Examples

From Info216

Here are the code examples we have used in the live sessions during the lectures - along with a few additional ones.

(More will appear as the course progresses.)

Contents

Lecture 1: Java, Jena, and Eclipse

Hello Jena

```
package no.uib.sinoa.info216;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;
import org.apache.jena.rdf.model.Resource;
import org.apache.jena.vocabulary.FOAF;

public class HelloJena {

    public static void main(String[] args) {

        Model model = ModelFactory.createDefaultModel();

        Resource andreas = model.createResource(base + "Andreas");
        Resource info216 = model.createResource(base + "INFO216");
        Property teaches = model.createProperty(base + "teaches");
        andreas.addProperty(teaches, info216);

        andreas.addLiteral(FOAF.name, "Andreas L Opdahl");

        model.write(System.out, "TURTLE");
    }
}
```

Lecture 2: RDF

Resource objects

```
package no.uib.infomedia.info216;

...

public class HelloJena {
    public static void main(String[] args) {
        String iriBase = "http://no.uib.infomedia.info216/";
        String iriDbpedia = "http://dbpedia.org/resource/";
```

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

        Resource resCadeTracy = model.createResource(iriBase + "Cade_Tracy");
        resCadeTracy.addLiteral(FOAF.name, "Cade Tracy");

        Resource resCanada = model.createResource(iriDbpedia + "Canada");
        Resource resFrance = model.createResource(iriDbpedia + "France");
        Property propVisited = model.createProperty(iriBase + "visited");
        resCadeTracy.addProperty(propVisited, resCanada);
        resCadeTracy.addProperty(propVisited, resFrance);

        model.write(System.out, "TURTLE");
    }
}
```

Language-tagged literals

```
        resFrance.addProperty(RDFS.label, "Frankrike", "no");
        resFrance.addProperty(RDFS.label, "France", "en");
        resFrance.addProperty(RDFS.label, "Francia", "es");
```

Typed literals

```
        Property propPopEst = model.createProperty(iriDbpedia + "ontology/populationEstimate");
        resFrance.addProperty(propPopEst, "66644000", XSDDatatype.XSDinteger);
```

Looping through statements

```
        for (Statement stmt : model.listStatements().toList()) {
            System.out.println(stmt.toString());
        }
```

Selecting statements

```
        for (Statement stmt : model
                .listStatements((Resource)null, RDFS.label, (RDFNode)null)
                .toList()) {
            System.out.println("Subject:   " + stmt.getSubject().toString());
            System.out.println("Predicate: " + stmt.getPredicate().toString());
            System.out.println("Object:    " + stmt.getObject().toString());
        }
```

Using a selector

```
        for (Statement stmt : model
                .listStatements(new SimpleSelector() {
                    public boolean test(Statement s) {
                        return (s.getPredicate().equals(FOAF.name));
                    }
                })
                .toList()) {
            System.out.println(stmt.getObject().toString());
        }
```

Writing to file

```
        try {
            model.write(new FileOutputStream("test.ttl"), "TURTLE");
        } catch (Exception e) {
            // TODO: handle exception
        }
```

Contents of test.ttl

```
<http://no.uib.infomedia.info216/Cade_Tracy>
        <http://no.uib.infomedia.info216/visited>
                <http://dbpedia.org/resource/France> , <http://dbpedia.org/resource/Canada> ;
        <http://xmlns.com/foaf/0.1/name>
                "Cade Tracy" .

