Introduction to Jena

Use RDF models in your Java applications with the Jena Semantic Web Framework

Philip McCarthy

June 23, 2004

The Resource Description Framework (RDF) recently became a W3C recommendation, taking its place alongside other Web standards such as XML and SOAP. RDF can be applied in fields that deal with ad-hoc incoming data, such as CRM, and is already being widely used in social networking and self-publishing software like LiveJournal and TypePad.

Java programmers will increasingly benefit from having the skills to work with RDF models. In this article, I'll take you through some of the features of HP Labs' open source Jena Semantic Web Framework (see Related topics). You'll learn how to create and populate RDF models, how to persist them to a database, and how to query them programmatically using the RDQL query language. Finally, I'll demonstrate how Jena's reasoning capabilities can be used to infer knowledge about models from an ontology.

This article assumes you have some familiarity with RDF -- in terms of concepts such as graphs, triples, and schemas -- as well as a basic knowledge of Java programming.

Creating a simple RDF model

Let's start with the basics: creating a model from scratch and adding RDF statements to it. For this section, I'll show how to create a model describing the relationships between a group of fictional family members, as illustrated in Figure 1:

Figure 1. A fictional family tree

adam + dotty ——> beth + chuck
|                   |
| edward            | fran + greg
|                   |
| harriet
You'll describe the different relationship types using the properties siblingOf, spouseOf, parentOf, and childOf, taken from the "Relationship" vocabulary (see Related topics). For simplicity, the family members are identified with URIs from a made-up namespace, http://family/. Vocabulary URIs are frequently used in Jena code, so it's useful to declare them as Java constants, reducing the risk of mistyping them.

Schemagen

When you're working with a model using Jena's API, it's useful to define constants for each property in the model's vocabulary. If you have an RDFS, a DAML, or an OWL representation of the vocabulary, Jena's Schemagen tool can make your life easier by generating these constants automatically.

Schemagen is run on the command line, and takes parameters including the location of the schema or ontology file, and the name and Java package of the class to output. The generated Java class can then be imported and its Property constants used to access the model.

It's also possible to run Schemagen as part of a build process with Ant, keeping the Java constants class in sync with an evolving vocabulary.

Jena's ModelFactory class is the preferred means of creating different types of models. In this case, you want an empty, in-memory model, so ModelFactory.createDefaultModel() is the method to call. This method returns an instance of Model, which you'll use to create a Resource representing each person in the family. After the resources are created, statements can be made about them and added the model.

In Jena, the subject of a statement is always a Resource, the predicate is represented by a Property, and the object is either another Resource or a literal value. Literals are represented in Jena by the Literal type. All of these types share a common interface, RDFNode. You'll need four different Property instances to represent the relationships in the family tree. These instances are created with Model.createProperty().

The simplest way to add statements to a model is by calling Resource.addProperty(). This method creates a statement in the model with the Resource as its subject. The method takes two parameters, a Property representing the predicate of the statement, and the statement's object. The addProperty() method is overloaded: One overload takes an RDFNode as the object, so a Resource or Literal can be used. There are also convenience overloads that take a literal represented by a Java primitive or a String. In the example, the objects of the statements are Resources representing other family members.

Statements can also be created directly on the model by calling Model.createStatement() with the subject, predicate, and object of the triple. Note that creating a Statement in this way doesn't add it into the model. If you want to add it into the model, call Model.add() with the created Statement, as shown in Listing 1:

Listing 1. Creating a model to represent the fictional family

```java
// URI declarations
String familyUri = "http://family/";
String relationshipUri = "http://purl.org/vocab/relationship/";
```
The full code example, FamilyModel.java, also demonstrates how batches of statements can be added to the model at once, either as an array or a java.util.List.

With the family model built, let's see how information can be extracted from it using Jena's query API.

**Interrogating an RDF model**

Interrogating a Jena model programmatically is primarily performed through list() methods on the Model and Resource interfaces. These methods can be used to obtain subjects, objects, and Statements matching certain conditions. They also return specialisations of java.util.Iterator, which have extra methods to return specific object types.

