XML in programming

Patryk Czarnik

Institute of Informatics University of Warsaw

XML and Modern Techniques of Content Management – 2012/13

XML in programming — what for?

- To access data in XML format
- To use XML as data carrier (storage and transmission)
- To support XML applications (Web, content management, . . .)
- To make use of XML-related standards (XML Schema, XInclude, XSLT, XQuery, XLink, . . .)
- To develop or make use of XML-based software technology (Web Services etc.)

XML in programming — how?

Bad way

- Treat XML as plain text files and write low-level XML support from scratch

Better approach

- Use existing libraries and tools

Even better

- Use standardised interfaces independent from particular suppliers

XML in Java

Java — motivation for us

- One of the most popular programming languages
- Very good XML support

Standards

Both included in Java Standard Edition from 6.0

- Java API for XML Processing (JAXP)
  - many interfaces and few classes, ”factories” and pluggability layer
  - support for XML parsing and serialisation (DOM, SAX, StAX)
  - support for XInclude, XML Schema, XPath, XSLT
- Java API for XML Binding (JAXB)
  - binding between Java objects and XML documents

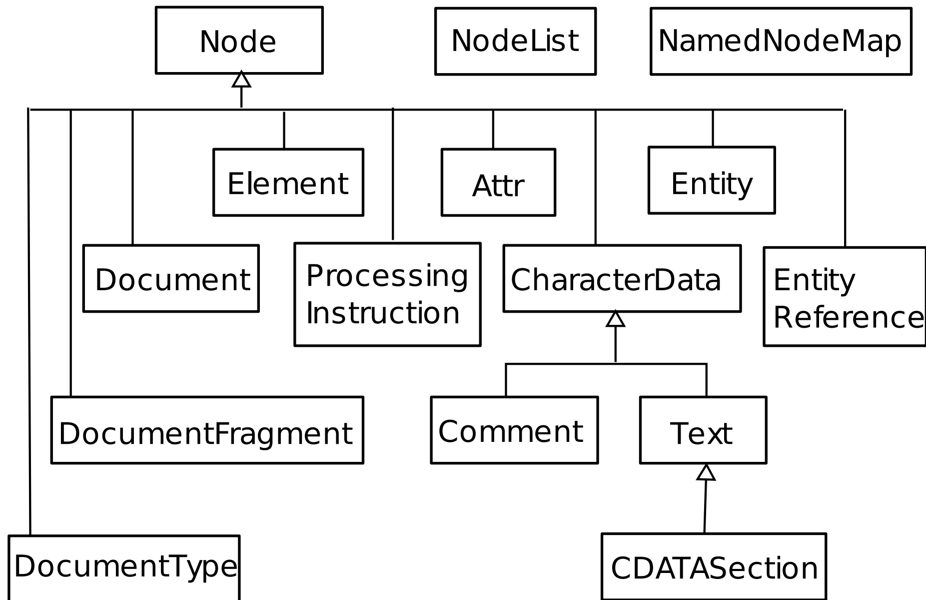
Classification of XML access models

- Document read into memory
  - generic interface
    example: **DOM**
  - interface depending on document type (schema)
    example: **JAXB**
- Document processed node by node
  - event model (*push parsing*)
    example: **SAX**
  - streaming model (*pull parsing*)
    example: **StAX**

Document Object Model

- Whole document in memory, as tree of objects
- W3C Recommendations
  - DOM Level 1 — 1998
  - DOM Level 3 — 2004
- Specification split into many modules
  (separate recommendations)
  - **Dom Core** — the most important DOM module for XML
- Standard independent of programming language
  (interfaces given in IDL)
- Java realisation presented here

DOM — important interfaces



Node interface — chosen methods

Content access

- getAttributes()
- getChildNodes()
- getFirstChild()
- getLastChild()
- getNextSibling()
- getNodeName()
- getNodeValue()
- getNodeType()
- getOwnerDocument()
- getParentNode()
- hasChildNodes()

Content modification

- appendChild(Node)
- insertBefore(Node, Node)
- removeChild(Node)
- replaceChild(Node, Node)
- setNodeValue(String)
- setNodeName(String)

Copying

- cloneNode(boolean)
- importNode()

