## 1.2 Jena API

- Apache Jena is "a free and open source Java framework for building Semantic Web and Linked Data applications"
- Originally developed by HP Labs (until October 2009)
- It is not the only Java framework that can manage RDF data and the possibility to query it with SPARQL queries, but it comes with a lot of useful additions
- Most notably is the in-built inference support and the possibility to include advanced reasoners (Pellet)
- Official Jena tutorial can be found at `https://jena.apache.org/tutorials/index.html`

## JENA MODULES

Apache Jena consists of:

- RDF API : read and manipulate RDF graphs
- ARQ : query RDF data with SPARQL up to SPARQL 1.1 and support for federated queries
- Ontology API : supports the usage of RDFS and OWL to add semantics to the RDF data
- TDB : a native triple store
- Fuseki : a SPARQL server to expose the RDF data over HTTP
- Inference API : reasoner support with inference rules and in-built or external RDFS/OWL reasoners

## JENA DOWNLOAD

- The newest version of Jena is available
  - on their website `jena.apache.org/download/index.cgi`
  - or through Maven:

```
<dependency>
    <groupId>org.apache.jena</groupId>
    <artifactId>apache-jena-libs</artifactId>
    <type>pom</type>
    <version>X.Y.Z</version>
</dependency>
```

## PELLET/OPENLLET DOWNLOAD

- The OWL 2 reasoner Pellet is since version 3.0 commercially licensed and built into Stardog. Older versions of Pellet are not compatible to newer versions of Jena, but the open source project Openllet builds upon an old version of Pellet and makes it available for current Jena versions. Openllet is available
  - through `https://github.com/Galigator/openllet`
  - Or through Maven:

```
<dependency>
    <groupId>com.github.galigator.openllet</groupId>
    <artifactId>openllet-jena</artifactId>
    <version>2.6.4</version>
</dependency>
```

- The complete package (Jena + dependencies + Pellet (Openllet)) through our home-brewed jar should be available on the website

## JENA RDF API

- The Jena framework gives access to various objects that are linked to each other for managing and manipulating RDF data:
  - Dataset : Contains 1 default model and 0-n named models
  - Model : A collection of RDF statements. Technically a wrapper around a *Graph*, which gives access to a lot of convenience methods
  - Statement : Represents a RDF statement (S,P,O). Technically a wrapper around a *Triple*, which gives access to a lot of convenience methods
  - Resource, Literal, Property, RDFNode, ... : The parts of a Statement

## Starting from scratch

- Create an empty model

  ```
  Model model = ModelFactory.createDefaultModel();
  ```

  or create a model by reading an input file

  ```
  Model model = RDFDataMgr.loadModel( "./resource/mondial-europe.n3" );
  ```

  The RDF format is automatically chosen from the file extension or can be specified.

- Add a resource with properties

  ```
  Resource johnSmith = model.createResource("http://somewhere/JohnSmith")
      .addProperty("http://somewhere/meta#FullName", "John Smith")
      .addProperty("http://somewhere/meta#Name", model.createResource()
              .addProperty("http://somewhere/meta#Firstname", "John")
              .addProperty("http://somewhere/meta#LastName", "Smith"));
  ```

## MODEL UTILITY

- Define prefixes

```
model.setNsPrefix("meta", "http://www.semwebtech.org/mondial/10/meta#");
```

- Output the Model in different RDF formats

```
model.write(System.out, "N-TRIPLES");
model.write(System.out, "RDF/XML-ABBREV");
```

- Model operations

```
Model mUnion = model.union(model2);

Model mIntersection = model.intersection(model2);

Model mDifference = model.difference(model2);
```

## ACCESS DATA IN THE MODEL

- Iterate over all statements

```
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());
    System.out.print(" " + predicate.toString() + " ");
    if (object instanceof Resource) {
        System.out.print(object.toString());
    } else {// object is a literal
        System.out.print(" \"" + object.toString() + "\"");
    }
    System.out.println(" .");
}
```

