J2EE

XML (Extensible Markup Language)

Java API for XML

Presented by Bartosz Sakowicz

Overview of XML

XML is a text-based markup language that is fast becoming the standard for data interchange on the Web. As with HTML, you identify data using <u>tags</u> (identifiers enclosed in angle brackets, like this: <...>). Collectively, the tags are known as "markup".

But unlike HTML, XML tags *identify* the data, rather than specifying how to display it. Where an HTML tag says something like "display this data in bold font" (<b>...</b>), an XML tag acts like a field name in your program. It puts a label on a piece of data that identifies it (for example: <message>...</message>).

Overview of XML(2)

In the same way that you define the field names for a data structure, you are free to use **any XML tags that make sense** for a given application.

Naturally, though, for multiple applications to use the same XML data, they have to agree on the tag names they intend to use.

Overview of XML(3)

XML does not do anything. XML is created to structure, store, and to send information.

XML tags are not predefined. You must "invent" your own tags.

XML example:

```
<message>
          <to>you@yourAddress.com</to>
          <from>me@myAddress.com</from>
          <subject>XML Is Really Cool</subject>
          <text> How many ways is XML cool? Let me count the
ways... </text>
</message>
```

Tags and attributes

Tags can also contain attributes. Example shows an email message structure that uses attributes for the "to", "from", and "subject" fields:

<message **to="**you@yourAddress.com**"**
from="me@myAddress.com**" subject="**XML Is Really Cool**"**>
        <text> How many ways is XML cool? Let me count the ways... </text>
</message>

As in HTML, the attribute name is followed by an equal sign and the attribute value, and multiple attributes are separated by spaces.

Forming XML document

- Every tag has closing tag: <to> ...</to> OR <to/>(empty tag)
- **XML is case-sensitive.**

  XML elements must follow these naming rules:
  - Names can contain letters, numbers, and other characters
  - Names must not start with a number or punctuation character
  - Names must not start with the letters xml (or XML or Xml ..)
  - Names cannot contain spaces

Forming XML document(2)

- All tags are completely nested. So you can have <message>..<to>..</to>..</message>, but never <message>..<to>..</message>..</to>.

- Every XML document starts with prolog (<?xml ...)

- All XML documents must contain a single tag pair to define the root element. All other elements must be nested within the root element:

  <root> <child> <subchild>.....</subchild> </child> </root>

- XML comment is identical to HTML comment:

  <!-- This is a comment -->

The XML prolog

The minimal prolog contains a declaration that identifies the document as an XML document:

<?xml version="1.0"?>

The XML declaration may contain the following attributes:

version

Identifies the version of the XML markup language used in the data. This attribute is not optional.

encoding

Identifies the character set used to encode the data. "ISO-8859-1" is "Latin-1" the Western European and English language character set. (The default is compressed Unicode: UTF-8.)

standalone

Tells whether or not this document references an external entity or an external data type specification. If there are no external references, then "yes" is appropriate

Processing instructions

An XML file can also contain *processing instructions* that give commands or information to an application that is processing the XML data. Processing instructions have the following format:

<?target instructions?>

where the *target* is the name of the application that is expected to do the processing, and *instructions* is a string of characters that embodies the information or commands for the application to process.

DTD

• The DTD specification is actually part of the XML specification, rather than a separate entity. On the other hand, it is optional -- you can write an XML document without it.

• A DTD specifies the kinds of tags that can be included in your XML document, and the valid arrangements of those tags.

• Unfortunately, it is difficult to specify a DTD for a complex document in such a way that it prevents all invalid combinations and allows all the valid ones.

• The DTD can exist at the front of the document, as part of the prolog. It can also exist as a separate entity.

Schema standards

A DTD makes it possible to validate the structure of relatively simple XML documents.
A DTD can't restrict the content of elements, and it can't specify complex relationships. For example, it is impossible to specify with a DTD that a <heading> for a <book> must have both a <title> and an <author>, while a <heading> for a <chapter> only needs a <title>. In a DTD, once you only get to specify the structure of the <heading> element one time. There is no context-sensitivity.

This issue stems from the fact that a DTD specification is not hierarchical. For a mailing address that contained several "**parsed character data" (PCDATA)** elements, for example, the DTD could be as introduced on following transparency.

Schema standards(2)

*<!ELEMENT mailAddress (**name**, address, zipcode)>*
*<!ELEMENT **name** (#PCDATA)>*
<!ELEMENT address (#PCDATA)>
<!ELEMENT zipcode (#PCDATA)>

The specifications are linear. That fact forces you to come up with new names for similar elements in different settings. So if you wanted to add another "name" element to the DTD that contained the <firstName>, <middleInitial>, and <lastName>, then you would have to come up with another identifier. You could not simply call it "name" without conflicting with the <name> element defined for use in a <mailAddress>.

XML schema

A large, complex standard that has two parts:

One part specifies structure relationships. (This is the largest and most complex part.)

The other part specifies mechanisms for validating the content of XML elements by specifying a (potentially very sophisticated) datatype for each element.

Attributes and elements

It is possible to model the title of a slide either as:

<slide> <title>This is the title</title> </slide>

or as:

<slide title="This is the title">...</slide>

In some cases, the different characteristics of attributes and elements make it easy to choose.

Attributes and elements(2)

Sometimes, the choice between an attribute and an element is forced on you by the nature of attributes and elements:

The data contains substructures

In this case, the data item must be modeled as an *element*. It can't be modeled as an attribute, because attributes take only simple strings. So if the title can contain emphasized text like this:

The <em>Best</em> Choice

, then the title must be an element.