Example — introduction

Example document

```
<department id="acc">
   <name>Accountancy</name>
   <person id="102103" position="specialist">
      <first-name>Dawid</first-name>
      <last-name>Paszkiewicz</last-name>
      <phone type="office">+48223213203</phone>
      <phone type="fax">+48223213200</phone>
      <email>paszkiewicz@example.com</email>
      <salary>2222</salary>
   </person>
   <person id="102104" position="chief">
      <first-name>Monika</first-name>
      <last-name>Domżałowicz</last-name>
      . . .
      <salary>3500</salary>
   </person>
   . . .
```

Problem: Count sum of salaries of assistants

DOM — example (1)

Program

```
int result = 0;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(args[0]);

Node cur = doc.getFirstChild();
while(cur.getNodeType() != Node.ELEMENT_NODE)
   cur = cur.getNextSibling();
cur = cur.getFirstChild();
while(cur != null) {
   if(cur.getNodeType() == Node.ELEMENT_NODE) {
      Node attr = cur.getAttributes().getNamedItem("position");
      if(attr != null && "specialist".equals(attr.getNodeValue())) {
         result += processPerson(cur);
   }  }
   cur = cur.getNextSibling();
}
```

DOM — example (2)

Method processPerson

```
private static int processPerson(Node person) {
   Node cur = person.getFirstChild();
   while(cur != null) {
      if(cur.getNodeType() == Node.ELEMENT_NODE
         && "salary".equals(cur.getNodeName())) {
         StringBuffer buf = new StringBuffer();
         Node child = cur.getFirstChild();
         while(child != null) {
            if(child.getNodeType() == Node.TEXT_NODE)
               buf.append(child.getNodeValue());
            child = child.getNextSibling();
         }
         return Integer.parseInt(buf.toString());
      }
      cur = cur.getNextSibling();
   }
   return 0; // no salary found
}
```

DOM — two styles of programming

Generic approach — use only Node interface

- nodeType property — kind of node
- properties: `nodeName`, `nodeValue`, `childNodes` ...
  — content access and modification
- methods: `appendChild(Node)`, `removeChild(Node)` ...
  — structure modification

Object-oriented approach — use specialised interfaces

- **Document**: `getDocumentElement()`, `getElementById(String)`
- **Element**: `getElementsByTagName(String)`,
  `getAttribute(String)`, `setAttribute(String, String)`
- **Attribute**: `boolean getSpecified()`
- **Text**: `String substringData(int, int)`,
  `insertData(int, String)`

DOM with subclasses — example (1)

Program

```
int result = 0;
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = factory.newDocumentBuilder();
Document doc = builder.parse(args[0]);

Element root = doc.getDocumentElement();
NodeList list = root.getElementsByTagName("person");
for(int i = 0; i < list.getLength(); ++i) {
   Element person = (Element) list.item(i);
   if("specialist".equals(person.getAttribute("position")))
      result += processPerson(person);
}
```

DOM with subclasses — example (2)

Method processPerson

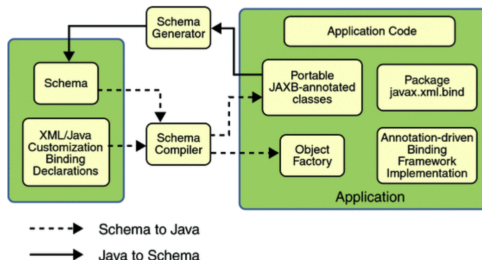
```
private static int processPerson(Element person) {
   NodeList list = person.getElementsByTagName("salary");
   if (list.getLength() > 0) {
      String value = list.item(0).getTextContent();
      return Integer.parseInt(value);
   } else {
      return 0;
}  }
```

XML binding — idea

- XML documents and objects (e.g. Java):
  - schema (type) corresponds to class,
  - document/node corresponds to object.
- Implementations:
  - JAXB (Sun), Castor (Exolab), XML Beans (Apache).

