

Chapter 10

Jena: an API for Semantic Web Applications

There exist different APIs:

- SPARQL (discussed above, supported by most APIs)
- [Jena API](#) – RDF/RDFS/OWL in Java
- RDF4J API – RDF/RDFS/OWL in Java
- OWL API – Ontology Editing
- Hermit API – Reasoner
- Protégé – Graphical Tool for Ontology management
- Virtuoso – Web Server for Semantic Web Data

545

Jena: Overview

- 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) since then at Apache.
- 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
 - Inference API : reasoner support with inference rules and in-built or external RDFS/OWL reasoners
 - TDB : a native triple store
 - Fuseki : a Web server to expose the RDF data over HTTP (SPARQL; also LOD??)
- An official Jena tutorial can be found at <https://jena.apache.org/tutorials/index.html>

546

Overview: Jena Model vs. InfModel/OntModel with Reasoning

Jena Model, InfModel, OntModel: Overview

- Model: the pure (RDF) graph.
 - Nodes, (Triple) Statements, RDF-Statements + reified Statements,
 - model operations: read, remove, add, union, difference, intersect, write
 - triple patterns, method: listStatements(s,p,o) [s,p,o constants or null]
- InfModel: (RDF) graph with any kind of inference and triple patterns
- OntModel (extends InfModel)
 - “understanding” of DL/OWL-level notions like Individual, OntClass, OntProperty; operations to create and handle complex class definitions (e.g., owl:Restrictions ...)
(ontology generation, not necessarily with reasoning)
 - * Lots of methods around the ontology metadata, like e.g.
with an OntProperty p: Iterator<Restriction> i = p.listReferringRestrictions();
to obtain all owl:restrictions that deal with p.
 - * <https://jena.apache.org/documentation/ontology/index.html>
 - optional: with DL reasoning support

547

10.1 The Jena RDF API (everything around Model/Graph)

- The Jena framework gives access to various objects that are linked to each other for managing and manipulating RDF data:
 - Dataset : Contains one default model and zero or more 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
- Example code: Java/JenaModel.java

548

First, generate a simple RDF Graph “Model”

- empty graph:

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

from a file:

```
Model europe = RDFDataMgr.loadModel("/home/Mondial/mondial-europe.n3");
```

- Add data from a file or another model:

```
model.add(Model othermodel);
```

```
model.add(RDFDataMgr.loadModel(filepath+name));
```

```
RDFDataMgr.read(model, filepath+name );
```

The RDF serialization format is automatically chosen from the file extension/automatically detected by the parser (?) or can be specified explicitly in the call.

- use the direct Java commands to manipulate the model element-wise.
- compute model as union, intersection etc from other models.

549

Operations with Models

- Define prefixes

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

```
family.setNsPrefix("", "foo://bla/meta#");
```

– is e.g. used when outputting the model in Turtle format.

- Output the Model in different RDF formats

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

```
family.write(System.out, "RDF/XML-ABBREV");
```

- Model operations (as sets of edges; vertices are only removed if *all* edges of them are removed)

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

```
Model mIntersection = model.intersection(model2);
```

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

550

Creation of Resources and Properties

- Resource and Property are interfaces (Property extends Resource)
 - later: with OntModel, all kinds of owl:Class'es are also Resources
- `model.createResource(String uri) → Resource` (uri = null → blank node)
`model.getResource(String uri) → Resource` returns that resource *if it already exists*)
- `model.createProperty(String namespace, String localname) → Property`
`model.getProperty(String uri, String localname) → Resource` returns that resource *if it already exists*)
Properties like `RDF.type`, `RDFS.label` are predefined in `org.apache.jena.vocabulary.RDFS` etc.
- `model.createLiteral(String value, String langtag) → language-tagged Literal`
- `res.getLocalName(), prop.getLocalName() → String`,
`res.getNameSpace(), prop.getNameSpace() → String`,
⇒ namespaces are just strings, there is no usage of prefixes here.

