Knowledge management: The Jena API

Introduction

- Jena is a Java API for semantic web applications
  - Manipulate RDF triples.
  - Read and create RDF/XML documents.
  - SPARQL interpreter.
  - Ontology management: RDF-Schema, DAML+OIL, OWL.
  - ...
- Open source software (Apache license) mainly developed by HP. http://jena.apache.org

Compilation / Execution

- Jena can be downloaded from http://www.apache.org/dist/jena/binaries/
- The jar files required to compile and execute a java program using Jena are available in the lib folder. They have to be specified in the CLASSPATH.
- To use Jena, j2sdk 1.6 is recommended.
- The javadoc is available at http://jena.apache.org/documentation/javadoc/jena/.

MANIPULATE RDF TRIPLES

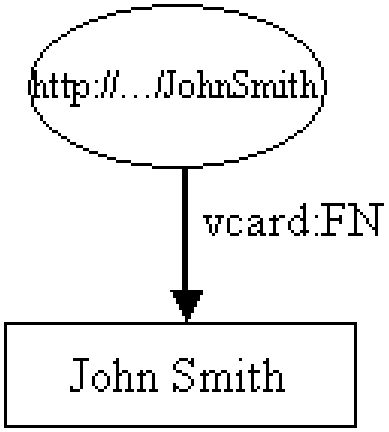
VCARD

- VCARD (ftp://ftp.isi.edu/in-notes/rfc2426.txt) defines a model to represent “visit cards’’: first name, last name, phone number, e-mail, birth date, URL, etc.

- Our examples in these slides are about people and use an RDF representation of vCard objects.
  - http://www.w3.org/TR/vcard-rdf

RDF triples

- Jena provides classes to represent RDF graphs (Model), resources (Resource), properties (Property), literals (Literal).




```
static String personURI    = "http://somewhere/JohnSmith";
static String fullName     = "John Smith";

// create an empty Model
Model model = ModelFactory.createDefaultModel();

// create the resource
Resource johnSmith = model.createResource(personURI);

// add the property
 johnSmith.addProperty(VCARD.FN, fullName);
```

RDF triples – Model and Resource

- Creating a Graph

  ModelFactory is a Model (i.e. RDF graph) Factory.

  – createDefaultModel creates a standard RDF graph in memory.

  – createFileModelMaker creates a graph on the disk.

  – createOntologyModel creates an ontology (RDF-Schema, etc.)

  – . . .

- Adding a Resource

  – createResource adds a Resource to the model. The created resource is returned.

RDF triples - Properties

- Adding a property to a Resource

  addProperty adds a property to a Resource.
  - addProperty(Property p, boolean o)
  - addProperty(Property p, char o)
  - addProperty(Property p, double o)
  - addProperty(Property p, float o)
  - addProperty(Property p, long o)
  - addProperty(Property p, java.lang.Object o)
  - addProperty(Property p, RDFNode o)
  - addProperty(Property p, java.lang.String o)
  - addProperty(Property p, java.lang.String o, java.lang.String l)

Resource and Literal are subclasses of RDFNode.

RDF triples - Properties

- addProperty returns the subject resource

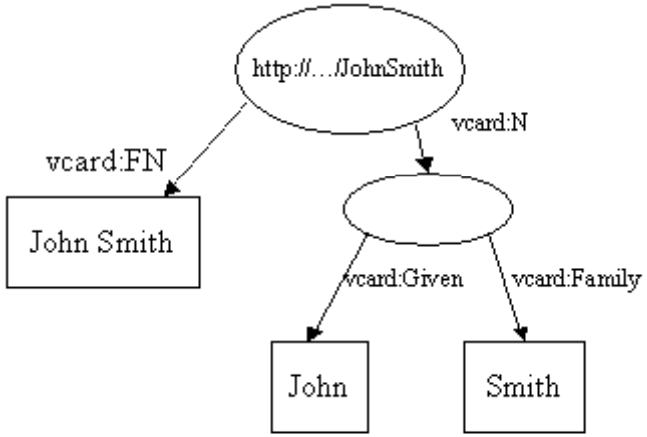
- Example:

```
Resource johnSmith =
  model.createResource(personURI)
    .addProperty(VCARD.FN, fullName)
      .addProperty(VCARD.TITLE, "Officer");
```

