UNIVERSITY OF JYVÄSKYLÄ

Lecture 5: Programming with Semantic Web
(RDF4J and Jena APIs)

TIES4520 Semantic Technologies for Developers
Autumn 2018

University of Jyväskylä

Khriyenko Oleksiy

Part 1

RDF4J (Sesame) API

RDF4J

- ***RDF4J*** (formerly known as Sesame) is an open source Java framework for processing RDF data. (*http://rdf4j.org/*)
  - includes *parsing*, *storing*, *inferencing* and *querying*.
  - offers two out-of-the-box RDF databases (the in-memory store and the native store), and in addition many third party storage solutions are available.
  - offers API that can be connected to all leading RDF storage solutions.
  - RDF4J-compatible RDF databases (*http://rdf4j.org/about/rdf4j-databases/*)
  - fully supports the SPARQL 1.1 query and update language for expressive querying. Allows you to connect with SPARQL endpoints and create applications that leverage the power of linked data and Semantic Web.
  - supports all mainstream RDF file formats, including RDF/XML, Turtle, N-Triples, N-Quads, JSON-LD, TriG and TriX.
  - available via *Apache Maven*, or download the SDK or onejar directly (*http://rdf4j.org/download/*).
  - Getting Started with RDF4J, Maven, and Eclipse (*http://docs.rdf4j.org/getting-started/*)

```
// gets you the entire RDF4J core framework
<dependency>
   <groupId>org.eclipse.rdf4j</groupId>
   <artifactId>rdf4j-runtime</artifactId>
   <version>${rdf4j.version}</version>
</dependency>
…
```

```
// includes the BOM (Bill Of Materials) into the project
<dependencyManagement>
 <dependencies>
  <dependency>
   <groupId>org.eclipse.rdf4j</groupId>
   <artifactId>rdf4j-bom</artifactId>
   <version>2.4.1</version>
   <type>pom</type>
   <scope>import</scope>
  </dependency>
 </dependencies>
</dependencyManagement>
```

RDF4J API: RDF Model API

- The core of the RDF4J framework is the ***RDF Model API***
  - defines how the building blocks of RDF (statements, URIs, blank nodes, literals, graphs and models) are represented
  - *org.eclipse.rdf4j.model.Statement* interface represents RDF Statement
  - subject, predicate, object and (optionally) context are *org.eclipse.rdf4j.model.Value* interface
  - *Value* interface is further specialized into *org.eclipse.rdf4j.model.Resource*, and *org.eclipse.rdf4j.model.Literal* interfaces
  - *Resource* represents any RDF value that is either a *blank node* or a *URI*:
    - *org.eclipse.rdf4j.model.URI*
    - *org.eclipse.rdf4j.model.BNode*
  - *Literal* represents RDF literal values (strings, dates, integer numbers, etc.)

RDF4J API: RDF Model API

- To create new values and statements use:
  - default *ValueFactory* implementation

```
import org.eclipse.rdf4j.model.ValueFactory;
import org.eclipse.rdf4j.model.impl.SimpleValueFactory;
ValueFactory factory = SimpleValueFactory.getInstance();
```

  - *ValueFactory* obtained from Repository you are working with (recommend)

```
ValueFactory factory = myRepository.getValueFactory();
```

Example:

```
URI bob = factory.createURI("http://example.org/bob");
URI name = factory.createURI("http://example.org/name");
Literal bobsName = factory.createLiteral("Bob");
Statement nameStatement = factory.createStatement(bob, name, bobsName);
```

- *RDF Model API* also provides pre-defined URIs for several well-known vocabularies, such as *RDF*, *RDFS*, *OWL*, *DC*, *FOAF*, and more.
  - These constants can all be found in the *org.eclipse.rdf4j.model.vocabulary* package

Example:

```
URI bob = factory.createURI("http://example.org/bob");
Statement typeStatement = factory.createStatement(bob, RDF.TYPE, FOAF.PERSON);
```

RDF4J API: RDF Model API

- In order to deal with *collections* of RDF statements, we can use the *org.eclipse.rdf4j.model.Model* interface
  - is an extension of the default Java Collection class *java.util.Set<Statement>.* You can use a Model like any other Java collection in your code.
  - provides two default implementations of the Model interface: *org.eclipse.rdf4j.model.impl.LinkedHashModel*, and *org.eclipse.rdf4j.model.impl.TreeModel*.

```
// create a new Model to put statements in
Model model = new LinkedHashModel();
// add an RDF statement
model.add(typeStatement);
// add another RDF statement by simply providing subject, predicate, and object.
model.add(bob, name, bobsName);
// iterate over every statement in the Model
for (Statement statement: model) { ... }
```

Model offers a number of useful methods to quickly get subsets of statements and otherwise search/filter your collection of statements:

  - to quickly iterate over all statements that make a resource an instance of the class *foaf:Person*

```
for (Statement typeStatement: model.filter(null, RDF.TYPE, FOAF.PERSON)) { ... }
```

  - to immediately iterate over all subject-resources that are of type *foaf:Person* and then retrieve each person's name

```
for (Resource person: model.filter(null, RDF.TYPE, FOAF.PERSON).subjects()) {
  // get the name of the person (if it exists)
  Optional<Literal> name = Models.objectLiteral(model.filter(person, FOAF.NAME, null));
...
}
```

RDF4J API: RDF Model API

- The *ModelBuilder* provides a fluent API to quickly and efficiently create RDF models programmatically.

```
ModelBuilder builder = new ModelBuilder();

// set some namespaces
builder.setNamespace("ex", "http://example.org/").setNamespace(FOAF.NS);

builder.namedGraph("ex:graph1")      // add a new named graph to the model
   .subject("ex:john")        // add  several statements about resource ex:john
      .add(FOAF.NAME, "John")  // add triple (ex:john foaf:name "John") to the named graph
      .add(FOAF.AGE, 42)
      .add(FOAF.MBOX, "john@example.org");

// add a triple to the default graph
builder.defaultGraph()
   .add("ex:graph1", RDF.TYPE, "ex:Graph");

// return the Model object
Model m = builder.build();
```

To model closed lists of items, RDF provides a Collection vocabulary. More about handling (creation, extraction, copying or deleting) an RDF Collection you may find from the documentation.

RDF4J API: Repository API

- ***Repository API*** - central access point for repositories:
  - gives a developer-friendly access point to RDF repositories
  - offers various methods for querying and updating the data
  - interfaces for the Repository API can be found in package *org.eclipse.rdf4j.repository* *(several implementations for these interface exist in various sub-packages)*
- The main three implementations of *Repository* interface:
  - *org.eclipse.rdf4j.repository.sail.SailRepository* is a *Repository* that operates directly on top of a *Sail*. This is the class most commonly used when accessing/creating a local RDF4J repository. An important thing to remember is that the behavior of a repository is determined by the Sail(s) that it operates on; for example, the repository will only support RDF Schema or OWL semantics if the Sail stack includes an inferencer for this.
  - *org.eclipse.rdf4j.repository.http.HTTPRepository* is a *Repository* implementation that acts as a proxy to a RDF4J repository available on a remote RDF4J(Sesame) server, accessible through HTTP.
  - *org.eclipse.rdf4j.repository.sparql.SPARQLRepository* is a *Repository* implementation that acts as a proxy to any remote SPARQL endpoint (whether that endpoint is implemented using RDF4J or not).

