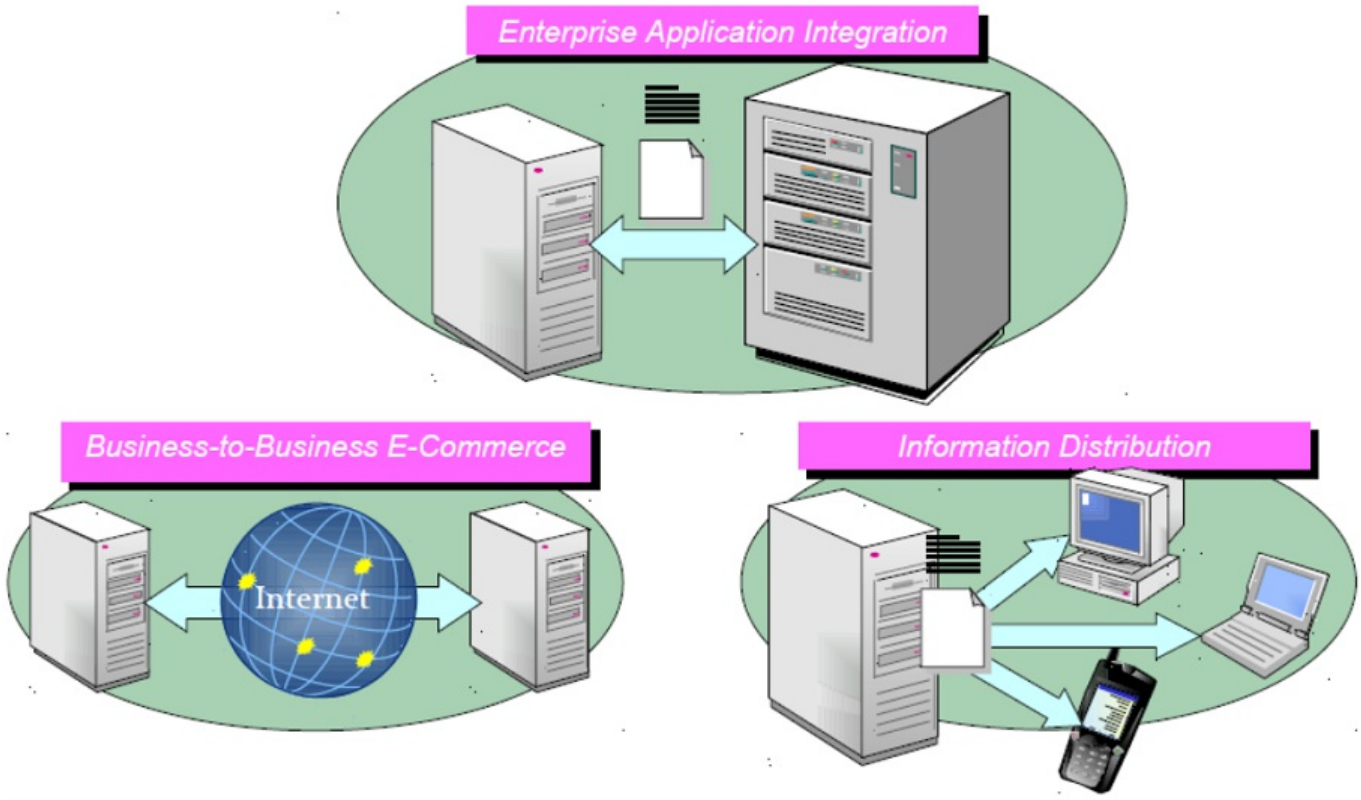
- XML is a purely syntactical convention
  - A simplified, streamlined subset of SGML
  - May be used to define any type of language: names and semantics of elements and attributes can be freely chosen
  - Not bound to any specific type of application

- HTML is a specific semantic schema (vocabulary and grammar)
  - meant primarily to be interpreted by one specific type of application (web browsers) which render information to human users
  - Pre-XHTML (up to HTML 4.0)
    - not strictly XML-compliant, syntactically forgiving
  - XHTML
    - conforms to strict XML syntax rules

2.6. XML Advantages

- Platform and language independent
- Easy to write, easy to process
- Understandable for humans and computers
- Open standard
  - Many libraries exist
  - Lots of literature available
  - Specialized XML-editors
- Possibility to check the document structure
- XML can be used for data transfer and web services
- XML is ideal for data storage
- XML can be transformed in HTML, text, PDF, ...
  - No need for special program solutions
  - Platform independent

2.7. XML Uses



2.8. Structure of an XML document

- XML declaration

```
<?xml version="1.0" ?>
```

- Document element
  - contains all the other elements in an XML document
- Elements containing:
  - Element content: other elements
  - Character content: text
  - Mixed content: elements and text
  - No content: empty elements
- Attributes: name / value pairs

```
<elementname attributename="value"/>
```

2.9. Criteria for well-formedness

- A well-formed XML document is one that conforms with the XML syntax rules, in

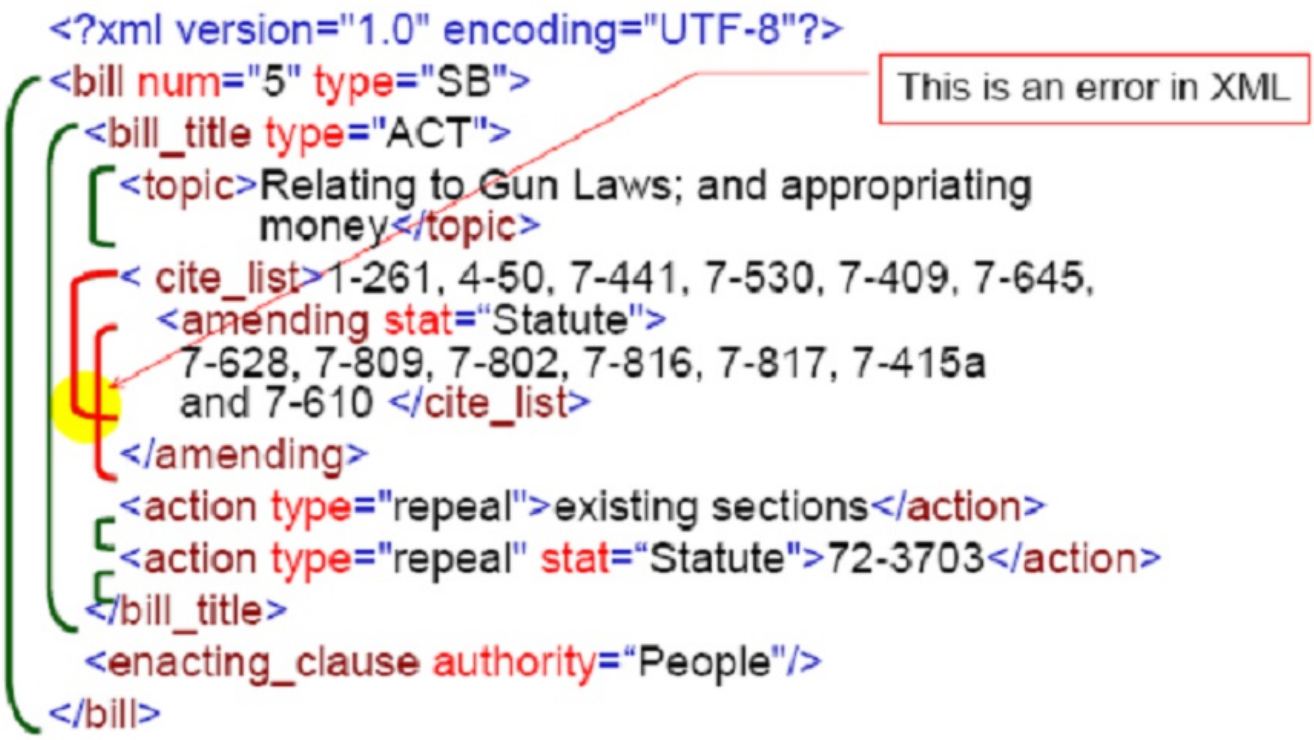
particular:

- A single, unique document element
- Start and end tags must match
- Consistent capitalization (case sensitive!!)
- Correctly nested elements
- No repeating attributes
- Attribute values enclosed in quotes

2.10. A test for proper nesting – OK

```
<?xml version="1.0" encoding="UTF-8"?>
<bill num="5" type="SB">
  <bill_title type="ACT">
    <topic>Relating to Gun Laws; and appropriating
           money</topic>
    <amending stat="Statute">
      < cite_list>1-261, 4-50, 7-441, 7-530, 7-409, 7-645,
      7-628, 7-809, 7-802, 7-816, 7-817, 7-415a
      and 7-610 </cite_list>
    </amending>
    <action type="repeal">existing sections</action>
    <action type="repeal" stat="Statute">72-3703</action>
  </bill_title>
  <enacting_clause authority="People"/>
</bill>
```

2.11. A test for proper nesting – NOK!



2.12. XML Namespaces

- Namespace declarations are in the form:

```
xmlns="aURI"
xmlns:prefix="http://www.realdolmen.com/education"
```

- The first example sets the default namespace
- The second sets a namespace prefix `prefix` equal to the URI `http://www.realdolmen.com/education` so it can be used like:

```
<prefix:course>content</prefix:course>
```

2.13. Additional remarks on XML syntax

- Full specifications: see http://www.w3.org/XML
- Some additional syntactical components
  - Comments `<!-- this is a comment -->`
    - Mainly intended for human readers but can be accessed programmatically
  - Processing instructions `<?xml ?>`
    - Specialized instructions for a particular receiving application

    - Namespace declarations and prefixes `xmlns:rd="my-uri"`
        - Allows for the unambiguous identification of elements and attributes from different vocabularies

- Text internationalization
    - Internally all XML processing is based on Unicode
    - External storage of XML documents can use any other standard encoding

---

2.14. XML Schemas

- Definition
    - A schema describes additional rules to which a document must conform, such as:
        - What elements and attributes may or must be present
        - In what relationship to one another elements and attributes may be combined
        - Data types and allowable values
        - What values must be unique within the document

2.15. Validity of XML documents

- A valid XML document is one that conforms to an associated schema
    - The document must be well-formed
    - The document must contain the required elements
    - Elements must be combined as described in the schema
    - Attributes and element content must be of the allowed data types
    - If a value must be unique, there can not be more than one in the document

---

2.16. XML Schema languages

- Document Type Definition (DTD)
    - An older schema language developed for XML's ancestor, SGML
    - Does not itself use XML notation
    - Has important functional weaknesses (e.g. limited support for datatypes or cardinality)
    - Universally supported but recently has been replaced by XML Schema

- XML Schema
    - Newer and semantically richer than DTD
    - Is itself an XML vocabulary
    - Full specifications available at http://www.w3.org/XML/Schema

---

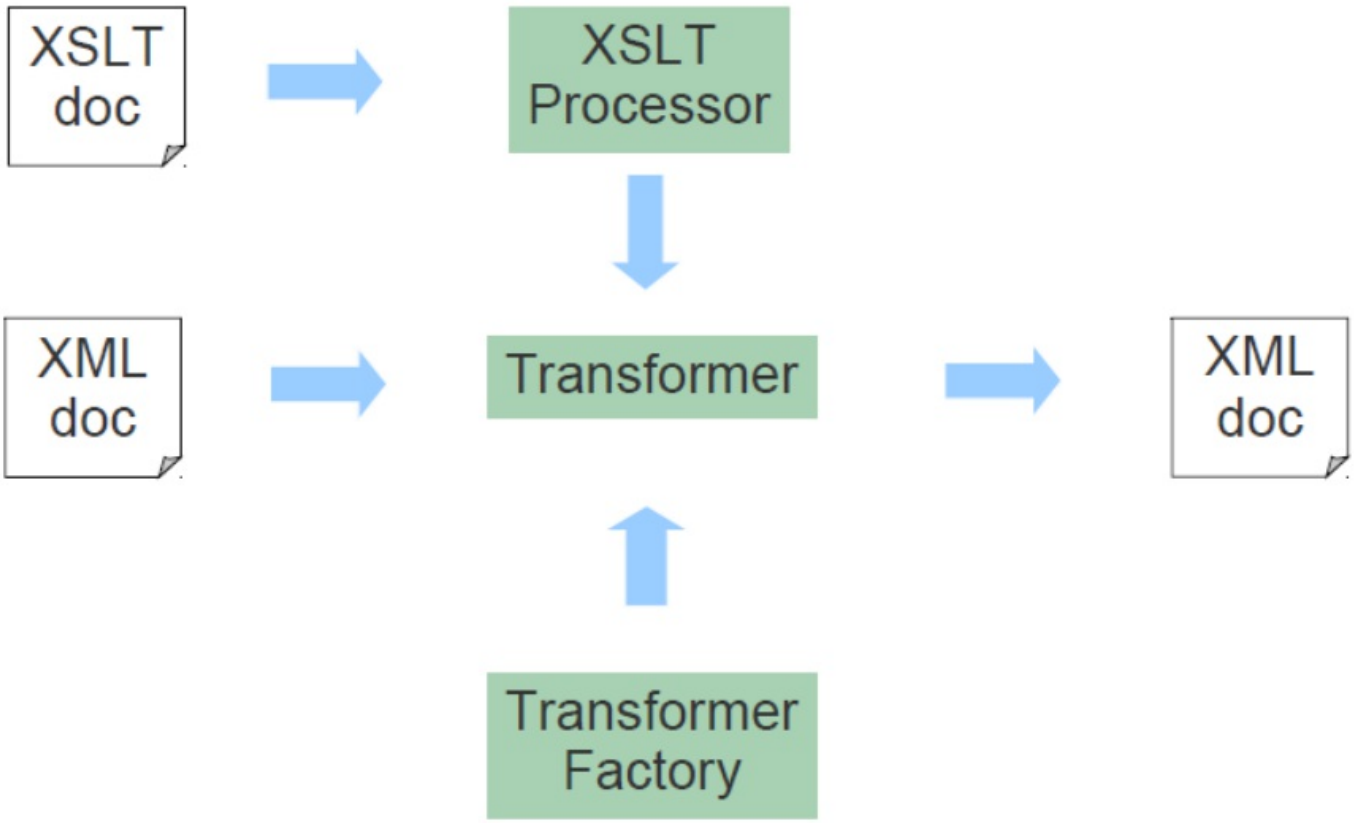
2.17. Rendering XML using XSLT

- XSLT means XML Stylesheet Language Transformations
  - Specification that allows:
    - Presentation (rendering) of XML
      - XML offers no support for the presentation (visual or otherwise) of documents
      - CSS (Cascading Style Sheets) offers limited functionality and serves only for presentation in a browser
    - Random transformation of XML
      - Translation between two different XML vocabularies
      - Transformation of XML to text

