WICKED COOL JAVA

Code Bits, Open-Source Libraries, and Project Ideas

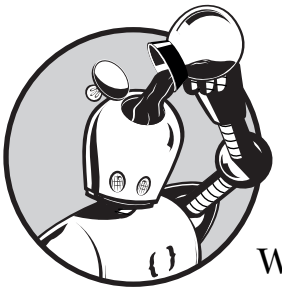
by Brian D. Eubanks

NO STARCH PRESS

San Francisco

4

CRAWLING THE SEMANTIC WEB



In this chapter, we examine techniques for extracting and processing data in the World Wide Web and the Semantic Web. The World Wide Web completely changed the way that people access information. Before the Web existed, finding obscure pieces of information meant taking a trip to the library, along with hours or perhaps days of research. In extreme cases, it meant calling or writing a letter to an expert and waiting for a reply. Today not only are there websites on every imaginable topic, but there are search engines, encyclopedias, dictionaries, maps, news, electronic books, and an incredible array of other data available online. Using search engines, we can find information on any topic within a few seconds. The Google search engine has even become so well known that it is now often used as a verb: "I Googled a solution." Online information is growing exponentially, and because of it we have a completely new problem on our hands that is not solved by simply using keyword searches to find our data. The problem is infoglut. Keyword searches return too many documents, and most of those documents don't have the information that we want.

Suppose that we wanted to search for a Java class library that converts data from one format to another. With all the open-source projects out there, someone may have already solved the problem for us, and we'd rather not reinvent the wheel. In theory, we should be able to search for matching projects that meet our needs. But running a query on related keywords may give us many results that are not related to what we really want. In an ideal world, we should be able to ask the computer a question: "Is there an open-source Java API that converts between FORMAT1 and FORMAT2?" The computer should then search the Web and give us the name of a suitable API if it exists, along with a short description of the standard and links to more detailed information. For this to happen, information about a hypothetical "J-convert-1-2" API would need to be encoded in such a way that the computer can find it easily without performing a keyword search and extracting data from the text results.

Information on the World Wide Web is mostly free-form text contained in HTML pages and is mostly not organized into categories and structures that search programs can easily query. At the very least, all web content ought to have subject indicators similar to the Library of Congress and Dewey Decimal codes for books. This is not yet the case, although it will most likely happen soon. Several new standards are rapidly leading us in that direction. So far, all of these standards rely on web content developers adding special tags to their data, and few developers know about these standards at the present time. In short, it's a mess out there, and we're trudging through this messy data looking for nuggets of gold.

The *Semantic Web* is the next-generation web of concepts linked to other concepts, rather than a collection of hypertext documents linked by keywords. If you think about it, an HTML anchor tag (link) is a keyword reference to another document. It supplies a word or phrase that links to another document, usually displayed as underlined text on a browser. But the link doesn't exactly say *how* the two documents are related to each other. HTML hyperlinks don't give any real indication about relationships between files, and the text in the link may be extremely vague. A new standard, the *Resource Description Framework (RDF)*, makes it possible to be much more specific about how things are related to each other. In fact, RDF describes much more than documents—any entities or concepts can be linked together. This is the basic idea behind the Semantic Web—that concepts, rather than documents, can be linked together.

As Java developers, how can we participate in building the Semantic Web? First, you'll need to know something about official standards such as RDF. You will then need to tag your documents appropriately. Many sites are already starting to do some of this by creating *RDF Site Summary (RSS)* feeds. An RSS feed syndicates the content from a website so that it can be combined with information from other sites and delivered to the users as *aggregated content*. RSS makes a small portion of a site available as a summary, similar to what you see in an article or news abstract. However, RSS enabling is only the first step in moving toward a Semantic Web. In this chapter we'll discuss enough to get you started working with RDF, and we'll introduce some APIs that help in producing or consuming content.

This *Somethings* That: A Short Introduction to N3 and Jena

JENA

The theory behind the RDF standard is actually quite simple. Everything has a Uniform Resource Identifier (URI), and by this I mean *everything*: not only documents but also generic concepts and relationships between them. Even though you are not a document (or are you?), there could be a URI assigned to represent you as an entity. This URI can then be used to make connections to other things. For the "you" URI, these connections might represent related organizations, addresses, and phone numbers. URIs do not have to return an actual document! This is what sometimes confuses developers when they see a URI referenced somewhere and find that there is nothing at the location. These addresses are often used as markers or unique identifiers to represent concepts. We make links between URIs to represent relationships between things. This functions much like a simple sentence in English:

```
Programmers enjoy Java.
```

To begin with, let's use a shorthand notation, called *N3*, to encode this as an RDF graph. N3 is an easy way to learn RDF because the syntax is only slightly more complex than the sentence above! In essence, N3 is merely a set of *triples*, or "subject predicate object" relationships. Here is the N3 version of the sentence:

```
@prefix wcj: <http://example.org/wcjava/uri/> .
wcj:programmers wcj:enjoy wcj:java .
```

We first define a prefix to make the N3 code less verbose. The prefix is used as the beginning part of a URI wherever it is found in the document, so that `wcj:java` then becomes `http://example.org/wcjava/uri/java` (the value is also placed within < and > markers—these have nothing to do with XML). The three items together are called a *triple*, and the verb is usually called a *predicate*. RDF makes a link by stating that a *subject URI* is related by a *predicate URI* to an *object URI*. The predicate represents some relationship between the subject and object—it tells *how* things link together. This is very different than an anchor in HTML, because here a relationship type is clearly defined. Remember that URIs in RDF could be anything: concepts, documents, or even (in some cases) `String` literals. In theoretical terms, we are creating a labeled directed graph of the relationship. A graph representation of the above might look like Figure 4-1.

