

9 XML & JSON

Programmeren 2 – Java

2017 - 2018

KdG Karel de Grote
Hogeschool

Kris Behiels

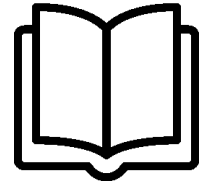
Jan De Rijke

Mark Goovaerts

Programmeren 2 - Java

1. Herhaling en Collections
2. Generics en documenteren
3. Annotations en Reflection
4. Testen en logging
5. Design patterns (deel 1)
6. Design patterns (deel 2)
7. Lambda's en streams
8. Persistentie (JDBC)
- 9. XML en JSON**
10. Threads
11. Synchronization
12. Concurrency





Voorlopig nog niet opgenomen in het e-book

Interessante links:

- XML: <http://tutorials.jenkov.com/java-xml/index.html>
- XML: http://www.tutorialspoint.com/java_xml/
- JSON: http://www.w3schools.com/js/js_json_intro.asp
- GSON: <https://github.com/google/gson/blob/master/UserGuide.md>

Agenda

1. Inleiding XML

- Voordelen XML
- Syntax XML
- XML tree



2. XML in Java

- Manuele parsing: DOM, StAX
- Automatische binding: JAXB

3. JSON

- JSON vs XML
- Syntax JSON
- Parsing met GSON

4. Streams

- Stream -> XML of JSON



Introducing XML

Inleiding XML

- **XML** = *eXtensible Markup Language*
- Tag-based zoals HTML
- Ontworpen voor data-transport en -storage
- In 1998 gedefinieerd door W3C
- XML document bevat data-elementen, gescheiden door tags →

```
<?xml version="1.0"?>
<Personnel>
  <Employee type="permanent">
    <Name>Johnny Depp</Name>
    <Id>3674</Id>
    <Age>34</Age>
  </Employee>
  <Employee type="contract">
    <Name>Tom Hanks</Name>
    <Id>3675</Id>
    <Age>25</Age>
  </Employee>
  <Employee type="permanent">
    <Name>Brad Pitt</Name>
    <Id>3676</Id>
    <Age>28</Age>
  </Employee>
</Personnel>
```

Voordelen XML



- XML-dataformaat heeft vele **voordelen**:
 - *Tekstformaat*: leesbaar en technologie-onafhankelijk
 - XML is *hiërarchisch* opgebouwd
 - eenvoudig te benaderen via standaard *parsers*
 - *transformeerbaar* naar andere formaten



- Ook een **nadeel**:
 - XML is relatief *uitgebreid* (redundante syntax) en *groot* dataformaat
 - daarom vaak omgezet naar JSON (zie verder)

Well-formed XML

- XML-formaat is gestandaardiseerd
 - Een document dat aan de syntax regels voldoet is **well-formed**

`<?xml version="1.0"?>`

declaration indien aanwezig op 1^e regel

`<!-- This is a comment -->`

Exact één **rootelement**

`<address>`

Elke **start tag** heeft ook een **end tag**

`<name>Kasper</name>`

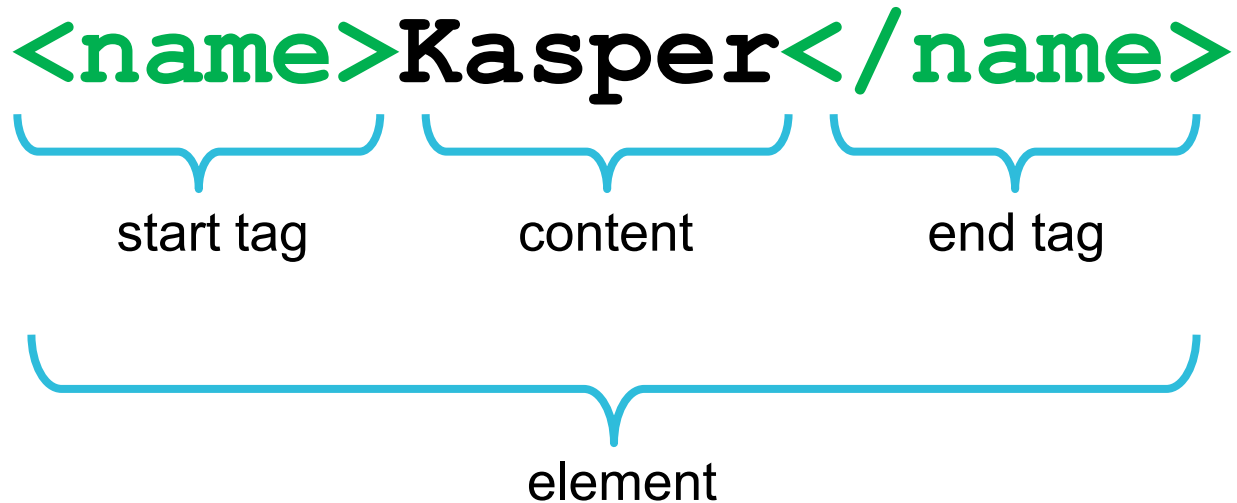
`<street>Kerkstraat 12</street>`

`<city code="2000">Antwerpen</city>`

`</address>`

Alle tags zijn volledig **genest**

XML Syntax : element & tag



XML Syntax : nested element

```
<user>
  <name>Kasper</name>
  <date-of-birth>23-10-2002</date-of-birth>
</user>
```

- Het eerste element van een XML file heet *de root*
- XML is hiërarchisch : elementen worden *genest*
- Code conventions : koppelteken in tag
- Commentaar: `<!-- Comment -->`

XML Syntax : gemengde inhoud

```
<user>
```

```
Ik ben Content
```

```
<name>Kasper</name>
```

```
Ik ook
```

```
<date-of-birth>23-10-2002</date-of-birth>
```

```
Ik ook
```

```
</user>
```

- Een element kan zowel content als geneste elementen bevatten
- Niet vaak gebruikt

XML Syntax : empty element


`<user/>`

- Wordt gebruikt wanneer het element verplicht is, maar er **geen content** voor bestaat.