## ACCESS DATA IN THE MODEL

- Get the properties of a specific Resource

```
Resource germany = model.getResource("http://www.semwebtech.org/"
                            + "mondial/10/countries/D/");
StmtIterator geriter = germany.listProperties();
// print out the predicate, subject and object of each statement
while (geriter.hasNext()) {
    Statement stmt      = geriter.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

    //...
}
```

## LANGUAGE TAGS

- Get the properties of a specific Resource

```
Resource r = model.createResource();
r.addProperty(RDFS.label, model.createLiteral("hello","en"));
r.addProperty(RDFS.label, model.createLiteral("hallo","de"));
//Get all resources with labels
ResIterator tagIter = model.listResourcesWithProperty(RDFS.label);
while(tagIter.hasNext()){
   Resource res = tagIter.nextResource();
   StmtIterator prop = res.listProperties(RDFS.label);
   while(prop.hasNext()){
      System.out.println(prop.nextStatement().getObject()
                  .asLiteral().getLanguage());
   }
}
```

## QUERYING WITH SELECTORS

- Query the Model/Dataset without the ARQ component
- More programming style instead of query language
  - Pattern matching through the Selector interface

    `SimpleSelector(Resource, Property, Object)`
  - Further filtering in the `selects(Statement s)` function

```
//Select all Statements with "hasCity" property that are located in France
StmtIterator qIter = model.listStatements(
    new SimpleSelector(null, model.getProperty("http://www.semwebtech.org/"
        + "mondial/10/meta#hasCity"), (RDFNode) null) {
        public boolean selects(Statement s){
            return s.getSubject().toString().endsWith("countries/F/");
        }
    }
);
```

## JENA ARQ API

The ARQ component provides the functionality to query the Model/Dataset through the use of SPARQL and iterate easily over the resulting answer bindings. Also supports:

- Named Graphs
- Prepared statements
- Federated queries

## Simple Querying

- `QueryFactory` converts a query String to a `Query`

```
String queryString =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "SELECT ?CN " +
    "WHERE {?C a mon:Country . ?C mon:name ?CN } " +
    "ORDER BY ?CN ";

Query query = QueryFactory.create(queryString);
```

- `QueryExecutionFactory` creates the query plan on a given Model/Dataset and creates a `QueryExecution`

  ```
  QueryExecution qe = QueryExecutionFactory.create(query, model);
  ```

- `QueryExecution` provides different execution functions for the different query types (Ask, Select, Construct, Describe)

  ```
  ResultSet results = qe.execSelect();
  ```

- `ResultSet` acts like an Iterator which consists of `QuerySolution`s

  ```
  while(results.hasNext()){
      QuerySolution sol = results.next();
      RDFNode n = sol.get("CN");
  }
  ```

- Don’t forget to close the `QueryExecution` to free up resources!

  ```
  qe.close();
  ```

## QUERYING NOTES

- A `ResultSet` can be only iterated once!
- The original `ResultSet` is bound to the `QueryExecution`, so it is no longer valid once the `QueryExecution` is closed!

→ Copy the `ResultSet` to iterate it multiple times and use it after the `QueryExecution` is closed

```
ResultSet persistentResult = ResultSetFactory.copyResults(results);
```

- `ResultSetFormatter` can be used to output `ResultSet`s in different formats (Table, CSV, XML, JSON, ...)

```
ResultSetFormatter.out(System.out, results, query);
ResultSetFormatter.outputAsXML(System.out, results);
```

## Try-with-resource

- The try-with-resource statement automatically closes resources that are declared in it
- The resource must implement java.lang.AutoClosable (includes all objects which implement java.io.Closable)

```
try(QueryExecution qe = QueryExecutionFactory.create(query, model)) {
    ResultSetFormatter.out(qe.execSelect());
}
```

