Markup

- A method of distinguishing text from instructions in typesetting systems.
- Markup = instructions to computerized typesetting systems.
- Special characters used to show when a markup instruction starts and stops.

- Example:

  <centre on> This is a <italics on> very serious <italics off> matter.<centre off>

  This is a *very serious* matter.

- "Tags" are easily understandable by both the human reader and the machine

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

XML Structure Definition

- XML stands for eXtensible Markup Language
- XML is a meta-language allowing the definition of mark-up language syntaxes
- XML files format is text, i.e. an XML file is defined as a string of chars
- Almost every legal UNICODE character may appear in an XML document
- XML documents are "human readable", i.e., they do not need any specific software to be understood
- XML is independent from any hardware platform or software
- XML allows to represent any type of document along with its structure independently from its use in applications
- XML is mainly used as a data exchange format

Basic Structure of XML

- XML consists of **tags** and **text**
- Tags come in **pairs** (start and end tags)
- <name>........</name>
- The characters between the start and end tags, if any, are the element's content
- XML as only one basic type: **text**
- <name>Nicola</name>, <amount>23.45</amount>
- A pair of tags with no text can be represented by the empty element tag
- <linebreak></linebreak> = <linebreak/>

XML Nesting

- Tags can be (properly) **nested**
- CORRECT: <lastname>...<name>...</name>...</lastname>
- WRONG: <name>...<lastname>...</name>...</lastname>
- Nested tags can be used to express tree-like structure

```
<person>
   <lastname>Tonellotto</lastname>
   <name>Nicola</name>
   <email>nicola.tonellotto@pippo.com</email>
</person>
```

- Element contents may contain other elements, which are called child elements

XML Lists

- A list/array can be represented using the same tag repeatedly:

```
<addressbook>
  <person>...</person>
  <person>...</person>
  <person>...</person>
</addressbook>
```

- Nested tags can be part of a list:

```
<addressbook>
  <person>
    <lastname>Tonellotto</lastname>
    <name>Nicola</name>
  </person>
  <person>
    <lastname>Sbazzeguti</lastname>
    <name>Pepito</name>
  </person>
</addressbook>
```

XML Terminology

Attribute

(child) element

```
<person type=”teacher”>
  <lastname>Tonellotto</lastname>
  <name>Nicola</name>
  <email>nicola.tonellotto@pippo.com</email>
</person>
```

(root) element

Text

XML Basic Example

```
<?XMLversion ="1.0" encoding="utf-8">
<!-- comment -->

<person type=”teacher”>
  <lastname>Tonellotto</lastname>
  <name>Nicola</name>
  <email>nicola.tonellotto@pippo.com</email>
</person>
```

XML Namespaces

- Namespaces in XML provide a facility for associating the elements and/or attributes in all or part of a document with a particular schema & avoiding name clashes.
- Namespace declarations have a scope.
- A namespace declaration is in scope for the element on which it is declared and of that element's children.
- The namespace name and the local name of the element together form a globally unique name known as a qualified name.

```
<?XMLversion ="1.0" encoding="utf-8">
<!-- comment -->

<person type=”teacher” xmlns="http://www.test.it/PersonInfo">
    <lastname>Tonellotto</lastname>
    <name>Nicola</name>
    <email>nicola.tonellotto@pippo.com</email>
</person>
```

Uniform Resource Identifier (URI)

Well-formed XML

- An XML document is well-formed if it satisfies a list of syntax rules provided in the official W3C specification
- Some key points in the fairly lengthy list include:
  - The document contains only properly encoded legal Unicode characters
  - None of the special syntax characters such as < and & appear except when performing their markup-delineation roles
  - The begin, end, and empty-element tags that delimit the elements are correctly nested, with none missing and none overlapping
  - The element tags are case-sensitive; the beginning and end tags must match exactly. Tag names cannot contain any "special" character, nor a space character, and cannot start with -, ., or a numeric digit.
  - A single "root" element contains all the other elements

