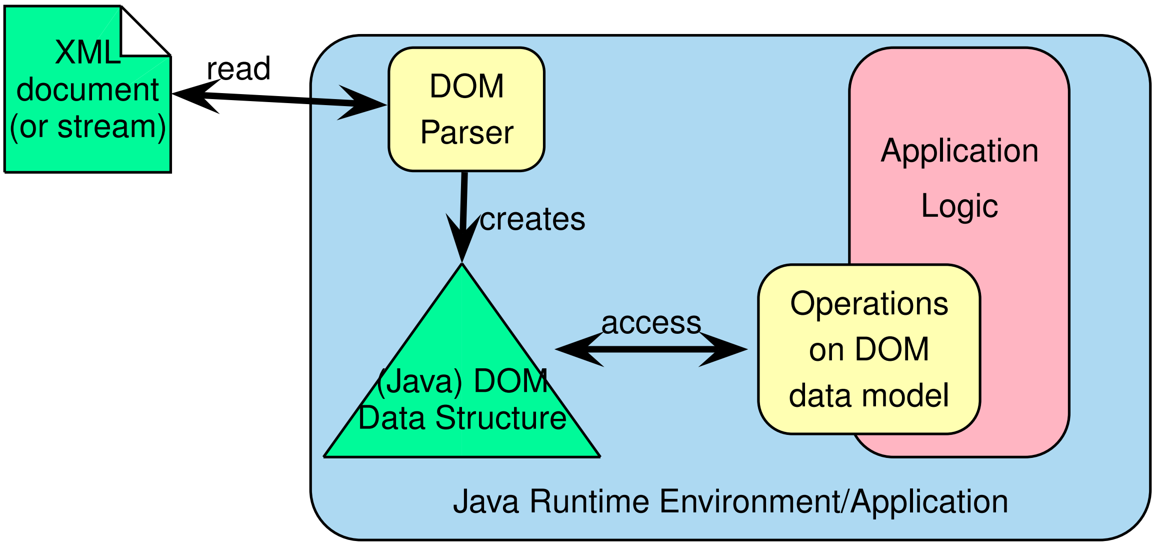
Algorithms and APIs (Cont'd)

- Stream Processing:
  - XML data transfer as sequence of events
  - SAX (Simple Application Interface for XML), StAX (Streaming API for XML)
- XML as Data Exchange Format in Web Services
  - serialize application objects as XML
  - SOAP: generic [not discussed in this course]
  - JAXB: "model-aware" infrastructure
- an intermediate rule-based concept:
  - apache.commons.digester

11.1 DOM

- DOM (Document Object Model) defines a platform- and language-independent object-oriented *interface* (i.e., an *abstract datatype*) for generating, processing and manipulating XML data.


XML document (or stream)
read
DOM Parser
creates
Application Logic
access
(Java) DOM Data Structure
Operations on DOM data model
Java Runtime Environment/Application

DOM

- DOM is a specification of an interface/abstract datatype for the XML data model, *not a* data model and *not a* programming language!
- implementations in Java, C++, etc; usually main-memory-based;
  specialized Java interface definitions:
  - recommended for this course: JDOM2: org.jdom2.*, jdom2.jar,
  - original jdom (=jdom1) deprecated (mainly XPath handling changed; 2013),
  - another alternative: dom4j,
  - not recommended: org.w3c.dom.* (the plain dom is an implementation that exists in nearly all programming languages and does not make use of Java's advantages);
- language base of the DOM specification: OMG-IDL
- Main-memory-based:
  - handling *small* XML fragments for data exchange

DOM: Principles

- only one document in a single DOM instance
- step-by-step-access to the data:
  based on variable assignments in the surrounding imperative/object-oriented programming language and on iterators (cf. proceeding in the network data model):
  - class "Document": represents the complete document,
    - doctype declaration, getRootElement()
  - class "Node": getNodeType(), getChildren(), getFirstChild(), getNextSibling(), getParentNode(), ...
  - class "Element": getName(), getAttributes(), getContent(), ...
  - class "Attribute": getName(), getValue(), ...
  - corresponding methods for generating and changing nodes.
- additionally, XPath and XSLT can be applied to instances of Document and Element;
- based on DOM, XPath and XQuery can be implemented (cf. Apache Xerces (XML/DOM)/Xalan (C++/Java; XPath 1.0/XSLT 1.0 [in 2016])
- XPath/XSLT often inefficient (no indexes, query optimization), restricted functionality

JDOM – sample code fragment

```
// apt-get install libjdom2-java; add jdom2.jar to the classpath
import java.io.File;
import java.util.List;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.input.SAXBuilder;

public class MondialJDOMSimple {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("../mondial.xml"));
      Element mondial = doc.getRootElement();
      List<Element> countries = mondial.getChildren("country");
      String firstcode = countries.get(0).getAttributeValue("car_code");
      System.out.println(firstcode);
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/JDOM/MondialJDOMSimple.java]

- SAXBuilder can be set on any input stream.
- XMLOutputter([Format])/SAXOutputter

JDOM2: XPath

- similar to JDBC/SQLJ statement concept for SQL in Java:
- Basic: compile Strings into XPath expressions,
  evaluate to Object or List<Object>,
- Optional: add (type) Filter for result node type,
- Advanced: XPath expression contains variables
  - must be declared as a map of variables (optionally with preset values)
  - call then requires also namespace blabla, see doc,
- XPath handling changed severely from jdom to jdom2 (2013).
- id(), number(), and thus also max(), sum() etc. not supported.

JDOM2: XPath example

```
import java.io.File;
import java.util.List;
import java.util.Map;
import java.util.HashMap;
import org.jdom2.Document;
import org.jdom2.Element;
import org.jdom2.Attribute;
import org.jdom2.Namespace;
import org.jdom2.input.SAXBuilder;
import org.jdom2.xpath.XPathFactory;
import org.jdom2.xpath.XPathExpression;
import org.jdom2.filter.Filters;

public class MondialJDOMXPath {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("../mondial.xml"));
      Element mondial = doc.getRootElement();
      XPathFactory xpf = XPathFactory.instance();
```

