Apache Jena Framework

Philippe Genoud – Université Joseph Fourier – Grenoble (France)
(Philippe.Genoud@imag.fr)

Introduction

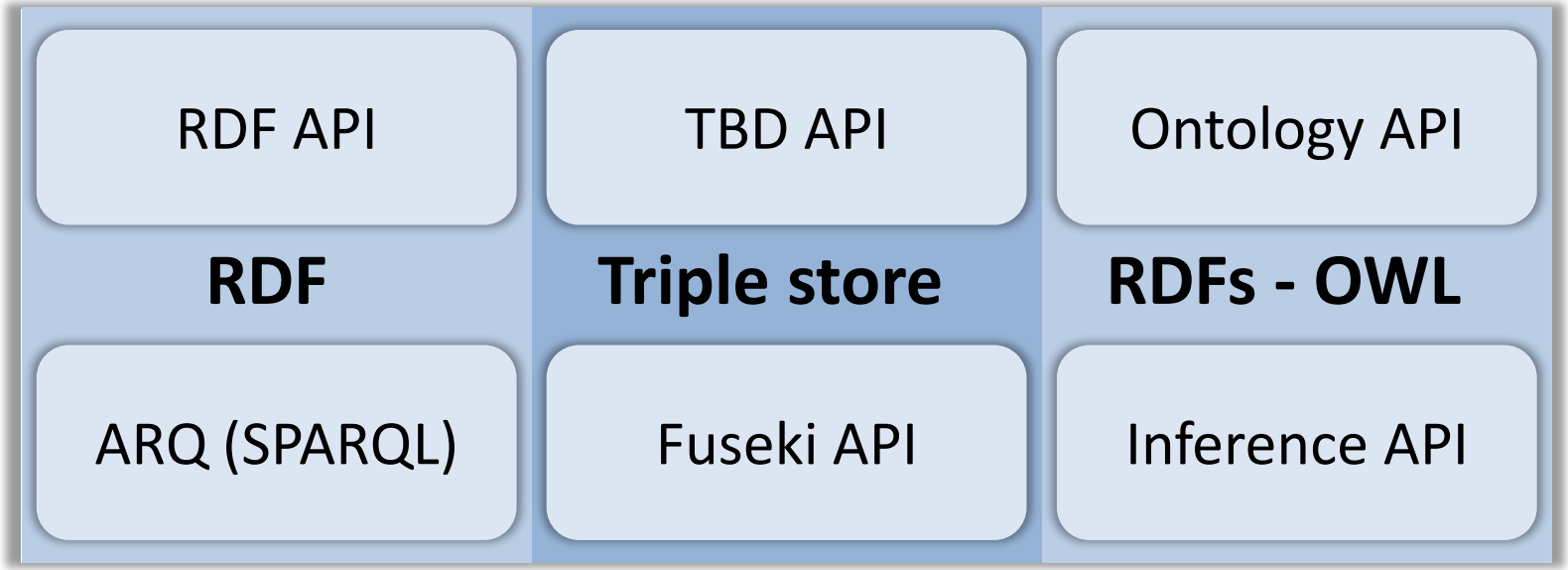
- What is Jena?
  - A free and open source Java framework for building Semantic Web and Linked Data applications.
  - developed at HP-Labs (Bristol-UK)
  - now an apache project : http://jena.apache.org/
    - November 2010: adopted by the Apache Software Foundation (incubation)
    - April 2012: graduated as a top-level project
    - Current version (Oct. 15, 2019) : 3.13.1

Jena APIs

Core API to create and read RDF graphs and serialize them in standard formats (RDF/XML, Turtle…)