551

Adding Statements to the Graph

- `resource.addProperty(Property p, RDFnode resource)`
`resource.addProperty(Property p, String stringvalue)`
`resource.addProperty(Property p, string lexical-form, RDFDatatype datatype)`
(Jena documentation: class `XSSDatatype` implements `RDFDatatype`)
`resource.addLiteral(Property p, float/double/long val)`
`resource.hasProperty(...)/hasLiteral(...) → boolean` for analogous tests
`resource.getProperty(Property p) → Statement` returns *some* such statement
`resource.getPropertyResourceValue(Property p) → Resource` gives *some* such resource
`resource.removeAll(Property p) → Resource` removes all such statements

```
Resource john = family.createResource("foo://bla/persons/john")
    .addProperty(nameP, "John")
    .addProperty(nameP, "Johannes", "de")
    .addProperty(nameP, family.createLiteral("Jean", "fr"))
    .addProperty(family.createProperty("foo://bla/meta#", "birthdate"),
        "1970-12-31", XSSDatatype.XSDdateTime)
    .addProperty(family.createProperty("foo://bla/meta#", "child"),
        family.createResource()
            .addProperty(nameP, "Alice")
            .addLiteral(family.createProperty("foo://bla/meta#", "age"), 10));
Resource alice = john.getPropertyResourceValue(family.getProperty("foo://bla/meta#", "child"));
```

552

Access Data in the Model

- Get all property statements of a specific resource

```
Resource andorra = europe.getResource("http://www.semwebtech.org/"
                                     + "mondial/10/countries/AND/");
StmtIterator andIter = andorra.listProperties();
while (andIter.hasNext()) {
    Statement stmt      = andIter.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
    // object might be Resource or Literal -> superclass RDFNode
    //...
}
```

- `res.listProperties(prop)` → StmtIterator lists all statements for a given property.
- for handling the object position, use
`rdfnode.asResource()` to “cast” it as `jena.Resource` (if it is one),
`rdfnode.asLiteral()` to “cast” it as `jena.Literal` (if it is one),
 - `literal.getValue()` → some of the Java classes String, Float, Integer etc.,
 - `literal.getFloat()`, `literal.getInt()`, `literal.getLanguage()` etc.

553

Access data in the model: all Statements

- Iterate over all statements

```
StmtIterator iter = model.listStatements();
... an iterator over all statements in the model:
while (iter.hasNext()) {
    Statement stmt      = iter.nextStatement(); // get next statement
    System.out.println(stmt.toString());
}
```

554

ListStatements with Pattern

- Query the Model/Dataset without the ARQ component
- More programming style instead of query language
- Pattern matching: `model.listStatements(Subject, Predicate, Object)`
- Preset or null s/p/o to match a triple pattern,
- if object is null, it must be casted (because `RDFNode` or `Literal (value)` is allowed).

Select all statements with 'hasCity' property that are located in France:

```
StmtIterator fcities = europe.listStatements(  
    europe.getResource("http://www.semwebtech.org/mondial/10/countries/F/"),  
    europe.getProperty("http://www.semwebtech.org/mondial/10/meta#hasCity"),  
    (RDFNode) null);
```

Select all things that have name "Monaco":

```
StmtIterator monacos = europe.listStatements(  
    null,  
    europe.getProperty("http://www.semwebtech.org/mondial/10/meta#name"),  
    "Monaco");
```

555

ListStatements with an ExtendedIterator

- `StmtIterators` (and `Nodelterator`, `Reslterator` etc) can use the interface `ExtendedIterator` that provides the following Methods:
- `filterDrop(Predicate<T> f)`,
- `filterKeep(Predicate<T> f)`

Example to do

556

ListStatements with a Selector

Define a Selector that further filters by its `selects(Statement s)` function:

Return all things that have a population of more than 1,000,000:

```
StmtIterator bigthings = europe.listStatements(  
    new SimpleSelector(null, europe.getProperty("http://www.semwebtech.org/"  
        + "mondial/10/meta#population"), (RDFNode) null) {  
        public boolean selects(Statement s){  
            return s.getObject().asLiteral().getFloat() > 1000000;  
        }  
    }  
);
```

- note: the object position for “population” is guaranteed to be a numeric literal.
- filtering it to cities is only possible when using the dirty way via testing for the URI substring “/cities/”.