XML Syntax : attribute

- Kan enkel in de start tag gezet worden:


```
<user number="12" valid="Yes">Kasper</user>
```



attribute

- Ook in een empty element:

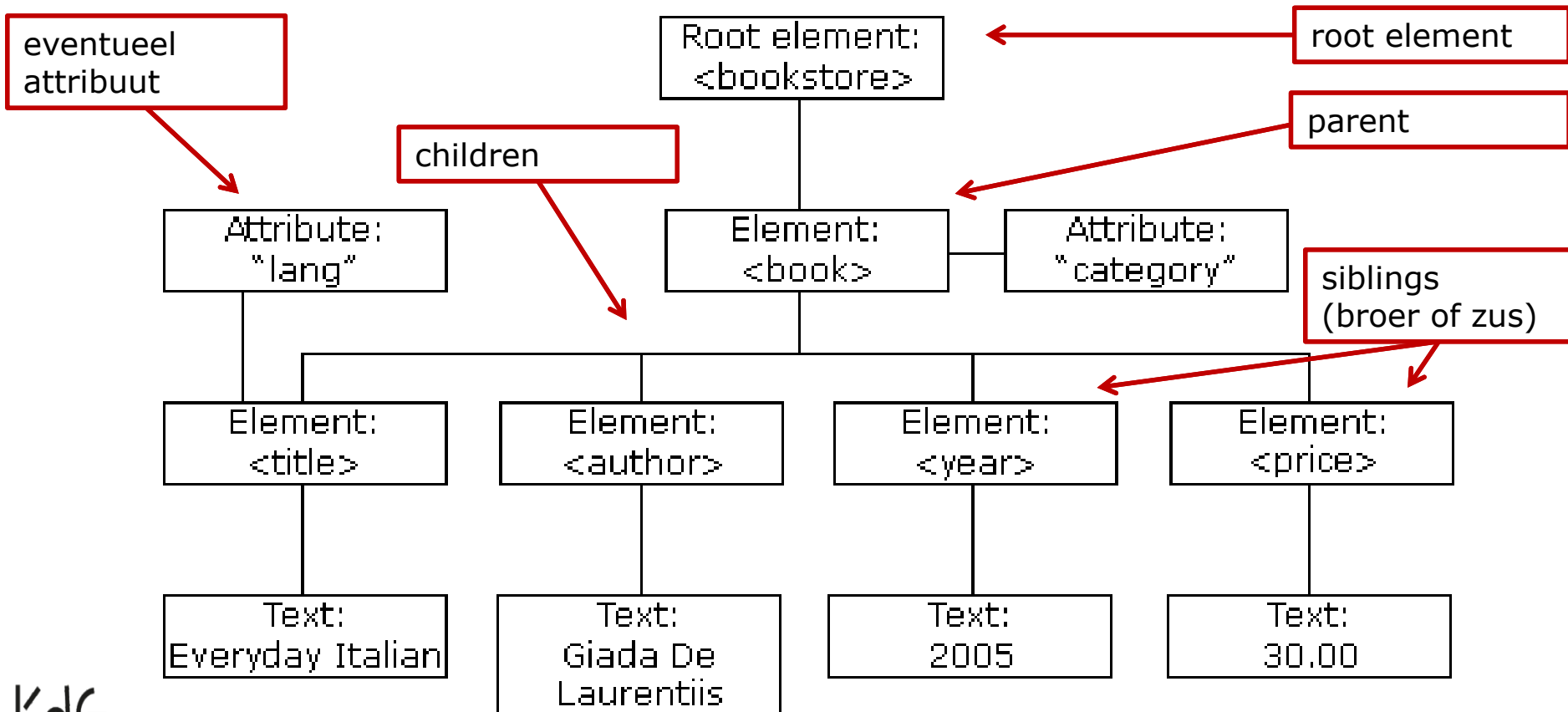
```
<user number="12" valid="Yes"/>
```



attribute

XML tree

XML document bevat boomstructuur: **XML tree**



XML tree

XML document gebaseerd op voorgaande XML tree:

```
<?xml version="1.0" encoding="UTF-8"?>
<bookstore>
  <book category="cooking">
    <title lang="en">Everyday Italian</title>
    <author>Giada De Laurentiis</author>
    <year>2005</year>
    <price>30.00</price>
  </book>
  <book category="children">
    <title lang="en">Harry Potter</title>
    <author>J K. Rowling</author>
    <year>2005</year>
    <price>29.99</price>
  </book>
  <book category="web">
    <title lang="en">Learning XML</title>
    <author>Erik T. Ray</author>
    <year>2003</year>
    <price>39.95</price>
  </book>
</bookstore>
```

eventueel
attribuut

root element

parent

children

siblings
(broer of zus)

XML schema

- Een **XML schema**:
 - beschrijft de **structuur** van een XML document
 - wordt gebruikt om een XML-file te **valideren**
 - wordt zelf ook in XML geschreven
 - heeft vaak de extensie **.xsd** (*XML Schema Definition*)
- Een document dat aan een schema voldoet is **valid**
 - Een document kan dus well-formed zijn, maar niet valid (het omgekeerde kan niet)
- Vanuit je Java code kan je het XML schema ophalen en gebruiken om je XML document te valideren

XML document: shiporder.xml

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

<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

XML schema: shiporder.xsd

```
<?xml version="1.0" encoding="UTF-8" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="shiporder">
    <xs:complexType> <xs:sequence>
      → <xs:element name="orderperson" type="xs:string"/>
        <xs:element name="shipto">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="name" type="xs:string"/>
              <xs:element name="address" type="xs:string"/>
              <xs:element name="city" type="xs:string"/>
              <xs:element name="country" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
        <xs:element name="item" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element name="title" type="xs:string"/>
              <xs:element name="note" type="xs:string" minOccurs="0"/>
              <xs:element name="quantity" type="xs:positiveInteger"/>
              <xs:element name="price" type="xs:decimal"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

shiporder.xml en shiporder.xsd

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

```
<shiporder orderid="889923"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="shiporder.xsd">
```