The data contains multiple lines

Here, it also makes sense to use an *element*. Attributes need to be simple, short strings or else they become unreadable, if not unusable.

Attributes and elements(3)

The data changes frequently

When the data will be frequently modified, especially by the end user, then it makes sense to model it as an *element*. XML-aware editors tend to make it very easy to find and modify element data. Attributes can be somewhat harder to get to, and therefore somewhat more difficult to modify.

The data is a small, simple string that rarely if ever changes

This is data that can be modeled as an *attribute*.

The data is confined to a small number of fixed choices

Here is one time when it really makes sense to use an *attribute*. Using the DTD, the attribute can be prevented from taking on any value that is not in the preapproved list. An XML-aware editor can even provide those choices in a drop-down list.

Attributes and elements(4)

```
<?xml version='1.0' encoding='utf-8'?>
<!-- A SAMPLE set of slides -->
<slideshow   title="Sample Slide Show"
             date="Date of publication"
             author="Yours Truly" >
<!-- TITLE SLIDE -->
<slide type="all"> <title>Wake up to Wonder!</title></slide>
<!-- OVERVIEW -->
<slide type="all">
     <title>Overview</title>
     <item>Why <em>Wonder</em> are great</item>
     <item/>
     <item>Who <em>buys</em> Wonder</item> </slide>
</slideshow>
```

Handling special characters

In XML, an entity is an XML structure (or plain text) that has a name. Referencing the entity by name causes it to be inserted into the document in place of the entity reference. To create an entity reference, the entity name is surrounded by an ampersand and a semicolon:

&entityName;

Predefined entities for special characters:

Character	***Reference***
&	&amp;
<	&lt;
>	&gt;
"	&quot;
'	&apos;

Character references

•A **character reference** like &#147; contains a hash mark (#) followed by a number.

• The number is the Unicode value for a single character, such as 65 for the letter “A”.

• In this case, the "name" of the entity is the hash mark followed by the digits that identify the character.

Handling text with XML-style syntax

When you are handling large blocks of XML or HTML that include many of the special characters, it would be inconvenient to replace each of them with the appropriate entity reference. For those situations, you can use a CDATA section.

A CDATA section works like <pre>...</pre> in HTML, only more so -- **all whitespace in a CDATA section is significant**, and characters in it are not interpreted as XML. A CDATA section starts with <![CDATA[and ends with]]>.

CDATA example

```
<item><![CDATA[Diagram:


        frobmorten <------------ fuznaten |


         <3> ^ | <1> | <1> = fozzle V |
        <2> = framboze Staten+
        <3> = frenzle <2>
 ]]></item>
```

Creating DTD

<?xml version='1.0' encoding='utf-8'?>
<!-- DTD for a simple "slide show". -->
<!ELEMENT slideshow (slide+)>

The DTD tag starts with **<!** followed by the tag name (**ELEMENT**). After the tag name comes the name of the element that is being defined (slideshow) and, in parentheses, one or more items that indicate the valid contents for that element. In this case, the notation says that a slideshow consists of one or more slide elements.

Here are the qualifiers you can add to an element definition:

Qualifier	***Name***	***Meaning***
?	Question Mark	Optional (zero or one)
*****	Asterisk	Zero or more
+	Plus Sign	One or more

Creating DTD(2)

• You can include multiple elements inside the parentheses in a comma separated list, and use a qualifier on each element to indicate how many instances of that element may occur. The comma-separated list tells which elements are valid and the order they can occur in.

• You can also nest parentheses to group multiple items. For an example, after defining an image element, you could declare that every image element must be paired with a title element in a slide by specifying:

((image, title)+).

The plus sign applies to the image/title pair to indicate that one or more pairs of the specified items can occur.

Creating DTD(3)

Defining text and nested elements:

<!ELEMENT slide (title, item*)>   <!-- (1) -->
<!ELEMENT title (#PCDATA)>   <!-- (2) -->
<!ELEMENT item (#PCDATA | item)* >   <!-- (3) -->

(1) - A slide consists of a title followed by zero or more item elements.
(2) - A title consists entirely of **parsed character data** (PCDATA). It is just text(Name distinguishes it from CDATA sections, which contain character data that is not parsed.) The "#" that precedes PCDATA indicates that what follows is a special word, rather than an element name.
(3) - The vertical bar (|) indicates an or condition

Creating DTD(4)

Special elements values in DTD:

• Rather than specifying a parenthesized list of elements, the element definition could use one of two special values: ANY or EMPTY.

• The ANY specification says that the element may contain any other defined element, or PCDATA.

• The EMPTY specification says that the element contains no contents.

Referencing the DTD

<!DOCTYPE slideshow SYSTEM "slideshow.dtd">

The DTD tag starts with "<!". The tag name, DOCTYPE, says that the document is a slideshow, which means that the document consists of the slideshow element and everything within it: <slideshow> ... </slideshow>

The DOCTYPE tag occurs after the XML declaration and before the root element. The SYSTEM identifier specifies the location of the DTD file. Since it does not start with a prefix like http:/ or file:/, the path is relative to the location of the XML document.

Referencing the DTD(2)

The DOCTYPE specification could also contain DTD definitions within the XML document, rather than referring to an external DTD file. Such definitions would be contained in square brackets:

<!DOCTYPE slideshow SYSTEM "slideshow1.dtd" ***[***

...local subset definitions here...