557

All resources with a certain property

Predefined namespaces in Jena

- Class resources cannot have user-defined properties except annotation properties like `rdfs:label`
- example: labels with language tags

```
Resource country = europe.getResource("http://www.semwebtech.org/"  
    + "mondial/10/meta#Country");  
country.addProperty(RDFS.label, europe.createLiteral("Country", "en"));  
country.addProperty(RDFS.label, europe.createLiteral("Land", "de"));  
//Get all resources with rdfs:label  
ResIterator tagIter = model.listResourcesWithProperty(RDFS.label);  
while(tagIter.hasNext()){  
    Resource res = tagIter.nextResource();  
    StmtIterator prop = res.listProperties(RDFS.label);  
    while(prop.hasNext()){  
        Statement stmt = prop.nextStatement();  
        System.out.print(stmt.getObject().asLiteral().getValue() + " ");  
        System.out.println(stmt.getObject().asLiteral().getLanguage());  
    }  
}
```

558

yes/no checks for Properties of Resources

- `res.hasProperty(prop)` – (res, prop, X)?
`res.hasProperty(prop, rdfnode)` – (res, prop, obj)?
`res.hasProperty(prop, string)`, `res.hasProperty(prop, string, langtag)`,
`res.hasLiteral(prop, int/long/float/double)`

Operations for Modifying and Deleting Statements

- `res.removeAll(prop)`: remove all statements (res, prop, X)
- `res.removeProperties()`: remove all statements (res, P, X)
- `stmt.changeObject(rdfnode)`,
`stmt.changeObject(string)`, `stmt.changeObject(string, langtag)`,
- `stmt.changeLiteralObject(int/long/float/double)`,
- `stmt.remove()`,
- `stmt.createReifiedStatement()` creates an `rdf:Statement` object with
`rdf:subject/rdf:predicate/rdf:object` triples,
- `stmt.removeReification()`,
- ... and many more, see documentation for all these classes.

559

Datasets

- a Dataset contains a default model (maybe empty) and zero or more named graphs/models

```
Dataset ds = DatasetFactory.create();
ds.setDefaultModel(europe);
alice.addProperty(family.createProperty("foo://bla/meta#", "livesIn"),
                 andorra);
    // alice is the alice object from the 'family' model,
    // andorra is the andorra object in the 'europe' model,
    // the triple is added to 'family' (to alice).
ds.addNamedModel("http://family", family);
```

- names of named graphs in such a dataset are usually not the filenames (like in SPARQL queries with FROM NAMED); here the name URI of the named graph is set to “http://family”.
 - the additional statement is only contained in the dataset ds, if it is added to the model *before* the model is added to the dataset ds.
⇒ the model is copied, not linked.
- ... this dataset will be queried on Slide 566.

560

10.2 Jena ARQ API

ARQ component: functionality to query Models/Datasets with SPARQL

Sample Code: Java/JenaModel.java

BASICS OF QUERYING

Define a Query object

- QueryFactory parses a query String to a Query (which then contains the parsetree)

```
String qs =
    "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 q = QueryFactory.create(qs);
```

561

Create a QueryExecution object for the query and choose the input data

The QueryExecution contains the algebra expression/query plan.

- QueryExecutionFactory creates the query plan and creates a QueryExecution:
 - if an existing Model is used, the query *must not* contain any FROM clause:
`QueryExecution qe = QueryExecutionFactory.create(q, europe);`
 - if a Dataset (containing a default model (maybe empty) and zero or more named graphs/models) is used, the query *must not* contain any FROM clause:
`QueryExecution qe = QueryExecutionFactory.create(q, ds);`
(this will create a temporary model/dataset during evaluation which is removed afterwards)
 - if the query contains a FROM clause, a model/dataset *can not be given*:
`QueryExecution qe = QueryExecutionFactory.create(q);`
 - Note: it is *not possible* to have a model/data set, *and* any FROM clause in the query. Then, the FROM clauses are *ignored*.