A native high performance triple store to persist RDF data

API for handling OWL and RDFS ontologies to add extra semantics to RDF data

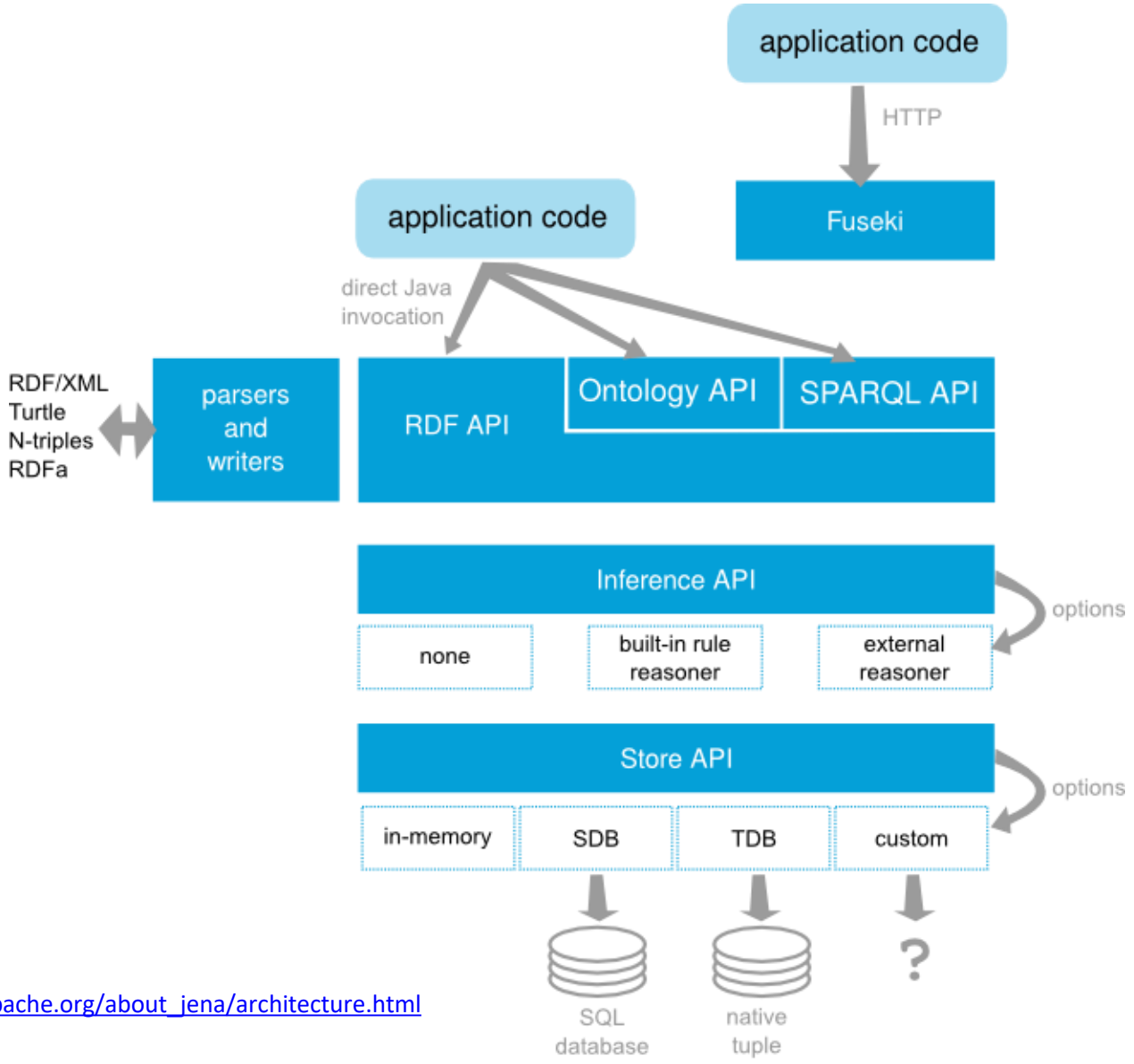


A query engine compliant with the latest SPARQL specification (1.1)

To expose RDF triples as a SPARQL end-point accessible over HTTP. Provides REST-style interaction with RDF data.

To reason over RDF data to expand and check it. Configure your own inference rules or use the built-in OWL and RDFS reasoners.