```
  <orderperson>John Smith</orderperson>
  <shipto>
```

```
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
```

```
  <item>
```

```
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
```

```
  <item>
```

```
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
```

```
</shiporder>
```

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

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema" >
```

```
  <xs:element name="shiporder">
```

```
    <xs:complexType>
```

```
      <xs:sequence>
```

```
        <xs:element name="orderperson" type="xs:string"/>
```

```
        <xs:element name="shipto">
```

```
          <xs:complexType>
```

```
            <xs:sequence>
```

```
              <xs:element name="name" type="xs:string"/>
```

```
              <xs:element name="address" type="xs:string"/>
```

```
              <xs:element name="city" type="xs:string"/>
```

```
              <xs:element name="country" type="xs:string"/>
            </xs:sequence>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
  <xs:element name="item" maxOccurs="unbounded">
```

```
    <xs:complexType>
```

```
      <xs:sequence>
```

```
        <xs:element name="title" type="xs:string"/>
```

```
        <xs:element name="note" type="xs:string"/>
```

```
        <xs:element name="quantity" type="xs:string"/>
```

```
        <xs:element name="price" type="xs:string"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Agenda

1. Inleiding XML

- Voordelen XML
- Syntax XML
- XML tree



2. XML in Java

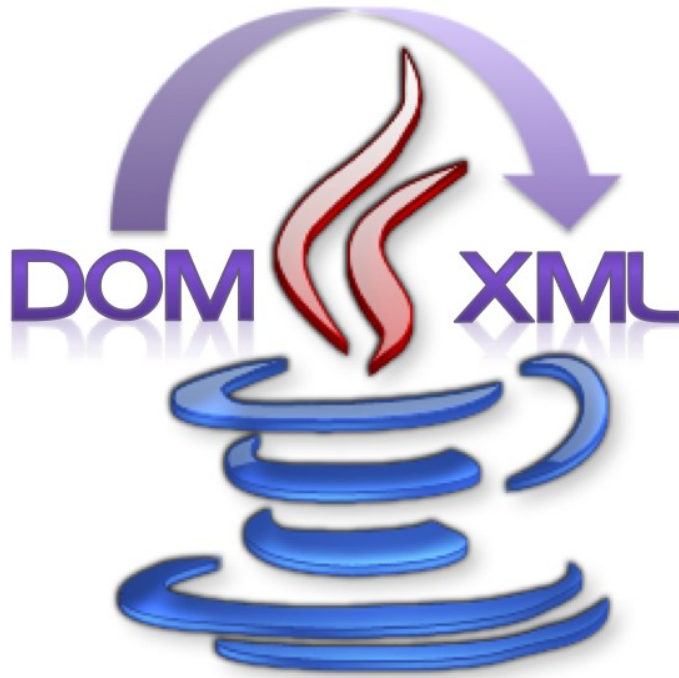
- Manuele parsing: DOM, StAX
- Automatische binding: JAXB

3. JSON

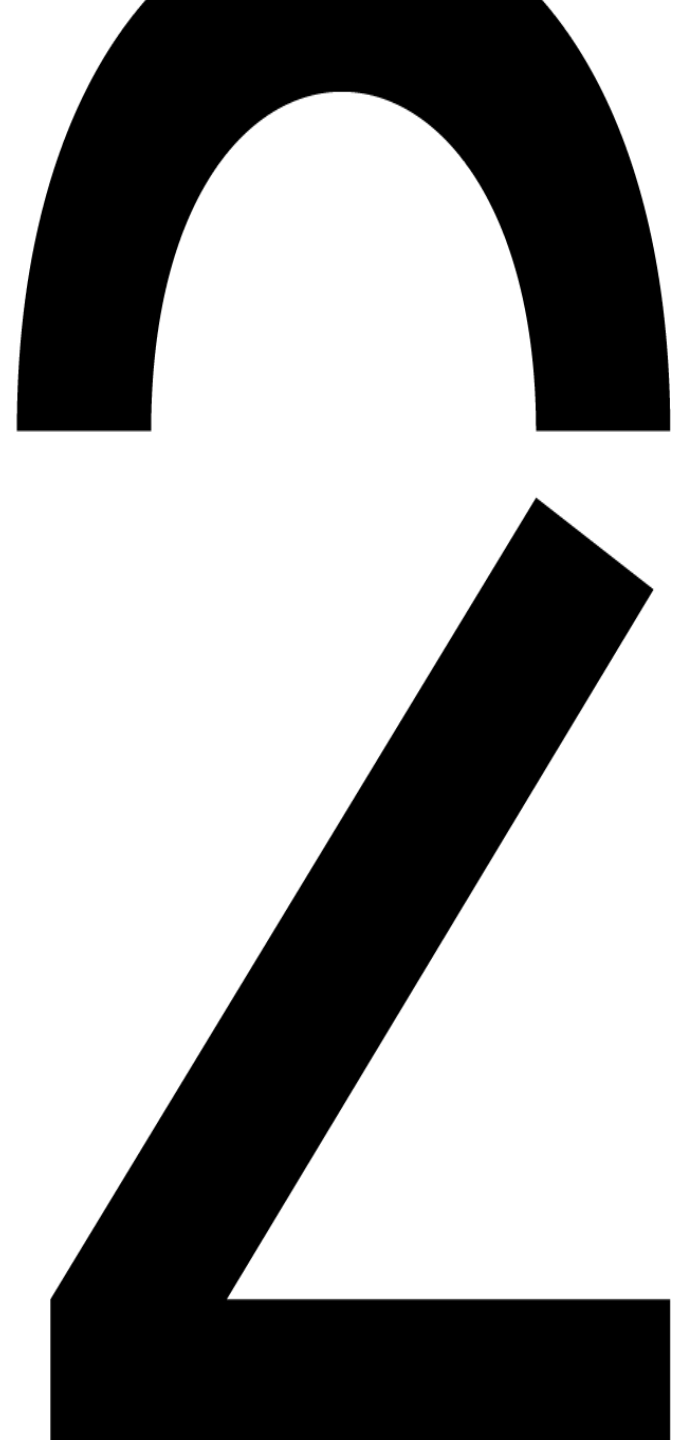
- JSON vs XML
- Syntax JSON
- Parsing met GSON

4. Streams

- Stream -> XML of JSON



XML in Java



XML in Java

- Via Java gaan we door het XML document om data te benaderen of te wijzigen.
→ **parsing XML**
- We bekijken 2 werkwijzen:
 - *Manuele parsing*: we benaderen zelf de data; tag per tag, element per element
 - *Automatische binding*: Java beans worden automatisch gemapt op een XML-file

XML parsers

- Er zijn verschillende **XML parsers**:

- **DOM** Parser

Het hele XML document wordt als hiërarchische XML tree in het geheugen geladen.

- SAX en **StAX** Parser

Het hele XML document wordt NIET in het geheugen geladen, maar *event-based* benaderd.

- XPath Parser

Het XML-document wordt *expression-based* benaderd.
XPath wordt gebruikt in combinatie met XSLT

worden niet
besproken

- DOM4J Parser

Een java library om XML, XPath and XSLT te parsen, gebruik makend van het Java Collections Framework

DOM: Document Object Model

- **W3C** standaard API
 - in meerdere programmeertalen (Java, EcmaScript...)
 - org.w3c.dom pakketten in JDK
- Elke XML tree wordt als een document (volledig bestand) benaderd.

DOM

- De belangrijkste **org.w3c.dom** interfaces:
 - **Document** bevat de hele *DOM tree*.
 - **Element** bevat een XML element.
 - **Attr** bevat het attribuut van een element.
 - **Text** bevat de tekst (content) van de XML tag.
 - **Comment** bevat de commentaar van een XML document.

1) Java → DOM → XML

- Stappen om vanuit Java XML te schrijven met **DOM**:
 1. Maak een document
 2. Creëer XML elementen met dit document
 3. Voeg attributes / text toe
 4. Voeg de XML elementen toe aan het document (root element) of aan andere XML elementen (child elementen)
 5. Zet het document om naar een DOMSource object
 6. Schrijf de DOMSource weg met een Transformer

Voorbeeld XML

- We willen het volgende XML-document aanmaken en vervolgens inlezen **via DOM**:

```
<?xml version="1.0" encoding="UTF-8"?>
<family>
  - <person roll-no="1">
      <firstname>Homer</firstname>
      <lastname>Simpson</lastname>
      <age>45</age>
    </person>
  - <person roll-no="2">
      <firstname>Marge</firstname>
      <lastname>Simpson</lastname>
      <age>42</age>
    </person>
  - <person roll-no="3">
      <firstname>Bart</firstname>
      <lastname>Simpson</lastname>
      <age>10</age>
    </person>
```

1) Java → DOM → XML