Valid XML

- The formal syntax of an XML language may be explicitly expressed
- Document Type Definition (DTD): list of markup declarations describing the structure of an XML document
  - Definition of element/attribute names
  - Definition of element/attribute contents
  - Data type of attributes
- XML Schema (XSD): successor of DTD, it uses a rich datatyping system and allow for more detailed constraints on an XML document's logical structure
- An XML document is valid if
  - it explicitly refers a DTD or a XSD
  - it satisfies its syntax constraints

XML Schema Example

File "Person.xml"

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

<Person
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:noNamespaceSchemaLocation="Person.xsd">

        <First>Sophie</First>
        <Last>Jones</Last>
        <Age>34</Age>

</Person>
```

File "Person.xsd"

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema
  xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="Person">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="First"
              type="xs:string"/>
        <xs:element name="Middle"
              type="xs:string"
              minOccurs="0"/>
        <xs:element name="Last"
              type="xs:string"/>
        <xs:element name="Age"
              type="xs:integer"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

XML usage

- Softwares deal with XML in three ways:
  1. **Parsing**: given an XML document as input, extract the data and process it programmatically
  2. **Generation**: give some structured data, generate their XML representation
  3. **Transformation**: given an XML document as input, extract the data, process is programmatically and generate their new XML representation
- Java libraries exist to read, parse, validate, generate and write XML documents (as well as in any other programming language)
- JAXP - The Java API for XML Processing (default implementation included since Java 5 SDK)
- Other Java APIs for XML exist: Apache Xerces (implements JAXP interfaces), DOM4J, JDOM, Woodstox (high-performance XML processor)

XML Parsing in Java

- There are two basic approaches to XML parsing in Java:
  - **SAX - Simple API for XML**
    - serial, sequential access to XML tags and content
    - event-oriented callback model,
    - fast and low overhead
    - difficult to use for transforms
  - **DOM - Document Object Model for XML**
    - Tree-based access to entire XML document
    - data traversal model
    - keeps entire document in memory
    - easy to use for transforms

SAX

- A SAX parser traverses an XML document and generates a sequence of events
- An event is the encountering of an element in the XML document
- The SAX parser delivers events (and errors, if any) to your program
- Each event must be handled by an handler object
- Handler objects must implement particular interfaces, for example:
  - org.xm.sax.ContentHandler: for receiving events about element contents
  - org.xml.sax.ErrorHandler: for receiving errors occurred during parsing
- SAX parsers can read XML documents from any java.io.Reader or directly from an existing URL
- SAX parsers can be instructed to validate the parsed XML document.

SAX Example - XML Document

```
<?xml version="1.0" encoding="utf-8"?>
<email>
    <message>
        <from>nicola.tonellotto@isti.cnr.it</from>
    </message>
</email>
```

SAX Example - Handlers (I)

```
import org.xml.sax.*;

public class SAXHandler implements ContentHandler, ErrorHandler
{
     private Locator loc = null;

     public void setDocumentLocator(Locator l) {
         loc = l;
     }

     // called when a set of characters is encountered
     public void characters(char ch[], int st, int len) {
         String s = new String(ch, st, len);
         System.out.println("Got content string '" + s + "'");
     }

     // called when a start tag is encountered
     public void startElement(String uri, String lname, String qname, Attributes attrs) {
         System.out.print(lname + " tag with ");
         System.out.print(attrs.getLength() + " attrs starts");
         System.out.println(" at line " + loc.getLineNumber());
     }

     // called when an end tag is encountered
     public void endElement(String uri, String lname, String qname) {
         System.out.print(lname + " tag ends ");
         System.out.println("at line " + loc.getLineNumber());
     }
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