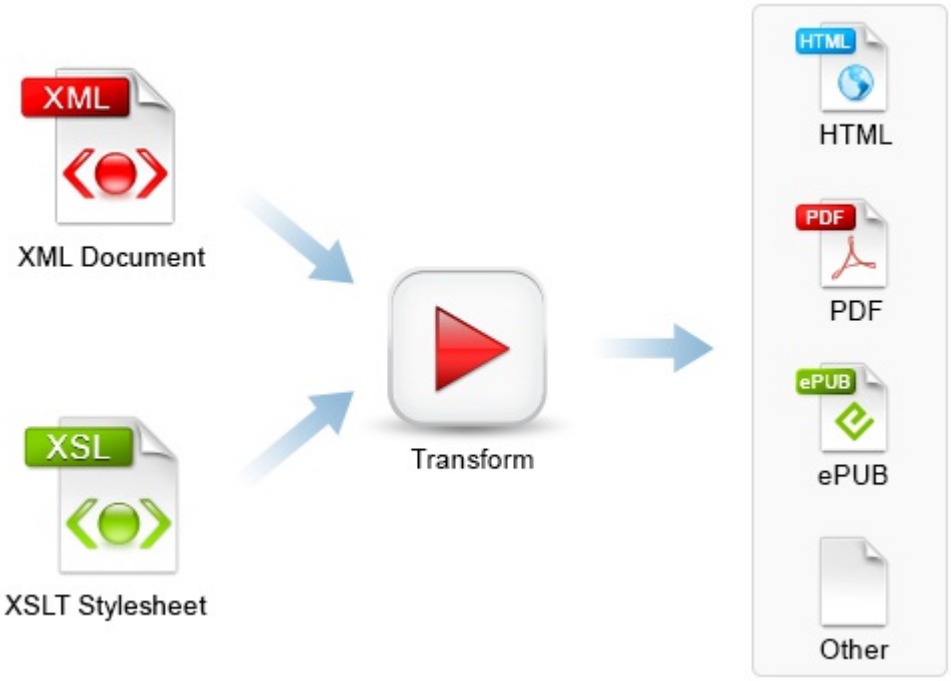
2.18. XSLT-XPath interaction

- XPath
  - XML Path Language
  - Syntax: a non-XML language
  - Purpose: expressing criteria to
    - Identify nodesets (sets of nodes within an XML document) that match the criteria
    - Determine whether a given node matches the criteria
- XSLT
  - Syntax: an XML vocabulary
    - Namespace URI "http://www.w3.org/1999/XSL/Transform"
  - Purpose: expressing actions to be taken on individual nodes
    - Actions fall into two major categories:
      - Identifying new nodesets
      - Generating various types of output

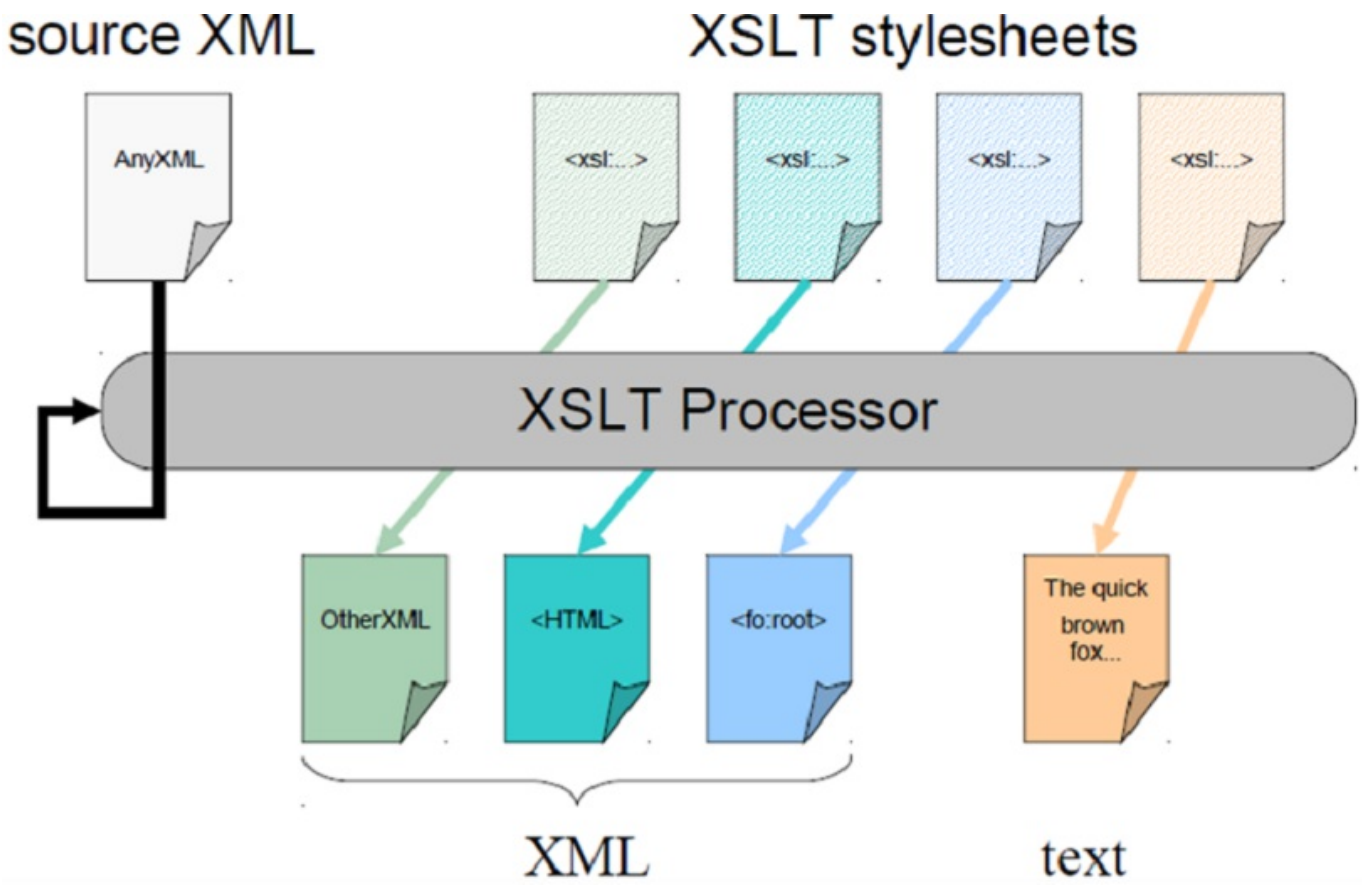
2.19. XSLT Structure



2.20. A simple XSLT transformation



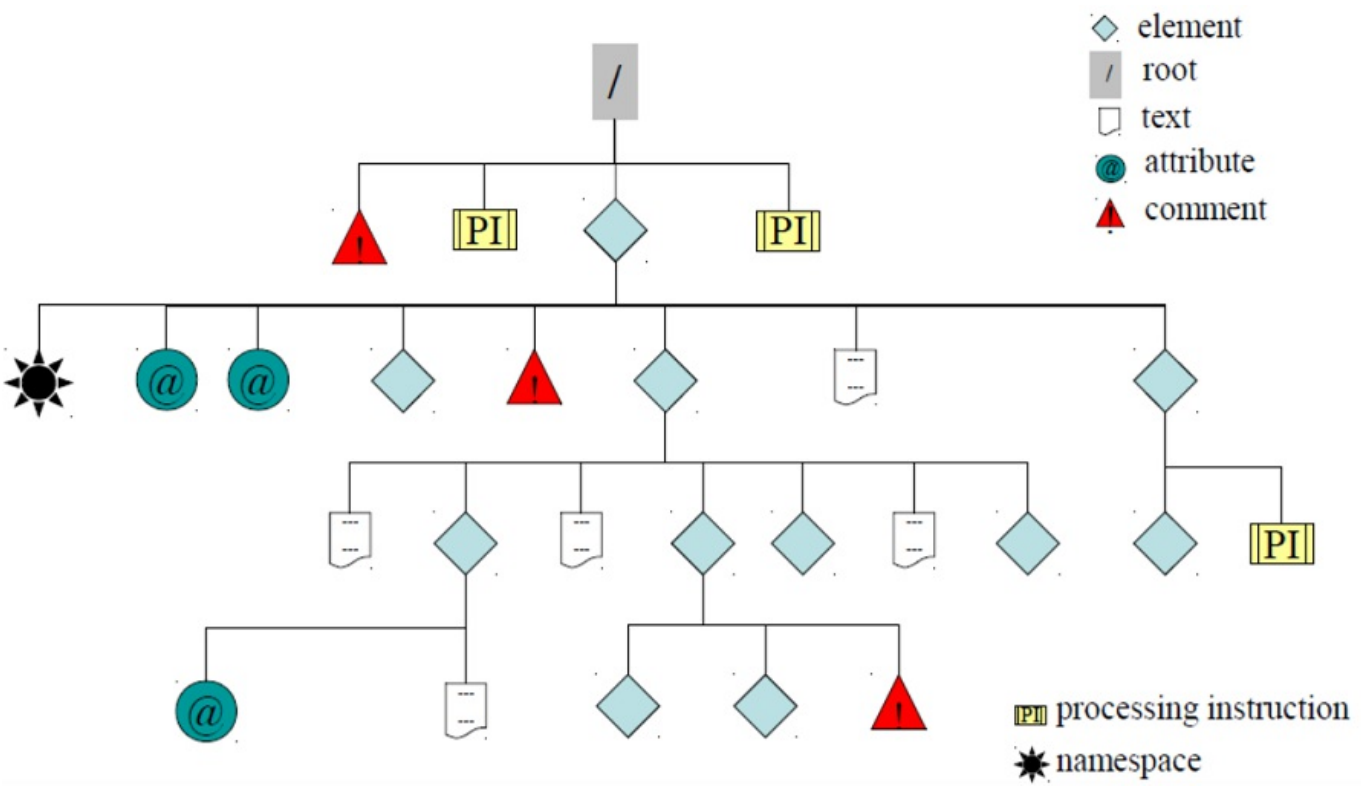
2.21. How does XSLT work?



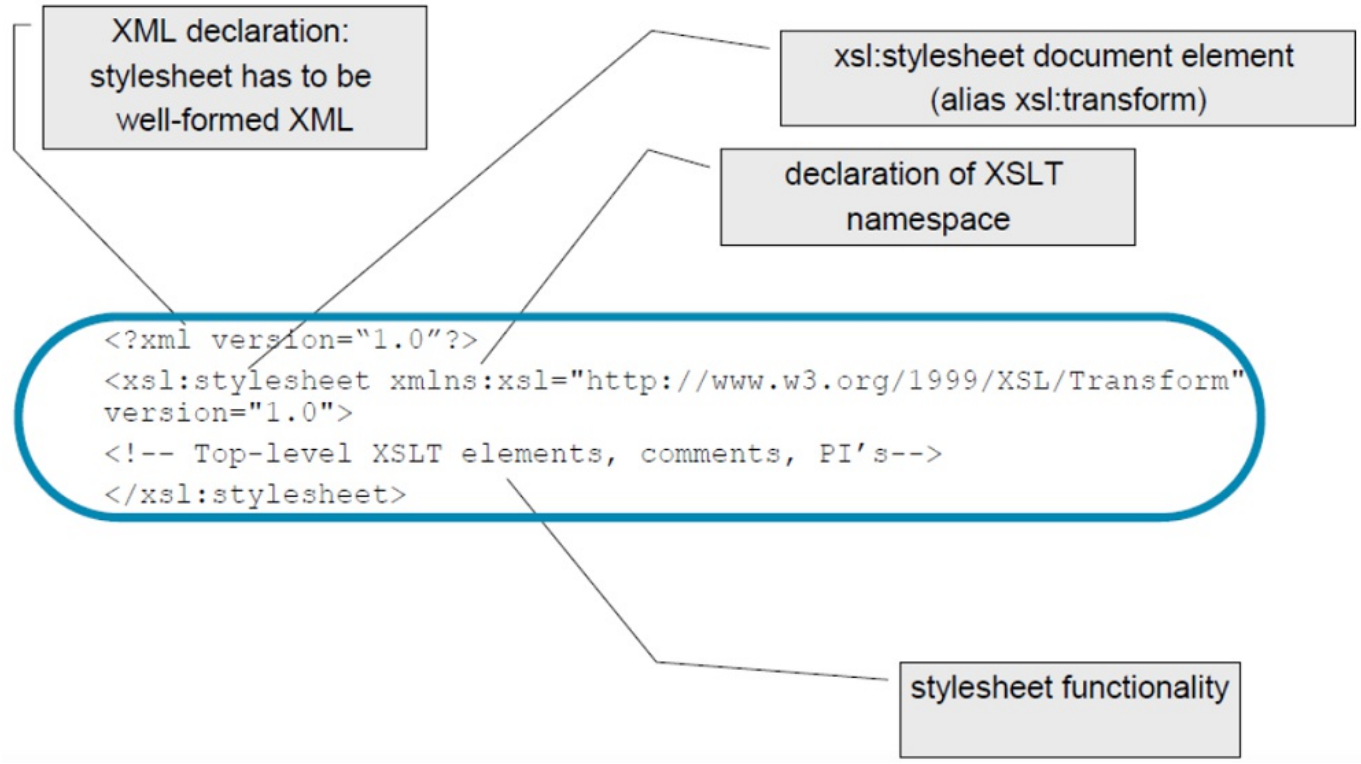
2.22. How does XSLT work

- Principle of operation
  - Converts a `Source` into a `Result` by invoking a `Transformer`
- Implementations of `Source` and `Result`
  - `DOMSource`, `DOMResult`: based on DOM nodes
  - `StreamSource`, StreamResult: based on sequential input and output streams
  - `SAXSource`: based on an `XMLReader`
  - `SAXResult`: based on a `ContentHandler`
- XSLT transformations
  - when passing to the `Transformer` constructor a `Source` argument that refers to an XSLT stylesheet, the latter will be applied during the transformation
  - if you do not pass a stylesheet, the identity transformation is used (which also enables you to serialize DOM nodes)

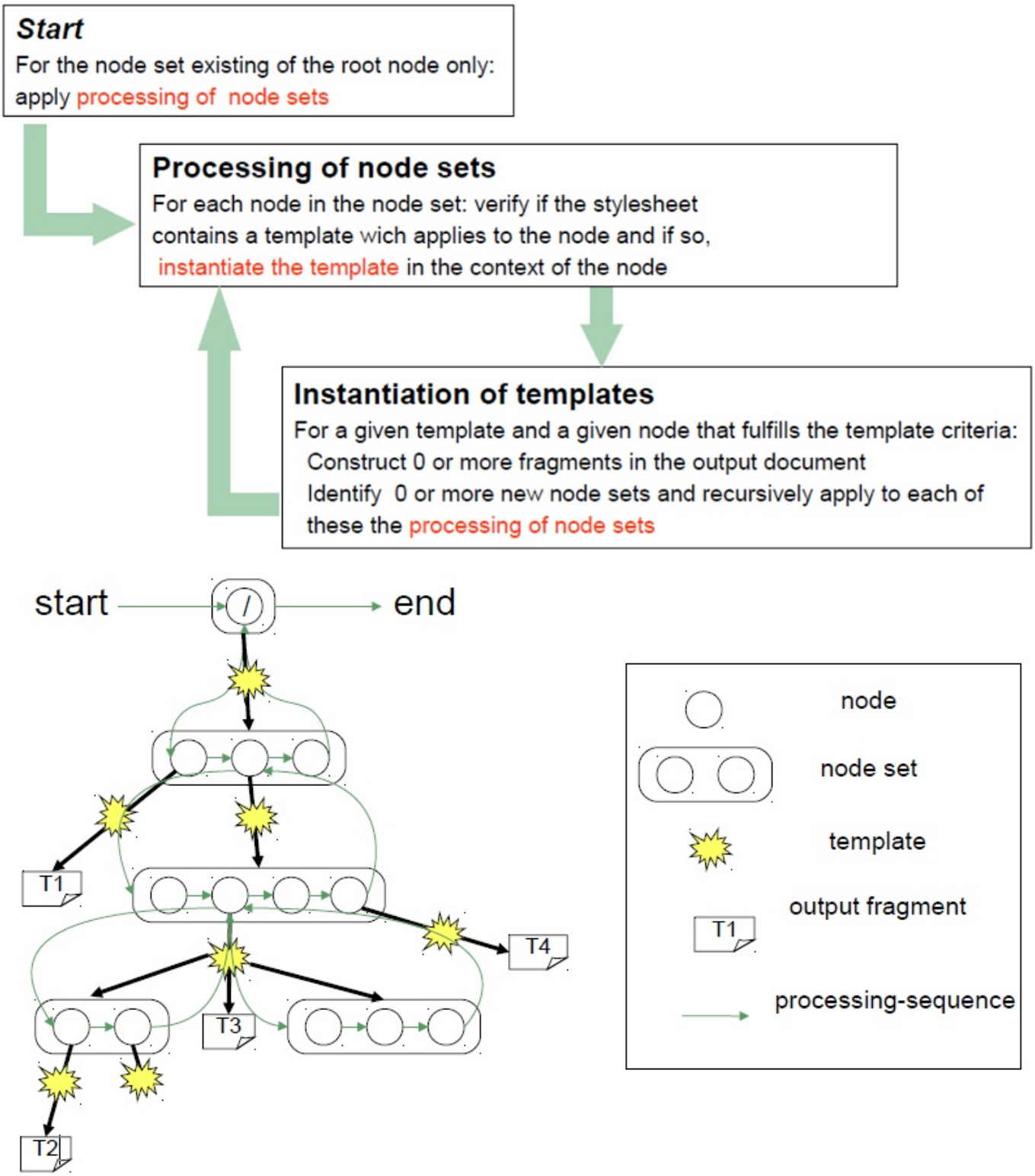
2.23. The XSLT data model



2.24. Structure of an XSLT document



2.25. The XSLT translation process



2.26. A complete example

- Semantics
  - Transfers the first text child node of each `<CourseCode>` element in the document to the output document