Java API for XML Binding (JAXB)

- Inspired by Sun, supported by java.net
- Current version: 2.0
- Motivated by Web Services
- Components
  - generic API fragment
  - two-way translation specification
  - customisation and annotations



JAXB — usage scenarios

Schema to Java

1. Preparation of schema
2. Schema to Java compilation (XJC)
   - translation customisable
   - result: annotated classes basing on schema types
3. Developing application making use of
   - generic part of JAXB API
   - generated classes

Java to Schema

1. Preparation of model classes
2. Adding annotations to classes
3. Making use of generic JAXB API to perform marshalling, unmarshalling etc.
4. (Optional) Generating schema basing on classes

JAXB — example (1)

Schema and generated classes

Schema

```
<xs:element name="department">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="name" type="xs:string" minOccurs="0"/>
      <xs:element name="person" type="TPerson" maxOccurs="unbounded"/>
    </xs:sequence>
    <xs:attribute name="id" type="xs:string"/>
  </xs:complexType>
</xs:element>
```

Generated class

```
@XmlRootElement(name = "department")
public class Department {
   protected String name;
   protected List<TPerson> person;
   @XmlAttribute(name = "id")
   protected String id;
   ...
```

JAXB — example (2)

Schema

```
<xs:complexType name="TPerson">
  <xs:sequence>
    <xs:element name="first-name" type="xs:string"/>
    <xs:element name="last-name" type="xs:string"/>
    <xs:element name="phone" type="TPhone"
        minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="email" type="xs:token"
        minOccurs="0" maxOccurs="unbounded"/>
    <xs:element name="salary" type="xs:int" minOccurs="0"/>
  </xs:sequence>
  <xs:attribute name="id" type="xs:integer" use="required"/>
  <xs:attribute name="position" type="xs:token"/>
</xs:complexType>
```

JAXB — example (3)

Generated class

```
public class TPerson {
   @XmlElement(name = "first-name", required = true)
   protected String firstName;
   @XmlElement(name = "last-name", required = true)
   protected String lastName;
   protected List<TPhone> phone;
   @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
   protected List<String> email;
   protected Integer salary;
   @XmlAttribute(name = "id", required = true)
   protected BigInteger id;
   @XmlAttribute(name = "position")
   protected String position;
   ...
```

JAXB — example(4)

Program

```
JAXBContext jc = JAXBContext.newInstance(Department.class);
Unmarshaller u = jc.createUnmarshaller();
Department doc = (Department)u.unmarshal(new FileInputStream(args[0]));
List<TPerson> employees = doc.getPerson();
for(TPerson person : employees) {
   if("specialist".equals(person.getPosition())) {
      if(person.getSalary() != null)
         result += person.getSalary();
   }
}
System.out.println("Result: "+result);
```

Processing documents node by node — motivation

Whole document in memory (DOM, JAXB)

- Convenient, high-level approach
- Cost for efficiency
  - memory usage
  - creating tree with references (DOM)
  - translating data between text and binary format (JAXB)

Reading document node by node

- Efficient
  - possible processing in constant memory
  - less operations to perform
- Harder to develop complex logic

Event model — idea

- Document seen as a stream of events: “element beginning”, “text node”, “element end”, . . .),
- Programmer registers code to be run during processing
- Parser reads document and calls programmer’s methods relevant to particular events
  - data (element names, text fragments) given in parameters

Separation of responsibility

- Parser responsible for physical-level processing
- Programmer responsible for logical-level processing

SAX — typical usage (1)

Status

- Simple API for XML. Version 1.0 in 1998
- Interfaces specified in Java
- Idea applicable for other programming languages

Typical usage

1. Programmer-provided class implementing `ContentHandler`
2. Optionally classes implementing `ErrorHandler`, `DTDHandler`, or `EntityResolver`
   - one class may implement all of them
   - `DefaultHandler` — convenient base class to start with

SAX — typical usage (2)

Typical application schema

1. Obtain `XMLReader` (or `SAXParser`) from factory
2. Create `ContentHandler` instance
3. Register handler in reader
4. Invoke `parse` method
5. Parser conducts processing and calls methods of our `ContentHandler`
6. Use data collected by `ContentHandler`

SAX example (1)