```
Document doc = DocumentBuilderFactory.newInstance()
    .newDocumentBuilder().newDocument();
Element rootElement = doc.createElement("family");
doc.appendChild(rootElement);
Element personElement = doc.createElement("person");
personElement.setAttribute("roll-no", "1");
Element firstnameElement = doc.createElement("firstname");
firstnameElement.setTextContent(person.getFirstName());
personElement.appendChild(firstnameElement);
Element lastnameElement = doc.createElement("lastname");
lastnameElement.setTextContent(person.getLastName());
personElement.appendChild(lastnameElement);
//...
DOMSource src = new DOMSource(doc);
Transformer xf = TransformerFactory.newInstance()
    .newTransformer();
xf.transform(src, new StreamResult(new File("simpsons.xml")));
```

Maak **Document**

Maak **Element**

Koppel rootelement aan Document

Voeg attribuut toe

Koppel child-element aan Element

Maak DOMsource

Maak **Formatter**



1) Java → DOM → XML: Pretty Printen

- Als het document door mensen gelezen moet worden kan je het **Pretty Printen** met de Formatter
- De Formatter gebruikt hiervoor **XSLT** (Extensible Stylesheet Language Transformation)

```
Transformer xf = TransformerFactory.newInstance()
    .newTransformer();

xf.setOutputProperty(OutputKeys.INDENT, "yes");
xf.setOutputProperty("{http://xml.apache.org/xslt}indent-
    amount", "2");
xf.transform(src, new StreamResult(new
    File("simpsons.xml")));
```

2) XML → DOM → Java

- Stappen om met DOM XML te parsen :
 1. Creëer een DocumentBuilder
 2. Creëer een document vanuit een file of stream
 3. Lees het root element
 4. Lees eventuele attributen
 5. Lees sub-elementen

2) XML → DOM → Java

```
DocumentBuilder builder = DocumentBuilderFactory.newInstance()
    .newDocumentBuilder();
Document doc = builder.parse(new File(file));
Element rootElement = doc.getDocumentElement();
NodeList personNodes = rootElement.getChildNodes();
for (int i = 0; i < personNodes.getLength(); i++) {
    if (personNodes.item(i).getNodeType() != Node.ELEMENT_NODE) {
        continue;
    }
    Element e = (Element) personNodes.item(i);
    System.out.println("Element:" + e.getNodeName());
    System.out.println("Attr:" + e.getAttribute("roll-no"));
    Element firstname =
        (Element) e.getElementsByTagName("firstname").item(0);
    System.out.println("firstname: " + firstname.getTextContent());
    //...
}
```

Maak DocumentBuider vanuit XML-file

Lees root-element

Lees enkel XML-elementen

Lees child element

Lees attribute

Lees content



- **StAX** (**St**reaming **A**PI for **X**ML) is een uitbreiding op **SAX** (**S**imple **A**PI for **X**ML)
- StAX werkt met XML document als een Stream (top -> bottom)
 - StAX leest Events van de Stream
 - StAX is een *pull* API, hij vraagt de stream om het volgende Event
- Via een *Event-reader* wordt er door het XML document geïtereerd
- DOM werkt steeds met volledige bestanden, StAX is veel performanter voor grote XML-bestanden



1) Java → StAX → XML

Vanuit Java een XML-document aanmaken met StAX:

- De klasse **XMLStreamWriter** met volgende interessante methoden:

`–writeStartElement`

`–writeCharacters`

`–writeEndElement`

`–writeAttribute`

Voorbeeld XML

- We willen het volgende XML-document aanmaken en vervolgens inlezen via **StAX**:

```
<?xml version="1.0" encoding="UTF-8"?>
<family>
  - <person roll-no="1">
    <firstname>Homer</firstname>
    <lastname>Simpson</lastname>
    <age>45</age>
  </person>
  - <person roll-no="2">
    <firstname>Marge</firstname>
    <lastname>Simpson</lastname>
    <age>42</age>
  </person>
  - <person roll-no="3">
    <firstname>Bart</firstname>
    <lastname>Simpson</lastname>
    <age>10</age>
  </person>
```

1) Java → StAX → XML

```
FileWriter file = new FileWriter("simpsons.xml"));
XMLStreamWriter xmlStreamWriter = XMLOutputFactory.newInstance()
    .createXMLStreamWriter(file);
xmlStreamWriter.writeStartDocument();
xmlStreamWriter.writeStartElement("family");
xmlStreamWriter.writeStartElement("person");
xmlStreamWriter.writeAttribute("roll-no", "1");
xmlStreamWriter.writeStartElement("firstname");
xmlStreamWriter.writeCharacters("Homer");
xmlStreamWriter.writeEndElement(); // </firstname>
//...
xmlStreamWriter.writeEndElement(); // </person>
xmlStreamWriter.writeEndElement(); // </family>
xmlStreamWriter.writeEndDocument();
xmlStreamWriter.close();
```

Maak **FileWriter**

koppel aan **XMLStreamWriter**

Elementen toevoegen

end-tags toevoegen

XMLStreamWriter sluiten



2) XML → StAX → Java

Een XML-document parsen met StAX:

- De klasse **XMLStreamReader** met volgende interessante methoden:

- next**

- hasNext**

- nextEvent**

- levert een **Event** op dat je verder kan onderzoeken of het een starttag, endtag, attribuut, content of commentaar is

2) XML → StAX → Java

```
XMLStreamReader eventReader = XMLInputFactory.newInstance()
    .createXMLStreamReader(new FileReader("simpsons.xml"));
while(eventReader.hasNext()) {
    XMLEvent event = eventReader.nextEvent();
    switch(event.getEventType()) {
        case XMLStreamConstants.START_ELEMENT:
            StartElement startElement = event.asStartElement();
            String tagName = startElement.getName().getLocalPart();
            Iterator<Attribute> attributes =
                startElement.getAttributes();
            String rollNo = attributes.next().getValue();
            //...
        case XMLStreamConstants.CHARACTERS:
            Characters characters = event.asCharacters();
            System.out.println("First Name: " + characters.getData());
            //...
```

Maak **XMLStreamReader**

itereren door **XMLStreamReader**

Events onderzoeken





- **JAXB** = "**J**ava **A**rchitecture for **X**ML **B**inding"
- JAXB-tools om objecten naar XML om te zetten en omgekeerd:
 - **Marshalling**: Java → XML
 - **Unmarshalling**: XML → Java
- Tools in de package: `javax.xml.bind`
 - vanaf JDK 9 niet meer gebundeld met Java SE



De klasse Product

```
import javax.xml.bind.annotation.XmlRootElement;
```

```
@XmlRootElement
```

← Annotatie plaatsen

```
public class Product {  
    private String code;  
    private String name;  
    private double price;
```

```
    Product() {...}
```

← Er moet een default constructor (zonder parameters) zijn

