Bernd Owsnicki-Klewe

Distributed Software Architectures

1. XML

1. Introduction to XML

XML (the eXtensible Markup Language) is a very general and widely used “Markup Language” [http://en.wikipedia.org/wiki/Markup_language]. Examples of such Markup Languages – aside from XML – are the TeX language [http://en.wikipedia.org/wiki/TeX] and HTML [http://en.wikipedia.org/wiki/HTML].

The main reason to include a section about XML stems from the fact that it's nowadays

- useful for transmitting data from servers to clients (→ AJAX)
- a vehicle of communication in message-oriented Distributed Architectures, like Web Services, and
- used to describe software configurations, especially of a web application on a server.

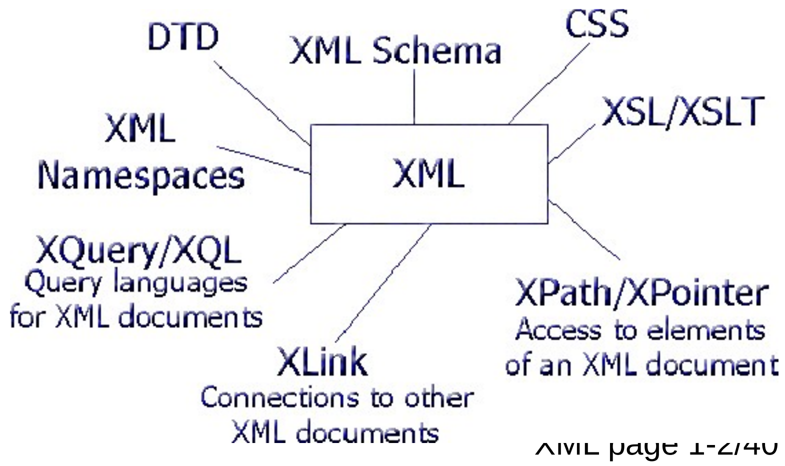
If you're proficient in HTML, you already know some things about the internal structure of an XML document: it's built from text (the content) and “tags” describing the use or meaning of the enclosed text. As in HTML these tags are enclosed in opening and closing angle brackets (“<” and “>”).

1.1 XML Philosophy

Before diving into the details of XML, it seems to be in order to try to describe some of the basic ideas underlying the development of this formalism.

- **XML documents carry no meaning at all.** This is probably the most important idea; any meaning ascribed to an XML document rests solely in the application that works on it. All there is to an XML document are the concepts of “well-formedness” and “validity”. Technically speaking, these are syntactical rather than semantical properties.

  Semantics can be provided by using structures on top of XML, lile RDF (the Resource Description Framework,


DTD
XML Schema
CSS
XML Namespaces
XML
XSL/XSLT
XQuery/XQL
Query languages for XML documents
XLink
Connections to other XML documents
XPath/XPointer
Access to elements of an XML document

http://en.wikipedia.org/wiki/Resource_Description_Framework) or OWL (the Web Ontology Language, http://en.wikipedia.org/wiki/Web_Ontology_Language [sic!]).

- **XML defines just a flexible and non-committing syntactical framework.** It was the goal of the XML development to not specify any committing implementation aspects for XML constructs. This is most obvious in the very abstract – but very flexible – notions involving URIs and Namespaces. We will get there sooner or later!

- **XML is a coordinated set of languages that build a framework to describe all kinds of documents.** XML itself is only the core part of a family of concepts and languages. We will walk through some of them here. The figure above shows some of the most important members of the XML framework.

1.2 XML Documents

An XML file is generally called a *document*. A document has two parts:

1. The **prolog,** describing the XML version used, character set, DTD/Schema to be used, style information, like CSS or XSL, etc.
2. An **element**, the main "payload" of the document, describing whatever the document is supposed to describe.

Here we will concentrate on the XML element, because that's the important part that's actually used by an application. The prolog can vary according to the specific circumstances, but generally looks somehow like this example:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE recipe SYSTEM “recipe.dtd”>
<?xml-stylesheet type="text/xsl" href="style.xsl"?>
```

Line 1 describes the XML version – 1.0 – and the character set; line 2 specifies that the DTD (see below) for this document is located in the reciple.dtd file; line 3 associates the file style.xls as the current style sheet for this document. We're not dealing with styles at this point in the course, but compare the remarks about style at the end of this chapter.

1.3 XML Building Blocks

As shown in the following small example – taken from http://en.wikipedia.org/wiki/XML – XML is very similar to HTML. Indeed HTML can be viewed as a "sloppy specific dialect" of XML. More about that later.

The main differences between XML and HTML can be summarized like this:

1. Tag names are generally case-sensitive and lowercase names are encouraged.
2. For each opening tag there must be a closing tag.
3. There is no predefined set of tag names – whoever is in charge of an XML document is free to create tag names as necessary.
4. Attribute values must be double-quoted.
5. There is a strict notion of well-formedness associated with XML documents, based on syntax rules that describe how tags are to be inserted into a document.
6. The structure of tags for a particular document type can be defined externally and the document can be validated against this structure definition.

```
<recipe name="bread" prep_time="5 mins" cook_time="3 hours">
   <title>Basic bread</title>
   <ingredient amount="3" unit="cups">Flour</ingredient>
   <ingredient amount="0.25" unit="ounce">Yeast</ingredient>
   <ingredient amount="1.5" unit="cups" state="warm">Water</ingredient>
   <ingredient amount="1" unit="teaspoon">Salt</ingredient>
   <instructions>
     <step>Mix all ingredients together.</step>
     <step>Knead thoroughly.</step>
     <step>Cover with a cloth, and leave for one hour in warm room.</step>
     <step>Knead again.</step>
     <step>Place in a bread baking tin.</step>
     <step>Cover with a cloth, and leave for one hour in warm room.</step>
     <step>Bake in the oven at 350(degrees)F for 30 minutes.</step>
   </instructions>
 </recipe>
```

1.3.1 Tags, Elements and Attributes

XML tags come as opening and closing tags – not unlike in HTML, but closing tags are required in any case.

A sequence **<tag>...</tag>** is referred to as an *element.* There are also empty tags, i.e. these tags don't span a part of the document; they are both opening and closing, like **<br />**.[1]

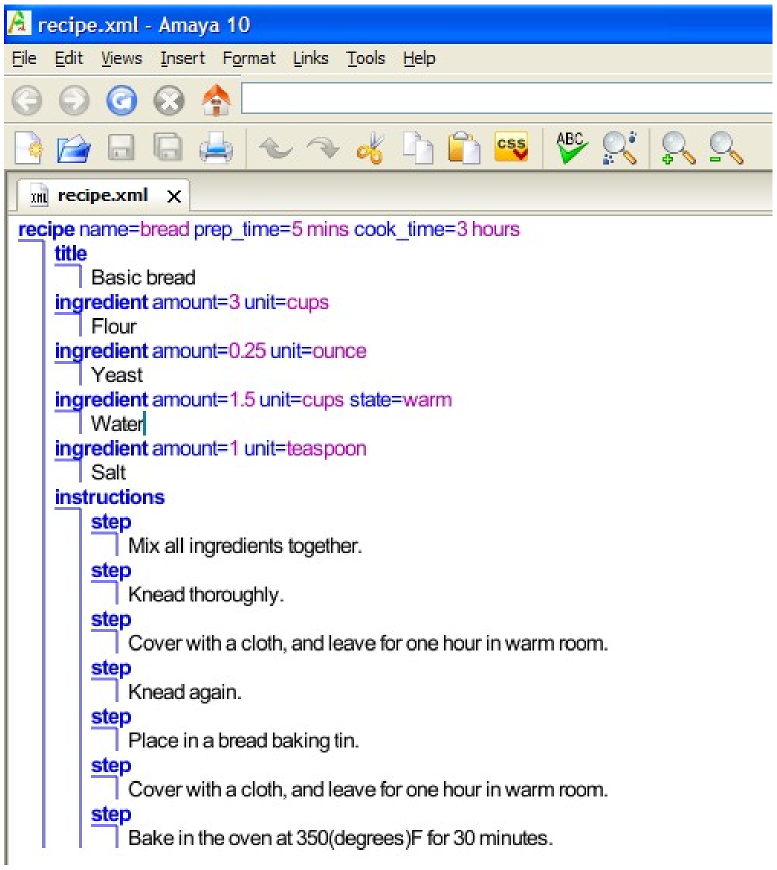
Elements can have one or more attributes. They are defined in the opening tag for this element and are of the form **attributeName = attributeValue**, where the value must be double quoted.[2]

1 It's important to not confuse “tag” and “element”. Tags are symbols that start with “<” and end with “>”; elements are building blocks of the logical structure of an XML document – they are delimited by tags.
2 The use of attributes in an XML document is very much debated – in many cases you can design an XML document in a

1.3.2 Nesting and Well-Formedness

XML Elements can be nested. This sentence describes the essence of well-formedness. The important aspect is that elements can be located inside other elements, or syntactically: if there is a sequence of opening tags **<tag1> .. .<tag2>... <tag3> ...** these tags must be closed in the reverse order, i. e. **</tag3> ... </tag2> ... </tag1>**. In other words, they must follow a Last-In-First-Out (stack) discipline[3].

One of these elements is the outermost one – int this case recipe. A well-formed XML document must have exactly one outermost (top, root) element.