]>

Defining attributes in DTD

```
<!ATTLIST   slideshow
            title       CDATA     #REQUIRED
            date        CDATA     #IMPLIED
            author      CDATA     "unknown" >
```

The DTD tag ATTLIST begins the series of attribute definitions. The name that follows ATTLIST specifies the element for which the attributes are being defined. In this case, the element is the slideshow element.

- Each attribute is defined by a series of three space-separated values. Commas and other separators are not allowed.
- **The first element** in each line is the name of the attribute: title, date, or author, in this case.
- **The second element** indicates the type of the data: CDATA is character data

Defining attributes in DTD(2)

The last entry in the attribute specification determines the attributes default value, if any, and tells whether or not the attribute is required. The possible choices:

#REQUIRED - The attribute value must be specified in the document.
#IMPLIED - The value need not be specified in the document. If it isn't, the application will have a default value it uses.
"defaultValue" - The default value to use, if a value is not specified in the document.
#FIXED "fixedValue" - The value to use. If the document specifies any value at all, it must be the same.

Defining entities in DTD

```
<!DOCTYPE slideshow SYSTEM "slideshow1.dtd" [
<!ENTITY product "Wonder">
<!ENTITY products "Wonder"> ]>
```

The ENTITY tag name says that you are defining an entity. Next comes the name of the entity and its definition. In this case, you are defining an entity named "product" that will take the place of the product name. Later when the product name changes you will only have to change the name one place, and all your slides will reflect the new value.

The last part is the substitution string that replaces the entity name whenever it is referenced in the XML document. The substitution string is defined in quotes, which are not included when the text is inserted into the document.

These kind of definitions should be in external DTD.

Using entities in XML

```
<slideshow title="&product; Slide Show" ...

<!-- TITLE SLIDE -->

        <slide type="all">
                <title>Wake up to &products;!</title>
        </slide>
```

Useful entities

Several other examples for entity definitions that you might find useful when you write an XML document:

```
<!ENTITY ldquo "&#147;"> <!-- Left Double Quote -->
<!ENTITY rdquo "&#148;"> <!-- Right Double Quote -->
<!ENTITY trade "&#153;"> <!-- Trademark Symbol (TM) -->
<!ENTITY rtrade "&#174;"> <!-- Registered Trademark (R) -->
<!ENTITY copyr "&#169;"> <!-- Copyright Symbol -->
```

Referencing external entities

Example:

```
<!DOCTYPE slideshow SYSTEM "slideshow.dtd" [
<!ENTITY product "Wonder">
<!ENTITY products "Wonder">
<!ENTITY copyright SYSTEM "copyright.xml">
]>
```

Copyright.xml:

```
<!-- A SAMPLE copyright -->
This is the standard copyright message that our lawyers make us
put everywhere .....
```

Parameter entities

Just as a general entity lets you reuse XML data in multiple places, a parameter entity lets you reuse parts of a DTD in multiple places.

```
<!ENTITY % inline "#PCDATA|em|b|a|img|br">
<!ELEMENT title (%inline;)*>
<!ELEMENT item (%inline; | item)* >
```

Conditional sections

Suppose, for example, that you wanted to use slightly different versions of a DTD, depending on whether you were treating the document as an XML document or as a SGML document. You could do that with DTD definitions like the following:

someExternal.dtd:

<![**INCLUDE** [... XML-only definitions]]>
<![**IGNORE** [... SGML-only definitions]]>
... common definitions

The conditional sections are introduced by "<![", followed by the INCLUDE or IGNORE keyword and another "[". After that comes the contents of the conditional section, followed by the terminator: "]]>". In this case, the XML definitions are included, and the SGML definitions are excluded. That's fine for XML documents, but you can't use the DTD for SGML documents. You could change the keywords, of course, but that only reverses the problem.

Conditional sections(2)

The solution is to use references to parameter entities in place of the INCLUDE and IGNORE keywords:

someExternal.dtd:

```
<![ %XML; [ ... XML-only definitions ]]>
<![ %SGML; [ ... SGML-only definitions ]]>
... common definitions
```

Then each document that uses the DTD can set up the appropriate entity definitions:

```
<!DOCTYPE foo SYSTEM "someExternal.dtd" [
<!ENTITY % XML "INCLUDE" >
<!ENTITY % SGML "IGNORE" >
]>
```

Using namespaces

The primary goal of the namespace specification is to let the document author tell the parser which DTD to use when parsing a given element. The parser can then consult the appropriate DTD for an element definition.

Conflict Example:

You can use <title> element in your book.xml. But for other purposes you can reference xhtml.dtd (because you will use XHTML) which already defines this element. How to avoid disambiguity?

When a document uses an element name that exists in only one of the .dtd files it references, the name does not need to be qualified. But when an element name that has multiple definitions is used, some sort of qualification is a necessity.

Using namespaces(2)

You qualify a reference to an element name by specifying the xmlns attribute:

```
<title xmlns="http://www.example.com/slideshow">
          Overview
</title>
```

The alternative is to define a *namespace prefix*:

```
 <SL:slideshow xmlns:SL='http:/www.example.com/slideshow' ...>
          ...
          <slide>
                    <SL:title>Overview<SL:title>
          </slide>
           ...
</SL:slideshow>
```

SAX & DOM

SAX - Simple API for XML

The "serial access" protocol for XML. This is the fast-to-execute mechanism you would use to read and write XML data. This is also called an event-driven protocol, because the technique is to register your handler with a SAX parser, after which the parser invokes your callback methods whenever it sees a new XML tag (or encounters an error, or wants to tell you anything else).