ContentHandler implementation

```
class StaffHandler extends DefaultHandler {
   enum State {EXTERNAL, PERSON, SALARY};
   private int result = 0;
   private State state = State.EXTERNAL;
   private StringBuffer buf;

   public int getResult() {
      return result;
   }

   public void startElement(String uri, String localName, String qName,
         Attributes attributes) throws SAXException {
      if("person".equals(qName)) {
         String attrVal = attributes.getValue("position");
         if("specialist".equals(attrVal))
            state = State.PERSON;
      } else if("salary".equals(qName)) {
         if(state == State.PERSON) {
            state = State.SALARY;
            buf = new StringBuffer();
   } } }
```

SAX example (2)

StaffHandler continued

```
   public void characters(char[] ch, int start, int length)
         throws SAXException {
      if(state == State.SALARY)
         buf.append(ch, start, length);
   }

   public void endElement(String uri, String localName, String qName)
         throws SAXException {
      if("person".equals(qName)) {
         if(state == State.PERSON) {
            state = State.EXTERNAL;
         }
      } else if("salary".equals(qName)) {
         if(state == State.SALARY) {
            state = State.PERSON;
            result += Integer.parseInt(buf.toString());
} } } }
```

SAX example (3)

Main program (Traditional SAX version)

```
XMLReader reader = XMLReaderFactory.createXMLReader();
StaffHandler handler = new StaffHandler();
reader.setContentHandler(handler);
reader.parse(new InputSource(args[0]));
System.out.println("Result: "+handler.getResult());
```

Main program (JAXP version)

```
SAXParserFactory factory = SAXParserFactory.newInstance();
SAXParser parser = factory.newSAXParser();
StaffHandler handler = new StaffHandler();
parser.parse(args[0], handler);
System.out.println("Result: "+handler.getResult());
```

SAX filters

- Implement `XMLFilter` interface, indirectly also `XMLReader`
  - behave like parser, but receive events from another `XMLReader` (parser or filter)
  - may be chained
- Default implementation: `XMLFilterImpl`:
  - implements `ContentHandler`, `ErrorHandler` and so on
  - passes all events through
- Capabilities:
  - events filtering
  - changing data or structure, by calling subsequent handler methods with changed parameters
  - on-line document processing
  - processing by many modules in one-pass reading

SAX filter example

```
public class AssistantFilter extends XMLFilterImpl {
   private boolean pass = true;

   public void startElement(String uri, String localName, String qName,
         Attributes atts) throws SAXException {
      if("person".equals(qName)
            && "assistant".equals(atts.getValue("position")))
         pass = false;
      if(pass)
         super.startElement(uri, localName, qName, atts);
   }

   public void endElement(String uri, String localName, String qName)
         throws SAXException {
      if(pass)
         super.endElement(uri, localName, qName);
      if("person".equals(qName))
      czyPrzepuszczac = true;
   }

   public void characters(char[] ch, int start, int length)
         throws SAXException {
      if(pass)
         super.characters(ch, start, length);
}  }
```

Stream model (*pull parsing*)

- Alternative for event model
    - application “pulls” events/nodes from parser
    - processing controlled by application, not parser
    - idea analogous to: iterator, cursor, etc.
- Advantages of SAX saved
    - high efficiency
    - possible to process large documents
- Standardisation
    - Common XmlPull API
    - Java Community Process, JSR 173: Streaming API for XML

StAX — Streaming API for XML

Important interfaces

- `XMLStreamReader` — low level parser
  - `hasNext()`, `int next()`, `int getEventType()`,
  - `getName()`, `getValue()`, `getAttributeValue()`, ...
- `XMLEventReader` — (relatively) high level parser
  - `XMLEvent next()`, `XMLEvent peek()`
- `XMLEvent` — represents event in “high level” approach
  - `getEventType()`, `isStartElement()`, `isCharacters()`, ...
  - podinterfejsy `StartElement`, `Characters`, ...
- `XMLStreamWriter`, `XMLEventWriter`
- `XMLStreamFilter`, `XMLEventFilter`

StAX — example (1)

Program

```
private static XMLStreamReader reader;

public void run() {
   int result = 0;
   XMLInputFactory factory = XMLInputFactory.newInstance();
   reader = factory.createXMLStreamReader(new FileInputStream(args[0]));

   while(reader.hasNext()) {
      int eventType = reader.next();
         if(eventType == XMLStreamConstants.START_ELEMENT) {
            if("person".equals(reader.getLocalName())) {
               String attrVal = reader.getAttributeValue(null,"position");
               if("specialist".equals(attrVal)) {
                  result += this.processGroup();
   } } } } }
   reader.close();
   System.out.println("Result: "+result);
}
```