```
recipe name=bread prep_time=5 mins cook_time=3 hours
  title
    Basic bread
  ingredient amount=3 unit=cups
    Flour
  ingredient amount=0.25 unit=ounce
    Yeast
  ingredient amount=1.5 unit=cups state=warm
    Water
  ingredient amount=1 unit=teaspoon
    Salt
  instructions
    step
      Mix all ingredients together.
    step
      Knead thoroughly.
    step
      Cover with a cloth, and leave for one hour in warm room.
    step
      Knead again.
    step
      Place in a bread baking tin.
    step
      Cover with a cloth, and leave for one hour in warm room.
    step
      Bake in the oven at 350(degrees)F for 30 minutes.
```

This convention has the effect that the inner structure of an XML document has the graphical form of a tree.

The illustration on the left shows the tree rendered from the previous example by the Amaya browser (http://www.w3.org/Amaya/).

Tree terminology applies to this model in full – so the “recipe” element is referred to as the *root* element. The “ingredient” elements are *siblings*, the “step” elements are *descendants* of the “instructions” element and so on.

2. The Document Object Model (DOM)

way that attributes are unnecessary, they can be replaced by suitable sub-elements. In fact, attributes can lead to a multitude of formatting issues when dealing with CSS and XSLT (see below).

3 Or to put it in even another way: the document structure can be described by a context-free grammar.

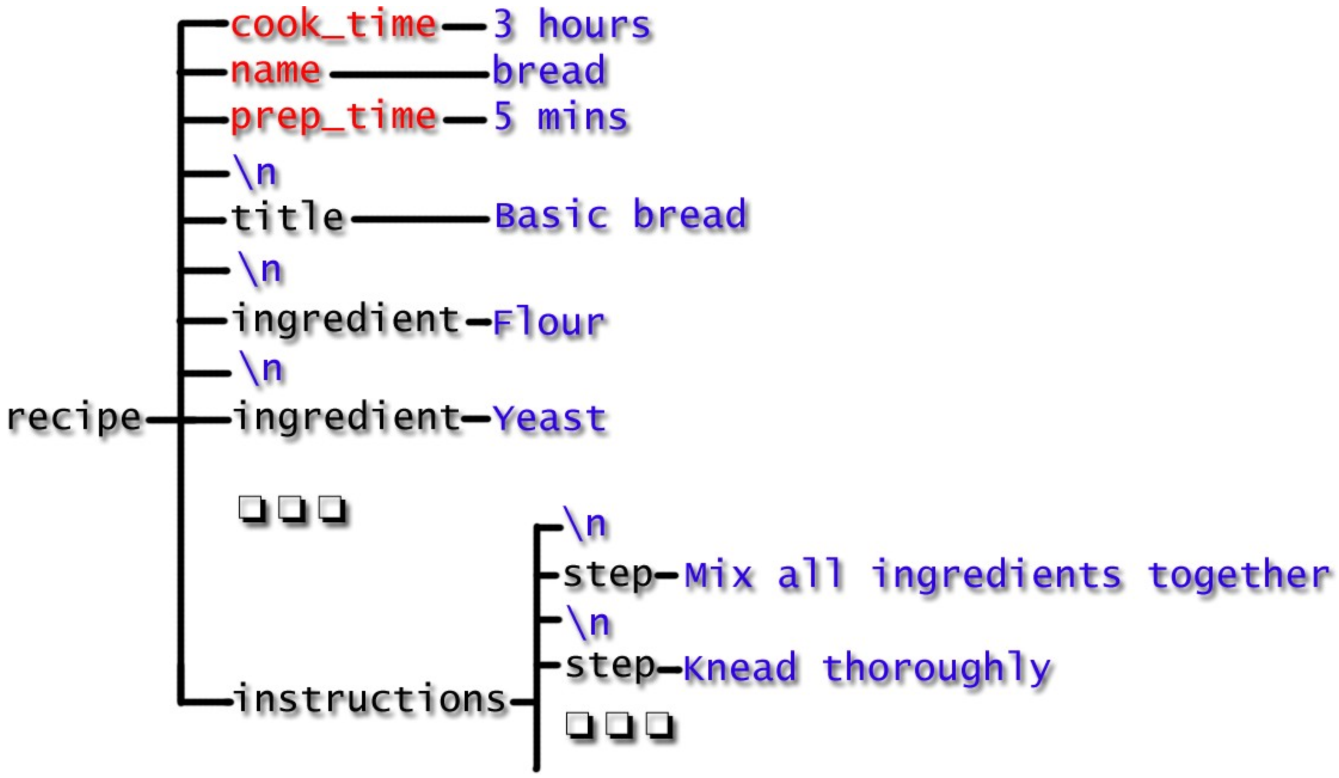
Those readers with some experience in web page design might have heard the term DOM before as the representational structure underlying an XHTML page. This DOM could then be used by scriptig elements, as JavaScript, to dynamically analyze and alter this document.

XML DOM (or the “XML Document Object Model”, but no one uses that name) defines the inner structure and the aces to elements in an XML document. Understanding DOM is one of the keys to accessing XML documents from an application.

DOM is specified by W3C in a number of documents describing the different aspects of DOM in detail. For us we will concentrate on the aspect of an XML DOM as a tree and ways to navigate through this tree.

A deeper study of the DOM specifications reveal a much more detailed picture of an XML DOM that shown by Amaya above. DOM distinguishes a variety or node types, the most important being:

- ELEMENT nodes: corresponding to XML Elements – shown in black
- ATTR nodes: corresponding to attributes of an element – shown in red
- TEXT nodes: for simple text elements – shown in blue



More details of the DOM

In this sketch, some nodes are omitted, like the attribute nodes for the title elements and some repetitions of ingredient and step, which should be self-explanatory.

In addition, the DOM contains some “newline character” (\n) text nodes. They are a result from the use of white space during the construction of the document, e. g. with a normal text editor. Most DOM-aware programs will ignore them (or some of them, at least), but if you want to manually traverse the DOM tree (see below), you should be aware of their presence.[4]

2.1 URIs and URLs

An essential part of XML and related technologies – basically all web technologies, as a matter of fact – is the problem of identifying data in a very general and unambiguous manner. From the daily use of the web, the notion of a Uniform Resource Locator (URL) is probably familiar – but that's only one (and a very special) way to identify resources on the web.

The creators of the underlying naming system stuck to the non-committing philosophy very strictly; that might explain some of the somewhat outlandish conventions in these systems. Let's start:

> *A Uniform Resource Identifier (URI) is a compact sequence of characters that identifies an abstract or physical resource.*
> [IETF RFC 3986, http://tools.ietf.org/html/rfc3986]

Not more, not less! URIs are just for identification purposes, they're not connected with any theoretical, practical or even technical means to find, describe or use this resource! URIs are non-committing in nature.

The true beauty of this definition becomes apparent when the term “resource” is described in this RFC:

(...) the term "resource" is used in a general sense for whatever might be identified by a URI.

Circular definitions at theirs finest! It's actually not too important what a resource can be – the non-committing nature of these definitions can safely leave it open. For us, resources will be anything that is represented by data on a computer system – even though it technically also could refer to physical objects.

4 It's more fun when you're using JavaScript/AJAX to examine an XML DOM on the client side, i. e. from within a browser. Different browsers deal with these white spaces differently, meaning your code might work in Firefox but break in MS Internet Explorer (or vice versa).

Not every string is a URI, though – there is a syntax specifying how it looks:

- Scheme: A string describing the intended use for this resource, generally – but not exclusively – a protocol, like http, ftp, mailto, etc. Scheme names registered with IANA can be found here: http://www.iana.org/assignments/uri-schemes.html
  The Scheme is followed by a colon, separating it from the rest of the URI.
- Scheme-specific part: The second part of the URI describes the resource depending on the scheme.

Here are some examples of URIs

- http://www.cs.uwf.edu/
- http://www.google.com/search?hl=en&q=uri&btnG=Search
- file:/localhost//C|/mydir/myfile.txt
- news:comp.lang.java.programmer
- jdbc:derby://localhost:1527/travel
- urn:isbn:096139210x

In fact, most URIs actually are more than just identifiers – they also contain information how to get to this resource. These URIs are known as URLs:

> *(...) a URL is a type of URI that identifies a resource via a representation of its primary access mechanism (e.g., its network "location"), rather than by some other attributes it may have.*
> [http://www.w3.org/TR/uri-clarification/]

2.2 XML Namespaces

Technically, a namespace (or name space) is used to separate identifiers used in different applications from overlapping – generally these applications are designed by different developers not knowing about the identifiers the other one might use. Namespaces are used in programming languages like C++ or Java (they're called “packages” there) and they're also a part of XML, since many documents will be composed from elements of different sources. Here's an example from http://www.w3schools.com:

```
<table>
  <tr>
  <td>Apples</td>
  <td>Bananas</td>
  </tr>
</table>
```

```
<table>
  <name>African Coffee Table</name>
  <width>80</width>
  <length>120</length>
</table>
```

Both these elements might have DTDs or Schemas associated with them, so using one would result in an ambiguity about which should be used for validation.

Using one of them in an XML document must then specify which namespace the table element belongs to:

```
<root>
<h:table xmlns:h="http://www.w3.org/TR/html4/">
  <h:tr>
  <h:td>Apples</h:td>
  <h:td>Bananas</h:td>
  </h:tr>
</h:table>

<f:table xmlns:f="http://www.w3schools.com/furniture">
  <f:name>African Coffee Table</f:name>
  <f:width>80</f:width>
  <f:length>120</f:length>
</f:table>
</root>
```

A namespace is used in two steps:

1. Define a prefix (in these cases “h” and “f”) together with an identifier (a URI)
2. Use the tags from this namespace with the corresponding prefix

The first step is usually accomplished by introducing *the first tag* - which , of course, is also the outermost tag - using this namespace together with the namespace URI by the xmlns (XML NameSpace) attribute.

Important: The XML namespace definition states that a namespace needs a URI[5] – not necessarily a URL! As a consequence, you cannot expect a parser to validate the use of an imported namespace against any DTD or XML Schema – even if the URI provided is actually a URL pointing to a DTD or Schema file. For more information about what namespaces are *not*, read through the “XML Namespace Myths” at [http://www.rpbourret.com/xml/NamespaceMyths.htm].

2.2.1 Multiple Namespaces

Sometimes XML documents are composed from tags taken from more than one

5And even that's not the complete truth! The specification was later clarified to refer to IRIs rather than URIs [http://www.w3.org/TR/xml-names11/#iri-use]. IRIs are “Internationalized Resource Identifiers” - a generalization of URIs accounting for the international UNICODE character set [http://www.w3.org/International/2002/draft-duerst-iri-01.txt]. We're not dealing with IRIs here, URIs are sufficient for the purpose of this course.

namespace. In this case for each namespace the description of prefix and URI has to be given, preferrably with the first (root) element of the document. It doesn't matter that this element can only be taken from one of these namespaces, all can and should be defined there to make the namespace declarations appear in one place, like in this little beauty:

```
<rdf:RDF
        xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
        xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
        xmlns:dc="http://purl.org/dc/elements/1.1/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
        xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
        xmlns:owl="http://www.w3.org/2002/07/owl#"
        ... >
```

This defines six namespaces with the prefixes daml (DARPA Agent Markup Language), rdf (Resource Description Framework), dc (Dublin Core) , xsd (XML Schema Description), rdfs (RDF Schema) and owl (Web Ontology Language). These prefixes can now be used without worrying about name collisions.

2.3 DTDs and XML Schema

The short introduction already showed that a given XML in general an XML document will probably conform to a certain structure. The recipe document above could come in the same “form” for different recipes, XML documents describing parameters for a certain application are structurally identical, XML-based messages that are exchanged between applications follow certain rules about the vocabulary used and the connections between elements.

2.3.1 DTDs

The first way to describe the internal structure of a set of XML documents is to use the **Data Type Definition** (DTD) language. Here is an example DTD describing the recipe class of XML documents – it's stored as recipe.dtd:

```
<!ELEMENT recipe (title,ingredient+,instructions)>
<!ATTLIST recipe
  cook_time CDATA #REQUIRED
  name CDATA #REQUIRED
  prep_time CDATA #REQUIRED>

<!ELEMENT title (#PCDATA)>

<!ELEMENT ingredient (#PCDATA)>
```

```
<!ATTLIST ingredient
  amount CDATA #REQUIRED
  state CDATA #IMPLIED
  unit CDATA #REQUIRED>

<!ELEMENT instructions (step)+>
<!ELEMENT step (#PCDATA)>
```

Basically, the DTD describes the elements of a recipe XML file:

- What other element types can occur inside this element, in which sequence and how often?
- What are the attributes and legal attribute values of this element?

The <!ELEMENT tag describes an element of the respective XML document, by giving the name of the element and its allowed contents:

<!ELEMENT recipe (title,ingredient+,instructions)>

The recipe element consists of a title element, at least one ingredient element and one instructions element - in this order.

```
<!ATTLIST recipe
  cook_time CDATA #REQUIRED
  name CDATA #REQUIRED
  prep_time CDATA #REQUIRED>
```

The recipe element has three attributes, cook_time, name and prep_time, all being required and having a value of “Character Data” (CDATA), i. e. strings.

```
<!ELEMENT title (#PCDATA)>

<!ATTLIST ingredient
  amount CDATA #REQUIRED
  state CDATA #IMPLIED
  unit CDATA #REQUIRED>
```

The title element only contains “Parseble Character Data” (PCDATA), i. e. strings that have an inner structure that can be inspected and interpreted.

```
<!ELEMENT ingredient (#PCDATA)>
<!ATTLIST ingredient
  amount CDATA #REQUIRED
  state CDATA #IMPLIED
  unit CDATA #REQUIRED>
```

The ingredient element also only contains PCDATA, All attributes except state are required, state is optional.

```
<!ELEMENT instructions (step)+>
<!ELEMENT step (#PCDATA)>
```

Instructions have one or more steps, steps are PCDATA.

Here's a short list of value types and rules to build new types:

- CDATA and PCDATA have already been studied.
- ANY allows for any kind of value (rare!)
- EMPTY doesn't allow for any content
- element-name allows for an element with this name
- ($type_1$, $type_2$, ..., $type_n$) – with the specified types – defines a sequence
- ($type_1$ | $type_2$ | ... | $type_n$) defines a choice of one of the given types.
- type? - where type is a any content type – makes it optional (0 or 1 times)
- type* makes it optional and repeatable (0 or more times)
- type+ makes it required and repeatable (1 or more times)

DTDs are widely used to describe the internal structure of an XML document, like our recipe document. There has been some debate over issues of expressiveness of DTDs as well as the problem that the DTD syntax doesn't conform to the XML definitions.

Based on these considerations a stronger XML-based formalism for describing the structure of a set of XML documents has been developed, called **XML Schema**.

2.3.2 XML Schema

DTDs are confronted with some criticism generally aimed at a) lack of expressiveness and b) the use of a “idiosyncratic” syntax, rather than the utilization of XML syntax.

The standard of XML Schema addresses these issues by providing a “data definition language” for XML structures. In essence it describes XML elements by basic types and grouping operators. Here's an example:

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

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:tns="http://xml.netbeans.org/schema/recipe"
   elementFormDefault="qualified">

   <xs:element name="ingredient">
      <xs:complexType>
         <xs:simpleContent>
            <xs:extension base="xs:string">
               <xs:attribute name="amount" use="required" />
               <xs:attribute name="unit" use="required" />
               <xs:attribute name="state" use="optional" />
            </xs:extension>
         </xs:simpleContent>
      </xs:complexType>
   </xs:element>

   <xs:element name="instructions">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="step" type="xs:string" maxOccurs="unbounded" />
         </xs:sequence>
      </xs:complexType>
   </xs:element>

   <xs:element name="recipe">
      <xs:complexType>
         <xs:sequence>
            <xs:element name="title" type="xs:string" />
            <xs:element ref="ingredient" minOccurs="0" maxOccurs="unbounded" />
            <xs:element ref="instructions" />
         </xs:sequence>
         <xs:attribute name="name" type="xs:string" use="required"/>
         <xs:attribute name="prep_time" type="xs:string" use="required"/>
         <xs:attribute name="cook_time" type="xs:string" use="required"/>
      </xs:complexType>
```

```
    </xs:element>

</xs:schema>
```

It's easiest to study the “instructions” element definition first. It describes the element as a sequence of elementary elements, named “step”, of type “string” with an unspecified number (but at least one).

The “recipe” element is defined as a sequence of three subordinate elements (“title” – an elementary string, “ingredient” – defined at the beginning of the schema document and “instructions” – we know that already). The attributes are listed as a part of the element description and should be self-explanatory).

The “ingredient” element is a little tricky, because it has no sub-elements, only a couple of attributes. The idiom needed for that is obviously a little complicated.[6]

It should be noted that an XML Scheme document is primarily XML, so a lot of problems can be detected by validating the schema directly against the “schema schema”.

2.4 XPath

XPath is a language to navigate through the tree of an XML document, to retrieve subtrees and single nodes. The basic syntax uses paths from one node (preferably the root node) to a particular node in the tree.

XPath is the basis for using DOM parsers and other advanced XML technologies, so here's just a quick overview.[7]

Paths

As expected, the main navigational element in XPath is the “path”, i.e. a sequence of opening tags that lead from the root element to the desired element node(s). The result of evaluating an XPath expression is in most cases a so called **node set**. Paths are specified, not unlike a unix path, by enumerating the nodes on this path, separated by a “/”:

- /recipe
  locates the recipe element, i.e. the whole DOM tree
- /recipe/title
  locates the title element: <title>Basic bread</title>

A double slash “//” locates all elements with a given name:

6 … and contributes to the feeling that attributes are something that should better be avoided
7 If you want to play with XPath, download SketchPath from http://pgfearo.googlepages.com/ - a free tool to get a feeling for it.