Apache Jena Architecture Overview



https://jena.apache.org/about_jena/architecture.html

Jena packages names

- prefixes for jena packages
  - `org.apache.jena.*`
    - ex: `org.apache.jena.riot`

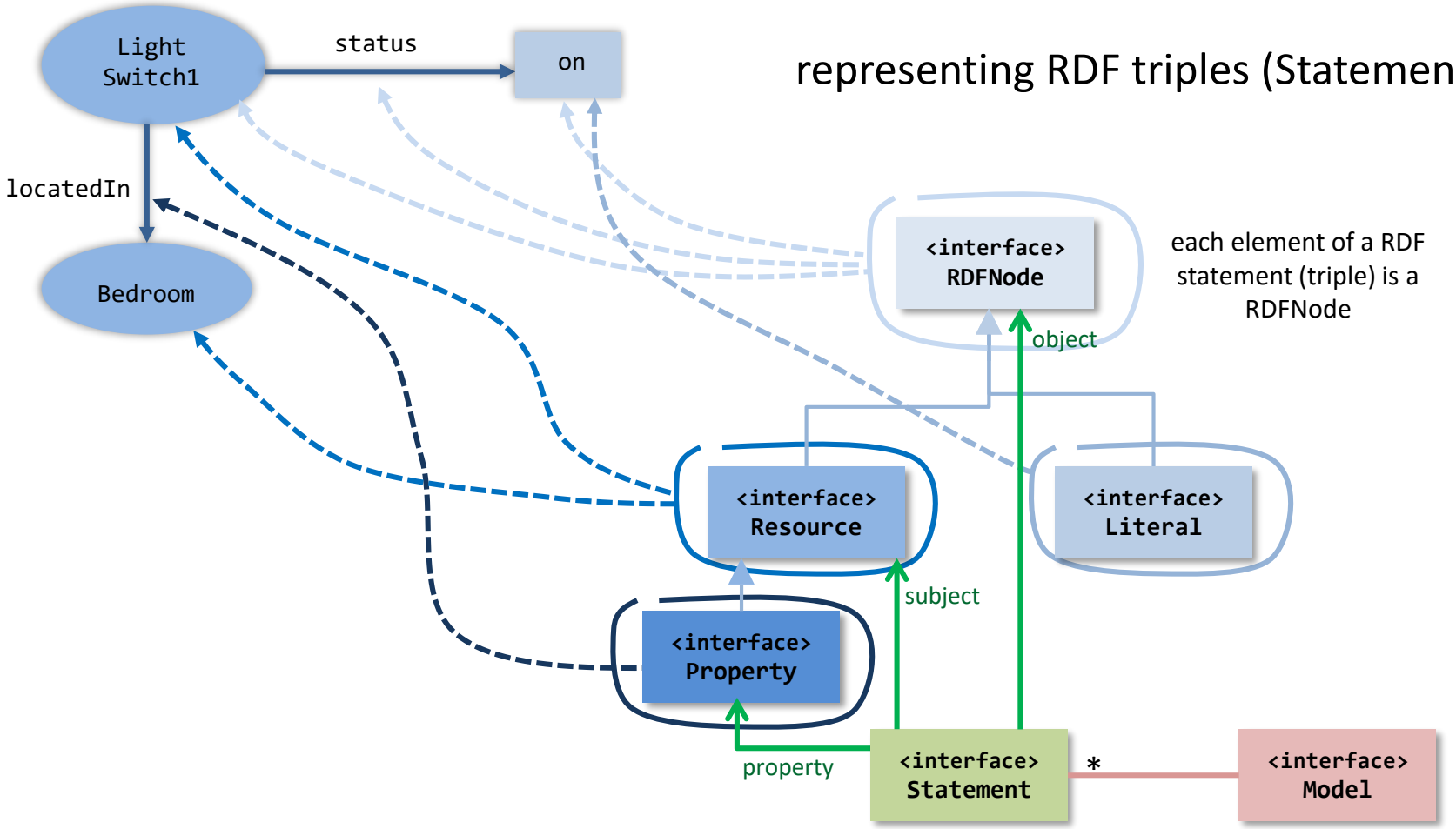
Core RDF API
package org.apache.jena.rdf.model

- interface **Model**
  - represents a RDF Graph
  - a model is a set of **Statements** (triples).
- methods are provided for
  - creating resources, properties and literals
  - creating the `Statements` (triples) which link them,
  - for adding statements to and removing them from a model,
  - for querying a model
  - set operations for combining models.