RDF4J API: Repository API

- Create and initialize a *non-inferencing main-memory repository*
  - the content will be lost when the object is garbage collected or when your Java program is shut down

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sail.SailRepository;
import org.eclipse.rdf4j.sail.memory.MemoryStore;

...

Repository repo = new SailRepository(new MemoryStore());
repo.initialize();
```

  - the *MemoryStore* will write its contents to the directory so that it can restore it when it is re-initialized in a future session

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sail.SailRepository;
import org.eclipse.rdf4j.sail.memory.MemoryStore;

...

File dataDir = new File("C:\\temp\\myRepository\\");
MemoryStore memStore = new MemoryStore(dataDir);
memStore.setSyncDelay(1000L);

Repository repo = new SailRepository(memStore);
repo.initialize();
```

RDF4J API: Repository API

- Creating a *Native RDF Repository*
  - does not keep data in main memory, but instead stores it directly to disk

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sail.SailRepository;
import org.eclipse.rdf4j.sail.nativerdf.NativeStore;
...
File dataDir = new File("/path/to/datadir/");
String indexes = "spoc,posc,cosp";
Repository repo = new SailRepository(new NativeStore(dataDir, indexes));
repo.initialize();
```

- Creating a repository with *RDF Schema inferencing*
  - *ForwardChainingRDFSInferencer* is a generic RDF Schema inferencer (*MemoryStore* and *NativeStore* support it)

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sail.SailRepository;
import org.eclipse.rdf4j.sail.memory.MemoryStore;
import org.eclipse.rdf4j.sail.inferencer.fc.ForwardChainingRDFSInferencer;
...
Repository repo = new SailRepository( new ForwardChainingRDFSInferencer( new MemoryStore() ) );
repo.initialize();
```

- Accessing a *server-side repository*

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.http.HTTPRepository;
...
String rdf4jServer = "http://example.org/rdf4j-server/";
String repositoryID = "example-db";
Repository repo = new HTTPRepository(rdf4jServer, repositoryID);
repo.initialize();
```

RDF4J API: Repository API

- Accessing a *SPARQL endpoint*
  - use the Repository interface to access any SPARQL endpoint

```
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sparql.SPARQLRepository;
...
String sparqlEndpoint = "http://example.org/sparql";
Repository repo = new SPARQLRepository(sparqlEndpoint);
repo.initialize();
```

- Management of repositories
  - with *RepositoryManager* and *RepositoryProvider* it is possible to create one central location where all information on several repositories in use (including id, type, directory for persistent data storage, etc.) is kept. RepositoryManager comes in two flavors: *org.eclipse.rdf4j.repository.manager.LocalRepositoryManager* manages repository handling for you locally, and is always created using a (local) directory; *org.eclipse.rdf4j.repository.manager.RemoteRepositoryManager* is used to create and manage Sesame repositories residing on a remotely running Sesame server.
  - it is possible to create a virtual repository that is a federation of existing repositories using *RepositoryManagerFederator* class.
  - check the documentation for more information and examples…

RDF4J API: Repository API

- Creating a repository with *Custom Rule inferencing*

```
import org.eclipse.rdf4j.query.QueryLanguage;
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.sail.SailRepository;
import org.eclipse.rdf4j.sail.memory.MemoryStore;
import org.eclipse.rdf4j.sail.inferencer.fc.CustomGraphQueryInferencer;
...
String pre = "PREFIX : <http://foo.org/bar#>\n";
String rule = pre + "CONSTRUCT { ?p :relatesTo :Cryptography } WHERE " +
                "{ { :Bob ?p :Alice } UNION { :Alice ?p :Bob } }";
String match = pre + "CONSTRUCT { ?p :relatesTo :Cryptography } " +
                 "WHERE { ?p :relatesTo :Cryptography }";
Repository repo = new SailRepository(new CustomGraphQueryInferencer(
                      new MemoryStore(), QueryLanguage.SPARQL, rule, match));
```

data sample given in the Turtle format

```
@prefix : <http://foo.org/bar#> .
:Bob   :exchangesKeysWith :Alice .
:Alice :sendsMessageTo    :Bob .
```

repository will also automatically have the following inferred statements

```
@prefix : <http://foo.org/bar#> .
:exchangesKeysWith :relatesTo :Cryptography .
:sendsMessageTo    :relatesTo :Cryptography .
```

The SPARQL graph query in '***rule***' defines a pattern to search on, and the inferred statements to add to the repository.

The SPARQL graph query in '***match***' is needed to decide what inferred statements already exist that may need to be removed when the normal repository contents change.

RDF4J API: Adding RDF to a repository

- The *Repository API* offers various methods for adding data to a repository. Data can be added by specifying the location of a file that contains RDF data, and statements can be added individually or in collections.
- Operations on a repository are performed through a *RepositoryConnection* (*org.eclipse.rdf4j.repository.RepositoryConnection*) requested from the repository. It allows perform various operations, such as *query evaluation*, *getting*, *adding*, or *removing* statements, etc.

```
import org.eclipse.rdf4j.RDF4JException;
import org.eclipse.rdf4j.repository.Repository;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4j.rio.RDFFormat;
import java.io.File;
import java.net.URL;
...
File file = new File("/path/to/example.rdf");
String baseURI = "http://example.org/example/local";
try {   RepositoryConnection con = repo.getConnection();
   try {   con.add(file, baseURI, RDFFormat.RDFXML);
           URL url = new URL("http://example.org/example/remote.rdf");
           con.add(url, url.toString(), RDFFormat.RDFXML);
   }
   finally {   con.close(); }
}
catch (RDF4JException e) {}
catch (java.io.IOException e) {}
```

RDF4J API: Statement manipulating

- Creating individual statements

```
import org.eclipse.rdf4j.model.vocabulary.RDF;
import org.eclipse.rdf4j.model.vocabulary.RDFS;
...
ValueFactory f = myRepository.getValueFactory();
URI alice = f.createURI("http://example.org/people/alice");
URI name = f.createURI("http://example.org/ontology/name");
URI person = f.createURI("http://example.org/ontology/Person");
Literal alicesName = f.createLiteral("Alice");
try {   RepositoryConnection con = myRepository.getConnection();
        try {
                con.add(alice, RDF.TYPE, person);
                con.add(alice, name, alicesName);
} finally {   con.close();   }
} catch (RDF4JException e) {}
```

- Retrieving statements (for example: all statements about Alice)

```
RepositoryResult<Statement> statements = con.getStatements(alice, null, null, true);
try {   while (statements.hasNext()) {
           Statement st = statements.next();
           ... // do something with the statement
        }
} finally {   statements.close(); // make sure the result object is closed properly }
```

- Remove statements

```
con.remove(alice, name, alicesName);
```

```
con.remove(alice, null, null);
```

```
con.remove(con.getStatements(alice, null, null, true));
```

RDF4J API: Statement manipulating

- Use of *named graphs*/*contexts*
  - RDF4J supports the notion of context, which you can think of as a way to group sets of statements together through a single group identifier (this identifier can be a blank node or a URI).