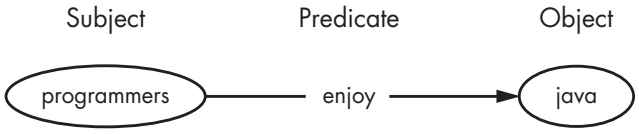


Figure 4-1: RDF subject, predicate, and object

As you might expect, there is a Java API for creating and managing RDF and N3 documents. *Jena* is an open-source API for working with RDF graphs. Here is one way to create the graph in Jena and serialize it to an N3 document:

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

Model model = ModelFactory.createDefaultModel();
Resource programmers =
   model.createResource("http://example.org/wcjava/uri/programmers");
Property enjoy =
   model.createProperty("http://example.org/wcjava/uri/enjoy");
Resource java =
   model.createResource("http://example.org/wcjava/uri/java");
model.add(programmers, enjoy, java);
FileOutputStream outStream = new FileOutputStream("out.n3");
model.write(outStream, "N3");
outStream.close();
```

Here, Jena is using the term *property* to refer to the predicate and *resource* to refer to something used as a subject or object. The model's `write` method also has options to write out the document in other formats besides N3. With the Jena API, you can connect many entities together into very large *semantic networks*. Let's make some additional relationships using the entities and relationships that we just created. We will produce the graph shown in Figure 4-2.

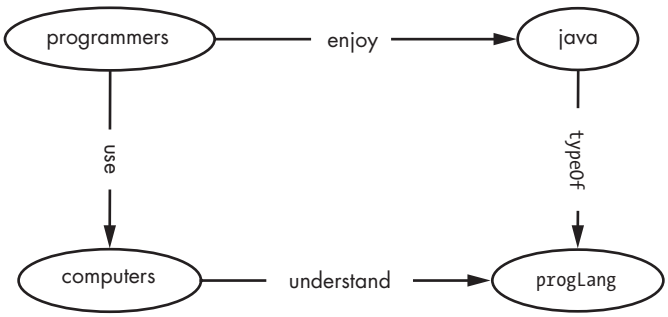


Figure 4-2: An RDF graph with multiple subjects

Here is the additional code to produce the network in Figure 4-2:

```
Property typeOf =
   model.createProperty("http://example.org/wcjava/typeOf");
Property use =
   model.createProperty("http://example.org/wcjava/use");
Property understand =
   model.createProperty("http://example.org/wcjava/understand");
```

```
Resource computers =
   model.createResource("http://example.org/wcjava/computers");
Resource progLang =
   model.createResource("http://example.org/wcjava/progLang");
model.add(java, typeOf, progLang);
model.add(programmers, use, computers);
model.add(computers, understand, progLang);
model.write(new java.io.FileOutputStream("out2.n3"), "N3");
```

The N3 output of this code is the following:

```
<http://example.org/wcjava/uri/java>
      <http://example.org/wcjava/typeOf>
              <http://example.org/wcjava/progLang> .

<http://example.org/wcjava/computers>
      <http://example.org/wcjava/understand>
              <http://example.org/wcjava/progLang> .

