INF3580 – Semantic Technologies – Spring 2010

Lecture 3: Jena – A Java Library for RDF

Martin Giese

9th February 2010

Department of Informatics

University of Oslo

Today's Plan

Outline

Reminder: RDF triples

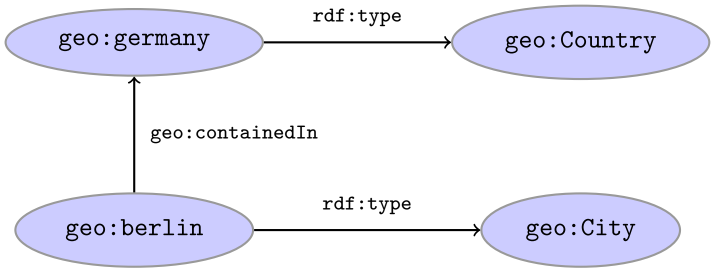
- The W3C representation of knowledge in the Semantic Web is RDF (Resource Description Framework)
- In RDF, all knowledge is represented by *triples*
- A triple consists of *subject*, *predicate*, and *object*
- For instance:
  `geo:germany rdf:type geo:Country .`
- These *qnames* are abbreviations for URIs:
  `rdf:` ≡ `http://www.w3.org/1999/02/22-rdf-syntax-ns#`
  `geo:` ≡ `http://geo.example.com/#`
- Expanded:
  `<http://geo.example.com/#germany>`
  `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`
  `<http://geo.example.com/#Country> .`

Reminder: RDF graphs

Sets of RDF triples are often represented as graphs:

Berlin is a City in Germany, which is a country

```
geo:germany rdf:type geo:Country .
geo:berlin rdf:type geo:City .
geo:berlin geo:containedIn geo:germany .
```

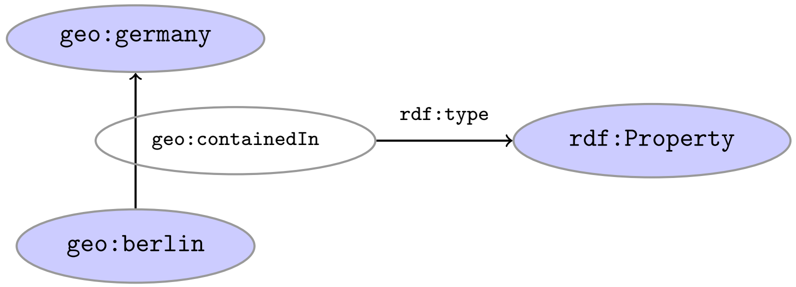


Reminder: RDF graphs (cont.)

Graph representation not always a perfect fit.

Berlin is contained in Germany, and containment is a property

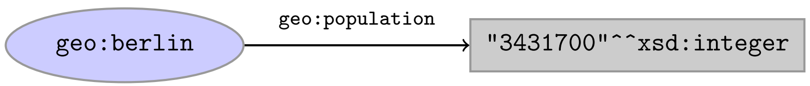
```
geo:berlin geo:containedIn geo:germany .
geo:containedIn rdf:type rdf:Property .
```



Usually speak about *RDF graphs* anyway

Reminder: RDF Literals

- *objects* of triples can also be *literals*
  - I.e. nodes in an RDF graph can be *resources* or *literals*
  - Subjects and predicates of triples can *not* be literals
- Literals can be
  - Plain, without language tag:
    `geo:berlin geo:name "Berlin" .`
  - Plain, with language tag:
    `geo:germany geo:name "Deutschland"@de .`
    `geo:germany geo:name "Germany"@en .`
  - Typed, with a URI indicating the type:
    `geo:berlin geo:population "3431700"^^xsd:integer .`
- Usually represented with rectangles:

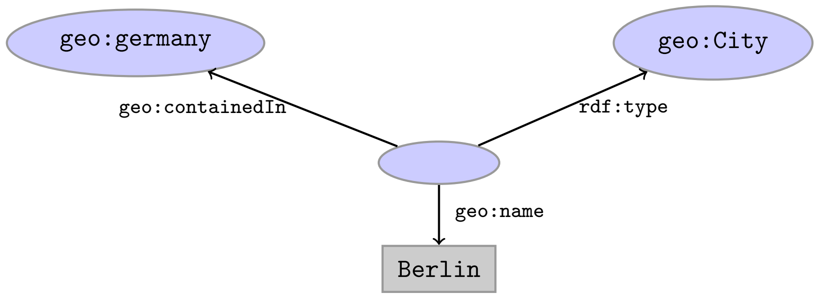


Reminder: RDF Blank Nodes

Blank nodes are like resources without a URI

There is a city in Germany called Berlin

```
_:x rdf:type geo:City .
_:x geo:containedIn geo:germany .
_:x geo:name "Berlin" .
```



Outline

Vital Statistics

- An open source Java framework for building Semantic Web applications.