```
String location = "http://example.org/example/example.rdf";
String baseURI = location;
URL url = new URL(location);
URI context1 = f.createURI(location);
conn.add(url, baseURI, RDFFormat.RDFXML, context1);
// Get all statements in the context “context1”
try (RepositoryResult<Statement> result = conn.getStatements(null, null, null, context1)) {
   while (result.hasNext()) {
      Statement st = result.next();
      ... // do something interesting with the result
}}
IRI context2 = f.createIRI("http://example.org/context2");
conn.add(bob, RDF.TYPE, person, context2);
conn.add(bob, name, bobsName, context2);
```

  - the context parameter is a *vararg*, meaning that you can specify an arbitrary number (zero, one, or more) of context identifiers. This way, you can combine different contexts together.

```
// Get all statements in either “context1” or “context2”
RepositoryResult<Statement> result = con.getStatements(null, null, null, context1, context2);
// Get all statements that do not have an associated context
RepositoryResult<Statement> result = con.getStatements(null, null, null,(Resource)null);
RepositoryResult<Statement> result = con.getStatements(null, null, null,(Resource)null, context1);
// Get all statements in the repository, ignoring any context information
RepositoryResult<Statement> result = con.getStatements(null, null, null);
```

RDF4J API: Model&Collection&Repository(Connection)

- Retrieve the statements from Repository, put them into the Model and remove from Repository

```
RepositoryResult<Statement> statements = con.getStatements(alice, null, null);
Model aboutAlice = QueryResults.asModel(statements);
con.remove(aboutAlice);
```

- insert any RDF Collection into your Repository

```
Model rdfList = ... ;
try (RepositoryConnection conn = repo.getConnection()) {
       conn.add(rdfList);
}
```

- retrieve all statements corresponding to an RDF Collection identified by the resource node from Repository and put them in a Model

```
try(RepositoryConnection conn = rep.getConnection()) {
   Model rdfList = Connections.getRDFCollection(conn, node, new LinkedHashModel());
}
```

```
try(RepositoryConnection conn = repo.getConnection()) {
    Connections.consumeRDFCollection(conn, node,
                        st -> { // ... do something with the triples forming the collection });
}
```

- Information about *Transactions* and *Multithreaded Repository Access* could be found from documentation…

RDF4J API: Querying a repository

- The *Repository API* has a number of methods for creating and evaluating queries.
- Three types of SPARQL queries are distinguished:
  - *tuple queries* The result of a tuple query is a set of tuples (or variable bindings), where each tuple represents a solution of a query. This type of query is commonly used to get specific values (URIs, blank nodes, literals) from the stored RDF data. SPARQL SELECT queries are tuple queries.
  - *graph queries* The result of graph queries is an RDF graph (or set of statements). This type of query is very useful for extracting sub-graphs from the stored RDF data, which can then be queried further, serialized to an RDF document, etc. SPARQL CONSTRUCT and DESCRIBE queries are graph queries.
  - *boolean queries* The result of boolean queries is a simple boolean value, i.e. *true* or *false*. This type of query can be used to check if a repository contains specific information. SPARQL ASK queries are boolean queries.
- SPARQL Update query

RDF4J API: Querying a repository

- *Tuple query* evaluation:

```
import java.util.List;
import org.eclipse.rdf4j.RDF4JException;
import org.eclipse.rdf4j.repository.RepositoryConnection;
import org.eclipse.rdf4jquery.TupleQuery;
import org.eclipse.rdf4j.query.TupleQueryResult;
import org.eclipse.rdf4j.query.BindingSet;
import org.eclipse.rdf4j.query.QueryLanguage;
...
try {   RepositoryConnection con = repo.getConnection();
        try {
                String queryString = " SELECT ?x ?y WHERE { ?x ?p ?y } ";
                TupleQuery tupleQuery = con.prepareTupleQuery(QueryLanguage.SPARQL, queryString);
                TupleQueryResult result = tupleQuery.evaluate();
                try {
                        List<String> bindingNames = result.getBindingNames();
                        while (result.hasNext()) {
                           BindingSet bindingSet = result.next();
                           Value firstValue = bindingSet.getValue(bindingNames.get(0));
                           Value secondValue = bindingSet.getValue(bindingNames.get(1));

                           // do something interesting with the values here...
                        }
                } finally {   result.close();   }
        } finally {   con.close();   }
} catch (RDF4JException e) {
   // handle exception
}
```

RDF4J API: Querying a repository

- To free the connection to the repository you may use a *try-with-resources* construction for handling proper opening and closing of the RepositoryConnection. It is also possible to quickly materialize the full query result (for example, convert it to a Java List) and then close the TupleQueryResult.

```
List<BindingSet> resultList;
try (TupleQueryResult result = tupleQuery.evaluate()) {
   resultList = QueryResults.asList(result);
}
```

- Tuple query evaluation in a single line of code using the *Repositories* utility (*org.eclipse.rdf4j.repository.util.Repositories*):

```
List<BindingSet> results = Repositories.tupleQuery(rep,
                "SELECT * WHERE {?s ?p ?o }", r -> QueryResults.asList(r));
```

- Directly process the query result by supplying a *TupleQueryResultHandler* to the query's *evaluate()* method.

```
String queryString = "SELECT * WHERE {?x ?p ?y }";
con.prepareTupleQuery(queryString).evaluate(new SPARQLResultsCSVWriter(System.out));
```

  - SPARQLResultsCSVWriter is a TupleQueryResultHandler implementation that writes SPARQL Results as comma-separated values.
  - RDF4J provides a number of standard implementations of TupleQueryResultHandler, and of course you can also supply your own application-specific implementation.

RDF4J API: Querying a repository

- *Graph query* evaluation:

```
import org.eclipse.rdf4j.query.GraphQueryResult;
GraphQueryResult graphResult = con.prepareGraphQuery(
                    QueryLanguage.SPARQL, "CONSTRUCT { ?s ?p ?o } WHERE {?s ?p ?o }").evaluate();
```

- A *GraphQueryResult* is similar to *TupleQueryResult*. However, the query results are *RDF statements*

```
while (graphResult.hasNext()) {
   Statement st = graphResult.next();
   // ... do something with the resulting statement here.
}
```

- It is possible to turn a *GraphQueryResult* into a *Model* (that is, a java collection of statements)

```
Model resultModel = QueryResults.asModel(graphQueryResult);
```

- Use the *Repositories* utility to obtain a result from a SPARQL CONSTRUCT/DESCRIBE query in a single line

```
Model m = Repositories.graphQuery(rep, “CONSTRUCT {…} WHERE {…}”, r -> QueryResults.asModel(r));
```

- Use a result handler for graph queries - *org.eclipse.rdf4j.rio.RDFHandler*

```
import org.eclipse.rdf4j.rio.Rio;
import org.eclipse.rdf4j.rio.RDFFormat;
import org.eclipse.rdf4j.rio.RDFWriter;
...
RepositoryConnection con = repo.getConnection();
try {
        RDFWriter writer = Rio.createWriter(RDFFormat.TURTLE, System.out);
        con.prepareGraphQuery(QueryLanguage.SPARQL,
                  "CONSTRUCT {?s ?p ?o } WHERE {?s ?p ?o } ").evaluate(writer);
}
finally {   con.close();  }
```

RDF4J API: Querying a repository

- It is possible to create *Query* objects (*TupleQuery* and *GraphQuery*) and reuse and evaluate them later. The *Query* object also has a *setBinding()* method, which can be used to specify specific values for query variables.