562

Execute the query

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

```
ResultSet res = qe.execSelect();
Model constrModel = qe.execConstruct(); // set of triples
```

- `ResultSet` acts like an iterator which consists of `QuerySolutionS`

```
while (results.hasNext()) {
    QuerySolution sol = res.next();
    RDFNode n = sol.get("CN"); // access by variable names
}
```

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

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

563

Handling of ResultSets

- A `ResultSet` can be only iterated (incl. printed) once (this runs the nested loops etc. for the actual evaluation)!
- The original `ResultSet` is bound to the `QueryExecution`, so it is no longer valid once the `QueryExecution` is closed!

⇒ Materialize the `ResultSet` into a `ResultSetRewindable` to iterate it multiple times and to use it after the `QueryExecution` is closed

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

```
ResultSetRewindable persistentRes = ResultSetFactory.copyResults(res);
qe.close();
int numberOfResults = ResultSetFormatter.toList(persistentRes).size();
System.out.println("Number of Results: " + numberOfResults);
persistentRes.reset(); // to use it again
while (persistentRes.hasNext()) {
    QuerySolution sol = persistentRes.next();
    RDFNode n = sol.get("CN"); // and do something with n
    System.out.println(n); }
}
```

564

Try-with-resource

- The “try-with-resource” (since Java 7) statement automatically closes resources that are declared in it (which has been done before with “finally {}”)
- The resource must implement `java.lang.AutoClosable` (includes all objects which implement `java.io.Closable`)

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

565

Example: query with a Dataset that combines two models

- consider the dataset from Slide 560.

```
String qs1 =  
    "prefix mon: <http://www.semwebtech.org/mondial/10/meta#> " +  
    "prefix : <foo://bla/meta#>\n " +  
    "SELECT ?C ?CN ?PN ?A " +  
    "WHERE { "  
    + "?C a mon:Country . ?C mon:name ?CN . "  
    + "    OPTIONAL { ?P :livesIn ?CC }"  
    + "GRAPH <http://family> { ?P :name ?PN . "  
    + "    OPTIONAL { ?P :livesIn ?C; :age ?A }}}";
```

```
Query q1 = QueryFactory.create(qs1);  
QueryExecution qe1 = QueryExecutionFactory.create(q1, ds);  
ResultSet res1 = qe1.execSelect();  
// do something with res1, e.g.  
ResultSetFormatter.out(System.out, res1, q1);  
qe1.close();
```

566

Example: query with FROM

- like in standalone-SPARQL, then use the result
- Example: extend the europe model with all islands (and their data) in seas that are mentioned in mondial-europe.n3 (those that do not belong to european countries):

```
String qs2 = " prefix : <http://www.semwebtech.org/mondial/10/meta#> " +
    " CONSTRUCT { ?I ?P ?O . ?O a ?C ; :name ?ON } " +
    " FROM <file:" + filepath + "mondial.n3> " +
    " FROM NAMED <file:" + filepath + "mondial-europe.n3> " +
    " WHERE { GRAPH <file:" + filepath + "mondial-europe.n3> " +
    "         { ?S a :Sea; :locatedIn ?CO. ?CO a :Country . " +
    "           FILTER NOT EXISTS { ?CO :name 'Russia' } } " +
    "         ?I :locatedInWater ?S . " +
    "         ?I ?P ?O . FILTER (!isBlank(?O)) " +
    "         OPTIONAL { ?O a ?C ; :name ?ON } " +
    "         FILTER NOT EXISTS " +
    "           { GRAPH <file:" + filepath + "mondial-europe.n3> { ?I a :Island }}} ";
Query q2 = QueryFactory.create(qs2);
QueryExecution qe2 = QueryExecutionFactory.create(q2);
Model islands = qe2.execConstruct();
islands.write(System.out, "Turtle");
Model europeAndIslands = europe.union(islands); qe2.close();
```

567

Remote queries

- Query remote SPARQL endpoints over HTTP
- similar functionality as the SERVICE keyword:

```
String qs = // country names as before ...
Query q = QueryFactory.create(qs); // as before
QueryEngineHTTP remoteQE = (QueryEngineHTTP) QueryExecutionFactory
    .sparqlService("http://www.semwebtech.org/mondial/10/sparql", q);
//remoteQE.setSelectContentType(WebContent.contentTypeResultsXML);
ResultSet res4 = remoteQE.execSelect();
ResultSetFormatter.out(System.out, res4, q);
remoteQE.close();
```