DOM - Document Object Model

The Document Object Model protocol converts an XML document into a collection of objects in your program. You can then manipulate the object model in any way that makes sense. This mechanism is also known as the "random access" protocol, because you can visit any part of the data at any time. You can then modify the data, remove it, or insert new data

Java API for XML processing

The main JAXP APIs are defined in the *javax.xml.parsers* package.

That package contains two vendor-neutral factory classes: SAXParserFactory and DocumentBuilderFactory that give you a SAXParser and a DocumentBuilder, respectively.

The DocumentBuilder creates DOM-compliant Document object.

Overview of packages

javax.xml.parsers

The JAXP APIs, which provide a common interface for different vendors' SAX and DOM parsers.

org.w3c.dom

Defines the Document class (a DOM), as well as classes for all of the components of a DOM.

org.xml.sax

Defines the basic SAX APIs.

javax.xml.transform

Defines the XSLT APIs that let you transform XML into other forms.

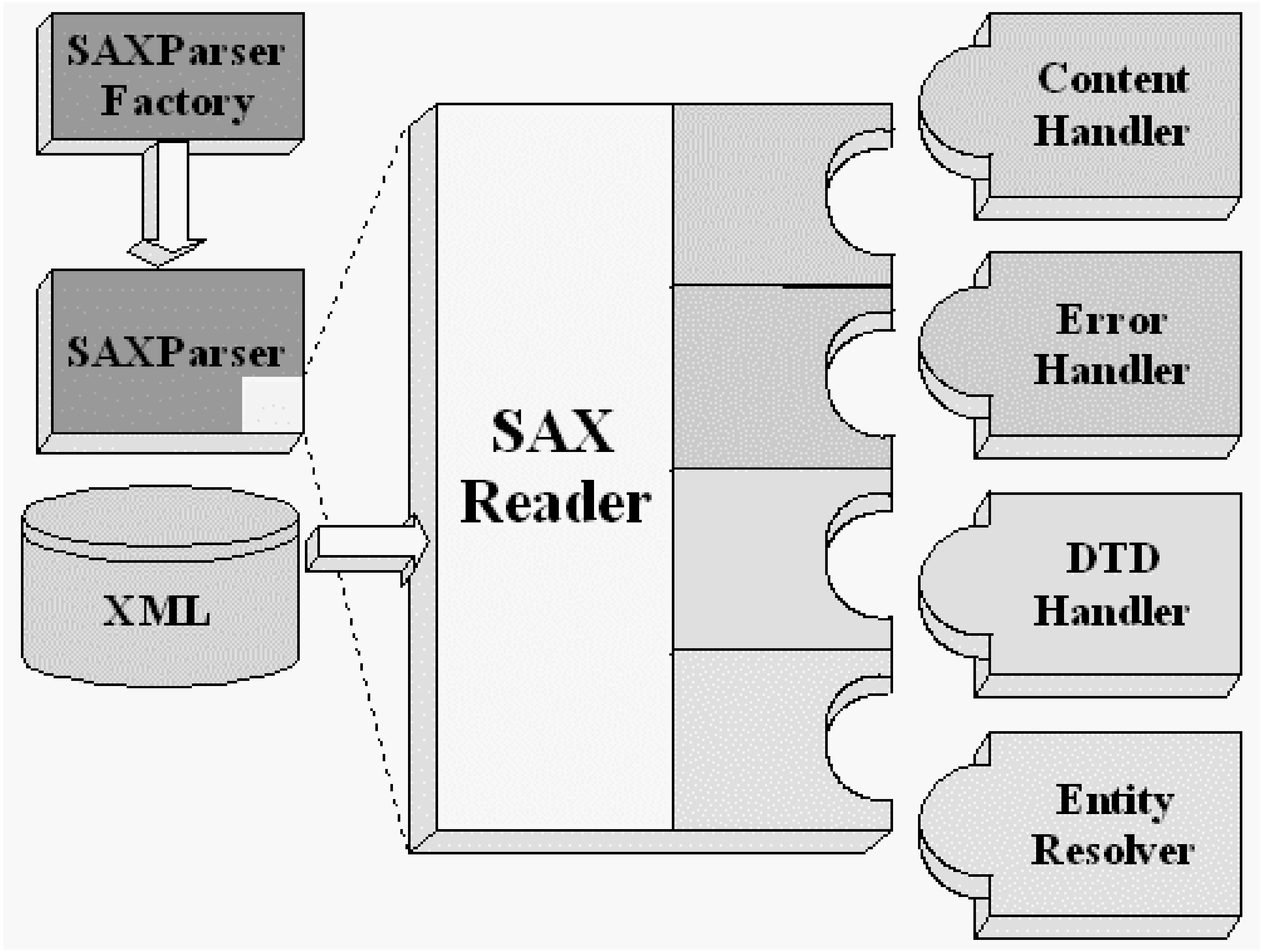
URI, URL, URN

URI - A "Universal Resource Identifier". A URI is either a URL or a URN. (URLs and URNs are concrete entities that actually exist. A "URI" is an abstract superclass -- it's a name we can use when we know we are dealing with either an URL or an URN, and we don't care which.

URL - Universal Resource Locator. A pointer to a specific location (address) on the Web that is unique in all the world. The first part of the URL defines the type of address. For example, http:/ identifies a Web location.

URN - Universal Resource Name. A unique identifier that identifies an entity, but doesn't tell where it is located. That lets the system look it up to see if a local copy exists before going out to find it on the Web. It also allows the web location to change, while still allowing the object to be found.

