Basics of programming 3

Structured data representations

Introduction

- Objects are in memory
  - how to transfer them
  - how to store them in files
- Structured representations
  - Serialization
  - XML
  - JSON

Serialization basics

- Problem
  - let’s save our objects and later read them back
  - what should be stored?
    - objects’ attributes
    - associations
    - static fields?
- Simple solution: *serialization*
  - built-in Java feature
  - converts objects’ data and their associations to and from byte-streams

Serialization concepts

- Serialization
  - converting objects into binary form (byte stream)
    - also called marshalling, deflating
  - binary data can be stored, transmitted, etc.
- Deserialization
  - converting binary data (byte stream) into objects
    - also called unmarshalling, inflating
  - binary data can be read from a file, got from a network connection, etc.

Serialization example: write

```
import java.io.Serializable;
public class SerializableClass implements Serializable
{ ... }
```

```
//import java.io.*;
...
SerializableClass ser = ...;
try {
      FileOutputStream f =
                  new FileOutputStream(”filename”);
      ObjectOutputStream out =
                  new ObjectOutputStream(f);
      out.writeObject(ser);
      out.close();
}
catch(IOException ex) { ... }
```

Serialization example: read

```
//import java.io.*;
...
SerializableClass ser2;
try {
        FileInputStream f =
                        new FileInputStream("filename");
        ObjectInputStream in =
                        new ObjectInputStream(f);
        ser2 = (SerializableClass)in.readObject();

        in.close();

} catch(IOException ex) {
        ...
} catch(ClassNotFoundException ex) {
        ...
}
```

Rules of serialization

- Only classes implementing interface *Serializable* can be serialized
  - if superclass implements, it's OK
  - arrays, String, Integer, Double, etc. OK
- Interface *Serializable*
  - no methods
  - just formal notification for the compiler
- *Not serializable*
  - *Object, Socket, Input/OutputStream, System, etc.*

Rules of serialization

- What is serialized?
  - primitive attributes
  - serializable attributes
    - recursively
- What is not serialized?
  - static fields
  - *transient* fields

```
public class Serial implements Serializable {
       transient private String secret;
       private String other;
       ...
```

Serialization process

- Serialization is recursive
  - how to avoid cyclic graphs?
  - object's data are written only once
  - consecutively only reference is written

```
out.writeObject(a);
out.writeObject(b);
out.writeObject(a); // only reference!
```

- Serialization of inherited members
  - starts from topmost serializable superclass

Serialization process 2

- Let's have the following classes!

```
class Student implements
    Serializable {
  String name;
  University uni;
  Address a;
  double average;
  // ctr, set, get
}
```

```
class Address implements
    Serializable {
  String country,
    city, street;
  // ctr, set, get
}
```

```
class University implements
    Serializable {
  String name;
  Address a;
  // ctr, set, get
}
```

Serialization process 3

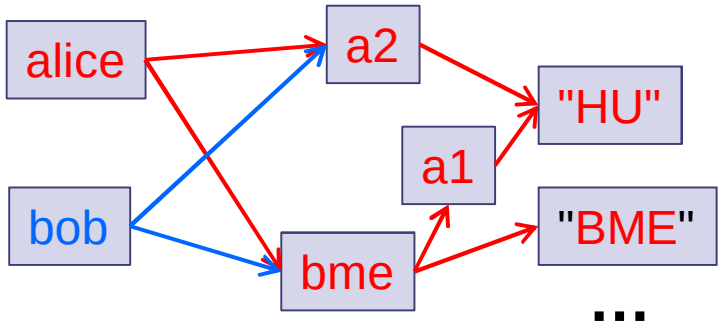
- Let's have the following objects!

```
Address a1 = new Address("HU", "Bp", "Műegyetem rkp 3");
Address a2 = new Address("HU", "Bp", "Alkotás u. 10");

University bme = new University("BME", a1);

Student alice = new Student("Alice", bme,  a2);
Student bob   = new Student("Bob", bme, a2);
```

```
ObjectOutputStream oos = ...;
oos.writeObject(alice);
oos.writeObject(bob);
oos.close();
```



Writing own serialization

- Use

```
private void writeObject(ObjectOutputStream out)
        throws IOException
private void readObject(ObjectInputStream in)
        throws IOException, ClassNotFoundException
```

- Super/subclass data handled automatically
- Default implementation
  - `out.defaultWriteObject(), in.defaultReadObject()`
- *out* and *in* have helper methods
  - reading/writing primitive and serializable types