  `http://jena.sourceforge.net/`
- Grown out of work with the HP Labs Semantic Web Programme
- includes:
  - An RDF API
  - Reading and writing RDF in RDF/XML, N3 and N-Triples
  - An interface to reasoning services
  - An OWL API
  - In-memory and persistent storage
  - A SPARQL query engine

Information About Jena

- *Public interface* of Jena has ca. 500 classes and interfaces in ca. 20 packages
- Can do useful things knowing only a small part of them!
- The Jena Tutorial:
  `http://jena.sourceforge.net/tutorial/RDF_API/index.html`
- The API Javadocs:

  `http://jena.sourceforge.net/javadoc/index.html`
- The Jena FAQ:

  `http://jena.sourceforge.net/jena-faq.html`
- In case of doubt: RTFM

Data Representations: URIs

- Start by investigating how different RDF concepts are represented in Jena.
- URIs are simply represented as strings:

```
String germanyURI="http://geo.example.com/#germany"
```

- Probably a good idea to put namespaces in separate strings:

```
String geoNS="http://geo.example.com/#";
String germanyURI=geoNS+"germany";
String berlinURI =geoNS+"berlin";
```

Data Representation: Resources

- Most of the basic RDF representations covered by classes in

  `com.hp.hpl.jena.rdf.model`
- Resources are represented by

  `Resource`
- Has a method

  `String getURI()`
- But wait. . . `Resource` is an interface. How do you create an instance?
- There is a class `ResourceFactory` with method

  `static Resource createResource(String uriref)`
- Beware: this is not usually what you want!

Data Representation: Models

- A `com.hp.hpl.jena.rdf.model.Model` represents a set of RDF statements (triples).
- In Jena, `Resource`s and `Statement`s are linked to the `Model`s they are part of.
- `Model`s also have the responsibility for *creating* `Resource`s, etc.
- Need to create a `Model` first.
- Also an interface! (Can this be on purpose?)
- Easiest way: `com.hp.hpl.jena.rdf.model.ModelFactory`

  `Model model = ModelFactory.createDefaultModel();`
- Other ways: with database storage, with reasoning, etc.
- Also deals with reading & writing various formats

Data Representation: Resources, 2nd try

- Given a model. . .

  `Model model = ModelFactory.createDefaultModel();`
- . . . and a URI. . .

  `String berlinURI = geoNS + "berlin";`
- . . . we can use it to create a `Resource`:

  `Resource berlin = model.createResource(berlinURI);`
- We can ask the `Resource` for the `Model`:

  `berlin.getModel()...`
- For a fresh blank node:

  `Resource blank = model.createResource();`

Data Representation: Properties

- Reminder: predicates are simply resources
- Jena defines a separate interface `Property`
- Doesn't add anything important to `Resource`
- To create a `Property object:`

  `Property name = model.createProperty(geoNS+"name");`

Data Representation: Literals

- Jena defines a `Literal` interface for all three kinds of literals.
- To create a plain literal:

```
Literal b = model.createLiteral("Berlin");
```

- To create a literal with language tag:

```
Literal d = model.createLiteral("Germany","en");
```

- To create a typed literal:

```
String type = "http://www.w3.org/2001/XMLSchema#byte";
Literal n = model.createTypedLiteral("42",type);
```

- Or, with a `com.hp.hpl.jena.datatypes.RDFDatatype`:

```
import com.hp.hpl.jena.datatypes.xsd.XSDDatatype;

RDFDatatype type = XSDDatatype.XSDbyte;
Literal n = model.createTypedLiteral("42",type);
```