- //ingredients
  locates all ingredient elements

If there is more then one match for a path – like in the above example – they can be accessed by index positions. The expression in square brackets describes additional properties for the selection of a node:

- //ingredient[1]
  locates the first ingredient element “Flour” - yes, numbering starts with 1!
- //ingredient[last()]
  locates the last ingredient element “Salt”
- //instructions/step[3]
  locates the third step of the instructions “Cover with a cloth...”

Attributes are described by using the ampersand “@” prefix and the square bracket notation:

- //ingredient[@state="warm"]
  locates all ingredients with the state attribute having the value “warm”
- //ingredient[@unit]
  locates all ingredients with a unit attribute
- //ingredient[@*]
  locates all ingredients with any attribute

Selections based on the element name are possible:

- //[starts-with(name(),”i”]
  locates all elements whose name starts with the letter “i”, i.e. all ingredient and instruction elements

Counting the number of nodes in a resulting node set:

- count(//ingredient[@state="warm"])
  yields the number of ingredients with a state “warm”, just one.
- count(//*[starts-with(name(),'i')])
  yields 5, of course

The vertical bar “|” serves as a union (“or”) operation:

- //step | //ingredient
  locates all step and all ingredient nodes
- count(//step | //ingredient)
  11, if you're not in the mood to count them by hand

2.4.1 Advanced XPath: Axes

So far a path is described by pointing down from one node to it's successors. More generally, XPath also allows for up and sideways movements. This is accomplished by a so-called **axis**.

An axis defines a node or node set starting from a given node. An axis is followed by a node selection – like already known – on this axis. Let's start with the asterisk '*' which stands for “all nodes”:

- //instructions/descendant::*
  gives all descendants of the instructions element
- //step/parent::*
  gives the parent of the step element, i.e. the instructions element
- //step/ancestor::*
  gives the ancestors (all nodes above it in the DOM tree)of the step element, i.e. the
  instructions and the recipe element

child is basically a complicated way of replacing the forward slash:

- /recipe/child::title
  is the same as /recipe/title

ancestor-or-self and descendant-or-self are like the ancestor and descendant but include the reference node. self is just the reference node itself.

preceding-sibling and following-sibling select the siblings before or after a given node:

- //ingredient[@unit="ounce"]/following-sibling::*
  Just a minute: The ingredient with unit “ounce” and then all siblings following it?
  Right! The “Water” and “Salt” ingredients and the instructions.

- //step[last()]/preceding-sibling::*
  Self-explanatory!

preceding and following are axes listing all elements (not only siblings) before or after the given element – combining it with a node selector we get:

- //step[3]/preceding::*[@amount="1"]
  Everything preceding the third step with an amount attribute of “1”. Must be “Salt”.

attribute selects a given attribute from a node

- //ingredient[3]/attribute::unit

This evaluates to “cups”

Ok, here's one to ponder at last:[8]

```
//ingredient[@unit="ounce"]/following-sibling::*[@state]/attribute::unit
```

8 ... and you thought, PHP was cryptic! It's “cups” by the way...

3 Accessing XML Documents from Java

Accessing XML documents from a program might mean to perform one or more of the following activities:

- *Parsing*: Investigating the structure of the document to find certain elements and process them.
- *Creation*: Setting up an XML document internally and publishing it for further processing.
- *Modification*: Taking an XML document and changing parts of it.
- *Transmission*: Sending an XML document over a network to another application
- *Transformation*: Performing an CSS/XSL transformation on a document.
- *Validation*: Validating an XML document against a DTD or Schema.

Java offers two ways to deal with XML documents:

1. JAXP: The “Java API for XML Processing” offers a standard API with different parser models (SAX and DOM) and features to perform XSL transformations.
2. JAXB: The “Java Architecture for XML Binding” integrates XML and the Java class/instance views in a declarative way with little explicit coding.

Actually, the use of JAXP and JAXB are not mutually exclusive. So, you can use JAXB annotations to map XML documents into Java instances and vice versa, but use JAXP to perform XSL transformations.

It's interesting to hint at the motivation behind JAXB – much of the arguments involved here are mirrored in the expositions of JAX-RS, JAX-WS and JPA later in this course. Just like JDBC (see chapter 3) is a precursor to JPA and suffers from a lack of integration into Java's OO paradigm, JAXP offered a “procedural” way to access XML documents – the processing of the XML information inside a Java object system requires still a lot of “boilerplate” code, leading to reduced efficiency of development, hidden incompatibilities and the potential for errors.

Beginning with the exposition of JAXB in 3.2, we will sketch some principles underlying JAXB, JAX-RS, JAX-WS and JPA: the use of “annotations” to decorate Java classes with additional semantics and the principle of “specification be exception”. So, that's something that should be studied in any case, even JAXB is not used.

3.1 JAXP (Java API for XML Processing)

Parsing first! Two different methods exist to parse an XML document, both available as a Java API:

1. Streaming or serial parsers: The document is read serially by the parser and events are activated upon encountering certain elements. These events are then

handled by the application to execute certain actions, not unlike a SWING application. **SAX** – the Simple API for XML – is probably the best-know streaming parser API for Java.

2. DOM parsers: The parser reads the complete document – generally by using a streaming parser – and constructs the DOM. The program can then navigate through this tree by a set of methods or simply by XPath expressions.

The difference can be characterized by looking at the behavior of a parser type on larger documents: A DOM parser does a lot of work building the tree before the application can even start examining the elements. A streaming parser will have problems navigating up and sideways in the DOM tree, but will run through the document exactly once, so it's probably a lot faster and less memory intensive.

We will study both parser models briefly by running through a small example – and of course it deals with recipes. The assumed task is to work out a Recipe class and construct an instance by reading it from an XML input – by either SAX or DOM. The base approach is relatively simple:

```
public class Recipe {

        // Normal instance fields
        private String title;
        private ArrayList<Ingredient> ingredients = new
                                        ArrayList<Ingredient>();
        private ArrayList<String> steps = new ArrayList<String>();

        // Just the usual interface - nothing special
        public String getTitle() { return title; }
        public List<Ingredient> getIngredients() { return ingredients; }
        public List<String> getSteps() { return steps; }

        // Factory methods
        public static Recipe fromSAX(String fileName) { ... }
        public static Recipe fromDOM(String fileName) { ... }
        ...
}
```

This goes together with a trivial Ingredient class:

```
public class Ingredient {
        private String name;
        private float amount;
        private String unit;
        private String state;
```

```
    public Ingredient(String name, float amount,
                                String unit, String state)  {
     this.name = name;
     this.amount = amount;
     this.unit = unit;
     this.state = state;
    }

    // That's actually a tricky decision:
    // Since the “state” attribute of an ingredient is optional,
    // there must be a default value.
    // Let the Ingredient class decide on this default
    // and not the parser. That's why this constructor comes into play!
    public Ingredient(String name, float amount, String unit) {
        this(name,amount,unit,"");
    }

    public String getName() { return name; }
    public float getAmount() { return amount; }
    public String getUnit() { return unit; }
    public String getState() { return state; }

    @Override
    public String toString() {
           return amount + " " + unit + " " + name + " " + state;
    }
}
```

3.1.1 SAX parsing

The main approach to SAX parsing is to handle the events the parser generates and to take appropriate actions when such an even occurs. The API is based on a set of Java interfaces and implementations of these interfaces in the org.xml.sax package. Here are the ones used in the example:[9]

The ContentHandler interface defines the handlers for the main event notifications:

- void characters(char[] ch, int start, int length)
  Receive notification of character data.

- void endElement(String uri, String localName, String qName)
  Receive notification of the end of an element.

- void startElement(String uri, String localName, String qName, Attributes atts)

9 Links to the APIs of all classes used in this section are found at the end.

Receive notification of the beginning of an element.

Other interfaces involved in SAX parsing – without actually being used in this example – are:

- DTDHandler defines event handlers for notifications from parsing the DTD – if present.
- EntityResolver is used to handle external entities that may be referenced from the current document.
- ErrorHandler defines event handlers parse errors or warnings.