```
      XPathExpression xpath = xpf.compile("//country[@car_code='R']/@area");
      Attribute a = (Attribute) xpath.evaluateFirst(doc);     // -> casting
      int area = Integer.valueOf(a.getValue());
      System.out.println(area);

      xpath = xpf.compile("//country[@area > 7000000]/name");
      List<Object> names = xpath.evaluate(doc);   //  <Object> -> casting
      for (Object n : names) System.out.println(((Element) n).getTextTrim());

      Map<String,Object> vars = new HashMap<String,Object>();
      vars.put("code","D");
      XPathExpression<Element>   // due to filter: result type guaranteed
        xp2 = xpf.compile("//country[@car_code=$code]/population[last()]",
                          Filters.element(), vars, (Namespace[]) null);
      Element res = xp2.evaluateFirst(doc);  // no casting necessary
      int pop = Integer.valueOf(((Element)res).getTextTrim());
      System.out.println(pop);
      xp2.setVariable("code", "F");
      res = (Element) xp2.evaluateFirst(doc);
      pop = Integer.valueOf(((Element)res).getTextTrim());
      System.out.println(pop);
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/JDOM/MondialJDOMXPath.java]

Deprecated: JDOM1 – sample code fragment: XPath

```
public Element getCity(Element country, String provname, String cityname)
{
  Element city = null;
  XPath xpath;
  try
  {  if (provname != null) {
        xpath = XPath.newInstance("./province[name=$pn]/city[name=$cn]");
        xpath.setVariable("pn", provname); }
     else
        xpath = XPath.newInstance(".//city[name=$cn]");
     xpath.setVariable("cn", cityname);
     // xpath.addNamespace(... an instance of Namespace ...);
     city = (Element) xpath.selectSingleNode(country);
  }
  catch (Exception e) {...}
  return city;
}
```

JDOM and XSLT

XSLT: $xsl(xml) \times xml \to xml$

- Simple: class org.jdom2.transform.XSLTransformer (XSLT 1.0 only?)
- More complex: JAXP TrAX classes
  - JDOMSource/JDOMResult as generic interfaces to XML representations
- saxon can/could also be combined with JDOM2 (via a JDOM wrapper class provided by Saxon)

JDOM Output etc

- XMLOutputter
- predefined formats available
- for adding a DTD reference:
  document.setDocType(new DocType("*elementname*", "*dtdfilename*"));

```
import java.io.File;
import org.jdom2.input.SAXBuilder;
import org.jdom2.Document;
import org.jdom2.transform.XSLTransformer;
import org.jdom2.output.XMLOutputter;
import org.jdom2.output.Format;

public class MondialJDOMXSL {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("mondial.xml"));
      XSLTransformer xsltr = new XSLTransformer("../../XSLT/mondial-simple.xsl");
      Document result = xsltr.transform(doc);
      Format fmt = Format.getPrettyFormat();
      fmt.setLineSeparator(System.getProperty("line.separator"));
        // default would be DOS/windows style \r\n
      XMLOutputter outputter = new XMLOutputter(fmt);
      outputter.output(result, System.out);
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/JDOM/MondialJDOMXSL.java]

```
import java.io.File;
import org.jdom2.input.SAXBuilder;
import org.jdom2.Document;
import org.jdom2.transform.JDOMSource;
import javax.xml.transform.*;
import javax.xml.transform.Transformer;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.stream.StreamResult;
import javax.xml.transform.stream.StreamSource;

public class MondialJDOMXSL2 {
  public static void main(String[] args) {
  try {
      SAXBuilder builder = new SAXBuilder();
      Document doc = (Document) builder.build(new File("mondial.xml"));
      TransformerFactory factory = TransformerFactory.newInstance();
      Source xsltSource = new StreamSource("../../XSLT/mondial-simple.xsl");
      Transformer transformer = factory.newTransformer(xsltSource);
      StreamResult result = new StreamResult("bla.xml");
      JDOMSource src = new JDOMSource(doc);
      transformer.transform(src, result);
      } catch (Exception e) { e.printStackTrace(); }
} }
```

[Filename: java/JDOM/MondialJDOMXSL2.java]

11.2 HTML vs XHTML in Java

- HTML/XHTML has a DTD that restricts the element types, nesting, and attributes,
- HTML applications know+use a fixed DTD,
- sloppyness/tolerance can exploit that the DTD is known, allow to omit “obvious” things.
- HTML *usage* is less strict than XHTML (cf. parser exercise to the lecture)
  - mainly concerned: HTML parsers in the browsers that should accept handwritten dirty/sloppy HTML.
  - data model tree: intermediate levels (e.g. table: thead, tbody) may be missing browsers “know” how to render it (best-effort)
  - parsing the input character stream:
    - closing tags may be missing, e.g.,
      <tr><td>bla <td>blubb </tr>
    - empty elements like <br> instead of <br/> (HTML tools know that br-elements are empty)
- XML tools must not accept any sloppyness

HTML to XHTML

- tidy: `https://www.html-tidy.org/`
  console application (→ can be used in shell scripts) C library libtidy
- JSoup: Java HTML DOM Parser and API: `https://jsoup.org/`
  - uses an own (HTML) DOM implementation
    ```
    Document doc = Jsoup.parse(string s; [baseUri for hrefs]);
    Document doc = Jsoup.connect("http://en.wikipedia.org/").get();
    Document doc = Jsoup.parse(File in, String charsetName, String baseUri);
    ```
  - stepwise navigation in the DOM
  - allows simple XPath style selects
    ```
    doc.select("selector pattern");
    ```
  - set/modify content, attributes and text contents
  - serialization: toString() (obviously!)
  - document properties (DTD, output mode, ...) can be set:
    charset, syntax (html/xml), indentAmount, escapeMode (how to handle HTML entities), e.g.
    document.outputSettings().syntax(Document.OutputSettings.Syntax.xml);
  - note: JSoup materializes the whole document tree

JSoup example