Let's return to the family model example from Listing 1 and look at different ways in which you can interrogate it, as shown in Listing 2:

**Listing 2. Querying the family model**

```java
// List everyone in the model who has a child:
ResIterator parents = model.listSubjectsWithProperty(parentOf);

// Because subjects of statements are Resources, the method returned a ResIterator
while (parents.hasNext()) {
    // ResIterator has a typed nextResource() method
    Resource person = parents.nextResource();

    // Print the URI of the resource
    System.out.println(person.getURI());
}

// Can also find all the parents by getting the objects of all "childOf" statements
// Objects of statements could be Resources or literals, so the Iterator returned
```
// contains RDFNodes
NodeIterator moreParents = model.listObjectsOfProperty(childOf);

// To find all the siblings of a specific person, the model itself can be queried
NodeIterator siblings = model.listObjectsOfProperty(edward, siblingOf);

// But it's more elegant to ask the Resource directly
// This method yields an iterator over Statements
StmtIterator moreSiblings = edward.listProperties(siblingOf);

The most generic query method, which underlies the convenience methods demonstrated here, is Model.listStatements(Resource s, Property p, RDFNode o). Any of these parameters can be left null, in which case they act as wildcards, matching anything. Some examples of Model.listStatements() usage are shown in Listing 3:

Listing 3. Using Selectors to query a model

// Find the exact statement "adam is a spouse of dotty"
model.listStatements(adam, spouseOf, dotty);

// Find all statements with adam as the subject and dotty as the object
model.listStatements(adam, null, dotty);

// Find any statements made about adam
model.listStatements(adam, null, null);

// Find any statement with the siblingOf property
model.listStatements(null, siblingOf, null);

Importing and persisting models

Not all applications will start with an empty model. More commonly, a model would be populated from existing data at startup. A disadvantage of using an in-memory model in this situation is that the model would have to be repopulated from scratch each time the application was launched. Additionally, any changes made to an in-memory model would be lost each time the application was shut down.

One solution is to use Model.write() to serialise the model to the filesystem, and Model.read() to deserialise it on startup. However, Jena also offers persistent models, which are continually and transparently persisted to a backing store. Jena can persist its models on the filesystem, or in a relational database. The database engines currently supported are PostgreSQL, Oracle, and MySQL.

WordNet

WordNet is “a lexical database for the English language.” The RDF representation I’m using is by Sergey Melnik and Stefan Decker. It comes as four separate models, and you’ll be using three of them for the examples in this article.

The WordNet-nouns model contains all of the “lexical concepts” represented by WordNet, and the “word forms” used to represent each. For example, it contains a lexical concept represented by the word forms “domestic dog,” “dog,” and “Canis familiaris.”

The second model is WordNet-glossary. It provides a brief definition of each lexical concept in the model. The lexical concept for “dog” has a glossary entry stating “a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times.”
WordNet-hyponyms is the third model. It defines a hierarchy of concepts in the model. The concept "dog" is a hyponym of the concept "canine," which is itself a hyponym of the concept "carnivore."

To demonstrate how to import and persist a model, I'm going to import into MySQL an RDF representation of the WordNet 1.6 database. Because the WordNet representation I'm using takes the form of several separate RDF documents, importing them into a single Jena model merges their statements. Figure 2 demonstrates the structure of a fragment of the WordNet model after merging the Nouns and Glossary models together:

**Figure 2. Structure of merged WordNet nouns and glossary models**

The first steps in creating a database-backed model are to instantiate the MySQL driver class, and create a `DBConnection` instance. The `DBConnection` constructor takes the ID and password of the user to log in to the database as. It also takes a database URL parameter, which contains the name of the MySQL database for Jena to use, in the form `jdbc:mysql://localhost/dbname`. Jena can create multiple models within a single database. The final `DBConnection` parameter is the database type, which for MySQL is "MySQL."

The `DBConnection` instance can then be used with Jena's `ModelFactory` to create the database-backed model.

Once the model has been created, the WordNet RDF documents can be read in from the filesystem. Various `Model.read()` methods can populate the model from a `Reader`, an `InputStream`, or a `URL`. Models can be parsed from Notation3, N-Triples, or, by default, RDF/XML syntax. The WordNet model is serialised as RDF/XML, so you don't need to specify the syntax. When reading models, a base URI can be provided. The base URI is used to convert any relative URIs in the model into absolute URIs. Because the WordNet documents don't contain any relative URIs, this parameter can be given as `null`.

