XML in Programming

Patryk Czarnik

XML and Applications 2015/2016
Lecture 5 – 4.04.2016

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 technology
  - XML RPC, Web Services (SOAP, WSDL)
  - REST, AJAX

XML in programming – how?

- Bad way
  - Treat XML as plain text and write low-level XML support from scratch
- Better approach
  - Use existing libraries and tools
- Even better
  - Use standardised interfaces independent of particular suppliers

XML and Java

- Propaganda
  - Java platform provides device-independent means of program distribution and execution.
  - XML is a platform-independent data carrier.
- Practice
  - Java – one of the most popular programming languages, open and portable.
  - Very good XML support in Java platform.
  - Many technologies use XML.

Of course you can find very good (or at least *not bad*) XML support on other programming platforms, but we have to choose one for presentation and exercises.

XML in Java – standards

Both included in Java Standard Edition since v.6

- Java API for XML Processing (**JAXP** 1.x – JSR-206)
  - many interfaces and few actual 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** 2.x – JSR-222)
  - binding between Java objects and XML documents
  - annotation-driven
  - strict relation with XML Schema

Classification of XML access models

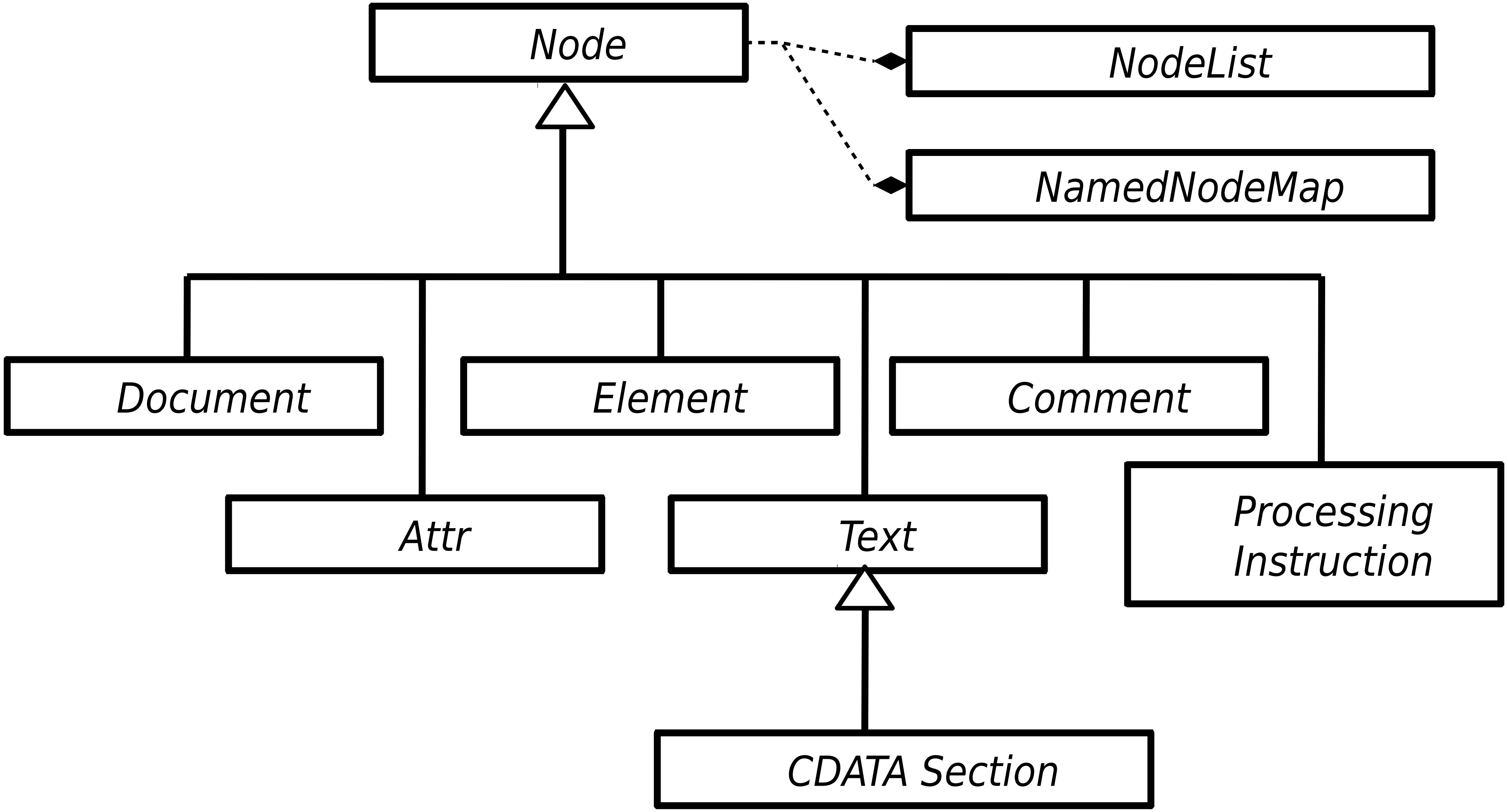
And their “canonical” realisations in Java