```
    Product(String code, String name, double price) {...}
```

```
    //getters & setters...
```

Default: alle publieke attributen en getter/setter paren worden van/naar XML omgezet

- **POJO** (Plain old Java Object): Een object dat niet afhangt van een framework via superklassen of interfaces.
- **Java Bean**: een klasse met een constructor zonder parameters en publieke getters en setters voor te behandelen attributen.

JAXB: marshal: Java → XML



```
public class MarshalDemo {  
    public static void main(String[] args) throws Exception {  
        JAXBContext context = JAXBContext.newInstance(Product.class);  
  
        Marshaller m = context.createMarshaller();  
        m.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, true);  
  
        Product product=new Product("W11","Widget Number One", 300.0);  
  
        m.marshal(product, new File("product.xml"));  
        System.out.println("File created");  
    }  
}
```



JAXB: unmarshal: XML → Java



```
public class UnmarshalDemo {  
    public static void main(String[] args) {  
        try {  
            JAXBContext jc = JAXBContext.newInstance(Product.class);  
            Unmarshaller u = jc.createUnmarshaller();  
  
            File f = new File("product.xml");  
  
            Product product = (Product) u.unmarshal(f);  
            System.out.println("Na unmarshal:");  
            System.out.println(product);  
        }  
        catch (JAXBException e) {  
            e.printStackTrace();  
        }  
    }  
}
```



JAXB: XML

- Resultaat in XML:

```
<?xml version="1.0" encoding="UTF-8" standalone="true"?>
<product>
  <code>WI1</code>
  <name>Widget Number One</name>
  <price>300.0</price>
</product>
```

JAXB voorbeeld 2: Museums

`@XmlElement`

```
public class Museums {  
    private List<Museum> museumList = new ArrayList<>();  
    public void setMuseumList(List<Museum> museumList) {  
        this.museumList = museumList;  
    }  
    public List<Museum> getMuseumList() {  
        return museumList;  
    }  
    public void add(Museum museum) {this.museumList.add(museum);  
    }  
}
```

JAXB voorbeeld 2: Museum

```
public class Museum {  
    private String name;  
    private String city;  
    private Boolean childrenAllowed;  
    //...  
    public void setChildrenAllowed(Boolean childrenAllowed) {  
        this.childrenAllowed = childrenAllowed;  
    }  
    //...  
}
```

JAXB voorbeeld 2: Exhibition

```
public class Exhibition {  
    private String name;  
    private LocalDate from;  
    private LocalDate to;  
    private List<String> artists;  
    //...  
    public void setFrom(LocalDate from) {  
        this.from = from;  
    }  
    //...  
}
```

Gegenereerde XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<museums>
  <museumList>
    <city>Madrid</city>
    <name>Prado Museum</name>
    <permanent>
      <artists>Velazquez</artists>
      <artists>Goya</artists>
      <artists>Zurbaran</artists>
      <artists>Tiziano</artists>
      <from/>
      <name>Permanent Exhibition - Prado Museum</name>
      <to/>
    </permanent>
    <special>
      <artists>Matisse</artists>
      <from/>
      <name>Game of Bowls (1908), by Henri Matisse</name>
      <to/>
    </special>
  </museumList>
  <museumList>
    <childrenAllowed>true</childrenAllowed>
    <city>Madrid</city>
    <name>Reina Sofia Museum</name>
    <permanent>
      <artists>Picasso</artists>
      <artists>Dali</artists>
      <artists>Miro</artists>
      <from/>
      <name>Permanent Exhibition - Reina Sofia Museum</name>
      <to/>
    </permanent>
  </museumList>
</museums>
```

JAXB – complexer

- JAXB gebruikt *convention over configuration*. De enige vereiste annotatie is **@XmlElement**. JAXB zal dan objecten (en alle attribuut objecten) van/naar XML omzetten
- We bespreken hoe we een specifiek XML bestand (zie volgende slide) kunnen bekomen :
 - Specifieke annotaties in de Java bean om de omzetting te beïnvloeden
 - Omgaan met een datum-formaat
 - Werken met een List



Gewenste XML

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<musea>
  <museum children-allowed="false">
    <name>Prado Museum</name>
    <city>Madrid</city>
    <permanent-exhibition>
      <name>Permanent Exhibition - Prado Museum</name>
      <artist>Velazquez</artist>
      <artist>Goya</artist>
      <artist>Zurbaran</artist>
      <artist>Tiziano</artist>
      <from>2016-11-30</from>
      <to>2018-06-15</to>
    </permanent-exhibition>
    <special-exhibition>
      <name>Game of Bowls (1908), by Henri Matisse</name>
      <artist>Mattise</artist>
      <from>2018-01-01</from>
      <to>2018-09-30</to>
    </special-exhibition>
  </museum>
  <museum children-allowed="true">
    <name>Reina Sofia Museum</name>
    <city>Madrid</city>
    <permanent-exhibition>
      <name>Permanent Exhibition - Reina Sofia Museum</name>
      <artist>Picasso</artist>
      <artist>Dali</artist>
      <artist>Miro</artist>
      <from>2017-11-01</from>
      <to>2018-01-01</to>
    </permanent-exhibition>
  </museum>
</musea>
```


De klasse Museums

```
@XmlElement(name = "musea")
public class Museums {
    private List<Museum> museumList = new ArrayList<>();

    @XmlElement(name = "museum")
    public void setMuseumList(List<Museum> museumList) {
        this.museumList = museumList;
    }

    public List<Museum> getMuseumList() {
        return museumList;
    }

    public void add(Museum museum) {
        this.museumList.add(museum);
    }
}
```

naam van element

Deze klasse bevat
een List van
Museum-objecten



De klasse Museum

```
import javax.xml.bind.annotation.*;
```

```
@XmlType(propOrder = {"name", "city", "permanent", "special"})
```

```
public class Museum {
```

```
    private String name;
```

```
    private String city;
```

```
    private Boolean childrenAllowed;
```

```
    //...
```

Volgorde van child-
elementen



```
@XmlAttribute(name = "children-allowed")
```

attribuut ipv element



```
public void setChildrenAllowed(Boolean childrenAllowed) {
```

```
    this.childrenAllowed = childrenAllowed;
```


```
}
```

```
//...
```

De klasse Exhibition

```
@XmlType(propOrder = {"name", "artists", "from", "to"})
public class Exhibition {
    private String name;
    private LocalDate from;
    private LocalDate to;
    private List<String> artists;
    //...
    @XmlJavaTypeAdapter(LocalDateAdapter.class)
    public void setFrom(LocalDate from) {
        this.from = from;
    }
}
```

Onze eigen omzetting:
LocalDateAdapter



De klasse LocalDateAdapter