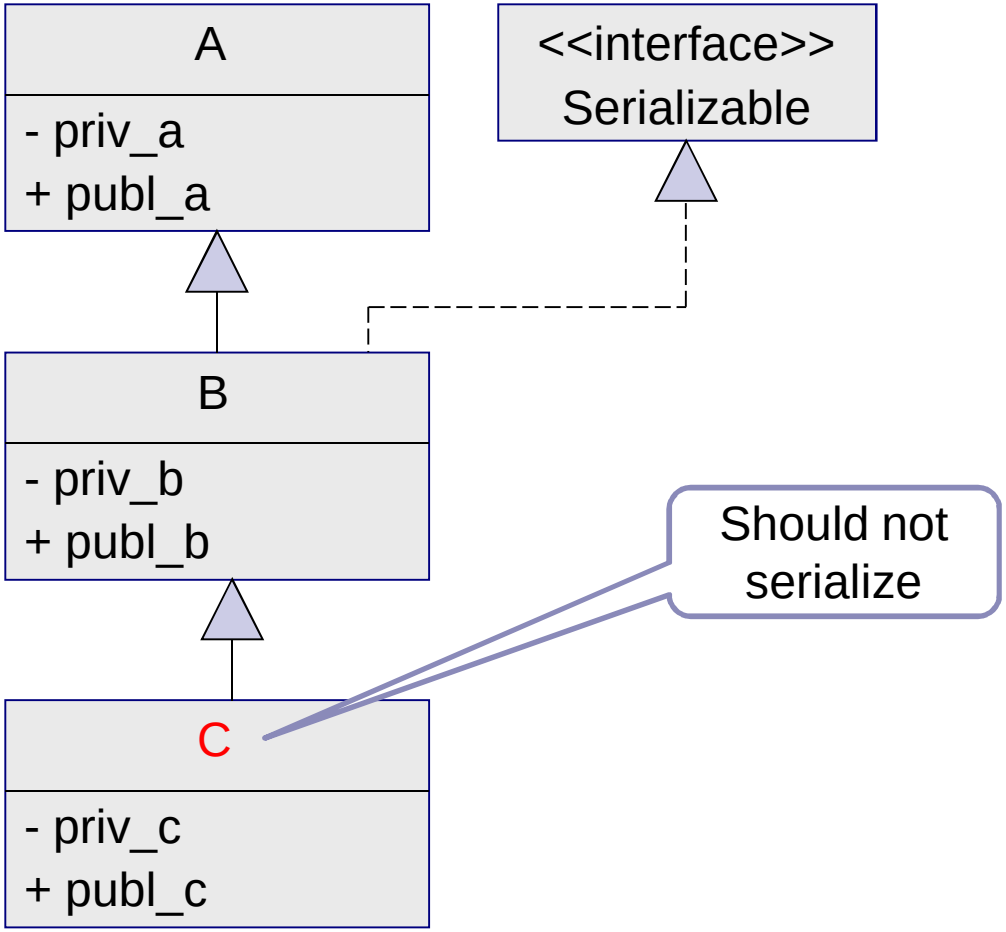
Writing own serialization 2

- For total control implement

```
interface Externalizable extends Serializable {
  public void writeExternal(ObjectOutput out)
    throws IOException;
  public void readExternal(ObjectInput in)
    throws IOException, ClassNotFoundException;
}
```

- Handle super/subclass data explicitly

Stopping serialization



Stopping serialization

```
private void writeObject(ObjectOutputStream out)
throws IOException {

      throw new NotSerializableException("C");
}


private void readObject(ObjectInputStream in)
throws IOException, ClassNotFoundException {

      throw new NotSerializableException("C");
}
```

Versioning

- Each class has a unique version ID
  - showing it: `serialver` *`ClassName`*
- Ensuring compatibility
  - use same version ID

```
static final long serialVersionUID
 = 1027553947283749 5L;
```

- Compatible changes
  - method/field addition or access modification change
  - static/transient → persistent

Other serialization formats

- Serialization is low level
  - handling serialized data is hard
- JSON
  - **J**ava**S**cript **o**bject **n**otation
  - http://www.json.org/
- XML
  - own protocol
  - JAXB
    - *javax.xml.bind* package
- …

JSON

- Lightweight data-interchange format
  - easy for humans to read and write
    - mostly…
  - easy for machines to parse and generate
  - caveat: no metadata, extra testing is needed!
- Two structures
  - *Object*: unordered set of name/value pairs
    - like object, struct, etc.
  - *Array*: ordered list of values
    - like array, vector, list, or sequence.