<http://example.org/wcjava/uri/programmers>
      <http://example.org/wcjava/uri/enjoy>
              <http://example.org/wcjava/uri/java> ;
      <http://example.org/wcjava/use>
              <http://example.org/wcjava/computers> .
```

The semicolon in the N3 document is a shortcut that indicates we are going to attach another property to the same subject ("programmers enjoy java, and programmers use computers"). The meanings of elements within a document are often defined in terms of a predefined set of resources and properties called a *vocabulary*. Your RDF data can be combined with other data in existing vocabularies to allow semantic searches and analysis of complex RDF graphs. In the next section, we illustrate how to build upon existing RDF vocabularies to build your own vocabulary.

Triple the Fun: Creating an RDF Vocabulary for Your Organization

An RDF graph creates a web of concepts. It makes assertions about logical relationships between entities. RDF was meant to fit into a dynamic knowledge representation system rather than a static database structure. Once you have information in RDF, it can be linked with graphs made elsewhere, and software can use this to make *inferences*. If you define how your own items are related in terms of higher-level concepts, your data can fit into a much larger web of concepts. This is the basis of the Semantic Web.

Every organization has relationships between information that is held in a data store such as a database or flat file (or human memory!). If your data is in a relational database, your data items probably have relationships between them that are hidden or implied within the database structure itself.

Your data may not be completely accessible, because there are relationships that an application cannot query. As an example, suppose that we have a relational database containing employees and departments within a company. A common approach is to create an `Employee` table, with columns for employee information such as ID number, date of birth, name, hire date, supervisor name, and department. There are many relationships hidden within the table and column names, and it is up to an application to know these relationships and take advantage of them. Column names alone would not give you the following information:

- A and B are employees.
- An employee is a person.
- A supervisor is an employee who directs another employee.
- C is a company.
- A company is an organization.
- A and B work for C.

Column and table names in a database are simply local identifiers and don't automatically map to any concepts that might be defined elsewhere. But this is domain knowledge that could be used more effectively by the application if it were defined in an extensible and machine-readable way. Having such information available would give our applications more flexibility, and this knowledge could also be reused elsewhere. How can we encode this information so that applications can make use of these relationships? And how can our application relate this to other information that we might find on the Semantic Web?

It may not make sense to put this metadata in your database, but you can create an RDF mapping outside the database schema that describes each item relative to the Semantic Web as a whole. We can represent some of these concepts using existing vocabularies. The rest of them we can define in our own terms. If you don't know where to connect a concept to an existing vocabulary, you can always define a URI for that concept now and make the connection to other systems later. At least you can use it to share data within your own organization if your vocabulary is well documented and the meaning of each item is clear. There are many basic vocabularies that RDF applications can use, and new ones are constantly being created (like yours!). The online resources page for this section has an updated listing of some existing vocabularies that you can use in defining your data.

The first step is to define a URI for each concept that is even remotely related to your application. This is much like the object-oriented development process, but these entities may also be things that are not directly used by the application. By defining your terms within a larger context, you can later map these entities to existing concepts on the Web. Let's try it with our employee example, by first listing some related concepts and their meanings (in English text). Here is a simplistic attempt to define some terms:

- http://example.org/wcjava/employee = an employee
- http://example.org/wcjava/person = a person

- http://example.org/wcjava/organization = an organization
- http://example.org/wcjava/employer = an organization that employs an employee

The important point is to make sure that each concept has a unique identifier. Make sure that the URIs will still be around a few years from now; you are building a complete concept space around these identifiers! If you have control over your domain name, it might be wise to have a policy that forbids anyone placing actual content under URIs beginning with some prefix (such as http://yourdomain/uri). We are using these names as globally unique identifiers, not as URLs for retrieving documents. There is nothing wrong with a document being there, but it could lead to confusion between the concept and the document. In this example, we are using the example.org domain, which is reserved solely for illustrative purposes within documentation. If you want to define a permanent URI, there are sites that will let you define your own permanent URI independent of future domain name ownership changes. (For more information on this, see this book's companion website.) The best known of these is http://purl.org.

After you have identified some concept URIs, it's time to define relationships between them. In the previous section, we showed how to do this in Jena using our own relationships. Now let's use some predefined relationships created by others and apply them to our entities. Adding another entity that was defined elsewhere is easy: just add its URI to the graph we are building. But if we want to do anything useful with these entities, we will also need to import the statements that define its related properties and resources. In our example, we will use the `subClassOf` property defined in the RDF schema, which works similarly to a subclass relationship in object-oriented programming. The graph in Figure 4-3 shows the relationships between our resources.

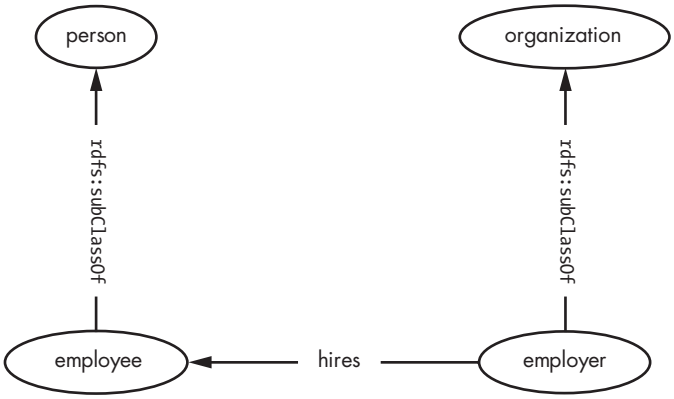


Figure 4-3: Using the `subClassOf` property from RDF schema

At first, you should do this mapping with pen and paper (archaic, but always accessible) or using an RDF visualization tool. This book's website has a list of some free tools that can be used for this purpose. When you have finished, you will have a graph of the relationships between entities in your system. Once you've created a hierarchy and vocabulary, you can create

N3 or RDF/XML files that you can use as metadata. Most RDF visualization tools will do this for you automatically. You'll want to familiarize yourself with some of the existing RDF vocabularies on which you can base your own hierarchy. Our resources page has links to some of these and examples of using them. Once you have designed a hierarchy, you can create and manipulate it from Jena. The next section shows how to do this.

Who's a What? Using RDF Hierarchies in Jena

JENA

Earlier we created a hierarchy of terms to use for our metadata. We used the word *vocabulary* to refer to this collection of terms, but it is often called an *ontology* if it defines relationships between the terms. According to the Wikipedia definition, an ontology (in the computer science sense) is a "data structure containing all the relevant entities and their relationships and rules (theorems, regulations) within a domain."

In Jena, there are built-in helper classes for working with commonly used ontologies. The RDF schema is one of these. Jena has a helper class called `RDFS`, which has a static variable for the `subClassOf` property. You can create the graph in the previous section by using this code:

```
Model model = ModelFactory.createDefaultModel();
model.setNsPrefix("wcj", "http://example.org/wcjava/");
Resource employee = model.createResource("wcj:employee");
Resource person = model.createResource("wcj:person");
Resource employer = model.createResource("wcj:employer");
Resource organization = model.createResource("wcj:organization");
Property hires = model.createProperty("wcj:hires");
model.add(employer, hires, employee);
model.add(employer, RDFS.subClassOf, organization);
model.add(employee, RDFS.subClassOf, person);
model.write(new FileWriter("ourEntities.rdf"), "RDF/XML");
```

The second line sets a namespace prefix for our graph, which makes the code easier to read because we can describe the URIs in a simpler way. There is nothing special about the choice of "wcj" as our prefix. It could have been any `String` of letters, but whichever value is used becomes the prefix that is sent to the output file. The RDF/XML output type is the XML representation of our RDF graph. Most applications will exchange RDF graphs using the XML format rather than N3. As you can see, Jena's RDF model can work with either type.

Once you have an RDF vocabulary defined for your data, you will want to put it onto a website so that applications can use it. You can use your new vocabulary to semantically tag any components within applications. For the database example above, you might create a new table to hold metadata linking each column and table name to their RDF types. It could be as simple as an entry for each table/column name and the corresponding URI from

your RDF vocabulary that describes its meaning. You might use this for automatically generating documentation or in analyzing and reusing application code. Using RDF for this type of metadata is a convenient way to tag the data without changing anything in the existing data structures. For our Java classes, we could also add code annotations or JavaDoc tags to semantically mark up our code to facilitate its reuse.

There are some well-known standard RDF vocabularies that you can use to build your own vocabulary. The first one to consider using is a vocabulary extension to RDF, created by the W3C, called the *OWL Web Ontology Language*. It includes vocabulary along with formal semantics that you can use in your own definitions. OWL builds on the framework created by the RDF and RDF schema vocabularies. Although we used the RDF schema's `subClassOf` property, OWL has a much more comprehensive version that adds formal semantics such as property restrictions and set operations. Jena has an OWL helper class with static variables for each of the OWL resources and properties. Another common RDF standard is the *Dublin Core (DC)*, an element set for describing metadata about information resources of any kind. It defines generic properties such as title, creator, type, format, language, and rights. The type property uses values from the *Type Vocabulary*, part of the Dublin Core. Some examples of types are collection, dataset, interactive resource, and software. In Jena, there is a `DC` class with static `Property` variables for each of the Dublin Core properties. You can add a type property to an item within a model by using:

```
model.add(myDatabaseResource, DC.type, DCTypes.Dataset);
```

This marks the resource `myDatabaseResource` as being a type of `Dataset`. Combining with RDF schema or OWL, you can create your own hierarchy of terms using these as a baseline. For example, you might create terms for "JDBC-accessible database," "relational database table," and "relational database column" that are RDF subclasses of `Dataset`. You could then define unique URIs for specific instances of these and make statements about them in RDF: "MySQL instance #743234 at OurOrganization contains data about employees, stored in the table named Employee." Having such metadata available can make managing IT resources much easier.

Eventually there will probably be a standard upper-level ontology for all information technology terms. Many groups are working to create standard vocabularies for various domains. One effort, the *Suggested Upper Merged Ontology (SUMO)*, aims to develop an upper-level hierarchy for all abstract concepts. Future applications that use ontologies based on this may be able to make high-level inferences using data from entirely different domains. There are some domain-specific hierarchies that are also based on SUMO. In this section's resource page, there is an updated list of some existing vocabularies that you can use. In the next section, we attach an RDF document as metadata for an HTML document.

Getting Attached: Attaching Dublin Core to HTML Documents

One of our original reasons for exploring RDF (besides it being cool!) was because of the limited linking capability of HTML. We'd like web browsers to still be able to display our HTML and web content, yet also have metadata available for processing by search engines and automated knowledge discovery systems. Given that most websites are probably still going to be using HTML for many more years, has RDF solved our link metadata problem yet? In some ways it has. There are several ways of marking up HTML documents with Dublin Core or other RDF metadata. The method I'll be using here is the method suggested by the Dublin Core, and it also embeds the metadata without affecting the browser's view of the data and without breaking the XHTML validation.

The browser may or may not know how to do anything with our RDF data, but we are assuming that other programs may be able to process it. We will need to embed the metadata so that it doesn't interfere with the browser's understanding or rendering of the HTML. We can do this by using `link` and `meta` tags in our HTML. Any programs that read this data should have a way to discover which technique we are using. Rather than let programs make assumptions (which could be wrong), we place a marker as an attribute of the `head` tag of the HTML, telling any programs how to retrieve this metadata:

```
<head profile="http://dublincore.org/documents/dcq-html/">
```

The profile URI means that there is metadata in the HTML document and that it should be interpreted in the manner associated with the given profile. Any software processing this document will also need to know the schemas for RDF prefixes used in the metadata. We do this by placing link tags in the `head` section:

```
<link rel="schema.DC" href="http://purl.org/dc/elements/1.1/" />
<link rel="schema.DCTERMS" href="http://purl.org/dc/terms/" />
```

You can now add the actual Dublin Core properties to `meta` tags in the `head` section. It's the same as using RDF triples, but the implicit subject of each triple is the current HTML document. Here is an example showing how to attach title and subject metadata to a document:

```
<meta name="DC.title" xml:lang="en"
      content="The World is Full of RDF" />
<meta name="DC.subject" content="earth" />
```

See this book's website for more information on HTML metadata and the Dublin Core.

What's the Reason? Making Queries with Jena RDQL

JENA

You've built the perfect ontology for your organization's knowledge base. You've encoded it in RDF based on standard vocabularies, so you can exchange data with other applications. And now you have a large amount of data encoded using this vocabulary. "But what can I do with all this data?" you think to yourself. "It's not like I can just use a query language like SQL!" Well, actually, you can—not specifically with the SQL language but with a similar structured language designed for querying knowledge bases. In this section, we'll use an RDF query language to retrieve information from an existing knowledge base.

Because RDF data is not organized into tables, columns, and rows like a relational database, SQL won't work for querying RDF graphs. Instead, we need to search within a graph to find subgraphs that match some pattern of RDF nodes (subject, predicate, and object). For instance, you might ask a knowledge base whether a particular employee is a supervisor. In this case, you know the subject, predicate, and object that you are looking for. You can directly ask whether the given structure exists in the RDF. However, most often you won't know every part of the target structure, such as when you want a list of supervisors having a salary less than $100,000. Because we don't know the URI of each item, we will have to use variables to represent the unknown items in the query. In this type of query, we are asking: "Show me all X where X is a supervisor, and X has salary Y, and Y < 100000." The response will list all the possible values for X that would match the desired properties. Jena's built-in query language is called *RDF Data Query Language (RDQL)*. An RDQL query has several parts:

- What values the query should return
- The RDF sources to query
- The query predicates
- Optional namespace prefixes

RDQL will let us declare the RDF source (where the data is coming from) directly within the query `String`, but that is very inefficient for multiple queries against the same source. It's usually better to run the query from an RDF model already in memory. Let's run a query on the Suggested Upper Merged Ontology (SUMO), a very high-level ontology created by the IEEE. SUMO has standard names for high-level abstractions such as Process, Organization, and GeopoliticalArea. These are not Java classes; they are classes in the mathematical sense: a set whose members share one or more properties in common. We'll look at Organization and find all of its direct subclasses, using the RDQL query:

```
SELECT ?x
WHERE (?x <rdfs:subClassOf> <sumo:Organization>)
USING rdfs FOR <http://www.w3.org/2000/01/rdf-schema#>
      sumo FOR <http://reliant.teknowledge.com/DAML/SUMO.owl#>
```

The ?x in this query is a variable representing something that we want the query to locate. The query engine will try to substitute a value for ?x wherever it finds a subclass of Organism. Remember that all entities in RDF are URIs. The rdfs and sumo prefixes make the URIs in the query much shorter and less awkward. To run the query in Jena, we first load the SUMO ontology into memory. Then we run the query using the static exec method of Jena's Query class and process the results. The following code performs this query:

```
Model sumo = ModelFactory.createOntologyModel();
String sumoURL = "http://reliant.teknowledge.com/DAML/SUMO.owl";
sumo.read(sumoURL);
sumo.setNsPrefix("sumo", sumoURL + "#");
String rdq = "SELECT ?x " +
   "WHERE (?x <rdfs:subClassOf> <sumo:Organization>) " +
   "USING rdfs FOR <http://www.w3.org/2000/01/rdf-schema#> " +
         "sumo FOR <" + sumoURL + "#>";
QueryResults results = Query.exec(rdq, sumo);
RDFVisitor aVisitor = new SysoutVisitor();
while (results.hasNext())
{
   ResultBindingImpl binding = (ResultBindingImpl) results.next();
   RDFNode node = (RDFNode) binding.get("x");
   node.visitWith(aVisitor);
}
```

This matches the known subclasses of the Organization entity in SUMO. To visit each node and display its URI, you'll need to write a visitor, using Jena's RDFVisitor interface. My SysoutVisitor class prints out the URI of each node that it visits. You can do more interesting things with a visitor besides just printing a node's value, such as visiting nodes connected to it by a particular property. Here is the code for SysoutVisitor:

```
public class SysoutVisitor implements RDFVisitor {
   public Object visitBlank(Resource r, AnonId id) {
      System.out.println("anon: " + id);
      return null;
   }

   public Object visitURI(Resource r, String uri) {
      System.out.println("uri: " + uri);
      return null;
   }

   public Object visitLiteral(Literal l) {
      System.out.println(l);
      return null;
   }
}
```

There is a feature of the Visitor pattern that lets a visitor return a value, but we are not using that feature here. To make the program do something else instead of print each node's value, all you need to do is plug in a different visitor. The previous query matches the following nodes:

```
http://reliant.teknowledge.com/DAML/SUMO.owl#Corporation
http://reliant.teknowledge.com/DAML/SUMO.owl#PoliticalOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#EducationalOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#JudicialOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#ReligiousOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#GovernmentOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#Organization
http://reliant.teknowledge.com/DAML/SUMO.owl#MercantileOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#Manufacturer
http://reliant.teknowledge.com/DAML/SUMO.owl#Government
http://reliant.teknowledge.com/DAML/SUMO.owl#PoliceOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#MilitaryOrganization
http://reliant.teknowledge.com/DAML/SUMO.owl#MilitaryForce
http://reliant.teknowledge.com/DAML/SUMO.owl#ParamilitaryOrganization
```

Jena can also make rule-based inferences. You can create a knowledge base, combine it with SUMO facts, and query the model while applying matching rules. See the documentation and tutorial links on the resource page for more details. The W3C recently created its own query language called SPARQL, which works very similarly to Jena's. See this book's website for updated information on this and other query languages.

Simply Logical: Lojban, RDF, and the Jorne Project

Lojban (www.lojban.org) is an artificial spoken and written language based on the concepts of predicate logic. While it was designed to be used by human beings, it has a parseable grammar and structured semantics that make it ideal for processing by computers. Lojban defines words based on predefined predicate root words called *gismu*. Each root word has a specific structure associated with it, containing one to five slots that can be filled with nouns (Lojban calls these items *sumti*). For example, the Lojban predicate "bevri" means the act or process of carrying something, and it functions much like a verb. Within its structure are also contained five other related concepts: carrier, cargo, delivery-destination, delivery-source, and delivery-path. While in English and most other languages these may be separate words, in Lojban they are references to positions within the bevri structure.

There are over 1,300 root gismu in the Lojban vocabulary, and these structures form a very interesting ontology of their own. Each of them has between one and five slots. Most of the gismu don't have five slots like bevri does. In fact, there are only a few gismu with five parameters. Table 4-1 shows the number of gismu of each *arity*, or parameter count, and the total number of slots as of this writing.

Table 4-1: Gismu Count, by Arity

Gismu Arity	Gismu Count	Total Slots
1	73	73
2	555	1110
3	535	1605
4	171	684
5	18	90
Total		**3562**

The slots in the root predicates give us 3,500+ base concepts. These can be combined in many different ways by using compound words and logical connectives, but for our purposes here we are looking at the root words only as base concepts. Perhaps you are now wondering, "So what does all this have to do with the Semantic Web?" In an earlier section, we discussed some existing ontologies with built-in relationships that we might use to describe our own entities. Lojban has a convenient set of base concepts that could be used in creating an ontology.

Lojban fits in very well with RDF, which also maps verbs as predicates, although RDF uses graphs of "subject verb object" predicates, and Lojban uses a slot-based approach. There is some mapping required in order to integrate the two, and although it can be done, no standard RDF ontology exists for Lojban—yet. In January 2005, I created an open-source project called Jorne to define standards for combining Lojban with the Semantic Web. Once these standards are complete, the project will release Java software to convert Lojban text to and from RDF triples. One of the goals of this project is for a human to be able to write Lojban text and have the computer automatically convert it into RDF statements for running queries against knowledge bases.

Published ontologies such as SUMO are great for mapping terms from one vocabulary to another, such as in creating dictionaries. The Jorne project is working to map Lojban terms onto well-known vocabularies, so that Lojban documents can share a common semantic space with RDF documents. When the Jorne project completes its first standards, the Jorne project page (www.jorne.org) will hold the latest RDF files along with some sample documents. For creating terms in your own vocabularies, you may want to build upon the SUMO vocabulary, since it is already linked to many others. In Chapter 5, we will discuss a dictionary standard based on English word senses, called *WordNet*, and a Java API for working with it. WordNet has also been mapped to RDF and SUMO. See this book's website for more information on these and other ontologies.

Guess What? Publishing RSS Newsfeeds with Informa

INFORMA

RDF Site Summary (RSS) is a standard for summarizing content on a web server. An RSS feed is stored in an XML file, and it might include items such as recent news, changes to a website, or new *blog* entries. A client program called an *aggregator* collects RSS feeds from multiple web servers and displays them in summary form, sorted by category. The user then chooses to view the full content of any summaries that are of interest. The summary has metadata, such as its subject, encoded along with a text summary. Over time I expect that document metadata will have much more than the Dublin Core and other terms that RSS currently uses. In theory, you could plug into other ontologies such as SUMO, and the meaning of an entire article could be encoded using RDF. This is possible only if you are using an ontology that is expressive enough. This is certainly a lot of effort, but the long-term advantage is that machines would have access to the fully encoded semantics of the text. This probably won't happen for a while, but adding metadata such as RSS descriptions is a good start in that direction and has an immediate benefit of giving us more accurate categorization of content.

There are several standards named RSS, all of them XML-based and used for similar purposes. Unfortunately the different standards not only have different XML structures but even use different definitions for the RSS acronym. Most aggregators are able to understand all RSS flavors, though. The version we discuss here, RDF Site Summary 1.0, uses RDF and is most closely related to the semantic work we've done so far in this chapter. However, it's still better to use *something* rather than encoding no metadata at all. There are ways to map between the semantics of each standard, although all of them are not equally expressive. One common practice is to use XSL-T stylesheets to transform between the different forms of RSS.

Because RSS 1.0 is built on RDF and XML, there are several ways of creating feeds: a DOM parser, an RDF API, or an RSS-specific API. DOM is more low-level than is necessary for creating RDF. Jena has RSS support through its `RSS` class, which has static objects that represent RSS properties you can use in building an RSS-compatible RDF graph. But if you're going to be working a lot with RSS, you'll want to use an RSS-specific API that can understand the different RSS versions that are commonly used.

Informa is an open-source API for reading and writing RSS in Java. One of its most powerful features is the ability to persist the feed metadata in a database. Informa can also read data from external feeds (as described in a later section), perform text-filtering tasks, and update RSS content on a periodic schedule. Let's use it to create a feed using the basic in-memory builder—the `ChannelBuilder` class from the `de.nava.informa.impl.basic` package. In RSS terminology, a *channel* is another name for metadata about some content (such as a website) and is the main entity in a newsfeed. Each RSS file defines a channel and items belonging to the channel. Rather than work with the XML directly, which can be somewhat tedious, we'll use a `ChannelBuilder` to create the RSS file.