SAX architecture



SAX architecture(2)

SAXParserFactory

A SAXParserFactory object creates an instance of the parser determined by the system property, javax.xml.parsers.SAXParserFactory.

SAXParser

The SAXParser interface defines several kinds of parse() methods. In general, you pass an XML data source and a DefaultHandler object to the parser, which processes the XML and invokes the appropriate methods in the handler object.

SAX architecture(3)

SAXReader

The SAXParser wraps a SAXReader.

DefaultHandler

DefaultHandler implements the ContentHandler, ErrorHandler, DTDHandler, and EntityResolver interfaces (with null methods), so you can override only the ones you're interested in.

ContentHandler

Methods like startDocument, endDocument, startElement, and endElement are invoked when an XML tag is recognized.

SAX architecture(4)

ErrorHandler

Methods error, fatalError, and warning are invoked in response to various parsing errors. The default error handler throws an exception for fatal errors and ignores other errors (including validation errors).

DTDHandler

Defines methods you will generally never be called upon to use. Used when processing a DTD .

EntityResolver

The resolveEntity method is invoked when the parser must identify data identified by a URI.

Echoing XML with the SAX

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.*;
public class Echo01 extends DefaultHandler {
    public static void main(String argv[])    {
        if (argv.length != 1) {
            System.err.println("Usage: cmd filename");
            System.exit(1);
            }
```

Echoing XML with the SAX(2)

```
// Use an instance of ourselves as the SAX event handler
DefaultHandler handler = new Echo01();
// Use the default (non-validating) parser
SAXParserFactory factory =
                        SAXParserFactory.newInstance();
try {
    // Set up output stream
    out = new OutputStreamWriter(System.out, "UTF8");
    // Parse the input
    SAXParser saxParser = factory.newSAXParser();
    saxParser.parse( new File(argv[0]), handler);
```

Echoing XML with the SAX(3)

```
        } catch (Throwable t) {
         t.printStackTrace();
      }
      System.exit(0);
   }
   static private Writer  out;
```

Echoing XML with the SAX(4)

```
    public void startDocument()    throws SAXException    {
    emit("<?xml version='1.0' encoding='UTF-8'?>"); // to output
     nl(); // inserts new line
 }
 public void endDocument()   throws SAXException    {
     try {
         nl();
         out.flush();
     } catch (IOException e) {
         throw new SAXException("I/O error", e);
     }   }
```

Presented by Bartosz Sakowicz

DMCS TUL

Echoing XML with the SAX(5)

```
public void startElement(String namespaceURI,
                         String lName, // local name
                         String qName, // qualified name
                         Attributes attrs)    throws SAXException    {
    String eName = lName; // element name
    if ("".equals(eName)) eName = qName;
      // namespaceAware = false
   emit("<"+eName);
```

Echoing XML with the SAX(6)

```
    if (attrs != null) {
        for (int i = 0; i < attrs.getLength(); i++) {
            String aName = attrs.getLocalName(i); // Attr name
            if ("".equals(aName)) aName = attrs.getQName(i);
            emit(" ");
            emit(aName+"=\""+attrs.getValue(i)+"\"");
        }
    }
    emit(">");
}
```

Echoing XML with the SAX(7)

```
  public void endElement(String namespaceURI,
                      String sName, // simple name
                      String qName  // qualified name
                    )    throws SAXException    {
      emit("</"+sName+">");  // or possibly qName
  }
  public void characters(char buf[], int offset, int len)
  throws SAXException    { // processes the tags body
      String s = new String(buf, offset, len);
      emit(s);
}
```

Echoing XML with the SAX(8)

```
 // Wrap I/O exceptions in SAX exceptions, to
// suit handler signature requirements
private void emit(String s)
throws SAXException
{
   try {
      out.write(s);
      out.flush();
   } catch (IOException e) {
      throw new SAXException("I/O error", e);
   }   }
```

Echoing XML with the SAX(9)

```
// Start a new line
private void nl()
throws SAXException
{
    String lineEnd =  System.getProperty("line.separator");
    try {
        out.write(lineEnd);
    } catch (IOException e) {
        throw new SAXException("I/O error", e);
    }   }  }
```

Processing instructions

It sometimes makes sense to code application-specific processing instructions in the XML data.

```
<slideshow ... >

<!-- PROCESSING INSTRUCTION -->
<?my.presentation.Program QUERY="exec, tech, all"?>
```

- The "data" portion of the processing instruction can contain spaces, or may even be null. But there cannot be any space between the initial <? and the target identifier.
- The data begins after the first space.

Processing instructions(2)