SAX Example - Handlers (II)

```
    // called when the start of the document is encountered
    public void startDocument() {}

    // called when the end of the document is encountered
    public void endDocument() {}

    // other handlers
    public void processingInstruction(String t, String d) { }
    public void skippedEntity(String name) { }
    public void ignorableWhitespace(char[] ch, int st, int len) { }
    public void startPrefixMapping(String p, String uri) { }
    public void endPrefixMapping(String p) { }

    // error handlers
    public void warning(SAXParseException e) {
        System.err.print("SAX Warning: " + e);
        System.err.println(" at line " + loc.getLineNumber());
    }

    public void error(SAXParseException e) {
        System.err.print("SAX Error: " + e);
        System.err.println(" at line " + loc.getLineNumber());
    }

    public void fatalError(SAXParseException e) {
        System.err.print("SAX Fatal Error: " + e);
        System.err.println(" at line " + loc.getLineNumber());
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

SAX Example - Program

```
import java.io.*;
import org.xml.sax.*;
import org.xml.sax.helpers.*;
import javax.xml.parsers.*;

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

            System.out.println("Creating and setting up the SAX parser.");
            SAXParserFactory sp_factory = SAXParserFactory.newInstance();
            XMLReader theReader = sp_factory.newSAXParser().getXMLReader();
            SAXHandler theHandler = new SAXHandler();
            theReader.setContentHandler(theHandler);
            theReader.setErrorHandler(theHandler);
            theReader.setFeature("http://xml.org/sax/features/validation", false);

            System.out.println("Making InputSource for " + args[0]);
            FileReader file_in = new FileReader(args[0]);

            System.out.println("About to parse... ");
            theReader.parse(new InputSource(file_in));
            System.out.println("...parsing done.");

        } catch (Exception e) { System.err.println("Error: " + e);  e.printStackTrace(); }
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

DOM

- A DOM builder traverses an XML document and builds an in-memory tree representation of the whole XML document
- A reference to this tree is returned to your program
- A DOM tree is composed by objects from the org.w3c.dom packages, most of them implementing the following interfaces:
  - Node — super-interface of all DOM tree nodes
  - Document — interface representing the entire XML document
  - Element — interface representing a tag in the XML document
  - Attr — interface representing an attribute of a tag
  - Text — content of an elements or an attribute
- A DOM tree can be manipulated as a normal tree: traverse, modify, print, transform, etc...
- DOM builders can be instructed to validate the parsed XML document.

DOM Example - Program

```
import org.w3c.dom.*;
import javax.xml.parsers.*;
import org.xml.sax.*;
import java.io.*;

public class SimpleXMLBuilder
{
     public static void main(String [] args)
     {
          String uri = args[0];
          try {
            System.out.println("Creating Document builder.");
            DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
            dbf.setValidating(false);
            DocumentBuilder db = dbf.newDocumentBuilder();

            System.out.println("Ready to parse!");
            FileReader file_in = new FileReader(uri);
            Document doc = db.parse(new InputSource(file_in));
            System.out.println("Parsed document, ready to process.");

            DOMTreeProcessor proc = new DOMTreeProcessor();
            proc.process(doc, System.out);
          }
          catch (Exception e) {
             System.err.println("XML Exception thrown: " + e);
             e.printStackTrace();
          }
     }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

DOM Example - Tree Processor (I)

```
import org.w3c.dom.*;

public class DOMTreeProcessor
{
  public void process(Document doc, java.io.PrintStream os) {
    printTree(doc, os, 0);
  }

  void printTree(Node n, java.io.PrintStream os, int indent) {
    switch (n.getNodeType()) {
      case Node.ATTRIBUTE_NODE:
        for (int i = 0; i < indent; i++) os.print(" ");
        os.println("Attr " + n.getNodeName() + "=" + n.getNodeValue());
        break;

      case Node.TEXT_NODE:
      case Node.CDATA_SECTION_NODE:
        for (int i = 0; i < indent; i++) os.print(" ");
      os.println("Text '" + n.getNodeValue() + "'.");
      break;

      case Node.DOCUMENT_NODE: {
        os.println("DOCUMENT:");
        NodeList kids = n.getChildNodes();
        for (int i = 0; i < kids.getLength(); i++)
        printTree(kids.item(i), os, indent + 2);
        break;
      }
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