Listing 4 shows the complete process of importing a WordNet RDF/XML file into a MySQL-persisted model:

**Listing 4. Importing and persisting the WordNet models**
```java
// Instantiate the MySQL driver
```
Class.forName("com.mysql.jdbc.Driver");

// Create a database connection object
DBConnection connection = new DBConnection(DB_URL, DB_USER, DB_PASSWORD, DB_TYPE);

// Get a ModelMaker for database-backed models
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);

// Create a new model named "wordnet." Setting the second parameter to "true" causes an
// AlreadyExistsException to be thrown if the db already has a model with this name
Model wordnetModel = maker.createModel("wordnet",true);

// Start a database transaction. Without one, each statement will be auto-committed
// as it is added, which slows down the model import significantly.
model.begin();

// For each wordnet model . . .
InputStream in = this.getClass().getClassLoader().getResourceAsStream(filename);
model.read(in,null);

// Commit the database transaction
model.commit();

Now that the wordnet model is populated, you can access it later with a call to
ModelMaker.openModel("wordnet",true);

Querying a model as large and as rich as WordNet using Jena's API alone would be restrictive,
as each type of query to be performed would require several lines of bespoke code to be written.
Fortunately, Jena provides a mechanism for expressing generic queries, in the form of RDQL.

RDF Data Query Language (RDQL)

RDQL is a query language for RDF. While not yet a formal standard, RDQL is widely implemented
by RDF frameworks. RDQL allows complex queries to be expressed concisely, with a query engine
performing the hard work of accessing the data model. RDQL's syntax superficially resembles that
of SQL, and indeed, some of its concepts will be familiar to anyone who has worked with relational
database queries. An excellent RDQL tutorial can be found on the Jena Web site, but a few brief
examples should go a long way toward illustrating the basics.

RDQL queries can be executed on the command line against a Jena model using the
jena.rdfquery tool. RDFQuery takes an RDQL query from a text file and runs it against a specified
model. Quite a few parameters are required to run against a model that is database-backed. The
full command line required to run the following examples is shown in Listing 5:

Listing 5. Running RDQL queries from the command line

```
$java jena.rdfquery --data jdbc:mysql://localhost/jena --user dbuser --password dbpass
--driver com.mysql.jdbc.Driver --dbType MySQL --dbName wordnet --query example_query.rdql
```

As you can see, most of these parameters provide the necessary details to create a connection to
MySQL. The important part is --query example_query.rdql, which is the location of the RDQL file.
Also note that all of the JAR files in Jena's lib directory are needed to run jena.rdfquery.

Listing 6 shows the first query you'll examine:
**Listing 6. RDQL query to find the WordNet glossary entry for "domestic dog"**

```rdql
SELECT ?definition
WHERE (?concept, <wn:wordForm>, "domestic dog"), (?concept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

The **SELECT** part declares the variables to be output by the query -- in this case, the variable named **definition**. The **WHERE** clause introduces a second variable, **concept**, and defines triples that are matched against the graph. The query finds statements in the graph for which all of the triples in the **WHERE** clause hold. So, in English, the **WHERE** clause reads something like "find concepts that have a 'domestic dog' as a wordform, and find glossary entries for those concepts," as illustrated in Figure 3. The **USING** clause is a convenience, used to declare prefixes for namespaces.

**Figure 3. The graph matched by the WHERE clause in Listing 6**

Running the query results in:

```
?definition
===============================================================================
"a member of the genus Canis (probably descended from the common wolf) that has been domesticated by man since prehistoric times; occurs in many breeds; "the dog barked all night"
```

So there's only one result in this case. The next query, shown in Listing 7, says "find concepts represented by the word 'bear' and find the glossary entries for those concepts."

**Listing 7. RDQL query to find the WordNet glossary entries for "bear"**

```rdql
SELECT ?definition
WHERE (?concept, <wn:wordForm>, "bear"), (?concept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

This query returns 15 results, because this wordform represents several different concepts. The results begin:
"massive plantigrade carnivorous or omnivorous mammals with long shaggy coats and strong claws"
"an investor with a pessimistic market outlook"
"have on one's person; "He wore a red ribbon"; "bear a scar"
"give birth (to a newborn); "My wife had twins yesterday!"

A more involved example, shown in Listing 8, finds the hypernym (parent word) of two others:

