Web Data Integration

Reading Guide 2: Data Exchange Formats

Authors:

Prof. Dr. Christian Bizer

Dr. Robert Meusel

University of Mannheim, Germany

Module 20200
Web Data Integration

Reading Guide 2: Data Exchange Formats

Authors:
Prof. Dr. Christian Bizer
Dr. Robert Meusel

2. edition

University of Mannheim, Germany

Faculty of Business Informatics and Mathematics
Chair: Business Informatics V
Course of Study: Data Science
68131 Mannheim

2. edition (2017-04-02)

Didactical and editorial processing:
Johannes Maneljuk, M.A. (Rhetoric/German studies)

To facilitate reading, only the masculine form is used in this document; all references to the male gender shall be deemed and construed to include the female gender.

Contents

Introduction to the Reading Guides

I. Icons and Colour Codes

Excursion	↗
Control task	✓
Source Code	</>
Video	▶

Reading Guide 2 Data Exchange Formats

data exchange

The term *data exchange* refers to the process of transferring data from one system to another. The formats (syntaxes) that are used to represent data while it is being transferred from system A to system B are called *data exchange formats*.

data exchange formats

A wide variety of different data exchange formats are used, which constitutes one of the reasons for the technical heterogeneity. Data exchange formats that are widely used in the context of the Web are: *Comma-separated Values* (CSV), *Extensible Markup Language* (XML), *JavaScript Object Notation* (JSON), and *Resource Description Framework* (RDF).

character encoding

The information inside a document is encoded using a specific character encoding schema. Within the document/file, each character is encoded by a bit sequence. The mapping between this sequence of 0s and 1s and the *real* character depends on the encoding and, in case the wrong encoding is used to read the data (due to lack of knowledge or misinformation), the data will be corrupted in the data exchange process. Thus, beside of the agreeing on a data exchange format, it is also necessary to agree on a specific character encoding in order to exchange data between systems smoothly.

2.1 Lecture: Data Exchange Formats

Please watch the video lectures *Data Exchange Formats - Part I* and *Data Exchange Formats - Part II* by Prof. Dr. Christian Bizer. The video lectures cover the data exchange formats *Comma-separated Values* (CSV), *Extensible Markup Language* (XML), *JavaScript Object Notation* (JSON), and *Resource Description Framework* (RDF). The lectures explain the basic structure as well as the advantages and disadvantages of each format. The lecture also provides code examples on how to read data that is represented using the different formats from within Java applications. For extracting subset of the data and answer queries against the data, the lecture covers the query languages XPath for XML data and SPARQL for RDF data.

Video 2.1: Lecture: *Data Exchange Formats - Part I*

Video: 20200-RG2-V1
Corresponding Set of Slides: 20200-RG2-S1

The second part of the lecture introduces the formats JSON and RDF as well as the query language SPARQL.

Video 2.2: Lecture: *Data Exchange Formats - Part II*

Video: 20200-RG2-V2
Corresponding Set of Slides: 20200-RG2-S2

2.2 Exercises: Data Exchange Formats

In the following three exercises, you are asked to write Java code for reading data from XML-, JSON- and RDF files and for querying the data using the XPath and SPARQL query languages. Each subsection is dedicated to one of the three data exchange formats. The tasks are rather basic and the goal is to refresh your knowledge in Java in general and in particular in parsing those formats. The used data sets are not related to the tasks of the next teaching letters and are selected solely to demonstrate different characteristics of the selected data exchange formats.

Besides the solution which is given at the end of this teaching letter, we also provide the solutions as a Java project, which should allow you to compare your own code and execute the sample solution. The project can be downloaded from the learning platform ILIAS and is just one out of many possible correct solutions.

Excursion 2.1: Install Eclipse IDE

In case you want to use an IDE to solve the following tasks, we recommend the usage of Eclipse. You also can make use of other IDEs but we might not be able to support you in case of problems.

Download the installer for your operating system from the download page of the eclipse website[1]. Start the installer and follow the instructions. For the exercises of this and the following projects it is enough to install the *Eclipse IDE for Java Developers*. Finish the installation process and you are ready to create your first Java projects using the Eclipse IDE.