568

Prepared Statements

- ... similar to PreparedStatements in JDBC, but ... different:
- create the Query parsetree from the String,
- add a QuerySolutionMap as initial bindings
- then create the query plan, and execute it:

```
String qs = // country names as before ...
    "SELECT ?CN " +
    "WHERE { ?C a mon:Country . ?C mon:name ?CN } ";
Query q = QueryFactory.create(qs);
QuerySolutionMap initialBinding = new QuerySolutionMap();
initialBinding.add("C", mondial.getResource("http://www.semwebtech.org/"
    + "mondial/10/countries/USA/"));
QueryExecution qe3 = QueryExecutionFactory.create(q, europeAndIslands,
    initialBinding);
ResultSet res3 = qe3.execSelect(); // ... and process it ...
qe3.close();
```

569

Prepared Statements in Jena: Discussion

- Allow user input (values) and protect from injection attacks:
Do not copy the user input into the string and compile it:
SELECT * FROM country where code= '\$input'
user inputs "D'; drop table city;" and the query becomes
SELECT * FROM country where code= 'D'; drop table city;
⇒ user input is only handled as values.
- in JDBC, the query plan is *prepared/precompiled*, and the values are just submitted to the query plan. For multiple executions, the query plan is only generated once.
- in Jena, the query parsetree is pre-generated and can be reused, but the query plan is newly generated for each execution
 - less efficient, of the query plan is always the same
 - might be more efficient, if different bindings result in different optimal query plans
- the setting is different from SQL: in SQL, the "?" *must be set*, in SPARQL, all answers would be returned.
 - with a partial tuple of initial bindings, the sets of answer bindings can be constrained (like a single VALUES tuple).

570

10.3 Jena OntModel and InfModel

Note: this chapter does not deal with details of the reasoning (see pp. 603), but focuses on the Jena API for handling models.

Sample Code: Java/JenaOntModel.java

571

Architecture: Model, Ontology, Inference

- Ontology model ↔ Reasoner ↔ Union Graph (Base RDF graph + imported ontologies)
GRAPHICS TODO
- triples of the base RDF graph (ABox)
- 0 . . . n imported ontologies (TBox knowledge, usually OWL-DL)
recall: many DL terms are represented by multiple triples – here, not the triples are part of the graph, but the whole DL concept is part of the knowledge
- the chosen reasoner & reasoner mode
- triples that were derived through the reasoner are ‘known’ to the Ontology access and to the reasoner
 - they can be materialized and stored in the RDF graph or in a separate RDF graph (bottom-up style, often followed by rule-based reasoners)
 - they might be “virtual”. Queries are then answered by a (tableau) reasoner. (Partial) materialization of such virtual conclusions (e.g. tabling) is possible.

572

Creation of an OntModel – Ontology Handling, optionally with Reasoning

- use a Jena OntModelSpec to distinguish the capabilities of the ontology (OWL, OWL-DL, OWL-Lite, RDFS) and the capabilities of the reasoner that are required:
<http://jena.apache.org/documentation/javadoc/jena/org/apache/jena/ontology/OntModelSpec.html>
 - OWL_DL_MEM: OWL DL model stored in memory, *no reasoning*
 - OWL_DL_MEM_TRANS_INF: ... use the transitive inferencer for `rdfs:subClassOf/subPropertyOf`
 - OWL_DL_MEM_RDFS_INF: ... same, use the RDFS inferencer
 - OWL_DL_MEM_RULE_INF: ... same, using the Jena OWL Rule Inferencer
 - OWL_MEM_MINI_RULE_INF: ... some OWL
 - OWL_MEM_MICRO_RULE_INF: ... little bit of OWL
 - and some more ...
 - “alien” reasoners like Pellet/Openllet can also provide an own spec.

