Solutions:
Exercise - Data Formats

Web Data Integration IE683
University of Mannheim, Germany

October 2020

In the following three exercise sections you will be asked to perform basic parsing and extraction tasks with Java. Each subsection is dedicated to one of the three formats: XML, JSON and RDF. The tasks are rather basic and the goal is to refresh your knowledge in Java in general and in particular in parsing those formats. Besides the solution and code given below the exercises you can find the complete solution as class (per Exercise) in a Java project located within ILIAS.

1 How to start with Java?

Download and unzip the DataParser project which can be found in ILIAS. Import the project as a Maven project in Eclipse. Now inspect the structure of the project. Inside the src.main.java folder, you can find three packages. Each package is dedicated to one data format: JSON, RDF and XML. For each format, we have prepared some Java classes which will help you solve the tasks of this exercise.

2 XML

This subsection is dedicated to the XML format. In particular you are asked to perform XPath queries on the *Mondial* dataset[1]. This dataset includes world geographic information integrated from the *CIA World Factbook*, the *International Atlas* and the *TERRA database*, to name just the pre-dominant sources. Please inspect the document manually (using a text editor) in order to explore the structure. You can also have a look at the *w3school XPath tutorial*[2] to solve the following tasks.

---

[1] The file can be downloaded from ILIAS but is also available here: `http://aiweb.cs.washington.edu/research/projects/xmltk/xmldata/data/mondial/mondial-3.0.xml`

[2] `https://www.w3schools.com/xml/xpath_intro.asp`

2.1 Starting point

Locate the XMLReader.java class in the package de.dwslab.lecture.wdi.xml. This class contains a main method which parses the mondial-3.0.xml file using the *JAXP* library and selects its root node.

2.2 Mondial — `Print XML node information`

TASK 1: Extend the XMLReader class and print the root node of the XML document.

SOLUTION: The following Java code reads the file and selects the node an root level. Then prints the name of the node.

Listing 1: Java class to parse XML and print the name of the root node.

```java
import java.io.IOException;

import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import javax.xml.parsers.ParserConfigurationException;
import javax.xml.xpath.XPath;
import javax.xml.xpath.XPathConstants;
import javax.xml.xpath.XPathExpression;
import javax.xml.xpath.XPathExpressionException;
import javax.xml.xpath.XPathFactory;

import org.w3c.dom.Document;
import org.w3c.dom.Node;
import org.xml.sax.SAXException;

public class 20200-RG2-Solution1 {

    public static void main(String[] args) throws ParserConfigurationException,
            SAXException, IOException, XPathExpressionException {
        // create the factory
        DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
        // create a new document builder
        DocumentBuilder builder = factory.newDocumentBuilder();
        // parse a document - make sure the file is located on root level
        Document doc = builder.parse("mondial-3.0.xml");

        // define an xpath expression
        XPathFactory xpathFactory = XPathFactory.newInstance();
        XPath xpath = xpathFactory.newXPath();
        // select the root nodes on root level
        XPathExpression expr = xpath.compile("/*");
        // parse the node
        Node root = (Node) expr.evaluate(doc, XPathConstants.NODE);
34                 // print the node name
35                 System.out.println(root.getNodeName());
36         }
37 }
```

Line 21 to 25 are actually reading the XML document. The following lines (28 to 31) build the XPath expression. As we want only the nodes on root level (which should be only one) we use the simple XPath expression: `/*` to select this particular node. Line 33 evaluates the expression against our document. As the selected object is a `Node`, we need to set this explicitly in `XPathExpression.evaluate(Document, XPathContants)`.
Answer: `mondial`

2.3 Mondial — `Schema Inspection`

Now that we have written our parser and explored the root node we can start digging deeper into the XML file.

TASK 2: Adapt the class from the previous task that a unique list of all nodes below the root node is printed.

SOLUTION: From the solution of the former task we exchange the code starting from line 30 with the following code.