```
public void processingInstruction(String target, String data)
          throws SAXException {
         nl();
         emit("PROCESS: ");
         emit("<?"+target+" "+data+"?>");
}
```

Using validating parser

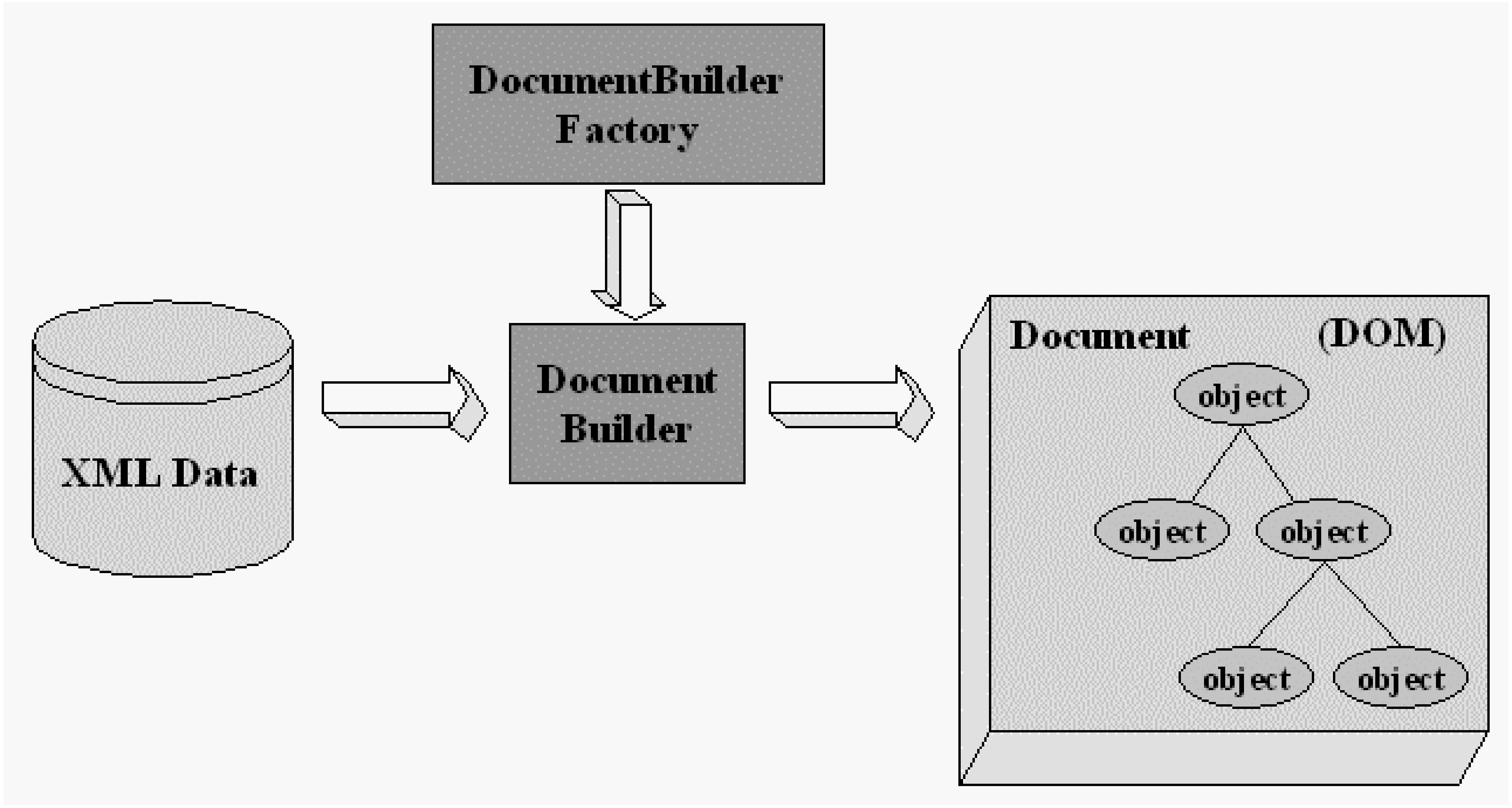
```
...

SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);

...
```

To use validating parser a DTD is required. The Validating parser will inform about all incompabilities between DTD and XML document.

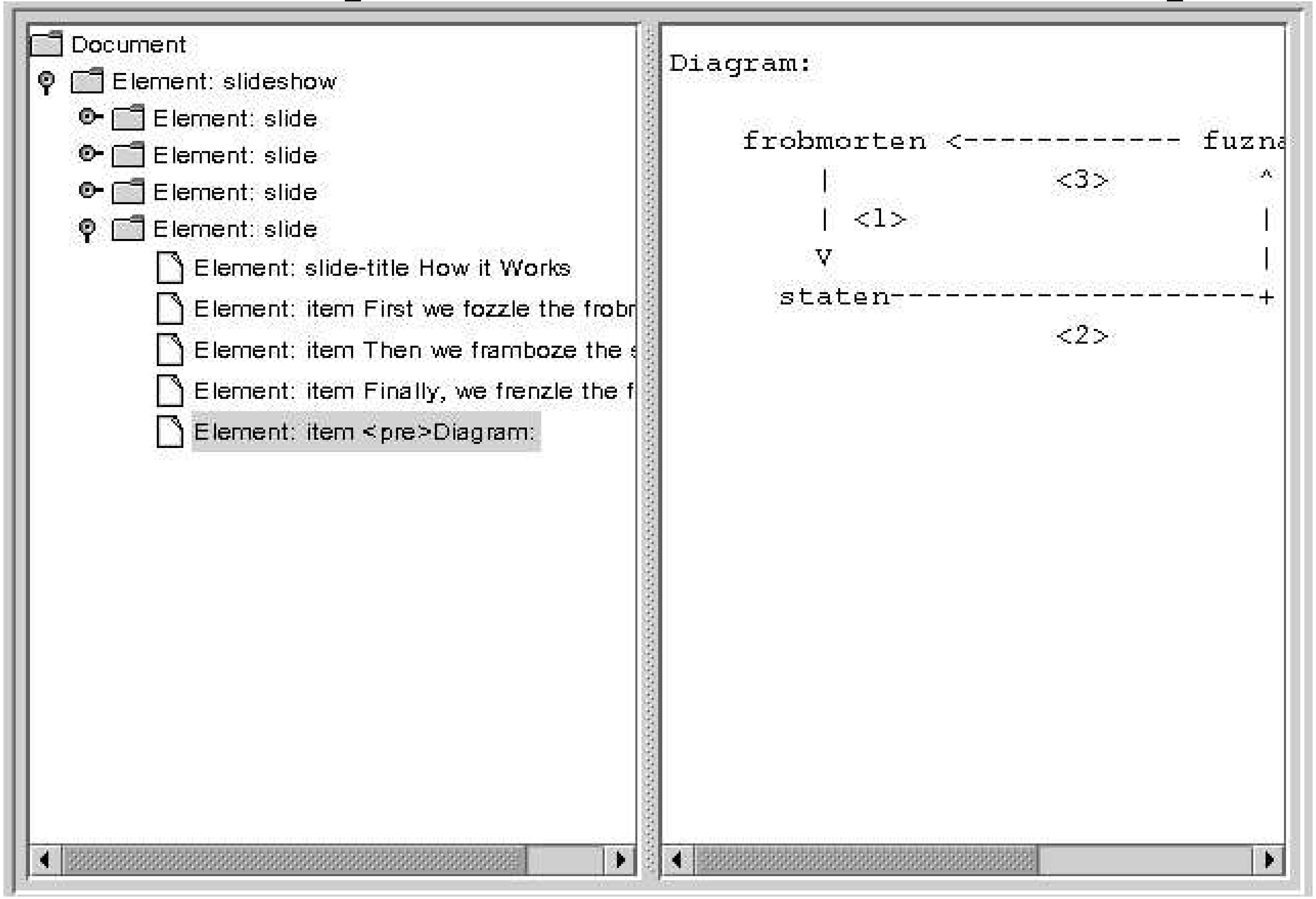
DOM architecture



Creating document