DOM Example - Tree Processor (II)

```
        case Node.DOCUMENT_TYPE_NODE:
          for (int i = 0; i < indent; i++) os.print(" ");
          os.println("Document type is " + n.getNodeName());
          break;

        case Node.ELEMENT_NODE: {
          for (int i = 0; i < indent; i++) os.print(" ");
          os.println("Element " + n.getNodeName());
          NamedNodeMap attrs = n.getAttributes();
          for (int i = 0; i < attrs.getLength(); i++)
            printTree(attrs.item(i), os, indent + 5);
          NodeList kids = n.getChildNodes();
          for (int i = 0; i < kids.getLength(); i++)
            printTree(kids.item(i), os, indent + 2);
          break;
        }

        default:
          os.println("Unexpected kind of node - ignored.");
          break;
      }
    }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

XML generation in Java

- XML generation is not as common as XML parsing.
- Main approaches:

1. **java.io.StringBuffer**: XML is text, so for very short XML files. However, it blurs the division between Java text and XML, does not check well-formed-ness and validity, and requires additional work for character escaping.
2. **SAX**: it is designed from the ground up to read XML, rather than write it, but it includes methods that can be used to generate XML. It is very possible to create invalid XML using this method.
3. **DOM**: its in-memory tree model is perfectly suitable for writing out XML files. if creating a large XML document, memory usage becomes a concern.

DOM Example - Generation

```
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.transform.TransformerFactory;
import javax.xml.transform.dom.DOMSource;
import javax.xml.transform.stream.StreamResult;

import org.w3c.dom.*;

public class SimpleDOMWriter
{
     public static void main(String[] args) throws Exception {
         // Create a new document.
         Document doc = DocumentBuilderFactory.newInstance().newDocumentBuilder().newDocument();

         // Create and add a root element and an attribute.
         Element root = doc.createElement("email");
         doc.appendChild(root);
         Element child1 = doc.createElement("message");
         root.appendChild(child1);
         Element child2 = doc.createElement("from");
         Text txt = doc.createTextNode("nicola.tonellotto@isti.cnr.it");
         child2.appendChild(txt);
         child1.appendChild(child2);

         // Output the document.
         TransformerFactory.newInstance().newTransformer().transform(new DOMSource(doc), new StreamResult(System.out));
     }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<email>
  <message>
    <from>nicola.tonellotto@isti.cnr.it</from>
  </message>
</email>
```

Java Annotations

- A **Java annotation** is a form of **syntactic metadata** that can be added to Java source code.
- Classes, methods, variables, parameters and packages may be annotated.
- When Java source code is compiled, annotations can be processed by compiler plug-ins called **annotation processors**.
- Processors can produce **informational messages** or **create additional Java source files** or resources, which in turn may be compiled and processed, and also **modify the annotated code itself**.
- Java code built-in annotations:
  - **@Override** Checks that the method is an override. Causes a compile error if the method is not found in one of the parent classes or implemented interfaces.
  - **@Deprecated** Marks the method as obsolete. Causes a compile warning if the method is used.
  - **@SuppressWarnings** Instructs the compiler to suppress the compile time warnings specified in the annotation parameters

Annotation Examples

```
public class Animal {

    public void speak() {
    }

    public String getType() {
        return "Generic animal";
    }
}

public class Cat extends Animal {

    @Override
    public void speak() { // This is a good override.
        System.out.println("Meow.");
    }

    @Override
    public String gettype() { // throws compile warning due to mistyped name.
        return "Cat";
    }
}
```

XML Data Models

- Given a valid XML document, its DTD/XSD defines a **unique data model**
- **DOM** is used to manipulate this data model
  - PROs: simple and general (DTD/XSD not required)
  - CONs: no typing, no compile-time checking:

```
BAD: root.getChild("email").getChild("message").getChild("from").getText(); // return a string

GOOD: root.getEmail().getMessage().getFrom().getAddress(); // return an Address object instance
```

- **JAXB** - The Java Architecture for XML Binding (https://jaxb.java.net/) - provides a **binding compiler** generating **Java interfaces** from DTDs and XSDs
  - PROs: preserve types, compile-time checking
  - CONs: complex, specific to a given data model
- The binding compiler generates **unmarshallers**, **validators** and **marshallers**.

JAXB Example - XML Schema

```
<xs:element name="Person" type="PersonType"/>
<xs:complexType name="PersonType">
  <xs:sequence>
    <xs:element name=“Name" type="xs:string"/>
    <xs:element name="Address" type="AddressType"
       minOccurs="1" maxOccurs="unbounded"/>
  </xs:sequence>
</xs:complexType>
<xs:complexType name="AddressType">
  <xs:sequence>
    <xs:element name="Number" type="xs:unsignedInt"/>
    <xs:element name="Street" type="xs:string"/>
  </xs:sequence>
</xs:complexType>
```

JAXB Example - XML File

```
<Person
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="/home/nic/demo.xsd">
   <Name>Sharon Krisher</Name>
   <Address>
      <Street>Iben Gevirol</Street>
      <Number>57</Number>
   </Address>
   <Address>
      <Street>Moshe Sharet</Street>
      <Number>89</Number>
   </Address>
</Person>
```

ISTITUTO DI SCIENZA E TECNOLOGIE DELL'INFORMAZIONE "A. FAEDO"

JAXB Example - Generated Interfaces

```
public interface AddressType
{
    long getNumber();
    void setNumber(long value);

    String getStreet();
    void setStreet(String value);
}

public interface PersonType
{
    String getName();
    void setName(String value);

    java.util.List<AddressType> getAddress();
}
```

JSON

- Lightweight Data Exchange Format
- Easy for humans/machines to read and write
- Minimal. Compact, textual, and subset of JavaScript
- Example:

```
{“name” : “John”, “age”:20, “phone”:[“2761234”, “1234567”]}
```

- Used heavily in RESTful web services, configuration, databases, browser <-> server communication
- Used by popular websites in their RESTful Web Services
  - Facebook, Twitter, Amazon
  - Twitter Streaming API discontinued XML

JSON Syntax

- Six data types: string, number, object, array, true, false and null.
- Only two data structures: objects and arrays
- An object is a set of name-value pairs
  - Objects are enclosed in curly braces ({})
  - Their name-value pairs are separated by a comma (,)
  - Name and value in a pair are separated by a colon (:).
  - Names in an object are strings,
  - Values may be of any of the six data types, including another object or an array
- An array is a list of values
  - Arrays are enclosed in square brackets ([])
  - Their values are separated by a comma (,)
  - Each value in an array may be of a different type, including another array or an object.
- When objects and arrays contain other objects or arrays, the data has a tree-like structure
- Unlike XML, JSON does not have a widely accepted schema for defining and validating the structure of JSON data.

JSON Example

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 18,
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
   "phoneNumbers": [
      { "Mobile": "111-111-1111" },
      { "Home": "222-222-2222" }
   ]
}
```

JSON Processing

- Two programming models:
  - The object model
    - creates a tree that represents the JSON data in memory.
    - the tree can then be navigated, analyzed, or modified.
  - The streaming model
    - uses an event-based parser that reads JSON data one element at a time
    - the parser generates events and stops for processing when an object or an array begins or ends, when it finds a key, or when it finds a value
    - Each element can be processed or discarded by the application code, then the parser proceeds to the next event.
- JSR 353 - Java API for JSON Processing (https://jsonp.java.net/), included in Java 7 EE.
  - provides an API to parse, transform, and query JSON data using the object model or the streaming model