## QUERYING NAMED GRAPHS

- Remember we need a `Dataset` for named graphs!
- SHOULD work just like with Models, but ...
- What works:
  - Not defining your own Dataset, just give the Query with FROM and FROM NAMED to the `QueryExecutionFactory`
    - External data is loaded with http or file:
    - Named Graphs can be addressed through ?variables, <exact/path/name>, p:name (with the prefix p describing the path)
  - Defining your own Dataset, setting a default model and adding named models manually
    - Setting the default model manually works
    - Adding a named model manually works (can be addressed through ?variables, <exact/path/name>, p:name (with the prefix p describing the path))

- What does not work:
  - Defining your own Dataset with an empty default model/no named models, Query with FROM and FROM NAMED
    - Does not load default model
    - Does not load named models (cannot be addresses via ?variables, <exact/path/name>, p:name (with the prefix p describing the path) )
    - Also does not work when using a simple Model instead of Dataset
  - Defining your own Dataset, setting a default model and adding named models manually
    - Setting default model manually AND adding another default model via FROM gives an empty result
    - Setting default model manually AND adding a named model via FROM NAMED gives an empty result
    - Adding a named model manually AND adding a named model via FROM NAMED gives an empty result

- Either define all models in the query string:

```
String namedQueryString =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "prefix p: <resource/> " +
    "SELECT ?CN ?CC ?PG " +
    "FROM <https://www.dbis.informatik.uni-goettingen.de/Mondial/"
       + "Mondial-RDF/mondial-europe.n3> " +
    "FROM NAMED <file:resource/mondial.n3> " +
    "WHERE { "
       + "?C a mon:Country . ?C mon:name ?CN . "
       + "GRAPH p:mondial.n3 { "
           + "?C mon:hasCity ?CC . } "
       + "GRAPH <resource/mondial.n3> { "
           + "?C mon:populationGrowth ?PG . } "
  + "} "
Query namedQuery = QueryFactory.create(namedQueryString);
QueryExecution qe2 = QueryExecutionFactory.create(namedQuery);
qe2.close();
```

- Or define all models in the program:

```
Dataset dataset = DatasetFactory.create();
dataset.setDefaultModel(model);
dataset.addNamedModel("http://foo/bla", mAll);
String namedQueryString =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "prefix p: <http://foo/> " +
    "SELECT ?CN ?CC ?PG " +
    "WHERE { "
        + "?C a mon:Country . ?C mon:name ?CN . "
        + "GRAPH p:bla { "
            + "?C mon:hasCity ?CC . } "
        + "GRAPH <http://foo/bla> { "
            + "?C mon:populationGrowth ?PG . } "
    + "} "
Query namedQuery = QueryFactory.create(namedQueryString);
QueryExecution qe2 = QueryExecutionFactory.create(namedQuery, dataset);
qe2.close();
```

## Prepared Statement

- Bind results to variables before the query plan is constructed

→ Faster execution when the query is used multiple times

→ Protect against Injection attacks

```
String queryString =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "SELECT ?CN " +
    "WHERE {?C a mon:Country . ?C mon:name ?CN } " +
    "ORDER BY ?CN ";
Query query = QueryFactory.create(queryString);
QuerySolutionMap initialBinding = new QuerySolutionMap();
initialBinding.add("C", model.getResource("http://www.semwebtech.org/"
    + "mondial/10/countries/GR/"));
QueryExecution qe3 = QueryExecutionFactory.create(query, model,
    initialBinding);
qe3.close();
```

## REMOTE QUERIES

- Query remote SPARQL endpoints over HTTP
- Similar to the SERVICE keyword

```
String queryString =
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +
    "prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> " +
    "SELECT ?CN " +
    "WHERE {?C a mon:Country . ?C mon:name ?CN } " +
    "ORDER BY ?CN ";
QueryEngineHTTP remoteQE = (QueryEngineHTTP) QueryExecutionFactory
    .sparqlService("http://www.semwebtech.org/mondial/10/sparql", query);
//remoteQE.setSelectContentType(WebContent.contentTypeResultsXML);
ResultSet remoteResults = remoteQE.execSelect();
ResultSetFormatter.out(System.out, remoteResults, query);
remoteQE.close();
```