```
public static void main(String argv[]) {
        if (argv.length != 1) { ... }
        DocumentBuilderFactory factory =
        DocumentBuilderFactory.newInstance();
        try {
                DocumentBuilder builder =
                        factory.newDocumentBuilder();
                Document document =
                        builder.parse( new File(argv[0]) );
        } catch (SAXParseException spe) { ... }
// The same as SAX !
```

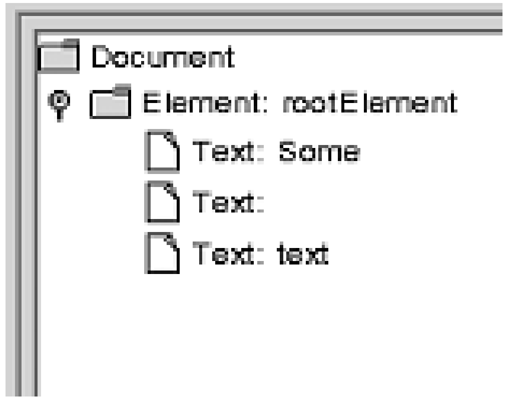
DOM representation example



Document
Element: slideshow
Element: slide
Element: slide
Element: slide
Element: slide
Element: slide-title How it Works
Element: item First we fozzle the frobr
Element: item Then we framboze the s
Element: item Finally, we frenzle the f
Element: item <pre>Diagram:

```
Diagram:

    frobmorten <------------ fuzna
        |            <3>         ^
        | <1>                    |
        V                        |
      staten---------------------+
                     <2>
```



Building DOM

