

---

# Java & XML

## Objectives

Learn the basics of processing XML with Java

- How to read XML documents
- How to write XML documents

Get familiar with the three common techniques used for processing XML docs with Java

- SAX (Simple API for XML)
- StAX (Streaming API for XML)
- DOM (Document Object Model)

# Reminder: XML

- XML data (not necessarily stored in files) is organized as a hierarchical tree, with optional attributes for the nodes:

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

```
<root attr1="example">
```

```
  <level1 attr2="abc">
```

```
    <level2>Content of level 2</level2>
```

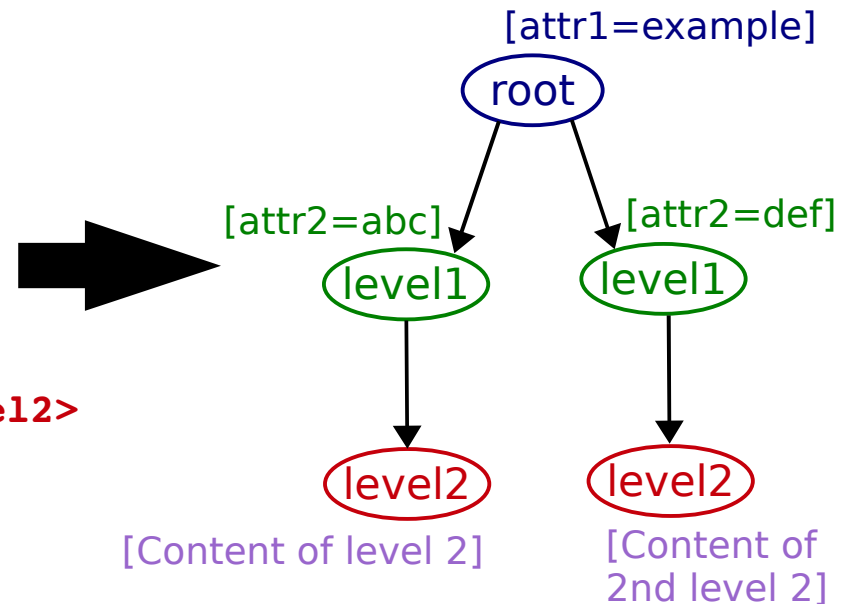
```
  </level1>
```

```
  <level1 attr2="def">
```

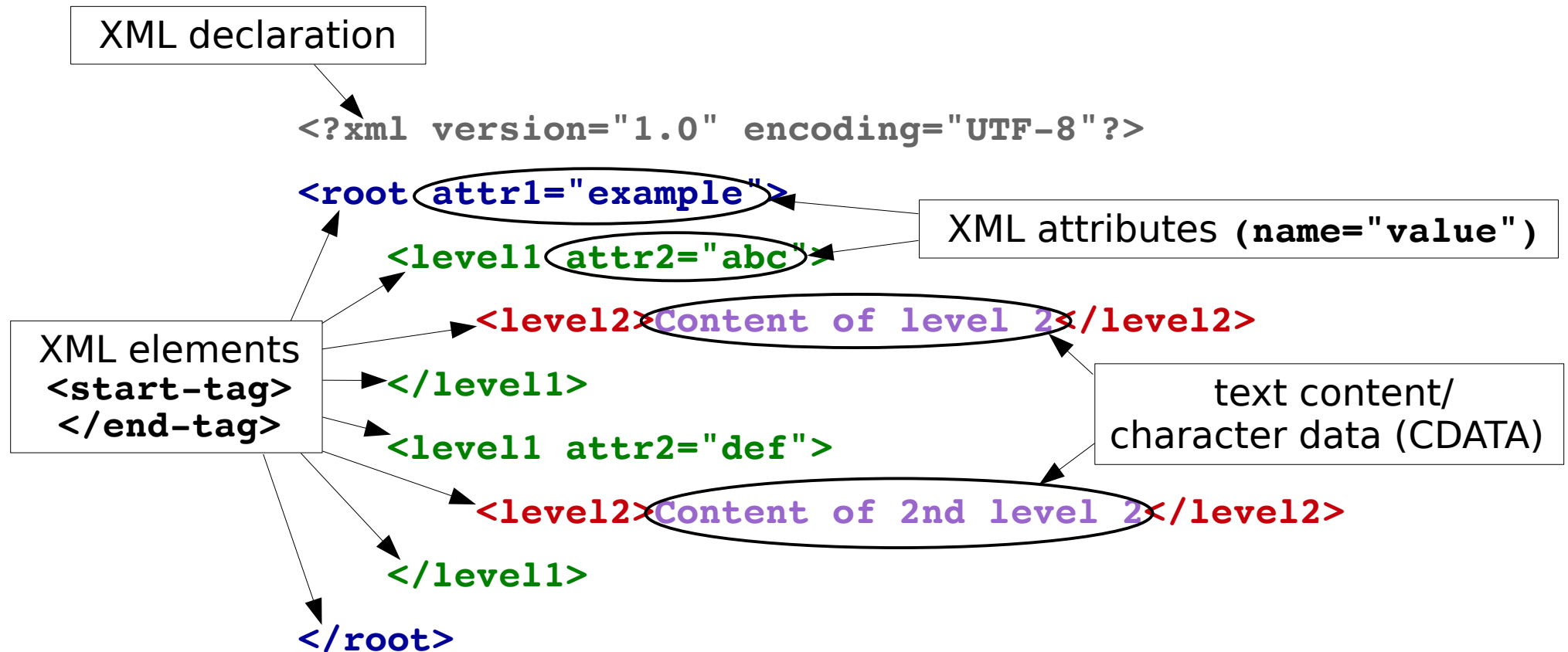
```
    <level2>Content of 2nd level 2</level2>
```

```
  </level1>
```

```
</root>
```