- Structure
  - 1 template for the root node and 1 template for arbitrary element nodes

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.1"
        xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
        xmlns="http://www.w3.org/TR/REC-html40">
    <xsl:template match="/">
        <xsl:apply-templates select="descendant::CourseCode"/>
    </xsl:template>
    <xsl:template match="*">
        <xsl:value-of select="child::text()[position()=1]"/>
    </xsl:template>
</xsl:stylesheet>
```

2.27. Steps in an XSLT application

- Instantiation of a `TransformerFactory`

```
TransformerFactory tff = TransformerFactory.newInstance();
```

- Instantiation of a `Transformer`
  - Constructor without argument: identity transformation
  - Constructor with argument: transformation according to the XSLT instructions in the argument stylesheet

```
Transformer transformer = tff.newTransformer(...);
```

- Performing the transformation
  - source is `DOMSource`, `SAXSource` or `StreamSource`
  - result is `DOMResult`, `SAXResult` or `StreamResult`

```
transformer.transform(source, result);
```

2.28. XSLT Example

- Writing out a subtree of a DOM document

```
public class ExtractBody {
    public static void main(String[] args) {
        try{
            File f = new File("email.xml");
            DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
            DocumentBuilder builder = factory.newDocumentBuilder();
            Document document = builder.parse(f);
```

```
            NodeList list = document.getElementsByTagName("body");
            Node node = list.item(0);
            TransformerFactory tFactory = TransformerFactory.newInstance();
            Transformer transformer = tFactory.newTransformer();
            DOMSource source = new DOMSource(node);
            StreamResult result = new StreamResult(System.out);
            transformer.transform(source, result);
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

2.29. Essential XSLT elements

XSLT element	Function	Syntax of the attribute value
xsl:template match="pattern"	Defines a template and the criteria for its application	XPath pattern syntax
xsl:apply-templates select="location-path"	Identifies a node set and starts the template processingof this set	XPath location path syntax
xsl:value-of select="expression"	Evaluates the value of the select attribute, converts the result to a string and writes this to the output document	XPath expression syntax

2.30. XPath examples

- What do the following examples mean?
  - `//para[5]`
  - `para[contains(text(),”SQL”)][last()]`
  - `company[@id="Realdolmen"]/department`
  - `count(company/employee)`
  - `count(company//employee)`
  - `//employee[salary > 2000]/ancestor::department/name/text()`
  - `/descendant::node() > 12`

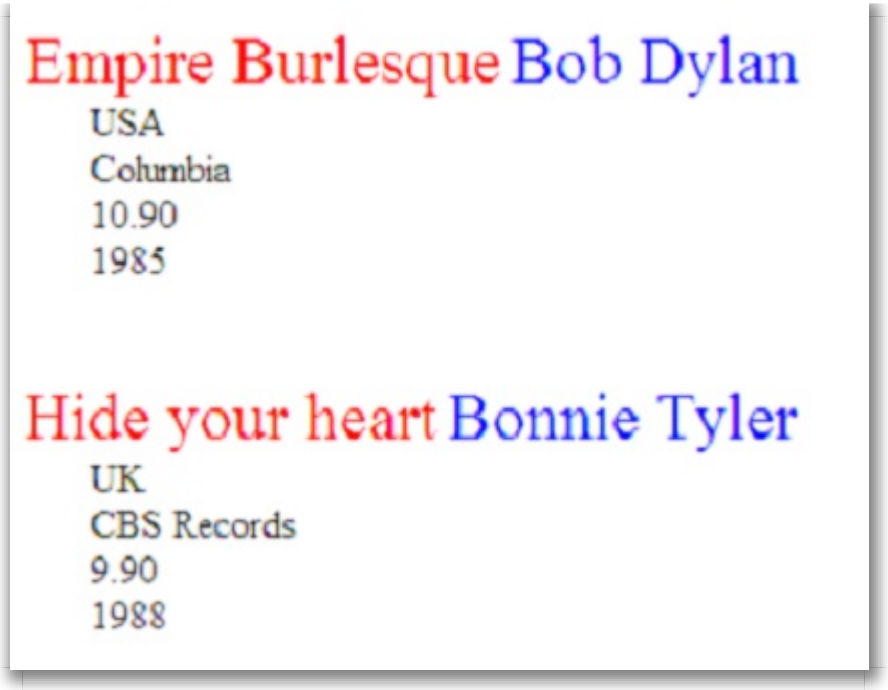
2.31. Displaying XML with CSS

```
<?xml version="1.0" encoding="UTF-8"?>
```

```xml
<?xml-stylesheet type="text/css" href="cd_catalog.css"?>
<CATALOG>
    <CD>
        <TITLE>Empire Burlesque</TITLE>
        <ARTIST>Bob Dylan</ARTIST>
        <COUNTRY>USA</COUNTRY>
        <COMPANY>Columbia</COMPANY>
        <PRICE>10.90</PRICE>
        <YEAR>1985</YEAR>
    </CD>
    <CD>
        <TITLE>Hide your heart</TITLE>
        <ARTIST>Bonnie Tyler</ARTIST>
        <COUNTRY>UK</COUNTRY>
        <COMPANY>CBS Records</COMPANY>
        <PRICE>9.90</PRICE>
        <YEAR>1988</YEAR>
    </CD>
</CATALOG>
```

```css
CATALOG {
    background-color: #FFFFFF;
    width: 100%;
}

CD {
    display: block;
    margin-bottom: 30pt;
    margin-left: 0;
}

TITLE {
    color: #FF0000;
    font-size: 20pt;
}

ARTIST {
    color: #0000FF;
    font-size: 20pt;
}

COUNTRY, PRICE, YEAR, COMPANY {
    display: block;
    color: black;
    margin-left: 20pt;
}
```

- You can use CSS to render XML in browsers, just as with HTML
- You need to link the XML file to CSS

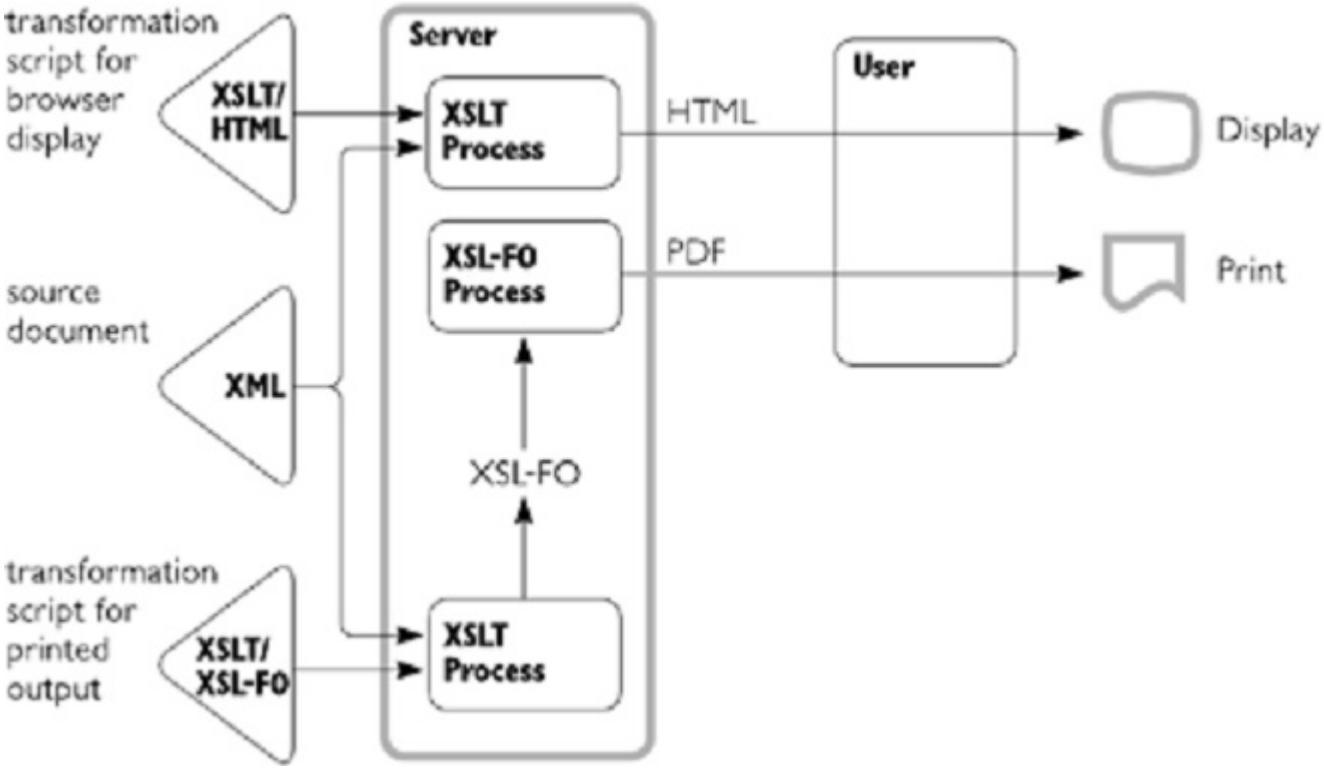


2.32. XSL:FO (Formatting Objects)

- Description
  - an XML vocabulary for the description of page layouts
  - FO documents transformed by rendering engine into an output format (e.g. PDF or RTF)
    - Example engine: Apache FOP (http://xml.apache.org/fop) Free and Open Source Java implementation
- Features
  - Very extensive specification
  - Only partial implementations available today
- Reference
  - http://www.w3.org/TR/xsl

2.33. The XSL-FO transformation

- The process of transforming an FO document is similar to XSLT but involves an extra processing step



2.34. XML to XSL-FO with XSLT

- XSL transformation to HTML

```
<xsl:template match="customer">
    <p><xsl:text>From: </xsl:text>
    <i><xsl:text>(Customer Reference) </xsl:text>
    <b><xsl:value-of select="@db"/></b></i></p>
</xsl:template>
```

- XSL transformation to XSL-FO

```
<xsl:template match="customer">
    <fo:block space-before.optimum="20pt" font-size="20pt">
        <xsl:text>From: </xsl:text>
        <fo:inline font-style="italic">
        <xsl:text>(Customer Reference) </xsl:text>
        <fo:inline font-weight="bold">
        <xsl:value-of select="@db"/></fo:inline></fo:inline>
    </fo:block>
</xsl:template>
```

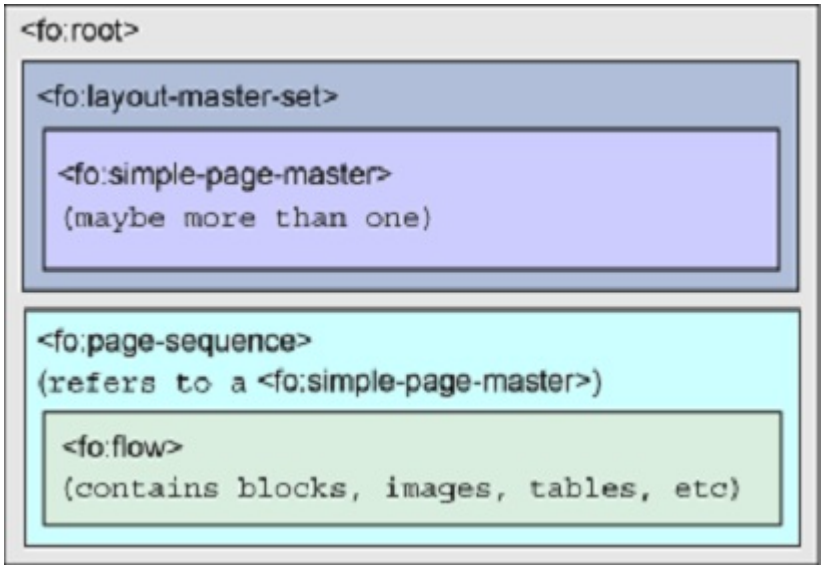
2.35. Rendering engine: Apache FOP

- FOP: Formatting Objects to PDF

- Written in Java
  - Runs on any platform and any OS
  - Open source, free!
- Does not completely support the XSL-FO specification
  - Certain property names and values are not supported
  - Some are supported with slightly different names

2.36. XSL-FO document overview

- XSL-FO documents define
  - Information about the physical size of the page (A4, ...)
  - Information about margins, headers, footers, ...
  - Information about fonts, font sizes, colors, ...
  - The actual text, with markups such as table, paragraph and highlighting



2.37. XSL-FO example

```
<fo:root xmlns:fo="http://www.w3.org/1999/XSL/Format">
    <fo:layout-master-set>
        <fo:simple-page-master master-name="main"
            margin-top="36pt" margin-bottom="36pt"
            page-width="8.5in" page-height="11in"
            margin-left="72pt" margin-right="72pt">
                <fo:region-body margin-bottom="50pt"
                    margin-top="50pt"/>
        </fo:simple-page-master>
    </fo:layout-master-set>
    <fo:page-sequence master-reference="main">
        <fo:flow flow-name="xsl-region-body">
            <fo:block font-size="14pt" line-height="17pt">
```

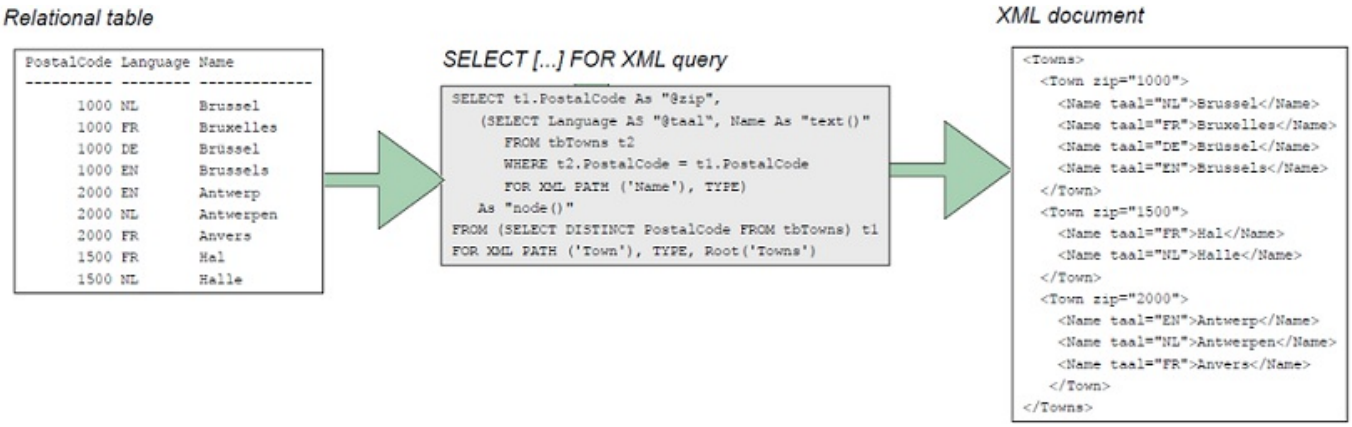
```
                This example shows of a complete XSL-FO, simple but
                    <fo:inline font-style="italic">totally meaningless
                    </fo:inline>document. Line breaks are calculated
                automatically by the renderer.
            </fo:block>
        </fo:flow>
    </fo:page-sequence>
</fo:root>
```

2.38. XML support in office suites

- Office suites store their data files in XML
- LibreOffice
- Microsoft Office
  - Word: WordML
    - Markup for content and details of Word documents
    - Supports XSLT
  - Excel: SpreadsheetML
    - Markup for Excel content
    - No charts or VBA
  - InfoPath
    - Form-building and processing toolkit
    - Uses XML, XSLT and other web technologies

2.39. Physical storage of XML data

- The OS file system
  - Simplest solution
- O-RDBMS with XML datatype support
  - Provides support for XML structures in existing relational databases
    - PostgreSQL, MySQL, Oracle, SQL Server, ...