## JENA ONTOLOGY API

- For handling ontological data Jena creates a new RDF model (named OntModel) ontop of the standard Model, which contains:
  - triples of the base RDF graph
  - 0:n imported ontologies
  - the chosen reasoner & reasoner mode
  - triples that were derived through the reasoner
- Ontology model $\leftrightarrow$ Reasoner $\leftrightarrow$ Union Graph (Base RDF graph + imported ontologies)

- This chapter covers the creation and management of classes, class-hierarchies and properties
- Actual reasoning in Jena is done via the Inference API

## Working with the OntModel

- Different `OntModelSpec`s can be specified to determine which entailments are seen by the ontology model

```
Model ontology = RDFDataMgr.loadModel("./resource/mondial-meta.n3");
OntModel ontM = ModelFactory
    .createOntologyModel(OntModelSpec.OWL_DL_MEM, ontology);
```

  Found at `http://jena.apache.org/documentation/javadoc/jena/org/apache/jena/ontology/OntModelSpec.html`

  – OWL_DL_MEM : no reasoner
  – OWL_DL_MEM_TRANS_INF : transitive class-hierarchy inference

- Load an ontology document into an ontology model

```
ontM.read("./resource/mondial-meta.n3");
```

- Retrieve an `OntResource` and list all its properties

```
OntResource res = ontM.getOntResource("http://www.semwebtech.org/"
    + "mondial/10/meta#Country");
for(StmtIterator props = res.listProperties(); props.hasNext(); ){
    Statement stmnt = props.next();
    System.out.println("property: " + stmnt.getPredicate()
        + " value: " + stmnt.getObject().toString());
}
```

- Change an `OntResource` into an `OntClass` and list all its superclasses

```
OntClass cls = res.as(OntClass.class);
//listSubClasses/SuperClasses also available with boolean argument
//for only directly inferred relationships
for(ExtendedIterator<OntClass> supers = cls.listSuperClasses();
    supers.hasNext(); ){
    OntClass superClass = supers.next();
    System.out.println(superClass + " is a superclass of " + cls.getURI());
}
```

## More complex class expressions

**Task**: Define a class for all individuals that love at least one cat and also love to run.

- New empty `OntModel` and new prefix

```
OntModel animalModel = ModelFactory
    .createOntologyModel(OntModelSpec.OWL_DL_MEM);
String catPrefix = "http://www.semwebtech.org/cats/";
```

- Define classes and properties

```
OntClass catClass = animalModel
    .createClass(catPrefix + "meta#" + "Cat");
OntClass activityClass = animalModel
    .createClass(catPrefix + "meta#" + "Activity");
ObjectProperty lovesProp = animalModel
    .createObjectProperty(catPrefix + "meta#" + "loves");
```

- Define "running" as an individual of the Activity class

```
Individual run = activityClass.createIndividual(catPrefix + "running");
```

- Define a catLover class

```
SomeValuesFromRestriction lovesACat = animalModel
    .createSomeValuesFromRestriction(null, lovesProp, catClass);
```

- Define a runner class

```
HasValueRestriction lovesRunning = animalModel
    .createHasValueRestriction(null, lovesProp, run);
```

- Combine the two restrictions

```
UnionClass runningCatLoverClass = animalModel
    .createUnionClass(catPrefix + "meta#runningCatLover",
                      animalModel.createList(new RDFNode[]
                            {lovesACat, lovesRunning})
                      );
```

- Define "running" as an individual of the Activity class

```
Individual run = activityClass.createIndividual(catPrefix + "running");
```

