Jena

Werner Nutt

Acknowledgment

These slides are based on the slide set

- RDF
  By Mariano Rodriguez (see http://www.slideshare.net/marianomx)

Jena

- Available at http://jena.apache.org/
- Available under the Apache license.
- Developed by HP Labs
  (now community based development)
- Best-known framework
- Used to:
  - Create and manipulate RDF graphs
  - Query RDF graphs
  - Read/Serialize RDF from/into different syntaxes
  - Perform inference
  - Build SPARQL endpoints
- Tutorial: http://jena.apache.org/tutorials/rdf

Basic operations

- Creating a graph from Java
  - URIs/Literals/Bnodes
- Listing all “Statements”
- Writing RDF (Turtle/N-Triple/XML)
- Reading RDF
- Prefixes
- Querying (through the API)

Creating a basic graph

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

- *URIs in Jena are strings*

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

- *RDF graphs are "models" in the Jena API*
  (and "graphs" in the SPI = Service Provider API)
- Model *is an interface (why?)*
- *The* DefaultModel *is an empty "model"*

Creating a basic graph

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

- *Resources are attached to a model*

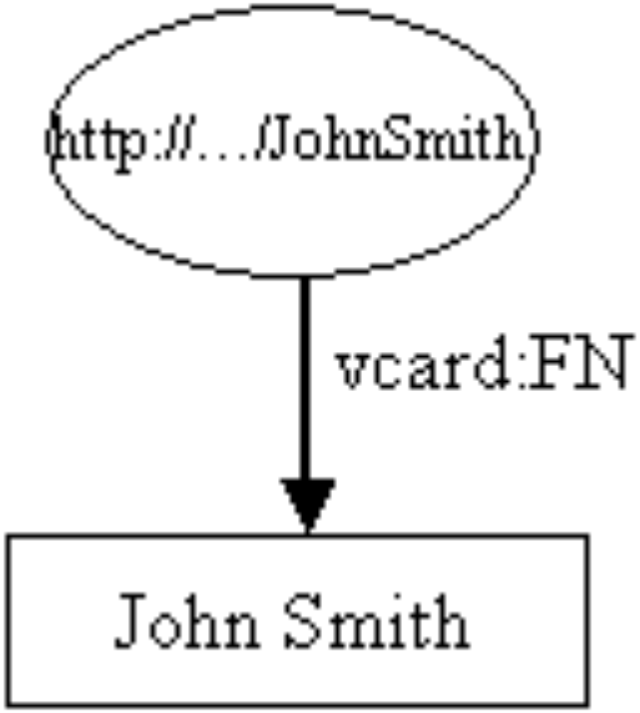
```
// add the property
 johnSmith.addProperty(VCARD.FN, fullName);
```

- *VCARD is the Jena "vocabulary class" for the vCard vocabulary*
  - *vCard terms are static attributes of the class VCARD*