```
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import java.io.*;
public class JSoupTest {
  public static void main(String[] args) {
    try {
      Document doc = Jsoup.connect("http://en.wikipedia.org/wiki/Ball's_Pyramid").get();
      // File input = new File("bla.html");
      // Document doc = Jsoup.parse(input, "iso-8859-1", "");
      doc.outputSettings().syntax(Document.OutputSettings.Syntax.xml);
      String xml = doc.toString();
      BufferedWriter writer = new BufferedWriter(new FileWriter("jsoupoutput.xml"));
      writer.write(xml);
      writer.close();
      } catch (Exception e) { e.printStackTrace(); }
} }
```

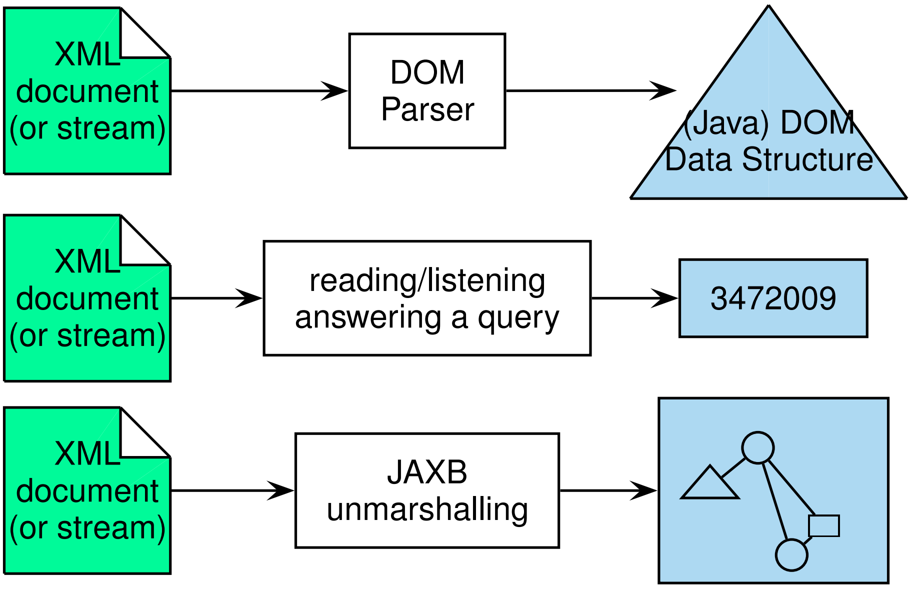
[Filename: java/JSoupTest.java]

11.3 XQJ: XQuery API for Java

- similar to JDBC for “SQL in Java”
- access to a remote XML DB: open a connection, send an XQuery query, receive an iterable set of results.
- Tutorial: `https://www.progress.com/tutorials/xquery/api-for-java-xqj`
- saxonEE (commercial version) implements XQJ (although it’s not a remote XML DB server then)

11.4 XML Stream Processing

- reading from a file or from an HTTP connection both is actually reading char by char from a stream
- the stream is parsed, resulting in something that can be used by application:



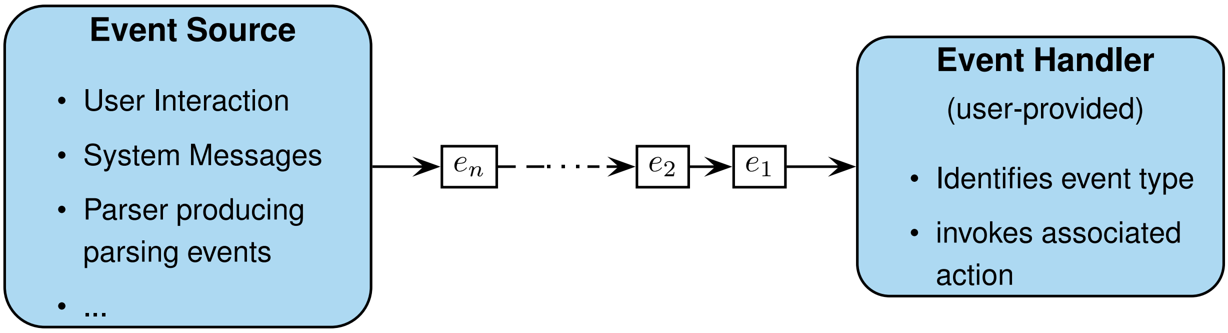
Parsing: General Concepts

- Compiler Construction in general:
  1. input: a sequence of characters
  2. lexical analysis ("lexer") extracts the keywords (e.g. “begin”, “end”, “for”) and outputs a sequence of *tokens*
  3. syntactical (grammar) analysis: check grammatical structure and generate the *parse tree* (e.g. via automaton)
  4. tools: lex & yacc/bison
  5. interpreter, optimizer, compiler, visualizer etc. process the parse tree
- XML:
  1. lexical: split unicode input sequence into opening tags, closing tags, attributes, PCDATA, processing instructions, etc.
  2. syntactical and structural: is it well-formed XML? (if not: “debug” it?)
  3. processing: build DOM tree, build JAXB structure, visualize, . . .
- the above DOM and JAXB are actually parser+specific processing

⇒ XML Stream Processing: works on the tokens sequence!

Event-Based Processing as a *General Design Pattern*

- A stream of (high-level) items that carry some inherent semantics can be seen as a stream of “events”
  (in contrast to a simple 0-1-stream, a byte stream or similar low-level streams)



Event Source

- User Interaction
- System Messages
- Parser producing parsing events
- ...

Event Handler

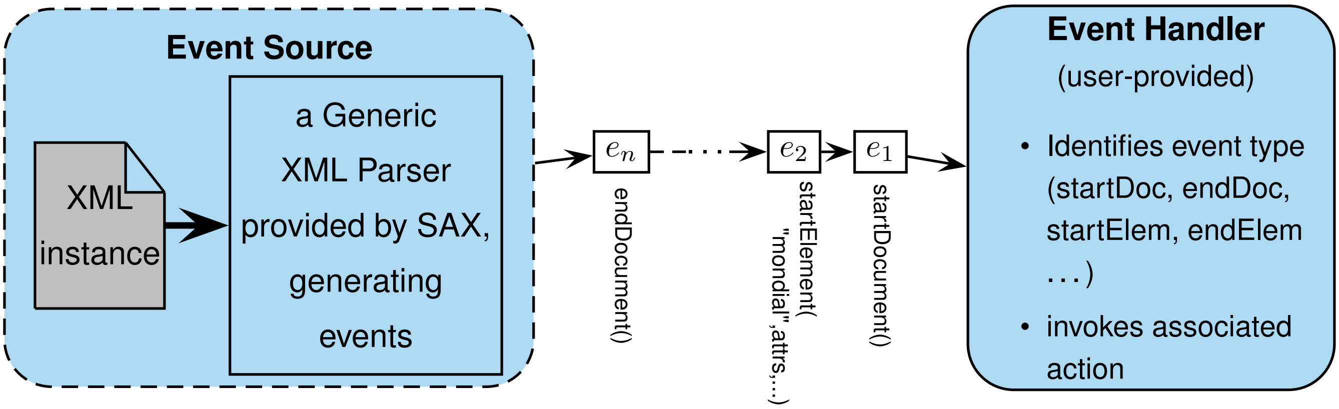
(user-provided)

- Identifies event type
- invokes associated action

- The application programmer provides the Event Handler implementation, containing actions for each type of event;
- kind of *rule-based*;
- programmer is *not* in charge of the control flow

11.5 Event-based XML Parsing with SAX

- SAX (“The Simple API for XML”) is an *event-based interface/model*



Event Source

a Generic XML Parser provided by SAX, generating events

Event Handler

(user-provided)

- Identifies event type (startDoc, endDoc, startElem, endElem . . .)
- invokes associated action

Represents/processes an XML document as a sequence of events (depth-first traversal), e.g.

- startDocument(), endDocument()
- startElement(Name, attributesList) – attributes not split
- endElement(Name)
- characters(string)

XML Parsing with SAX

SAX: parse XML tokens from a character stream (e.g., a file)

- import classes: javax.xml.parsers.*, org.xml.sax.*
- a generic XML Parser is parameterized with a *Content Handler* (plus *Error Handler*, *DTD Handler*, and *EntityResolver*) implementation.
- The most trivial Content Handler is the *DefaultHandler* that does nothing: the document is parsed, events are detected, but no action is performed (DTD / XML Schema validation can be switched on).
- Event handler programmed wrt. a “*push* API”.
- Normally, the user-provided Content Handler extends the DefaultHandler, overwriting (some of) its event methods.
- With the content handler implementation, the user provides “actions” in form of Java code, associated with specific events (and even dependent on context information).
- If during parsing of the XML document, a specific event occurs, the code of the associated action from the content handler is invoked (*“callback”*).

SAX (and StAX): Applications

Only events are signaled: linear processing based on incoming sequence of events (=tokens).

- ... among many other things, one can generate a DOM tree structure,
- validation according to a DTD (using an automaton as given on Slide 176) in linear time,
  - SAX (and StAX) are only on an intermediate level between character streams and XML
    - ⋆ have lower requirements on input syntax and structure
    - ⋆ input must be allowed XML tokens incl. text
    - ⋆ closing tags may be missing, entities may be undeclared

    ⇒ allows e.g. for fault-tolerant parsing of sloppy HTML.
- stream-processing of XML input
  - start processing already when input document is not yet complete,
  - filtering for elements that are relevant for a given application,
  - linear search for something, e.g., names of countries,
  - stop evaluation when finished before reading the whole document.
- if necessary: application needs to maintain context.

SAX Example Code

Consider a very simple application that

- detects all elements with attributes,
- for each element, output the element's name,
- for each element, output the name-value pairs of its attributes,
- end the evaluation when "Göttingen" is found.