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

Document Object Model

- W3C Recommendations
  - DOM Level 1 – 1998
  - DOM Level 3 – 2004
  - Several modules. We focus on DOM Core here
- Document model and universal API
  - independent of programming language (IDL)
  - independent of particular XML application
- Used in various environments
  - notable role in JavaScript / ECMA Script model
  - available (in some form) for all modern programming platforms

Primary DOM types



DOM key ideas

- Whole document in memory
- Tree of objects
- Generic interface Node
- Specialised interfaces for particular kinds of nodes
- Available operations
  - reading document into memory
  - creating document from scratch
  - modifying content and structure of documents
  - writing documents to files / streams

Example: problem introduction

- Count the number of seats in rooms equipped with a projector.

```
<rooms>
    <room>
        <number>2120</number>
        <floor>1</floor>
        <equipment projector="false" computers="false"/>
        <seats>50</seats>
    </room>
    <room>
        <number>3180</number>
        <floor>2</floor>
        <equipment projector="true" computers="false"/>
        <seats>100</seats>
    </room>
    <room>
        <number>3210</number>
        <floor>2</floor>
        <equipment />
        <seats>30</seats>
    </room>
</rooms>
```

DOM in Java example

Parsing and basic processing

```
DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
DocumentBuilder builder = dbf.newDocumentBuilder();
Document doc = builder.parse(fileName);

for(Node node = doc.getFirstChild();
       node != null;
       node = node.getNextSibling()) {
    if(node.getNodeType() == Node.ELEMENT_NODE
           && "rooms".equals(node.getNodeName())) {
        this.processRooms(node);
    }
}
```

Whole example in `CountSeats_DOM_Generic.java`

DOM in Java example

Visiting nodes in the tree

```
private void processRooms(Node roomsNode) {
    for(Node node = roomsNode.getFirstChild();
            node != null;
            node = node.getNextSibling()) {
        if(node.getNodeType() == Node.ELEMENT_NODE
                && "room".equals(node.getNodeName())) {
            this.processRoom(node);
}   }   }
```

```
private void processRoom(Node roomNode) {
    boolean hasProjector = false;
    Node seatsNode = null, equipmentNode = null;

    for(Node node = roomNode.getFirstChild();
            node != null;
            node = node.getNextSibling()) {
        // searching for <equipment> node
        if(node.getNodeType() == Node.ELEMENT_NODE
                && "equipment".equals(node.getNodeName())) {
            equipmentNode = node;
            break;
    }   }
...
```

DOM in Java example

Access to attributes and text nodes

```
...
if(equipmentNode != null) {
    NamedNodeMap equipmentAttributes = equipmentNode.getAttributes();
    Node projectorNode = equipmentAttributes.getNamedItem("projector");
    if(projectorNode != null) {
        String projector = projectorNode.getNodeValue();
        if("true".equals(projector) || "1".equals(projector)) {
            hasProjector = true;
}   }   }
...
```

```
...
if(seatsNode != null) {
    String seatsString = seatsNode.getTextContent();
    try {
        int seats = Integer.parseInt(seatsString);
        sum += seats;
    } catch (NumberFormatException e) {
        // Incorrect number format is silently ignored (sum is not increased).
}   }
...
```

Approaches to using DOM

- Two approaches in DOM programming
  - Use only generic `Node` interface
  - Use specialised interfaces and convenient methods
- Example features of specialised `Element` interface:
  - searching the subtree for elements of the given name `getElementsByTagName`, `getElementsByTagNameNS`
  - direct access to attribute values `getAttribute`, `getAttributeNS`, `setAttribute`, `setAttributeNS`

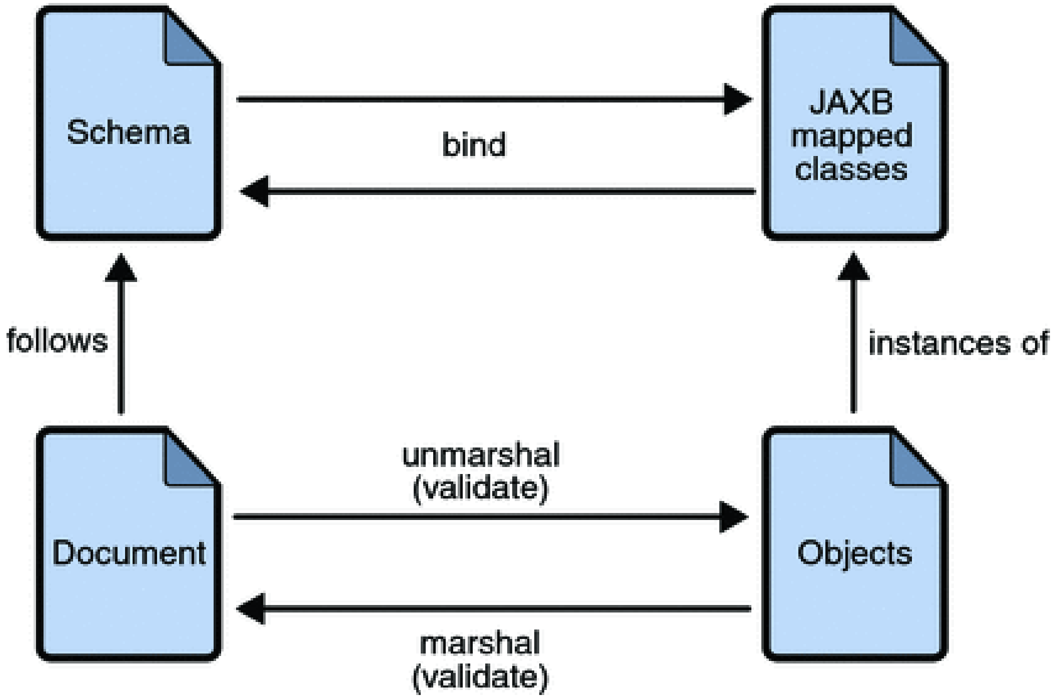
Using specialised interfaces (fragments)

```
...
Document doc = builder.parse(fileName);
Element rooms = doc.getDocumentElement();
if("rooms".equals(rooms.getNodeName()))
    this.processRooms(rooms);
...
NodeList list = roomsElem.getElementsByTagName("room");
for(int i=0; i < list.getLength(); ++i) {
    this.processRoom(list.item(i));
}
...
Element equipmentElem = (Element) roomElem.
                          getElementsByTagName("equipment").item(0);
...
if(equipmentElem != null) {
    String projector = equipmentElem.getAttribute("projector");
    if("true".equals(projector) || "1".equals(projector)) {
        hasProjector = true;
}   }
```