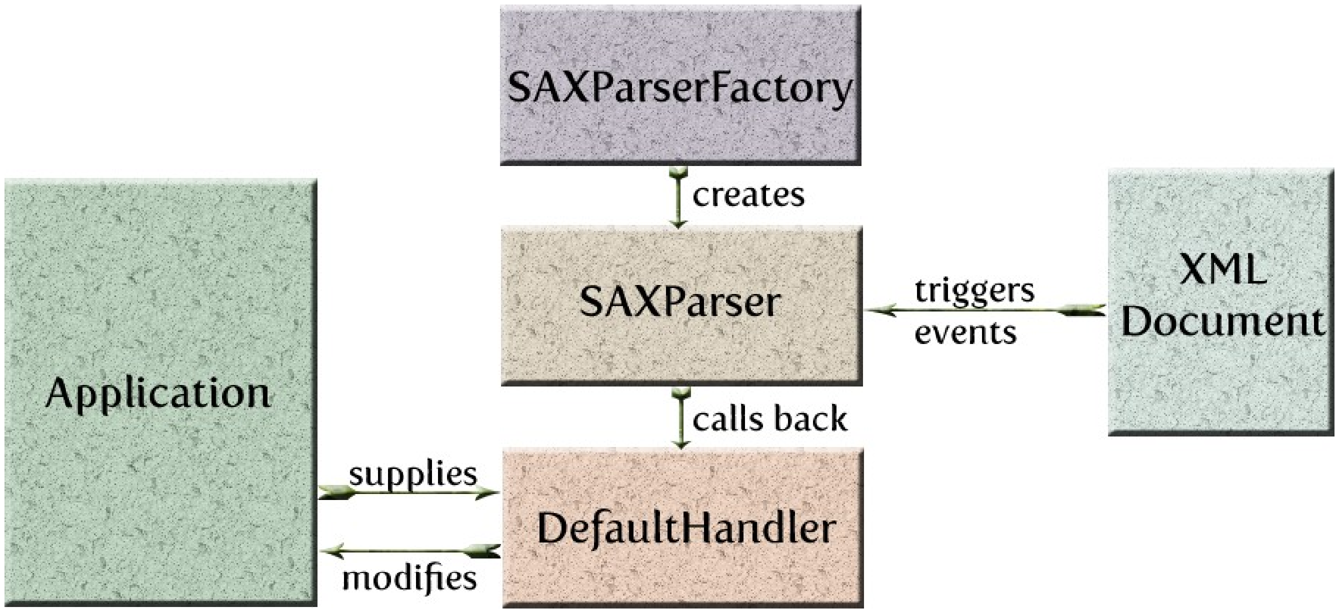
A SAX-based program would have to provide implementations for all those four interfaces, so the SAX API defines a “convenience” class DefaultHandler which implements these methods mostly by letting them do nothing. An application will then extend DefaultHandler and override these methods accordingly.

The SAXParser class starts the parser which will fire the events as it encounters is by calling back to the DefaultHandler instance. The implementation of the SAXParser instance is generally system-dependent, for example the parser used under Windows/Java 6 is the Apache Xerces parser.

Because of that the parser instantiation is handled by a factory SAXParserFactory which produces a parser which then will be used to parse the document. The following sketch shows roughly how these components interact.

These mechanisms are implemented in an instance method parseDocument(String fileName) making the fromSAX method more or less trivial:

```
public static Recipe fromSAX(String fileName) {
	Recipe r = new Recipe();
	r.parseDocument(fileName);
	return r;
}
```



Interactions in a SAX Parser

These mechanisms are implemented in an instance method parseDocument(String fileName) making the fromSAX method more or less trivial:

```
public static Recipe fromSAX(String fileName) {
        Recipe r = new Recipe();
        r.parseDocument(fileName);
        return r;
}
```

To deal with the call-backs we will have to provide an instance of DefaultHandler and as usual (?) there are at least two ways to do that:

1. Let Recipe itself extend DefaultHandler and provide the overriden callback methods.
2. Define an inner class, like RecipeHandler, that extends DefaultHandler and implements the callbacks there.

Without participating in the endless discussions about the “proper” way to do it, let's do it the proper way: by defining the handler as an inner class. Here's the stub, we'll fill it out later:

```
class RecipeHandler extends DefaultHandler
{
        @Override
        public void characters(char[] ch, int start, int length)
```

```
                    throws SAXException { ... }
        @Override
        public void startElement(String uri, String localName,
                    String qName, Attributes attributes)
                    throws SAXException { ... }
        @Override
        public void endElement(String uri, String localName, String qName)
          throws SAXException { ... }
}
```

The internal workings of the handler will be discussed in a minute. How to start the parser is now clear:

```
private void parseDocument(String fileName) {
        // Get a factory
        SAXParserFactory spf = SAXParserFactory.newInstance();
        try {
                // Get a new instance of the parser class
                SAXParser sp = spf.newSAXParser();
                // Parse the file and also register a new handler for callbacks
                sp.parse(fileName, new RecipeHandler());
        } catch (SAXException se) { se.printStackTrace();
        } catch (ParserConfigurationException pce) { pce.printStackTrace();
        } catch (IOException ie) { ie.printStackTrace();
        }
}
```

To complete the application, we just have to implement the callbacks in the RecipeHandler. Let's look at the three methods in detail:

public void characters(char[] ch, int start, int length)

This is called when the parser has finished reading a CDATA or PCDATA item, i.e. a string. It takes the characters read as a char[] with start index and length. In order to find out what this character sequence belongs to, the easiest way is to save it as a Java String object and process it in the endElement method. Assuming an instance variable elemString it's then implemented as

```
        public void characters(char[] ch, int start, int length)
                throws SAXException {
                elemString = new String(ch,start,length);
        }
```

public void startElement(String uri, String localName,
String qName, Attributes attributes)

This is called when the parser encounters an opening tag for an element. Since most of the element's content will appear afterwards, not much can be done here, except picking up possible attributes into another instance variable and saving them for later, i.e. the endElement method.

```
public void startElement(String uri, String localName, String qName,
        Attributes attributes) throws SAXException {
        savedAttributes = attributes;
}
```

public void endElement(String uri, String localName, String qName)

This is called when the parser encounters a closing tag. Now all what has been saved during the parse can be applied depending on the tag name. Not terribly difficult, just something that requires a little thought while coding.

```
public void endElement(String uri, String localName, String qName)
        throws SAXException {
        if (qName.equals("title"))
                title = elemString; // Save the string as title value

        if (qName.equals("step"))
                steps.add(elemString); // Add the string to the steps

        if (qName.equals("ingredient")) {
                // Grab the attributes saved from the opening tag
                float amount =
                        Float.parseFloat(savedAttributes.getValue("amount"));
                String unit  = savedAttributes.getValue("unit");
                String state = savedAttributes.getValue("state");

                // Call the correct constructor (state is optional!)
                // and add the Ingredient to the list
                ingredients.add((state == null
                        ? new Ingredient(elemString,amount,unit)
                        : new Ingredient(elemString,amount,unit,state)));
        }

}
}
```

And that's it! Provide a toString method in the Recipe class:

```
public String toString() {
        String result = "Recipe for " + title + ":\n";
        for (Ingredient i : ingredients) result += i + "\n";
        for (String s : steps) result += s + "\n";
        return result;
}
```

Then call System.out.println(Recipe.fromSAX("recipe.xml")) and you'll get

```
Recipe for Basic bread:
3.0 cups Flour
0.25 ounce Yeast
1.5 cups Water warm
1.0 teaspoon Salt
Mix all ingredients together.
Knead thoroughly.
Cover with a cloth, and leave for one hour in warm room.
Knead again.
Place in a bread baking tin.
Cover with a cloth, and leave for one hour in warm room.
Bake in the oven at 350(degrees)F for 30 minutes.
```

3.1.2 DOM Parsing

Once you got the philosophy behind parsers, factories. DOM and XPath, DOM parsers are not really difficult. The main strategy is the same as with SAX parsing, except that we might have to deal with a lot of exceptions:

```
public static Recipe fromDOM(String fileName) {
        Recipe r = new Recipe();
        try {
                r.buildDOM(fileName);
        } catch (Exception e) { e.printStackTrace(); return null; }
    return r;
}
```

As always, setting up an XML parser requires some well-thought-out footwork, invloving factories and the interplay between DOM and the SAX parser. We need the following objects:

1. A DocumentBuilderFactory to create a DocumentBuilder which will (behind the scenes) use a SAX parser to build the DOM tree.
2. This will construct a Document object, representing the DOM tree. Since the implementation varies between platforms, Document is an interface.
3. Before looking into the tree, an object representing the XPath processor is needed. It will implement the XPAth interface and be produced by a factory (XPathFactory), because it's also implementation-dependent.

```
private void buildDOM(String fileName)
```

```
throws  SAXException, IOException,
        ParserConfigurationException, XPathExpressionException {
// Step 1: Get a working DocumentBuilder
DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();

// Turn off validation and make it ignore white space
factory.setValidating(false);
factory.setIgnoringElementContentWhitespace(true);
DocumentBuilder builder = factory.newDocumentBuilder();

// Step 2: Open input file and create the DOM from the DocumentBuilder
File f = new File(fileName);
Document doc = builder.parse(f);

// Step 3: Get the XPath processor
XPathFactory xpfactory = XPathFactory.newInstance();
XPath path = xpfactory.newXPath();
```

Actually constructing the Recipe instance takes just some XPath (in green) and some string manipulations:

```
// Grab the title and place it into the instance field
title = path.evaluate("//title", doc);

// Count the ingredient elements and iterate through them
int numIngredients =
                Integer.parseInt(path.evaluate("count(//ingredient)",doc));

// But watch it! They start counting at 1!
for (int i = 1; i <= numIngredients; i++) {

        // Get //ingredient[i] and its attibutes
        String name = path.evaluate("//ingredient[" + i + "]",doc);
        float amount = Float.parseFloat(
                path.evaluate("//ingredient[" + i + "]/@amount",doc));
        String unit =
                path.evaluate("//ingredient[" + i + "]/@unit",doc);
        String state =
                path.evaluate("//ingredient[" + i + "]/@state",doc);

        // As above: Call the right constructor
        ingredients.add((state.equals("")
                ? new Ingredient(name,amount,unit)
                        : new Ingredient(name,amount,unit,state)));
        }

        // The same for the steps
        int numSteps =
                Integer.parseInt(path.evaluate("count(//step)",doc));
        for (int i = 1; i <= numSteps; i++) {
                steps.add(path.evaluate("//step[" + i + "]",doc));
}
```

3.1.3 Transforming XML Documents

Transforming XML documents is one of the most important operations that are supported by XML; it describes ways to influence how XML documents are presented – mostly by "Cascading Style Sheets" (CSS) – and how XML can be completely transformed into another surface structure, most notably HTML via "XML Style Sheet Transformations" (XSLT).