**Listing 8. RDQL query to find the WordNet hypernym of "panther" and "tiger"

```rdql
SELECT ?wordform, ?definition
WHERE
  (?firstconcept, <wn:wordForm>, "panther"),
  (?secondconcept, <wn:wordForm>, "tiger"),
  (?firstconcept, <wn:hyponymOf>, ?hypernym),
  (?secondconcept, <wn:hyponymOf>, ?hypernym),
  (?hypernym, <wn:wordForm>, ?wordform),
  (?hypernym, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

Here, the query says "find the concepts referred to by the words 'panther' and 'tiger,' find a third concept of which the first two are hyponyms, and find the possible words and the glossary entry for that concept," as illustrated in Figure 4:

**Figure 4. The graph matched by the WHERE clause in Listing 8**

Both `wordform` and `definition` are declared in the `SELECT` clause, so both are output. Although this query only matches a single WordNet concept, the query's graph can be matched in two ways, because the concept has two different wordforms:

<table>
<thead>
<tr>
<th>wordform</th>
<th>definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;big cat&quot;</td>
<td>&quot;any of several large cats typically able to roar and living in the wild&quot;</td>
</tr>
<tr>
<td>&quot;cat&quot;</td>
<td>&quot;any of several large cats typically able to roar and living in the wild&quot;</td>
</tr>
</tbody>
</table>

**Using RDQL in Jena**

Jena's `com.hp.hpl.jena.rdql` package contains all of the classes and interfaces needed to use RDQL in your Java code. To create an RDQL query, put the RDQL in a `String`, and pass it to the constructor of `Query`. It's usual to explicitly set the model to use as the source for the query, unless
otherwise specified with a \texttt{FROM} clause in the RDQL itself. Once a \texttt{Query} is prepared, a \texttt{QueryEngine} can be created from it, and the query executed. This process is demonstrated in Listing 9:

**Listing 9. Creating and running an RDQL query**

```java
// Create a new query passing a String containing the RDQL to execute
Query query = new Query(queryString);
// Set the model to run the query against
query.setSource(model);
// Use the query to create a query engine
QueryEngine qe = new QueryEngine(query);
// Use the query engine to execute the query
QueryResults results = qe.exec();
```

A useful technique to use with a \texttt{Query} is to set some of its variables to fixed values prior to execution. This usage pattern is similar to that of \texttt{javax.sql.PreparedStatement}. Variables are bound to values through a \texttt{ResultBinding} object, which is passed to the \texttt{QueryEngine} at execution time. It's possible to bind variables to either Jena Resources or to literal values. Wrap a literal with a call to \texttt{Model.createLiteral} before binding it to a variable. Listing 10 illustrates the prebinding approach:

**Listing 10. Binding query variables to values**

```java
// Create a query that has variables x and y
Query query = new Query(queryString);
// A ResultBinding specifies mappings between query variables and values
ResultBinding initialBinding = new ResultBinding();
// Bind the query's first variable to a resource
Resource someResource = getSomeResource();
initialBinding.add("x", someResource);
// Bind the query's second variable to a literal value
RDFNode foo = model.createLiteral("bar");
initialBinding.add("y", foo);
// Execute the query with the specified values for x and y
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```

The \texttt{QueryResults} object returned by \texttt{QueryEngine.exec()} implements \texttt{java.util.Iterator}. Its \texttt{next()} method returns \texttt{ResultBinding} objects. All of the variables used in the query can be obtained from the \texttt{ResultBinding} by name, regardless of whether they were part of the \texttt{SELECT} clause. Listing 11 shows how to do this, again using the RDQL query in Listing 6:

**Listing 11. RDQL query to find the WordNet glossary entry for "domestic dog"**

```
SELECT ?definition
WHERE (?concept, <wn:wordForm>, "domestic dog"),
      (?concept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```
The ResultBinding obtained from running this query contains the literal glossary entry for the word, as expected. Additionally, you can access the variable concept. Variables are obtained by name by calling ResultBinding.get(). All variables returned by this method can be cast to RDFNode, which is useful if you want to bind them back into a further RDQL query.

In this case, the concept variable represents an RDF resource, so the object obtained from ResultBinding.get() can be cast to Resource. The query methods of the Resource can then be called to further explore this part of the model, as shown in Listing 12:

**Listing 12. Working with query results**

```java
// Execute a query
QueryResults results = qe.exec();

// Loop over the results
while (results.hasNext()) {
    ResultBinding binding = (ResultBinding)results.next();

    // Print the literal value of the "definition" variable
    RDFNode definition = (RDFNode) binding.get("definition");
    System.out.println(definition.toString());

    // Get the RDF resource used in the query
    Resource concept = (Resource)binding.get("concept");

    // Query the concept directly to find other wordforms it has
    List wordforms = concept.listObjectsOfProperty(wordForm);
}
```

The program FindHypernym.java included in source download (see Related topics) summarises the areas you've looked at here. It finds hypernyms of a word given on the command line, using the query shown in Listing 13:

**Listing 13. RDQL query to find the wordforms and glossary entries of a concept's hypernyms**