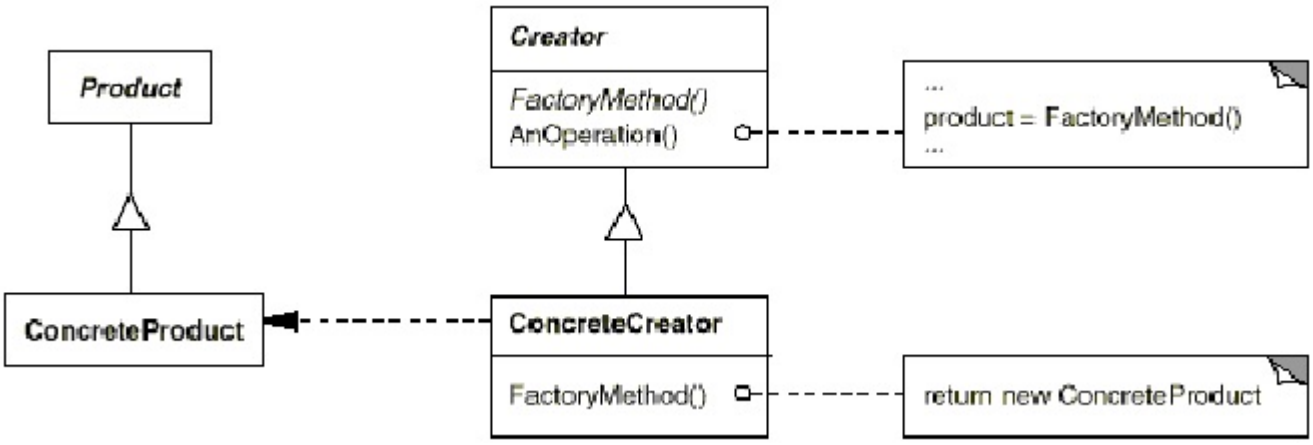


- XML Databases
  - Provide native storage for XML

3. Overview of Java APIs for XML

3.1. Overview

- Provides common interfaces
  - SAX, DOM, XSLT
- Independent of particular implementation
  - Based on Factory Method design pattern
  - Define an interface for creating an object
    - Defer instantiation to subclasses



- Push-type streaming APIs (SAX)
  - Pushes events to your program, which will handle them; unnecessary events are discarded
- Tree-based APIs (DOM, JDOM)
  - the parser converts the XML document to a tree representation which is entirely loaded into memory
  - the API consists of an extensive set of methods for selecting, retrieving and manipulating the nodes of the tree
- Pull-type streaming APIs (StAX)
  - Your code will ask the parser for the next event, instead of being notified by the parser
- Data binding APIs (JAXB)
  - Eliminates references to XML nodes from your code
  - Binds XML Schema specific Java classes
  - Can convert in both directions
- Query APIs (XPath)
  - The JAXP XPath support makes it possible to evaluate XPath expressions on your XML documents

3.2. Comparing SAX to DOM

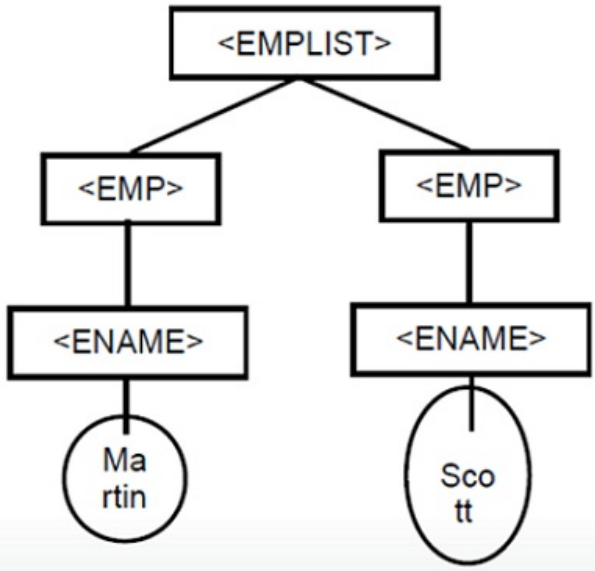
- SAX
  - Event based model
  - Flow of events
  - Low memory usage
  - Serial access
  - Process parts of the document
  - Process the document once
- SAX is faster and better suited for fragments of data (e.g. not the entire document as a whole)
- DOM
  - Tree data structure
  - In memory data structure
  - High memory usage
  - Random access
  - Edit the document
  - Process the document multiple times
- DOM is more flexible and better suited for entire documents
- Event-based API: SAX
  - Uses callbacks to report parsing events to the application
  - Application deals with these events through customized event handlers
  - Parse tree cannot be manipulated
- Tree-based API: DOM
  - Provides objects and methods to be used by the application
  - Application uses these methods to navigate and manipulate the tree
  - Allows adding / modifying / deleting elements and attributes

```
<?xml version="1.0"?>
<EMPLIST><EMP><ENAME>Martin</ENAME></EMP>
    <EMP><ENAME>SCOTT</ENAME></EMP></EMPLIST>
```

SAX

start document
start element: EMPLIST
start element: EMP
start element: ENAME
characters: Martin
end element: ENAME
end element: EMP
start element: EMP
start element: ENAME
characters: Scott
end element: ENAME
end element: EMP
end element: EMPLIST

DOM



3.3. The JAXP packages

- The JAXP APIs are spread throughout a number of different packages

Package	Content
`javax.xml.parsers`	The DOM and SAX parsers and their respective factory classes
`org.xml.sax`	Essential SAX interfaces and exception classes
`org.xml.sax.ext`	Additional SAX interfaces: LexicalHandler and DeclHandler
`org.xml.sax.helpers`	Various useful implementation classes, including DefaultHandler
`org.w3c.dom`	DOM interfaces and the DOMException class
`javax.xml.transform`	Essential transformation interfaces and classes, including Transformer
`javax.xml.transform.dom`	Classes for transformation of DOM objects
`javax.xml.transform.sax`	Classes for the transformation of the input to or the output from a SAX parser
`javax.xml.transform.stream`	Classes for transformation into or from streams

- Part of JavaSE

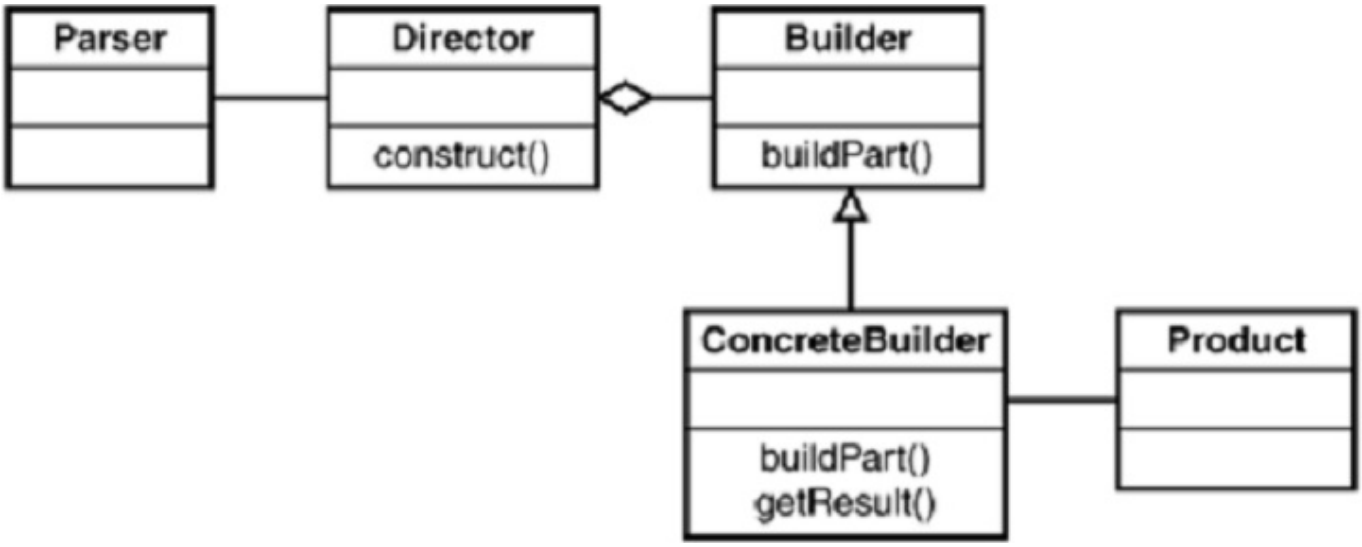
4. XML Uses and Best Practices

4.1. XML uses in applications

- Lightweight data storage
- Configuration and persistence
- To configure/define electronic forms
- As content syndication (ex. RSS)
- As intermediate format for exporting
- As format for importing
- As communication language (ex. Web Services)
- As intermediate format for representation
- To be searched and queried

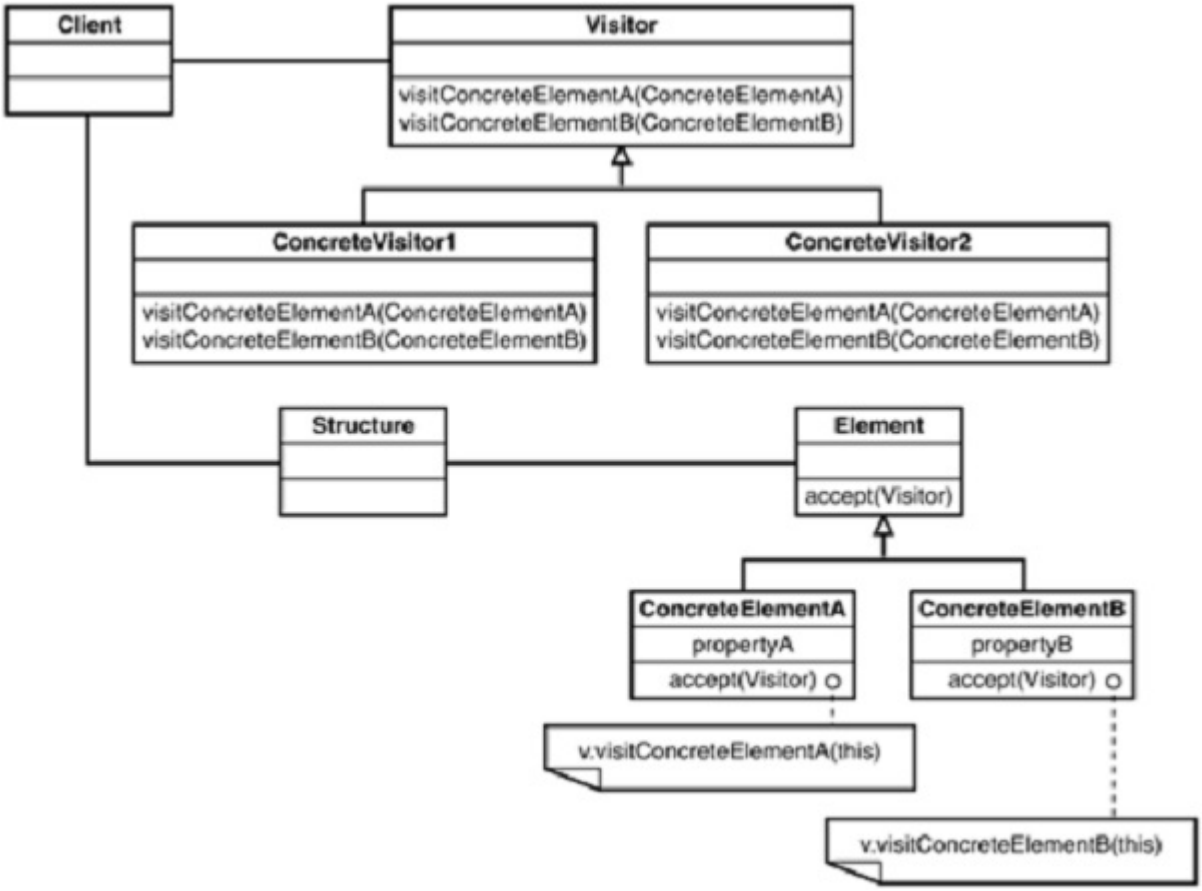
4.2. Using patterns: Builder

- When parsing XML documents with SAX or StAX, to create an in memory structure of Java objects



4.3. Using patterns: Visitor

- For printing the structure back to XML
- For traversing and accumulating values


Client
Visitor
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteVisitor1
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
ConcreteVisitor2
visitConcreteElementA(ConcreteElementA)
visitConcreteElementB(ConcreteElementB)
Structure
Element
accept(Visitor)
ConcreteElementA
propertyA
accept(Visitor)
ConcreteElementB
propertyB
accept(Visitor)
v.visitConcreteElementA(this)
v.visitConcreteElementB(this)

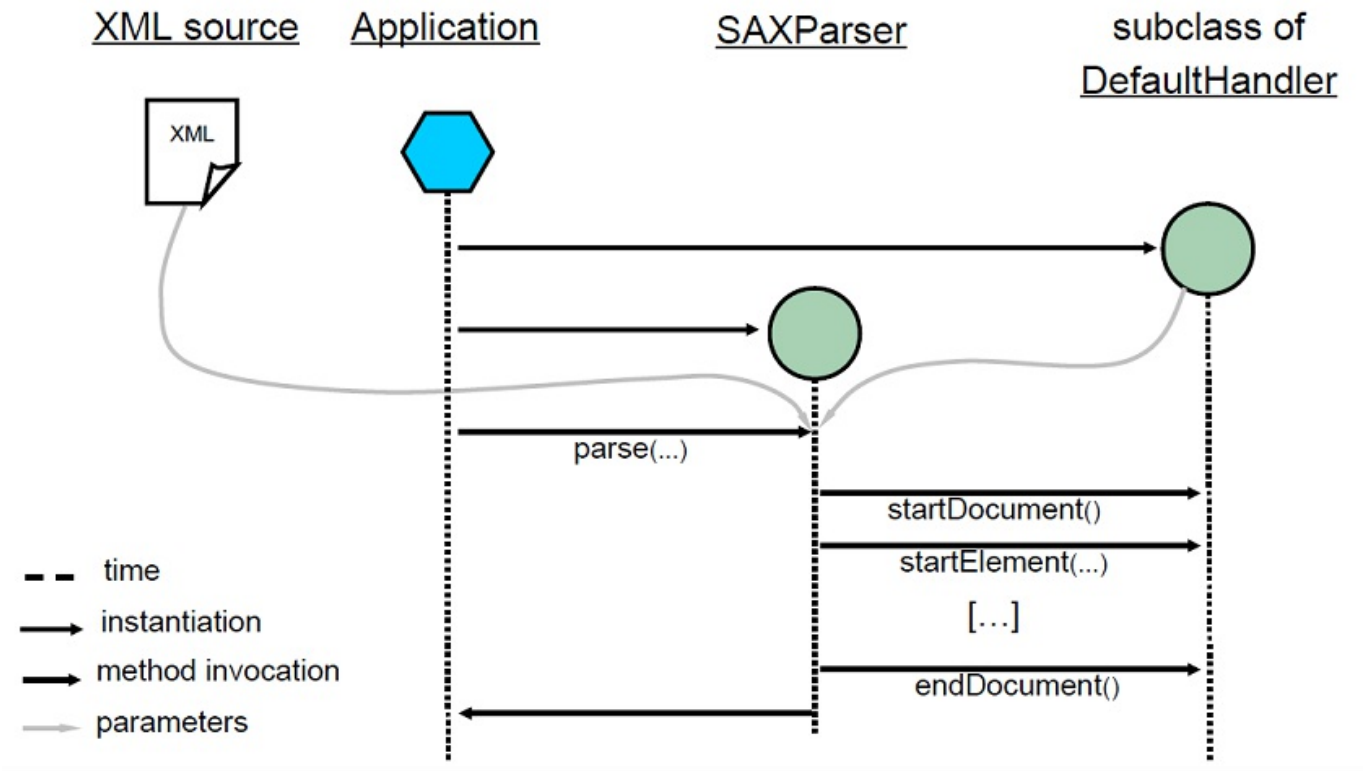
5. SAX: Simple API for XML

5.1. SAX

- Simple API for XML (SAX)
  - Event-driven
    - Parser passes recognized tokens as an event to the client
  - Low-level and general-purpose
    - Parser scans the document once from start to finish and examines every statement independently of all the others
  - Is a push-type streaming API
    - Pushes events to your program (using callbacks), which will handle them unnecessary events are ignored
    - The API consists of a limited set of methods called by the parser every time it detects a particular type of character string in the source document (e.g. a start tag)