Whole example in `CountSeats_DOM_Specialized.java`

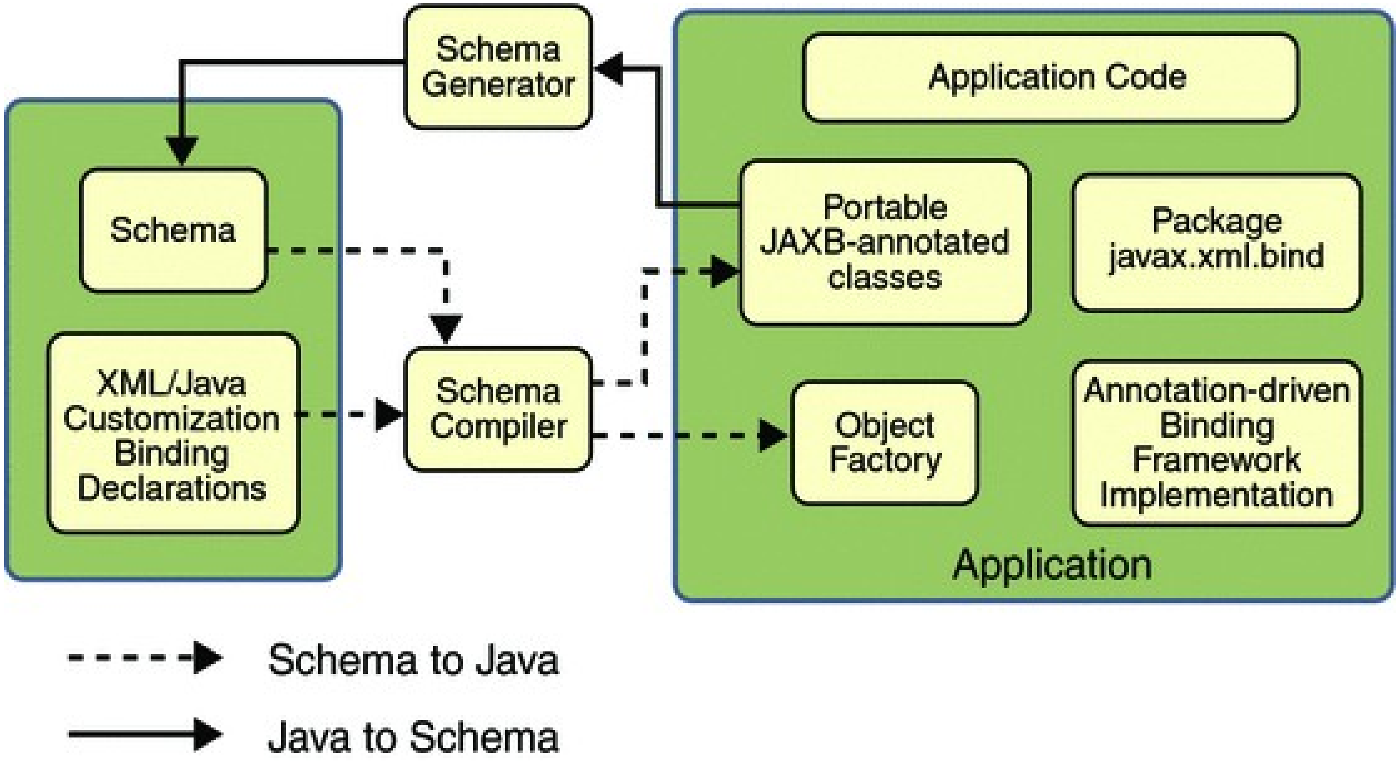
XML binding and JAXB

- Mapping XML to Java
- High-level view on documents
- From programmer's point of view:
  - instead of Integer.parseString(room.getElementsByTagsName("seats").item(0).getTextContent())
  - we simply have room.getSeats()