Listing 2: Java snippet to select all nodes below the root and print them as a unique list.

```java
// select the level below the root
XPathExpression expr = xpath.compile("/mondial/*");
NodeList list = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
// helper to print unique node names
HashSet<String> uniqueNodes = new HashSet<String>();
for (int i = 0; i < list.getLength(); i++) {
        // if its not in the uniqueNodes set we did not print it yet
        if (!uniqueNodes.contains(list.item(i).getNodeName())) {
                uniqueNodes.add(list.item(i).getNodeName());
                System.out.println(list.item(i).getNodeName());
        }
}
```

In comparison, we now select all nodes below the root node `/mondial` and further expect a `NODESET`. We need to iterate through this set (which is not really a set, as it can include duplicates), and print only the names of a node, if the name was not printed in a former iteration.
Answer: `continent`, `country`, `organization`, `mountain`, `desert`, `island`, `river`, `sea`, `lake`

2.4 Mondial — `Basic XPath`

Now that we got an idea about the structure of the XML we are interested in the content.

TASK 3: Adapt the solution of the previous task in the way that it prints the names of all countries which belong to the continent with the name `Europe`. (Hint: Have a look at the schema of the node `country` to see how it is linked to the continent.)

SOLUTION: Based on the schema which is used within the XML the information about the continent can be found in the attribute `encompassed` within the `country` node. Here the continent is given with its identifier. (Europe has the identifier `f0_119`). This means a possible solution for the XPath query would be:
`/mondial/country[encompassed/@continent='f0_119']/@name`
In case we do not want to look the identifier up, we can also extend this predicate to:
`/mondial/country[encompassed/@continent=/mondial/continent [@name='Europe']/@id]/@name`
Update the code from the previous solution with the following code and enter the XPath query.

Listing 3: Java snippet to select all countries in Europe.

```
// select the countries, who are encompassed in a continent which
    has the name Europe
XPathExpression expr = xpath.compile("/mondial/country[encompassed/
    @continent=/mondial/continent[@name='Europe']/@id]/@name");
NodeList list = (NodeList) expr.evaluate(doc, XPathConstants.
    NODESET);
for (int i = 0; i < list.getLength(); i++) {
        System.out.println(list.item(i).getTextContent());
}
```

For each retrieved node, calling the `getTextContent()` function the name is returned.
Answer: You should retrieve a list of 51 country names, starting with `Albania` ending with `Turkey`.

2.5 Mondial — `XPath Predicates I`

With the solution of the previous task we are now able to get the countries for a selected continent. In a next step we want to extend this query that we can get countries which belong to two continents.

TASK 4: Extend the XPath for the former task in order to retrieve only countries which are part of `Europe` and `Asia`.

SOLUTION: Extending the XPath and adding a second condition using `and` where a node also needs to be encompassed in `Asia` leads to the following XPath: `/mondial/country[encompassed/@continent=/mondial/continent[@name='Europe']/@id and encompassed/@continent=/mondial/continent[@name='Asia']/@id]/@name`
Answer: `Russia` and `Turkey`

2.6 Mondial — `XPath Predicates II`

In a final step we want to gather all attributes from a selection of nodes, without explicitly knowing their names.

TASK 5: Extend the solution of the former task in order to navigate (using XPath) to the `country` node and print all attribute names and values. (Hint: You can use the `getAttributes()` method to detect all available attributes of the current node).

SOLUTION: In contrast to the former XPath query, we are not interested in the node itself, not the `name` attribute. This means we need to remove it from the end of the query. The XPath expression now returns two nodes, the one representing `Russia` and the one representing `Turkey`. Using the `getAttributes()` method on each of those both nodes allows us to get a `NamedNodeMap` including all attributes and values of this particular node. The following code prints out all this key-value pairs.

Listing 4: Java snippet to print all attributes of countries which are in Europe and Asia.

```java
// select the country nodes of countries which are in Europe and Asia
XPathExpression expr = xpath.compile("/mondial/country[encompassed/@continent=/mondial/continent[@name='Europe']/@id␣and␣encompassed/@continent=/mondial/continent[@name='Asia']/@id]");
NodeList list = (NodeList) expr.evaluate(doc, XPathConstants.NODESET);
// iterate over the country nodes
for (int i = 0; i < list.getLength(); i++) {
        System.out.println("New␣Country␣...");
        // get the node
        Node n = (Node) list.item(i);
        // get the attributes of the node
        NamedNodeMap map = n.getAttributes();
        // iterate over the attributes
        for (int j = 0; j < map.getLength(); j++) {
                // print them
                System.out.println(map.item(j).getNodeName() + ":"
                                + map.item(j).getTextContent());
        }
}
```

In this snippet the usage of the `getAttributes()` method is shown in line 10.
Answer: The output starts with the following lines:

```
New Country ...
capital:f0_1598
car_code:R
datacode:RS
gdp_agri:6
gdp_ind:41
...
```

3 JSON

In the second part of this exercise we focus on the JSON format. As you already have some experience with the *Mondial* dataset in a first step, you are asked to transform parts of the XML into a JSON. In order to do so make use of the `Google Gson` Java library[3].