RDF triples - Properties

- Jena supports the following properties:
  - RDF (com.hp.hpl.jena.vocabulary.RDF)
    for e.g. RDF.Bag, RDF.predicate
  - RDF-Schema (com.hp.hpl.jena.vocabulary.RDFS)
    for e.g. RDFS.Class, RDFS.subClassOf
  - VCARD (com.hp.hpl.jena.vocabulary.VCARD)
    for e.g. VCARD.FN, VCARD.BDAY
  - Dublin Core (com.hp.hpl.jena.vocabulary.DC)
    for e.g. DC.creator, DC.description

RDF triples – Blank Nodes




```
String personURI = "http://somewhere/JohnSmith";
String givenName = "John";
String familyName = "Smith";
String fullName = givenName + " " + familyName;
Model model = ModelFactory.createDefaultModel();
Resource johnSmith
= model.createResource(personURI)
    .addProperty(VCARD.FN, fullName)
        .addProperty(VCARD.N,
            model.createResource()
                .addProperty(VCARD.Given, givenName)
                    .addProperty(VCARD.Family, familyName));
```

RDF triples – Browsing a Graph

- An RDF Model is represented as a *set* of statements.
- Each call of addProperty adds a new statement to the Model
- The listStatements() methods returns an StmtIterator, a subtype of Java's Iterator over all all the statements in a Model
  - hasNext returns a boolean
  - nextStatement returns the next statement from the iterator.
- The Statement interface provides accessor methods to the subject (getSubject), predicate (getPredicate) and object (getObject) of a statement

RDF triples – Browsing a Graph Example

```
StmtIterator iter = model.listStatements();
while (iter.hasNext()) {
        Statement stmt = iter.nextStatement();
        Resource subject = stmt.getSubject();
        Property predicate = stmt.getPredicate();
        RDFNode object = stmt.getObject();
        System.out.print(subject.toString());
        System.out.print(" " + predicate.toString() + " ");
        if (object instanceof Resource)
                System.out.print(object.toString());
        else
        System.out.print(" \"" + object.toString() + "\"");
        System.out.println(" .");
}
```

- Output:

```
http://somewhere/JohnSmith  http://www.w3.org/2001/vcard-rdf/3.0#N
anon:14df86:ecc3dee17b:-7fff .
anon:14df86:ecc3dee17b:-7fff  http://www.w3.org/2001/vcard-rdf/3.0#Family  "Smith" .
....
```

READ / WRITE RDF MODELS

Writing an RDF Model

- Jena has methods for reading and writing RDF as XML. These can be used to save an RDF model to a file and later read it back in again.
- The *write*(...) method in the class *Model*:

```
write(java.io.OutputStream out, java.lang.String lang, java.lang.String base)
```

  - **out** - The output stream to which the RDF is written
  - **lang** - The language in which the RDF should be written. "RDF/XML" (défaut), "RDF/XML-ABBREV" (RDF/XML plus compact), "N3" (Notation 3).
  - **base** - The base uri to use when writing relative URI's. null means use only absolute URI's.

Writing an RDF Model – Example 1

- Example:

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

- Output:

```
<rdf:RDF
        xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
>
        <rdf:Description rdf:about='http://somewhere/JohnSmith'>
                <vcard:FN>John Smith</vcard:FN>
                <vcard:N rdf:nodeID="A0"/>
        </rdf:Description>
        <rdf:Description rdf:nodeID="A0">
                <vcard:Given>John</vcard:Given>
                <vcard:Family>Smith</vcard:Family>
        </rdf:Description>
</rdf:RDF>
```

Writing an RDF Model – Example 2

- Example:

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

- Output:

```
<rdf:RDF
        xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'
        xmlns:vcard='http://www.w3.org/2001/vcard-rdf/3.0#'
>
        <rdf:Description rdf:about="http://somewhere/JohnSmith">
                <vcard:FN>John Smith</vcard:FN>
                <vcard:N rdf:parseType="Resource">
                        <vcard:Family>Smith</vcard:Family>
                        <vcard:Given>John</vcard:Given>
                </vcard:N>
        </rdf:Description>
</rdf:RDF>
```

Reading an RDF Model

- The *read*(...) method in the class *Model*:
  - read(java.lang.String url) reads the model from an XML document.
  - read(java.lang.String url, java.lang.String lang) reads the model from an XML document in a specified language (i.e lang).
  - read(java.io.InputStream in, java.lang.String base, java.lang.String lang) reads the model from an input stream in a specified language, using the base URI.