- Define a catLover class

```
SomeValuesFromRestriction lovesACat = animalModel
    .createSomeValuesFromRestriction(null, lovesProp, catClass);
```

- Define a runner class

```
HasValueRestriction lovesRunning = animalModel
    .createHasValueRestriction(null, lovesProp, run);
```

- Combine the two restrictions

```
UnionClass runningCatLoverClass = animalModel
    .createUnionClass(catPrefix + "meta#runningCatLover",
                      animalModel.createList(new RDFNode[]
                            {lovesACat, lovesRunning})
                      );
```

## Enumerated classes

- Defining a class by stating the individuals the class contains

```
String dogPrefix = "http://www.semwebtech.org/dogs/";
String humanPrefix = "http://www.semwebtech.org/humans/";
OntClass humanClass = animalModel
    .createClass(humanPrefix + "meta#Human");

EnumeratedClass dogLoverClass = animalModel
    .createEnumeratedClass(dogPrefix + "meta#dogLover", null);
dogLoverClass.addOneOf(humanClass.createIndividual(humanPrefix + "Basti"));
dogLoverClass.addOneOf(humanClass.createIndividual(humanPrefix + "Lars"));
```

## JENA TDB API

The Jena TDB component features a RDF quad store supporting:

- the full range of Jena APIs
- bulk loading
- transactions
- customizable query optimization
- quad filtering

## Constraints

- RDF store on a single machine
- TDB2 is the newer version, but not compatible with TDB1
- A TDB dataset can only be directly accessed from a single JVM
  - Use Fuseki to make it available for multiple applications

## Command-line tools

Most of the following functionality is also available through the use of command-line scripts provided in the Jena binary distribution located in the `bin / bat` folders.

- `tdb2.tdbloader` , `tdb2.tdbquery` , `tdb2.tdbdump` , ...
- Note: `tdbloader2` is a TDB1 script and will destroy your TDB2 database

## Creation

- Creating/loading a TDB2-backed dataset

```
Dataset dataset = TDB2Factory.createDataset(); //In-memory

Dataset dataset = TDB2Factory.connectDataset("./resource/TDB2test");
```

## Administration

- TDB2 database folder structure
  - □ Backups/
  - □ Data-0001/
  - ...
  - tdb.lock
- Compaction & backup of a live database

```
DatabaseMgr.compact(dataset.asDatasetGraph());

DatabaseMgr.backup(dataset.asDatasetGraph());
```

- Dataset storage
  - one directory is used to store one rdf dataset
  - unnamed graph of the dataset as a single graph
  - named graphs in a collection of quad indexes
  - All the named graphs can be treated as a single graph which is the union (RDF merge) of all the named graphs. This is given the special graph name <urn:x-arq:UnionGraph> in a GRAPH pattern. (default graph not included)
- Dynamic Datasets
  - Normally, ARQ interprets FROM/FROM NAMED as coming from the web; the graphs are read using HTTP GET
  - TDB modifies this behaviour; instead of the universe of graphs being the web, the universe of graphs is the TDB data store

  → FROM and FROM NAMED describe a dataset with graphs drawn only from the TDB data store

## Transactions

- Provides ACID (Atomicity, Consistency, Isolation, Durability) transaction support through the use of write-ahead-logging
  - All changes are written to journals, then propagated to the main database at a suitable moment
- One active write transaction, multiple read transaction at the same time
- Classic way of implementing transactions

```
dataset.begin(ReadWrite.READ);
try{
    Model tdbM = dataset.getDefaultModel();
    ... read operation
} finally {dataset.end(); }

dataset.begin(ReadWrite.WRITE);
try{
    ... write operation
    dataset.commit(); //dataset.abort()
} finally {dataset.end();}
```

- Transactions with the TXN interface

```
Txn.executeWrite(dataset, () -> {
        RDFDataMgr.read(dataset, "./resource/mondial-europe.n3");
    });

Txn.executeRead(dataset, () -> {
        RDFDataMgr.write(System.out, dataset, Lang.TRIG);
    });
```