StAX — example (2)

Method processPerson

```
private int processPerson() throws XMLStreamException {
   int result = 0;

   while(reader.hasNext()) {
      int eventType = reader.next();
      switch(eventType) {
      case XMLStreamConstants.START_ELEMENT :
         if("salary".equals(reader.getLocalName())) {
            String val = reader.getElementText();
            result += Integer.parseInt(val);
         }
      break;
      case XMLStreamConstants.END_ELEMENT :
         if("person".equals(reader.getLocalName())) {
            return result;
         }
   }   }
   return result;
}
```

Which model to choose? (1)

- Document tree in memory:
  - small documents (must fit in memory)
  - concurrent access to many nodes
  - creating new and editing existing documents “in place”
- Generic document model (like DOM):
  - not established or not known structure of documents
  - lower efficiency accepted
- XML binding (like JAXB):
  - established and known structure of documents
  - XML as data serialisation method

Which model to choose? (2)

- Processing node by node
  - potentially large documents
  - relatively simple, local operations
  - efficiency is the key factor
- Event model (like SAX):
  - using already written logic (SAX is more mature)
  - filtering of events
  - asynchronous events
  - several aspects of processing during one reading of document
- Streaming model (like StAX):
  - processing depending on context; complex states
  - processing should end after data is found
  - reading several documents simultaneously

Validation against DTD during parsing

Parsing using *validating parser*

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
XMLReader reader = factory.newSAXParser().getXMLReader();

reader.setContentHandler(mojConentHandler);
reader.setErrorHandler(mojErrorHandler);

reader.parse(args[0]);
```

Note

Validating parser defined formally in XML Recommendation.

Validation against XML Schema during parsing

One of possible approaches

```
SchemaFactory schemaFactory =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(new StreamSource("staff.xsd"));

SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(false);
factory.setSchema(schema);
factory.setNamespaceAware(true);

XMLReader reader = factory.newSAXParser().getXMLReader();
reader.setContentHandler(myConentHandler);
reader.setErrorHandler(myErrorHandler);
reader.parse(args[0]);
```

Validators, saving DOM tree to file

In-memory validation of DOM tree

```
SchemaFactory schemaFactory =
    SchemaFactory.newInstance(XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = schemaFactory.newSchema(new StreamSource("staff.xsd"));
Validator validator = schema.newValidator();
validator.setErrorHandler(myErrorHandler);
validator.validate(new DOMSource(doc));
```

DOM tree saved using *DOM Load and Save* interface

```
DOMImplementationLS lsImpl =
    (DOMImplementationLS)domImpl.getFeature("LS", "3.0");
LSSerializer ser = lsImpl.createLSSerializer();
LSOutput out = lsImpl.createLSOutput();
out.setByteStream(new FileOutputStream("out.xml"));
ser.write(doc, out);
```

Transformers for XSLT

XSLT run for external XML files

```
TransformerFactory tf = TransformerFactory.newInstance();
Transformer transformer = tf.newTransformer(
     new StreamSource("staff_html.xsl"));

Source src = new StreamSource("staff.xml");
Result res = new StreamResult("result.html");

transformer.transform(src, res);
```

Transformers — other applications

SAX filter + Transformer = on-line document processing

```
SAXParserFactory parser_fact = SAXParserFactory.newInstance();
XMLReader reader = parser_fact.newSAXParser().getXMLReader();

TransformerFactory tf = TransformerFactory.newInstance();
Transformer transformer = tf.newTransformer();

XMLFilter filter = new AssistantFilter();
filter.setParent(reader);

InputSource doc = new InputSource(args[0]);

Source src = new SAXSource(filter, doc);
Result res = new StreamResult("out.xml");

transformer.transform(src, res);
```

Source, Result, and Transformers

- Source and Result — universal wrappers for various Java representations of XML data
- Transformer — translation between them

Source

- DOMSource
- JAXBSource
- SAXSource
- StAXSource
- StreamSource

Result

- DOMResult
- JAXBResult
- SAXResult
- StAXResult
- StreamResult
- SAAJResult