JSON syntax

- Object
  - begins with **{**
  - ends with **}**
  - each name is followed by **:**
  - name/value pairs are separated by **,**

```
{
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2
}
```

- Array
  - begins with **[**
  - ends with **]**
  - values are separated by **,**

```
[ "mon", "tue", "wed",
  "thu", "fri", "sat",
  "sun"
]
```

JSON syntax 2

- Value
  - *string* in double quotes
  - *number*
  - true or false or null
  - *object*
  - *array*
- Structures can be nested.

```
{
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2,
    "siblings" : [
        "bob",
        "charlie"
    ],
    "degree" : null
}
```

JSON handling in Java

- Libraries
  - org.json (e.g. in Android)
    - types mapped to classes
    - build element by element
  - javax.json (JEE 7)
    - builder-based building of objects
  - Google gson
  - …

javax.json: creating JSON object

```
// sometimes trivial
// nesting is used

{ "student" : {
    "name" : "alice",
    "university" : {
        "name" : "BME"
    },
    "average" : 4.2
  }
}
```

```
JsonObject value =
  Json.createObjectBuilder()
  .add("name", "alice")
  .add("university",
     json.createObjectBuilder()
     .add("name", "BME")
  )
  .add("average", 4.2)
  .build();

JsonObject student =
  Json.createObjectBuilder()
  .add("student", value).build();
```

javax.json: reading and writing

```
// src can be InputStream or Reader
JsonReader reader = Json.createReader(src);
JsonObject obj = reader.readObject();
// JsonArray arr = reader.readArray();
reader.close();
```

```
// target can be OutputStream or Writer
JsonWriter w = Json.createWriter(target);
JsonObject o = ...;
w.writeObject(o);
w.close();
```

javax.json: accessing data

- In *JsonObject* and *JsonArray* getters for
  - boolean, int, String (also with def. value)
  - JsonArray, JsonNumber, JsonObject, JsonString
  - *isNull*
- Indexing
  - JsonObject: *String*
  - JsonArray: *int*

```
JsonObject obj = ...;
String name = obj.getString("name");
String uni_name =
  obj.getJsonObject("university")
  .getString("name");
```

XML intro

- e**X**tensible **M**arkup **L**anguage
- Goal: standard, human readable data format
  - storing
  - transmitting
- backed by the success of HTML
- text-based
  - cf. **`toString()`**
- Today: „Whatever the question, XML is the answer„
  - or JSON…

XML features

- Tree structure: hierarchy
  - tags, attributes and text
- W3C standard
  - syntax
  - parsing rules
  - well-formedness and validity
- Meta-structure, can be extended

XML well-formedness

- Syntax rules
  - optional header

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

  - each tag has a closing counterpart

```
<p>Hello <img src=„kitty.png"/> </p>
```

  - there is a root
  - tags sequentially or embedded

XML syntax

- comment

```
<!– this is a comment -->
```

- tag attribute

```
<img src="cat.png"/>
```

  - always use quotation marks

- special signs

XML	text
&amp;	&
&lt;	<
&gt;	>
&apos;	'
&quot;	"

XML validity

- Semantic rules
- Constraints can be given as…
  - schema
  - DTD (document type definition)
- Specifies the structure of a document
  - allowed elements
  - allowed hierarchy
  - allowed attributes
  - ...

XML semantics specification

- DTD
  - old
  - missing features
  - deprecated
- XML schema (XSD)
  - new
  - structured: elements and connections
  - namespaces
  - in XML itself

XML example

```
<?xml version="1.0" encoding="UTF-8"?>
<people_list>
  <person>
    <name>Neil Armstrong</name>
    <birthdate>1930-08-05</birthdate>
    <gender>Male</gender>
  </person>
  <person>
    <name>Buzz Aldrin</name>
    <birthdate>1930-01-20</birthdate>
    <gender>Male</gender>
    <neptuncode>M00N02</neptuncode>
  </person>
</people_list>
```

DTD description

```
<!ELEMENT people_list (person*)>
<!ELEMENT person (name, birthdate, gender?,
      neptuncode?)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT birthdate (#PCDATA)>
<!ELEMENT gender (#PCDATA)>
<!ELEMENT neptuncode(#PCDATA)>
```