3.1 Mondial — `XML to JSON`

TASK 6: Create a JSON file (*.json) which contains all countries which are located in Europe with the attributes of the `country` node from the original `mondial-*.xml`. (Hint: Have a look at the last exercise of the former section. `Gson` offers a method to simply translate a `HashMap` into a JSON string, which than can be written to a file.)

SOLUTION: Starting from the code of Task 4, we first need to adapt the XPath and remove the restriction, that the countries need to be in Europe and Asia, as we want all countries in Europe. We further need to store all attribute-name-value pairs in a `Map` which we can later transform into a JSON string using the `Gson` object. The following code creates a *.json file including all countries in Europe:

Listing 5: Java class to store countries from Europe from the *Mondial* XML file into a JSON file.

```java
// ... not all import is shown, due to space reasons

import com.google.gson.Gson;

public class WDI_20200_RG2_Solution6 {

        public static void main(String[] args) throws
            ParserConfigurationException,SAXException, IOException,
             XPathExpressionException {
                // create the factory
```

---

[3] You can find the library at the google code page: `https://sites.google.com/site/gson/`. A user guide can be found on this page: `https://github.com/google/gson/blob/master/UserGuide.md`

```
9           DocumentBuilderFactory factory =
                DocumentBuilderFactory.newInstance();
10          // create a new document builder
11          DocumentBuilder builder = factory.
                newDocumentBuilder();
12          // parse a document
13          Document doc = builder.parse("mondial-3.0.xml");
14          // define an xpath expression
15          XPathFactory xpathFactory = XPathFactory.
                newInstance();
16          XPath xpath = xpathFactory.newXPath();
17          // select the countries of Europe and all their
                attributes
18          XPathExpression expr = xpath.compile("/mondial/
                country[encompassed/@continent=/mondial/
                continent[@name='Europe']/@id]");
19          NodeList list = (NodeList) expr.evaluate(doc,
                XPathConstants.NODESET);
20          // create a gson object
21          Gson gson = new Gson();
22          // open a writer to write some output
23          BufferedWriter bw = new BufferedWriter(new
                FileWriter(new File("mondial-3.0-europe-
                countries.json")));
24          // iterate over all country nodes
25          for (int i = 0; i < list.getLength(); i++) {
26              // get the node
27              Node n = (Node) list.item(i);
28              // get the attributes of the node
29              NamedNodeMap map = n.getAttributes();
30              // create an empty hashmap
31              Map<String, String> values = new HashMap<
                    String, String>();
32              // iterate over the attributes of the node
33              for (int j = 0; j < map.getLength(); j++) {
34                  // add the attribute name and the
                        value to the map
35                  values.put(map.item(j).getNodeName
                        (), map.item(j).getTextContent
                        ());
36              }
37              // parse the hashmap to a json string
38              String jsonString = gson.toJson(values);
39              // write the string to the file
40              bw.write(jsonString + "\n");
41              // print the string to the console
42              System.out.println(gson.toJson(values));
43          }
44          // close the writer
45          bw.close();
46      }
47  }
```

The lines till 20 are similar to the exercises before. In line 21 the `Gson` object is initialized. In line 31 we create an empty `HashMap` object for the attribute-name-value pairs which we put into this map in line 35. This object is then transformed into a JSON string in line 38. The string is written to a file using

a `BufferedWriter` (which was initialized in line 23) in line 40. One line in this file look like:
`{"id":"f0_320","total_area":"1.9","infant_mortal...`

3.2 Mondial — `Reading JSON`

In a second step we want to create Java objects from the JSON file we just created, but we are not interested in all attributes.