```
try {
DocumentBuilder builder = factory.newDocumentBuilder();
 document = builder.newDocument();
Element root = (Element)
        document.createElement("rootElement");
 document.appendChild(root);
root.appendChild( document.createTextNode("Some") );
root.appendChild( document.createTextNode(" ") );
root.appendChild( document.createTextNode("text") );
 } catch (ParserConfigurationException pce) {
```

Manipulating DOM

Traversing Nodes

The org.w3c.dom.Node interface defines a number of methods you can use to traverse nodes, including ***getFirstChild, getLastChild, getNextSibling, getPreviousSibling, and getParentNode***. Those operations are sufficient to get from anywhere in the tree to any other location in the tree.

Creating Attributes

The org.w3c.dom.Element interface, which extends *Node*, defines a ***setAttribute*** operation, which adds an attribute to that node.

Removing and Changing Nodes

To remove a node, you use its parent Node's ***removeChild*** method. To change it, you can either use the parent node's ***replaceChild*** operation or the node's ***setNodeValue*** operation.

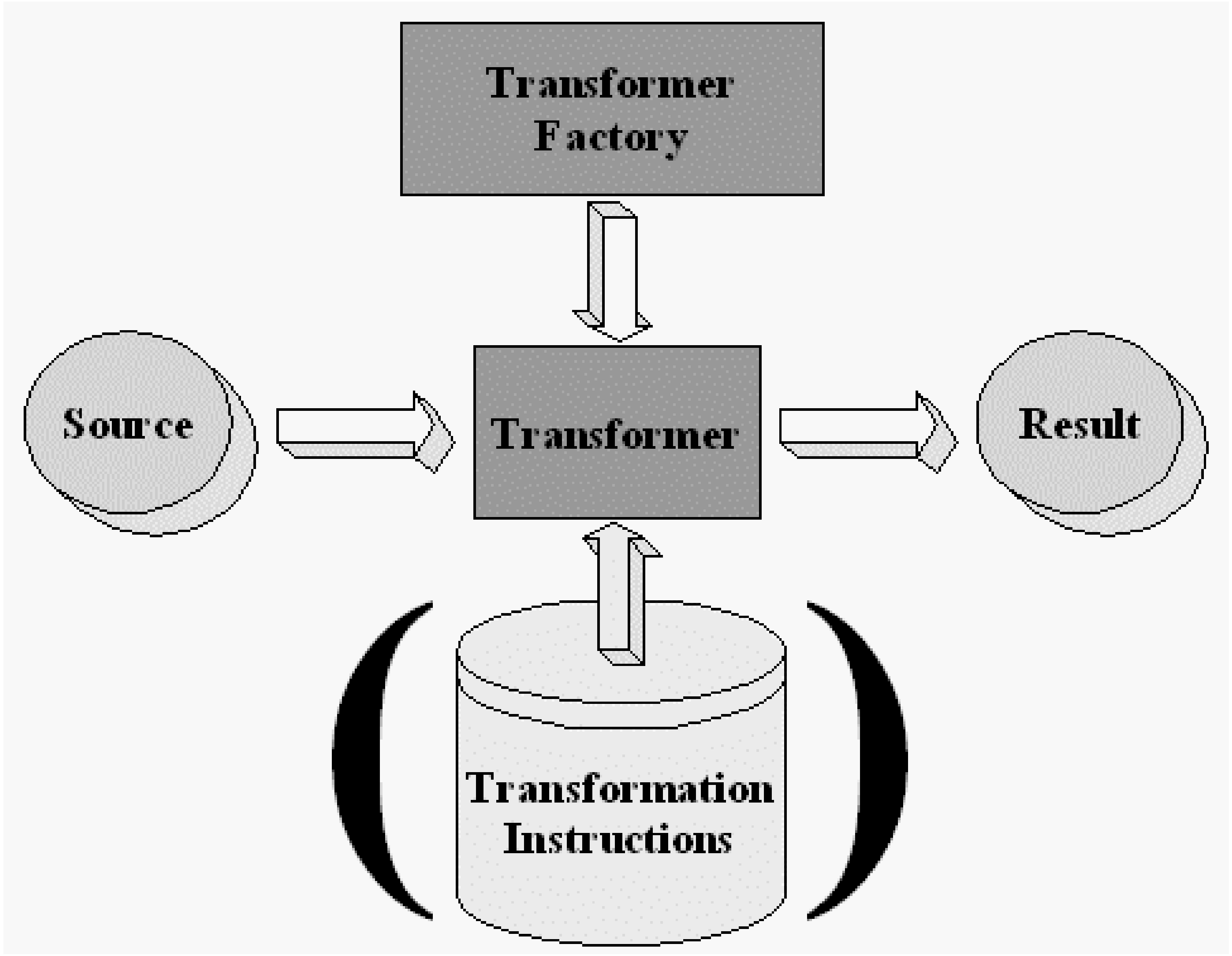
XSLT + XPATH

The XSLT (Extensible Stylesheet Language for Transformations) transformation standard is essentially a translation mechanism that lets you specify what to convert an XML tag into so that it can be displayed -- for example, in HTML.

Different XSL formats can then be used to display the same data in different ways, for different uses.

The XPATH standard is an addressing mechanism that you use when constructing transformation instructions, in order to specify the parts of the XML structure you want to transform.

XSLT transformation schema



XSLT usage example

Sample input XML:

```
<?xml version="1.0"?>
<ARTICLE>
	<TITLE>A Sample Article</TITLE>
	<SECT>The First Major Section
		<PARA>This section will introduce a subsection.
		</PARA>
		<SECT>The Subsection Heading
			<PARA>This is the text of the subsection.
			</PARA>
		</SECT>
	</SECT>
</ARTICLE>
```

XSLT usage example(2)

Prefered output:

```
<html>
<body>
        <h1 align="center">A Sample Article</h1>
        <h1>The First Major Section</h1>
                <p>This section will introduce a subsection.</p>
                <h2>The Subsection Heading</h2>
                        <p>This is the text of the subsection.</p>
</body>
</html>
```

XSLT usage example(3)

XSLT Stylesheet:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<xsl:stylesheet
xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0"
>
<xsl:template match="/">
        <html><body> <xsl:apply-templates/> </body></html>
</xsl:template>
<xsl:template match="/ARTICLE/TITLE">
        <h1 align="center">
        <xsl:apply-templates/>
        </h1>
</xsl:template>
```

XSLT usage example(4)

```
<!-- Top Level Heading -->
<xsl:template match="/ARTICLE/SECT">
          <h1> <xsl:apply-templates select="text()|B|I|U|DEF|LINK"/>
           </h1>
          <xsl:apply-templates select="SECT|PARA/>
</xsl:template>

<!-- Second-Level Heading -->
<xsl:template match="/ARTICLE/SECT/SECT">
          <h2> <xsl:apply-templates select="text()|B|I|U|DEF|LINK"/>
          </h2>
          <xsl:apply-templates select="SECT|PARA"/>
</xsl:template>
```

XSLT usage example(5)

```
<!-- Third-Level Heading -->
<xsl:template match="/ARTICLE/SECT/SECT/SECT">
        <xsl:message terminate="yes">Error: Sections can only be
nested 2 deep.
        </xsl:message>
</xsl:template>
<!-- Paragraph -->
<xsl:template match="PARA">
        <p><xsl:apply-templates/></p>
</xsl:template>
<!-- Text -->
<xsl:template match="text()">
        <xsl:value-of select="normalize-space()"/>
</xsl:template>

</xsl:stylesheet>
```