Core RDF API
package `org.apache.jena.rdf.model`

- Factory design pattern to create models
  - class **`ModelFactory`**
    - `static Model createDefaultModel()`
      - answer a new memory based Model (RDF Graph)
    - `static ModelMaker createFileModelMaker(String root)`
      - answer a `ModelMaker` that constructs memory-based Models that are backed by files
    - `static OntModel createOntologyModel()`
      - answer a new ontology model which will process in-memory models of ontologies expressed the default ontology language (OWL)
    - …

Core RDF API
package org.apache.jena.rdf.model



- **Satement** ⇔ < Resource , Property , Resource | Literal>

Core RDF API
package org.apache.jena.rdf.model

- `Model` methods to create a `Resource`
  - `Resource createResource(String uri)`

    A URI resource is .equals() to any other URI Resource with the same URI (even in a different model - be warned).
    This method may return an existing Resource with the correct URI and model, or it may construct a fresh one, as it sees fit.
  - `Resource createResource()`

    Create a new anonymous resource whose model is this model. This blank node will have a new AnonId distinct from any allocated by any other call of this method.
  - …
- `Model` methods to create a `Property`
  - `Property createProperty(String nameSpace, String localName)`

    Create a property with a given URI composed from a namespace part and a localname part by concatenating the strings.
    This method may return an existing property with the correct URI and model, or it may construct a fresh one, as it sees fit.

Core RDF API
package org.apache.jena.rdf.model

- `Model` methods to create a `Literal`
  - `Literal createLiteral(String v, String language)`
    Create an untyped literal from a String value with a specified language.
  - `Literal createTypedLiteral(String lex, RDFDatatype dtype)`
    Build a typed literal from its lexical form. The lexical form will be parsed now and the value stored. If the form is not legal this will throw an exception.
  - …

Core RDF API
package org.apache.jena.rdf.model

- `Model` method to create a `Statement`
  - `Statement createStatement(Resource s, Property p, RDFNode o)`

Creating a new Statement does not add it to the set of statements in the model

- `Model` methods to add it newly created `Statements`
  - `Model add(Statement stmt)`
  - `Model add(List<Statement> statements)`
  - `Model add(Statement[] statements)`
  - `Model add(StmtIterator iter)`

- `Statement` accessor methods
  - `Resource getSubject()`
  - `Property getPredicate()`
  - `RDFNode getObject()`

Core RDF API
package org.apache.jena.rdf.model

- `Resource` methods to create triples with the resource (this) as subject.
  - `Resource addProperty(…)`
    - `addProperty(Property p, RDFNode o)`
    - `addProperty(Property p, java.lang.String o)`
    - …
  - `Resource addLiteral(…)`
    - `addLiteral(Property p, Literal o)`
    - `addLiteral(Property p, double d)`
    - …

newly created Statements are automatically added to the set of statements in the model the Resource belongs to.

Core RDF API
package org.apache.jena.rdf.model

- **addProperty** and **addLiteral** methods return the **Resource** (this)