```
ChannelBuilder builder = new ChannelBuilder();
ChannelIF myChannel = builder.createChannel("Latest Bug Fixes");
// This is the URL for which we are describing the metadata
URL channelURL = new URL("http://example.org/wcj/bugs.rss");
myChannel.setLocation(channelURL);
myChannel.setDescription("The latest news on our bug fixes");

// We create a first item
String title = "Annoying Bug #25443 Now Fixed";
String desc = "A major bug in OurGreatApplication is fixed. " +
  "Bug #25443, which has been annoying users ever since 3.0, " +
  "was due to a rogue null pointer.";
URL url = new URL("http://example.org/wcj/bugfix25443.html");
ItemIF anItem =
  builder.createItem(myChannel, title, desc, url);
anItem.setCreator("Ecks Amples");

// We create a second item
title = "Bug #12121 not Fixed in 7.1";
desc = "Bug #12121 will not be fixed in OurGreatApplication " +
  "release 7.1, so that developers can focus on adding " +
  "the WickedCool feature.";
url = new URL("http://example.org/wcj/bugfix12121.html");
anItem = builder.createItem(myChannel, title, desc, url);
anItem.setCreator("Dee Veloper");

// export the document to disk, in RSS 1.0 format
ChannelExporterIF exporter = new RSS_1_0_Exporter("bugs.rss");
exporter.write(myChannel);
```

You can place the XML-encoded RSS feed anywhere on your site. The main page of your site should include a link to the feed. For automated discovery by RSS crawlers such as Syndic8, you can do this with a `link` tag in the page's `head` section:

```
<link rel="alternate" type="application/rss+xml"
title="Bugs" href="http://your-site/bugs.rss" />
```

You'll also want a hypertext link for human visitors, so they can add your site to their aggregator. If you are going to be creating large feeds that change often or working with many feeds simultaneously, use the *Hibernate*-based version of the builder, which will persist the RSS metadata in a database. Hibernate is an API for mapping Java objects to relational database structures and automatically translating data between them. See the Informa documentation, and this section's resource page, for more information. In the next section, we'll see how to read newsfeeds with Informa.

What's Up? Aggregating RSS Newsfeeds

INFORMA

JAVA 5+

In the previous section, we used the Informa library to create RSS content, so that visitors with content aggregators can be automatically informed about updates to your site. Another great use of RSS within your site is displaying recent news related to your industry. You can get these newsfeeds from many sources, such as news sites, websites in your industry, and aggregator sites like Syndic8. Make sure to check whether the sites you are syndicating will allow you to incorporate items from their feeds into your site. Usually this is the case, but not always.

Let's start by reading items from a newsfeed and displaying them as text. Using Informa, reading an RSS feed is easy. You can populate the same `ChannelBuilder` object that we used in the previous section with data from an existing RSS feed. The `FeedParser` class has a `parse` method that returns a `ChannelIF` instance containing the channel data from the RSS feed. The RSS standards may be in a state of confusion, but the Informa API reads all of them and gives us a common object model for working with them.

```
import de.nava.informa.impl.basic.Channel;
import de.nava.informa.impl.basic.ChannelBuilder;
import de.nava.informa.impl.basic.Item;
import de.nava.informa.parsers.FeedParser;

ChannelBuilder builder = new ChannelBuilder();
String url = "http://wickedcooljava.com/updates.rss";
Channel channel = (Channel) FeedParser.parse(builder, url);
System.out.println("Description: " + channel.getDescription());
System.out.println("Title: " + channel.getTitle());
System.out.println("===================================");
// using Java 5 syntax in this for loop
for (Object x : channel.getItems())
{
    Item anItem = (Item) x;
    System.out.print(anItem.getTitle() + " - ");
    System.out.println(anItem.getDescription());
}
```

This will print some basic information about the channel and its items. If you want to include these in a web page, it's now just a matter of wrapping HTML tags around the text. If you are including RSS files that are outside your control, you may want to filter data from the channels before displaying them. We'll discuss this in a later section.

Heading to the Polls: Polling RSS Feeds with Informa

INFORMA

We just showed how Informa can retrieve data from an RSS channel, using the `ChannelBuilder` class. Ideally, updating your copy of the feed should be an automated process, and Informa can also do this. The `Poller` class (located

in the de.nava.informa.utils.poller package) can periodically poll a Channel object's RSS feed and trigger some action whenever there are changes. By default, this polling occurs every 60 minutes but can be configured to use longer or shorter periods. The Poller class works by notifying an observer object whenever something changes in the feed. To use this process, you must first create a class implementing the PollerObserverIF interface. This interface has methods for poll tracking, error handling, and feed change notification.

Let's look at an example of a PollerObserverIF that uses the newItem method, which the Poller calls whenever the feed has a new item. However, the new item will not be added to the copy in your Channel object unless the observer explicitly adds it. Here is a PollerObserverIF implementation that does not add feed changes to the Channel object but instead prints a notification message to the console:

```
public class AnObserver
implements de.nava.informa.utils.poller.PollerObserverIF
{
   public void itemFound(ItemIF item, ChannelIF channel) {
      System.out.println("New item found");
      channel.addItem(item);
   }

   public void pollStarted(ChannelIF channel) {
      System.out.println(
         "Started poll with " + channel.getItems().size() +
         " items in channel");
   }
      public void pollFinished(ChannelIF channel) {
      System.out.println(
         "Finished poll with " + channel.getItems().size() +
         " items in channel");
   }
      public void channelChanged(ChannelIF channel) {}
   public void channelErrored(ChannelIF channel, Exception e) {}
}
```

This observer will print information about the beginning and end of each polling event, list any new items in the feed, and add new items to the object model. Warning: An observer does not add new items to the Channel object unless you explicitly call the addItem method. If you have more than one observer attached, one of them should be assigned the task of adding the new item to the Channel. With real RSS feeds, you'll want to set a polling frequency that doesn't clog the network or the site with unnecessary traffic. A polling period of 60 minutes (the default) or longer should be frequent enough for most sites. The following code fragment uses the observer that we just defined and polls the RSS feed for a previously loaded Channel object every 60 minutes.