<http://dbpedia.org/resource/France>
        <http://www.w3.org/2000/01/rdf-schema#label>
                "Francia"@es , "France"@en , "Frankrike"@no ;
        <http://dbpedia.org/resource/ontology/populationEstimate>
                66644000 .
```

Reading from file

```
package no.uib.infomedia.sinoa.info216;

import java.io.FileInputStream;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;

public class ReadJena {

    public static void main(String[] args) {
        Model model = ModelFactory.createDefaultModel();

        try {
            model.read(new FileInputStream("test.ttl"), "http://ex.org/", "TURTLE");
        } catch (Exception e) {
            // TODO: handle exception
        }

        model.write(System.out);
    }
}
```

Reading from web resource

```
package no.uib.infomedia.sinoa.info216;

import java.io.InputStream;
import java.net.HttpURLConnection;
import java.net.URL;

import org.apache.jena.rdf.model.Model;
import org.apache.jena.rdf.model.ModelFactory;

public class HttpTest {

    public static void main(String[] args) {
        Model model = ModelFactory.createDefaultModel();

        try {
            URL url = new URL("http://people.uib.no/sinoa/test.ttl");
            HttpURLConnection urlConnection = (HttpURLConnection) url.openConnection();
            InputStream is = urlConnection.getInputStream();
```

```
            model.read(is, "http://ex.org/", "TURTLE");
        } catch (Exception e) {
            // TODO: handle exception
        }

        model.write(System.out);
    }
}
```

(There are more advanced ways to download web resources from Java, but HttpUrlConnection is a straightforward and built-in way to get started.)

Lecture 3: SPARQL Query and Update

Basic INSERT DATA update

```
                Dataset dataset = DatasetFactory.create();

                UpdateAction.parseExecute(""
                                + "PREFIX info216: <http://ex.org/teaching#>"
                                + "INSERT DATA {"
                                + "    info216:cade info216:teaches info216:ECO001 . "
                                + "    GRAPH <http://ex.org/personal#Graph> {"
                                + "        info216:cade info216:age '29' . "
                                + "    }"
                                + "}", dataset);

                RDFDataMgr.write(System.out, dataset, Lang.TRIG);
```

To output only the default graph use:

```
                dataset.getDefaultModel().write(System.out, "TURTLE");
```

The method dataset.getNamedModel("http://ex.org/personal#Graph"); lets you output a named model instead.

Basic SELECT query

```
    ResultSet resultSet = QueryExecutionFactory
        .create(""
            + "SELECT ?s ?p ?o WHERE {"
            + "    ?s ?p ?o ."
            + "}", dataset)
        .execSelect();

    resultSet.forEachRemaining(qsol -> System.out.println(qsol.toString()));
```

Convert the ResultSet into a JSON object

```
    List<Map> jsonList = new Vector<Map>();
    while (resultSet.hasNext()) {
        QuerySolution qsol = resultSet.nextSolution();
        Iterator<String> varNames = qsol.varNames();
        Map<String, Object> jsonMap = new HashMap<String, Object>();
        while (varNames.hasNext()) {
            String varName = varNames.next();
            jsonMap.put(varName, qsol.get(varName).toString());
        }
        jsonList.add(jsonMap);
```

```
    }
    System.out.println(JsonUtils.toPrettyString(jsonList));
```

SELECT query with Query object

```
    // select example
    Query query = QueryFactory.create(""
            + "SELECT ?s ?p ?o WHERE {"
            + "    ?s ?p ?o ."
            + "}");
    QueryExecution queryExecution = QueryExecutionFactory.create(query, dataset);
    ResultSet resultSet = queryExecution.execSelect();
```

SELECT query from SPARQL endpoint

If a named graph in your dataset has a triple with a subject that is a DBpedia IRI:

```
    ResultSet resultSet = QueryExecutionFactory.create(""
                    + "SELECT ?s ?p ?o WHERE {"
                    + "    GRAPH ?g { ?s ?p2 ?o2 . } "
                    + "       SERVICE <http://dbpedia.org/sparql> {"
                    + "        ?s ?p ?o ."
                    + "    }"
                    + "}", dataset).execSelect();

    while (resultSet.hasNext()) {
        QuerySolution qs = resultSet.nextSolution();
        System.out.println(qs.toString());
    }
```

Basic ASK query