  → facilitates creation of multiple triples with the same subject

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

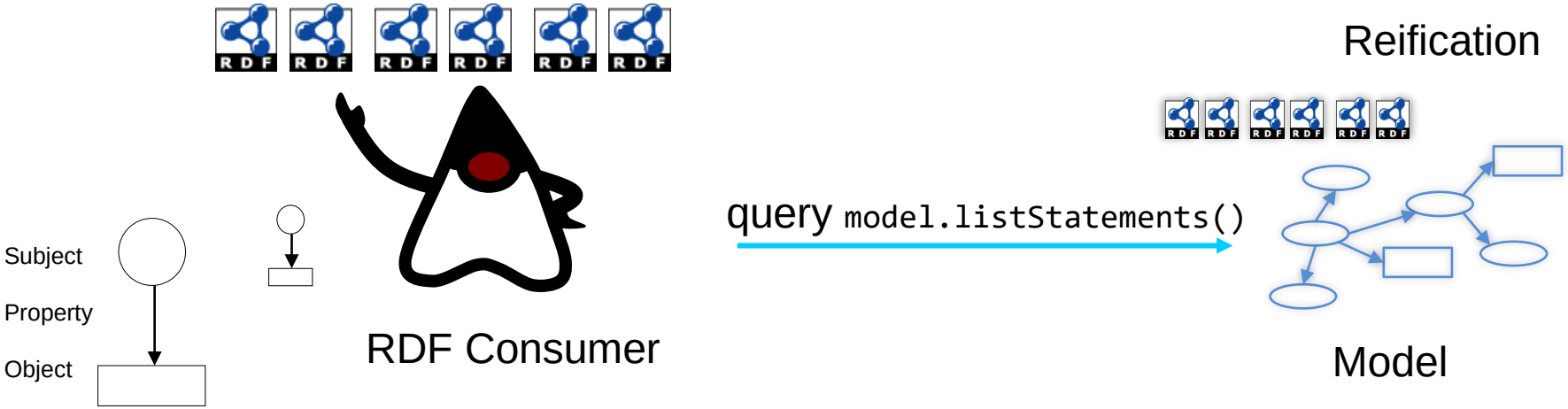
- vCard : a standard file format for electronic business cards
  - e.g. vCards are often attached to e-mail messages
  - http://www.w3.org/TR/vcard-rdf/ → RDF vocabulary corresponding to vCard

- **org.apache.jena.vocabulary** : package containing constant classes with predefined constant objects for classes and properties defined in well known vocabularies.
  - **org.apache.jena.vocabulary.VCARD** : vCard
    - static Property NAME
    - .
  - **org.apache.jena.vocabulary.RDF** : The standard RDF vocabulary
    - static Property type
    - …
  - **org.apache.jena.vocabulary.DC** : Dublin Core
    - …

Core RDF API
package org.apache.jena.rdf.model

- `StatementIterator` to access statements asserted in a RDF Model
  - `Model` methods to get a `StatementIterator`
    - `StmtIterator listStatements()`
      List all statements of the model
    - `StmtIterator listStatements(Resource s, Property p, RDFNode o)`
      List all the statements matching a pattern