```
element: country
 - attribute: 'car_code' value: 'AL' type: 'ID'
 - attribute: 'area' value: '28750' type: 'CDATA'
 - attribute: 'capital' value: 'cty-cid-cia-Albania-Tirane' type: 'IDREF'
 - attribute: 'memberships' value: 'org-BSEC org-CE org-CCC org-ECE org-EU ...' type: 'IDREFS'
element: encompassed
 - attribute: 'continent' value: 'europe' type: 'IDREF'
 - attribute: 'percentage' value: '100' type: 'CDATA'
element: ethnicgroups
 - attribute: 'percentage' value: '3' type: 'CDATA'
element: ethnicgroups
 - attribute: 'percentage' value: '95' type: 'CDATA'

... Göttingen found - ready.
```

```
import java.io.IOException;
import javax.xml.parsers.*;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
public class PrintAttributes {

 public static void main(String[] args) {
  DefaultHandler handler = new ContentHandlerPrintAttributes();
  SAXParserFactory factory = SAXParserFactory.newInstance();
  try {
     SAXParser parser = factory.newSAXParser();
     parser.parse("file:../mondial.xml", handler);  // <<<<<<<<< and now it runs
  } catch (ContentHandlerPrintAttributes.MySAXTerminatorException e) {
    // exception for smoothly stopping the processing:   <<<<<<<<<<<<<<<
    System.out.println(" ... Göttingen found - ready."); }
   catch (IOException e1) { e1.printStackTrace(); }
   catch (ParserConfigurationException e) { e.printStackTrace(); }
   catch (SAXException e) { e.printStackTrace(); }
}  }
```

[Filename: java/SAX/PrintAttributes.java]

```
import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;

public class ContentHandlerPrintAttributes extends DefaultHandler {
    // react on opening elements:
    public void startElement(String url, String localName, String qName,
            Attributes attrs) throws SAXException {
        if (attrs.getLength() > 0) {
            String elementName;
            if (qName == null || qName.equals("")) elementName = localName;
            else elementName = qName;
            System.out.println("element: " + elementName);
            for (int i = 0; i < attrs.getLength(); i++) {
                System.out.println(" - attribute: '" + attrs.getQName(i)
                        + "' value: '" + attrs.getValue(i) + "' type: '"
                        + attrs.getType(i)+"'");
            }
            System.out.println();
    }   }
    // methods for endElement(), startDocument(), endDocument() omitted