- *addProperty takes two arguments*
  - *a property*
  - *a resource*

  and creates a statement as a side effiect

Result



Creating a basic graph

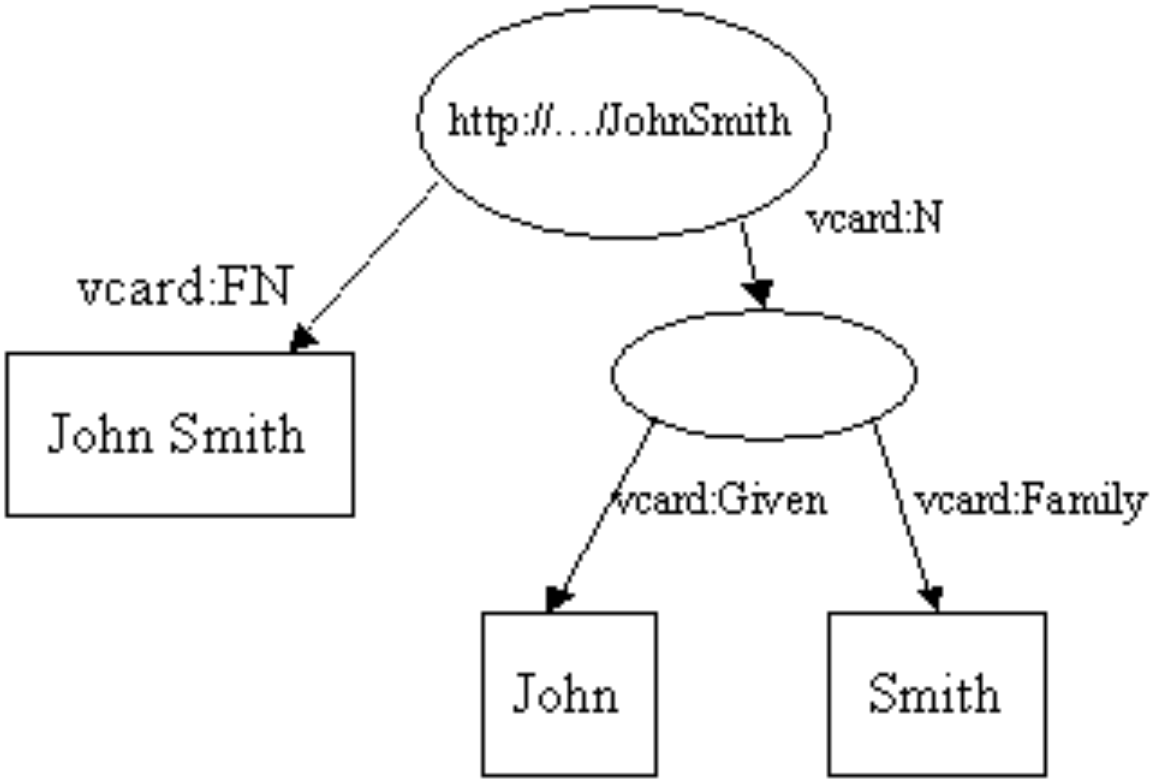
```
// some definitions
String personURI    = "http://somewhere/JohnSmith";
String givenName    = "John";
String familyName   = "Smith";
String fullName     = givenName + " " + familyName;

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

// create the resource
//   and add the properties cascading style
Resource johnSmith
  = model.createResource(personURI)
        .addProperty(VCARD.FN, fullName)
        .addProperty(VCARD.N,
                model.createResource()
                    .addProperty(VCARD.Given, givenName)
                    .addProperty(VCARD.Family, familyName));
```

createResource()
creates a blank node

Result


http://.../JohnSmith
vcard:FN
vcard:N
John Smith
vcard:Given
vcard:Family
John
Smith

Statements

- Statements and triples are the same kind of thing
- Models are sets of statements
  ➔ adding duplicate statements has no effect
- Statements are inserted into a model by calls of `addProperty`
- `Statements` in Jena have subject, predicate, and object, like in the RDF data model
  - and selectors `getSubject`, `getPredicate`, `getObject`
  - `getObject` returns instances of `RDFNode`, a superclass of `Resource` and `Literal`
- Statements can be retrieved from a model with a statement iterator `StmtIterator`

Listing the statements of a model

```
// list the statements in the Model
StmtIterator iter = model.listStatements();

// print out the subject, predicate, and object of each statement
while (iter.hasNext()) {
   Statement stmt      = iter.nextStatement();  // get next statement
   Resource  subject   = stmt.getSubject();     // get the subject
   Property  predicate = stmt.getPredicate();   // get the predicate
   RDFNode   object    = stmt.getObject();      // get the object

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

Output

Something like this:

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

Writing RDF

- Use the `model.write(OutputStream s)` method
  - any output stream is valid
- By default it will write in RDF/XML format
- Change format by specifying the format with:
  - `model.write(OutputStream s, String format)`
  - possible format strings:
    - "RDF/XML-ABBREV"
    - "N-TRIPLE"
    - "RDF/XML"
    - "TURTLE"
    - "TTL"
    - "N3"

Writing RDF

```
// now write the model in XML form to a file
model.write(System.out);
```

```
<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>
```

Default XML writer: Shortcomings

- Blank notes are not represented correctly
  - the blank node in the graph has been given a URI reference
- XML/RDF can represented blank nodes only to some extent:
  - only bNodes that appear no more than once as an object
- The Jena PrettyWriter (invoked by "RDF/XML-ABBREV")
  - produces more compact XML
  - gets bNodes right when possible
  - needs a lot of time

Reading RDF

- Use model.read(InputStream, String syntax)

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

// use the FileManager to find the input file
 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);

// write it to standard out
model.write(System.out);
```

the second argument of read(.,.) is the URI used for resolving relative (= shortened) URIs

Controlling Prefixes

- Prefixes are used in Turtle/RDF and other syntaxes
- Define prefixes prior to writing to obtain a “short” rendering