JAXB 2.x architecture

- Application operates basing on (usually annotated) “JAXB classes”
  - generated from a schema
  - or written manually



JAXB example

- We generate Java classes basing on our schema
  - `xjc -d src -p` *`package_name`* `school.xsd`
- One of generated classes:

```
@XmlAccessorType(XmlAccessType.FIELD)
@XmlType(name = "Room", propOrder = {
    "number", "floor", "equipment", "seats"})
public class Room {
    @XmlElement(required = true)
    @XmlJavaTypeAdapter(CollapsedStringAdapter.class)
    @XmlSchemaType(name = "token")
    protected String number;
    protected byte floor;
    @XmlElement(required = true)
    protected RoomEquipment equipment;
    @XmlSchemaType(name = "unsignedShort")
    protected Integer seats;
...
```

All generated classes are in ...`jaxb_generated`
and the program in `CountSeats_JAXB`

JAXB example

```
JAXBContext jaxbContext = JAXBContext.newInstance(Rooms.class);
Unmarshaller u = jaxbContext.createUnmarshaller();
Rooms rooms = (Rooms) u.unmarshal(new File(fileName));
if(rooms != null)
    this.processRooms(rooms);
```

```
private void processRooms(Rooms rooms) {
    for(Room room : rooms.getRoom()) {
        if(room.getEquipment().isProjector()
                && room.getSeats() != null) {
            sum += room.getSeats();
}   }   }
```

JAXB – applications and alternatives

- Primary applications:
  - high-level access to XML documents
  - serialisation of application data
  - automatic mapping of method invocations to SOAP messages in JAX-WS
- Many options to customise the mapping using Java or XML annotations
- Some alternatives:
  - Castor
  - Apache XML Beans
  - JiBX

Streaming (and event) processing
Motivation

- Whole document in memory (DOM, JAXB)
  - convenient
  - but expensive
    - memory for document
      (multiplied by an overhead for structure representation)
    - time for building the tree
    - reading always whole document, even if required data present at the beginning
  - sometimes not possible at all
    - more memory required than available
    - want to process document before it ends
- Alternative: Reading documents node by node

Event model

- Document seen as a sequence of events
  - “an element is starting”,
  - “a text node appears”, etc.
- Programmer provides code fragments – “event handlers”
- Parser reads a document and
  - controls basic syntax correctness
  - calls programmer's code relevant to actual events
- Separation of responsibility:
  - Parser responsible for physical-level processing
  - Programmer responsible for logical-level processing

SAX

- Simple API for XML – version 1.0 in 1998
- Original standard designed for Java
- Idea applicable for other programming languages

Typical usage:

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

SAX

Typical usage (ctnd):

- Obtain `XMLReader` (or `SAXParser`) from factory
- Create `ContentHandler` instance
- Register handler in reader
- Invoke `parse` method
    - Parser conducts processing and calls methods of our `ContentHandler`
- Use data collected by `ContentHandler`

SAX events in run

```
<?xml-stylesheet ...?>

<room>

  <equipment projector="true"/>


  <seats>
    60
  </seats>
</room>
```

- startDocument()
- processingInstruction(
  "xml-stylesheet", ...)
- startElement("room")
- startElement("equipment",
  {projector="true"})
- endElement("equipment")
- startElement("seats")
- characters("60")
- endElement("seats")
- endElement("room")
- endDocument()

SAX example (fragments)

```
CSHandler handler = new CSHandler();
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setContentHandler(handler);
reader.parse(new InputSource(fileName));
return handler.getSum();
```

```
public class CSHandler implements ContentHandler {
    ...
        public void startElement(String uri, String localName,
            String qName, Attributes atts) throws SAXException {
        switch(state){
        ...
        case IN_ROOM:
            if("equipment".equals(qName)) {
                String projector = atts.getValue("projector");
                if("true".equals(projector) || "1".equals(projector))
                    state = CSHandler_States.IN_ROOM_WITH_PROJECTOR;
            }
...
```