Data Representation: Statements

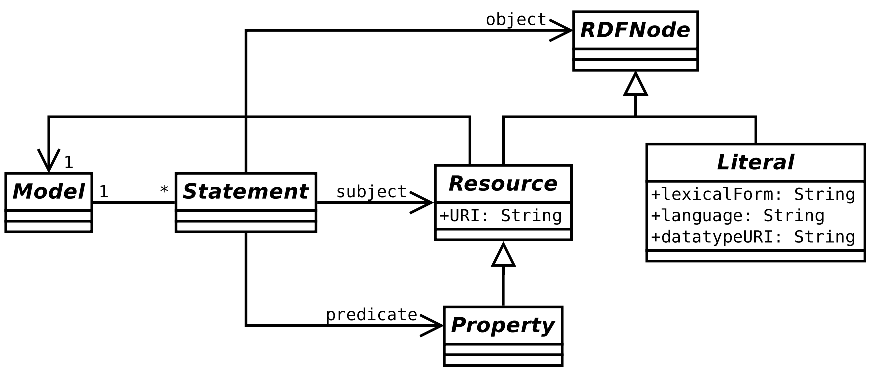
- To construct a `Statement`, you need
  - A subject, which is a `Resource`
  - A predicate, which is a `Property`
  - An object, which can be a `Resource` or a `Literal`
- Again, use the methods in `Model`:

```
Resource berlin = model.createResource(geoNS+"berlin");
Property name = model.createProperty(geoNS+"name");
Literal b = model.createLiteral("Berlin");
Statement stmt = model.createStatement(berlin,name,b);
```

- Not yet asserted in the model.
- To add this statement to the model:

```
model.add(stmt);
```


Overview



Convenience Methods in `Resource`

- Can directly add statements to the model.
- Given some properties and resources...

```
Property name = model.createProperty(geoNS+"name");
Property cont = model.createProperty(geoNS+"containedIn");
Property pop = model.createProperty(geoNS+"population");

Resource berlin = model.createProperty(geoNS+"berlin");
Resource germany = model.createProperty(geoNS+"germany");
```

- ...we can write:

```
berlin.addProperty(cont, germany);
berlin.addProperty(name, "Berlin");
germany.addProperty(name, "Tyskland","no");
berlin.addLiteral(pop, 3431700);
```

- Directly adds statements to `model`!
- Converts Java datatypes to RDF literals.

Models and Graphs, Statements and Triples

- In Jena, they have both *triples* and *statements!?*
- There are also both *graphs* and *models!?*
- Jena is a *framework!*
  - unified view for differing implementations of data storage and processing
- High-level interface
  - API: application programming interface
  - Convenient to use
  - Classes `Resource`, `Statement`, `Model`
- Low-level interface
  - SPI: service provider interface
  - Easy to implement
  - Classes `Node`, `Triple`, `Graph`
- We will be concerned mostly with the API!

Outline

Retrieving Information from a `Model`

- We've seen how to add statements to a `Model`
- Two ways to retrieve information:
  - Via `Resources`
  - Via the `Model`
- Navigation through resources delegates to model, but sometimes more convenient

Retrieving Information from a `Resource`

- `Resource` has methods to retrieve statements having the resource as subject.
- `To find all statements about berlin`

```
Iterator<Statement> it = berlin.listProperties();
```

- to print them all out:

```
while (it.hasNext()) {
    System.out.println(it.next());
}
```

- to find all statements with a particular predicate:

```
Property name = model.createProperty(geoNS+"name");
```

```
Iterator<Statement> it = berlin.listProperties(name);
```

Retrieving Information from a Resource (cont.)

- To get *some* statement, without iterating:

```
Property pop = model.createProperty(geoNS+"population");
berlin.getProperty(pop)
```

- B.t.w., to access the object of a statement as a Java type:

```
int n = berlin.getProperty(pop).getInt();
```

- See also methods
  - getRequiredProperty
  - hasProperty,
  - hasLiteral,

Retrieving information from a Model

- To get *all* statements from a Model:

```
Iterator<Statement> sit = model.listStatements();
```

- To get all resources that are subject of some statement:

```
Iterator<Resource> rit =
    model.listSubjects();
```

- To get all resources with a statement for a given predicate:

```
Iterator<Resource> rit =
    model.listResourcesWithProperty(name);
```

- ... with a given value for a property:

```
Iterator<Resource> rit =
    model.listResourcesWithProperty(cont, germany);
```


Simple Pattern Matching

- To get all statements that have
  - a given subject and object,
  - a given object,
  - a given predicate and subject,
  - or any other combination...
- ... use

```
Iterator<Statement> sit =
    model.listStatements(subj, pred, obj);
```

- where subj, pred, obj can be null to match any value ("wildcard")
- e.g. to print everything contained in Germany:

```
Iterator<Statement> sit =
    model.listStatements(null, cont, germany);
while (sit.hasNext()) {
    System.out.println(sit.next().getSubject());
}
```


Complex Pattern Matching

- W3C has defined the SPARQL language
- SPARQL Protocol And RDF Query Language
- The Semantic Web equivalent of SQL
- Jena Models can process SPARQL queries
- A much more powerful way of retrieving data from a Model
- More about this next week!