Reading an RDF Model– Example

- Example 1:

```
Model model = ModelFactory.createDefaultModel();
model.read("file:///home/moi/base.rdf", "RDF/XML");
```

- Example 2:

```
Model model = ModelFactory.createDefaultModel();
InputStream in = FileManager.get().open("base.rdf");
if (in != null)
        model.read(in, "");
```

MANIPULATING RDF GRAPHS

Creating / Accessing RDF Resources

- *Model.createResource(String uri)* and *Model.getResource(String uri)* return the created resource or the resource of the specified URI.
- *Resource.getProperty(Property)* returns a Statement (or NULL).
  - Note: it returns only one even if many exist.
- Example:

```
String johnSmithURI = "http://somewhere/JohnSmith/";
Resource vcard=model.createResource(johnSmithURI);
Resource name=vcard.getProperty(VCARD.N).getResource();
```

  - *Statement.getResource* returns the object resource.
  - *Statement.getString* returns the object Literal.

Accessing Properties

- It is possible that a property may occur more than once, then the *Resource.listProperties(Property p)* method can be used to return an iterator which will list them all.
- Example:

```
// list the nicknames
StmtIterator iter = vcard.listProperties(VCARD.NICKNAME);
while (iter.hasNext()) {
    System.out.println("    " +
        iter.nextStatement().getObject().toString());
}
```

Navigating an RDF Graph (1/2)

- *Model.listStatements* returns an iterator which lists all the statements of the model.
- *Model.listStatements(Resource s, Property p, RDFNode o)* returns an iterator which lists some specific statements in the model.
- *listSubjects* returns a *ResIterator* which lists all the resources that are the subject of some statement.
- *listSubjectsWithProperty(Property p, RDFNode o)* returns a *ResIterator* over all the resources which have property p with value o.

Navigating an RDF Graph (2/2)

- The generic method to select statements:
  - *listStatements(Selector)*. *Selector* defines the statements to browse.
  - *SimpleSelector(subject, predicate, object)*, an implementation of the *Selector* interface, allows to select some statements.
- **Example**: select all the resources with a VCARD.FN property whose value ends with "Smith".

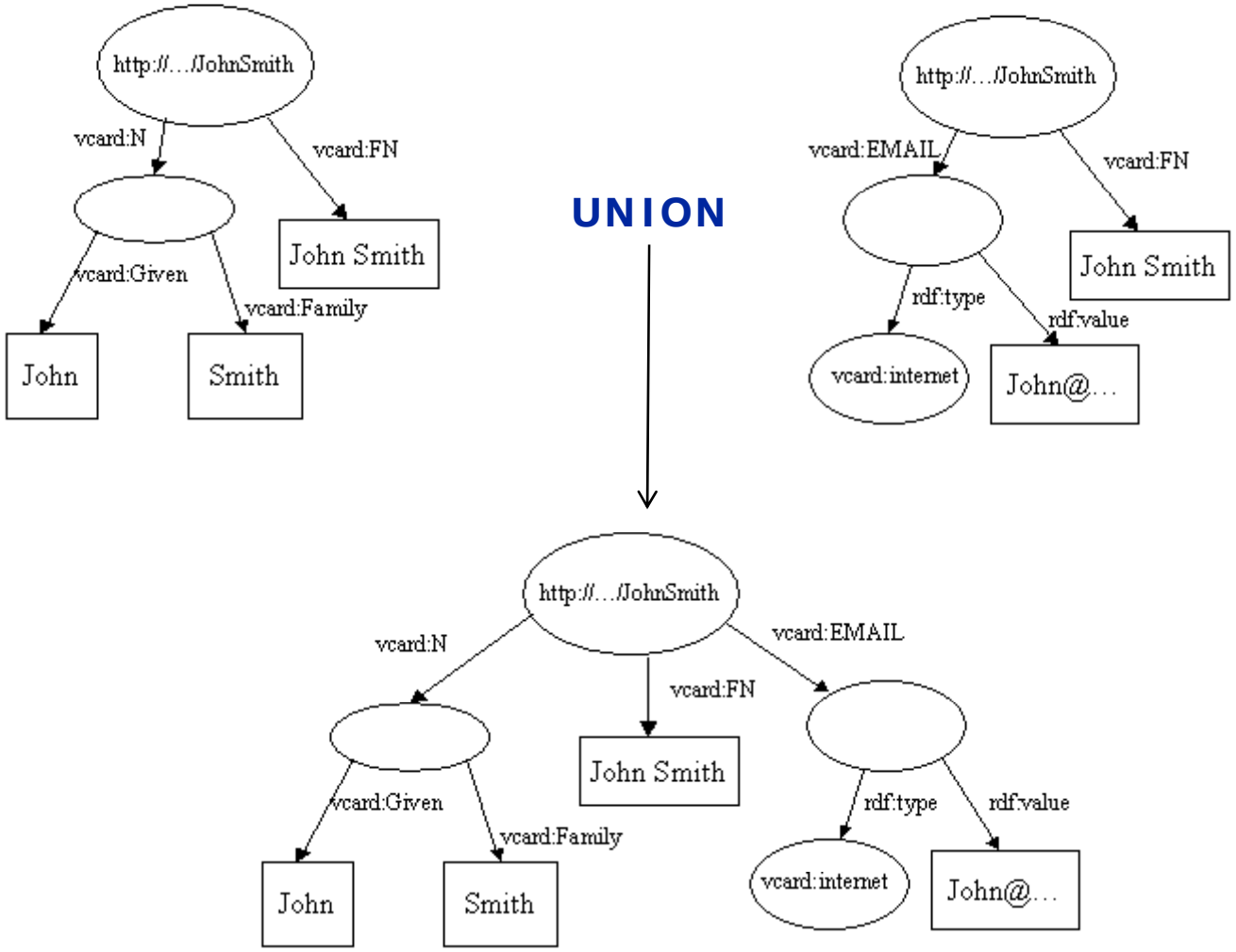
```
StmtIterator it = model.listStatements(
  new SimpleSelector(null, VCARD.FN, (RDFNode) null) {
    public boolean selects(Statement s){
      return s.getString().endsWith("Smith");
    }
  });
```