TASK 7: Write a small program, which reads the JSON file (which was the output of the former task) and transforms each line into a Java object (named `Country.java`). The country should have four values: the `id` (String), the `name` (String), the `car_code` (String), and the `population` (Long). Do you have to pay attention to type conversion? What is the total number of inhabitants of those countries? (Hint: Have a look in the example code of the lecture.)

SOLUTION: First we need to generate a new Java class called `Country` with the four named attributes:

Listing 6: Java object Country.

```java
public class Country {
        String id;
        String name;
        String car_code;
        Long population;
}
```

In order to read the file we can use a `BufferedReader` and process each JSON object line by line (as we also stored it in that way). We then can parse the JSON string using a `Gson` object into the `Country.class` object.

Listing 7: Java class to parse the country JSON file and calculate the total population.

```java
import java.io.BufferedReader;
import java.io.File;
import java.io.FileReader;
import java.io.IOException;

import com.google.gson.Gson;

public class WDI_20200_RG2_Solution7 {

        public static void main(String[] args) throws IOException {
                // creat gson object
                Gson gson = new Gson();
                // create a reader
                BufferedReader br = new BufferedReader(new
                    FileReader(new File(
                                "src/main/resources/mondial-3.0-
                                    europe-cities.json")));
                // initalize total count of population
17                Long population_total = 0l;
18                // iterate through the file - line by line
19                while (br.ready()) {
20                        // read the line
21                        String jsonLine = br.readLine();
22                        // convert it to a country object
23                        Country country = gson.fromJson(jsonLine,
                              Country.class);
24                        // sum the population
25                        population_total += country.population;
26                }
27                // close the reader
28                br.close();
29                // print result
30                System.out.println("Total population is: " +
                      population_total);
31        }
32 }
```

As you can see, also within the JSON file itself the population is stored as string value, the `Gson` parser automatically tries to convert it to a Long value. To calculate the total population, for each line the population is selected from the `Country` object and added to the `total_population` value (see line 25).
Answer: The total population of those countries is: 792 002 189

4 RDF

In the last part of this exercise session we will focus on RDF and SPARQL. In ILIAS you can find the European countries with their name, population and the spoken languages stored as RDF file. The file was generated from the original *mondial* XML file.[4] In the following you will be asked to formulate SPARQL queries to answer questions about the dataset using the *Jena* Java Framework[5]. In addition to the lecture the *W3* site of *SPARQL Query Language* can help you to answer the questions.[6]

4.1 Mondial — `Query with SPARQL I`

TASK 8: Write a small program, which reads the RDF file (from ILIAS) and formulate a SPARQL query which returns the name and id of all countries within the dataset ordered by the name. What is the last country in this list. In order to explore the property names and namespaces have a look at the RDF file or at the code which was used to generate the file. (Hint: Have a look at the example code of the lecture.)

---

[4] The code which was used to generate the file can also be found in the Java project of this (see `de.dwslab.lecture.wdi.rdf.Converter.java`).

[5] The documentation of the framework can be found at their website: `https://jena.apache.org/`

[6] `https://www.w3.org/TR/rdf-sparql-query/`

SOLUTION: Following the example in the lecture slides we generate the model and read the input data before constructing the query and parsing the results.

Listing 8: Java class to read the country RDF and list all countries with their id.

```
 1 import com.hp.hpl.jena.query.Query;
 2 import com.hp.hpl.jena.query.QueryExecution;
 3 import com.hp.hpl.jena.query.QueryExecutionFactory;
 4 import com.hp.hpl.jena.query.QueryFactory;
 5 import com.hp.hpl.jena.query.QuerySolution;
 6 import com.hp.hpl.jena.query.ResultSet;
 7 import com.hp.hpl.jena.rdf.model.Model;
 8 import com.hp.hpl.jena.rdf.model.ModelFactory;
 9 import com.hp.hpl.jena.vocabulary.RDFS;
10
11 public class WDI_20200_RG2_Solution8 {
12
13         public static void main(String[] args) {
14                 // create RDF model
15                 Model model = ModelFactory.createDefaultModel();
16                 // fill the model with the data from the file
17                 model.read("mondial-3.0-europe-countries.rdf");
18                 // the sparql query to select the names and ids of
                       all countries
19                 // differentiate between the country and the
                       language entities by keeping the entities that
                       have a population property
20                 String queryString = "SELECT␣?country␣?label␣WHERE␣
                       {?country␣<"
21                                 + RDFS.label + ">␣?label␣.␣?country
                                       ␣<http://www.geonames.org/
                                       ontology#population>␣?pop␣}␣
                                       ORDER␣BY␣?label";
22                 // create the query
23                 Query query = QueryFactory.create(queryString);
24                 QueryExecution qe = QueryExecutionFactory.create(
                       query, model);
25                 // execute the query
26                 ResultSet results = qe.execSelect();
27                 // parse the results
28                 while (results.hasNext()) {
29                         QuerySolution sol = results.next();
30                         System.out.println(sol.get("label").
                               toString() + "\t" + sol.get("country").
                               toString());
31                 }
32         }
33 }
```