```
import javax.xml.bind.annotation.adapters.XmlAdapter;
import java.time.LocalDate;

public class LocalDateAdapter extends XmlAdapter<String, LocalDate> {

    public LocalDate unmarshal(String myString) throws Exception {
        return LocalDate.parse(myString);
    }

    public String marshal(LocalDate myDate) throws Exception {
        return myDate.toString();
    }
}
```

LocalDate heeft geen setters (geen JavaBean):

JAXB genereert default een leeg `</from>` element

We doen hier dus ZELF de omzetting `String` → `LocalDate` en omgekeerd.

XML verificatie

- Als de XML die je leest niet *well-formed* is geven JAXB (en de andere API's) altijd een exception
- Je kan ook nagaan of XML *valid* is ahv een **XML-schema**
 - Je kan niet enkel checken of de juiste elementen aanwezig zijn, maar ook hoeveel keer ze voorkomen, of de waarden geldig zijn...
- In JAXB
 - geef je het schema mee
 - Als je validity fouten wil loggen of erop reageren moet je ook een event handler toevoegen, met de code die dit doet
 - Als de event handler **true** (default) teruggeeft gaat de marshalling/unmarshalling verder. Als de event handler **false** teruggeeft gooit JAXB een Exception

XML verificatie: voorbeeld schema

- ```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
 <xs:element name="customer">
 <xs:complexType>
 <xs:sequence>
 <xs:element name="name">
 <xs:simpleType >
 <xs:restriction base="xs:string">
 <xs:maxLength value="5"/>
 </xs:restriction>
 </xs:simpleType>
 </xs:element>
 <xs:element name="phone-number" maxOccurs="2"/>
 </xs:sequence>
 </xs:complexType>
 </xs:element>
</xs:schema>
```

XML

```
<customer>
 <name>Jane</name>
 <phone-number>123</phone-number>
 <phone-number>456</phone-number>
</customer>
```

Naam is maximum 5 tekens

Customer heeft maximaal 2 telefoonnummers



# XML schema validatie met JAXB

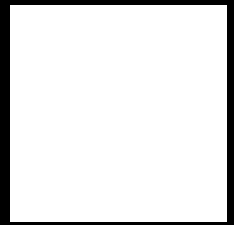
```
SchemaFactory sf = SchemaFactory.newInstance
 (XMLConstants.W3C_XML_SCHEMA_NS_URI);
Schema schema = sf.newSchema(new File("customer.xsd"));
JAXBContext jc = JAXBContext.newInstance(Customer.class);
Unmarshaller unmarshaller = jc.createUnmarshaller();
unmarshaller.setSchema(schema);
unmarshaller.setEventHandler(event -> {
 //...logging
});
Customer customer = (Customer) unmarshaller.unmarshal
 (new File("input.xml"));
System.out.println("\n" + customer);
```

← hier eventuele logging indien XML document niet valid is.



# Opdrachten

---



- Groeiproject
  - module 8  
(deel 1 en 2: "XML")



- Opdrachten op BB
  - Opdracht "Apen 1"  
(XML met DOM)
  - Opdracht "Apen 2"  
(XML met StAX en JAXB)





# Agenda

---

## 1. Inleiding XML

- Voordelen XML
- Syntax XML
- XML tree



## 2. XML in Java

- Manuele parsing: DOM, StAX
- Automatische binding: JAXB

## 3. JSON

- JSON vs XML
- Syntax JSON
- Parsing met GSON

## 4. Streams

- Stream -> XML of JSON

{JSON}



# {JSON}

---

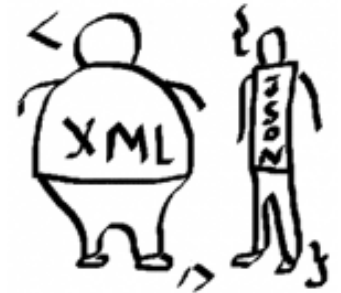
- **JSON** staat voor **J**ava**S**cript **O**bject **N**otation
- Vergelijkbaar met XML maar toch verschillen:
  - kortere notatie (geen end-tags)
  - geen attributen
  - geen schema (validatie)
  - sneller
  - eenvoudiger
- <http://www.json.org/>



# XML vs JSON

**XML**

```
<employees>
 <employee>
 <firstName>John</firstName> <lastName>Doe</lastName>
 </employee>
 <employee>
 <firstName>Anna</firstName> <lastName>Smith</lastName>
 </employee>
 <employee>
 <firstName>Peter</firstName> <lastName>Jones</lastName>
 </employee>
</employees>
```



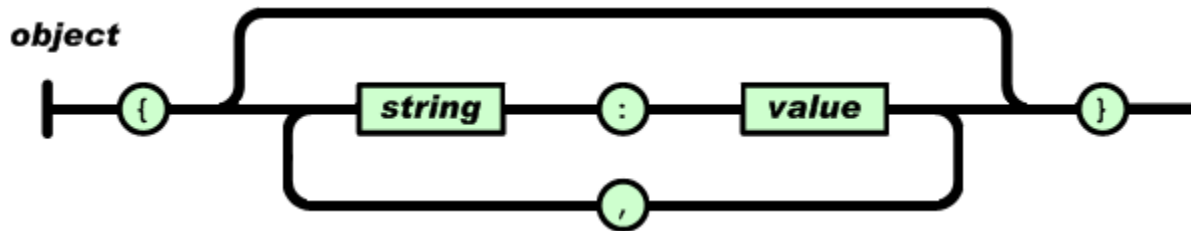
**JSON**

```
{ "employees": [
 { "firstName": "John", "lastName": "Doe" },
 { "firstName": "Anna", "lastName": "Smith" },
 { "firstName": "Peter", "lastName": "Jones" }
] }
```



# JSON object

- JSON object bestaat uit meerdere **name – value** pairs:



`{ "firstName" : "John" , "lastName" : "Doe" }`

**name** altijd  
tussen ""

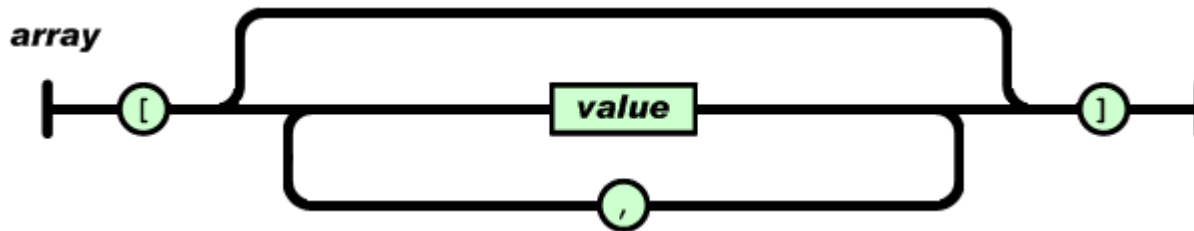
mogelijke **values**:

- getal (int, double)
- string (tussen "")
- boolean
- array (tussen [ ])
- object (tussen { })
- null

**Accolades**  
omgeven het  
object

# JSON array

- JSON objecten kunnen gegroepeerd worden in een **array**:



`"employees": [`