Operations on Models (1/2)

- *Model.union(Model)* returns a new Model, the model passed as parameter merged with the current model. Resources having the same URI will be merged.
- *Model.intersection(Model)* returns a new Model containing only common declarations.
- *Model.difference(Model)* returns a new Model containing only declarations that are in the current model but not in the one passed as parameter.

Operations on Models (2/2)

- Union example:



Containers (1/2)

- Jena supports the three kinds of container:
  - BAG: the *createBag* method
  - ALT: the *createAlt* method
  - SEQ: the *createSeq* method
- These methods can be called:
  - Without parameters: Creating an anonymous container.
  - With a String parameter: Creating a container identified through a passed URI.
- Supported methods on containers: add, contains, size, etc.

Containers (1/2)

- Example: a Bag containing the vcards of the Smith's

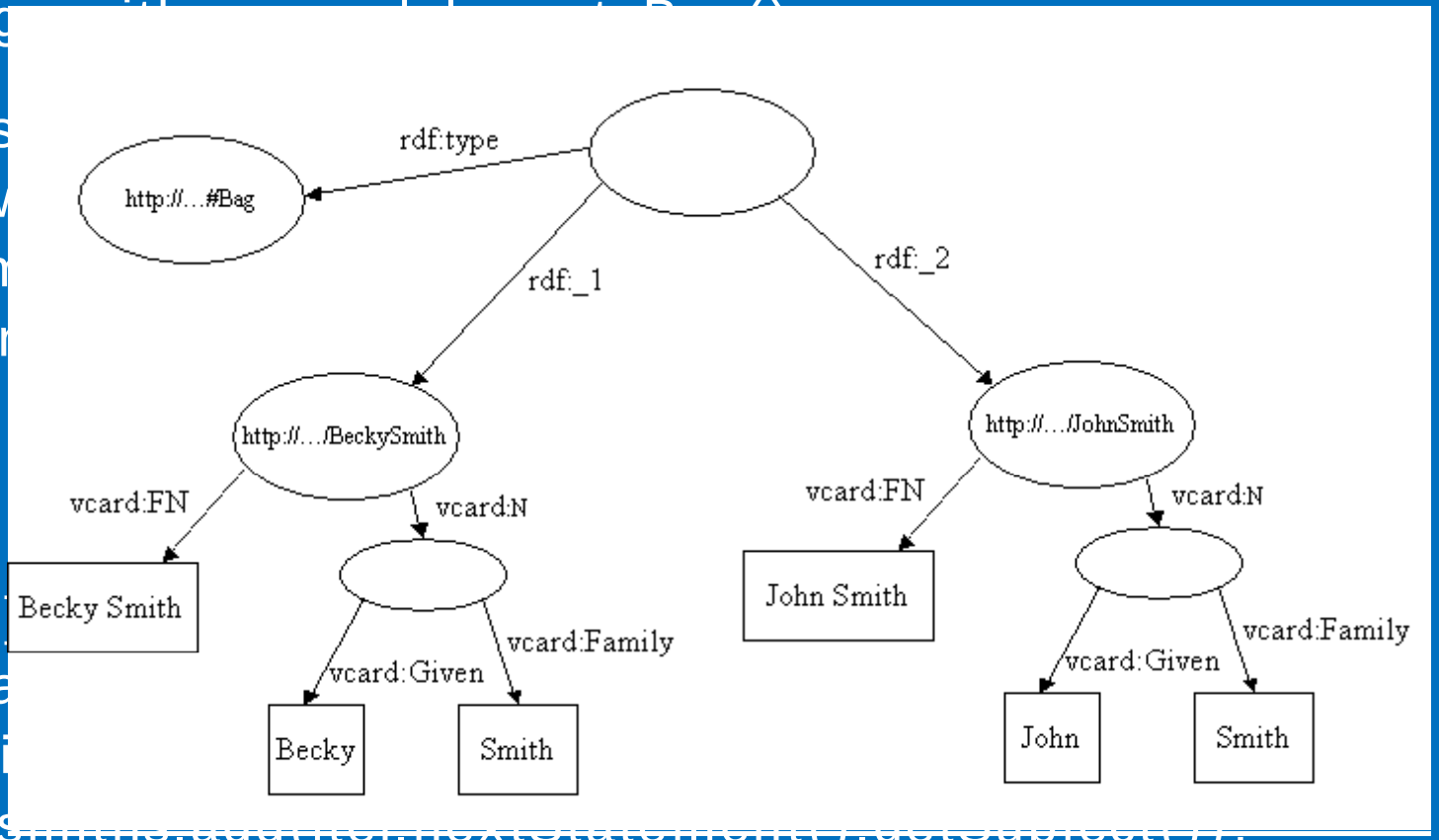
```
// create a bag
Bag smiths = model.createBag();

// select all the resources with a VCARD.FN property
// whose value ends with "Smith"
StmtIterator iter = model.listStatements(
    new SimpleSelector(null, VCARD.FN, (RDFNode) null) {
        public boolean selects(Statement s) {
                return s.getString().endsWith("Smith");
        }
    });
// add the Smith's to the bag
while (iter.hasNext()) {