XSD description

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="people_list">
    <xs:complexType><xs:sequence>
        <xs:element name="person" maxOccurs="unbounded">
          <xs:complexType><xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="birthdate" type="xs:string"/>
            <xs:element name="gender" type="xs:string"
                minOccurs="0"/>
            <xs:element name=„neptuncode" type="xs:string"
                minOccurs="0"/>
          </xs:sequence></xs:complexType>
        </xs:element>
      </xs:sequence></xs:complexType>
  </xs:element>
</xs:schema>
```

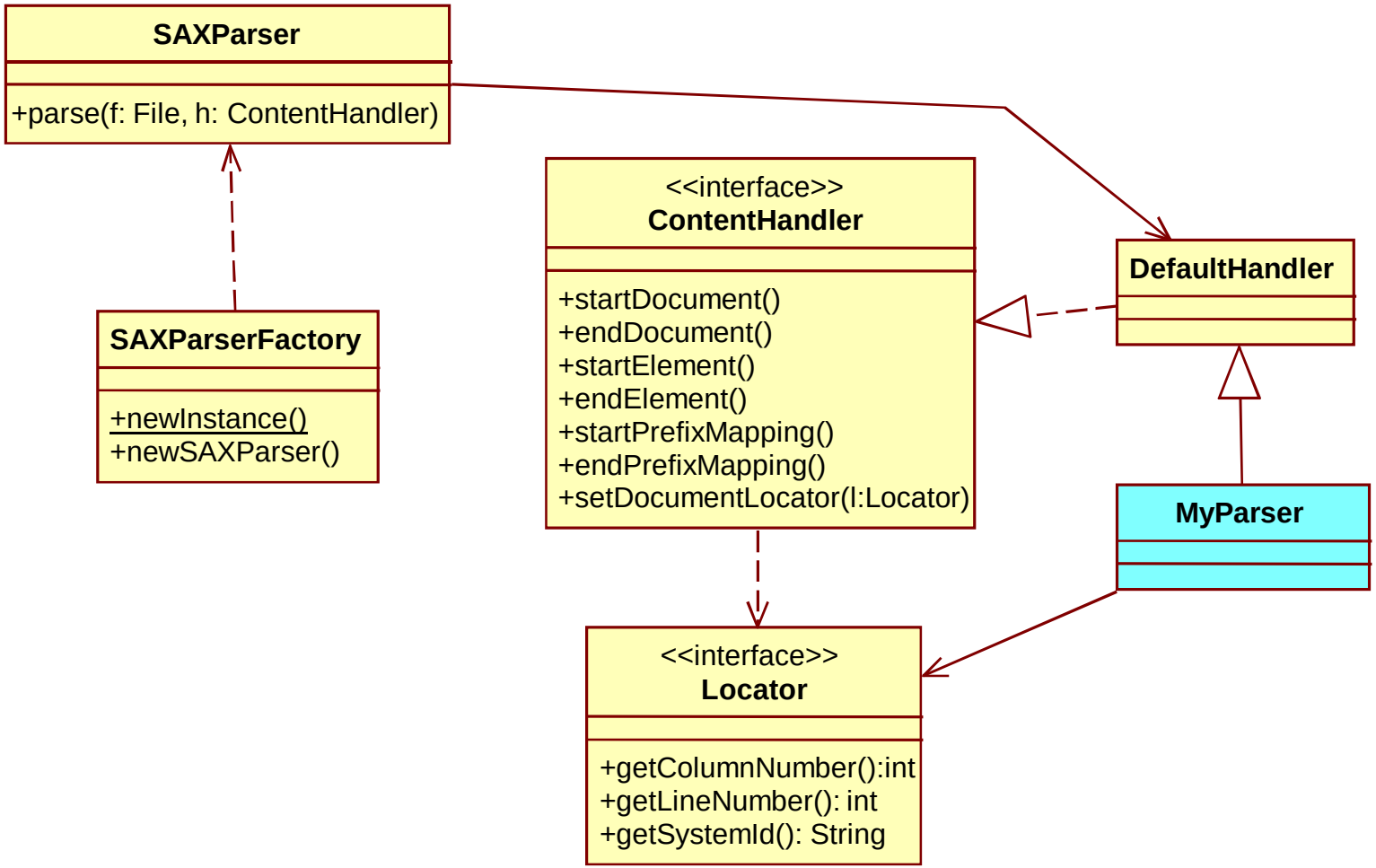
XML handling

- manual
  - simply don’t
- SAX *(simple API for XML)*
  - event based
  - sequential
- DOM *(Document Object Model)*
  - dumb object structure
- JDOM *(Java DOM)*
  - smart object structure

SAX parser usage

- Event handling
  - start/end of document
  - start/end of tag
  - start/end of prefixMapping
  - characters (text)
  - whitespace
  - skipped entities
  - processing instructions

SAX parser classes



ContentHandler interface

- **`org.xml.sax.ContentHandler`**
- This is to be implemented
- Callback-based event handling
  - for each event a method is provided
- Empty implementation
  - **`org.xml.sax.helpers.DefaultHandler`**
  - all methods are empty

ContentHandler

- `void startDocument()`
- `void endDocument()`

- `void startElement(String uri, String localName, String qName, Attributes atts)`
- `void endElement(String uri, String localName, String qName)`

- `void startPrefixMapping(String prefix, String uri)`
- `void endPrefixMapping(String prefix)`

ContentHandler

- `void characters(char[] ch, int start, int length)`
- `void ignorableWhitespace(char[] ch, int start, int length)`
- `void processingInstruction(String target, String data)`
- `void skippedEntity(String name)`
- `void setDocumentLocator(Locator locator)`
  - locator can get access to further processing data

ContentHandler example

- Problem:
  - Let’s create a simple Java application
  - that prints out the XML tree
    - attributes are omitted
    - texts are omitted

ContentHandler example

- Solution:
  - implement interface *ContentHandler*
    - using class *DefaultHandler*
  - register handler
  - parse the XML file

ContentHandler example