2.2.1 XML

This subsection is dedicated to the XML format. In particular, you are asked to perform *XPath* queries on the Mondial dataset[2]. 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*[3] to solve the following tasks.

Control task 2.1: Mondial – `Read XML`

In order to get started with automated parsing of XML, write a Java class which reads the file using the *JAXP* library and prints the the root node of the XML document. (Hint: Have a look at the lecture slides in order to get an idea how to start.)

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

Control task 2.2: Mondial – `Schema Inspection`

Adopt the class from the former task so that a unique list of all nodes below the root node is printed.

Since we got an idea about the structure of the XML, we are now interested in the content.

[1] `http://eclipse.org/downloads/`

[2] The file can be downloaded from ILIAS, but is also available here: `http://www.cs.washington.edu/research/xmldatasets/www/repository.html#mondial`

[3] `http://www.w3schools.com/xsl/xpath_intro.asp`

Control task 2.3: Mondial – `Basic XPath` ☑

Adopt the solution of the former task in a way that prints the names of all countries which are contained in 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.)

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

Control task 2.4: Mondial – `XPath Predicates` ☑

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

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

Control task 2.5: Mondial – `XPath Predicates` ☑

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

The solutions for the former exercises can also be found in the Java project for this teaching letter, which can be downloaded from ILIAS. Each exercise/solution has a dedicated class including a `main`-function which can be executed directly within your IDE.

2.2.2 JSON

In the second part of this teaching letter's exercises, 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 to JSON. In order to do so, make use of the `Google Gson` Java library[4].

Control task 2.6: Mondial – `XML to JSON` ☑

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 then can be written to a file.)

[4] 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://sites.google.com/site/gson/gson-user-guide`

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.

Control task 2.7: Mondial – `Reading JSON`

Write a small program that 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: `id` (String), `name` (String), `car_code` (String), and `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 at the example code given in the lecture.)

The solutions for the former exercises can also be found in the Java project for this teaching letter, which can be downloaded from ILIAS. Each exercise/solution has a dedicated class including a `main`-function which can be executed directly within your IDE.

2.2.3 RDF

In the last part of this teaching letter's exercise section, we will focus on RDF and SPARQL. In ILIAS, you can find the European countries together with their name, population and the spoken languages, stored as RDF file. The file was generated from the original *Mondial* XML file.[5] In the following, you will be asked to formulate SPARQL queries to answer questions about the dataset using the *Jena* Java Framework[6]. In addition to the lecture, the site of *SPARQL Query Language* and the site of *W3* can help you to answer the questions.[7]

Control task 2.8: Mondial – `Query with SPARQL I`

Write a 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 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 given in the lecture.)

As we now have set up the code to query against our dataset, we are interested in countries with a high population. But as we already know that Russia and Germany are pretty strong populated (in absolute numbers), we want to leave out the first five most populated countries and return the top 6 to 10.

Control task 2.9: Mondial – `Query with SPARQL II`

Which SPARQL query returns theses five countries? And which countries are these?

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

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

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

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

Control task 2.10: Mondial – `Query with SPARQL III`

How does a SPARQL query look like which returns a list of all German-speaking countries together with their `name` and `id`?

The solutions for the former exercises can also be found in the Java project for this teaching letter, which can be downloaded from ILIAS. Each exercise/solution has a dedicated class including a `main`-function which can be executed directly within your IDE.

2.3 Optional Reading Material: Data Exchange Formats

Having learned about the most common data exchange formats on the Web, the following pages/chapters can help you to gain additional, optional knowledge about data exchange formats:

- Doan et al.: Principles of Data Integration, Chapter 11.
- Leser and Naumann: Informationsintegration, Chapter 2.1.
- Harold & Means: XML in a Nutshell. O'Reilly.

The following online tutorials also cover the different formats:

- **GSON**: `http://code.google.com/p/google-gson/`
- **RDF**: `https://www.w3.org/TR/rdf-primer/`
- **JENA**: `http://jena.apache.org/documentation/`
- Euclid Curriculum covering **SPARQL**: `http://www.euclid-project.eu/`

List of Solutions to the Control Tasks

Solution to control task 2.1 on page 6

The following Java code reads the file and selects the node at an root level. Then, it prints the node's name.