```
Poller poller = new Poller();
poller.addObserver(new AnObserver());
poller.registerChannel(channel);
```

To use a three-hour interval instead of the default, you can call:

```
poller.registerChannel(channel, 3 * 60 * 60 * 1000);
```

Make sure to remember that the polling interval is specified in milliseconds! If you are going to filter items from the feed, the observers should not be doing the filtering. There is a separate component that can approve polled changes prior to observer notification. This keeps the observers focused on their task of propagating changes rather than filtering data. The process is more scalable that way, as you may want many observers to receive approved changes. This filtering and approval process is described in the next section.

All the News Fit to Print: Filtering RSS Feeds with Informa

INFORMA

In the previous section, we polled an RSS feed and wrote some code that automatically updates our copy of the `Channel` object whenever the feed changes. Our `PollerObserverIF` implementation added the item to a `Channel` object. You may think that the observer would be a good candidate for doing some filtering of the feed content, such as deciding whether to add new items to our copy. This could work, but since there can be more than one observer connected to a `Poller`, it's better to have a separate object do the filtering. By doing this, we won't need to duplicate any filtering functions, and all the observers can benefit equally from the filtering process.

Informa implements filters through an approval process. You can add one or more approvers to a `Poller`. The observers will see a new item only if all of the approvers accept it. The approval must be a unanimous vote or the change will remain invisible to the observers (that is, the observers' `newItem` method is not called). To add an approver, implement the `PollerApproverIF` interface and pass it to the `Poller`'s `addApprover` method. By making fine-grained approvers, you can use them in a plug-and-play manner. For example, you could have a `NoBadWordsApprover` that checks for the existence of words that you don't want to appear on your website or to be added to the `Channel`. In a similar way, a `RelevancyApprover` class could check for keywords that are relevant to your intended usage of the feed.

Approvers check properties within each item, such as the category list and subject, to determine whether an item should be approved. `PollerApproverIF` has only a single method, as indicated in this example that checks the title and the description of each item using regular expressions (as discussed in Chapter 2). Here is the approver class:

```
public class RelevancyApprover
implements PollerApproverIF {
```

```
 public boolean canAddItem(ItemIF item, ChannelIF channel) {
    String title = item.getTitle();
    String description = item.getSubject();
    if (title.matches(".*Java.*") || description.matches(".*Java.*"))
    {
      return true;
    }  else  {
      return false;
    }
  }

}
```

As you might guess, this approver accepts only items that have "Java" somewhere in the title or description. The next code fragment adds this approver to a `Poller`. The approver should be added before the observer, and the observer added before registering the channel:

```
Poller poller = new Poller();
poller.addApprover(new RelevancyApprover());
poller.addObserver(new AnObserver());
poller.registerChannel(channel);
```

There is another class similar to the `Poller`, the `Cleaner`, that can periodically remove unwanted items in a channel. It uses a similar process: `CleanerObserverIF` observers are added to a `Cleaner`, and `CleanerMatcherIF` instances decide what should be removed. Perhaps these interfaces should be called "JuryMember" and "Executioner," because that is a very good metaphor for what they do! You might use the `Cleaner` to remove items that are older than a few days or meet some other criteria for removal. For both the `PollerApproverIF` and `CleanerMatcherIF` decision making, you might want to integrate Lucene text matching, as described in Chapter 3. This would give much more sophisticated text-matching abilities, such as similarity ("fuzzy") matches.

Chapter Summary

The techniques of semantic tagging that we've described in this chapter are quickly becoming popular in large published data sets, and in the next few years the Semantic Web will see an exponential growth. The latest news and website updates, along with what your colleagues are *blogging*, are already being gathered automatically by RSS aggregators and organized by category. In business-to-business transactions, common high-level ontologies are beginning to connect domains with completely different terminology in ways that were impossible before. For example, within highly specific scientific disciplines, new discoveries often use domain-specific terms to describe their findings. This information could lead to breakthroughs in other disciplines, if it were only translated into the appropriate terminology.

Structured newsfeeds are already bringing current news and other information to anyone with an aggregator and a network connection. Using more detailed semantic markup (with SUMO or other high-level ontologies), information could be made even more accessible to everyone—even if the original document uses obscure terminology or a foreign language. We will soon see new types of aggregators and intelligent agents that make logical inferences based on the news and perhaps act on our behalf. Organizations that are properly prepared for this will be able to use the Semantic Web much more effectively. One way to start preparing now is by identifying each type of data with a URI, adding a machine-readable RDF type description (for example, that the item is a person, hardware, software, or some other entity), and using standard ontologies where possible. Jena, Informa, and the ontologies discussed in this chapter are some tools that can help you with this process. In the next chapter, we discuss intelligent software agents and explore some of the scientific and mathematical APIs for Java.