```
ValueFactory factory = myRepository.getValueFactory();

String keyword = “Semantic Web”;
// prepare a query that retrieves all documents for a keyword.
TupleQuery keywordQuery = con.prepareTupleQuery("SELECT ?document WHERE { ?document ex:keyword
?keyword . }");

// set the binding to a literal representation of the keyword.
keywordQuery.setBinding("keyword", factory.createLiteral(keyword));

// evaluate the prepared query and can process the result
TupleQueryResult keywordQueryResult = keywordQuery.evaluate();
```

- *SPARQL Update*:

```
import org.eclipse.rdf4j.query.Update;
...
String updateQuery = “ … ”;
Update update = con.prepareUpdate(QueryLanguage.SPARQL, updateQuery);
update.execute();
```

- Creating SPARQL Queries with the *SparqlBuilder*: RDF4J SparqlBuilder is a fluent Java API used to programmatically create SPARQL query strings. More information could be found in corresponding documentation (*http://docs.rdf4j.org/sparqlbuilder/*).
- *GeoSPARQL*: RDF4J offers an extended algebra for partial GeoSPARQL support. When enabled, this offers additional geospatial functionality as part of the SPARQL engine, on top of any RDF4J repository, using the well-known Spatial4J and JTS libraries for geospatial reasoning (*http://docs.rdf4j.org/programming/#_geosparql*).

RDF4J API: Parsing with Rio

- *RDFHandler* is the most useful Listener interface (listener that receives parsed RDF triples). It contains just five methods: *startRDF*, *handleNamespace*, *handleComment*, *handleStatement*, and *endRDF.*
- *Rio* provides a number of default implementations of *RDFHandler*. Depending on what you want to do with parsed statements, you can either reuse one of the existing *RDFHandlers*, or, if you have a specific task in mind, you can simply write your own implementation of *RDFHandler*. (see documentation)

Example: parse an RDF document and collect all the parsed statements in a Java Collection object (in a *Model* object).

```
java.net.URL documentUrl = new URL("http://example.org/example.ttl");
InputStream inputStream = documentUrl.openStream();

RDFParser rdfParser = Rio.createParser(RDFFormat.TURTLE);

Model myGraph = new LinkedHashModel();
rdfParser.setRDFHandler(new StatementCollector(myGraph));

try {
   rdfParser.parse(inputStream, documentURL.toString());
} catch (IOException e) {
  // handle IO problems (e.g. the file could not be read)
} catch (RDFParseException e) {
  // handle unrecoverable parse error
} catch (RDFHandlerException e) {// handle a problem encountered by the RDFHandler}
```

The *Rio utility* class provides additional helper methods, to make parsing to a Model a single API call:

```
Model results = Rio.parse(inputStream, documentUrl.toString(), RDFFormat.TURTLE);
```

RDF4J API: Detecting the file format

- You may not always know in advance what exact format the RDF file is in.
- The *Rio utility* class has a couple of utility methods for guessing the correct format, given either a filename or a MIME-type.
- if *Rio* can not guess the parser format based on the file name, using *.orElse()* we may simply return particular format as a default value, or to return *null* or even to throw an exception.

```
RDFFormat format = Rio.getParserFormatForFileName(documentURL.toString());
…
RDFFormat format = Rio.getParserFormatForFileName(documentURL.toString()).orElse(RDFFormat.RDFXML);
…
RDFFormat format = Rio.getParserFormatForMIMEType(contentType);

RDFParser rdfParser = Rio.createParser(format);
```

RDF4J API: Writing with Rio

- Rio also allows you to write RDF, using *RDFWriters*, which are a subclass of *RDFHandler* that is intended for writing RDF in a specific syntax format.

Example: write our statements from *Model* to a file in RDF/XML syntax.

```
Model myGraph; // a collection of several RDF statements
FileOutputStream out = new FileOutputStream("/path/to/file.rdf");
RDFWriter writer = Rio.createWriter(RDFFormat.RDFXML, out);
try {   writer.startRDF();
        for (Statement st: myGraph) {
           writer.handleStatement(st);
        }
        writer.endRDF();
} catch (RDFHandlerException e) {}
```

```
Model myGraph; // a collection of several RDF statements
FileOutputStream out = new FileOutputStream("/path/to/file.rdf")
Rio.write(myGraph, out, RDFFormat.RDFXML);
```

RDF4J API: Format converting

- Now we may convert file from one format to another. But, it may be problematic for very large files: we are collecting all statements into main memory (in a *Model* object).
- We can eliminate use of a *Model*. *RDFWriters* are also *RDFHandlers* and we can simply use the *RDFWriter* directly.

```
// open our input document
java.net.URL documentUrl = new URL(“http://example.org/example.ttl”);
InputStream inputStream = documentUrl.openStream();
// create a parser for Turtle and a writer for RDF/XML
RDFParser rdfParser = Rio.createParser(RDFFormat.TURTLE);
RDFWriter rdfWriter = Rio.createWriter(RDFFormat.RDFXML,
                            new FileOutputStream("/path/to/example-output.rdf");

// link our parser to our writer...
rdfParser.setRDFHandler(rdfWriter);

// ...and start the conversion!
try {
   rdfParser.parse(inputStream, documentURL.toString());
} catch (IOException e) {
  // handle IO problems (e.g. the file could not be read)
} catch (RDFParseException e) {
  // handle unrecoverable parse error
} catch (RDFHandlerException e) {
  // handle a problem encountered by the RDFHandler
}
```

RDF4J API: Reasoning with SPIN

- The SPARQL Inferencing Notation (SPIN) is a way to represent a wide range of business rules on top of an RDF dataset. The *SpinSail* (currently in beta) is a StackedSail component that adds a forward-chaining SPIN rule engine on top of any store. In its most basic form it can be used directly on top of a Sail.
- Since the SpinSail relies on basic RDFS inferencing to be supplied by the underlying Sail stack, for use cases where you need to rely on things like transitivity of rdfs:subClassOf relations, you should configure a Sail stack that includes the *ForwardChainingRDFSInferencer*. In addition, a *DedupingInferencer* is supplied which is a small optimization for both reasoners: it takes care to filter out potential duplicate results – though at the cost of an increase in memory usage.