    smiths.add(iter.nextStatement().getSubject());
}
```

Containers (1/2)

- Example: a Bag containing the vcards of the Smith's

```
// create a bag
Bag
// s
// v
Stm
    r                                                    {
    }
// a
whi
    Smiths.add(iter.nextStatement().getSubject());
}
```



USING SPARQL IN JENA

Usage from Command Line

- Jena provides a SPARQL interpreter that can be used through a command line interface.
- Configuration:
  - Define the **`JENAROOT`** environment variable
  - Add the **`%JENAROOT%\bat`** folder to your **`path`** variable
- Usage:
  - **`sparql`**
    - --graph URL (optional) : specify the used graph. Must be specified if there is no FROM clause in the query.
    - --query FILE : a text file containing the query.

Usage from a Java Program

1. Create a Query using the *QueryFactory.create(String query)* method. *query* is the text of the SPARQL query.
2. Create a *QueryExecution* using the *QueryExecutionFactory.create* method. Parameters are a *Query* and a *Model*.
3. Call *execSelect* on the created *QueryExecution*. This method returns a *ResultSet*.
4. Browse the returned *ResultSet* using the *hasNext* and *nextSolution* methods returning a *QuerySolution*. *QuerySolution* offers the *getResource* and *getLiteral* methods with the SELECT variable names as parameters.
5. End the execution of the *QueryExecution* using *close*.