3.1.3.1 CSS

Cascading Style Sheets (CSS) are one of the three "pillars" of Web Page Design (with HTML and JavaScript). Since CSS is also part of the XML family, it can be used to render XML documents in a relatively simple way – all that's needed is a browser.

CSS is based on targeting one or more elements in an XML document and giving "style rules" that describe properties, such as font, color or position. A style rule is of the form

property: value

where the property refers to a styling feature and the value defines how this feature will look like on the output medium.[10]

In general an element is targeted by giving its name and then attaching one or more style rules to it.

element-name { rule_1; rule_2; …; rule_n }

Best, look at a slightly involved example:

```
recipe { background:white; color:black;
         font-family: Georgia, Times, serif;
         font-size:14pt; margin-left: 100px }
```

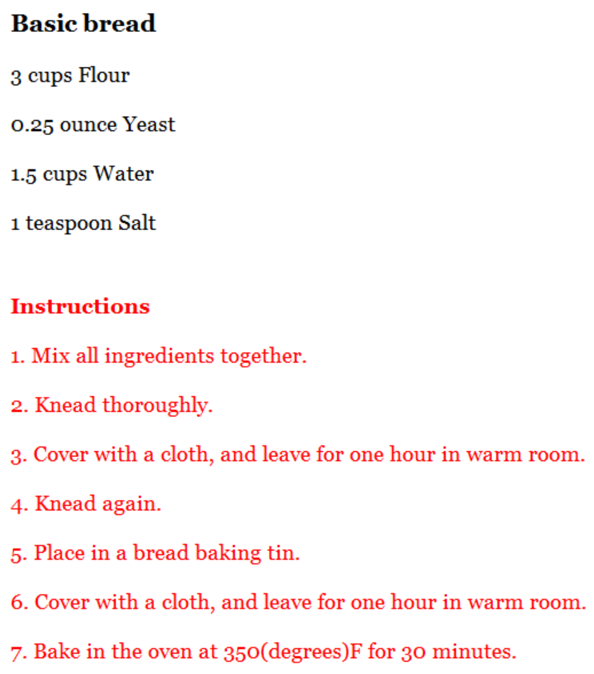
10 One of the interesting features of CSS is that different style sheets can be defined for different media, such as screens or printers. Nothing we will deal with here, though!

```
title { font-size: 16pt; font-weight:bold }

ingredient, step { display: block; padding-top:20px }
ingredient:before { content: attr(amount) " " attr(unit) " " }

instructions { color: red; position:relative; top:50px; font-weight: bold }
instructions:before { content: "Instructions" }

step { font-weight: normal; counter-increment: steps }
step:before { content: counter(steps) ". " }
```

In order to apply a style sheet to an XML document, it's enough to place a directive into the document's prolog:

```
<?xml-stylesheet type="text/css" href="recipe.css"?>
```

Applying this style sheet to the recipe.xml file results in the following output in a browser:



Basic bread

3 cups Flour

0.25 ounce Yeast

1.5 cups Water

1 teaspoon Salt

Instructions

1. Mix all ingredients together.

2. Knead thoroughly.

3. Cover with a cloth, and leave for one hour in warm room.

4. Knead again.

5. Place in a bread baking tin.

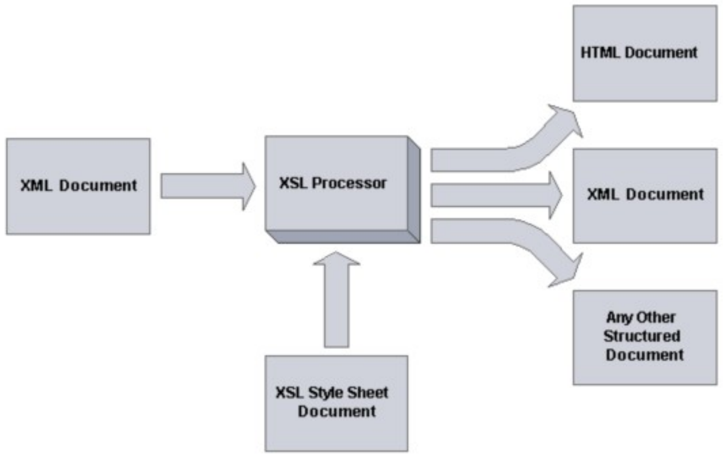
6. Cover with a cloth, and leave for one hour in warm room.

7. Bake in the oven at 350(degrees)F for 30 minutes.

3.1.3.2 XSLT

Next to DOM and SAX parsers that make the content of an XML document accessible to a program, maybe an equally important capability of XML is the capability of XML to undergo syntactical transformations, in most cases a transformation between XML and (say!) HTML for presentation purposes. Yet, HTML as an output format might be the most popular although variations (SQL, Prolog, ...) are also possible.

XSLT is an XML vocabulary that was designed to define the translation of an XML document into any other format, not unlike *javadoc* generates a new documents out of an existing Java source file. The main mechanism is shown below:

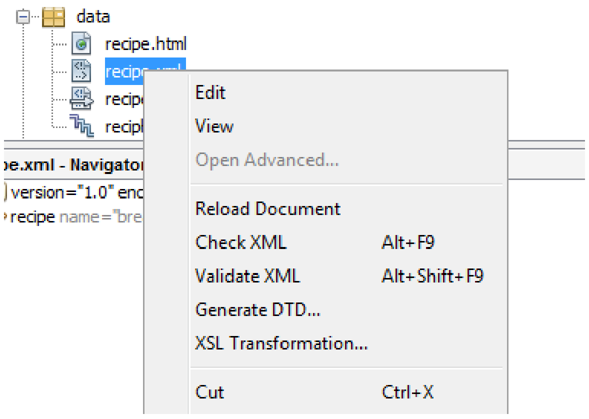


The scenario now involves a specific piece of software – the XSL Processor – that takes an XML document and a style sheet document as input and produces another output document from it.

XSL processors are available in virtually all platforms - Java, PHP and most development environments, such as NetBeans.

Style sheet transformations can be produced manually or by a program. NetBeans allows to apply XSLT to any XML document.

The XSL syntax is pretty idiosyncratic and really understanding what's going on can be a little tricky, but the next example should give a starting point.



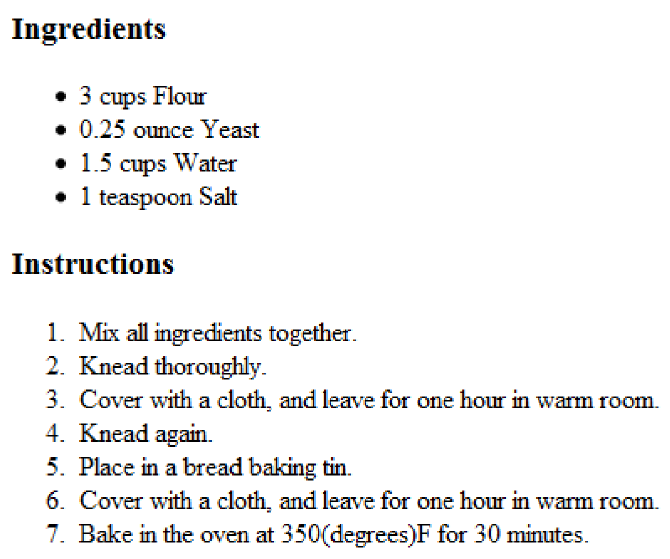
Four things are important: The application of styles is controlled by so-called “matches” that use XPath expressions to target specific elements; element contents is referenced by specific language “functions”, the Schema language provides limited control structures for conditions and repetitions and all of this is connected to the output format by embedding literal target language expressions. In the following example the target language is HTML, so the embedded HTML tags are marked red:

```
<?xml version="1.0"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
   <xsl:output method="html"/>

   <xsl:template match="/">
      <html>
         <head>
            <title>My recipe</title>
         </head>
         <body>
            <xsl:apply-templates />
         </body>
      </html>
   </xsl:template>

   <xsl:template match="recipe">
      <h3>Ingredients</h3>
      <ul>
         <xsl:for-each select="ingredient">
            <li>
               <xsl:value-of select="@amount" />
               <xsl:text> </xsl:text>
               <xsl:value-of select="@unit" />
               <xsl:text> </xsl:text>
               <xsl:value-of select="." />
            </li>
         </xsl:for-each>
      </ul>
      <h3>Instructions</h3>
      <ol>
         <xsl:for-each select="instructions/step">
            <li>
               <xsl:value-of select="." />
            </li>
         </xsl:for-each>
      </ol>
   </xsl:template>
</xsl:stylesheet>
```