# Reminder: XML Terminology



# Reminder: XML Terminology

- ♦ **Parsing** XML data means reading an XML document into the application's memory
- ♦ Parsing XML resembles *traversing* a tree
- ♦ **Serialize** XML is the opposite of parsing XML, i.e., XML data is written to a document
- ♦ **Well-formedness** of an XML document means that:
  - ♦ the document has exactly one root element and
  - ♦ all begin- and end-tags of elements are correctly nested, with none missing and none overlapping

# Java and XML

- ♦ Java provides a broad functionality for dealing with XML data
- ♦ Two main mechanisms to parse XML:
  - ♦ To parse your data sequentially as a stream of events
  - ♦ To build an object representation of it
- ♦ Several XML toolkits are available for Java, e.g.:
  - ♦ JAXP is part of the Java platform
  - ♦ JDOM is open source

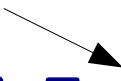
# Java API for XML Processing (JAXP)

- ♦ **JAXP** is a Java API for processing XML data
- ♦ Independent of a particular XML processor implementation
- ♦ Provides the capability of parsing, validating, transforming, and querying XML documents
- ♦ Implements common XML techniques such as:
  - ♦ **SAX** (Simple API for XML)
  - ♦ **DOM** (Document Object Model)
  - ♦ **StAX** (Streaming API for XML)

# Java Factories

- ♦ A **Factory** (or **virtual constructor**) is an object which can be used to create another object
- ♦ The **Factory** object has a static method to create the new **Factory** object (**Factory.newInstance()**)
- ♦ This **Factory** object offers static methods for creating new objects (e.g., **newDocument()**)

```
Factory factory = Factory.newInstance();  
Builder builder = factory.newDocument();
```



- ♦ **Factories** are often used in APIs that have **more than one** possible implementation
- ♦ JAXP provides several **Factories** for creating objects that can be used for processing XML

# SAX Parsing

- ♦ SAX (Simple API for XML) allows you to parse your XML data as a stream of events
- ♦ Traverses the XML document from the beginning to the end (serial) while interpreting the XML syntax
- ♦ A SAX parser reacts on XML components while reading the XML data and creates (SAX-)events:
  - ♦ Start/end reading a document
  - ♦ Opening/closing an element
  - ♦ etc.