- The TDB dataset can be used without these constructs, but then it is up to the user to implement the 1-write/multi-read policy
  - If the database becomes corrupted, it is not repairable
- No autocommit mode
- Models & graphs can be passed across transactions

## TDB Optimizer

The TDB optimizer utilizes both static and dynamic optimization to improve query execution

- Static optimization : transformations of the SPARQL algebra performed before query execution begins
- Dynamic optimization : deciding the best execution approach during the execution phase (may take into account data so far retrieved)

- Supported strategies: statistics based, fixed, no reordering
- TDB chooses a strategy depending on the presence of one of the following files in the database directory:
  - none.opt : No reording - execute triple patterns in the order in their query (still does filter placement)
  - fixed.opt : Built-in reordering based on the number of variables in a triple pattern
  - stats.opt : Content of the file are weighing rules for approximate matching counts of triple patterns
  - (none.opt & fixed.opt can be zero-length files)

## Logging query execution

- Messages send on "info" level
  - E.g. log4j in log4j.properties file:

    `log4j.logger.org.apache.jena.arq.exec=INFO`
- Fine tuning output with Explain.InfoLevel
  - INFO : Log each query
  - FINE : Log each query and it's algebra form after optimization
  - ALL : Log query, algebra and every database access (can be expensive)
  - NONE : No information logged
- Activating the query logger:
  - globally: `TDB.getContext().set(ARQ.symLogExec,Explain.InfoLevel.FINE);`
  - single query:

    `qExec.getContext().set(ARQ.symLogExec,Explain.InfoLevel.FINE);`
  - commandline: `tdbquery -set tdb:logExec=FINE -file queryfile`

## Filter placement

- optimization of filtered basic graph patterns
- decides the best order of triple patterns in a basic graph pattern and also the best point at which to apply the filters within the triple patterns
- any filter expression of a basic graph pattern is placed immediately after all it's variables will be bound
- conjunctions at the top level in filter expressions are broken into their constituent pieces and placed separately.

```
SELECT ?c ?city                                SELECT ?c ?city
WHERE {                                        WHERE {
        ?c a mon:Country .                             ?c a mon:Country .
                                  =>                   FILTER (?c="<.../countries/D>")
        ?c :hasCity ?city .                            ?c :hasCity ?city .
        ?city :population ?cpop .                      ?city :population ?cpop .
        FILTER (?c = "<.../countries/D>"               FILTER (?cpop > 100000)
                && ?cpop > 100000)
    }                                              }
```

## Statistics rule file

- Generating a statistics file:
  - commandline: `tdbstats -loc=DIR [-graph=URI]`
  - don't feed the output directly into the stats.opt
- File structure

```
(prefix ...
    (stats
        (meta ...)
        rule
        rule
    ))
```

- Rule structure

```
( (subj pred obj) count)
```

- Where `subj pred obj` are RDF terms or one of the following tokens:
  TERM, VAR, URI, LITERAL, BNODE, ANY

- Excerpt of stats.opt when using on mondial-europe

```
(stats
    (meta
        (timestamp "2019-05-09T16:37:00.929+02:00"
                        ^^<http://www.w3.org/2001/XMLSchema#dateTime>)
        (run@ "2019/05/09 16:37:00 MESZ")
        (count 62463))
    ((VAR <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.semwebtech.org/mondial/10/meta#River>) 185)
    ((VAR <http://www.w3.org/1999/02/22-rdf-syntax-ns#type>
            <http://www.semwebtech.org/mondial/10/meta#Country>) 54)
    ...
    (<http://www.semwebtech.org/mondial/10/meta#latitude> 1757)
    (<http://www.semwebtech.org/mondial/10/meta#isBorderOf> 180)
    (<http://www.w3.org/1999/02/22-rdf-syntax-ns#type> 12123)
    ...
    (other 0))
```