    // continue next page
```

```
    // continue next page


    public void characters(char[] text, int from, int length)
       throws SAXException  {
       // stop evaluation by throwing an exception:
       String textString = (new String(text)).substring(from, from + length);
       if (textString.contains("Göttingen"))
          throw new MySAXTerminatorException();
    }
    // the exception stub:
    public class MySAXTerminatorException extends SAXException { ; }
}
```

[Filename: java/SAX/ContentHandlerPrintAttributes.java]

- evaluation can only be stopped by raising an exception;
- the events are on the level of structural XML parsing,
  an XML element/subtree consists of several (often: many) events.
- all PCDATA/CDATA values are strings
  → numeric computations require conversion to Java literals or class instances.

SAX: Applications to XPath Query Answering

Forward queries

XPath-queries like //country[@car_code='D']/population[last()] can be answered very (time- and memory-)efficient,

- use the sequence of events (linear)
- maintain some context (often LOGSPACE/additional LOGTIME sufficient)

... works only for queries, that contain only forward steps,

General queries

which XPath expressions can be *transformed* in equivalent forward-expressions (and with what efforts)?

- "XPath: Looking forward"; F. Bry et al ; 2002; LMU München
- theory: complexity, connections to linear temporal logic
  For every linear temporal logic formula that uses past and future operators, there is an equivalent formula that uses only future operators
  ... but in general of exponential size.

11.6 XML Streams/StAX - The Streaming API for XML

Higher abstraction level (than character-based XML) for XML data exchange:
javax.xml.stream (rt.jar)

Reconsider SAX

- on-the fly processing, no in-memory representation for good performance
- idea of "XML Event Stream": a char stream (File, HTTP) can be converted into an XML Event Stream by an XML parser; see example's main() method.
- SAX does not make the XML Event Stream accessible, but only via calls of methods of the Event Handler.

Generalization and Abstraction: XML Streams

- XMLEvent types: StartDocument, DTD, StartElement, Characters, EndElement, EndDocument, (Attribute), (Namespace) . . .
- XMLStreamReader, XMLStreamWriter,
  XMLEventReader, XMLEventWriter,
- XML Streams also can be connected *directly* as an *abstract* means to exchange XML

XML Streams: Application Scenarios

- READ: usage analogous to SAX: process an XML file input as an XML Event Input Stream:
  control flow is not passed to the parser (**unlike** *SAX*), but XML events are accessed using an *iterator*, controlled by the Java program using the StAX API (*Pull-API*).
  [Note: iterators are a common design pattern, not only applied to collections, but as we see here also to streams: init(), next(), ...]
  ⇒ application code: same as for SAX, only operational embedding done differently.
- WRITE and READ: streamed data exchange between processed on the XML level.
- Two variants exist:
  - XMLStreamReader, XMLStreamWriter ("Cursor")
  - XMLEventReader, XMLEventWriter ("Iterator")
- XML-S/E-Readers/Writers can be put on any input/output stream (FileInput/OutputStream, BufferedInput/OutputStream, System.out, HTTP stuff (see Web Services) or directly connected to each other:
  XMLS/EWriter->PipedOutputStream->PipedInputStream->XMLS/EReader of the next application)

INTERFACES XMLSTREAMREADER, XMLSTREAMWRITER

XMLStreamReader

- int eventtype = r.next() and then switch based on eventtype
  javax.xml.stream.XMLStreamConstants.XX:
  START_DOCUMENT, START_ELEMENT, CHARACTERS, END_ELEMENT, . . .
- access methods when on START_DOCUMENT: getEncoding() etc.
- goal-driven access methods on the reader when on START_ELEMENT:
  r.getLocalName(), r.getAttributeValue(*name*),
  r.getAttributeCount(), getAttributeLocalName(*n*), getAttributeValue(*n*) for iteration,
  r.getElementText() (reads also the next EndElement from the stream!),
  getName() (as qname),
  namespace handling: getPrefix(), getNamespaceURI() (default NS),
  getNamespaceURI(prefix),
- goal-driven access method when on CHARACTERS: r.getText(), r.isWhiteSpace();
- goal-driven access methods on the reader when on END_ELEMENT:
  r.getLocalName() + namespace handling
- note again: all PCDATA/CDATA values are strings.

XMLStreamWriter

- w.writeStartDocument()
- w.writeStartElement(*name*),
- w.writeEmptyElement(*name*),
  Note: there is writeEmptyElement(*name*), although for the Reader, there is no event type EMPTY_ELEMENT; instead also for empty Elements, START_ELEMENT and END_ELEMENT are separately read
  ⇒ copying straightly to output will create an non-empty element with “”-content!
- w.writeAttribute(*name*, *value*), (and all three also with namespace handling)
- w.writeCharacters(*text*);
- w.writeEndElement(): closes the innermost open element;
- w.writeEndDocument(): closes all open elements.
- w.flush(): force write any data to the underlying output mechanism.

StAX StreamReader Example

```
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;
import java.io.FileInputStream;

public class StAXPrintAttributes {
  public static void main(String[] args) {

  try {
    FileInputStream inputStream = new FileInputStream("../mondial.xml");
    XMLInputFactory inputFactory = XMLInputFactory.newInstance();
    XMLStreamReader parser = inputFactory.createXMLStreamReader(inputStream);
    boolean goOn = true;

    while (goOn) {
      int eventtype = parser.next();
      switch(eventtype) {
        case XMLStreamConstants.END_DOCUMENT:
          goOn = false;
          break;   // << break after each case!