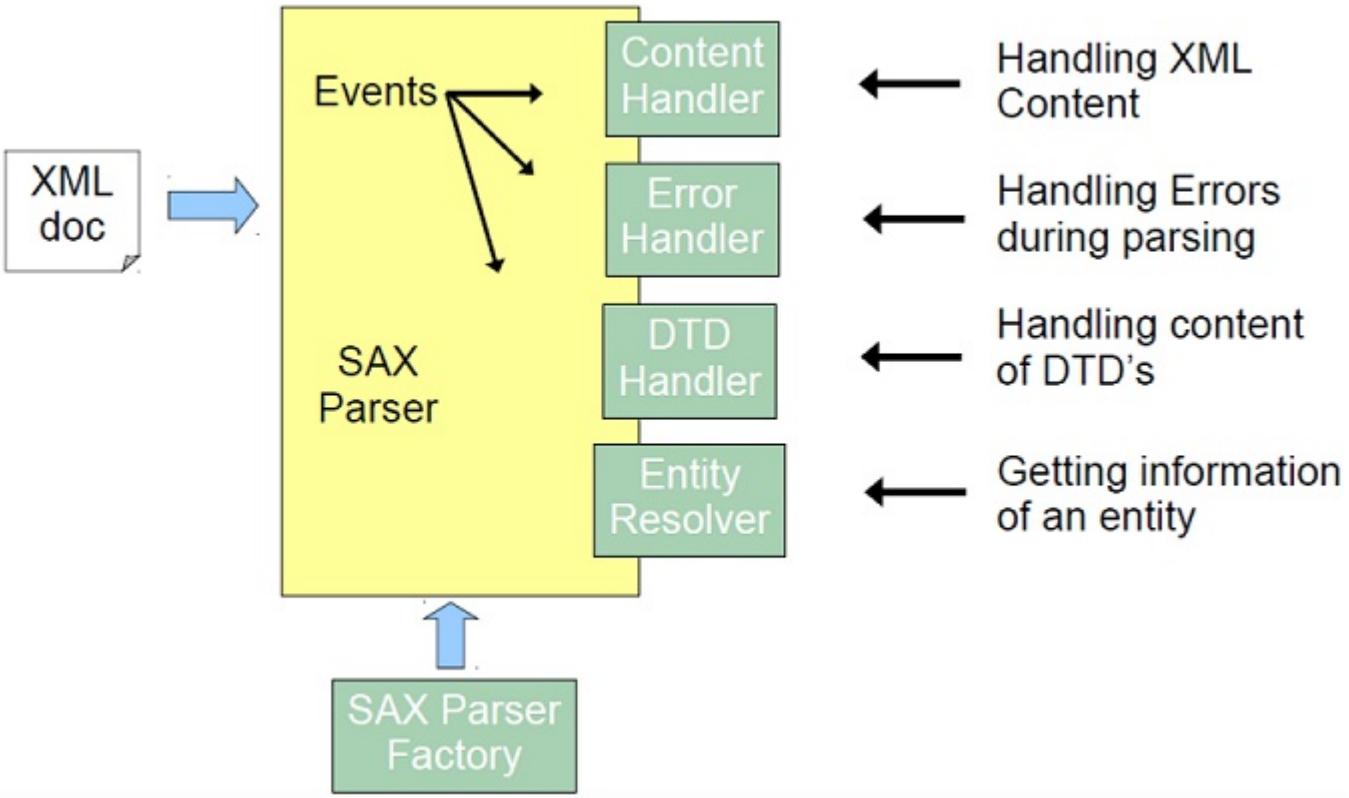
5.2. Anatomy of a SAX application

- This sequence diagram shows how the SAX API is used:



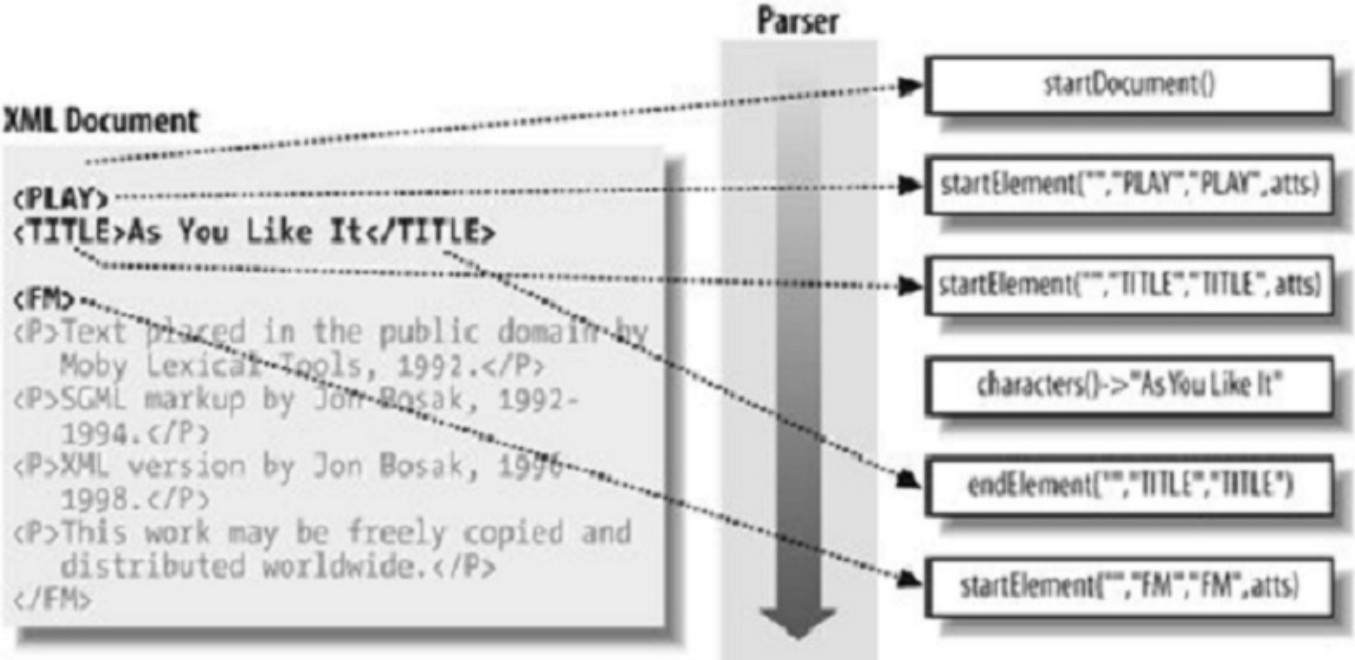
5.3. SAX structure

- The client should provide an implementation of `DefaultHandler` (yellow), which a number event callback interfaces (green)



5.4. SAX callback mechanism

- When the parser recognizes a token, its corresponding callback is triggered.



5.5. SAXParser instantiation

- Instantiation of a SAXParserFactory

```
SAXParserFactory factory = SAXParserFactory.newInstance();
```

- Instance methods of SAXParserFactory (optional)
  - To obtain a validating parser:
  - To obtain recognition by the parser of XML namespaces:

```
// Enable validation using XSD
spf.setSchema(SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_INSTANCE_NS_URI)
        .newSchema(new File("myschema.xsd")));
```

```
factory.setNamespaceAware(true);
```

- Instantiation of the SAXParser

```
SAXParser saxParser = factory.newSAXParser();   // Workhorse!
```

5.6. Class DefaultHandler

- Implemented interfaces
  - `ContentHandler`: handles events generated by the content of the document
  - `ErrorHandler`: handles errors and warnings
  - `DTDHandler`: handles events generated on the basis of DTD declarations
  - `EntityResolver`: handles resolution of external entities (e.g. `&euro;`, ...)
- Implementation
  - `DefaultHandler` contains no-op (empty) implementations of all methods
    - With the exception of `fatalError()`, used for validation
  - Clients create their own subclass of `DefaultHandler`
- Most of the methods inside these interfaces throw `SAXException`

5.7. ContentHandler interface

- Called once at the beginning and end of the entire document:

```
void startDocument();
void endDocument();
```

- Called every time a start tag `<tag>` and end tag `</tag>` is recognized:
  - `qName` is used when namespace aware is set
  - `Attributes` is a map-like structure containing the tag's attributes

```
void startElement(String namespaceURI, String localName, String qName, Attributes
atts);
void endElement(String namespaceURI, String localName, String qName);
```

- Called whenever there is a text node
  - Use `new String(ch, start, length)`
  - Ignorable whitespace is whitespace in between tags (if declared in schema as no PCDATA)

```
void characters(char[] ch, int start, int length);
void ignorableWhitespace(char[] ch, int start, int length);
```

- Called whenever a namespace `xmlns:rd='url'` is recognized or closed:

```
void startPrefixMapping(String prefix, String uri);
void endPrefixMapping(String prefix);
```

- Called whenever a processing instruction `<? ... ?>` is recognized:

```
void processingInstruction(String target, String data);
```

- Called whenever an entity (e.g. `&rd;` is recognized that's not declared in the DTD):

```
void skippedEntity(String name):
```

- Configure a custom `Locator`:
  - Allows line number information to become available
  - Must be set before parsing starts!

```
void setDocumentLocator(Locator locator):
```

- An simple example on how to create and use a `DefaultHandler`

```
public class MyHandler extends DefaultHandler {
    boolean insideMovie = false;

    public void startElement(String nsUri, String localName, String qName, Attributes a)
            throws SAXException {
        if(localName.equals("movie")) { insideMovie = true; }
    }

    public void endElement(String nsUri, String localName, String qName) throws SAXException {
        if(localName.equals("movie")) { insideMovie = false; }
    }

    public void characters(char[] chars, int offset, int length) throws SAXException {
        String title = new String(chars, offset, length);
        if(insideMovie) {
            System.out.println("Movie found: " + title);
        }
```

```
    }
}
```

```
saxParser.parse(new File("movies.xml", new MyHandler()));
```

SAX exception handling

- There are two types of exceptions

Exception	Purpose
`SAXException`	Thrown by the client (your code) if something's not right while parsing
`SAXParserException` (extends `SAXException`)	Thrown if the parser itself or validation fails (not your code) Has interesting methods `getLineNumber()` and `getColumnNumber()`

- `SAXParseException` is received by `ErrorHandler` (part of `DefaultHandler`)
  - `fatalError()`: Document is not well-formed
    - Default is to rethrow
  - `error()`: Document is not valid (only is validating mode is set)
    - Default is to do nothing (!)
  - `warning()`: Non fatal issue occurred
- Implementations of `DefaultHandler` should wrap application specific exceptions inside `SAXException`
  - They are first passed to the `SAXParser` and then propagate out of `parse()` for the calling application to catch

```
class MyHandler extends DefaultHandler { // Your handler containing callback implementations
    public void startElement(String ns, String ln, String qn, Attributes a) throws SAXException {
        throw new SAXException("Requested node found. Stopping processing.");
    }
    public void characters(char[] ch, int start, int length) throws SAXException {
        try {
            // ...
        } catch(IOException e) { throw new SAXException(e); }
    }
}
```

```
try { // Client code
    myParser.parse(myFile, new MyHandler());
} catch(SAXException e) {
    // Your exception is propagated up to here
}
```

SAX Parser Properties

- A `LexicalHandler` allows access to the pre-parsing phase
  - Allows you to "see" comments and other low level tokens as well
- Vendor specific properties can be configured

```
XMLReader xmlReader = saxParser.getXMLReader();
LexicalHandler handler = ...;
xmlReader.setProperty("http://xml.org/sax/properties/lexicalhandler", handler);
// Other properties can set in the same way
```

SAX Example

- Consider the following XML document:

```
<email>
    <head>
        <from><name>David Wouters</name>
        <address>David.Wouters@realdolmen.com</address></from>
        <to><name>Christian Devalez</name>
        <address>Christian.Devalez@realdolmen.com</address></to>
        <to><name>Yves Slincx</name>
        <address>Yves.Slincx@realdolmen.com</address></to>
        <subject>Just a mail</subject>
    </head>
    <body>
        <p>First Paragraph</p><p>Second Paragraph</p>
        <attach encoding="mime" name="myfile.html" />
    </body>
</email>
```

- This implementation of `DefaultHandler` filters out the recipients

```
class FindTo extends DefaultHandler {
    String to = "";
    boolean foundTo = false, foundName = false;

    public void startElement(String namespaceURI, String name, String qname, Attributes attr) {
        if (name.equals("to"))
            foundTo = true;
        else if (foundTo && name.equals("name"))
            foundName = true;
    }
    public void characters(char[] ch, int start, int length) {
        if (foundName)
            to += new String(ch, start, length);
    }
    public void endElement(String namespaceURI, String
            name, String qname){
        if (foundName && name.equals("name")){
            foundTo = false; foundName = false;
            System.out.println("To: " + to);
            to = "";
        }
    }
}
```

- The main application that uses the previous handler:

```
class ExtractTo {
    public static void main(String[] args){
        try {
            SAXParserFactory sf= SAXParserFactory.newInstance();
            sf.setNamespaceAware(false);
            SAXParser parser = sf.newSAXParser();
            parser.parse(new File("email.xml"), new FindTo());
        }catch (Exception e) {
            System.out.println("Error " + e.getMessage());
        }
    }
}
```

XML and Java Workshop

- The following are proposals for the exercises
- You can choose what you like, or invent another exercise
- In the end, we will use almost all APIs, but the project can still be extended, updated and improved
- Try to do at least the exercise tasks for your project
- First thing: create an XML data file for the chosen project
- Ask for suggestions/confirmation
  - Ask questions aloud if you are stuck
  - We will use Eclipse, so if you are not familiar with it, ask for a small tutorial for creating your project, etc...
  - Use and search the online API documentation

Workshop proposals

- A Movie Browser ***
- A Small Java Forum
- A Questionnaire Website
- An Online Poll System
- An Online Cookbook
- An Online Calendar/Planning
- An Online Tasklist
- An Online Addressbook