573

Creation of a Jena OntModel with a Jena Spec

- create an empty OntModel:
`OntModel m = ModelFactory.createOntologyModel(OntModelSpec spec)`
- for given: Model base = ...:
`OntModel m = ModelFactory.createOntologyModel(OntModelSpec spec, Model base)`
- initialize with an ontology from <uri>:
`OntDocumentManager dm = OntDocumentManager.getInstance() OntModel m = dm.getOntology(String uri, OntModelSpec spec)`
- read data from another file into an existing OntModel:
`ontmodel.read(filepath+name);`
- `ontmodel.getReasoner()` → gives the reasoner.

Creation of a Jena OntModel with Pellet

- `Model model = ... initialization ... ;`
`OntModel pelletmodel =`
`ModelFactory.createOntologyModel(PelletReasonerFactory.THE_SPEC);`
`pelletmodel.add(model);`
`// pelletmodel.prepare(); -- not needed`
`// Reasoner pellet = pelletmodel.getReasoner(); -- if needed`

574

Usage of Jena's OntModel

- all methods from InfModel (see Slide 578)
- Classes/Interfaces (below called XXX) for each type of DL expressions:
Individual, OntClass, Restriction (SomeValuesFromRestriction, ..., ComplementClass, IntersectionClass MaxCardinalityQRestriction, HasValueRestriction, ...), Property (DatatypeProperty, ObjectProperty, TransitiveProperty, ...)
- createXXX(*parameters*): constructors for things and expressions ... e.g.
createIndividual(*string uri*), createClass(*string uri*),
createMaxCardinalityQRestriction(*string uri, prop, int cardinality, OntClass cls*)
createHasValueRestriction(*string uri, prop, RDFNode value*)
createComplementClass(*string uri, Resource cls*)
- Ontology-level methods to access the model:
 - m.getXXX(*string uri*): accessor for things and expressions (to result in an instance of the correct Java class)
 - m.listXXX(...): return an iterator over all things of a certain type,
listIndividuals(), listIndividuals(Resource cls), listNamedClasses(),
listFunctionalProperties(), listRestrictions(), ...

575

Usage of Jena's OntModel (cont'd)

- write(...): export the *raw model* (base graph + explicit OWL stuff) *without inferred things* to a file or stream.
- writeAll(...): export all triples (base + inferred) to a file or stream

576

INFMODEL

- Pure InfModel: RDFS with a reasoner – methods for controlling the reasoning capabilities.
- OntModel extends InfModel with the previously described DL stuff.

Creation of an InfModel – Graph + Reasoner

- An RDFS Model is not an OntModel (it understands nothing than RDF/RDFS):
`ModelFactory.createRDFSModel(Model model)`
- With RDF data and RDFS metadata from different Model instances:
`ModelFactory.createRDFSModel(Model schema, Model model)`
- One can also use an OWL Reasoner with just an InfModel:
`ModelFactory.createInfModel(Reasoner reasoner, Model model)`
`Reasoner reasoner = ReasonerRegistry.getOWLReasoner();`
`InfModel infmodel = ModelFactory.createInfModel(reasoner, model);`
- `ModelFactory.createInfModel(Reasoner reasoner, Model schema, Model model)`

577

Usage of Jena's InfModel (and OntModel)

- `validate()` check consistency of the underlying data; returns a `ValidityReport`.
 - `boolean isValid()` Note: "inconsistent classes" i.e., classes that can be derived to be empty are only considered to be non-valid if also an instance is asserted.
 - `boolean isClean()` if there are inconsistencies *or* other warnings (e.g., inconsistent classes).
 - `Iterator<...> getReports()` for details
- `void prepare()` Perform any initial processing and caching.
(usually done automatically when needed)
- `void rebind()` reconsult the underlying data to take into account changes that did not happen via the InfModel/OntModel interface to the underlying graph.
- `void reset()` Reset any internal caches (e.g. Tabling)

578

ONTMODEL: THE JENA ONTOLOGY API

- Ontology model: strict separation between individuals, classes, and properties:
 - Individuals have user-defined properties
 - classes and properties have only properties defined in RDFS/OWL, and they might have *owl:AnnotationProperties* (e.g. *rdfs:label*)
 - these properties might be queried and modified (which changes the reasoner's knowledge base)
- Individuals and classes are anonymous (blank nodes) or have a URI;
 - `createClass()`, `createClass(uri-string)`,
[more on classes corresponding to DL concepts see subsequent slides]
 - `createIndividual(Resource class)`, `createIndividual(String uri, Resource class)`,
(where *class* might be *owl:Thing*),