Outline

Reading RDF

- `Model` contains several `read(...)` methods for reading RDF.
- `read` does not create a new `Model` object.
  - First create a model, then add statements with `read`.
  - Can call read several times to accumulate information.
- Available formats: RDF/XML, N-triples, Turtle, N3.
  - Format defaults to RDF/XML
  - Variants with format (`lang`) parameter exist
- Can read from `InputStream` or `Reader`, or directly from a `URL`.
- Some `read` variants take a "base URI".
  - Used to interpret relative URIs in the document.
  - Usually not needed: absolute URIs are a better idea.
- Example: Load Martin Giese's FOAF file from the 'net:

```
Model model = ModelFactory.createDefaultModel();
model.read("http://heim.ifi.uio.no/martingi/foaf.rdf");
```


Writing RDF

- `Model` contains several `write(...)` methods for writing RDF.
- Available formats: RDF/XML, N-triples, Turtle, N3.
  - Format defaults to RDF/XML
  - Variants with format (`lang`) parameter exist
- Can write to `OutputStream` or `Writer`.
- Some `write` variants take a "base URI".
  - Used to make some URIs relative in the output.
  - Absolute URIs are a better idea.
- Example: write `model` to a file:

```
try {
    model.write(new FileOutputStream("output.rdf"));
}catch (IOException e) {
    // handle exception
}
```

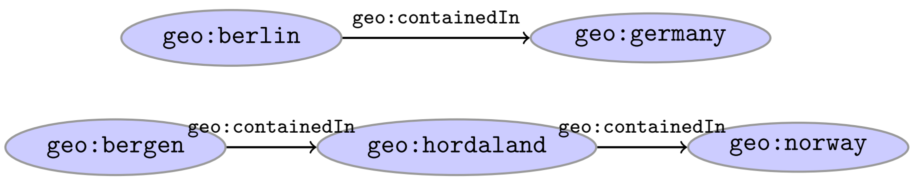

Outline

A Containment Example

Given an RDF/XML file with information about containment of places in the following form:

Geographic containments

```
geo:berlin geo:containedIn geo:germany .
geo:bergen geo:containedIn geo:hordaland .
geo:hordaland geo:containedIn geo:norway .
...
```

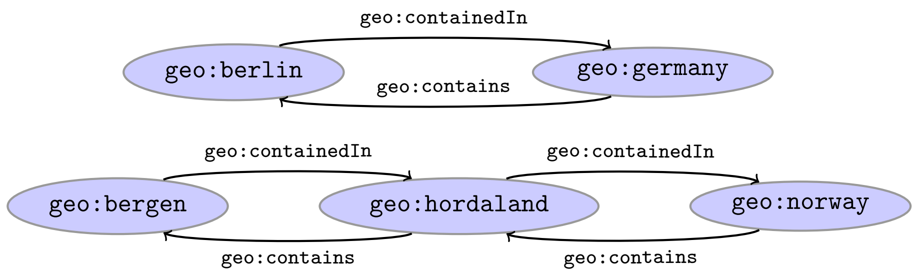


A Containment Example (cont.)

Add inverse statements using property `geo:contains`:

Inverted Containment Statements

```
geo:germany geo:contains geo:berlin .
geo:hordaland geo:contains geo:bergen .
geo:norway geo:contains geo:hordaland .
...
```



Solution: Creating the Model, Reading the File

```java
import java.io.*;
import java.util.*;
import com.hp.hpl.jena.rdf.model.*;

public class Containment {

   public static String GEO_NS = "http://geo.example.com/#";

   public static void main(String[] args) throws IOException {
      Model model = ModelFactory.createDefaultModel();
      model.read(new FileInputStream("places.rdf"), null);

      Property containedIn =
         model.getProperty(GEO_NS+"containedIn");
      Property contains =
         model.getProperty(GEO_NS+"contains");
```


Solution: Adding Statements, Writing a File

```java
      Iterator<Statement> it =
            model.listStatements((Resource)null,
                                 containedIn,
                                 (Resource)null);
      while ( it.hasNext() ) {
         Statement st = it.next();
         model.add((Resource)st.getObject(),
                   contains,
                   st.getSubject());
      }

      model.write(new FileOutputStream("output.rdf"));
   } // main()

} // class Containment
```