```
// create a basic Sail Stack with a simple Memory Store, full RDFS reasoning,
// and SPIN inferencing support
SpinSail spinSail = new SpinSail();
spinSail.setBaseSail(
        new ForwardChainingRDFSInferencer(
                new DedupingInferencr(new MemoryStore())
        )
);
// create a repository with the Sail stack
Repository rep = new SailRepository(spinSail);
rep.initialize();
```

Create the rule according to SPIN format. To get the SpinSail to execute the rule, all you need to do is upload both initial RDF dataset and the dataset with the rule to the Repository.

Part 2

Jena API

Jena API: Capabilities

- ***RDF API*** (*http://jena.apache.org/tutorials/rdf_api.html*)
- ***OWL API*** (*http://jena.apache.org/documentation/ontology/*)
- Reading and writing
- In-memory and persistent storage
- Reasoning (*http://jena.apache.org/documentation/inference/index.html*)
- ***SPARQL*** query engine

Apache Jena distribution contains the APIs, SPARQL engine, the TDB native RDF database and a variety of command line scripts and tools for working with these systems. It is available via:

- Apache Maven (*http://jena.apache.org/download/maven.html*),
- binary distribution (*http://jena.apache.org/download/index.cgi*),

Jena API: RDF API

- ***Model*** interface is used to represent RDF graphs, to *obtain*/*create*/*remove* statements. Classes/interfaces that can be used in order to construct RDF graphs from scratch, or edit existent graphs reside in the *com.hp.hpl.jena.rdf.model* package.

- Create an empty model

```
Model model = ModelFactory.createDefaultModel();
String ns = new String("http://www.example.com/example#");
```

- Create two Resources

```
Resource john = model.createResource(ns + "John");
Resource jane = model.createResource(ns + "Jane");
```

- Create the *hasBrother* Property to associate *jane* to *john*

```
Property hasBrother = model.createProperty(ns, "hasBrother");
jane.addProperty(hasBrother, john);
```

- Create the *hasSister* Property to associate *john* to *jane* with *Statement*

```
Property hasSister = model.createProperty(ns, "hasSister");
Statement sisterStmt = model.createStatement(john, hasSister, jane);
model.add(sisterStmt);
```

- Arrays of *Statement*s can also be added to a Model

```
Statement statements[] = new Statement[5];
statements[0] = model.createStatement(john, hasSister, jane);
statements[1] = model.createStatement(jane, hasBrother, john);
model.add(statements);
```

Jena API: RDF API

- Data retrieving from the model

```
// List persons who have brother
    ResIterator persons = model.listSubjectsWithProperty(hasBrother);
// Since subjects of statements are Resources, the method returned a ResIterator
    while (persons.hasNext()) {
       Resource person = persons.nextResource();
       System.out.println(“resource URI: "+person.getURI());
    }
// List all the nicknames of John
    Property hasNickname = model.createProperty(ns, "hasNickname");
    NodeIterator nicknames = model.listObjectsOfProperty(john, hasNickname);
    System.out.println("****List of John’s nicknames***");
    while (nicknames.hasNext()) {
       System.out.println(nicknames.nextNode().toString());
    }
```

```
StmtIterator iter = model.listStatements();
// print out the predicate, subject and object of each statement
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement();  // get next statement
    Resource  subject   = stmt.getSubject();     // get the subject
    Property  predicate = stmt.getPredicate();   // get the predicate
    RDFNode   object    = stmt.getObject();      // get the object
    System.out.print(subject.toString() + " " + predicate.toString() + " ");
    if (object instanceof Resource) { System.out.print(object.toString()); }
    else { // object is a literal
       System.out.print(" \"" + object.toString() + "\""); }
    System.out.println(" .");
}
```

Jena API: RDF API

- ***RDF Models*** can be retrieved from external sources (files, databases).
- Model retrieved by a file uri:

```
String fileURI = "file:myRDF.rdf";
Model modelFromFile = ModelFactory.createDefaultModel();
modelFromFile.read(fileURI);
```

- Model retrieved by a file using RDFDataMgr: "load" operations create an in-memory container (model or dataset), "read" operations add data into an existing model or dataset.

```
// Create a model and read into it from file "data.ttl" assumed to be Turtle.
Model model = RDFDataMgr.loadModel("data.ttl") ;
// Create a dataset and read into it from file "data.trig" assumed to be TriG.
Dataset dataset = RDFDataMgr.loadDataset("data.trig") ;
// Read into an existing Model
RDFDataMgr.read(model, "data2.ttl") ;
```

- Model retrieved by a file using file manager:

```
String inputFileName  = "myRDF.rdf";
Model model = ModelFactory.createDefaultModel();
// use the FileManager to find the input file. The input file must be in the current directory
InputStream in = FileManager.get().open( inputFileName );
if (in == null) {
    throw new IllegalArgumentException("File: " + inputFileName + " not found"); }
model.read(in, null);
```

- Model written to the standard output in RDF/XML:

```
Model.write(System.out);
```

```
Model.write(System.out, "RDF/XML");
```

```
Model.write(System.out, "TURTLE");
```

Jena API: Reasoning

- ***InfModel*** interface is used to infer new data

```
String NS = "www.example.org/myEx/";
// Build an example data set
Model rdfsExModel = ModelFactory.createDefaultModel();
Property hasBrother = rdfsExModel.createProperty(NS, "hasBrother");
Property hasYoungerBrother = rdfsExModel.createProperty(NS, "hasYoungerBrother");
rdfsExModel.add(hasYoungerBrother, RDFS.subPropertyOf, hasBrother);
Resource bob_j = rdfsExModel.createResource(NS+"Bob_junior");
Resource bob = rdfsExModel.createResource(NS+"Bob").addProperty(hasYoungerBrother, bob_j);
// Create an inference model performing RDFS inference over the data
InfModel inf = ModelFactory.createRDFSModel(rdfsExModel);
// Check that resulting model shows that "Bob" also has property "hasBrother" of value
// "Bob_junior" by virtue of the subPropertyOf entailment
Resource bob_inf = inf.getResource(NS+"Bob");
System.out.println("Statement: " + bob_inf.getProperty(hasBrother));
```

Resulting output:

```
Statement: [www.example.org/myEx/Bob, www.example.org/myEx/hasBrother, www.example.org/myEx/Bob_junior ]
```

- Create inference model out of schema and data sources

```
Model schema = FileManager.get().loadModel("file:data/rdfsDemoSchema.rdf");
Model data = FileManager.get().loadModel("file:data/rdfsDemoData.rdf");
InfModel inf = ModelFactory.createRDFSModel(schema, data);
```

Jena API: Reasoning

- It is possible to use different reasoner which is not available as a convenience method as well as to configure one
- *ReasonerRegistry* has prebuilt instance of each of the main reasoners (getTransitiveReasoner, getRDFSReasoner, getRDFSSimpleReasoner, getOWLReasoner, getOWLMiniReasoner, getOWLMicroReasoner)

```
Reasoner reasoner = ReasonerRegistry.getRDFSReasoner();
InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExModel);
```

```
Reasoner reasoner = ReasonerRegistry.getOWLReasoner();
reasoner = reasoner.bindSchema(schema);
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

```
Reasoner reasoner = RDFSRuleReasonerFactory.theInstance().create(null);
InfModel inf = ModelFactory.createInfModel(reasoner, rdfsExModel);
```

- Model Validation:

```
Model data = FileManager.get().loadModel(fname);
InfModel infmodel = ModelFactory.createRDFSModel(data);
ValidityReport validity = infmodel.validate();
if (validity.isValid()) {
    System.out.println("OK");
} else {
    System.out.println("Conflicts");
    for (Iterator i = validity.getReports(); i.hasNext(); ) {
        System.out.println(" - " + i.next());
    }
}
```

Jena API: Reasoning

- *GenericRuleReasoner* with parameter based configuration.

```
// Register a namespace for use in the demo
String demoURI = "http://jena.hpl.hp.com/demo#";
PrintUtil.registerPrefix("demo", demoURI);
// Create an (RDF) specification of a hybrid reasoner which loads its data from an external file.
Model m = ModelFactory.createDefaultModel();
Resource configuration = m.createResource();
configuration.addProperty(ReasonerVocabulary.PROPruleMode, "hybrid");//"forward", "backward"
configuration.addProperty(ReasonerVocabulary.PROPruleSet, "data/demo.rules");
// Create an instance of such a reasoner
Reasoner reasoner = GenericRuleReasonerFactory.theInstance().create(configuration);
// Load test data
Model data = FileManager.get().loadModel("file:data/demoData.rdf");
InfModel infmodel = ModelFactory.createInfModel(reasoner, data);
```