Applying this transformation will result in a target document (recipe.html) that can be viewed in a browser:

```
<html>
<head>
<META http-equiv="Content-Type" content="text/html; charset=UTF-8">
<title>My recipe</title>
</head>
<body>
<h3>Ingredients</h3>
<ul>
<li>3 cups Flour</li>
<li>0.25 ounce Yeast</li>
<li>1.5 cups Water</li>
<li>1 teaspoon Salt</li>
</ul>
<h3>Instructions</h3>
<ol>
<li>Mix all ingredients together.</li>
<li>Knead thoroughly.</li>
<li>Cover with a cloth, and leave for one hour in warm room.</li>
<li>Knead again.</li>
<li>Place in a bread baking tin.</li>
<li>Cover with a cloth, and leave for one hour in warm room.</li>
<li>Bake in the oven at 350(degrees)F for 30 minutes.</li>
</ol>
</body>
</html>
```

This will display in the following way:[11]



Ingredients

- 3 cups Flour
- 0.25 ounce Yeast
- 1.5 cups Water
- 1 teaspoon Salt

Instructions

1. Mix all ingredients together.
2. Knead thoroughly.
3. Cover with a cloth, and leave for one hour in warm room.
4. Knead again.
5. Place in a bread baking tin.
6. Cover with a cloth, and leave for one hour in warm room.
7. Bake in the oven at 350(degrees)F for 30 minutes.

11 Of course, you can reference a CSS from inside the target document – makes it look much nicer!

Well, applying such an XSL transformation manually is not the most valuable operation in itself – it's much more important to be able to do these transformations via software, in most cases to create a document version that can be presented to a user.

Java offers an elaborate API to produce these transformations; actually, you don't have to do a lot do achieve that – it's all built in:

```
import javax.xml.transform.*;
import javax.xml.transform.stream.*;
import java.io.*;

public class XML1 {

   public static void main(String[] args)
         throws TransformerException,
               TransformerConfigurationException,
               FileNotFoundException, IOException {

      Transformer transformer =
            TransformerFactory.newInstance().
                  newTransformer(new StreamSource("src/data/stocks.xsl"));

      transformer.transform(new StreamSource("src/data/stocks.xml"),
                                    new StreamResult(System.out));
   }
}
```

The Transformer class is an abstract class that defines operations to transform document trees – the specific instance is created by the factory.[12]

3.1.3 Creating XML documents

The last section about JAXP describes how to create an XML document from any data source in Java. There are actually a couple of ways to do that – the one that fits best into this exposition uses

- the DocumentBuilderFactory and the DocumentBuilder
- the Document interface, both to create the DOM tree for the resulting XML document
- the TransformerFactory and Transformer classes to convert the tree into a textual format, more or less like it was done in the XSLT example.

That's a bit of a lengthy program, but it shows a lot of features involved in this process:

12 In this example the actual instance is built from the class com.sun.org.apache.xalan.internal.xsltc.trax.TransformerImpl

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import javax.xml.transform.*;
import javax.xml.transform.dom.*;
import javax.xml.transform.stream.*;

public class XML3 {

    public static void main(String[] args) {
        // Set up all the data – we might pull these out of a database later!
        String[] ingredients = {"Flour", "Yeast", "Water", "Salt"};
        String[] amounts = {"3", "0.25", "1.5", "1"};
        String[] units = {"cups", "ounce", "cups", "teaspoon"};
        String[] states = {null, null, "warm", null};

        String[] steps = {
            "Mix all ingredients together.",
            "Knead thoroughly.",
            "Cover with a cloth, and leave for one hour in warm room.",
            "Knead again.",
            "Place in a bread baking tin.",
            "Cover with a cloth, and leave for one hour in warm room.",
            "Bake in the oven at 350(degrees)F for 30 minutes."
        };

        try {
            // We need a Document – Yes, you can write this in one statement, but why?
            DocumentBuilderFactory dbfac = DocumentBuilderFactory.newInstance();
            DocumentBuilder docBuilder = dbfac.newDocumentBuilder();
            Document doc = docBuilder.newDocument();

            // Create the root element and add it to the document
            Element root = doc.createElement("recipe");
            doc.appendChild(root);

            // Put in the title
            Element child = doc.createElement("title");
            child.appendChild(doc.createTextNode("Basic bread"));
            root.appendChild(child);

            // Dysmal coding - put in the ingredients
            // use a "normal" for loop - we access four arrays in parallel
            for (int i = 0; i < ingredients.length; i++) {
                child = doc.createElement("ingredient");
                child.setAttribute("amount", amounts[i]);
            child.setAttribute("unit", units[i]);
            if (states[i] != null) child.setAttribute("state", states[i]);
            child.appendChild(doc.createTextNode(ingredients[i]));
            root.appendChild(child);

         }

          // Put the steps in – this time an “enhanced” loop does the trick!
         for (String step : steps) {
            child = doc.createElement("step");
            child.appendChild(doc.createTextNode(step));
            root.appendChild(child);
         }

         Transformer transformer = TransformerFactory.newInstance().newTransformer();
         // You might want to set that to "no" later...
         transformer.setOutputProperty(OutputKeys.INDENT, "yes");
         transformer.transform(new DOMSource(doc), new StreamResult(System.out));
      } catch (Exception e) {
         System.out.println(e); // Or do something smarter...
      }
   }
}
```

Run this and you get:[13]

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<recipe>
<title>Basic bread</title>
<ingredient amount="3" unit="cups">Flour</ingredient>
<ingredient amount="0.25" unit="ounce">Yeast</ingredient>
<ingredient amount="1.5" state="warm" unit="cups">Water</ingredient>
<ingredient amount="1" unit="teaspoon">Salt</ingredient>
<step>Mix all ingredients together.</step>
<step>Knead thoroughly.</step>
<step>Cover with a cloth, and leave for one hour in warm room.</step>
<step>Knead again.</step>
<step>Place in a bread baking tin.</step>
<step>Cover with a cloth, and leave for one hour in warm room.</step>
<step>Bake in the oven at 350(degrees)F for 30 minutes.</step>
</recipe>
```

13 But don't ask me why the state attribute appears before the unit attribute!

3.2 JAXB

After understanding the basics of XML, DOM, XSLT and their treatment with the JAXP API, we're ready to sit back for a moment and consider pros and cons of this approach:

Pro: 1. JAXP is easy to understand – you just need some examples and you can get started.
2. JAXP gives the programmer a lot of control over the process – choosing between DOM and SAX parsing is probably the most important distinction the programmer can make.

Con: 1. JAXP requires a lot of code to create Java objects that conform to a given XML document.
2. Mapping of embedded XML structures (like the "ingredients" inside a "recipe") can be done in multiple ways leading to complicated code dealing with the mapping of these substructures into instance variables of a container type[14].

This is where JAXB comes in. JAXB is built around the objective to provide a direct mapping of XML documents to Java instances – not unlike the DOM example given in 3.1. The difference is that these mappings are done automatically without the need for extensive "boilerplate code", i. e. extensive but standard code that doesn't add to the functionality of the application.

3.2.1 JAXB Architecture (and similar ones)

Understanding the basic ideas behind JAXB is also helpful to understand similar approaches, such as JAX-RS, JAX-WS, JPA, etc. All these architectures aim at removing coding requirements needed for faster and more reliable integration of services into a "normal" Java program. Of course, the well-known concept of an API (as an access mechanism to a library) already does that in many cases; JAXB and other architectures add another feature to this: *configuration*.

Configuration here describes the idea of taking a application object (sometimes referred to as a "Plain Old Java Object" or POJO) and declaratively (!) describing its mapping into the desired service. In JAXB that means, describing how the data inside a Java instance is mapped into the elements of an XML document.

The classical "procedural" API functionality is then reduced to invoke methods that actually perform this transformation – these methods access the configuration and and generate the target structure automatically.

The tool to do that are Java "annotations".

14 Just imagine the DOM parser example a couple of pages before applied to a more complex XML document!

Annotations decorate the source code of a Java class with directives that can be used by other classes to add additional functionality to this class or instances of it. You probably know some elementary annotations already:

@author, @param, @return, …	annotations used by the javadoc utility
@Override, @Deprecated, …	annotations used for fine-grained control over the compile process

JAXB – as well as the other interesting Java architectures – introduce new sets of annotations that are used by the API functions to control their operations[15].