```
public class MyParser extends DefaultHandler {

    public static void main(String[] args) {
        DefaultHandler h = new MyParser();
        SAXParserFactory factory =
            SAXParserFactory.newInstance();
        try {
            SAXParser p = factory.newSAXParser();
            p.parse(new java.io.File(args[0]), h);
        } catch (Exception e) {e.printStackTrace();}
    }

    ...
```

ContentHandler example

```
...
int tab=0;
public void println(String s) {
    for (int i = 0; i < tab; i++) {
        System.out.print("  ");
    }
    System.out.println(s);
}
public void startDocument() throws SAXException {
    println("Start document");
}
public void endDocument() throws SAXException {
    println("End document");
}
...
```

ContentHandler example

```
    ...
    public void startElement(String namespaceURI,
       String sName, String qName, Attributes attrs)
       throws SAXException {
        tab++;
        println("start element: "+qName);
    }

    public void endElement(String namespaceURI,
       String sName, String qName)
       throws SAXException {
        println("end element: "+qName);
        tab--;
    }
}
```

ContentHandler example input

```
<!-- test.xml -->
<level1>
        <level2>
                <level3 attr1="test1">
                </level3>
                <level3 attr1="test2" attr2="second">
                </level3>
                <level3 attr1="test3">
                </level3>
        </level2>
</level1>
```

ContentHandler example output

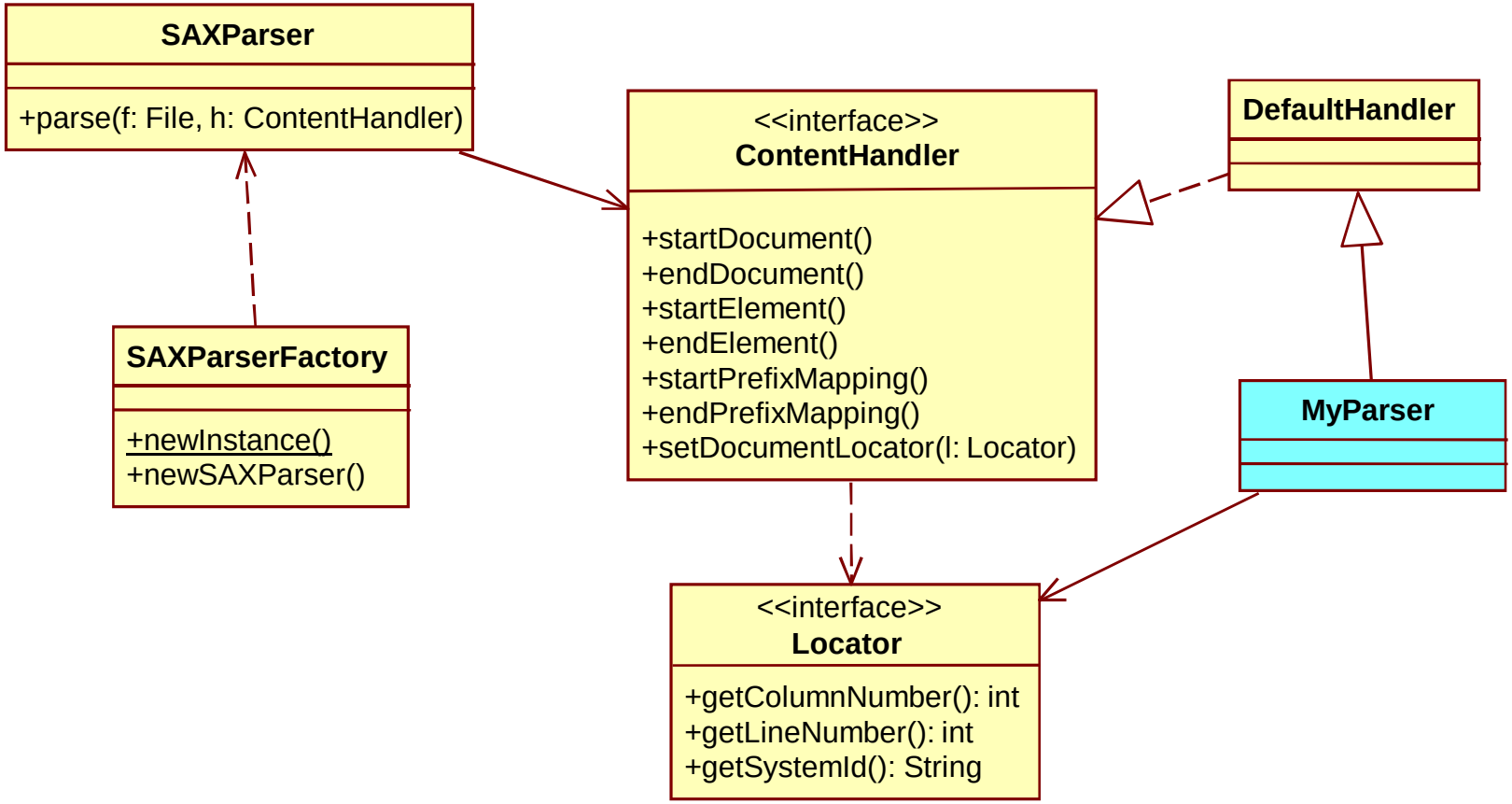
```
$ java MyParser test.xml
Start document
  start element: level1
    start element: level2
      start element: level3
      end element: level3
      start element: level3
      end element: level3
      start element: level3
      end element: level3
    end element: level2
  end element: level1
End document
```

Locator

- Provides information about processed file

- **`void setDocumentLocator(Locator l)`**
  - **`int getColumnNumber()`**
    - index of character in current line the handler is processing
  - **`int getLineNumber()`**
    - index of line being processed
  - **`String getSystemId()`**
    - name of document (e.g. filename) as a URL

Locator classes



Locator example