        // continue next page
        // continue next page

        case XMLStreamConstants.START_ELEMENT:
          if (parser.getAttributeCount() > 0) {
            System.out.println(parser.getLocalName());
            for (int i = 0; i < parser.getAttributeCount(); i++) {
           System.out.println(" - attribute: '" + parser.getAttributeLocalName(i)
                                  + "' value: '" + parser.getAttributeValue(i)
                           + "' type: '"  + parser.getAttributeType(i));
            }
          }
          break;
        // cases for endElement(), startDocument(), endDocument() omitted
        case XMLStreamConstants.CHARACTERS:
          String textString = parser.getText();
          if (textString.contains("Göttingen"))
               goOn = false;
      }
    }
    System.out.println(" ... Göttingen found - ready.");
    parser.close();
  } catch (Exception e) { e.printStackTrace(); }
}}
```

[Filename: java/StAX/StAXPrintAttributes.java]

XMLEventReader/XMLEventWriter

- above: XML*Stream*Reader/Writer:
  - XML-parsing level “events” like in SAX
  - the reader is the central object (`r.next()`→`int, r.getLocalName(), ...`)
- alternative: XML*Event*Reader/Writer:
  - consider (empty or CDATA) XML Elements as *events* on the application level,
  - XMLEventReader as an *Iterator* over a sequence of events (actually, `XMLEventReader extends Iterator { ...}`), applicable to pure XML files, but also to incoming HTTP-XML streams (→ Web Services)
    - hasNext() → boolean: check if there are more events.
    - nextEvent() → XMLEvent: get the next XMLEvent
    - getElementText() → String: reads the content of a text-only element.
    - nextTag() → XMLEvent: skips any insignificant space events until a START_ELEMENT or END_ELEMENT is reached. [what about CHARACTERS?]
    - peek()→ XMLEvent: check the next XMLEvent without reading it from the stream.

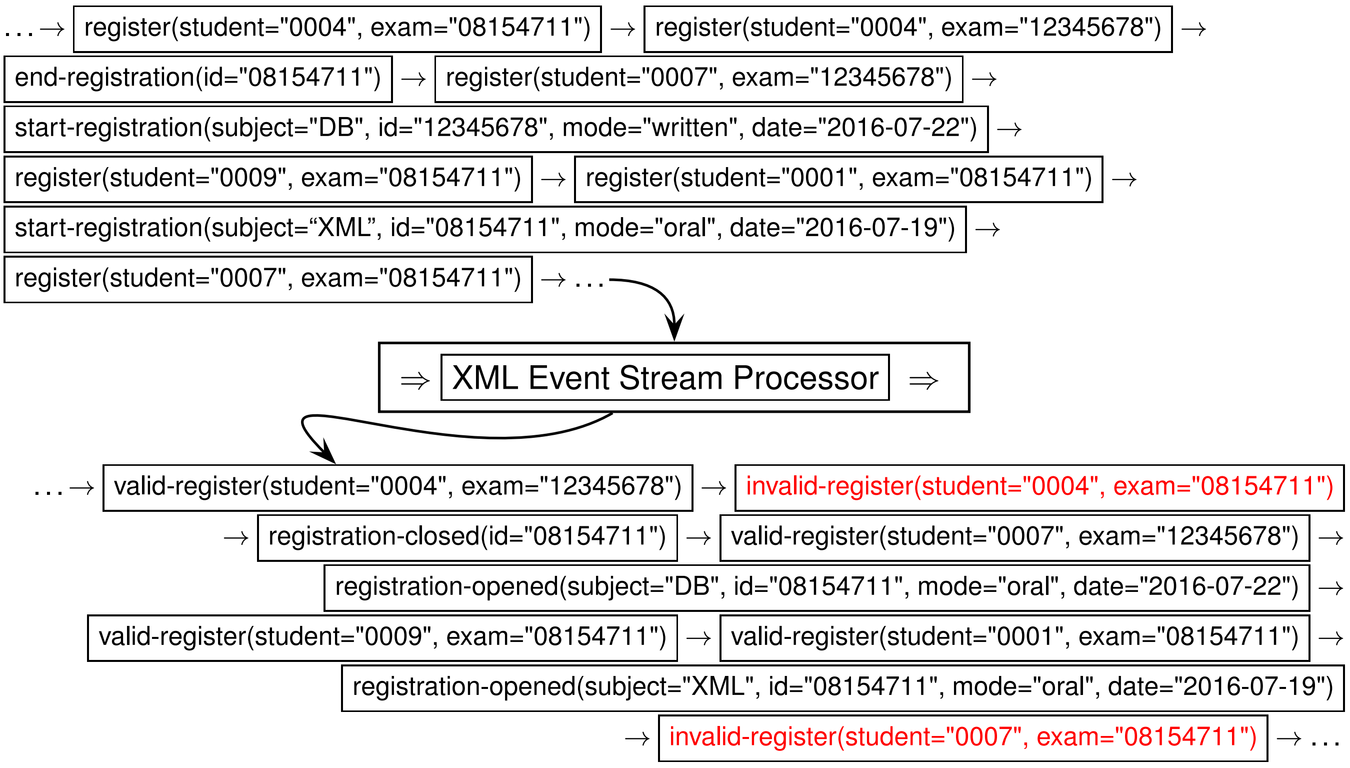
StAX Event Example: Exam Registration

Assume the administration of exams in a student's office ("Prüfungsamt"):

- The *subject* (e.g., "Semi-Structured Data and XML") and ID of lectures/exams,
- whether the exam is *written* or *oral*,
- for written exams, the date of the exam,
- for oral exams, a number of dates is given when the single exams are held.
- the registration period *starts* when receiving an incoming XML message `start-registration`
- the registration period *ends* when receiving an incoming XML message `end-registration`
- for all students that did (`register`) correctly, the student's relevant details are extracted and written to an XMLOutputStream stream (`valid-register`; in the example, we pipe it to `stdout`.)
- students that register before beginning or after the end of registration, are not accounted for the exam; an error message/event `invalid-register` goes to the XMLOutputStream.

StAX Example: Exam Registration

- the program should allow the management of registrations for multiple exams at one time (all incoming over the same continuous input stream).



... → register(student="0004", exam="08154711") → register(student="0004", exam="12345678") →
end-registration(id="08154711") → register(student="0007", exam="12345678") →
start-registration(subject="DB", id="12345678", mode="written", date="2016-07-22") →
register(student="0009", exam="08154711") → register(student="0001", exam="08154711") →
start-registration(subject="XML", id="08154711", mode="oral", date="2016-07-19") →
register(student="0007", exam="08154711") → ...

⇒ XML Event Stream Processor ⇒

... → valid-register(student="0004", exam="12345678") → invalid-register(student="0004", exam="08154711")
→ registration-closed(id="08154711") → valid-register(student="0007", exam="12345678") →
registration-opened(subject="DB", id="08154711", mode="oral", date="2016-07-22") →
valid-register(student="0009", exam="08154711") → valid-register(student="0001", exam="08154711") →
registration-opened(subject="XML", id="08154711", mode="oral", date="2016-07-19")
→ invalid-register(student="0007", exam="08154711") → ...

StAX Example Cont'd:

Consider the following XML sequence as input stream:

```
<?xml version="1.0" encoding="UTF-8"?>
<stream>
  <register student="0007" exam="08154711"/>
  <start-registration id="08154711" subject="Semistructured Data and XML"
     date="2016-07-19" mode="oral"/>
  <register student="0001" exam="08154711"/>
  <register student="0009" exam="08154711"/>
  <start-registration id="12345678" subject="Databases"
     date="2016-07-22" mode="written"/>
  <register student="0007" exam="12345678"/>
  <end-registration id="08154711"/>
  <register student="0004" exam="08154711"/>
  <register student="0004" exam="12345678"/>
</stream>
```

[Filename: java/StAX/exams.xml]

StAX Example Cont'd:

Code for the `Exam` bean, containing the exam's properties and some constants):