```rdql
SELECT ?hypernym, ?definition
WHERE
    (?firstconcept, <wn:wordForm>, ?hyponym),
    (?firstconcept, <wn:hyponymOf>, ?secondconcept),
    (?secondconcept, <wn:wordForm>, ?hypernym),
    (?secondconcept, <wn:glossaryEntry>, ?definition)
USING wn FOR <http://www.cogsci.princeton.edu/~wn/schema/>
```

The word given on the command line is bound to the hyponym term, and the query finds the concept that that word represents, finds a second concept that the first concept is a hyponym of, and then outputs that concept's wordform and definition. Listing 14 shows what its output looks like:
Listing 14. Running the example FindHypernym program

$ java FindHypernym "wisteria"

Hypernyms found for 'wisteria':

vine: weak-stemmed plant that derives support from climbing, twining, or creeping along a surface

Adding meaning with OWL

You may be wondering why a search for hypernyms of "wisteria" only returns its immediate hypernym, "vine." If you are botanically minded, you might also expect "traceophyte" to show up as a hypernym, and of course "plant." Indeed, the WordNet model says that "wisteria" is a hyponym of "vine," and "vine" is a hyponym of "traceophyte." Intuitively, you know that "wisteria" is therefore a hyponym of "traceophyte," because you know that the "hyponym of" relationship is transitive. So you need a way to incorporate this knowledge into your FindHypernym program, which is where OWL comes in.

### Transitive relationships

A relationship is transitive for three elements -- $a$, $b$, and $c$ -- such that the existence of the relationship between $a$ and $b$, and between $b$ and $c$, implies the existence of the relationship between $a$ and $c$.

An example of a transitive relationship is "greater than": If $a$ is greater than $b$, and $b$ is greater than $c$, then it follows that $a$ must be greater than $c$.

Web Ontology Language, or OWL, is a W3C Recommendation intended to "explicitly represent the meaning of terms in vocabularies and the relationships between those terms." Together with RDF Schema, OWL provides a mechanism to formally describe the content of RDF models. In addition to defining hierarchical classes that resources can belong to, OWL allows the characteristics of resources' properties to be expressed. For instance, in the Relationship vocabulary used in Listing 1, OWL could be used to state that the `childOf` property is the inverse of the `parentOf` property. Another example is to state that the WordNet vocabulary's `hyponymOf` property is transitive.

In Jena, an ontology is treated as a special type of RDF model, `OntModel`. This interface allows the ontology to be manipulated programmatically, with convenience methods to create classes, property restrictions, and so forth. An alternative approach is to treat the ontology as a regular RDF model, and simply add statements defining its semantic rules. Both of these techniques are demonstrated in Listing 15. Note that it's also possible to add ontological statements to an existing data model, or merge an ontology model with a data model using `Model.union()`.

Listing 15. Creating an OWL ontology model for WordNet

```java
// Make a new model to act as an OWL ontology for WordNet
OntModel wnOntology = ModelFactory.createOntologyModel();

// Use OntModel's convenience method to describe
// WordNet's hyponymOf property as transitive
wnOntology.createTransitiveProperty(WordnetVocab.hyponymOf.getURI());

// Alternatively, just add a statement to the underlying model to express that
// hyponymOf is of type TransitiveProperty
wnOntology.add(WordnetVocab.hyponymOf, RDF.type, OWL.TransitiveProperty);
```
Inference with Jena

Given an ontology and a model, Jena's inference engine can derive additional statements that the model doesn't express explicitly. Jena provides several Reasoner types to work with different types of ontology. Because you want to use the OWL ontology with the WordNet model, an OWLReasoner is needed.

The following example shows how to apply your OWL WordNet ontology to the WordNet model itself to create an inference model. I'm actually going to use a subset of the WordNet model here, containing only those nouns beneath "plant life" in the hyponym hierarchy. The reason for using only a subset is that the inference model needs to be held in memory, and the WordNet model is too large for an in-memory model to be practical. The code I used to extract the plants model from the full WordNet model is included in the article source, and is named ExtractPlants.java (see Related topics).

First, I'll get an OWLReasoner from the ReasonerRegistry. ReasonerRegistry.getOWLReasoner() returns an OWL reasoner in its standard configuration, which is fine for this simple case. The next step is to bind the reasoner to the WordNet ontology. This operation returns a reasoner ready to apply the ontology's rules. Next I'll use the bound reasoner to create an InfModel from the WordNet model.

Having created an inference model from the original data and the OWL ontology, it can be treated just like any other Model instance. Therefore, as Listing 16 shows, the Java code and RDQL query used with a regular Jena model by FindHypernym.java can be re-applied to the inference model without any changes:

**Listing 16. Creating and querying an inference model**