# SAX Parsing

```
<root>  
  <level1>Content of level 1</level1>  
  <level1>Content of 2nd level 1</level1>  
</root>
```

- ♦ SAX events that are created by the SAX parser while reading the example document:
  1. document starts
  2. start tag **<root>** found
  3. start tag **<level1>** found
  4. text **"Content of level 1"** found
  5. end tag **</level1>** found
  6. start tag **<level1>** found
  7. text **"Content of 2nd level 1"** found
  8. end tag **</level1>** found
  9. end tag **</root>** found
  10. document ends

# SAX Parsing

- ♦ SAX parsing is like “waiting” for the content of an XML document which will be delivered by the parser
- ♦ We do not have to care about the flow of control, but wait for the events to *push* our class (“inversion of control”)
- ♦ This is also called *push parsing*

# SAX (Dis-)Advantages

- ♦ Advantages

- ♦ Fast (good performance), few overhead
- ♦ Can handle big amounts of XML data

- ♦ Disadvantages

- ♦ Linear access to the XML data (we only see that part of the document that the parser is processing at a certain moment)
- ♦ Access to the information of one node at a time – no access to the tree context/hierarchy
- ♦ No write support: XML data is parsed in readonly modus  
拉丁语 read mode

# Loading an XML File with SAX

- ♦ SAX Parsers can be used to access huge amounts of XML data in a serial way

- ♦ Your class has to be extended:

```
import org.xml.sax.helpers.DefaultHandler;  
public class ParseXml extends DefaultHandler {
```

- ♦ Creating a SAX parser:

```
DefaultHandler handler = new ParseXml();  
SAXParser saxParser = SAXParserFactory.newInstance().newSAXParser();  
saxParser.parse(new File("file.xml"), handler);
```

→ The parsing process starts and the XML data is *pushed into your application* (push parser)

# Reacting on SAX Events

- ♦ To catch the pushed in data, methods have to be written which are reacting on XML events:

```
public void startDocument()  
  
public void startElement(String namespaceURI,  
                        String localName, // local name  
                        String qName, // qualified name  
                        Attributes attrs)  
  
public void endElement(String namespaceURI,  
                      String localName, // local name  
                      String qName)  
  
public void characters(char[] buf, int offset, int len)  
  
public void endDocument()
```

# Reacting on SAX Events

```
public void startElement(String namespaceURI,  
                        String localName, // local name  
                        String qName, // qualified name  
                        Attributes attrs)  
                        throws SAXException {  
  
    if(qName.equals("level1")){  
  
        String temp = attrs.getValue(attrs.getIndex("attr2"));  
  
        //Do something ...  
  
    }  
  
    ...  
  
}
```

# StAX Parsing

- ♦ **StAX (Streaming API for XML)** allows you to parse XML data serial, like SAX
- ♦ Our calling class has the control over the parsing process and pulls (reads) the data from the XML parser
- ♦ This is also called ***pull parsing***
- ♦ Two variants of StAX
  - ♦ ***Cursor API***
  - ♦ ***Event-iterator API***  
(we take a look at this variant)

# StAX Parsing

- ♦ The event-iterator API allows high-level access to XML events
- ♦ A parser of the event-iterator API represents an **Iterator** over **XMLEvent** objects
- ♦ This **Iterator** provides XML events one after the other
- ♦ Methods **hasNext()** and **next()** iterate on XML components
- ♦ Every XML component is represented by a separate Java interface that is derived from **XMLEvent**



---

# StAX (Dis-)Advantages

- ♦ Advantages
  - ♦ Can handle huge amounts of XML data
  - ♦ Additionally, it can create (serialize) XML documents
- ♦ Disadvantages
  - ♦ No treeview available
  - ♦ Not possible to modify an existing XML document
  - ♦ Usually more complex than SAX

# Loading an XML File with StAX

- ♦ Necessary imports:

```
import javax.xml.stream.*; //provides XMLStreamReader/-factory
import javax.xml.stream.events.*; //all types of XMLEvents
```

- ♦ Creating a StAX parser:

```
XMLInputFactory inputFactory = XMLInputFactory.newInstance();
XMLStreamReader eventReader = inputFactory.createXMLStreamReader(
    new StreamSource(new File("file.xml")));
```

- ♦ **XMLInputFactory**: Defines a factory API that enables applications to obtain a parser that represents an Iterator over **XMLEvent** objects from XML documents.
- ♦ **XMLStreamReader**: Defines the API to obtain **XMLEvents**. Using this class, an application programmer can iterate over **XMLEvent** objects.