```
import java.util.Date;
import java.text.SimpleDateFormat;
import javax.xml.namespace.QName;
import javax.xml.stream.events.StartElement;

public class Exam {
  private String id;     private String subject;
  private String date;   private boolean oral;
  private boolean registeringClosed = false;
  private String startOfReg;   private String endOfReg;

  public Exam(StartElement ev) {  // the <start-registration> element "event"
    this.id = ev.getAttributeByName(new QName("id")).getValue();
    this.subject = ev.getAttributeByName(new QName("subject")).getValue();
    this.oral = "oral".equals(ev.getAttributeByName(new QName("mode")).getValue());
    this.date = ev.getAttributeByName(new QName("date")).getValue();
    this.setStartOfReg(getTodayDate());
  }

  // continue next page
  public String getId() { return id; }
  public String getDate() { return date; }
  public String getSubject() { return subject; }
  public boolean isOral() { return oral; }
  public boolean isWritten() { return (!oral); }
  public String getMode() {  if (oral) return "oral"; else return "written"; }
  public boolean isRegisteringClosed() { return registeringClosed; }
  public void setRegisteringClosed(boolean registeringClosed) {
    this.registeringClosed = registeringClosed;  }
  public String getEndOfReg() { return endOfReg; }
  public String getStartOfReg() {  return startOfReg; }
  public void setStartOfReg(String startOfReg) { this.startOfReg = startOfReg;  }
  public void setEndOfReg(String endOfReg) { this.endOfReg = endOfReg; }

  public static String getTodayDate() {
    return new SimpleDateFormat().format(new Date());   }
/*  private String getTodayDate() {
    DateFormat format = new SimpleDateFormat().;
    return format.format(new Date());   }
*/
}
```

[Filename: java/StAX/Exam.java]

StAX Example Cont'd:

Code for the main parser class, containing the `main` method:

```
import java.io.File;           import java.io.FileInputStream;
import java.io.OutputStream;
import java.text.DateFormat;   import java.text.SimpleDateFormat;
import java.util.HashMap;      import java.util.Map;
import java.util.Iterator;

import javax.xml.namespace.QName;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.events.XMLEvent;
import javax.xml.stream.XMLEventFactory;
import javax.xml.stream.events.StartElement;
import javax.xml.stream.events.EndElement;
import javax.xml.stream.events.Attribute;
import javax.xml.stream.events.Characters;
import javax.xml.stream.XMLEventReader;
import javax.xml.stream.XMLEventWriter;

// continue next page
```

```
import javax.xml.stream.XMLEventWriter;

// continue next page

public class ExamStreamParser {
  public static void main(String[] args) {
  try{
    FileInputStream in = new FileInputStream("exams.xml");
    OutputStream out = System.out;
    XMLInputFactory inputFactory = XMLInputFactory.newInstance();
    XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
    XMLEventReader parser = inputFactory.createXMLEventReader(in);
    XMLEventWriter writer = outputFactory.createXMLEventWriter(out);
    XMLEventFactory eventFactory = XMLEventFactory.newInstance();
    Map<String,Exam> exams = new HashMap<String,Exam>();
    boolean goOn = true;

    while (goOn) {
      XMLEvent event = parser.nextEvent();
      int eventtype = event.getEventType();
      switch(eventtype) {




      // continue next page
```

```
      // continue next page

        case XMLStreamConstants.START_ELEMENT:
          StartElement ev = (StartElement)(event.asStartElement());
          if("start-registration".equals(ev.getName().getLocalPart())) {
            Exam exam = new Exam(ev);
            exams.put(exam.getId(), exam);
            Iterator<Attribute> attrs = ev.getAttributes();
            StartElement se = eventFactory.createStartElement("", null, "registration-opened",
            writer.add(se);
            writer.add(eventFactory.createEndElement("", null, "registration-opened"));
            writer.add(eventFactory.createCharacters("\n"));
          }
          else if("end-registration".equals(ev.getName().getLocalPart())) {
              String examId = ev.getAttributeByName(new QName("id")).getValue();
            Exam exam = exams.get(examId);
            if(exam == null) {
              System.err.println("no such exam with id '"+examId+"' open for registration!");
              break;
              }
            exam.setEndOfReg(Exam.getTodayDate());
            exam.setRegisteringClosed(true);
            }
          // continue next page
```

```
     // continue next page

     else if("register".equals(ev.getName().getLocalPart())) {
      String studentId = ev.getAttributeByName(new QName("student")).getValue();
      String examId = ev.getAttributeByName(new QName("exam")).getValue();
      if(exams.containsKey(examId)) {
       Exam exam = exams.get(examId);
       if(! exam.isRegisteringClosed()) {
        StartElement se = eventFactory.createStartElement(
          "", null, "valid-register", ev.getAttributes(), null);
        writer.add(se);
        writer.add(eventFactory.createEndElement("", null, "valid-register"));
        writer.add(eventFactory.createCharacters("\n"));
       }
       else {  //  exam.isRegisteringClosed()
        StartElement se = eventFactory.createStartElement(
          "", null, "invalid-register", ev.getAttributes(), null);
        writer.add(se);
        writer.add(eventFactory.createCharacters("(reg. ended on " + exam.getEndOfReg
        writer.add(eventFactory.createEndElement(
         "", null, "invalid-register"));
        writer.add(eventFactory.createCharacters("\n"));
       }
      }
      // continue next page
```