```
Locator loc = null;
public void setDocumentLocator(Locator l) {
    println("LOCATOR");
    loc = l;
}
```

```
public void startElement(String namespaceURI,
   String sName, String qName, Attributes attrs)
   throws SAXException {
     tab++;
     println("start element: "+qName);
     println("  Locator: ("+loc.getLineNumber()
       +","+loc.getColumnNumber()+") "
       +loc.getPublicId()+", "+loc.getSystemId());
     ...
}
```

Locator example input

```
<!-- test.xml -->
<level1>
        <level2>
                <level3 attr1="test1">
                </level3>
                <level3 attr1="test2" attr2="second">
                </level3>
                <level3 attr1="test3">
                </level3>
        </level2>
</level1>
```

Locator example output

```
$ java MyParser test.xml
LOCATOR
Start document
  start element: level1
    Locator: (1,9) null,
file:/home/balage/sax-example/example4/test.xml
    start element: level2
      Locator: (3,10) null,
file:/home/balage/sax-example/example4/test.xml
      start element: level3
        Locator: (4,25) null,
file:/home/balage/sax-example/example4/test.xml
        attr0: attr1=test1
      end element: level3
...
```

Document validity

- Let’s add validation

```
SAXParserFactory factory = SAXParserFactory.newInstance();
factory.setValidating(true);
factory.setNamespaceAware(true);
```