```
    QueryExecution queryExecution = QueryExecutionFactory.create(""
            + "ASK { GRAPH ?g { ?s ?p ?o } }"
            + "", dataset);
    boolean res = queryExecution.execAsk();
    System.out.println("The result is " + res);
```

ASK query from IRL

```
    QueryExecution queryExecution = QueryExecutionFactory.create(""
            + "ASK "
            + "    FROM <http://people.uib.no/sinoa/european-populations.ttl> { "
            + "       <" + iriDbpedia + "Vatican_City> ?p ?o . "
            + "    }"
            + "");
    boolean res = queryExecution.execAsk();
    System.out.println("The result is " + res);
```

Basic DESCRIBE query

```
    Model franceModel = QueryExecutionFactory.create(""
            + "DESCRIBE <" + iriDbpedia + "France>"
            + "", dataset).execDescribe();
    franceModel.write(System.out, "TURTLE");
```

Basic CONSTRUCT query

```
    Model franceModel = QueryExecutionFactory.create(""
            + "CONSTRUCT { ?s ?p ?o . } WHERE { "
            + "    GRAPH ?g { ?s ?p ?o . } "
            + "}", dataset).execConstruct();
    franceModel.write(System.out, "TURTLE");
```

CONSTRUCT query from IRL

```
    Model franceModel = QueryExecutionFactory.create(""
            + "CONSTRUCT { ?s ?p ?o . } "
            + "    FROM <http://people.uib.no/sinoa/european-populations.ttl> "
            + "WHERE { "
            + "    ?s ?p ?o . "
            + "}").execConstruct();
    franceModel.write(System.out, "TURTLE");
```

Complex SPARQL predicates (Fuseki)

In the apache-jena-fuseki-*version* folder:

```
fuseki-server --localhost --update --mem /mem
```

In your web browser, goto http://localhost:3030/ . Use SPARQL ENDPOINT http://localhost:3030/mem/update for the INSERT updates and http://localhost:3030/mem/query for the SELECT queries below.

```
PREFIX x: <http://example.org/myex#>

INSERT DATA {
    x:IngridAlexandra x:father x:HaakonMagnus ;
                      x:mother x:MetteMarit .
    x:HaakonMagnus x:sister x:MarthaLouise .
}
```

Keep the PREFIX line in all the following queries:

```
SELECT ?s ?o WHERE {
    ?s (x:father | x:mother) ?o .
}
```

```
SELECT ?s ?o WHERE {
    ?s (x:father / x:sister) ?o .
}
```

```
SELECT ?s ?o WHERE {
    ?s ^(x:father / x:sister) ?o .
}
```

```
SELECT ?s ?o WHERE {
    ?s (^x:sister / ^x:father) ?o .
}
```

```
SELECT ?s ?o WHERE {
    ?s !x:sister ?o .
}
```

Add some mother-triples:

```
INSERT DATA {
    x:HaakonMagnus x:father x:Harald .
    x:Harald x:father x:Olav .
    x:Olav x:father x:Haakon .
}
```

```
SELECT ?o WHERE
{
    x:IngridAlexandra x:father+ ?o .
}
```

```
SELECT ?o WHERE
{
    x:IngridAlexandra x:father* ?o .
}
```

```
SELECT ?o WHERE
{
    x:IngridAlexandra x:father? ?o .
}
```

```
SELECT ?o WHERE
{
    x:IngridAlexandra x:father{2} ?o .
}
```

```
SELECT ?o WHERE
{
    x:IngridAlexandra x:father{2,4} ?o .
}
```

SPARQL SELECT VALUES (and services)

The code below retreives DBpedia descriptions of the three Scandinavian capitals, using VALUES:

```
                Dataset dataset = DatasetFactory.create();