# StAX Reading

- ♦ Iterating through all **XMLEvents** (representing XML components) in the whole XML document:

```
while (eventReader.hasNext()) {  
    XMLEvent event = eventReader.nextEvent();  
  
    // process XML event  
    ...  
}
```

- ♦ **XMLEvent**: An **XMLEvent** is the base interface for all the other (more specific) interfaces such as **StartDocument**, **EndElement**, **Characters**, **Attribute**, etc.

# StAX Event Types

- ♦ There is a separate interface for each XML event that might occur in an XML document
- ♦ All XML event types inherit from **XMLEvent**
- ♦ All interfaces are in package: **javax.xml.stream.events**
- ♦ Some of the interfaces:

Interface name	Interface represents
<b>XMLEvent</b>	base event interface for handling markup events
<b>StartDocument</b>	start of the document
<b>EndDocument</b>	end of the document
<b>StartElement</b>	start of an element
<b>EndElement</b>	end of an element
<b>Attribute</b>	an attribute
<b>Characters</b>	content, CData, and whitespace
<b>Comment</b>	a comment

# StAX Event Types

- ◆ Retrieve event type (as **int**) with method **XMLEvent.getEventType()**:
  - 1: StartElement**
  - 2: EndElement**
  - 4: Characters**
  - 5: Comment**
  - 7: StartDocument**
  - 8: EndDocument**
  - 10: Attribute**
  - ...**
- ◆ Or with individual methods (as **boolean** values):
  - XMLEvent.isStartElement()**
  - XMLEvent.isEndElement()**
  - XMLEvent.isAttribute()**
  - XMLEvent.isEndDocument()**
  - XMLEvent.isCharacters()**
  - ...**

# StAX Event Types

- ♦ In order to query the properties of a specific event, we might want to cast an **XMLEvent** to the subinterface that it represents
- ♦ Casting may result in a class cast exception if the event to be casted is of another type

```
if (event.isStartElement()){ //prevent class cast exception
    StartElement se = event.asStartElement(); //casting
    System.out.println(se.getName()); //access properties
}
```

```
if (event.isCharacters()) { //prevent class cast exception
    Characters c = event.asCharacters(); //casting
    System.out.println(c.getData()); //access properties
}
```

# Reading the Whole Document with StAX

```
Stack<String> stack = new Stack<String>();
while (eventReader.hasNext()) {
    XMLEvent event = eventReader.nextEvent();
    if (event.isStartElement()) {
        stack.push(event.asStartElement().getName().getLocalPart());
        Iterator<Attribute> it = event.asStartElement().getAttributes();
        while (it.hasNext()) {
            Attribute a = it.next();
            System.out.println(stack + " " + a.getName().getLocalPart()
                               + "=\\"" + a.getValue() + "\"");
        }
    }
    if (event.isCharacters()) {
        String s = event.asCharacters().getData();
        if (s.trim().length() > 0) {
            System.out.println(stack + " \\"" + s + "\"");
        }
    }
    if (event.isEndElement()) {
        stack.pop();
    }
}
```

Example output:

```
[root] attr1="example"
[root, level1] attr2="abc"
[root, level1, level2] "Content of level 2"
[root, level1] attr2="def"
[root, level1, level2] "Content of 2nd level 2"
```

Example taken from:

<http://www.torsten-horn.de/techdocs/java-xml.htm>

# StAX Writing

- ♦ Necessary imports:

```
import javax.xml.stream.*; //provides XMLEventReader/-factory
```

- ♦ Creating a StAX writer:

```
XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
XMLEventWriter eventWriter = outputFactory.createXMLEventWriter(
    new FileOutputStream("out.xml"));
XMLEventFactory eventFactory = XMLEventFactory.newInstance();
```

- ♦ **XMLOutputFactory**: Defines a factory API that enables applications to obtain XML writers.
- ♦ **XMLEventWriter**: Defines the interface to write XML documents.
- ♦ **XMLEventFactory**: Defines the interface for creating instances of **XMLEvents**.