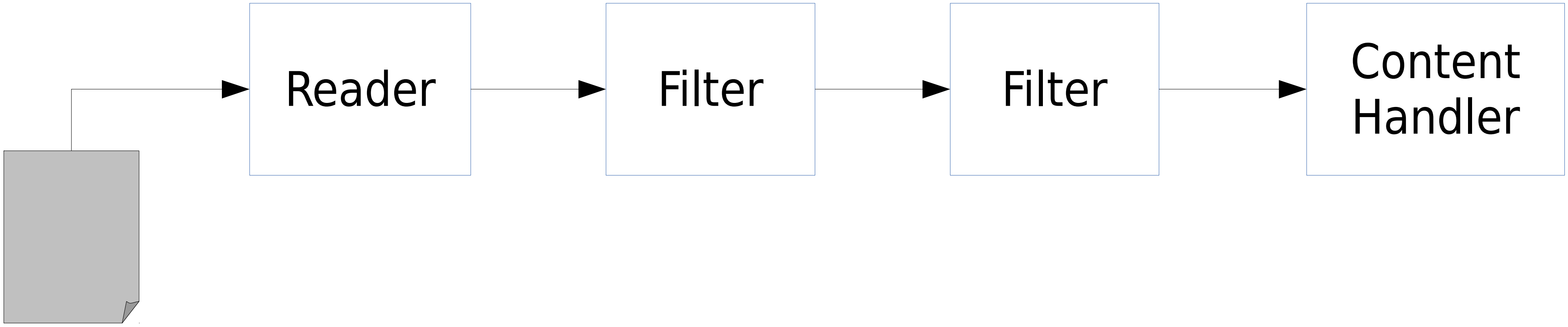
SAX examples

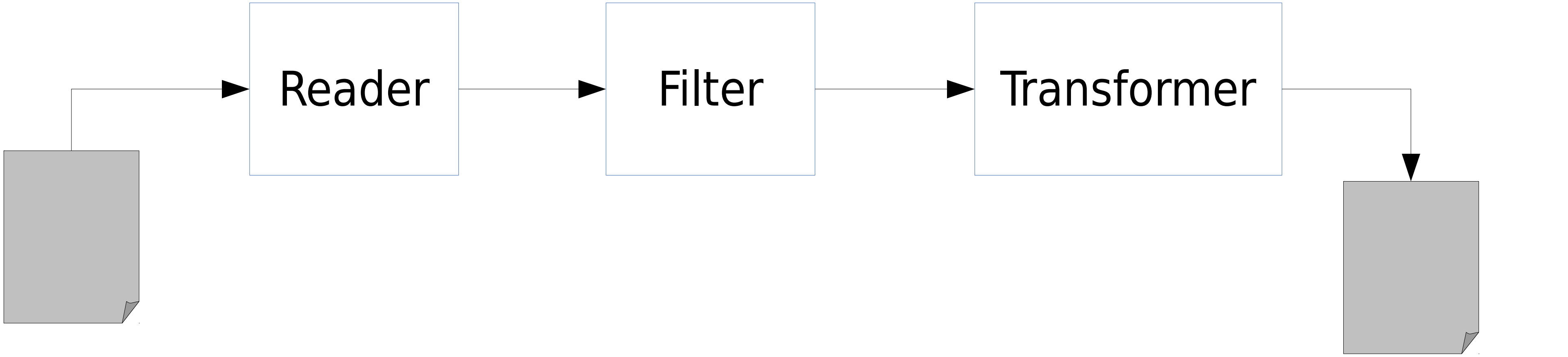
- See whole example classes:
  - CountSeats_SAX_Traditional and CSHandler_Traditional for traditional scenario of creating parses instance and registering a `ContentHandler`
  - CountSeats_SAX_JAXP and CSHandler_JAXP for modern JAXP-conformant scenario of combining things together

SAX filters

- Motivation: Joining ContentHandler-like logic into chains
- Realisation:
  - interface `XMLFilter`
    (`XMLReader` having a parent `XMLReader`)
  - in practice filters implements also `ContentHandler`
  - convenient start-point: `XMLFilterImpl`
- Typical implementation of a filter:
  - handle incoming events like in a `ContentHandler`
  - pass events through by manual method calls on the next item in chain
- Filters can:
  - pass or halt an event
  - modify an event or a sequence of events!

Possible usage of SAX filters





SAX Filters – example?

- We're not going to resolve our example program using filters, as it makes a little sense.
- An example filter can be found in `more_sax/UpperCaseFilter`

SAX – typical problems

- To make implementations portable – we should manually join adjacent text nodes in an element
  - `StringBuilder` is a convenient class
- The same method called for different elements, in different contexts
  - Typical solution – remembering the state:
    - one boolean flag in simplest cases
    - enum is usually enough
    - elaborated structures may be required for complex logic
  - It may become tedious in really complex cases.

StAX: Pull instead of being pushed

- Alternative for event model
  - application “pulls” events/nodes from parser
  - processing controlled by application, not parser
  - idea analogous to: iterator, cursor, etc.
- More intuitive control flow
  - reduced need of remembering the state etc.
- Advantages of SAX saved
  - high efficiency
  - possibility to process large documents

StAX

- Streaming API for XML
- Available in Java SE since version 6

Two levels of abstraction:

- `XMLStreamReader`
  - one object for all purposes
  - most efficient approach
- `XMLEventReader`
  - subsequent events (nodes) provided as separate objects
  - more convenient for high-level programming, especially when implementing modification of the document “on-the-fly”

StAX example with XMLStreamReader (fragments)

```
XMLInputFactory xif = XMLInputFactory.newInstance();
reader = xif.createXMLStreamReader(new FileInputStream(fileName));
while (reader.hasNext()) {
    if (reader.isStartElement()
            && "rooms".equals(reader.getLocalName())) {
        this.processRooms();
    }
    reader.next();
}
```

```
while (reader.hasNext()) {
    if (reader.isStartElement()
            && "equipment".equals(reader.getLocalName())) {
        String projector = reader.getAttributeValue(
                XMLConstants.NULL_NS_URI, "projector");
        if ("true".equals(projector) || "1".equals(projector)) {
            hasProjector = true;
        }
    } else if (hasProjector && reader.isStartElement()
            && "seats".equals(reader.getLocalName())) {
        ...
```

StAX example with XMLEventReader (fragments)