Source Code 2.1: Java class to parse XML and print the name of the root node.

```
1  import java.io.IOException;
2  import javax.xml.parsers.DocumentBuilder;
3  import javax.xml.parsers.DocumentBuilderFactory;
4  import javax.xml.parsers.ParserConfigurationException;
5  import javax.xml.xpath.XPath;
6  import javax.xml.xpath.XPathConstants;
7  import javax.xml.xpath.XPathExpression;
8  import javax.xml.xpath.XPathExpressionException;
9  import javax.xml.xpath.XPathFactory;
10 import org.w3c.dom.Document;
11 import org.w3c.dom.Node;
12 import org.xml.sax.SAXException;
13
14 public class 20200-RG2-Solution1 {
15
16   public static void main(String[] args) throws
         ParserConfigurationException, SAXException, IOException,
         XPathExpressionException {
17     // create the factory
18     DocumentBuilderFactory factory = DocumentBuilderFactory.
           newInstance();
19     // create a new document builder
20     DocumentBuilder builder = factory.newDocumentBuilder();
21     // parse a document - make sure the file is located on root
             level
22     Document doc = builder.parse(”mondial-3.0.xml”);
23
24     // define an xpath expression
25     XPathFactory xpathFactory = XPathFactory.newInstance();
26     XPath xpath = xpathFactory.newXPath();
27     // select the root nodes on root level
28     XPathExpression expr = xpath.compile(”/*”);
29     // parse the node
30     Node root = (Node) expr.evaluate(doc, XPathConstants.NODE);
31     // print the node name
32     System.out.println(root.getNodeName());
33 }
34 }
```

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 the 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`

Solution to control task 2.2 on page 6

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

Source Code 2.2: 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`

Solution to control task 2.3 on page 7

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 together 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`

If we do not want to look up the identifier, we can also extend this predicate to:
`/mondial/country[encompassed/`
`@continent=/mondial/continent[@name=’Europe’]/@id]/@name`

Update the code from the former solution with the following code and enter the XPath query.

Source Code 2.3: Java snippet to select all countries in Europe.

```java
// 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 returns the name.

Answer: You should retrieve a list of 51 country names, starting with `Albania` and ending with `Turkey`.

Solution to control task 2.4 on page 7

Extending the XPath and adding a second condition using `and`, where a node also needs to be encompassed in `Afrika`, 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`

Solution to control task 2.5 on page 7

In contrast to the former XPath query, we are now 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 other representing `Turkey`. Using the `getAttributes()` method on each of those two nodes allows us to get a `NamedNodeMap` including all attributes and values of this particular node. The following code returns all of these key-value pairs.

Source Code 2.4: Java snippet to print all attributes of countries which are in Europe and Asia.

```
1 // select the country nodes of countries which are in Europe
      and Asia
2 XPathExpression expr = xpath.compile("/mondial/country[
      encompassed/@continent=/mondial/continent[@name='Europe']/
      @id␣and␣encompassed/@continent=/mondial/continent[@name='
      Asia']/@id]");
3 NodeList list = (NodeList) expr.evaluate(doc, XPathConstants.
      NODESET);
4 // iterate over the country nodes
5 for (int i = 0; i < list.getLength(); i++) {
6   System.out.println("New␣Country␣...");
7   // get the node
8   Node n = (Node) list.item(i);
9   // get the attributes of the node
10  NamedNodeMap map = n.getAttributes();
11  // iterate over the attributes
12  for (int j = 0; j < map.getLength(); j++) {
13    // print them
14    System.out.println(map.item(j).getNodeName() + ":"
15        + map.item(j).getTextContent());
16  }
17 }
```

In this snippet, the usage of the `getAttributes()` method is shown in line 10.

Answer: The output starts with the following lines:

```
New Node ...
capital:f0\_1598
car\_code:R
datacode:RS
gdp\_agri:6
gdp\_ind:41
...
```

Solution to control task 2.6 on page 7

Starting from the code of Task 2.5, 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:

Source Code 2.5: Java class to store countries from Europe from the *Mondial* XML file into a JSON file.