```
 { "firstName": "John", "lastName": "Doe" },
 { "firstName": "Anna", "lastName": "Smith" },
 { "firstName": "Peter", "lastName": "Jones" }
```

`]`

**naam** van  
de array

**komma's** om  
te scheiden

**Rechte haken**  
omgeven de  
array

# JSON en GSON

---

- **GSON** is een van de meest gebruikte libraries om JSON naar Java objecten om te zetten en omgekeerd.
- Alternatieven:
  - EclipseLink MOXy is een JAXB provider die ook JSON leest/schrijft
  - Java EE8 bevat JSON-B standaard JSR 367
- library: *gson-2.5.jar*
- `import google.gson.*` in je source code
- Annotaties en constructor zonder parameters zijn niet verplicht
- Bronnen:
  - <https://github.com/google/gson>
  - <https://github.com/google/gson/blob/master/UserGuide.md>

# GSON annotations

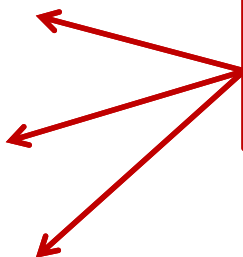
```
import com.google.gson.annotations.SerializedName;
```

```
import java.util.Objects;
```

```
public class Box {
 @SerializedName("breedte")
 private int width;

 @SerializedName("hoogte")
 private int height;

 @SerializedName("diepte")
 private int depth;
```



Zo kan je andere **tags** in je JSON gebruiken dan de naam van het attribuut

```
{"breedte":30,"hoogte":15,"diepte":20} in plaats van :
```

```
{"width":30,"height":15,"depth":20}
```





# Java → JSON → Java

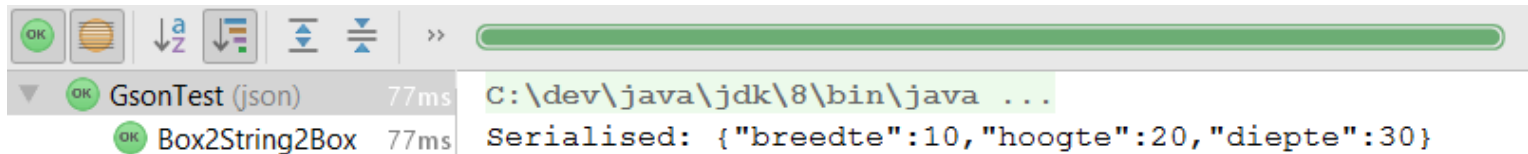
```
import com.google.gson.Gson;
import com.google.gson.GsonBuilder;
import org.junit.*;

public class GsonTest {
 @Test
 public void Box2String2Box() {
 GsonBuilder builder = new GsonBuilder();
 Gson gson = builder.create();

 Box box = new Box(10, 20, 30);

 String jsonString = gson.toJson(box);
 System.out.printf("Serialised: %s\n", jsonString);

 Box otherBox = gson.fromJson(jsonString, Box.class);
 assertEquals("Not same box", box, gson.fromJson(jsonString,
 Box.class));
 }
}
```



## List → JSON → file

```
List<Person> family = Arrays.asList(
 new Person("Homer", "Simpson", 45),
 new Person("Marge", "Simpson", 42),
 //...);
GsonBuilder builder = new GsonBuilder()
Gson gson = builder.setPrettyPrinting()
 .create();
String jsonString = gson.toJson(family)
System.out.printf("Serialised:\n\t%s\n"
 jsonString);
try (FileWriter jsonWriter =
 new FileWriter("simpsons.json")) {
 jsonWriter.write(jsonString);
} catch (FileNotFoundException |
 IOException e) { //...}
```

`setPrettyPrinting()`: voeg toe als je je bestand leesbaarder wil maken voor mensen

Serialised:

```
[
 {
 "firstName": "Homer",
 "lastName": "Simpson",
 "age": 45
 },
 {
 "firstName": "Marge",
 "lastName": "Simpson",
 "age": 42
 },
 {
 "firstName": "Bart",
 "lastName": "Simpson",
 "age": 10
 },
 ...
]
```



# file → JSON → array → List

```
try (BufferedReader data = new BufferedReader(
 new FileReader("simpsons.json"))) {
```

```
 Person[] personArray = gson.fromJson(data, Person[].class);
 List<Person> otherList = Arrays.asList(personArray);
```

```
 System.out.println("Deserialised:");
 for (Person person : otherList) {
 System.out.println("\t" + person);
 }
```

```
} catch (FileNotFoundException e) {
```

```
 Deserialised:
```

```
 Person{firstName='Homer', lastName='Simpson', age=45}
 Person{firstName='Marge', lastName='Simpson', age=42}
 Person{firstName='Bart', lastName='Simpson', age=10}
 Person{firstName='Lisa', lastName='Simpson', age=8}
 Person{firstName='Maggie', lastName='Simpson', age=1}
```