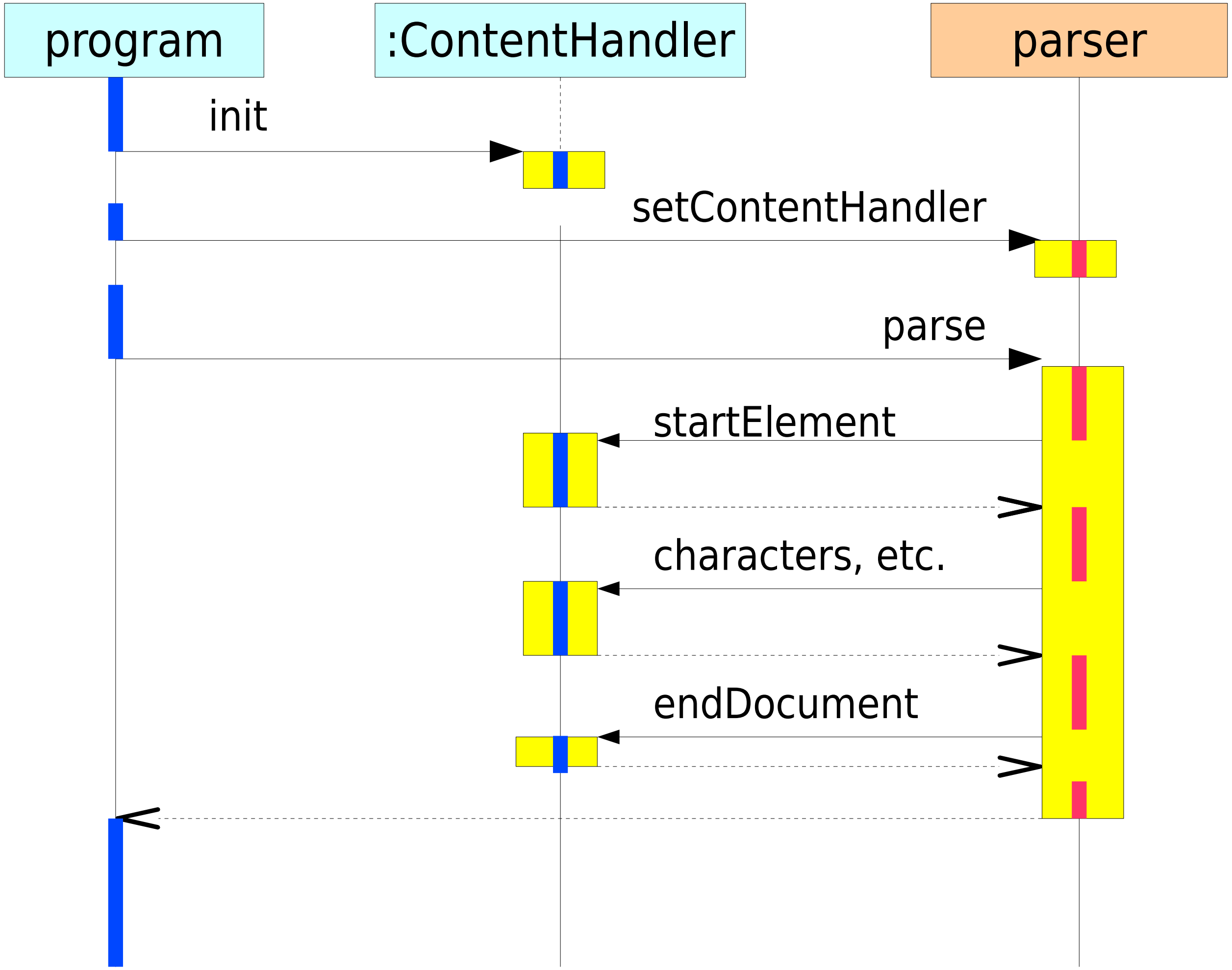
```
XMLInputFactory xif = XMLInputFactory.newInstance();
reader = xif.createXMLEventReader(new FileInputStream(fileName));
while (reader.hasNext()) {
    XMLEvent event = reader.nextEvent();
    if (event.isStartElement()
        && "rooms".equals(event.asStartElement().
                                    getName().getLocalPart())) {
        this.processRooms();
    }
}
```

```
while (reader.hasNext()) {
    XMLEvent event = reader.nextEvent();
    if (event.isStartElement() && "equipment".equals(
                event.asStartElement().getName().getLocalPart())) {
        Attribute projectorEvent = event.asStartElement().
                getAttributeByName(new QName(XMLConstants.NULL_NS_URI,
                                               "projector"));
        if(projectorEvent != null) {
            String projector = projectorEvent.getValue();
...
```

StAX Example

- Whole programs:
  - `CountSeats_Stax_Stream`
    presents the usage of the low-level `XMLStreamReader`
  - `CountSeats_Stax_Event`
    presents the usage of `XMLEventReader`