Usage from a Java Program

- Example:

```
import java.util.*; import com.hp.hpl.jena.rdf.model.*;
import com.hp.hpl.jena.util.iterator.*;
import com.hp.hpl.jena.util.*; import com.hp.hpl.jena.query.*;
public class sparql {
   public static void main(String args[]) {
      Model model = FileManager.get().loadModel("file:personnes.n3");
1     String queryString = "PREFIX m: <http://exemple.fr/mv#> " +
                           "SELECT ?uri ?nom ?photo WHERE { "?uri m:nom ?nom . " +
                           "OPTIONAL { ?uri m:photo ?photo . } . }";
2     Query query = QueryFactory.create(queryString) ;
      QueryExecution qexec=QueryExecutionFactory.create(query,model);
      try {
3        ResultSet results = qexec.execSelect() ;
         for ( ; results.hasNext() ; ) {
4          QuerySolution soln = results.nextSolution() ;
           Resource uri = soln.getResource("uri") ;
           Resource photo = soln.getResource("photo") ;
           Literal nom = soln.getLiteral("nom") ;
           System.out.print(uri.toString()+" "+ nom.toString()+" ");
           if (photo != null) System.out.print(photo.toString());
           System.out.println();
         }
5     } finally { qexec.close(); };
```

MANIPULATING ONTOLOGIES

Ontologies and Jena

- With Jena we can handle ontologies (RDF-Schema, DAML+OIL, OWL) and we can ‘‘reason’’ using the knowledge offered by the ontology.
- An ontology is a special type of *Model* (i.e. *OntModel*) offering some ontologies specific methods:
  - Create a class: the method *createClass* that returns an *OntClass* (*OntClass* is a specialization of *Resource*)
  - Create a property: the method *createObjectProperty* that returns an *ObjectProperty* (*ObjectProperty* is a specialization of *Resource*)

Create an Ontology

- The method *ModelFactory.createOntologyModel* using as parameter:
  - *OntModelSpec.RDFS_MEM* : For an RDF-S ontology in memory whitout reasoning.
  - *OntModelSpec.RDFS_MEM_RDFS_INF* : For an RDF-S ontology in memory, with reasoning
  - etc.
- Generally, the ontology is loaded from a stream: using the *read* method.

Elements of an Ontology

- The elements of an ontology (*OntClass*, *ObjectProperty*) are specializations of the *OntResource* class defining some extra methods:
  - *getComment*, *setComment*, . . .
  - *getLabel*, *setLabel*, . . .
  - *getSeeAlso*, *setSeeAlso*, . . .
  - *getIsDefinedBy*, *setIsDefinedBy*, . . .

Classes: OntClass

- Example 1: create classes

```
String exns = "http://www.exemple.com/voc#";
OntClass veh = m.createClass(exns + "Vehicle");
OntClass car= m.createClass(exns + "Car");
car.addSuperClass(veh);
```

- Example 2: read classes

```
OntClass cl = m.getOntClass(exns + "Vehicle");
for (ExtendedIterator i = cl.listSubClasses();i.hasNext();)
{
    OntClass c = (OntClass) i.next();
    System.out.print(c.getLocalName() + " ");
}
```

Properties: ObjectProperty

- Some Methods:
  - [*has*, *list* , *set*, *add*]
    - *subProperty*, *superProperty*
    - *domain*. If several domains, their intersection is given.
    - *range*. If several domains, their intersection is given.

```
OntClass engine= m.createClass(exns + "Engine");
ObjectProperty pcomp = m.createObjectProperty(exns +"composant");
ObjectProperty pengine = m.createObjectProperty(exns +"engine");
pengine.addSuperProperty(pcomp);
pengine.addDomain(veh);
pengine.addRange(engine);
```

- *createDatatypeProperty* for properties with Literals in the domain.

Ontologies and Factual Knowledge

- Commonly, the ontology is stored in one (or several) files (RDFS for eg.) and the factual knowledge in others (RDF for eg.) …
- In such cases, we have to:
  - Create a Model for the ontology.
  - Create a Model for the facts.
  - Create a Model for the inferences (a union of both)
  - Use the model that allows the inference.