- *Rule set* could be passed directly into the constructor

```
List rules = Rule.rulesFromURL("file:myfile.rules");
...
BufferedReader br = / open reader / ;
List rules = Rule.parseRules( Rule.rulesParserFromReader(br) );
...
String ruleSrc = / list of rules in line /;
List rules = Rule.parseRules(ruleSrc);
...
Reasoner reasoner = new GenericRuleReasoner(rules);
```

Jena API: Reasoning

- If the rule set is read from a *URL* or a *BufferedReader,* the rule file is preprocessed by a simple processor which strips comments and supports some additional macro commands:
  - **# ...** or ***// ...*** to comment line.
  - ***@prefix pre: <http://domain/url#>.*** Defines a prefix *pre* (local to the rule file) which can be used in the rules.
  - ***@include <urlToRuleFile>.*** Includes the rules defined in the given file in this file. The included rules will appear before the user defined rules, irrespective of where in the file the *@include* directive appears. A set of special cases is supported to allow a rule file to include the predefined rules for RDFS and OWL - in place of a real URL for a rule file use one of the keywords: *RDFS*, *OWL*, *OWLMicro*, *OWLMini* (case insensitive).

  An example complete rule file which includes the RDFS rules and defines a single extra rule is:

```
# Example rule file
@prefix pre: <http://jena.hpl.hp.com/prefix#>.
@include <RDFS>.
[rule1: (?f pre:father ?a) (?u pre:brother ?f) -> (?u pre:uncle ?a)]
```

- *It is not supported when the rules are defined as a local string variable*
  - *Use full URI instead of short qualified names with customized prefixes. Main default prefixes are supposed to be supported automatically.*
  - *Write the string with the rules into temporally created file and read them via reader.*
  - *Also, may try to separately add prefixes to the model before adding the rules that use them. (Should be checked)…*

Jena API: Reasoning

- ***Derivation*** helps to trace where an inferred statement was generated from.

  Derivation information is rather expensive to compute and store. For this reason, it is not recorded by default and *InfModel.setDerivationLogging(true)* must be used to enable derivations to be recorded. This should be called before any queries are made to the inference model.

Example:

```
eg:Employee_A   eg:hasBoss   eg:Employee_B .
eg:Employee_B   eg:hasBoss   eg:Employee_C .
eg:Employee_C   eg:hasBoss   eg:Employee_D .
```

```
// Create a trivial rule set which computes the transitive closure over eg:hasBoss
String rules = "[rule1: (?a eg:hasBoss ?b) (?b eg:hasBoss ?c) -> (?a eg:hasBoss ?c)]";
Reasoner reasoner = new GenericRuleReasoner(Rule.parseRules(rules));
reasoner.setDerivationLogging(true);
InfModel inf = ModelFactory.createInfModel(reasoner, rawData);
```

- Query whether *eg:Employee_A* is related through *eg:hasBoss* to *eg:Employee_D* and list the derivation route using the following code fragment:

```
PrintWriter out = new PrintWriter(System.out);
for (StmtIterator i = inf.listStatements(employee_A, hasBoss, employee_D); i.hasNext(); )
{   Statement s = i.nextStatement();
    System.out.println("Statement is " + s);
    for (Iterator id = inf.getDerivation(s); id.hasNext(); ) {
        Derivation deriv = (Derivation) id.next();
        deriv.printTrace(out, true);   }
} out.flush();
```

```
Statement is [ …/Employee_A, …/hasBoss, …/Employee_D]
    Rule rule1 concluded (eg:Employee_A eg:hasBoss eg:Employee_D) <-
        Fact (eg:Employee_A eg:hasBoss eg:Employee_B)
    Rule rule1 concluded (eg:Employee_B eg:hasBoss eg:Employee_D) <-
        Fact (eg:Employee_B eg:hasBoss eg:Employee_C)
        Fact (eg:Employee_C eg:hasBoss eg:Employee_D)
```

Jena API: OWL API

- ***OntModel*** interface is used to manage ontologies. Classes/interfaces that represents all aspects of the OWL language reside in the *com.hp.hpl.jena.ontology* package.

- Create an empty model

```
OntModel ontModel = ModelFactory.createOntologyModel();
String ns = new String("http://www.example.com/onto1#");
String baseURI = new String("http://www.example.com/onto1");
Ontology onto = ontModel.createOntology(baseURI);
```

- Create '*Person*', '*MalePerson*' and '*FemalePerson*' classes

```
OntClass person = ontModel.createClass(ns + "Person");
OntClass malePerson = ontModel.createClass(ns + "MalePerson");
OntClass femalePerson = ontModel.createClass(ns + "FemalePerson");
```

- Set FemalePerson and MalePerson as *subclasses* of Person

```
person.addSubClass(malePerson);
person.addSubClass(femalePerson);
```

- FemalePerson and MalePerson are *disjoint*

```
malePerson.addDisjointWith(femalePerson);
femalePerson.addDisjointWith(malePerson);
```

Jena API: OWL API

- Create datatype property '*hasAge*' that takes integer values. Basic datatypes are defined in the *com.hp.hpl.jena.vocabulary.XSD* package.

```
DatatypeProperty hasAge = ontModel.createDatatypeProperty(ns + "hasAge");
hasAge.setDomain(person);
hasAge.setRange(XSD.integer);
```

- Create *individuals* and *statements*

```
Individual john = malePerson.createIndividual(ns + "John");
Individual jane = femalePerson.createIndividual(ns + "Jane");
Individual bob = malePerson.createIndividual(ns + "Bob");
Literal age20 = ontModel.createTypedLiteral("20", XSDDatatype.XSDint);
Statement johnIs20 = ontModel.createStatement(john, hasAge, age20);
ontModel.add(johnIs20);
```

- Create object property '*hasSibling*' and annotate John and Jane as siblings

```
ObjectProperty hasSibling = ontModel.createObjectProperty(ns + "hasSibling");
hasSibling.setDomain(person);
hasSibling.setRange(person);

Statement siblings1 = ontModel.createStatement(john, hasSibling, jane);
Statement siblings2 = ontModel.createStatement(jane, hasSibling, john);
ontModel.add(siblings1);
ontModel.add(siblings2);
```

Jena API: OWL API

- Constrain *MalePerson* with the two constraints on *hasSpouse* property

```
// Create object property 'hasSpouse'
ObjectProperty hasSpouse = ontModel.createObjectProperty(ns + "hasSpouse");
hasSpouse.setDomain(person);
hasSpouse.setRange(person);
Statement spouse1 = ontModel.createStatement(bob, hasSpouse, jane);
Statement spouse2 = ontModel.createStatement(jane, hasSpouse, bob);
ontModel.add(spouse1);
ontModel.add(spouse2);

// Create an AllValuesFromRestriction on hasSpouse (hasSpouse only FemalePerson)
AllValuesFromRestriction onlyFemalePerson =
   ontModel.createAllValuesFromRestriction(null, hasSpouse, femalePerson);

// A MalePerson can have at most one spouse -> MaxCardinalityRestriction
MaxCardinalityRestriction hasSpouseMaxCard =
   ontModel.createMaxCardinalityRestriction(null, hasSpouse, 1);

// Constrain MalePerson with the two constraints defined above
malePerson.addSuperClass(onlyFemalePerson);
malePerson.addSuperClass(hasSpouseMaxCard);
```