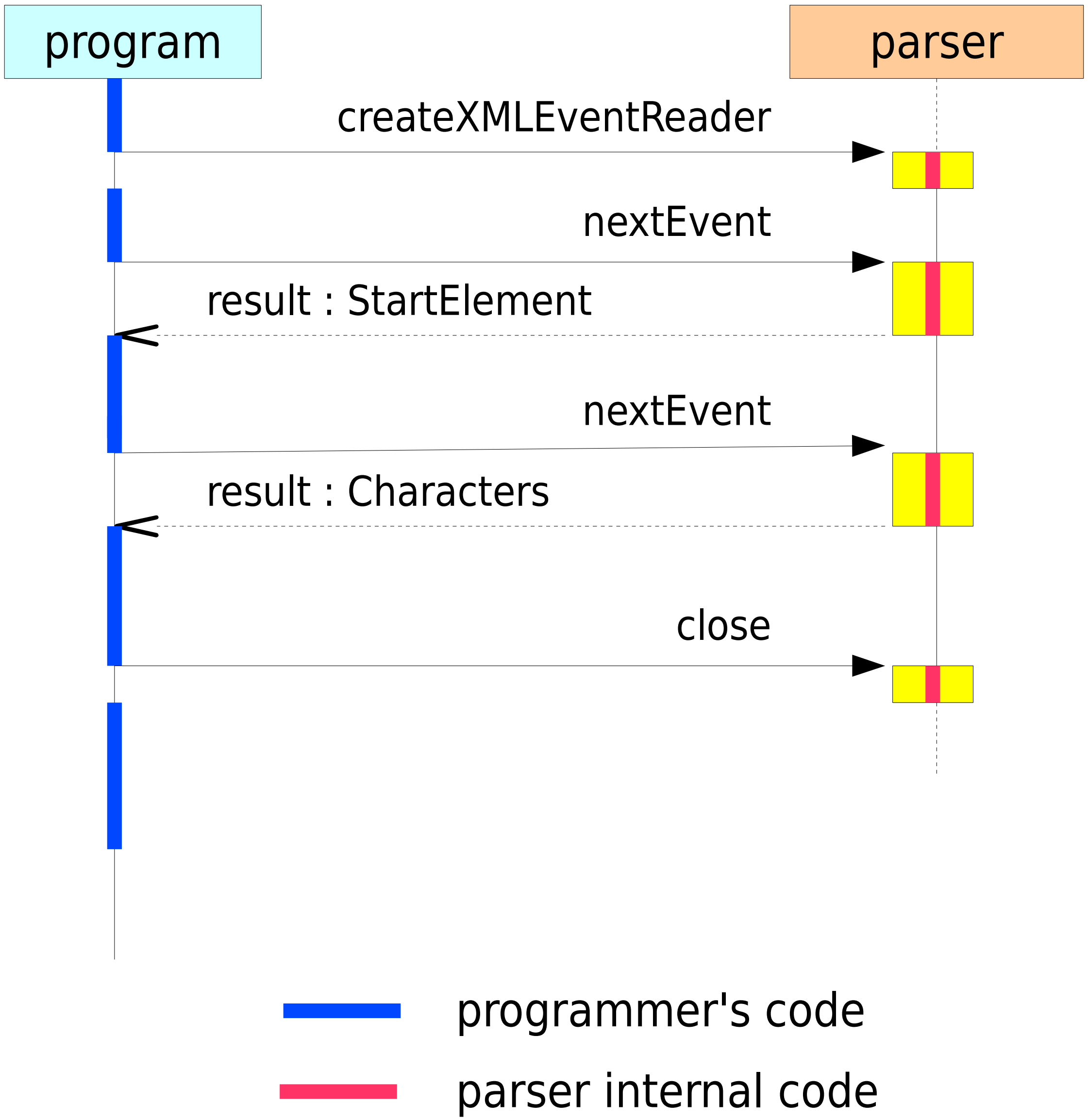
Control flow in SAX



🟦 programmer's code

🟥 parser internal code

Control flow in StAX



program

parser

createXMLEventReader

nextEvent

result : StartElement

nextEvent

result : Characters

close

programmer's code

parser internal code

StAX features

- API for reading documents:
  `XMLStreamReader`, `XMLEventReader`
- API for writing documents:
  `XMLStreamWriter`, `XMLEventWriter`
- Filters
  - simple definition of a filter: accept(Event): boolean
  - “filtered readers”

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 a data serialisation method