Outline

57 Varieties of Models

- Until now: "default" models:

  ```
  ModelFactory.createDefaultModel();
  ```
- A simple collection of statements stored in memory
  - Large datasets require lots of RAM
  - Not persistent, need to read/write to files manually
- Models created by ModelFactory differ in
  - backing storage (Memory, files, RDB)
  - inferencing
    - automatically add triples that are consequences of others
    - more on this in lecture 6 and later!
  - reification style
    - resources representing statements
    - won't go into this

ModelMaker

- Jena likes to store models in groups, identified by names.
- ModelMaker organizes collections of *named* models.
- To create one that handles models stored in memory:

  ```
  ModelMaker mm = ModelFactory.createMemModelMaker();
  ```
- ...in a collection of file system files:

  ```
  ModelMaker mm =
      ModelFactory.createFileModelMaker("/path/to/files");
  ```
- ...a relational database:

  ```
  IDBConnection conn =
      new DBConnection(DB_URL,DB_USER,DB_PASSWD,DB_TYPE);

  ModelMaker mm =
      ModelFactory.createRDBModelMaker(conn);
  ```
- See book for example of creating a DBConnection!

ModelMaker (cont.)

- Given a ModelMaker object, you can...
  - create a new model if none under that name exists:

    ```
    Model model = mm.createModel("CitiesOfNorway");
    ```
  - open an already existing model:

    ```
    Model model = mm.openModel("CitiesOfNorway");
    ```
  - (also strict variants which throw an exception in the other case)
  - remove an already existing model from memory:

    ```
    mm.removeModel("CitiesOfNorway");
    ```
  - check if there is a model with a given name:

    ```
    if (mm.hasModel("CitiesOfNorway")) {...};
    ```
- All models are stored as tables in one RDB, files in one file system directory, etc.

Outline

Many Models

- Jena can manage many models simultaneously.
- E.g. some in memory, some in databases, etc.
- Different `Model` objects don't know of each other
- It is however possible to combine models:

```
Model u = model1.union(model2);
Model i = model1.intersection(model2);
Model d = model1.difference(model2);
```

- Models contain set union/intersection/difference of statements in `model1`/`model2`.
- These are new *independent* models:
  - adding/removing statements in `model1`/`model2` does not affect `u`/`i`/`d`
  - adding/removing statements in `u`/`i`/`d` does not affect `model1`/`model2`
- Typically a fresh memory model holding all data.

Dynamic Unions

- Also possible to create *dynamic* unions:

```
Model u = ModelFactory.createUnion(model1,model2);
```

- Model `u` contains set union of statements in `model1`/`model2`.
- `u` remains connected to `model1` and `model2`:
  - adding/removing statements in `model1`/`model2` adds/removes them in `u`
  - adding/removing statements in `u` adds/removes them in `model1`
- Union model delegates storage to other models

The Alignment Problem

- We built a database `places.rdf` with
  - Information about resources like
    `http://geo.example.com/#oslo`
    `http://geo.example.com/#germany`
  - Expressed in terms like
    `http://geo.example.com/#City`
    `http://geo.example.com/#Country`
    `http://geo.example.com/#containedIn`
- Now we discover `http://dbpedia.org/` with
  - information about resources like
    `http://dbpedia.org/resource/Oslo`
    `http://dbpedia.org/resource/Germany`
  - Expressed in terms like
    `http://dbpedia.org/ontology/PopulatedPlace`
    `http://dbpedia.org/ontology/Country`
    `http://dbpedia.org/property/subdivisionName`

The Alignment Problem (cont.)

- We can now construct the union of both information sources
- But the union will not be very useful :-(
- The data is not linked!
  - The same entities are identified by different URIs
  - The same types are identified by different URIs
  - Similar properties are identified by different URIs
- Need some way to "align" the vocabularies
  - Say that `geo:oslo` equals `dbpedia:Oslo`.
  - Say that a `geo:City` is a kind of `dbpedia-owl:PopulatedPlace`.
  - Say that subdivisions are contained in each other.
- You will learn how to do this later in the course. . .
- . . . but to get it right, some theory is needed!

Outlook

Lecture 4: The SPARQL Query Language
Lecture 5: Semantics of RDF
Lecture 6: The RDFS Vocabulary Definition Language
Lecture 7: OWL basics: Web Ontology Language
Lecture 8: More about OWL

- All this will be explained with examples
- There will be practical exercises
- But there are some theoretical concepts to grasp!