Lecture 5: Advanced Concepts (Sesame and Jena APIs)

TIES452 Practical Introduction to Semantic Technologies
Autumn 2014

University of Jyväskylä

Khriyenko Oleksiy

Part 1

Sesame API

Sesame API: RDF Model API

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

Sesame API: RDF Model API

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

```
ValueFactory factory = ValueFactoryImpl.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.openrdf.model.vocabulary* package

Example:

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

Sesame API: RDF Model API

- In order to deal with *collections* of RDF statements, we can use the *org.openrdf.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:

```
// 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
  Literal name = model.filter(person, FOAF.NAME, null).objectLiteral();
  ...
}
```

Sesame 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.openrdf.repository* *(several implementations for these interface exist in various sub-packages)*
- The main three implementations of *Repository* interface:
  - *org.openrdf.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 Sesame repository
  - *org.openrdf.repository.http.HTTPRepository* is a *Repository* implementation that acts as a proxy to a Sesame repository available on a remote Sesame server, accessible through HTTP.
  - *org.openrdf.repository.sparql.SPARQLRepository* is a *Repository* implementation that acts as a proxy to any remote SPARQL endpoint (whether that endpoint is implemented using Sesame or not).

Sesame 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.openrdf.repository.Repository;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.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.openrdf.repository.Repository;
import org.openrdf.repository.sail.SailRepository;
import org.openrdf.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();
```

Sesame API: Repository API

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

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

- Accessing a *server-side repository*

```
import org.openrdf.repository.Repository;
import org.openrdf.repository.http.HTTPRepository;
...
String sesameServer = "http://example.org/openrdf-sesame/";
String repositoryID = "example-db";
Repository repo = new HTTPRepository(sesameServer, repositoryID);
repo.initialize();
```

Sesame API: Repository Manager

- Using the *RepositoryManager* for handling repository creation and administration offers a number of advantages:
  - a single *RepositoryManager* object can be more easily shared throughout your application than a host of static references to individual repositories;
  - you can more easily create and manage repositories ‘on-the-fly’, for example if your application requires creation of new repositories on user input;
  - the *RepositoryManager* stores your configuration, including all repository data, in one central spot on the file system.
- The *RepositoryManager* comes in two flavours:
  - *org.openrdf.repository.manager.LocalRepositoryManager* manages repository handling for you locally, and is always created using a (local) directory.
  - *org.openrdf.repository.manager.RemoteRepositoryManager* is used to create and manage Sesame repositories residing on a remotely running Sesame server.

Sesame 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.openrdf.repository.RepositoryConnection*) requested from the repository. It allows perform various operations, such as *query evaluation*, *getting*, *adding*, or *removing* statements, etc.

```
import org.openrdf.OpenRDFException;
import org.openrdf.repository.Repository;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.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 (OpenRDFException e) {}
catch (java.io.IOException e) {}
```

Sesame 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

Sesame API: Querying a repository

- *Tuple query* evaluation:

```
import java.util.List;
import org.openrdf.OpenRDFException;
import org.openrdf.repository.RepositoryConnection;
import org.openrdf.query.TupleQuery;
import org.openrdf.query.TupleQueryResult;
import org.openrdf.query.BindingSet;
import org.openrdf.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...
                        }
                } finaly {   result.close();   }
        } finally {   con.close();   }
} catch (OpenRDFException e) {
   // handle exception
}
```

Sesame API: Querying a repository

- *Graph query* evaluation:

```
import org.openrdf.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 a result handler for graph queries - *org.openrdf.rio.RDFHandler*

```
import org.openrdf.rio.Rio;
import org.openrdf.rio.RDFFormat;
import org.openrdf.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();  }
```

Sesame API: Querying a repository

- *SPARQL Update*:

```
import org.openrdf.query.Update;
...
String updateQuery = "…";
Update update = con.prepareUpdate(QueryLanguage.SPARQL, updateQuery);
update.execute();
```

Sesame API: Statement manipulating

- Creating individual statements

```
import org.openrdf.model.vocabulary.RDF;
import org.openrdf.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 (OpenRDFException 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));
```

Sesame 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*.

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);

org.openrdf.model.Model myGraph = new org.openrdf.model.impl.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
}
```

Sesame 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) {}
```

Sesame 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
}
```

Sesame API: Detecting the file format

- You may not always know in advance what exact format the RDF file is in.
- *RDFFormat* has a couple of utility methods for guessing the correct format, given either a filename or a MIME-type.

```
RDFFormat format = Rio.getParserFormatForFileName(documentURL.toString());
RDFFormat format = Rio.getParserFormatForMIMEType(contentType);
RDFParser rdfParser = Rio.createParser(format);
```

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

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);
// Because subjects of statements are Resources, the method returned a ResIterator
    while (persons.hasNext()) {
// ResIterator has a typed nextResource() method
       Resource person = persons.nextResource();
// Print the URI of the resource
       System.out.println("The list of persons (URIs) who have property"+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());
    }
```

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 file manager:

```
String inputFileName  = "myRDF.rdf";
Model model = ModelFactory.createDefaultModel();
// use the FileManager to find the input file. Note that 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");
}
// read the RDF/XML file
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, "N-TRIPLES");
```

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

Jena API: RDF API

- The package *com.hp.hpl.jena.db* is used to provide persistent storage of Jena Models
- Accessing a Model in a MySQL DB

```
try {
// Load MySQL driver
           Class.forName("com.mysql.jdbc.Driver"); }
catch(ClassNotFoundException e) { ... }

// Create a database connection
IDBConnection conn =
           new DBConnection("jdbc:mysql://localhost/jenadb", “user”, “pass”, "MySQL");
ModelMaker maker = ModelFactory.createModelRDBMaker(conn);

// Retrieve Model
Model dbModel = maker.openModel(“http://www.example.com/example", true);

// View all the statements in the model as triples
StmtIterator iter = dbModel.listStatements();
while(iter.hasNext()) {
   Statement stmt = (Statement)iter.next();
   System.out.println(stmt.asTriple().toString());
}
```

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

- `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.serDerivationLogging(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 form 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

Semargl

- ***Semargl*** is a modular framework for crawling linked data from structured documents. Due to small memory footprint, and CPU requirements allow framework to be embedded in any system. (http://semarglproject.org)
- Semargl offers lightweight and high-performant:
  - streaming parsers for *RDF/XML*, *RDFa*, *N-Triples*, *JSON-LD*
  - streaming serializers for *Turtle*, *NTriples*, *Nquads*
  - integration with *Jena*, *Clerezza* and *Sesame*.

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)
- 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.

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");
```

JavaScript tools

- ***Green Turtle*** is an implementation of RDFa 1.1 for browsers. (http://code.google.com/p/green-turtle/)
- ***Jsonld.js*** is a JSON-LD Processor and API implementation in JavaScript. (https://www.npmjs.org/package/jsonld)
- Etc.

Final Assignment

- Instruction will be available: **Monday, 24.11.2014**
- Deadline: **Sunday, 21.12.2014**