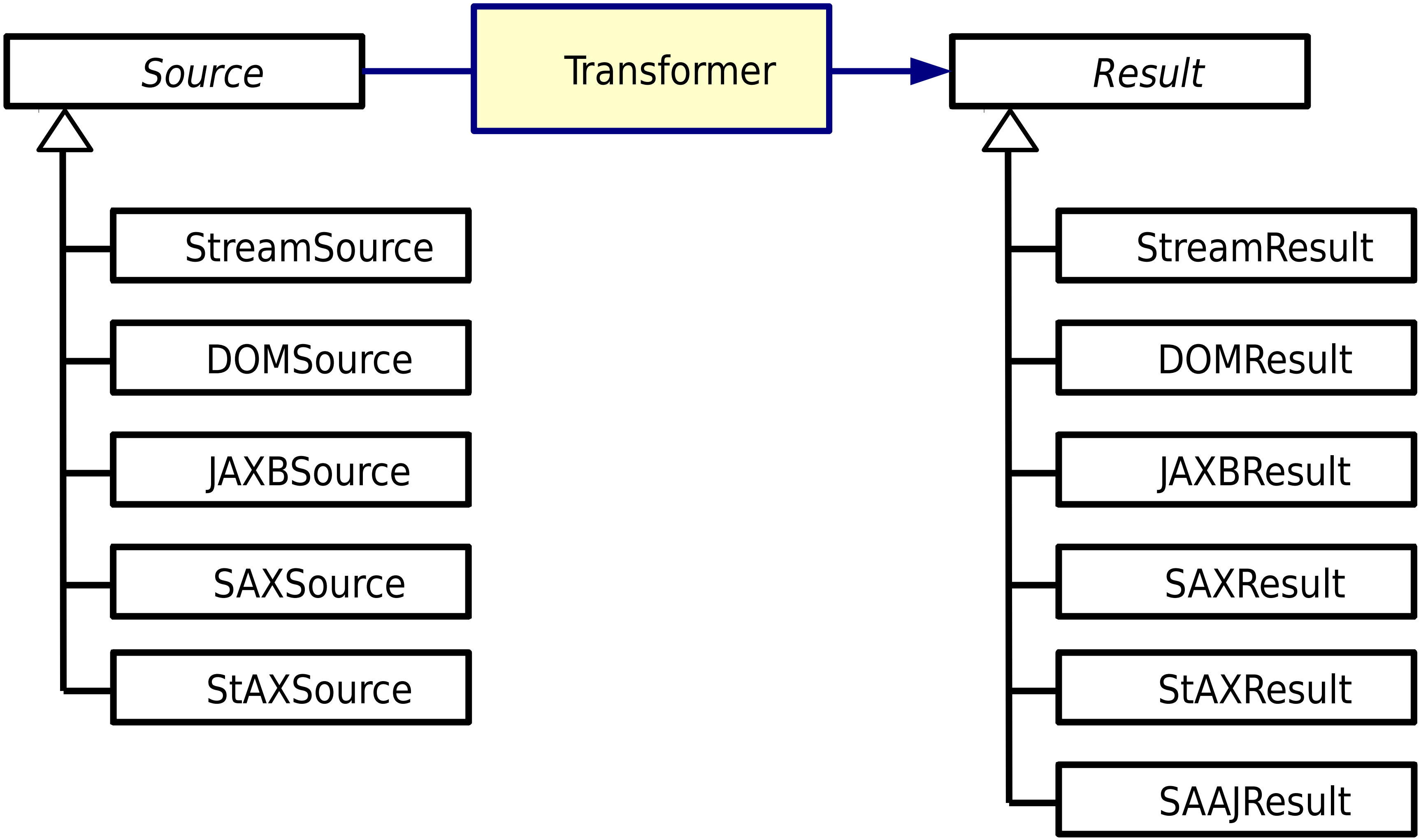
Which model to choose? (2)

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

Features of JAXP

- 3 models of XML documents in Java: DOM, SAX, StAX
  - Formally JAXB is a separate specification
- Reading and writing documents
- Transformations of XML documents (Transformers)
  - applying XSLT in our programs
  - translating internal form of representation
- XPath support
- Validation
  - against DTD (only during parsing)
  - against XML Schema (during parsing or using Validators)
  - against XML Schema 1.1, Relax NG, or other alternative standards – when implementation supports

Transformer: source and result



Applications of Transformers

- Simple:
  - invoking XSLT transformations from Java
  - changing internal representation of XML in our program
- Tricky:
  - parsing and writing documents, e.g. serialisation of a DOM tree
  - serialisation of modified (or generated) sequences of SAX events
  - (together with SAX filters) enabling “on-the-fly” processing of large XML documents

Editing XML documents

- More natural when whole document present in memory
  - DOM – generic API
  - JAXB – deep embedding of XML in application model
- Harder, but possible, using node-by-node processing
  - required when processing big documents while having little memory
  - suggested for big (“long and flat”) documents and simple local operations – then we can save substantial resources
  - StAX – possible using “writers”
    - IMO XMLEventWriter more convenient than XMLStreamWriter
  - SAX
    - no direct support for editing/writing
    - available indirect solution: SAX filters and Transformer

Validation

- Against DTD
  - setValidating(true) before parsing
- Against XML Schema (or other schema formats, if implementation supports)
  - setSchema(schema) before parsing
  - using Validator
- Validator API
  - validate(Source) – only checking of correctness
  - validate(Source, Result) – augmented document returned
    - not possible to use as Transformer – source and result must be of the same kind
    - (my private observation) – not always working as expected

Handling errors

- Most JAXP components (specifically SAX and DOM parsers, Validators)
  - may throw SAXException
  - signal errors through ErrorHandler events
- Interface ErrorHandler
  - 3 methods (and severity levels): warning, error, fatalError
  - registering with setErrorHandler allows to override default error handling
- Required to manually handle validation errors

XPath support in Java

- DOM XPath module implementation
  - org.w3c.dom.xpath
  - officially not a part of Java SE, but available in practice (by inclusion of Xerces in Oracle Java SE runtime)
- JAXP XPath API
  - javax.xml.xpath
  - most efficient when applied for documents in memory (DOM trees)
  - our examples show this solution
- Note: using XPath may significantly reduce developer's work, but the application may be less efficient (than if we used SAX, for example)