```
SAXParser p = factory.newSAXParser();
String JAXP_SCHEMA_LANGUAGE =
 "http://java.sun.com/xml/jaxp/properties/schemaLanguage";
String W3C_XML_SCHEMA =
 "http://www.w3.org/2001/XMLSchema";
p.setProperty(JAXP_SCHEMA_LANGUAGE, W3C_XML_SCHEMA);
```

XML error handling

- Error types
  - fatal error
    - document is not well formed
  - error
    - document is not valid
  - warning
    - small error, e.g. same type declared twice

Document Object Model

DOM

- *Document Object Model*
  - builds an object model based on the XML content
- Object model can be modified
  - and printed out as XML
- Validation included

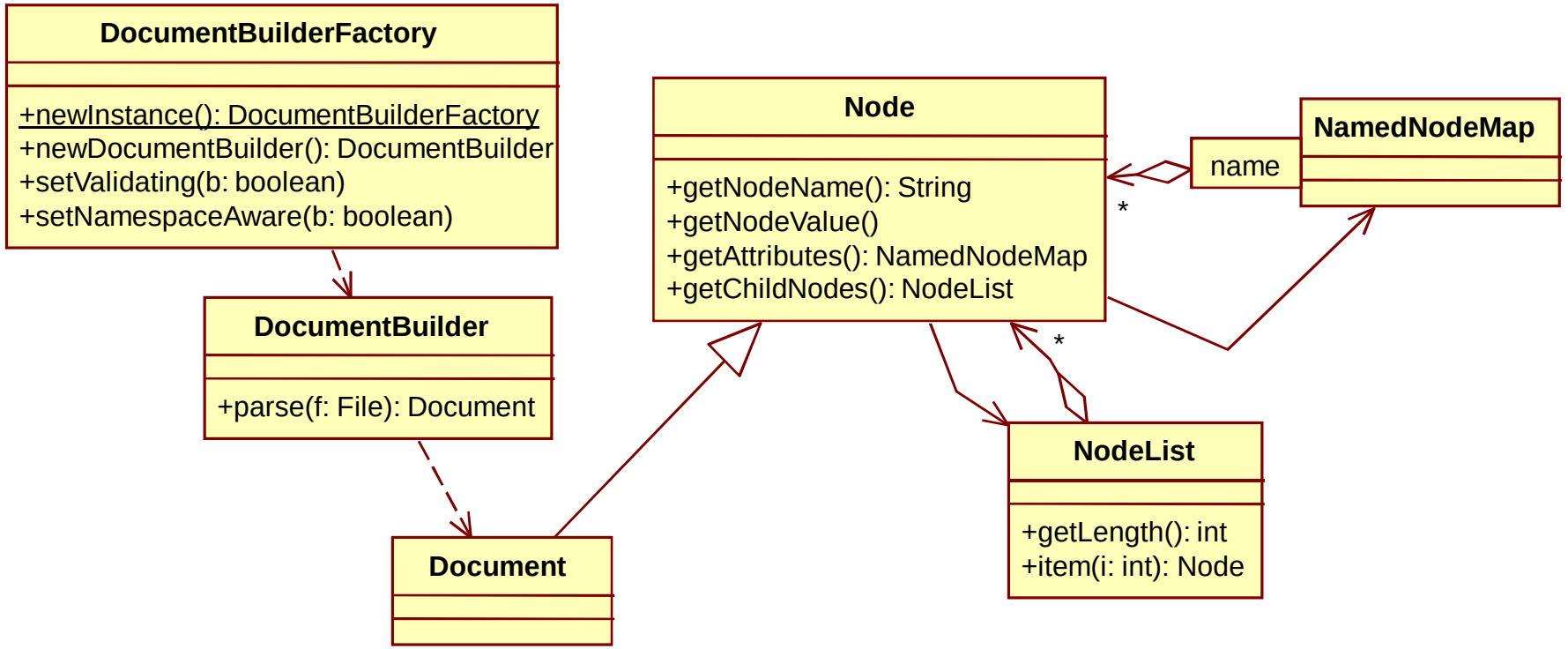
DOM introductory example

```
try {
  DocumentBuilderFactory factory =
    DocumentBuilderFactory.newInstance();
  factory.setValidating(true);
  factory.setNamespaceAware(true);

  DocumentBuilder builder =
    factory.newDocumentBuilder();