Example

```
Model m = ModelFactory.createDefaultModel();
String nsA = "http://somewhere/else#";
String nsB = "http://nowhere/else#";
Resource root = m.createResource( nsA + "root" );
Property P = m.createProperty( nsA + "P" );
Property Q = m.createProperty( nsB + "Q" );
Resource x = m.createResource( nsA + "x" );
Resource y = m.createResource( nsA + "y" );
Resource z = m.createResource( nsA + "z" );
m.add( root, P, x ).add( root, P, y ).add( y, Q, z );
System.out.println( "# -- no special prefixes defined" );
m.write( System.out );
System.out.println( "# -- nsA defined" );
m.setNsPrefix( "nsA", nsA );
m.write( System.out );
System.out.println( "# -- nsA and cat defined" );
m.setNsPrefix( "cat", nsB );
m.write( System.out );
```

Output 1

```
# -- no special prefixes defined

<rdf:RDF
    xmlns:j.0="http://nowhere/else#"
    xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
    xmlns:j.1="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
    <j.1:P rdf:resource="http://somewhere/else#x"/>
    <j.1:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
    <j.0:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

Output 2

```
# -- nsA defined

<rdf:RDF
   xmlns:j.0="http://nowhere/else#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:nsA="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
   <nsA:P rdf:resource="http://somewhere/else#x"/>
   <nsA:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
   <j.0:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

Output 3

```
# -- nsA and cat defined

<rdf:RDF
   xmlns:cat="http://nowhere/else#"
   xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
   xmlns:nsA="http://somewhere/else#" >
  <rdf:Description rdf:about="http://somewhere/else#root">
   <nsA:P rdf:resource="http://somewhere/else#x"/>
   <nsA:P rdf:resource="http://somewhere/else#y"/>
  </rdf:Description>
  <rdf:Description rdf:about="http://somewhere/else#y">
   <cat:Q rdf:resource="http://somewhere/else#z"/>
  </rdf:Description>
</rdf:RDF>
```

Prefixes: Summary

Namespace prefixes can be

- set by

  `m.setNsPrefix(` *prefix*, *uri*`);`

- dropped by

  `m.removeNsPrefix(` *prefix*`);`

The effect of prefixes depends on the linearization chosen for the output *(why?)*

Navigating the model

- The API allows one to query the model to get specific statements
- Use

  `model.getResource(`*uriString*`)`:

  to retrieve the resource or create a new one:
- With a resource, use `getProperty` to retrieve statements and their objects

  `resource.getProperty(`*uriString*`).getObject()`

  ➔ *what if a resource has several statements with the same predicate?*
- We can add further statements to the model through the resource

```
// retrieve the John Smith vcard resource from the model
Resource vcard = model.getResource(johnSmithURI);


// retrieve the value of the N property
Resource name = (Resource) vcard.getProperty(VCARD.N)
                          .getObject();


// retrieve the value of the N property
Resource name = vcard.getProperty(VCARD.N)
                  .getResource();


// retrieve the first name property
String fullName = vcard.getProperty(VCARD.FN)
                    .getString();
```

Adding statements

We add two nickname properties to John Smith's vcard

```
vcard.addProperty(VCARD.NICKNAME, "Smithy")
     .addProperty(VCARD.NICKNAME, "Adman");
```

We retrieve all the nicknames with an iterator

```
// set up the output
System.out.println("The nicknames of \""
                   + fullName + "\" are:");
// list the nicknames
StmtIterator iter =
vcard.listProperties(VCARD.NICKNAME);
while (iter.hasNext()) {
   System.out.println("    " + iter.nextStatement()
                              .getObject()
                              .toString());
}
```

We retrieve all the nicknames with an iterator

```
// set up the output
System.out.println("The nicknames of \""
                   + fullName + "\" are:");
// list the nicknames
StmtIterator iter =
vcard.listProperties(VCARD.NICKNAME);
while (iter.hasNext()) {
   System.out.println("    " + iter.nextStatement()
                              .getObject()
                              .toString());
}
```

Output:

```
The nicknames of "John Smith" are:
    Smithy
    Adman
```

Final notes

- Key API objects: DataSet, Model, Statement, Resource and Literal
- The default model implementation is in-memory
- Other implementations exists that use different storage methods
  - Native Jena **TDB**. Persistent, on disk, storage of models using Jena's own data structures and indexing techniques
  - **SDB**. Persistent storage through a relational database.
- We'll see more features as we advance in the course
- Third parties offer **their own triple stores** through Jena's API (OWLIM, Virtuoso, etc.)