```
// ... 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
    DocumentBuilderFactory factory = DocumentBuilderFactory.
        newInstance();
    // create a new document builder
    DocumentBuilder builder = factory.newDocumentBuilder();
    // parse a document
    Document doc = builder.parse(”mondial-3.0.xml”);
    // define an xpath expression
    XPathFactory xpathFactory = XPathFactory.newInstance();
    XPath xpath = xpathFactory.newXPath();
    // select the countries of Europe and all their attributes
    XPathExpression expr = xpath.compile(”/mondial/country[
        encompassed/@continent=/mondial/continent[@name=’Europe
        ’]/@id]”);
    NodeList list = (NodeList) expr.evaluate(doc,
        XPathConstants.NODESET);
    // create a gson object
    Gson gson = new Gson();
    // open a writer to write some output
    BufferedWriter bw = new BufferedWriter(new FileWriter(new
        File(”mondial-3.0-europe-cities.json”)));
    // iterate over all country nodes
    for (int i = 0; i < list.getLength(); i++) {
      // get the node
      Node n = (Node) list.item(i);
      // get the attributes of the node
      NamedNodeMap map = n.getAttributes();
      // create an empty hashmap
      Map<String, String> values = new HashMap<String, String
          >();
      // iterate over the attributes of the node
      for (int j = 0; j < map.getLength(); j++) {
        // add the attribute name and the value to the map
        values.put(map.item(j).getNodeName(), map.item(j).
            getTextContent());
      }
      // parse the hashmap to a json string
      String jsonString = gson.toJson(values);
      // write the string to the file
      bw.write(jsonString + ”\n”);
      // print the string to the console
      System.out.println(gson.toJson(values));
    }
    // close the writer
    bw.close();
  }
}
```

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 looks like `{”id”:”f0_320”,”total_area”:”1.9”,”infant_mortal...`.

Solution to control task 2.7 on page 8

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

Source Code 2.6: 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). Then, we can parse the JSON string into the `Country.class` object using a `Gson` object.

Source Code 2.7: Java class to parse the country JSON file and calculate the total population.

```
1  import java.io.BufferedReader;
2  import java.io.File;
3  import java.io.FileReader;
4  import java.io.IOException;
5  import com.google.gson.Gson;
6
7  public class WDI_20200_RG2_Solution7 {
8
9    public static void main(String[] args) throws IOException {
10     // creat gson object
11     Gson gson = new Gson();
12     // create a reader
13     BufferedReader br = new BufferedReader(new FileReader(new
           File(
14         ”src/main/resources/mondial-3.0-europe-cities.json”)));
15     // initalize total count of population
16     Long population_total = 0l;
17     // iterate through the file - line by line
18     while (br.ready()) {
19       // read the line
20       String jsonLine = br.readLine();
21       // convert it to a country object
22       Country country = gson.fromJson(jsonLine, Country.class);
23       // sum the population
24       population_total += country.population;
25     }
26     // close the reader
27     br.close();
28     // print result
29     System.out.println(”Total␣population␣is:␣” +
           population_total);
30   }
31 }
```

As you can see, also within the JSON file itself the population is stored as string value, and 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

Solution to control task 2.8 on page 8

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

Source Code 2.8: 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     String queryString = ”SELECT␣?country␣?label␣WHERE␣{?
           country␣<”  + RDFS.label + ”>␣?label}␣ORDER␣BY␣?label”;
20     // create the query
21     Query query = QueryFactory.create(queryString);
22     QueryExecution qe = QueryExecutionFactory.create(query,
           model);
23     // execute the query
24     ResultSet results = qe.execSelect();
25     // parse the results
26     while (results.hasNext()) {
27       QuerySolution sol = results.next();
28       System.out.println(sol.get(”label”).toString() + ”\t” +
             sol.get(”country”).toString());
29     }
30   }
31 }
```

In our query (line 19), 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. Between line 26 and 29, the code iterates over the set of results and collects the attributes (`label` and `country`) from each result (`QuerySolution`) 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.

Solution to control task 2.9 on page 8

Based on the SPARQL query of the former exercise, we need to adopt the code starting from line 19 and change it to the following:

Source Code 2.9: Java class to read the country RDF and list rank 6 to 10 of the most populated countries.

```
// 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
```

Solution to control task 2.10 on page 9

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:

Source Code 2.10: 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`

Indexes

I. Excursions

II. Control tasks

III. Tables

IV. Videos

V. Bibliography

Index