In our query (line 20) we make use of the predefined property `RDFS.label` which is included in the *Jena* library to formulate our query. We could also simply use the property itself: `http://www.w3.org/2000/01/rdf-schema\#label`. In order to differentiate between the language and the country entities we select those that have a `population` property. Between line 26 and 29 the code iterates over the set of results and collects from each result (`QuerySolution`) the

attributes (`label` and `country`) as we have defined them in the query.
Answer: The last country in the list is *United Kingdom (http://dwslab.de/wdi/country#f0_418)*.

4.2 Mondial — `Query with SPARQL II`

As we now have setup the code to query against our dataset, we are interested in the largest countries. But as we already know that Russia and Germany are pretty large, we want to generate a list of the second top 5 largest countries by population.

TASK 9: What is the SPARQL query which returns the second five (6th to 10th) most populated countries in Europe? And which countries are these?

SOLUTION: Based on the SPARQL query of the former exercise we need to adopt the code starting from line 20 and change it to the following:

Listing 9: Java class to read the country RDF and list the second 5 most populated countries.

```java
// the sparql query to select the second five most populated countries
String queryString = "SELECT␣?country␣?label␣?population␣WHERE␣{?country␣<"      + RDFS.label + ">␣?label␣.␣?country␣<http://www.geonames.org/ontology#population>␣?population␣.}␣ORDER␣BY␣DESC(?population)␣OFFSET␣5␣LIMIT␣5";
// create the query
Query query = QueryFactory.create(queryString);
QueryExecution qe = QueryExecutionFactory.create(query, model);
// execute the query
ResultSet results = qe.execSelect();
// parse the results
while (results.hasNext()) {
        QuerySolution sol = results.next();
        System.out.println(sol.get("label").toString() + "\t" + sol.get("country").toString() + "\t" + sol.get("population").asLiteral().getLong());
}
```

Within the SPARQL query we make use of `OFFSET` and `LIMIT` to skip the first five entries and limit the list to five entries. Beforehand we order the list descending. To print the population (which is marked as `Long` datatype within the data, we make use of the `getLong()` function of the `Literal`.
Answer: The code would create the following output:

```
Italy     http://dwslab.de/wdi/country#f0_268   57460272
Ukraine   http://dwslab.de/wdi/country#f0_411   50864008
Spain     http://dwslab.de/wdi/country#f0_385   39181112
Poland    http://dwslab.de/wdi/country#f0_337   38642564
Romania   http://dwslab.de/wdi/country#f0_351   21657162
```

4.3 Mondial — `Query with SPARQL III`

In a last exercise you are asked to write a SPARQL query which selects all countries whose inhabitants speak a defined language.

TASK 10: How does the SPARQL query look like, which returns a list of all German-speaking countries with their name and id?

SOLUTION: To answer the question, we need to extend the SPARQL using a `FILTER` which limits the returned solutions to those where we know they speak German:

Listing 10: Java class to read the country RDF and list the second 5 most populated countries.

```
String queryString = "SELECT␣?country␣?label␣WHERE␣{?country␣<" +
    RDFS.label + ">␣?label␣.␣?country␣<" + DCTerms.language + ">␣?
    language␣.␣?language␣<" + RDFS.label + ">␣?languageName␣.␣
    FILTER(?languageName=\"German\")}";
```

Answer: The code would create the following output:

```
Austria       http://dwslab.de/wdi/country#f0_149
Switzerland   http://dwslab.de/wdi/country#f0_404
Belgium       http://dwslab.de/wdi/country#f0_162
Germany       http://dwslab.de/wdi/country#f0_220
```