```java
// Get a reference to the WordNet plants model
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
Model model = maker.openModel("wordnet-plants",true);

// Create an OWL reasoner
Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();

// Bind the reasoner to the WordNet ontology model
Reasoner wnReasoner = owlReasoner.bindSchema(wnOntology);

// Use the reasoner to create an inference model
InfModel infModel = ModelFactory.createInfModel(wnReasoner, model);

// Set the inference model as the source of the query
query.setSource(infModel);

// Execute the query as normal
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```

The full listing is available in the article source, named FindInferredHypernyms.java. Listing 17 shows what happens when the inference model is queried for hypernyms of "wisteria":

```java
// Get a reference to the WordNet plants model
ModelMaker maker = ModelFactory.createModelRDBMaker(connection);
Model model = maker.openModel("wordnet-plants",true);

// Create an OWL reasoner
Reasoner owlReasoner = ReasonerRegistry.getOWLReasoner();

// Bind the reasoner to the WordNet ontology model
Reasoner wnReasoner = owlReasoner.bindSchema(wnOntology);

// Use the reasoner to create an inference model
InfModel infModel = ModelFactory.createInfModel(wnReasoner, model);

// Set the inference model as the source of the query
query.setSource(infModel);

// Execute the query as normal
QueryEngine qe = new QueryEngine(query);
QueryResults results = qe.exec(initialBinding);
```
Listing 17. Running the example FindInferredHypernyms program

$ java FindInferredHypernyms wisteria

Hypernyms found for 'wisteria':

vine: weak-stemmed plant that derives support from climbing, twining, or creeping along a surface
tracheophyte: green plant having a vascular system: ferns, gymnosperms, angiosperms
vascular plant: green plant having a vascular system: ferns, gymnosperms, angiosperms
plant life: a living organism lacking the power of locomotion
flora: a living organism lacking the power of locomotion
plant: a living organism lacking the power of locomotion

The information contained in the OWL ontology has allowed Jena to infer that "wisteria" has hypernyms right up through the model.

Summary

This article has demonstrated some of the most important features of the Jena Semantic Web Toolkit, with examples showing how to create, import, and persist RDF models. You've also looked at different methods of querying a model and seen how RDQL can be used to concisely express arbitrary queries. Additionally, you've seen how Jena's inference engine can be used to make inferences about a model based on an ontology.

The examples in this article have demonstrated some of the power of representing data as RDF models, and the flexibility of RDQL in extracting data from them. The basic approaches explained here will be a useful starting point when employing RDF models in your own Java applications.

Jena is a comprehensive RDF toolset, and its abilities extend beyond what you've learned here. The Jena project's homepage is a great place to start learning more about what it can do.
### Downloadable resources

<table>
<thead>
<tr>
<th>Description</th>
<th>Name</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>j-jena-examples.zip</td>
<td>8KB</td>
</tr>
</tbody>
</table>
Related topics

- [jena.sourceforge.net](http://jena.sourceforge.net) is the home of the Jena project, which is distributed under a BSD-style license. You'll find a superb range of documentation here, and you can download the latest version of the Jena framework.
- The [Relationship](http://jena.sourceforge.net) vocabulary is useful when describing people's relationships to one another in RDF.
- [Jena-dev](http://jena.sourceforge.net/dev) is a mailing list for developers working with (and on) Jena. If you have a question about Jena, this is the place to get it answered.
- Hosted on the Jena site is "[A Programmer's Introduction to RDQL](http)," a comprehensive tutorial with lots of examples.
- You can obtain the WordNet RDF representation used in this article from [SemanticWeb.org](http://semanticweb.org). The RDF Schema for the representation is also available.
- Dave Beckett maintains an exhaustive collection of semantic Web and RDF links on his [RDF Resource Guide](http) page.
- Uche Ogbuji explored applications of RDF in his [Basic XML and RDF techniques for knowledge management series](http://developerWorks, July 2001).
- Shelley Power's book, [Practical RDF](http://O'Reilly, 2003) explores RDF from its basic concepts to real-world applications.
- The W3C's [OWL Overview](http) is the best place to get an understanding of what OWL can do.

© Copyright IBM Corporation 2004
Trademarks
(www.ibm.com/developerworks/ibm/trademarks/)