# StAX Writing

- ♦ **Procedure** to write an XML document with StAX:
  1. **Get XMLEvents**: either take **XMLEvents** from a parsed XML document or create new **XMLEvents** with the help of an **XMLEventFactory**.
  2. **Write XMLEvents**: an XML document can be written with the help of an **XMLEventWriter**.
- ♦ The order in which **XMLEvents** are written determines the structure of the resulting XML document.
- ♦ The **order** in which the **XMLEvents** are created is independent of the output.

# Creating XMLEvents in StAX

- Some important methods of the **XMLEventFactory** for creating **XMLEvents**:

Method	Parameter	Return type
<code>newInstance</code>		<code>XMLEventFactory</code>
<code>createStartDocument</code>	<code>String encoding, String version, boolean standalone</code>	<code>StartDocument</code>
<code>createEndDocument</code>		<code>EndDocument</code>
<code>createStartElement</code>	<code>String prefix, String namespaceUri, String localName</code>	<code>StartElement</code>
<code>createEndElement</code>	<code>String prefix, String namespaceUri, String localName</code>	<code>EndElement</code>
<code>createAttribute</code>	<code>String localName, String value</code>	<code>Attribute</code>
<code>createCharacters</code>	<code>String content</code>	<code>Characters</code>
<code>createComment</code>	<code>String text</code>	<code>Comment</code>

# Creating XMLEvents in StAX

- ◆ Some examples of creating XMLEvents:

```
StartDocument startDocument = eventFactory.createStartDocument();
StartElement startRoot = eventFactory.createStartElement(
    "", "", "root");
Attribute attr1 = eventFactory.createAttribute("attr1", "example");
EndElement endRoot = eventFactory.createEndElement("", "", "root");
StartElement startLevel1 = eventFactory.createStartElement(
    "", "", "level1");
Attribute attr2 = eventFactory.createAttribute("attr2", "abc");
StartElement startLevel2 = eventFactory.createStartElement(
    "", "", "level2");
Characters contentLevel2 = eventFactory.createCharacters(
    "Content of level 2");
EndElement endLevel2 = eventFactory.createEndElement("", "", "level2");
EndElement endLevel1 = eventFactory.createEndElement("", "", "level1");
EndDocument endDocument = eventFactory.createEndDocument();
```

# Writing XMLEvents in StAX

- Assuming we have created an **XMLEventWriter** and all the **XMLEvents** from the last slides, we can write the data to an XML document:

```
writer.add(startDocument);  
writer.add(startRoot);  
writer.add(attr1);  
writer.add(startLevel1);  
writer.add(attr2);  
writer.add(startLevel2);  
writer.add(contentLevel2);  
writer.add(endLevel2);  
writer.add(endLevel1);  
writer.add(endRoot);  
writer.add(endDocument);  
writer.close(); //do not forget to close the writer
```

- This output is not well formatted: it will produce a file with all content on one line and without spaces.

# Writing XMLEvents in StAX

- It is better to create further **Characters** events in order to adjust line breaks and indentation. For example:

```
Characters indent = eventFactory.createCharacters("    ");
Characters newLine = eventFactory.createCharacters("\n");
```

```
writer.add(startDocument);
writer.add(newLine);
writer.add(startRoot);
writer.add(attr1);
writer.add(newLine);
writer.add(indent);
writer.add(startLevel1);
writer.add(attr2);
writer.add(newLine);
writer.add(indent);
writer.add(indent);
writer.add(startLevel2);
writer.add(contentLevel2);
writer.add(endLevel2);
writer.add(newLine);
writer.add(indent);
writer.add(endLevel1);
writer.add(newLine);
writer.add(endRoot);
writer.add(newLine);
writer.add(endDocument);
```

Example output file "out.xml":

```
<?xml version="1.0" encoding="UTF-8"?>
<root attr1="example">
    <level1 attr2="abc">
        <level2>Content of level 2</level2>
    </level1>
</root>
```