  Document document =
    builder.parse(new java.io.File(args[0]));
  ...
} catch (Exception e) {
  e.printStackTrace();
}
```

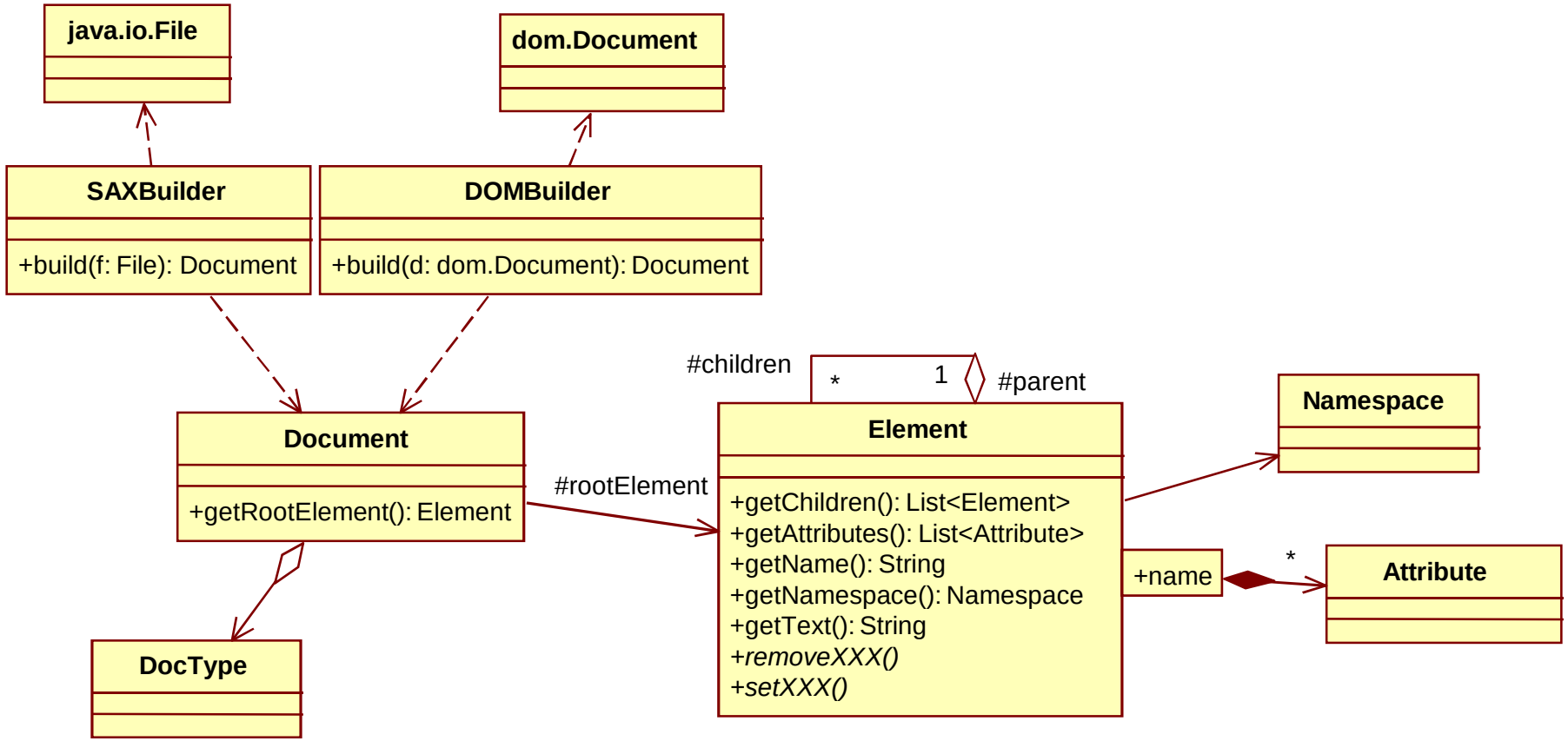
DOM classes


DocumentBuilderFactory
+newInstance(): DocumentBuilderFactory
+newDocumentBuilder(): DocumentBuilder
+setValidating(b: boolean)
+setNamespaceAware(b: boolean)
DocumentBuilder
+parse(f: File): Document
Document
Node
+getNodeName(): String
+getNodeValue()
+getAttributes(): NamedNodeMap
+getChildNodes(): NodeList
name
NamedNodeMap
NodeList
+getLength(): int
+item(i: int): Node
*
*

JDOM

- Object model closer to XML
  - **`Attribute, CDATA, Comment, Content, DefaultJDOMFactory, DocType, Document, Element, EntityRef, Namespace, ProcessingInstruction, Text`**
- Attributes, children are easier to access, modify
  - no **`NamedNodeMap`**, **`NodeList`**
  - uses **`java.util.List`**

JDOM classes (partial)



JDOM features

- Parsing is out-sourced
  - **`DOMBuilder`**
  - **`SAXBuilder`**
- Document can be saved
  - as XML document
  - as a DOM model
  - using SAX event-generator

JDOM features

- Filters can be specified for searching
  - **`org.jdom.filter.Filter`**
    - **`boolean matches(java.lang.Object obj)`**
- Supports XSL transformations
  - **`org.jdom.transform.XSLTranformer`**
- Supports XPATH searches
  - **`org.jdom.xpath.XPath`**

JDOM Example: simple printout

```
public class JDOMParse {
  static void print(Element n, String tab) ...
  public static void main(String[] args) {
      SAXBuilder b = new SAXBuilder();
      File f = new File("test.xml");
          try {
              Document doc = (Document)b.build(f);
              Element r = doc.getRootElement();
              print(r, "");
          } catch (IOException io) {
               System.out.println(io.getMessage());
          } catch (JDOMException je) {
               System.out.println(je.getMessage());
          }
    }
}
```

JDOM Example: simple printout

```
static void print(Element n, String tab) {
    System.out.println(tab+"("+n.getName()
       +") \""+n.getValue()+"\"");
    if (n.hasAttributes()) {
        List<Attribute> list = n.getAttributes();
        for (Attribute a : list) {
            System.out.println(tab+"attr: "+a);
        }
    }
    List<Element> nl = n.getChildren();
    for (Element e : nl) {
        print(e, tab+"  ");
  }
}
```

Recursion