```
}
```



# Agenda

---

## 1. Inleiding XML

- Voordelen XML
- Syntax XML
- XML tree



## 2. XML in Java

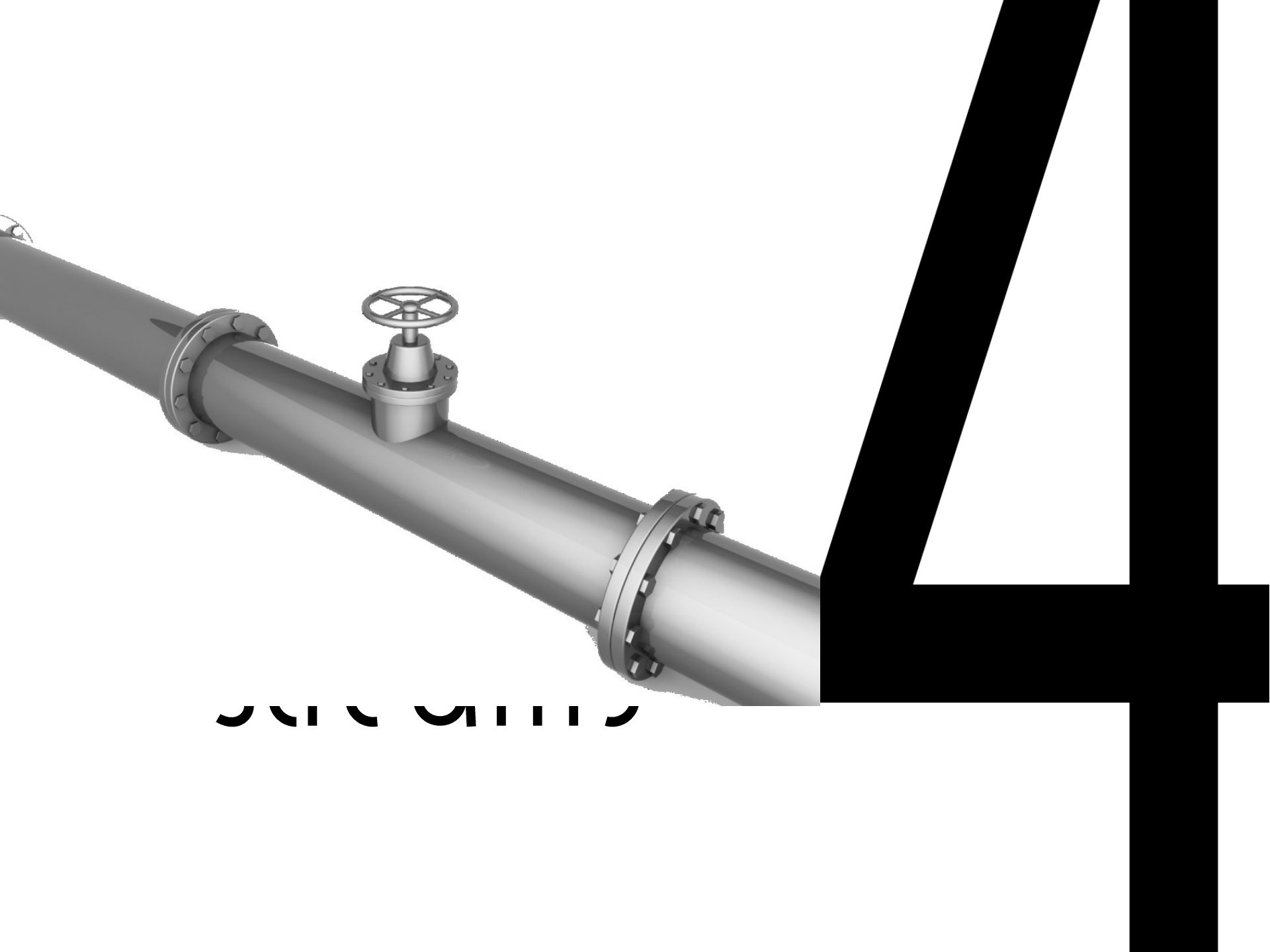
- Manuele parsing: DOM, StAX
- Automatische binding: JAXB

## 3. JSON

- JSON vs XML
- Syntax JSON
- Parsing met GSON

## 4. Streams

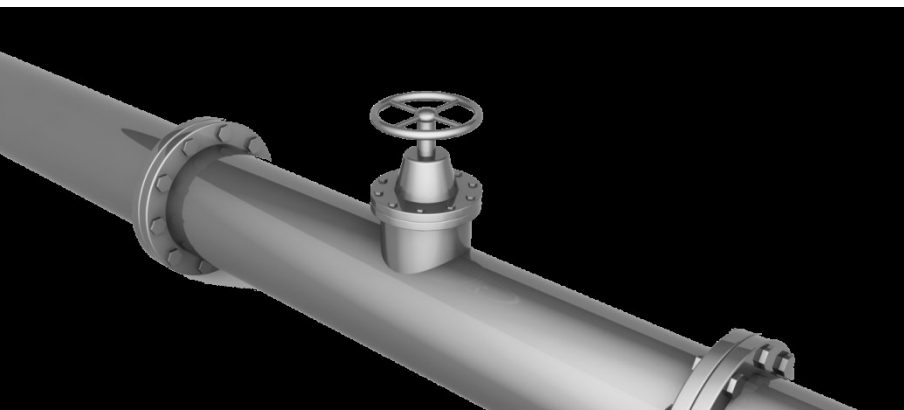
- Stream -> XML of JSON



# Streams

---

- De belangrijkste nieuwe features in Java SE8: **lambda's** en **streams**
- In deze sectie gebruiken we streams gebruiken om data uit een collection om te zetten naar een XML / JSON formaat **zonder XML-libraries**



# 1) List → Stream → XML

*//Testdata:*

```
List<Person> familyList = Arrays.asList(
 new Person("Homer", "Simpson", 45),
 new Person("Marge", "Simpson", 42),
 new Person("Bart", "Simpson", 10),
 new Person("Lisa", "Simpson", 8),
 new Person("Maggie", "Simpson", 1)
);
```

*//1) Voornamen van de ouders (age>40) als XML*

```
System.out.println("<family data='Simpsons'>" +
 familyList.stream()
 .filter(it -> it.getAge() > 40) ← Predicate
 .map(it -> "<person>" + it.getFirstName() + "</person>")
 .reduce("", String::concat)
 + "</family>");
```

*Voornamen van de ouders (XML):*

```
<family data='Simpsons'>
 <person>Homer</person><person>Marge</person>
</family>
```



## 2) List → Stream → XML v2

//Testdata:

```
List<Person> public String toXML() {
 new E return "<person>"
 new E + "<firstname>" + firstName + "</firstname>"
 new E + "<lastname>" + lastName + "</lastname>"
 new E + "<age>" + age + "</age>"
 new E + "</person>";
 new E }
);
```

methode **toXML** in  
klasse **Person**

//1) Voornamen van de ouders (age>40) als XML

```
System.out.println("<family data='Simpsons'>" +
 familyList.stream()
 .filter(it -> it.getAge() > 40)
 .map(Person::toXML)
 .reduce("", String::concat)
 + "</family>");
```

Volledige data van de ouders (XML):

```
<family data='Simpsons'>
<person><firstname>Homer</firstname><lastname>Sim
pson</lastname><age>45</age></person>
... </family>
```





### 3) List → Stream → JSON

//Testdata:

```
List<Person> public String toJSON() {
 new E return "{" +
 new E "firstName: \"" + firstName + "\", " +
 new E "lastName: \"" + lastName + "\", " +
 new E "age: " + age + " " +
 new E "}";
 new E }
);
```

methode toJSON  
in klasse Person

//3) Json:

```
System.out.println(familyList.stream()
 .filter(it -> it.getAge() > 40)
 .map(Person::toJSON)
 .collect(Collectors.joining(", ", "[", "]"));
```

Volledige data ouders(JSON):  
[{firstName: "Homer", lastName: "Simpson", age: 45 },  
{firstName: "Marge", lastName: "Simpson", age: 42 }]



## 4) List → Stream → sorted JSON

*//Testdata:*

```
List<Person> familyList = Arrays.asList(
 new Person("Homer", "Simpson", 45),
 new Person("Marge", "Simpson", 42),
 new Person("Bart", "Simpson", 10),
 new Person("Lisa", "Simpson", 8),
 new Person("Maggie", "Simpson", 1)
);
```

*//4) Json gesorteerd:*

```
System.out.println(familyList.stream()
 .sorted(Comparator.comparing(Person::getFirstName))
 .map(Person::toJSON)
 .collect(Collectors.joining(", ", "[", "]"));
```

Volledige data gesorteerd op voornaam (JSON):

```
[{firstName: "Bart", lastName: "Simpson", age: 10 },
{firstName: "Homer", lastName: "Simpson", age: 45 },
...]
```



# Opdrachten

---

- Groeiproject
  - module 8  
(deel 3: "JSON")
- Opdrachten op BB
  - Opdracht "Apen 3"  
(JSON met GSON)
- Zelftest op BB