# DOM Parsing

- ♦ DOM (Document Object Model) is an official standard of the W3C
- ♦ Allows you to build an object representation of your XML data
- ♦ The data is represented as a tree:
  - ♦ The tree resists *completely* in the memory
- ♦ DOM is not restricted to XML data/files

---

# DOM (Dis-)Advantages

- ♦ Advantages

- ♦ It is possible to navigate the tree (back and forth)
- ♦ Nodes can be modified, added, or removed
- ♦ XML documents can be created (serialized)

- ♦ Disadvantages

- ♦ DOM parsing needs huge resources of computer memory and performance
- ♦ → DOM parsing is only possible for relative small amounts of XML data (~ 10 MB)

# Loading an XML File into a DOM Object

```
import javax.xml.parsers.*; //provides DocumentBuilder/-factory
import org.w3c.dom.*; //the Document and many more tools
```

```
DocumentBuilderFactory fac = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = fac.newDocumentBuilder();
Document document = builder.parse(new File("file.xml"));
```

- ♦ **DocumentBuilderFactory:** Defines a factory API that enables applications to obtain a parser that produces DOM object trees from XML documents.
- ♦ **DocumentBuilder:** Defines the API to obtain DOM **Document** instances from an XML document. Using this class, an application programmer can obtain a **Document** from XML.
- ♦ **Document:** The **Document** interface represents the entire HTML or XML document. Conceptually, it is the root of the document tree, and provides the primary access to the document's data.



# Handling an (XML) String as a DOM Object

```
import java.io.StringReader;  
import org.xml.sax.InputSource;  
//rest as above
```

```
//variable "xml" contains the XML data  
String xml = "<root><element>Hello world</element></root>";
```

```
DocumentBuilderFactory fac = DocumentBuilderFactory.newInstance();  
DocumentBuilder builder = fac.newDocumentBuilder();  
StringReader stringReader = new StringReader(xml);  
InputSource inputSource = new InputSource(stringReader);  
Document document = builder.parse(inputSource);
```

# DOM Reading

- ♦ Visit all child **Nodes** of a **Node**:

```
private void visitNode(Node node) {  
    // process node...  
  
    // iterate over all children of a node  
    for (int i = 0; i < node.getChildNodes().getLength(); i++) {  
        // recursively visit all child nodes  
        visitNode(node.getChildNodes().item(i));  
    }  
}
```

- ♦ Go through all **Nodes** in the whole XML document by calling the **visitNode** method with the root node of an XML DOM tree: **visitNode(document.getDocumentElement());**
- ♦ **org.w3c.dom.Node**: The **Node** interface is the primary datatype for the entire Document Object Model. It represents a single node in the document tree and is the base interface for all the other (more specific) nodes such as **Document**, **Element**, **Text**, **Attr**, etc.

# DOM Node Types

- ♦ There is a separate interface for each node type that might occur in an XML document
- ♦ All node types inherit from class **Node**
- ♦ All interfaces are in package **org.w3c.dom**
- ♦ Some of the interfaces:

Interface name	Interface represents
<b>Node</b>	base node interface for handling nodes in an XML document
<b>Document</b>	the document
<b>Element</b>	an element
<b>Attr</b>	an attribute of an element
<b>Text</b>	textual content
<b>CDATASection</b>	CDATA content

---

# DOM Node Types

Retrieve node type with method

**short Node.getNodeType():**

**1:** element node

**2:** attribute node

**3:** text node

**4:** cdata

**8:** comment

...

# Important Methods of the **Node** Object (reading)

Method	Return type	Explanation
<b>getChildNodes</b>	<b>NodeList</b>	A list of all child nodes
<b>getAttributes</b>	<b>NamedNodeMap</b>	The attributes of the node
<b>getNodeName</b>	<b>String</b>	Name of the node
<b>getParentNode</b>	<b>Node</b>	The parent of the node
<b>getNodeType</b>	<b>short</b>	Type of the node
<b>getNodeValue</b>		The value of the node
<b>getElementsByName</b>	<b>NodeList</b>	All nodes of a given name

Iterating over all children of a **Node**:

```
for (int i=0; i< node.getChildNodes().getLength(); i++) {  
    // the actual child:  
    Node aChild = node.getChildNodes().item(i);  
  
    // process child node  
    ...  
}
```

# DOM Nodes & Elements

- ◆ Differences between a **Node** and an **Element**:

The **Element** can query its properties (for example, attributes) by name, while the **Node** has just an anonymous Iterator-view on them.