Exercise: SAX

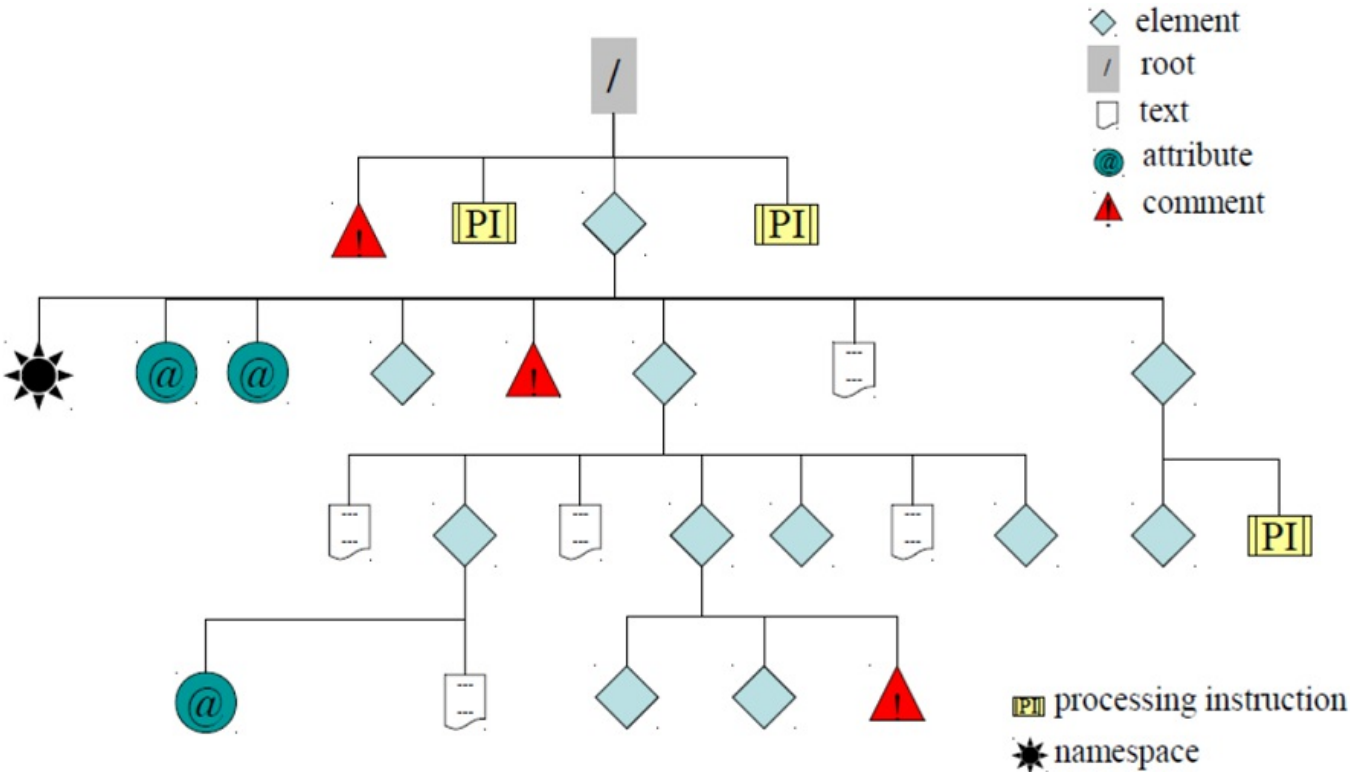
- Read your basic XML file with SAX and use your DefaultHandler to print the contents on the screen
- Note:
  - You can also import the XML in a database by using SAX and JDBC (if time permits)
  - Tip: Divide your application in layers by creating an architecture first
  - Tip: Use the movies.xml provided by your instructor

6. DOM: Document Object Model

6.1. Document Object Model

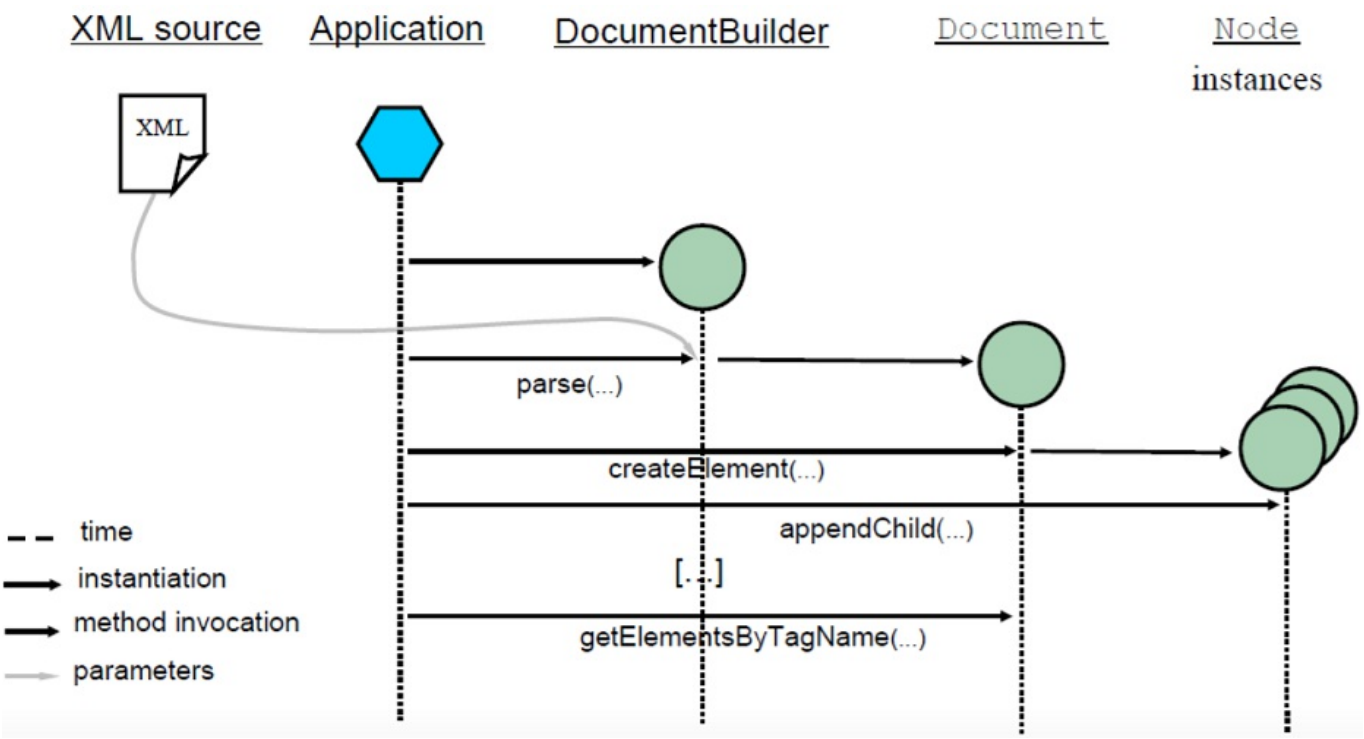
- DOM (Document Object Model) is a general-purpose tree-based API
  - XML document is represented in memory (RAM) as a tree of nodes (objects)
  - Nodes have an extensive set of methods for selection, retrieval and manipulation
  - Reflects the hierarchy of data in an XML document
- Document is an XML term for "A fragment of XML"
  - A full XML document
  - A subtree of an XML document

6.2. Tree View of an XML Document



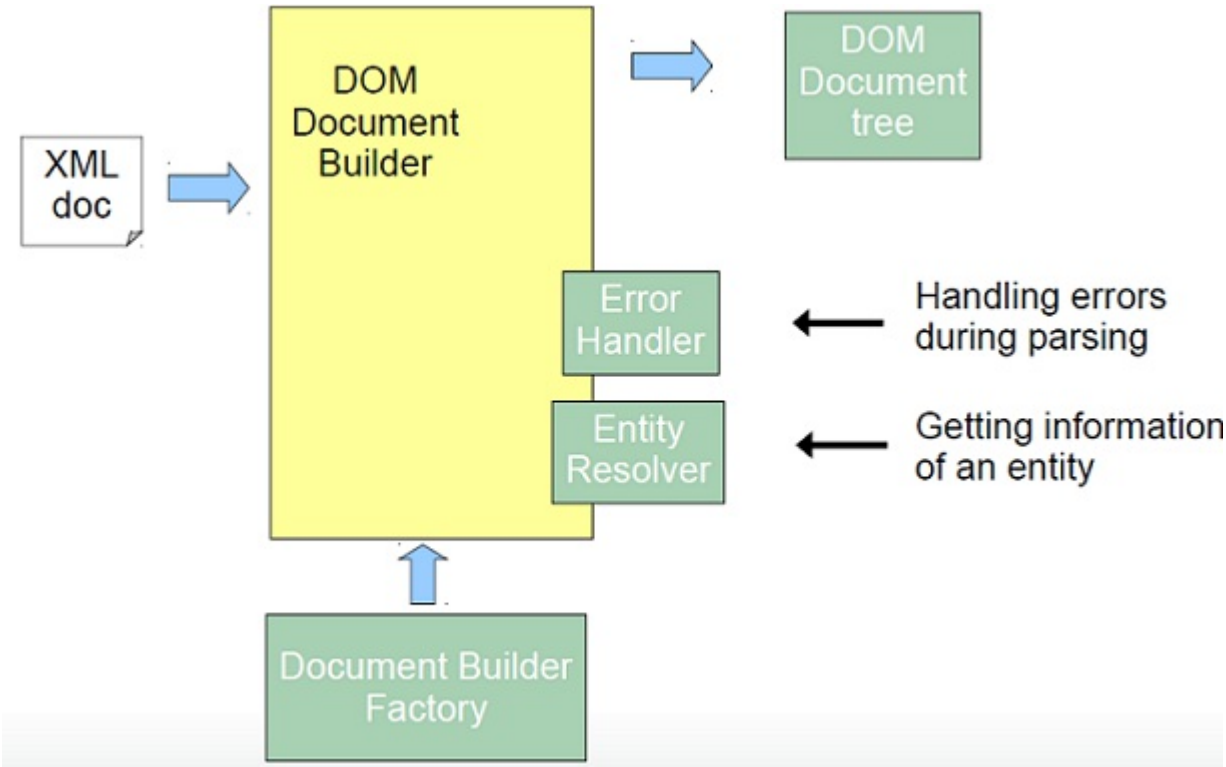
6.3. Anatomy of a DOM application

- This sequence diagram shows how the DOM API is used



6.4. DOM Structure

- Using a `DocumentBuilder` an XML document is parsed to a `Document`
  - `DocumentBuilder` is obtained from a `DocumentBuilderFactory`



- Reuses some of SAX's interfaces `ErrorHandler`, `EntityResolver`

6.5. DocumentBuilder instantiation

- Instantiation of a `DocumentBuilderFactory`

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
```

- Instance methods of `DocumentBuilderFactory` (optional)
  - To obtain a validating parser
  - To obtain a parser that recognizes XML namespaces

```
// Enable validation using XSD
dbf.setSchema(SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_INSTANCE_NS_URI)
        .newSchema(new File("myschema.xsd")));
```

```
dbf.setNamespaceAware(true);
```

- Instantiation of a `DocumentBuilder`

```
DocumentBuilder db = dbf.newDocumentBuilder();  // Workhorse!
```

6.6. Parsing a DOM Document

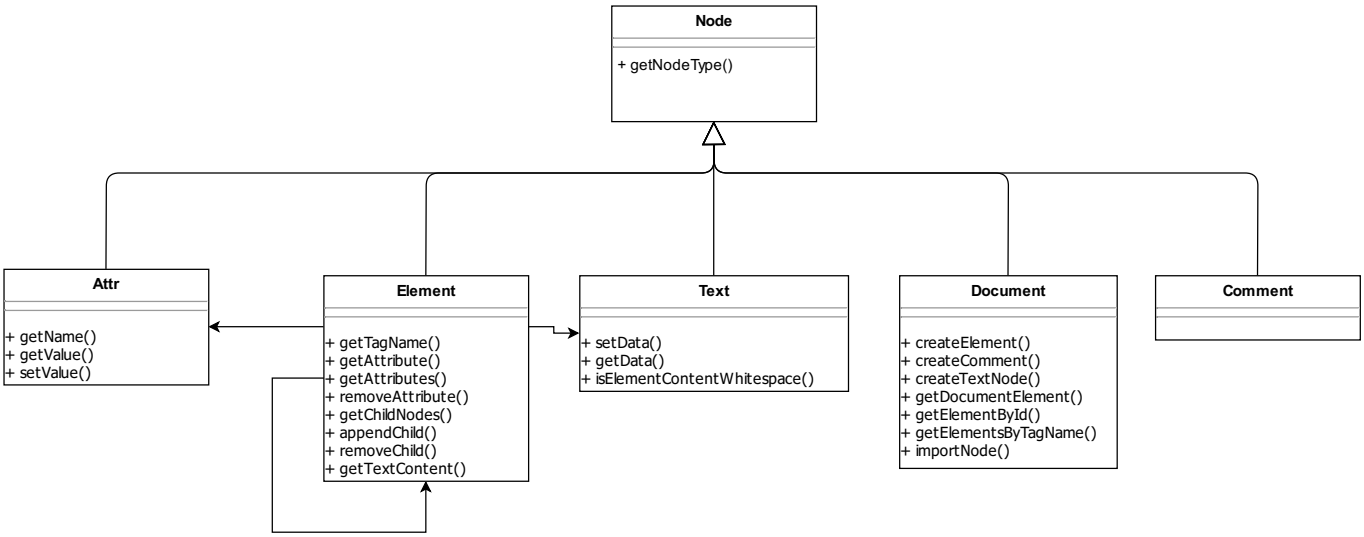
- A document can be parsed using `parse()`

```
Document d = db.parse("file.xml"); // Also allows input File, InputStream
```

- `Document` is the DOM tree's root
  - Not the same as the root element
  - `Document` may contain
    - Comments (e.g. `<!-- x -->`)
    - Processing instructions (e.g. `<? x ?>`)
    - One root element (e.g. top-level `<element>`)

6.7. DOM Nodes

- Tree nodes are represented as a hierarchy of `Node` objects



- Note: this diagram is simplified for educational purposes
- The most important node types

Node type	Purpose
`Node`	Interface for all node types
`Document`	Represents the entire document. Not the same as the root element
`Element`	Represents an `<element>`. Has child elements and attributes
`Text`	Represents a text node. Terminal.
`Comment`	Represents a comment
`Attr`	Represents an attribute. Has name and value. Terminal.
`ProcessingInstruction`	Represents a processing instruction `<? x ?>`
`Entity`	Represents an XML entity `&euro;`
...	...

6.8. DOM Navigation

- Both `Document` and `Element` contain methods to walk through the tree

Method	Purpose
`getElementById()`	Searches descendant element with ID attribute (Not name `id`, see `Attr.isId`)
`getElementsByTagName()`	Searches all descendant elements with a given tag name. Returns `NodeList`
`getChildNodes()`	Retrieves all direct children of an element. Returns `NodeList`

`getFirstChild()`	Retrieves an element's first child
`getLastChild()`	Retrieves an element's last child

```
Element tracks = (Element) d.getElementsByTagName("tracks").item(0);
NodeList nodes = tracks.getChildNodes();
for(int i = 0; i < nodes.getLength(); i++) {
    String title = nodes.item(i).getTextContent();
    System.out.println(title);
}
```

6.9. Document Interface

- `Document` contains `create*()` methods to build new nodes

Method	Purpose
`createElement()`	Creates a new `<element>` node
`createAttribute()`	Creates a new attribute node
`createTextNode()`	Creates a text node
`createComment()`	Creates a comment node

- A node can then be added to the tree using `appendChild()`, `replaceChild()` or `insertBefore()`

```
// <track artist='Jimi Hendrix'>Hey Joe</track>
Element t = d.createElement("track");
Attr a = d.createAttribute("artist");
a.setValue("Jimi Hendrix");
t.getAttributes().setNamedItem(a);
t.appendChild(d.createTextNode("Hey Joe"));
```

- `importNode()` can be used to move a node between documents

6.10. DOM Exception Handling

- When an error occurs during parsing, a `SAXException` is thrown
  - This is because the parse step uses a SAX parser behind the scenes
  - Handled the same way as in a SAX parser: `ErrorHandler`