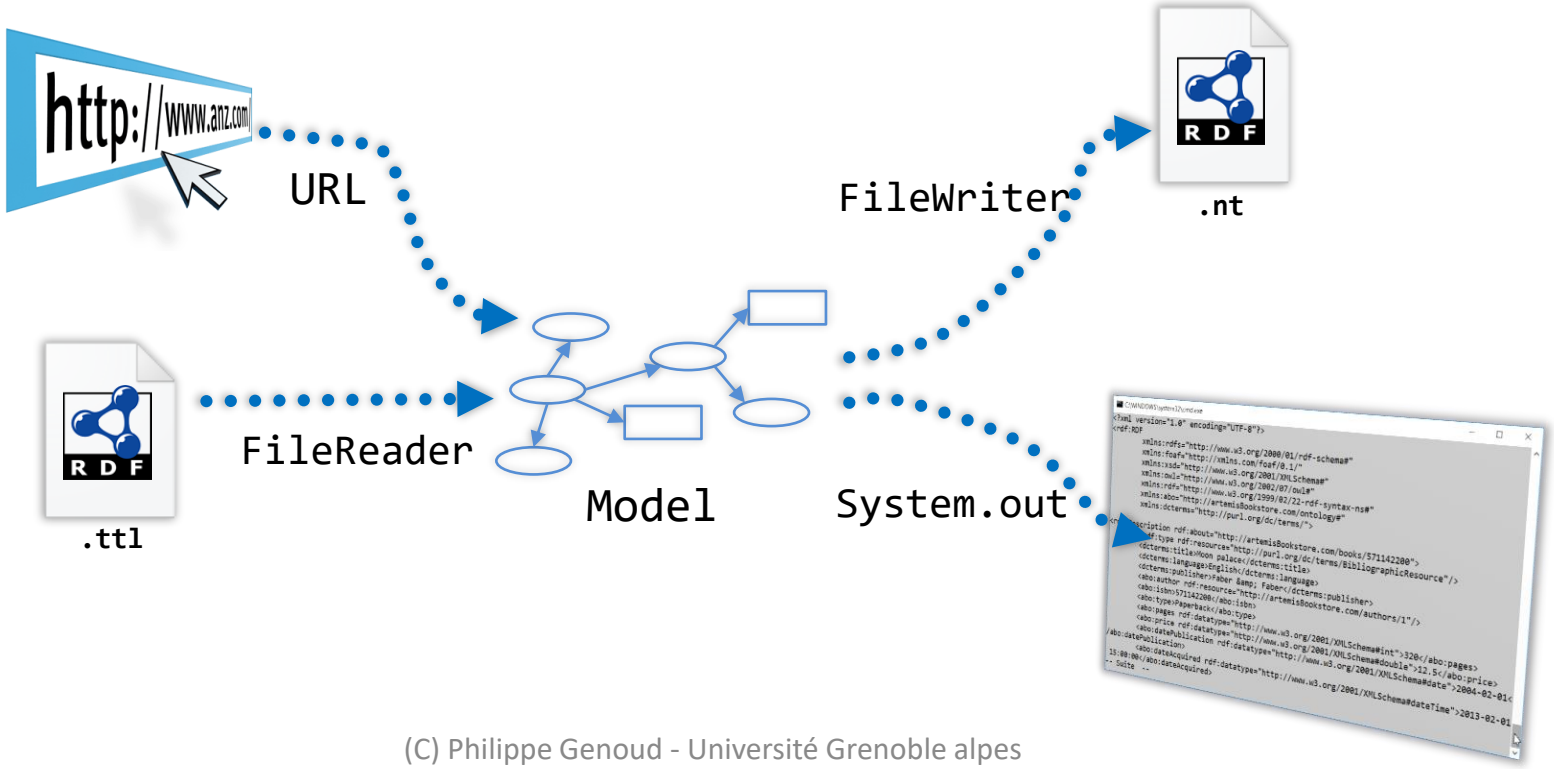
```
for(StatementIterator it = model.listStatements();it.hasNext();) {
    Statement stmt = it.nextStatement();
   System.out.println(stmt.getSubject().getLocalName());
}
```



animation from [www.try.idv.tw/try/files/070517-**jena**.ppt](www.try.idv.tw/try/files/070517-jena.ppt) C.F.Liao (廖峻鋒)

Model I/O

- `read` and `write` methods of `Model` interface
  - e.g. `write(java.io.OutputStream out, java.lang.String lang, java.lang.String base)`
    - `lang` : output format "RDF/XML" (default), "RDF/XML-ABBREV" , "N3", "TURTLE", "N-TRIPLE"
    - `base` : The base uri for relative URI calculations. null means use only absolute URI's



Jena API for SPARQL

- **ARQ** : query engine for Jena that supports the SPARQL RDF Query language.