- ◆ Example 1: Extract root node as a **Node** object and extract attribute **attr1**:

```
Node rootNode = document.getDocumentElement();  
NamedNodeMap nnm = rootNode.getAttributes();  
Node attr1Node = nnm.getNamedItem("attr1");  
String attr1 = ((Attr) attr1Node).getValue();
```

- ◆ Example 2: Extract root node as an **Element** object and extract attribute **attr1**:

```
Element rootElement = document.getDocumentElement();  
String attr1 = rootElement.getAttribute("attr1");
```

---

# DOM Nodes & Elements

- ♦ **org.w3c.dom.NodeList**: The **NodeList** interface provides the abstraction of an ordered collection of **Nodes** (for example: the children of a **Node**).
- ♦ **org.w3c.dom.NamedNodeMap**: Objects implementing the **NamedNodeMap** interface are used to represent collections of **Nodes** that can be accessed by name (for example: attributes).
- ♦ **org.w3c.dom.Element**: The **Element** interface represents an element in an HTML or XML document (inherits from **Node**).

# Reading the Whole Document with DOM

```
private static void visitNode(Node node) {
    if (node.getNodeType() == 1) {
        System.out.print("\n" + node.getNodeName() + ": ");
        NamedNodeMap attributes = node.getAttributes();
        if (attributes != null) {
            for (int i = 0; i < attributes.getLength(); i++) {
                System.out.print(attributes.item(i) + " ");
            }
        }
    }

    if (node.getNodeType() == 3 && !node.getTextContent().trim().isEmpty()) {
        System.out.print "\"" + node.getTextContent().trim() + "\"";
    }

    NodeList nodeList = node.getChildNodes();
    for (int i = 0; i < nodeList.getLength(); i++) {
        visitNode(nodeList.item(i));
    }
}
```

Initial call:

```
visitNode(document.getDocumentElement());
```

Example output:

**root: attr1="example"**

**level1: attr2="abc"**

**level2: "Content of level 2"**

**level1: attr2="def"**

**level2: "Content of second level 2"**



# Important Methods of the **Node** Object (writing)

Method	Return type	Explanation
<b>createElement</b>	<b>Element</b>	Create a new node
<b>createTextNode</b>	<b>Text</b>	Text-content of a node
<b>appendChild</b>	<b>void</b>	Add a child to a node
<b>createAttribute</b>	<b>Attr</b>	Creates an attribute

In general, creating XML trees with DOM is a bottom-up procedure:

1. create a **Node**
2. create the content of the **Node** (text, attributes, ...) and add it to the **Node**
3. add the **Node** to its parent **Node**

# Creating Nodes with DOM

```
DocumentBuilder documentBuilder =
    DocumentBuilderFactory.newInstance().newDocumentBuilder();
Document document = documentBuilder.newDocument();

// First, we create all the necessary elements:

Element root = document.createElement("root");
root.setAttribute("attr1", "example");

Element level1 = document.createElement("level1");
level1.setAttribute("attr2", "abc");

Element level2 = document.createElement("level2");
level2.setTextContent("Content of level 2");

// Appending the children in bottom-up-order:

level1.appendChild(level2);
root.appendChild(level1);
document.appendChild(root);
```

Resulting XML:

```
<root attr1="example">
  <level1 attr2="abc">
    <level2>Content of level 2</level2>
  </level1>
</root>
```

# Modifying Nodes with DOM

- ♦ It is also possible to modify existing elements with DOM.
- ♦ Adding new elements to existing **Nodes**:

```
Element level1 = doc.createElement("level1");  
level1.setAttributes("attr1", "content");  
root.addChild(level1); // "root" already existed
```