Let's look: the following introductory example demonstrates the basic three steps to map a Java instance into an XML document.

A. Annotate the Java class. Here we provided only the minimum annotation @XMLRootElement, telling JAXB that this class is to be mapped into the root element of the XML document. Everything else that might be of interest for the transformation process is set at default values – we'll deal with more control over the process later.

```
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement
public class Person {

   private String firstName;
   private String lastName;
   private int age;

   public int getAge() {...}
   public void setAge(int age) {...}
   // the other setters/getters
}
```

B. Instantiate the class and set the respective instance variable values – that's standard Java:

```
Person p = new Person();
p.setFirstName("Gerald");
p.setLastName("Bostock");
p.setAge(55);
```

C. Call the API function to "marshal" the instance into the required XML format. We'll look at the API in a moment, here's the result:

15 For more information about annotations see http://docs.oracle.com/javase/tutorial/java/annotations/ and http://docs.oracle.com/javase/1.5.0/docs/guide/language/annotations.html

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<person>
   <firstName>Gerald</firstName>
   <lastName>Bostock</lastName>
   <age>55</age>
</person>
```

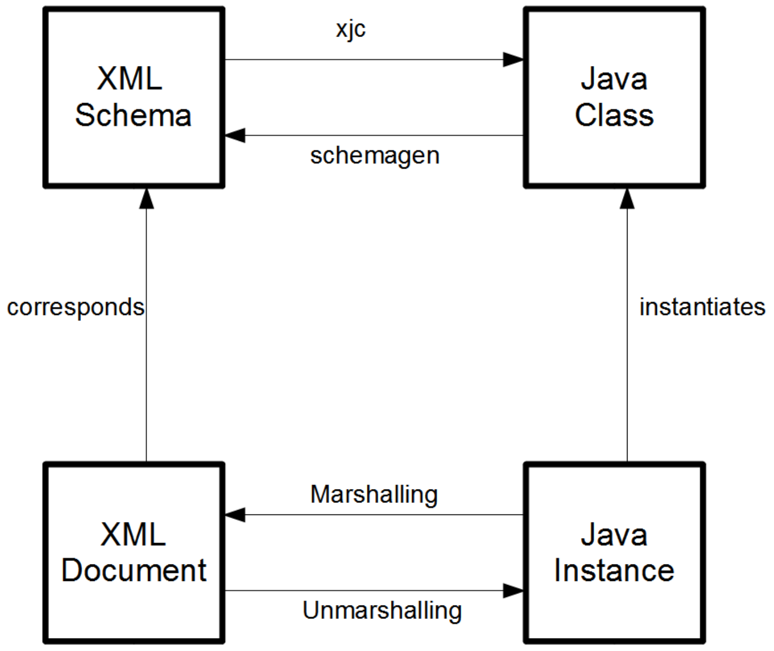
Of course, this can also be reversed – going from an XML document into the instance form – by the mirror process of "unmarshalling".

A couple of remarks are in order. First of all, JAXB et al. follow the principle of "Configuration by Exception", meaning that they come with a complete set of default configuration options that can be overridden, if necessary. If these defaults are acceptable, the annotation effort is minimal. We used this feature in the example.

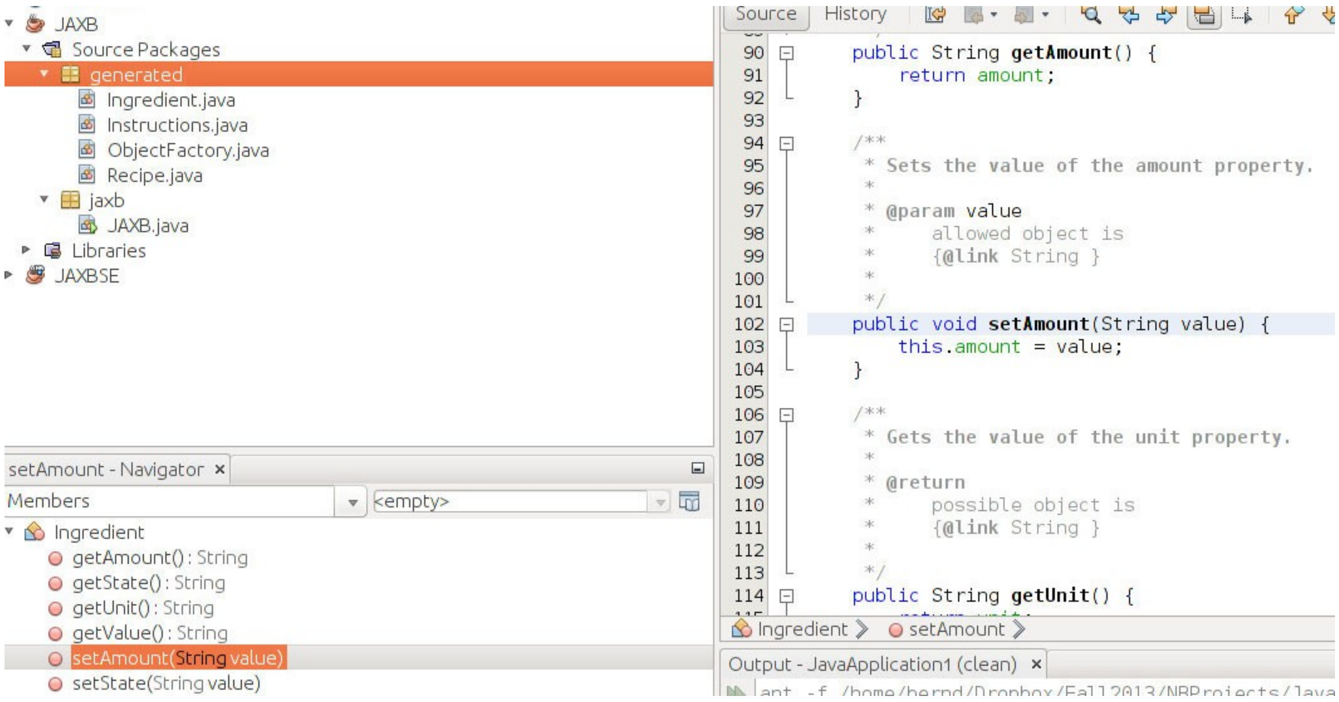
Secondly, in many cases the classes involved have to follow certain coding guidelines, most of them easy and standard – in the given example, the most important coding requirement is that the the class has a default constructor (parameterless constructor).

JAXB supports four different transformations:

1. XML Schema to Java class(es) over xjc
2. Java class(es) to XML Schema over schemagen
3. XML document to instance(s) over unmarshalling
4. instance(s) to XML document over marshalling



2.12.1 From XML Schema to Java classes (xjc)



```
    public String getAmount() {
        return amount;
    }

    /**
     * Sets the value of the amount property.
     *
     * @param value
     *     allowed object is
     *     {@link String }
     *
     */
    public void setAmount(String value) {
        this.amount = value;
    }

    /**
     * Gets the value of the unit property.
     *
     * @return
     *     possible object is
     *     {@link String }
     *
     */
    public String getUnit() {
```

```
ant -f /home/bernd/Dropbox/Fall2013/NBProjects/Java
```

List of Java Classes and Interfaces used in the examples - XML

1. SAX

Interface ContentHandler
http://java.sun.com/javase/6/docs/api/org/xml/sax/ContentHandler.html

Interface DTDHandler
http://java.sun.com/javase/6/docs/api/org/xml/sax/DTDHandler.html

Interface EntityResolver
http://java.sun.com/javase/6/docs/api/org/xml/sax/EntityResolver.html

Interface ErrorHandler
http://java.sun.com/javase/6/docs/api/org/xml/sax/ErrorHandler.html

Class DefaultHandler
http://java.sun.com/javase/6/docs/api/org/xml/sax/helpers/DefaultHandler.html

Class SAXParser
http://java.sun.com/javase/6/docs/api/javax/xml/parsers/SAXParser.html

Class SAXParserFactory
http://java.sun.com/javase/6/docs/api/javax/xml/parsers/SAXParserFactory.html

Interface Attributes
http://java.sun.com/j2se/1.5.0/docs/api/org/xml/sax/Attributes.html

Class SAXException
http://java.sun.com/j2se/1.4.2/docs/api/org/xml/sax/SAXException.html

Class ParserConfigurationException

http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/ParserConfigurationException.html

2. DOM

Interface Document
http://java.sun.com/j2se/1.4.2/docs/api/org/w3c/dom/Document.html

Class DocumentBuilder
http://java.sun.com/javase/6/docs/api/javax/xml/parsers/DocumentBuilder.html

Class DocumentBuilderFactory

http://java.sun.com/j2se/1.4.2/docs/api/javax/xml/parsers/DocumentBuilderFactory.html

Interface XPath
http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/xpath/XPath.html

Class XPathFactory
http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/xpath/XPathFactory.html

XPathExpressionException

http://java.sun.com/j2se/1.5.0/docs/api/javax/xml/xpath/XPathExpressionException.html