```
      // continue next page

      else {  // not (exams.containsKey(examId))
       StartElement se = eventFactory.createStartElement(
          "", null, "invalid-register", ev.getAttributes(), null);
       writer.add(se);
       writer.add(eventFactory.createCharacters("reg. for exam '"+examId+"' not open"))
       writer.add(eventFactory.createEndElement("", null, "invalid-register"));
       writer.add(eventFactory.createCharacters("\n"));
      }
     }
     break;
    case XMLStreamConstants.END_DOCUMENT:
     parser.close();
     writer.flush();
     writer.close();
     goOn = false;
     break;
  }
 }} catch (Exception e) { e.printStackTrace(); }
 }
}
```

[Filename: java/StAX/ExamStreamParser.java]

Some comments on XMLEventReader/Writer

- XMLEventReader/Writer
  - no simple getLocalName()/getAttributeByName(), but only via qnames or getAttributes() as Iterator<Attribute>.
    Note: real event-based applications usually use namespaces.
  - no getAttributeValue(. . .), but only via getAttribute(...).getValue().
  - generates instances of Event class
    - memory-intensive, garbage-collector-intensive
    - instances can be given away to threads for processing
  - For output, also event instances have to be created (use EventFactory).
  - No EmptyElement class - neither for Reader nor Writer.
  - EndElement explicitly needs element name again.

Some notes for both XMLStreamReader and XMLEventReader

- Only XMLStreamWriter has a notion of empty elements:
  - XMLStream/EventReader: empty elements also have an EndElement event;
  - XMLEventWriter: empty elements require to write an explicit EndElement!
- The accessors to attributes differ between XMLStreamReader on START_ELEMENT and XMLEventReader→StartElement.

StAX Comparison with SAX

SAX:
- “Push” API
- feeds a single event handler that is given “away” to the parser
- Common pattern: methods for each event type, where startElement() and endElement() contain large `ifs`.

StAX: ... next page ...

StAX Comparison with SAX (Cont'd)

StAX:
- "Pull" API
- Common pattern: huge `switch` command whose cases again contain large `ifs`.
- the design as a "pull-interface" where the user has control allows to use Reader.next()/Reader.nextEvent() whenever the programmer wants it:
  - in the "case"-code for StartElement, one can call next() to read the text content immediately for further processing. This saves some booleans.
- Modularization:
  the while(...){... parser.next() ...} loop is parameterized with the parser. For "subtasks", other procedures can be called that have specific implementation of the cases:

```
case XMLStreamConstants.START_ELEMENT:
  if parser.getLocalName.equals("country") {
    Country country = processCountry(parser); // run another while-loop
    // (that first must process the attributes of the START_ELEMENT)
    mondial.add(country);  }
```

  Note that such methods can be wrapped as constructor calls like

```
 prov = new Province(parser, country)
```

  to parse a province into an object of class Province that belongs to an object of class Country. (e.g. to implement JAXB; see Slide 545 ff.)

StAX Comparison with SAX (Cont'd)

- Performance: no difference.
  The underlying XMLStream is the same.
- both can easily produce XML output via `XMLStreamWriter/XMLEventWriter` (e.g. to another SAX/StAX appl.)
- The actual code to be written is not much different in both cases.
- SAX maps a unicode input stream directly to the EventHandler calls.
- StAX makes the intermediate abstraction level of XML event streams accessible.

SAX and StAX: Applications

Stream-based processing can be applied to XML data on multiple levels:

- low-level applications:
  SAX is often used for building a DOM from Unicode XML input: “opening tag with attributes”, “text”, “closing tag” can immediately be translated into the DOM constructors.
- low-level streaming of an XML instance:
  answering XPath (forward-axes only) queries; optionally maintaining some context (e.g., stack).
- higher level “application-level events”:
  the XML stream is not seen as the traversal of a large instance, but as a sequence of (independent) XML fragments that are seen as application-level events
  [RFID applications, time series of stock quotes, RSS feeds]

Example: XML Stream Communication

```java
import java.io.PipedInputStream;
import java.io.PipedOutputStream;
import java.io.OutputStream;
import javax.xml.stream.XMLOutputFactory;
import javax.xml.stream.XMLStreamWriter;

public class XMLStreamTestWriter implements Runnable
{
  OutputStream outputStream;

  public XMLStreamTestWriter(OutputStream out) {
    this.outputStream = out;
  }

  public void run() {
    try {
      XMLOutputFactory outputFactory = XMLOutputFactory.newInstance();
      XMLStreamWriter writer = outputFactory.createXMLStreamWriter(outputStream);
      writer.writeStartElement("foo");
      int i=1;
      while (i<100) {
        writer.writeStartElement("bla");
        writer.writeCharacters(" " + i);
        writer.writeEndElement();
        System.out.print("Write  <bla>" + i + "</bla> ");
        //writer.flush(); // if not uncommented: strictly alternating
        // comment out flush: sleep < 700 causes alternating after blocks of 2...5 elements
        try{ java.lang.Thread.sleep(50); }
          catch (Exception e) { e.printStackTrace(); }
        i++;
      }
      // writer.writeEndElement();  // close </foo> is done by the next line:
      writer.writeEndDocument(); // docu: closes all tags, but does not send anything else
      writer.flush();
      writer.close();
    } catch (Exception e) { e.printStackTrace(); }
    System.out.println("Writer finished");
  }

  public static void main(String[] args) throws Exception{
    PipedOutputStream pos = new PipedOutputStream();
    PipedInputStream pis = new PipedInputStream();
    pis.connect(pos);
    new Thread (new XMLStreamTestWriter(pos)).start();
    new Thread (new XMLStreamTestReader(pis)).start();
  }
}
```

[Filename: java/StAX/XMLStreamTestWriter.java]

- underlying: connected PipedOutput/InputStream

Example: XML Stream Communication (Cont’d)

```
import java.io.PipedInputStream;
import java.io.InputStream;
import javax.xml.stream.XMLInputFactory;
import javax.xml.stream.XMLStreamConstants;
import javax.xml.stream.XMLStreamReader;

public class XMLStreamTestReader implements Runnable {

  InputStream inputStream;

  public XMLStreamTestReader(PipedInputStream in) {
    this.inputStream = in;
  }

  public void run() {
    try {
      XMLInputFactory inputFactory = XMLInputFactory.newInstance();
      XMLStreamReader parser = inputFactory.createXMLStreamReader(inputStream);
      boolean goOn = true;
      while (goOn) {
        int event = 0;
        try {
          event = parser.next();
          switch(event) {
            case XMLStreamConstants.START_ELEMENT:
              System.out.println("Read start element " + parser.getLocalName());
              break;
            case XMLStreamConstants.CHARACTERS:
              System.out.println("Read " + parser.getText());
              break;
            case XMLStreamConstants.END_ELEMENT:
              System.out.println("Read end element " + parser.getLocalName());
              break;
              case XMLStreamConstants.END_DOCUMENT: // never happens!
              System.out.println("Read end document");
              goOn = false;
            default: System.out.println("Read something else. event: " + event);
          }}
          catch(Exception e) { parser.close(); goOn = false; }
      }
      parser.close();
      System.out.println("Reader finished");
    }
    catch (Exception e) { e.printStackTrace(); }
  }
}
```

[Filename: java/StAX/XMLStreamTestReader.java]