- ♦ Existing attributes will be overwritten:

```
// "level1" already had an attribute "attr1"  
level1.setAttributes("attr1", "newattr");
```

# Transformers

- ♦ **Transformers** are generic APIs for processing transformation instructions and performing a transformation from a source (DOM tree) to another output format.
- ♦ Examples: transforming a DOM tree to a String, transforming DOM trees with XSL

# Transforming a DOM Tree to a String

```
import java.io.*;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.OutputKeys;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.dom.DOMSource;

TransformerFactory transformerFactory = TransformerFactory.newInstance();
transformerFactory.setAttribute("indent-number", 4);

Transformer transformer = transformerFactory.newTransformer();
transformer.setOutputProperty(OutputKeys.OMIT_XML_DECLARATION, "no");
transformer.setOutputProperty(OutputKeys.INDENT, "yes");

StringWriter stringWriter = new StringWriter();

StreamResult result = new StreamResult(stringWriter);
DOMSource source = new DOMSource(document);
transformer.transform(source, result);
String xml = writer.toString();

Writer writer = new BufferedWriter(new OutputStreamWriter(new
FileOutputStream(new File("out.xml"))));
writer.write(xml);
writer.close();
```

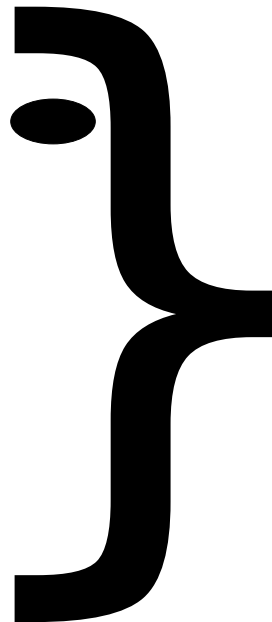
# Which Technology is Better?

- ♦ There is no one answer to that question
- ♦ All three technologies – SAX, StAX, and DOM – have their advantages and disadvantages
- ♦ It always depends on the concrete application which technology to choose
  - ♦ Stream-based processing is e.g. preferred if the documents are huge, but their structure is rather simple
  - ♦ Model-based processing is e.g. preferred if the documents are complex and much navigation (back and forth) is required

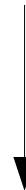
# Combining S(t)AX & DOM

- ◆ Sometimes, huge amounts of XML data are a *serial collection* of trees:

•  
•  
•  
**<sentence>**  
    ...  
**</sentence>**  
  
**<sentence>**  
    ...  
**</sentence>**  
•  
•  
•



Use Sax/StaX to parse the tree and  
extract the individual trees



Parse each individual tree as DOM tree

# Reading Material

- ♦ For this topic of XML processing with Java, there is no single book chapter that you have to read.
- ♦ Please find all information in one of the following books and internet sources or google for further information:
  - ♦ “Java und XML – Grundlagen, Einsatz, Referenz” by Michael Scholz & Stephan Niedermeier (chs. 2.1, 2.3, 2.4.1 + appropriate subchapters of chs. 3, 4, 6)
  - ♦ “Java ist auch eine Insel” by Christian Ullenboom (ch. 13):  
[http://openbook.galileocomputing.de/javainsel/javainsel\\_16\\_001.html#dodtp411227dd-8e3b-4ef7-9be3-33b57be542fe](http://openbook.galileocomputing.de/javainsel/javainsel_16_001.html#dodtp411227dd-8e3b-4ef7-9be3-33b57be542fe)
  - ♦ “Java 7 - Mehr als eine Insel” by Christian Ullenboom (ch. 7):  
[http://openbook.galileocomputing.de/java7/1507\\_07\\_001.html#dodtp4f411983-98d0-4afd-bfac-b60f6ec2991a](http://openbook.galileocomputing.de/java7/1507_07_001.html#dodtp4f411983-98d0-4afd-bfac-b60f6ec2991a)
  - ♦ “Java & XML” by Brett D. McLaughlin & Justin Edelson
  - ♦ <http://www.torsten-horn.de/techdocs/java-xml.htm>