579

Properties in the OntModel

- Properties must have a URI, and they must be created according to their specific characteristics:
 - `createDatatypeProperty(uri)`, `createDatatypeProperty(uri, boolean isfunctional)`,
`createObjectProperty(uri)`, `createObjectProperty(uri, boolean isfunctional)`,
More specific subclasses of `ObjectProperties`:
`createTransitiveProperty(uri)`, `createSymmetricProperty(uri)`,
`createInverseFunctionalProperty(uri)`,
`createInverseFunctionalProperty(uri, boolean isfunctional)`

580

DL Concept definitions in the OntModel

Each DL expression type has a constructor:

- TODO

581

- Add/remove class membership of some Individual:

```
indiv.addOntClass(Resource cls); res.removeOntClass(Resource cls);
```

- Retrieve an OntResource and list all its properties

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

582

Accessing Resources in an OntModel

- Access an `OntResource` as 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()); }
```

- list all instances of an `OntClass`:

```
ExtendedIterator<? extends OntResource> listInstances()
```

and many other methods ...

- Analogous: access an `OntResource` as an `OntProperty` and investigate/manipulate its properties.

583

AllDifferent in Jena

- (like many things in Jena) `AllDifferent` is an interface ... so it does not have a constructor, but things (here: an existing class, or an ad hoc set of instances) have to be “turned into” an `AllDifferent`:
- many such questions can be answered e.g. by google “jena usage alldifferent”
- Pseudo-“Cast” a class as `AllDifferent`, *and then add all its instances as different members*

```
Resource c = mondial.getResource("http://www.semwebtech.org/"  
    + "mondial/10/meta#Country"));  
OntClass oc = c.as(OntClass.class);  
AllDifferent ac = c.asAllDifferent();  
// add the instances to the AllDifferent specification list:  
ac.addDistinctMembers(oc.listInstances());
```

584

EXAMPLE: CREATE 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");
```

585

- 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})
    );
```

586

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"));
```

587

Use auxiliary Ad-hoc-“query”-classes with InfModel (and OntModel)

- extension of Model.listStatements(s,p,o):

```
StmtIterator listStatements(Resource subject, Property predicate,
                            RDFNode object, Model posit)
```

- Preset or null s/p/o to match a triple pattern,
- if object is null, it must be casted:
(RDFNode) null
- “posit” is an additional separate model, where e.g. complex classes might be defined that are only used in a match pattern
 - * typical case: match (null, rdf:type, class) where class is a complex ad hoc class definition.
 - * [Java Code next slide.](#)
 - note: example uses simple OWL_DL_MEM spec without reasoning; no reasoning needed here,
 - Creates a class HasValue(neighbor,germany) in the posit model and asks for its members.

588

```

import org.apache.jena.rdf.model.Statement;
import org.apache.jena.rdf.model.StmtIterator;
import org.apache.jena.util.FileManager;
import org.apache.jena.vocabulary.RDF;

public class JenaListStmtsPosit {
    static String filepath = "/home/may/teaching/SemWeb/RDF/";
    public static void main(String[] args){
        String monPrefix = "http://www.semwebtech.org/mondial/10/";
        Model m = FileManager.get().loadModel(filepath + "mondial-europe.n3");
        OntModel ontM = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM, m);

        Model m2 = ModelFactory.createDefaultModel();
        OntModel posit = ModelFactory.createOntologyModel(OntModelSpec.OWL_DL_MEM, m2);

        // create Class: HasValue(neighbor,germany) in posit and output its members
        OntClass Country = posit.createClass(monPrefix + "meta#Country");
        Individual germany = Country.createIndividual(monPrefix + "countries/D/");
        ObjectProperty neighborProp = posit.createObjectProperty(monPrefix + "meta#neighbor");
        HasValueRestriction GermanNeighbor =
            posit.createHasValueRestriction(null, neighborProp, germany);

        for (StmtIterator i =
            ontM.listStatements(null, RDF.type, GermanNeighbor, posit);
            i.hasNext(); ) {
            Statement s = i.nextStatement();
            Resource r = s.getSubject();
            System.out.println(r);
        }
    }
}