```
db.setErrorHandler(new org.xml.sax.ErrorHandler() {
    public void fatalError(SAXParseException fe) throws SAXException {
        // Triggered when document is not well-formed
    }

    public void error(SAXParseException e) throws SAXException {
```

```
        // Triggered when document does not pass validation
    }

    public void warning(SAXParseException w) throws SAXException {
        // Triggered when a non critical issue happens
    }
});
```

- When an error occurs during modification of the DOM tree, a `DOMException` is thrown
  - Raised during execution of operations on nodes
  - Reports error conditions specific to XML (e.g. `appendChild()` on a text node)

```
Node node = d.createTextNode("Hello World");
node.appendChild(d.createElement("foo"));       // No no no! Can't add elements to
text node!
                                                // throw DOMException
```

```
org.w3c.dom.DOMException: HIERARCHY_REQUEST_ERR:
An attempt was made to insert a node where it is not permitted.
```

6.11. DOM Example

- Consider the following XML structure:

```
<tracks>
    <track artist="Jimi Hendrix">Hey Joe</track>
    <track artist="Jimi Hendrix">Purple Haze</track>
    <track artist="Janis Joplin">Me And Bobby McGee</track>
    <track artist="Amy Winehouse">Rehab</track>
</tracks>
```

- The following program prints out the tracks

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse("tracks.xml");
NodeList trackNodes = d.getElementsByTagName("track");
for(int i = 0; i < trackNodes.getLength(); i++) {
    Element track = (Element)trackNodes.item(i);
    String artist = track.getAttribute("artist");
    String title = track.getTextContent();
    System.out.printf("%s\t%s\n", artist, title);
}
```

- Another example that adds a track to the document

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder db = dbf.newDocumentBuilder();
Document d = db.parse("tracks.xml");
```

```
Element tracksElement = d.getDocumentElement();
Element trackElement = d.createElement("track");
Attr artistAttribute = d.createAttribute("artist");
artistAttribute.setValue("Janis Joplin");
trackElement.getAttributes().setNamedItem(artistAttribute);
trackElement.setNodeValue("Try (Just a Little Bit Harder)");
tracksElement.appendChild(trackElement);
```

- Note: you can not save a file using the DOM API. See transformers later.

6.12. DOM Levels

- The DOM API has had some updates over the years
  - Referred to as level 1, 2, 3 and 4

Level	Feature
Level 1	Core support for both XML and HTML representation
Level 2	Support for event listeners on some DOM nodes
	Added support for DOM traversal (`getElementBy*()`)
	Support for CSS style access
	Support for interaction between CSS and XML
Level 3	Support for loading and saving
	In-memory DOM validation
	Support for XPath
Level 4	2015 snapshot of W3C's WHATWG living standard

- The contents of this chapter included features from different levels

6.13. Exercise: DOM

- Use DOM to read your XML as a tree in memory
- Adapt the DOM tree by adding child elements to some of the nodes of your document
- Use abstraction by creating classes that perform the tasks for you
- Note:
  - Tip: You will have to wait for TrAX or JDOM to serialize your DOM document, but you can already prepare the code
  - Tip: Add score and description elements to the movies

7. StAX: Streaming API for XML

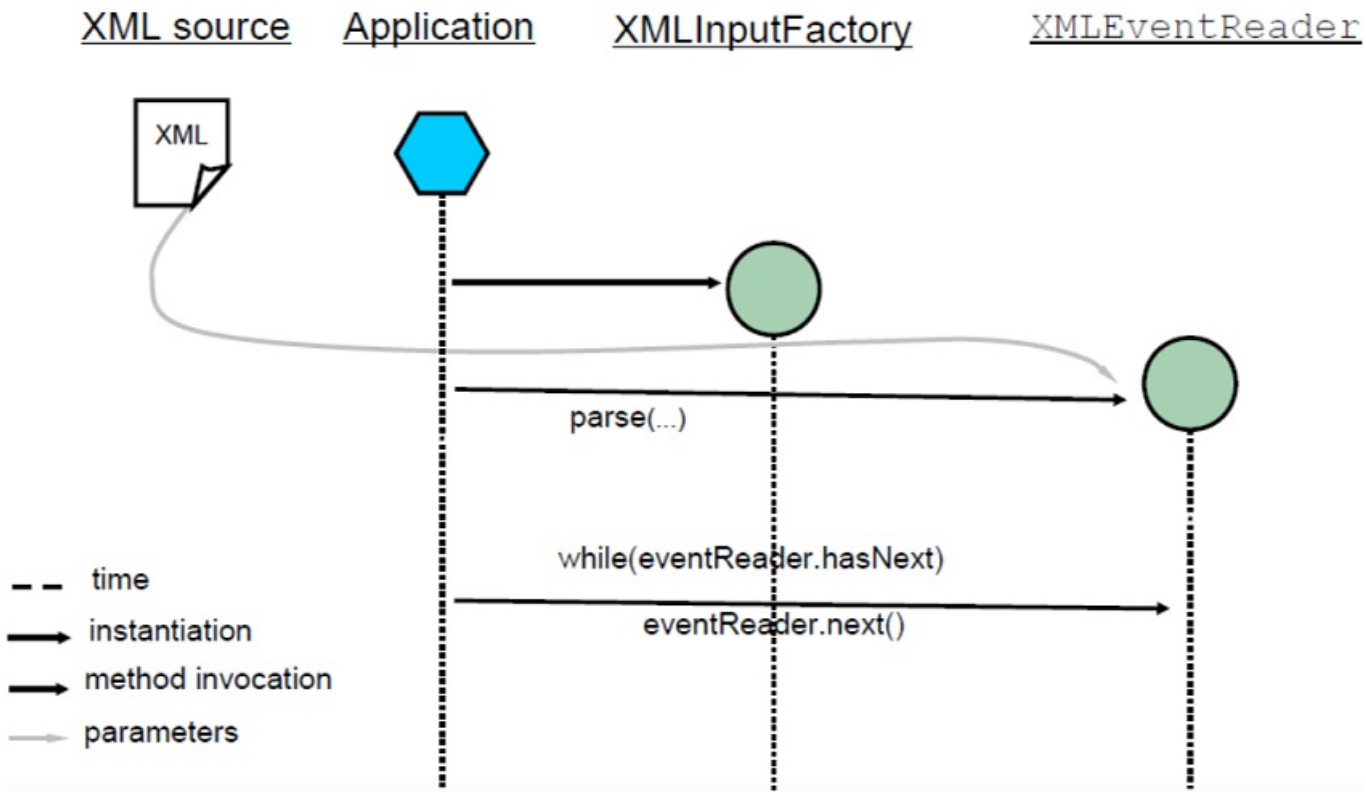
7.1. Pull-type streaming API: StAX

- Pull-type streaming API's are similar to push-type streaming API's but differ in the way they interact with your program
- With pull-type API's, your code will ask the parser for the next event, instead of being notified by the parser
- Advantages
  - No need to implement special interfaces
  - Code will be more concise and easier to read
- StAX (Streaming API for XML) is a pull-type streaming API
- StAX provides a set of classes for writing XML documents

7.2. Obtaining StAX

- Is included in Java SE 6
- A reference implementation can be downloaded from http://stax.codehaus.org

7.3. Anatomy of a StAX application



7.4. Reading XML using StAX

- Two choices
  - `XMLStreamReader`
  - `XMLEventReader`

```
FileInputStream input = new FileInputStream(new File('input.xml'));
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLEventReader reader = inputFactory.createXMLEventReader(input);
while(reader.hasNext()){
    XMLEvent event = reader.nextEvent();
    // handle events here
}
```

```
FileInputStream input = new FileInputStream(new File('input.xml'));
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader streamReader = inputFactory.createXMLStreamReader(input);
while (streamReader.hasNext()){
    int eventType = streamReader.next();
    // handle events here
}
```

7.5. StAX Event Types

Event type ID	Event type name	Event interface name
1	START_ELEMENT	StartElement
2	END_ELEMENT	EndElement
3	PROCESSING_INSTRUCTION	ProcessingInstruction
4	CHARACTERS	Characters
5	COMMENT	Comment
6	SPACE	Characters
7	START_DOCUMENT	StartDocument
8	END_DOCUMENT	EndDocument
9	ENTITY_REFERENCE	EntityReference
10	ATTRIBUTE	Attribute
11	DTD	DTD
12	CDATA	Characters
13	NAMESPACE	n/a
14	NOTATION_DECLARATION	NotationDeclaration
15	ENTITY_DECLARATION	EntityDeclaration

	ENTITY_DECLARATION	EntityDeclaration

7.6. Testing Event Types

- `XMLStreamReader`

```
XMLStreamReader r // ...
if(reader.getEventType() == XMLStreamConstant.START_ELEMENT) {
    // ...
}
```

- `XMLEventReader`

```
// using ==
if (event.getEventType( ) == XMLStreamConstant.START_ELEMENT){
    System.out.println("I'm a start element event!");
}

// using isStartElement
if (event.isStartElement( )) {
    System.out.println("I'm a start element event!");
}

// using instanceof
if (event instanceof StartElement) {
    System.out.println("I'm a start element event!");
}
```

7.7. Extracting Data

- Extracting data from an event

```
// XMLStreamReader
XMLStreamReader reader = // ...;
String elementText = reader.getElementText();
String val = reader.getAttributeValue("ns", "attrName");
```

```
// XMLEventReader
XMLEvent event = // ...;
QName elementName = event.asStartElement().getName();
event.asStartElement().getAttributeByName(new QName("attrName"));
String contents = event.asCharacters().getData();
```

7.8. Writing XML Using StAX

- StAX can also be used to write XML output

- Using `XMLStreamWriter`
- Using `XMLEventWriter`

- `XMLStreamWriter`

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLStreamWriter writer = outputFactory.createXMLStreamWriter(System.out); // Or any
other out stream
writer.writeStartDocument("1.0");

writer.writeStartElement("addresses");              // <addresses>
writer.writeStartElement("address");                // <address
writer.writeAttribute("type", "work");              // type="work">
writer.writeStartElement("street");                 // <street>
writer.writeCharacters("1515 Broadway");            // 1515 Broadway
writer.writeComment("in the heart of Times Square");// <!-- in the heart of Times
Square -->
writer.writeEndElement();                           // </street>
writer.writeEndElement();                           // </address>
writer.writeEndElement();                           // </addresses>

writer.writeEndDocument();
writer.flush();                                     // Important if underlying stream
is buffered
```

- `XMLEventWriter`

```
XMLEventFactory eventFactory = XMLEventFactory.newInstance();
XMLOutputFactory output = XMLOutputFactory.newInstance();
XMLEventWriter xmlwriter = output.createXMLEventWriter(System.out);
xmlwriter.add(eventFactory.createStartDocument("UTF-8", "1.0");

// Write contents (see next slide)

xmlwriter.add(eventFactory.createEndDocument());
xmlwriter.flush();  // When underlying stream is buffered
xmlwriter.close();
```

```
// Attributes
Attribute att = eventFactory.createAttribute("year", "2003");
ArrayList attArr = new ArrayList();
attArr.add(att);

// Namespaces
Namespace namespace = eventFactory.createNamespace("foo","http://www.foo.org");
ArrayList nameArr = new ArrayList();
nameArr.add(namespace);

// Qualified names (element + namespace)
QName qname = new Qname("http://www.foo.org", "HockeyTeam", "foo");

// Create and write elements
xmlwriter.add(eventFactory.createStartElement(qname, attArr.iterator(),
nameArr.iterator()));
xmlwriter.add(eventFactory.createCharacters("Los Angeles Kings"));
xmlwriter.add(eventFactory.createEndElement(qname));
```

- `XMLStreamReader` and `XMLStreamWriter` (cursor API) is more efficient
- `XMLEventReader` and `XMLEventWriter` (iterator API) is more flexible

7.9. Exercise: StAX

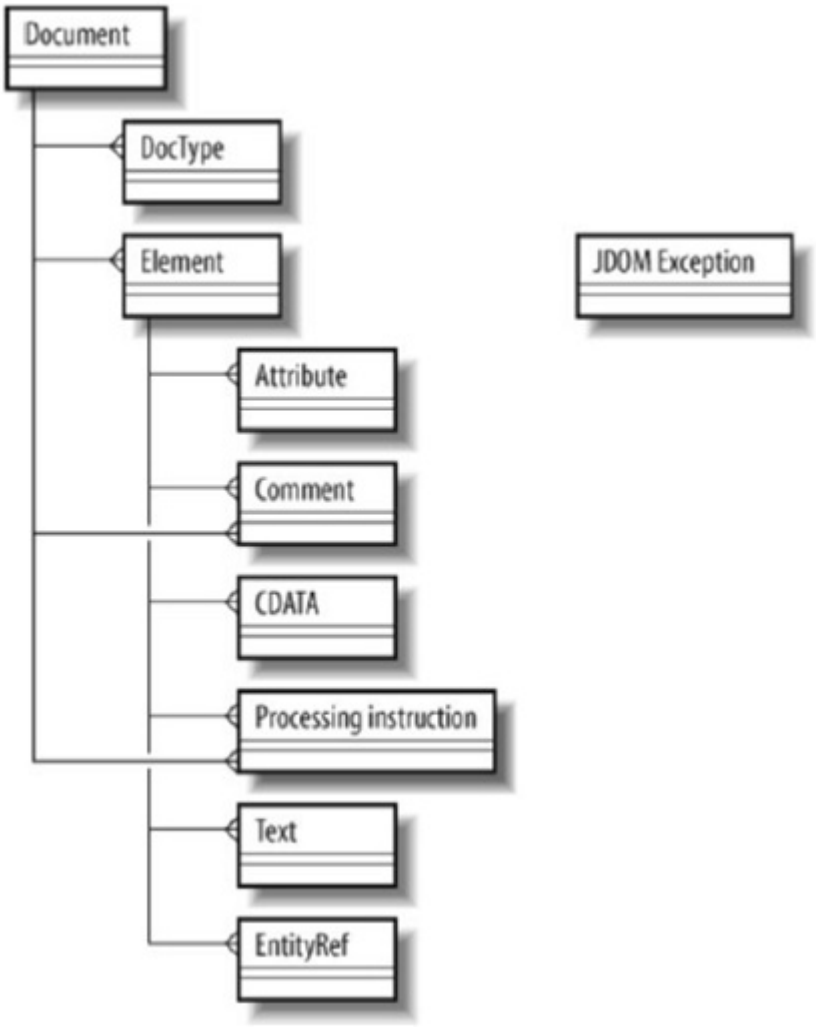
- Read your XML using StAX
- Accumulate and perform statistics on your document (how much elements of a type, etc...)
- Write your XML back into another document with StAX document output
- Note:
  - Tip: You could also use StAX and combine it with your SAX application to import XML to your database (or not)
  - Tip: Calculate the total number of movies, and number of movies by category

8. JDOM: Java Document Object Model

8.1. Why another tree-based API?

- JDOM was built specifically for Java (DOM is more generic)
- It is more intuitive for a Java developer than DOM
- JDOM is open source, so you can suggest and implement changes yourself
- JDOM uses standard Java collections instead of `NodeList` or `NamedNodeMap`
- JDOM uses concrete classes that you can instantiate with new, instead of calling the `DocumentBuilderFactory` or `DocumentBuilder`