Ontologies and Factual Knowledge

- Example (RDF-S Ontology : exemplerdfs/rdfs.rdf)

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
<!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
<!ENTITY rdfs 'http://www.w3.org/2000/01/rdf-schema#'>
<!ENTITY ex 'file:exemplerdfs/rdfs.rdf#'>
<!ENTITY xsd "http://www.w3.org/2001/XMLSchema#">
]>
<rdf:RDF xmlns:rdf="&rdf;" xmlns:rdfs="&rdfs;" xmlns:ex="&ex;" xmlns="&ex;">
<rdf:Description rdf:about="&ex;mum">
        <rdfs:subPropertyOf rdf:resource="&ex;parent"/>
</rdf:Description>
<rdf:Description rdf:about="&ex;parent">
        <rdfs:range rdf:resource="&ex;Person"/>
        <rdfs:domain rdf:resource="&ex;Person"/>
</rdf:Description>
<rdf:Description rdf:about="&ex;age">
        <rdfs:range rdf:resource="&xsd;integer" />
</rdf:Description>
</rdf:RDF>
```

Ontologies and Factual Knowledge

- Example (Facts: exemplerdfs/rdf.rdf)

```
<?xml version="1.0"?>
<!DOCTYPE rdf:RDF [
<!ENTITY rdf 'http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
<!ENTITY ex 'file:exemplerdfs/rdfs.rdf#'>
]>
<rdf:RDF xmlns:rdf="&rdf;" xmlns:ex="&ex;">
<ex:Teenager rdf:about="http://www.exemple.com/colin">
        <ex:mum rdf:resource="rosy" />
        <ex:age>13</ex:age>
</ex:Teenager>
</rdf:RDF>
```

Ontologies and Factual Knowledge

- Example: method the prints the statement

```
public static void printStatements(Model m, Resource s, Property p, Resource o)
{
        for (StmtIterator i=m.listStatements(s,p,o); i.hasNext();)
        {
                Statement stmt = i.nextStatement();
                System.out.println(" - " + PrintUtil.print(stmt));
        }
}
```

Ontologies and Factual Knowledge

- Example:

```
public static void main(String args[])
{
   Model schema = FileManager.get().loadModel("file:exemplerdfs/rdfs.rdf");
   Model data = FileManager.get().loadModel("file:exemplerdfs/rdf.rdf");
   InfModel infmodel = ModelFactory.createRDFSModel(schema, data);
   Resource colin = infmodel.getResource("http://www.exemple.com/colin");
   System.out.println(colin);
   printStatements(infmodel, colin, RDF.type, null);
   Resource Person = infmodel.getResource("file:exemplerdfs/rdfs.rdf#Person");
   printStatements(infmodel, Person, RDF.type, null);
}
```

  - Create a Model for the ontology.
  - Create a Model for the facts.
  - Create a Model for the inferences (a union of both)
  - Print Resources that are of type colin/Person

Ontologies and Factual Knowledge

- Output of the execution:
  - RDF:types of colin:
    - file:exemplerdfs/rdfs.rdf#Teenager

    Cause: declared in the RDF file.
    - file:exemplerdfs/rdfs.rdf#Person

    Cause : colin has a property mum and that is a subProperty of parent. And the domain of parent is Person
    - rdfs:Resource
  - RDF:types of Person
    - rdfs:Class
    - rdfs:Resource
  - And rosy is a Person

Ontologies and Factual Knowledge

- Checking the validity of a model:

```
ValidityReport validity = infmodel.validate();
        if (validity.isValid())
                System.out.println("OK");
        else
        {
                System.out.println("Conflicts");
                for (Iterator i = validity.getReports(); i.hasNext(); )
                {
                   ValidityReport.Report report =(ValidityReport.Report)i.next();
                   System.out.println(" - " + report);
                }
        }
```

Ontologies and Factual Knowledge

- Output of the execution:
  - Conflicts

```
Error (dtRange): Property
file:exemplerdfs/rdfs.rdf#age has a typed range
Datatype[http://www.w3.org/2001/XMLSchema#integer ->
class java.math.BigInteger]that is not compatible with
"13"
```

- Correction of the RDF document

```
<ex:age rdf:datatype="http://www.w3.org/2001/XMLSchema#integer">
      13
</ex:age>
```

References

- Slides based on:
  - The Jena tutorial “An Introduction to RDF and the Jena RDF API”. http://jena.sourceforge.net/tutorial/RDF_API/
  - SPARQL Tutorial http://jena.apache.org/tutorials/sparql.html
  - Jena

    http://jena.apache.org/