- javadoc
  - http://jena.apache.org/documentation/javadoc/arq/index.html

- `org.apache.jena.query` package for ARQ API

ARQ (SPARQL) API
package chh.jena.rdf.query

Key classes

- **Query** represents the application query.
  - **QueryFactory** : to create Query instances
- **QueryExecution** - represents one execution of a query.
  - **QueryExecutionFactory** - a place to get QueryExecution instances
- **Dataset**, a collection of named graphs and a default graph (background graph or unnamed graph).
  - **DatasetFactory** - a place to make datasets

ARQ (SPARQL) API
package chh.jena.rdf.query

Key classes

- For SELECT queries:
  - **QuerySolution** - A single solution to the query
  - **ResultSet** - An iterator over all the the QuerySolutions.
  - **ResultSetFormatter** : turns a ResultSet into various forms
    - text,
    - Model (an RDF graph)
    - plain XML
    - …

ARQ (SPARQL) API
package org.apache.jena.query

SELECT queries
① Preparing the request

```
import org.apache.jena.query.* ;

Model model = ... ;
String queryString = " ...SELECT... " ;
Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
```

import for ARQ classes
better to have explicit imports
`import org.apache.jena.query.Query`

construct the SPARQL query String

construct the SPARQL query String

produce an instance of `QueryExecution`

can be reduced to one step in some common cases:

```
QueryExecution qexec =
      QueryExecutionFactory.create(queryString, model);
```

ARQ (SPARQL) API

package org.apache.jena.query

SELECT queries

② Executing the request and exploiting the results

```
import com.hp.hpl.jena.query.* ;

Model model = ... ;

String queryString = " ...SELECT... " ;

Query query = QueryFactory.create(queryString) ;

QueryExecution qexec = QueryExecutionFactory.create(query, model) ;

try {
    ResultSet results = qexec.execSelect();

    for ( ; results.hasNext() ; ) {
      QuerySolution soln = results.nextSolution() ;
      RDFNode x = soln.get("varName") ;
      Resource r = soln.getResource("varR") ;
      Literal l = soln.getLiteral("varL") ;
    }
} finally {
   qexec.close() ;
}
```

it's an ARQ Resultset not a java.sql.ResultSet !

execute the query

Iteration loop to traverse `QuerySolutions`

`ResultSet` presents the results of a SELECT query in table-like manner.

Each row (`QuerySolution`) corresponds to a set of bindings which fulfill the conditions of the query.

SELECT queries

exploiting the results

```
for ( ; results.hasNext() ; ) {
   QuerySolution soln = results.nextSolution() ;
   RDFNode n = soln.get("varName") ;
   Resource r = soln.getResource("varR") ;
   Literal l = soln.getLiteral("varL") ;
}
```

Instead of a loop to deal with each row in the result set,
the application can call an operation of the `ResultSetFormatter`.

process results to produce a simple text presentation

```
ResultSetFormatter fmt = new ResultSetFormatter(results, query) ;
fmt.printAll(System.out) ;
```

or simply

```
ResultSetFormatter.out(System.out, results, query) ;
```

ARQ (SPARQL) API
package org.apache.jena.query

CONSTRUCT and DESCRIBE queries

- CONSTRUCT

```
import com.hp.hpl.jena.query.* ;

Model model = ... ;

String queryString = " ...CONSTRUCT... " ;

Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
Model resultModel = qexec.execConstruct() ;
qexec.close() ;
```

CONSTRUCT queries return a single RDF graph

- DESCRIBE

```
import com.hp.hpl.jena.query.* ;

Model model = ... ;

String queryString = " ...DESCRIBE... " ;

Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
Model resultModel = qexec.execDescribe() ;
qexec.close()
```

DESCRIBE queries return a single RDF graph

ASK queries

- ASK

```
import com.hp.hpl.jena.query.* ;

Model model = ... ;

String queryString = " ...ASK... " ;

Query query = QueryFactory.create(queryString) ;
QueryExecution qexec = QueryExecutionFactory.create(query, model) ;
boolean result = qexec.execAsk() ;
qexec.close() ;
```

ASK queries return a boolean

Datasets

- A SPARQL query is made on a **dataset** : a default graph and zero or more named graph

```
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) ;
QueryExecution qExec = QueryExecutionFactory.create(query, dataset) ;
try {
   ...
}
finally {
   qExec.close() ;
}
```