Jena API: OWL API

- '*MarriedPerson*' class as an *intersection* of other classes

```
// Create class 'MarriedPerson'
OntClass marriedPerson = ontModel.createClass(ns + "MarriedPerson");
MinCardinalityRestriction mincr =
ontModel.createMinCardinalityRestriction(null, hasSpouse, 1);

// A MarriedPerson is a Person, AND with at least 1 spouse (min cardinality restriction)
RDFNode[] constraintsArray = { person, mincr };
RDFList constraints = ontModel.createList(constraintsArray);

// The two classes are combined into one intersection class
IntersectionClass ic = ontModel.createIntersectionClass(null, constraints);

// 'MarriedPerson' is declared as an equivalent of the intersection class defined above
marriedPerson.setEquivalentClass(ic);
```

Jena API: Ontology Model with Reasoner

- ***Inference engines*** can be 'plugged' in Models and reason with them. The reasoning subsystem of Jena is found in the *com.hp.hpl.jena.reasoner* package. All reasoners must provide implementations of the 'Reasoner' Java interface. Once a Reasoner object is obtained, it must be 'attached' to a Model.

- Objects of the *OntModelSpec* class are used to form model specifications
  - Storage scheme (e.g. in-memory)
  - Inference engine (transitive, OWL rule-based, RDFS-level rule-based, generic rule-based reasoners)
  - Language profile (RDFS, OWL-Lite, OWL-DL, OWL Full, etc.)
- Jena provides predefined *OntModelSpec* objects for basic Model types
  - e.g. The *OntModelSpec.OWL_DL_MEM_RULE_INF* object is a specification of OWL-DL models, stored in memory, which use rule-based reasoner with OWL rules. Default settings for ontology Model are: OWL-Full, in-memory, RDFS inference.
  - In case no reasoner is included to the model specification, reasoner implementations can then be attached, as in the following example:

```
// PelletReasonerFactory is found in the Pellet API
Reasoner reasoner = PelletReasonerFactory.theInstance().create();
// Obtain standard OWL-DL spec and attach the Pellet reasoner
OntModelSpec ontModelSpec = OntModelSpec.OWL_DL_MEM;
ontModelSpec.setReasoner(reasoner);
// Create ontology model with reasoner support
OntModel ontModel = ModelFactory.createOntologyModel(ontModelSpec, model);
```

Jena API: Ontology Model with Reasoner

- To enable reasoning, we need to refer to a Reasoner object.
  - *OntModels* without reasoning support will answer queries using only the asserted statements
  - *OntModels* with reasoning support will infer additional statements

```
// MarriedPerson has no asserted instances
// However, two of the three individuals in the example will be recognized as
// MarriedPersons, if an inference engine is used.

OntClass marriedPerson = ontModel.getOntClass(ns + “MarriedPerson”);
ExtendedIterator married = marriedPerson.listInstances();
while(married.hasNext()) {
          OntResource mp = (OntResource)married.next();
          System.out.println(mp.getURI());
}
```

Jena API: SPARQL query

- ***ARQ engine*** is used for the processing of SPARQL queries.

The ARQ API classes are found in *com.hp.hpl.jena.query* . Basic classes in ARQ:

- *Dataset*: The knowledge base on which queries are executed (Equivalent to RDF Models)
- *QueryFactory*: Can be used to generate *Query* (a single SPARQL query) objects from SPARQL strings
- *QueryExecution*: Provides methods for the execution of queries

- ***SELECT queries***
  - *ResultSet*: Contains the results obtained from an executed query
  - *QuerySolution*: Represents a row of query results. If there are many answers to a query, a ResultSet (with many QuerySolutions) is returned after the query is executed.
  - *ResultSetFormatter* - turn a ResultSet into various forms; into text, into an RDF graph (Model, in Jena terminology) or as plain XML.

```
// Prepare query string
String queryString = "PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>\n" +
    "PREFIX : <http://www.example.com/onto1#>\n" +
    "SELECT ?married ?spouse \n" +
    "WHERE {  ?married rdf:type :MarriedPerson.  ?married :hasSpouse ?spouse. }";
// Create a Dataset object using the ontology model. If no reasoner has been attached to
// the model, no results will be returned (MarriedPerson has no asserted instances)
Dataset dataset = DatasetFactory.create(ontModel);
// Parse query string and create Query object
Query q = QueryFactory.create(queryString);
// Execute query and obtain result set
QueryExecution qexec = QueryExecutionFactory.create(q, dataset);
ResultSet resultSet = qexec.execSelect();
while(resultSet.hasNext()) {
// Each row contains two fields: ‘married’ and ‘spouse’, as defined in the query string
   QuerySolution row = (QuerySolution)resultSet.next();
   RDFNode nextMarried = row.get("married");
   RDFNode nextSpouse = row.get("spouse");
   System.out.print(nextMarried.toString()+" is married to "+nextSpouse.toString()); }
```

Jena API: SPARQL query

- ***CONSTRACT queries*** return a single RDF graph.

```
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
Model resultModel = qexec.execConstruct() ;
qexec.close() ;
```

- ***DESCRIBE queries*** return a single RDF graph describing a resource(s).

```
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
Model resultModel = qexec.execDescribe() ;
qexec.close() ;
```

- ***ASK queries*** return a boolean value indicating whether the query pattern matched the graph/dataset or not.

```
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
boolean result = qexec.execAsk() ;
qexec.close() ;
```

Jena API: SPARQL query

- ***Datasets*** can be constructed using the *DatasetFactory*.

```
// Build Dataset from files
String dftGraphURI = "file:default-graph.ttl" ;
List namedGraphURIs = new ArrayList() ;
namedGraphURIs.add("file:named-1.ttl") ;
namedGraphURIs.add("file:named-2.ttl") ;

Query query = QueryFactory.create(queryString) ;

Dataset dataset = DatasetFactory.create(dftGraphURI, namedGraphURIs) ;
try(QueryExecution qExec = QueryExecutionFactory.create(query, dataset)) {
    ...
}
```

```
// Build Dataset from existing models
Dataset dataset = DatasetFactory.create() ;
dataset.setDefaultModel(model) ;
dataset.addNamedModel("http://example/named-1", modelX) ;
dataset.addNamedModel("http://example/named-2", modelY) ;
 try(QueryExecution qExec = QueryExecutionFactory.create(query, dataset)) {
    ...
}
```

Jena API: SPARQL Update query

- A ***SPARQL Update*** request is composed of a number of update operations, so in a single request graphs can be *created*, *loaded with RDF data* and *modified*.