8.2. The core JDOM classes



8.3. Creating Elements

- You can create elements with their constructors

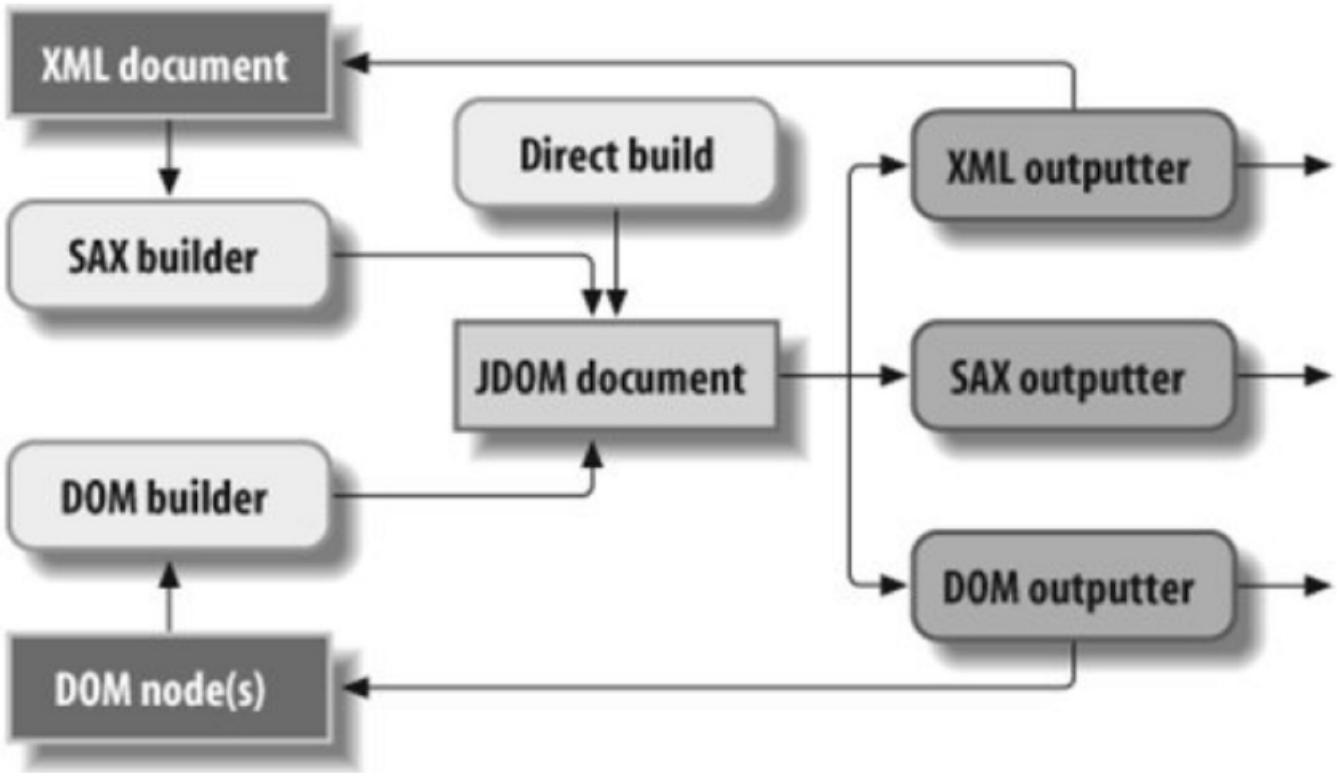
```
Element rootElement = new Element("root");
```

```
Document document = new Document(rootElement);
```

- Use setters and adders to build a document tree

```
Document doc = new Document(
    new Element("root")
        .setAttribute("attribute", "value")
        .addContent(new Element("inner"))
        .addContent( new Comment("comment text"))
        .addContent("some inline text")
        .addContent(new Element("inner2"))
);
```

8.4. Input and output



8.5. Input examples

- Using `SAXBuilder` to load a document directly from a stream

```
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(new FileInputStream("contents.xml"));
```

- Using `DOMBuilder` when you already have a W3C DOM structure

```
DOMBuilder builder = new DOMBuilder();
Document doc = builder.build(myDomDocumentObject);
```

8.6. Output examples

- To output back to a file

```
XMLOutputter outputter = new XMLOutputter( );
outputter.output(jdomDocumentObject, new FileOutputStream("results.xml"));
```

- Outputters support formatting
  - Useful for "pretty print"

```
Format format = Format.getPrettyFormat( );
XMLOutputter outputter = new XMLOutputter(format);
FileOutputStream output = new FileOutputStream(xmlFilename);
outputter.output(doc, output);
```

- Alternatively use `SAXOutputter` or `DOMOutputter` to convert back into W3C API models

8.7. Creating documents and elements

- Start building a document tree from scratch

```
Element root = new Element("movies");
Document doc = new Document(root);
```

- Creating an element

```
Element element = new Element(propertyName);
element.setText(propertyValue);
root.addContent(element);
```

- Adding attributes

```
Attribute attribute = new Attribute("value", propertyValue);
last.setAttribute(attribute);
current.addContent(last);
```

8.8. Reading and accessing elements

- Using standard Java `Collection` and `Iterator`

```
List<Element> elements = doc.getRootElement().getChildren();
for(Element current : elements) {
    String name = current.getName( );
    String text = current.getTextNormalize( );
    if ((text == null) || (text.equals(""))) {
        List children = current.getChildren();
        // new loop or recursion here
    }
}
```

- Much more convient to work with compared to W3C DOM model

8.9. The Namespace Class examples

- Namespaces are created using a cached namespace pool
  - Access through static `Namespace.getNamespace()`

```
// Create namespace with prefix
Namespace realdolmenNamespace = Namespace.getNamespace("m",
"http://www.realdolmen.com/movies");

// Create namespace without prefix
Namespace namelessNamespace =
Namespace.getNamespace("http://www.realdolmen.com/movies");
```

- Using namespaces

```
// Adding namespace declaration (e.g. xmlns:m="http://www.realdolmen.com/movies")
Element m = new Element("movies");
e.addNamespaceDeclaration(realdolmenNamespace);

// Creating elements with namespace (e.g. <m:actor> if qualified)
Element a = new Element("actor", realdolmenNamespace);

// Filtering using namespaces (e.g. list elements <m:chapter> if qualified)
List chapterElements = contentElement.getChildren("chapter", realdolmenNamespace);
```

8.10. Other possibilities of JDOM

- You can create your own subclass of `Element` etc...
- You can use JDOM to perform transformations similar to TrAX
- You can use JDOM to perform XPath expressions on the tree-structure
- You can still use factories and overwrite/replace JDOM standard factories
  - Gives you more flexibility in how your tree is modeled

8.11. Exercise: JDOM

- Adapt your previous exercise to write your new file to disk with JDOM
- Read your new XML file with JDOM
- Create classes that will abstract the creation of new documents, to add elements, to remove elements, etc... (think Factory, Builder, ...)
- Use JDOM to change your source XML by adding extra child elements
- Write your new XML file to disk at the end
- Note:
  - Tip: Add user comments to your movies

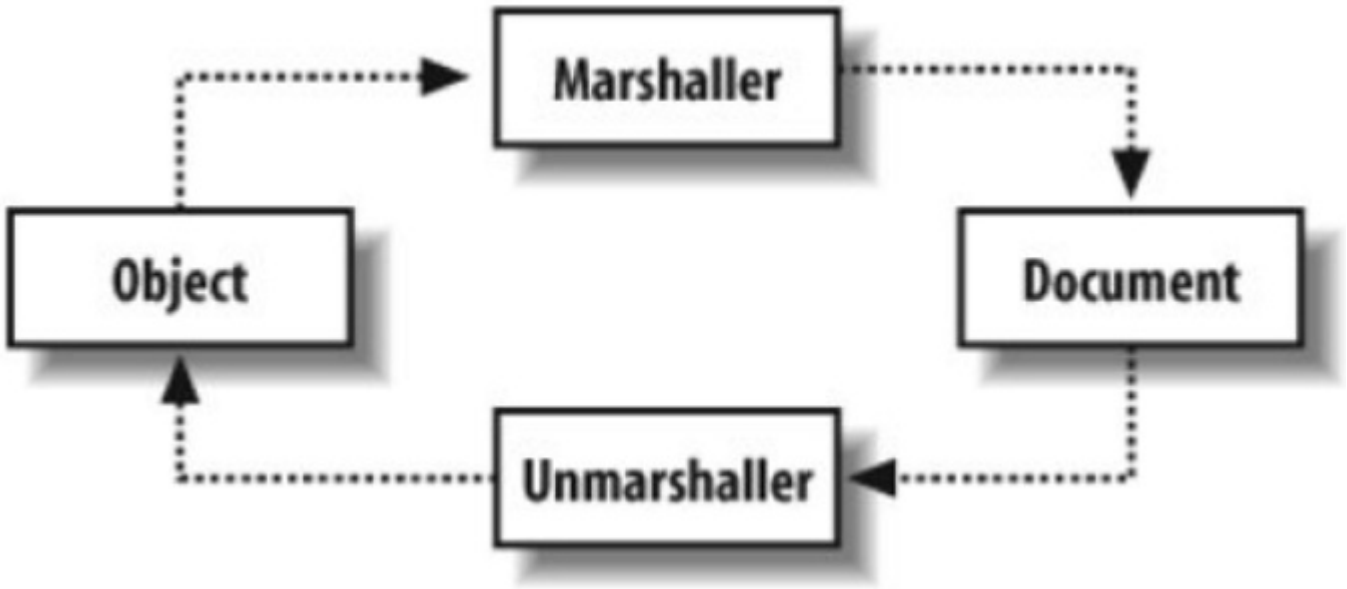
9. JAXB: Java Architecture for XML Binding

9.1. JAXB: Java Architecture for XML Binding

- Data binding eliminates references to XML nodes from your code
- You will first define the structure of your XML with an XSD schema
- Then you will bind this schema to specific Java classes using JAXB
  - Uses code generation tools (xjc) to generate Java from XSD schema
  - Uses a runtime library to convert XML documents into Java objects and back
- Schema and XML can be generated through annotations

9.2. Marshalling and unmarshalling

- Instead of parsing or serializing documents, you marshall and unmarshall
- You will not use generic interfaces, but specific classes that have a meaning beyond the XML document
  - E.g. not `Element` but your own classes (`Movie`, `Actor`, ...)
- Use the `Marshaller` and `Unmarshaller` components



- Consider this XML document

```
<?xml version="1.0"?>
<person xmlns="http://www.realdolmen.com/people">
    <firstName>Ganoes</firstName>
    <lastName>Paran</firstName>
</person>
```

```
JAXBContext context = JAXBContext.newInstance("com.realdolmen.movies.xml");

// Marshalling is converting from Java objects to XML
```

```
Marshaller marshaller = context.createMarshaller();
Person person = new Person( );
person.setFirstName("Ganoes");
person.setLastName("Paran");
marshaller.marshal(person, new FileWriter("person.xml")); // Write Person to XML
document

// Unmarshalling is converting from XML to Java objects
Unmarshaller u = context.createUnmarshaller();
Person person = (Person) unmarshaller.unmarshal(new File("people.xml"));
System.out.println(person.getFirstName()); // Conversion all done for you behind the
scenes!
```

9.3. Data binding, metadata and Schemas

- To tell the data binding mechanism how to proceed, you need to configure it
  - Add metadata in static fields and methods
  - Add metadata with Java annotations
  - Use an external mapping configuration file (itself in XML)
- Data binding also need a Schema
  - Documents produced and consumed must conform to it
  - Used to validate the result
  - May be able to generate the Java classes from the Schema

9.4. When to use data binding

- Data binding helps to allow application to move data between Java classes and XML
- Both the Java classes and XML are defined in some way
- Do not use data binding when
  - You do not have a Schema (work contract first!)
  - You have a Schema that uses mixed content
  - You are dealing with large documents (memory, ...)

9.5. JAXB Features

- Defines a standardized API for marshalling and unmarshalling as well as validation
- Defines how a schema compiler binds a Schema to a Java representation
- Implementations can package their own schema compiler
- The schema compiler will produce packages, interfaces, an `ObjectFactory`, and classes for enumerations
- Generates POJOs with annotations
- Annotations allow binding of arbitrary classes

- Can generate Schemas from Java classes
- Portability increases due to POJOs
- Integration with StAX
- Integration with JAXP validation
- Complete W3C XML Schema support
- Callback methods as listeners when needed

9.6. Finding JAXB

- The JAXB 2.0 reference implementation is open source and can be downloaded from
  - https://jaxb.dev.java.net/
- JAXB 2.0 is included in JEE 6 compliant servers and is used by JAX-WS and JAX-RS

9.7. Working with JAXB: the JAXBContext

- The JAXBContext enables you to obtain instances of Marshaller, Unmarshaller or Validator interfaces
- You need to have a compiled schema in the package

```
JAXBContext context =
    JAXBContext.newInstance("com.realdolmen.person");
Marshaller marshaller = context.createMarshaller( );
Unmarshaller unmarshaller = context.createUnmarshaller( );
Validator validator = context.createValidator( );
```

9.8. Compiling a Schema

- Most implementations provide an Ant task or commandline application, called XJCTask or xjc respectively
- Example for compiling to src directory with user provided namespace

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
    <xs:element name="person" type="atype"/>
        <xs:complexType name="atype">
        <xs:sequence>
            <xs:element name="firstName" type="xs:string"/>
            <xs:element name="lastName" type="xs:string"/>
        </xs:sequence>
        </xs:complexType>
</xs:schema>
xjc -p com.realdolmen.person -d src person.xsd
```

9.9. The inverse: Creating a Schema from class

- With JAXB 2.0 it is possible to create a schema from annotated classes
- These are Java 5 annotations
- Simplest example

```
package com.realdolmen.person;
import javax.xml.bind.annotation.XmlRootElement;
@XmlRootElement
public class Person {
    private String firstName;
    private String lastName;
    public String getFirstName( ) { return firstName; }
    public String getLastName( ) { return lastName; }
    public void setFirstName(String s) { firstName = s; }
    public void setLastName(String s) { lastName = s; }
}
```

9.10. Simple marshalling

```
ObjectFactory factory = new ObjectFactory( );
// return new Person()
Person person = factory.createPerson( );
person.setFirstName("Alan");
person.setLastName("Turing");
JAXBContext context =
    JAXBContext.newInstance("com.realdolmen.person");
Marshaller marshaller = context.createMarshaller( );
marshaller.marshal(person, System.out);
```

9.11. Simple unmarshalling

```
<?xml version="1.0" encoding="UTF-8"?>
<person>
        <firstName>James</firstName>
        <lastName>Gosling</lastName>
</person>
// more code
JAXBContext context =
    JAXBContext.newInstance("com.realdolmen.person");
Unmarshaller unmarshaller = context.createUnmarshaller( );
Person person = (Person) unmarshaller.unmarshal(new
    File("james.xml"));
System.out.println(person.getFirstName( ));
```

9.12. Schema validation when marshalling

```
Person p = new Person( );
p.setFirstName("Burt"); // incomplete person!
JAXBContext context =
    JAXBContext.newInstance(Person.class);
Marshaller marshaller = context.createMarshaller( );
SchemaFactory sf = SchemaFactory
    .newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = sf.newSchema(new File("person.xsd"));
marshaller.setSchema(schema);
marshaller.marshal(p, System.out); // throws an Exception
```

9.13. More advanced JAXB uses

- Next to @XMLRootElement, other annotations let you customize the Schema generation
  - @XMLElement: change element names
  - @XMLAttribute: declare as attributes
  - @XMLElementWrapper: wrap an element around others
  - @XMLAccessorType: annotations on fields or getters?
  - @XMLTransient: do not marshall to XML
  - @XMLType: lets you set the order of properties
  - ...
- The Marshaller is capable of pretty-printing the XML result
- Use validation events to see where the validation problem occurred
- Use JAXB callback classes to listen to marshal and unmarshal events

9.14. Exercise: JAXB

- Create Java objects with annotations corresponding to your XML structure
- Try to unmarshal your XML file to your object tree
- Try to make changes to the object tree, and marshall back to XML
- Confirm the changes in both directions
- Note
  - Tip: Change the score of all movies back to 0 and marshal to another XML file
  - Tip: Update the categories of all movies by choosing different names for each category