- Already existing models can also be used

```
DataSet dataSource = DatsetFactory.createMem() ;// Creates an in-memory, modifiable Dataset
dataSource.setDefaultModel(model) ;
dataSource.addNamedModel("http://example/named-1", modelX) ;
dataSource.addNamedModel("http://example/named-2", modelY) ;
QueryExecution qExec = QueryExecutionFactory.create(query, dataSource) ;
```

Remote Query

- Create a QueryExecution that will access a SPARQL service over HTTP

```
String queryStr = "select distinct ?Concept where {[] a ?Concept} LIMIT 10";
Query query = QueryFactory.create(queryStr);

// Remote execution.
QueryExecution qexec =
QueryExecutionFactory.sparqlService("http://dbpedia.org/sparql", query);
// Set the DBpedia specific timeout.
((QueryEngineHTTP)qexec).addParam("timeout", "10000") ;

// Execute.
ResultSet rs = qexec.execSelect();
ResultSetFormatter.out(System.out, rs, query);
qexec.close();
```

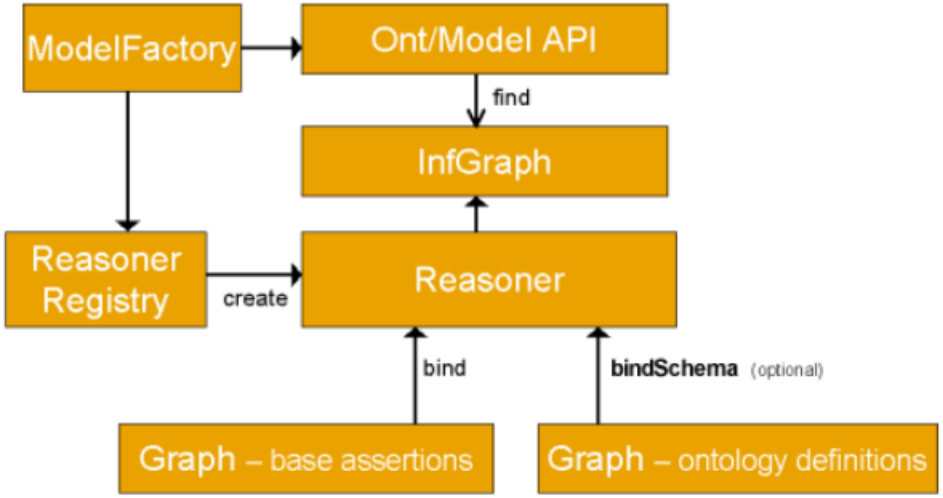
RDFConnection SPARQL API

-
  - https://jena.apache.org/documentation/rdfconnection/

Jena inferences

- Jena predefined reasoners:
  - **Transitive reasoner**: implements just the transitive and reflexive properties of rdfs:subPropertyOf and rdfs:subClassOf.
  - **RDFS rule reasoner**: Implements a configurable subset of the RDFS entailments.
  - OWL, OWL Mini, OWL Micro Reasoners: A set of useful but incomplete implementation of the OWL/Lite subset of the OWL/Full language.
  - **Generic rule reasoner:** A rule based reasoner that supports user defined rules. Forward chaining, tabled backward chaining and hybrid execution strategies are supported.
- possibility to connect to external reasoners
  - Pellet, Fact++, RacerPro …

Jena inferences : org.apcahe.jena.reasoner



example: SKOS inferences

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

InputStream in = new FileInputStream(new File("mySkosVoc.ttl"));
model.read(in,"Turtle");
model.read("http://www.w3.org/2004/02/skos/core#");


Reasoner reasoner = ReasonerRegistry.getOWLMicroReasoner();
InfModel infModel = ModelFactory.createInfModel(reasoner, model);

OutputStream out =  new FileOutputStream(new File("mySkosInferred.ttl"));

infModel.write(out, "Turtle");
```

Jena inferences : command line tools

- possibility to perform RDFS inferences with riot command line tool

- ex:

```
turtle --rdfs=foaf.rdf –formatted=turtle writers.ttl
            > writersWithInferences.ttl
```