JSON Object Model Parsing Example

```
{
   "firstName": "Duke",
   "lastName": "Java",
   "age": 18,
   "streetAddress": "100 Internet Dr",
   "city": "JavaTown",
   "state": "JA",
   "postalCode": "12345",
   "phoneNumbers": [
      { "Mobile": "111-111-1111" },
      { "Home": "222-222-2222" }
   ]
}
```

```
import java.io.FileReader;
import javax.json.*

public class JsonObjectModelParser
{
  public static void main(String args[]) throws Expcetion
  {
    JsonReader reader = Json.createReader(new FileReader(arg[0]));
    JsonStructure jsonst = reader.read();
    navigateTree(jsonst, null);
  }
```

JSON Object Model Navigation Example

```
public static void navigateTree(JsonValue tree, String key) {
   if (key != null)
      System.out.print("Key " + key + ": ");
   switch(tree.getValueType()) {
      case OBJECT:
         System.out.println("OBJECT");
         JsonObject object = (JsonObject) tree;
         for (String name : object.keySet()) navigateTree(object.get(name), name);
         break;
      case ARRAY:
         System.out.println("ARRAY");
         JsonArray array = (JsonArray) tree;
         for (JsonValue val : array) navigateTree(val, null);
         break;
      case STRING:
         JsonString st = (JsonString) tree;
         System.out.println("STRING " + st.getString());
         break;
      case NUMBER:
         JsonNumber num = (JsonNumber) tree;
         System.out.println("NUMBER " + num.toString());
         break;
      case TRUE:
      case FALSE:
      case NULL:
         System.out.println(tree.getValueType().toString());
         break;
   }
}
}
```

JSON Object Model Writing Example

```
import java.io.StringWriter;
import javax.json.JsonWriter;
...
StringWriter stWriter = new StringWriter();
JsonWriter jsonWriter = Json.createWriter(stWriter);
jsonWriter.writeObject(model);
jsonWriter.close();

String jsonData = stWriter.toString();
System.out.println(jsonData);
```

JSON Streaming Model Parsing Example

```
import javax.json.Json;
import javax.json.stream.JsonParser;
...
JsonParser parser = Json.createParser(new StringReader(jsonData));
while (parser.hasNext()) {
   JsonParser.Event event = parser.next();
   switch(event) {
      case START_ARRAY:
      case END_ARRAY:
      case START_OBJECT:
      case END_OBJECT:
      case VALUE_FALSE:
      case VALUE_NULL:
      case VALUE_TRUE:
         System.out.println(event.toString());
         break;
      case KEY_NAME:
         System.out.print(event.toString() + " " + parser.getString() + " - ");
         break;
      case VALUE_STRING:
      case VALUE_NUMBER:
         System.out.println(event.toString() + " " + parser.getString());
         break;
   }
}
```

JSON Streaming Model Writing Example

```
FileWriter writer = new FileWriter("test.txt");
JsonGenerator gen = Json.createGenerator(writer);
gen.writeStartObject()
   .write("firstName", "Duke")
   .write("lastName", "Java")
   .write("age", 18)
   .write("streetAddress", "100 Internet Dr")
   .write("city", "JavaTown")
   .write("state", "JA")
   .write("postalCode", "12345")
   .writeStartArray("phoneNumbers")
      .writeStartObject()
         .write("type", "mobile")
         .write("number", "111-111-1111")
      .writeEnd()
      .writeStartObject()
         .write("type", "home")
         .write("number", "222-222-2222")
      .writeEnd()
   .writeEnd()
.writeEnd();
gen.close();
```

ISTITUTO DI SCIENZA E TECNOLOGIE
DELL'INFORMAZIONE "A. FAEDO"

ISTITUTO DI SCIENZA E TECNOLOGIE
DELL'INFORMAZIONE "A. FAEDO"