Classes are found in *com.hp.hpl.jena.update* and *com.hp.hpl.jena.query*. The main important classes are:

- *GraphStoreFactory* - A graph store is the container of graphs that is being updated. It can wrap RDF Datasets.
- *UpdateRequest* - A list of Update to be performed.
- *UpdateFactory* - Create UpdateRequest objects by parsing strings or parsing the contents of a file.
- *UpdateAction* - execute updates.

```
Dataset ds = ...
GraphStore graphStore = GraphStoreFactory.create(ds) ;
// Execute a SPARQL Update request as a script from a file
UpdateAction.readExecute("update.ru", graphStore) ;
// Execute a SPARQL Update request as a string
UpdateAction.parseExecute("DROP ALL", graphStore) ;
// Read from file and execute operations
UpdateRequest request = UpdateFactory.read("update.ru") ;
UpdateAction.execute(request, graphStore) ;
// Create and execute operations
UpdateRequest request = UpdateFactory.create() ;
request.add("DROP ALL")
       .add("CREATE GRAPH <http://example/g2>")
       .add("LOAD <file:etc/update-data.ttl> INTO <http://example/g2>") ;
UpdateAction.execute(request, graphStore) ;
// Create and execute operations through programmatic update
UpdateRequest request = UpdateFactory.create() ;
request.add(new UpdateDrop(Target.ALL))
       .add(new UpdateCreate("http://example/g2"))
       .add(new UpdateLoad("file:etc/update-data.ttl", "http://example/g2")) ;
UpdateAction.execute(request, graphStore) ;
```

Part 3

Some Tools

Apache Any23

- ***Anything To Triples (Any23)*** is a library, a web service and a command line tool that extracts structured data in RDF format from a variety of Web documents.

  (*http://any23.apache.org*) (*https://search.maven.org/search?q=g:org.apache.any23*)

- Currently it supports the following input formats:
  - RDF/XML, Turtle, Notation 3
  - RDFa with RDFa1.1 prefix mechanism
  - Microformats: Adr, Geo, hCalendar, hCard, hListing, hRecipe, hReview, License, XFN and Species
  - HTML5 Microdata: (such as Schema.org)
  - JSON-LD: JSON for Linking Data. a lightweight Linked Data format based on the already successful JSON format and provides a way to help JSON data interoperate at Web-scale.
  - CSV: Comma Separated Values with separator auto detection.
  - Vocabularies: Extraction support for CSV, Dublin Core Terms, Description of a Career, Description Of A Project, Friend Of A Friend, GEO Names, ICAL, lkif-core, Open Graph Protocol, BBC Programs Ontology, RDF Review Vocabulary, schema.org, VCard, BBC Wildlife Ontology and XHTML.
  - …

Apache Any23

```
// read the model from the XHTML file (sourceURL) that contains RDFa based RDF content.
String sourceURL = "http://example.org/rdfa_test.xhtml";
Model modelFromFile = null;
try { Reader in = Utils.getRequest(sourceURL);
      modelFromFile = ModelFactory.createDefaultModel();
      modelFromFile.read(in,"","RDF/XML");
} catch (IOException e) { System.out.println(e.getMessage()); }
```

- Implementation of the *getRequest()* method mentioned above…

```
…
public static Reader getRequest(String url) throws IOException{
    Any23 runner = new Any23();
    runner.setHTTPUserAgent("test-user-agent");
    HTTPClient httpClient = runner.getHTTPClient();
    DocumentSource source = null;
    try {
        source = new HTTPDocumentSource(httpClient, url);
    } catch (URISyntaxException e) { e.printStackTrace(); }
    ByteArrayOutputStream output = new ByteArrayOutputStream();
    TripleHandler handler = new RDFXMLWriter(output);
    try {
        runner.extract(source, handler);
    } catch (ExtractionException e) { e.printStackTrace();
    } finally {
        try { handler.close();
        } catch (TripleHandlerException e) { e.printStackTrace(); }
    }
    return new StringReader(output.toString());
}
```

java-rdfa

- ***java-rdfa*** is RDFa parser written by Damian Steer. (*https://github.com/shellac/java-rdfa*)
- *java-rdfa* can be used from *Jena*. Simply invoke:

```
// it will hook the two readers in to Jena
Class.forName("net.rootdev.javardfa.jena.RDFaReader");

// then you will be able to:
model.read(url, "XHTML"); // xml parsing
model.read(other, "HTML"); // html parsing
```

Example:

```
Class.forName("net.rootdev.javardfa.jena.RDFaReader");

String sourceURL = "http://example.org/rdfa_test.xhtml";
Model model = ModelFactory.createDefaultModel();
model.read(sourceURL, "XHTML");
model.write(System.out, "TURTLE");
```

Relevant links:

- *https://mvnrepository.com/artifact/net.rootdev/java-rdfa/0.4.2*
- *http://www.java2s.com/Code/Jar/j/Downloadjavardfa042jar.htm*

Important notes:

- *There were some conflicts with latest versions of Jena. But, worked with older versions (e.g. 2.11) – different packaging structure of the library…*
- *Might not work properly with object property defined in RDFa via "property" , but works when it is done via "rel".*

Semargl

- ***Semargl*** a Highly performant, lightweight framework for linked data processing. Supports RDFa, JSON-LD, RDF/XML and plain text formats, runs on Android and GAE, provides integration with Jena, Sesame and Clerezza. (*https://github.com/semarglproject/semargl*)

  Relevant links: *http://grepcode.com/project/repo1.maven.org/maven2/org.semarglproject/semargl-jena/*

```
…
import org.semarglproject.jena.rdf.rdfa.JenaRdfaReader;
…

JenaRdfaReader.inject();
Model m = ModelFactory.createDefaultModel();
// read using FileRead
   try {
       m.read(new FileReader("C:\\data\\websites\\myfile.xhtml"), "", "RDFA");
   } catch (FileNotFoundException e) {
       e.printStackTrace();
   }
// read from InputStream
File f = new File("/Users/user1/file.xhtml");
m.read(new FileInputStream(f), "RDFA");
```

- ***NxParser*** a Java parsers for different RDF serialisations + API + tools + JAX-RS integration. It is an open source, streaming, non-validating parser for the Nx format, where x = Triples, Quads, or any other number. The NxParser Parser family also includes a RDF/XML and a Turtle parser. Moreover, we attached a JSON-LD parser (jsonld-java) and a RDFa parser (*semargl*) such that they emit Triples in the NxParser API. (*https://github.com/nxparser/nxparser*)

JavaScript tools

- **`RDFa-parser`** JavaScript (Node) based program to crawl WWW and parse RDFa annotated information. (*https://github.com/rolandg/rdfa-parser*)
- **`Green Turtle`** (Javascript librart) is an implementation of RDFa 1.1 for browsers.
  - *http://code.google.com/p/green-turtle*
  - *https://github.com/alexmilowski/green-turtle*
- **`Jsonld.js`** is a JSON-LD Processor and API implementation in JavaScript. (*https://www.npmjs.org/package/jsonld*)
- Etc.

Other tools

- **RDF2Go** is an abstraction over triple (and quad) stores. It allows developers to program against rdf2go interfaces and choose or change the implementation later easily. (*http://semanticweb.org/wiki/RDF2Go*)

- **EasyRdf** is a PHP library designed to make it easy to consume and produce RDF. (*http://www.easyrdf.org/* , *https://github.com/njh/easyrdf/*)

- More tools could be found there:
  - *https://www.w3.org/wiki/SemanticWebTools*
  - *https://www.w3.org/wiki/SparqlImplementations*

Final Assignment