```

[Filename: Java/JenaListStmtsPosit.java]

589

Aside: How does the Reasoner “find” and distinguish DL Specifications from input

- Recall Slide 391: owl:restriction Specifications that are not given using blank nodes are ignored
 - ⇒ when reading an input file, all triples forming such an expression must be “encapsulated” by a blank node (which then actually exists; cf. Slide 410)
- what happens if such a blank node is constructed with the Model API:

TODO Java

590

GENERAL NOTES FOR USING JENA AND PELLET/OPENLLET IN THIS COURSE

Three possibilities:

- the semweb.jar is an “uber-jar” which contains *all* classes from jena+pellet (and some more). Put it in the class path or the lib directory/build path – this is sufficient.
 - shell: call

```
javac -cp .:path-to-semweb.jar bla.java
java -cp .:path-to-semweb.jar bla.java
```

(don’t forget to have “.” on the path for your own class)
 - note: such an “uber-jar” does not contain “nested” jars, but all jars (basically, these are zip-archives) are unpacked and re-packed with a manifest (pointing to the main class if existing)
- download Jena as jar files and put all in your classpath
- download and install Jena as maven project.
 - maven allows to specify dependencies wrt. exact versions of packages
 - automatically transitively downloads everything

591

10.4 Aside: Jena TDB API

- Basic Jena: a simple main-memory graph structure
- The Jena TDB component features a RDF quad store supporting the following:
 - quads for triples in optionally named graphs:
([Subj](#), [Pred](#), [Obj](#), [GraphID](#)).
 - unchanged usage of the full range of Jena APIs
 - bulk loading
 - transactions
 - customizable query optimization
 - quad filtering
- **Note:** TDB2 is the newer version, but not compatible with TDB1 (which is obsolete)

592

TDB-based Datasets

- A `Dataset` instance contains the RDF data graphs (cf. Slide 560)
- 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/ (with a lot of files in it that form the RDF store)
 - ...
 - tdb.lock
- the newest `Data-000 n` is the *current* database where TDB works with
- compaction & backup of the database in Java:
 - `DatabaseMgr.compact(dataset.asDatasetGraph());`
creates `Data-000 $n+1$` by reorganizing the current `Data-000 n` , further accesses will then use the new one.
 - `DatabaseMgr.backup(dataset.asDatasetGraph());`

593

Constraints

- RDF store only (in a single directory) on a single machine, not distributed
- A TDB directory can only be accessed at the same time from a single JVM (but from multiple threads/processes/commandline calls; for transactionality see Slide 596)
 - Use the Fuseki server to make it available for multiple applications (Web server)

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`, ...
- all with `-loc unixpath` argument to address the TDB2 directory on a computer
- (Note: `tdbloader2` is a TDB1 script and will destroy your TDB2 database)

594

Dataset in TDB

- 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)
- Working with TDB 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

595

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
- Constraint: One active write transaction, multiple read transaction at the same time (globally to the whole TDB, not on the data-item level like in SQL database)
- Classical way of implementing transactions by BOT/EOT:

```
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();}
```

596

TXN interface for Transactions

- Transactions using the TXN interface:

```
Txn.executeWrite(dataset, () -> {  
    RDFDataMgr.read(dataset, "./resource/mondial-europe.n3");  
    ... further commands that read and write the TDB contents ...  
});
```

```
Txn.executeRead(dataset, () -> {  
    RDFDataMgr.write(System.out, dataset, Lang.TRIG);  
    ... further commands that read the TDB contents ...  
});
```

Note: variables that are communicated into the Lambda-function have to be final or effectively final.

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

597

TDB QUERY OPTIMIZATION

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 which of the following files is present in the database directory:
 - none.opt : No reordering - execute triple patterns in the same order as in the 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)

598

Logging query execution – similar to SQL's query plan

- 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 -loc=unixpath -set tdb:logExec=FINE -file queryfile`

599

Internal Optimization: Filter placement (visible with the logger's query plan functionality)

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

=>

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

600

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

601

- 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))
```

602