                ResultSet table = QueryExecutionFactory.create(""
                                + "SELECT * WHERE {"
                                + "    VALUES ?city {"
                                + "            <http://dbpedia.org/resource/Copenhagen>"
                                + "            <http://dbpedia.org/resource/Oslo>"
                                + "            <http://dbpedia.org/resource/Stockholm>"
                                + "    }"
                                + "    SERVICE <http://dbpedia.org/sparql> {"
                                + "                ?city <" + RDFS.comment + "> ?comment ."
                                + "    }"
                                + "}", dataset).execSelect();

                table.forEachRemaining(row -> System.out.println(row));
```

To retrieve only English-language descriptions, you can add a FILTER inside the SERVICE call:

```
...
                + "    SERVICE <http://dbpedia.org/sparql> {"
                + "                ?city <" + RDFS.comment + "> ?comment ."
                + "        FILTER( lang(?comment) = 'en' )"
                + "    }"
...
```

Language-tagged literals (and functions, and services...)

This works because we use the language-tagged literal 'Copenhagen'@en in the INSERT DATA update (and, as a result, it outputs DBpedia-triples about Copenhagen):

```
Dataset dataset = DatasetFactory.create();

String prefixes = ""
                + "PREFIX rex: <http://ex.org#>"
                + "PREFIX dbpedia: <http://dbpedia.org/resource/>";
UpdateAction.parseExecute(prefixes
                + "INSERT DATA {"
                + "    rex:Margrethe <" + FOAF.based_near + "> 'Copenhagen'@en ."
                + "}", dataset);

ResultSet table = QueryExecutionFactory.create(prefixes
                + "SELECT * WHERE {"
                + "    ?person <" + FOAF.based_near + "> ?label ."
                + "    SERVICE <http://dbpedia.org/sparql> {"
                + "        ?city <" + RDFS.label + "> ?label ."
                + "    }"
                + "}", dataset).execSelect();

table.forEachRemaining(row -> System.out.println(row.toString()));
```

If we do INSERT DATA *without* the language tag @en, we get no result (because the city labels in DBpedia are language-tagged):

```
UpdateAction.parseExecute(prefixes
                + "INSERT DATA {"
                + "    rex:Margrethe <" + FOAF.based_near + "> 'Copenhagen'@en ."
                + "}", dataset);
```

We can, however, rewrite the query to use the *strlang* function that adds language tags to labels. So this works with the previous INSERT DATA:

```
ResultSet table = QueryExecutionFactory.create(prefixes
                + "SELECT * WHERE {"
                + "    ?person <" + FOAF.based_near + "> ?label ."
                + "    BIND(strlang(?label, 'en') AS ?taglabel)"
                + "    SERVICE <http://dbpedia.org/sparql> {"
                + "        ?city <" + RDFS.label + "> ?taglabel ."
                + "    }"
                + "}", dataset).execSelect();
```

To go the other way (if our local labels were language-tagged and DBpedia's not, we could do a similar thing, but use the reverse *str*-function instead of *strlang*.

Explanation of table.forEachRemaining(...)

The lambda-syntax in this code line may be new for you:

```
	table.forEachRemaining(row -> System.out.println(row.toString()));
```

The straightforward way to write it **(that doesn't work)**, would be just:

```
	table.forEachRemaining(System.out.println(row.toString()));
```

But this doesn't work because we cannot send a method call as a parameter in Java, and our code introduces a *row* variable that is not declared anywhere.

Instead, we could try to define a new method somewhere else inside our class:

```
	void printRow(QuerySolution row) {
		System.out.println(row.toString());
	};
```

and then pass that function to forEachRemaining **(doesn't work either)**:

```
	table.forEachRemaining(printRow);
```

But this doesn't work because you cannot pass functions as arguments like that in Java.

Finally, we could try to define a whole new class - a subclass of a Consumer class:

```
class RowPrinter implements Consumer<QuerySolution> {
	public void accept(QuerySolution row) {
		System.out.println(row.toString());
	}
}
```

Here, we define the class RowPrinter as a *Consumer* of *QuerySolution*s, because Jena defines each row in a ResultSet-table to be a QuerySolution. In other words, we can view a ResultSet (table) as a *stream* of QuerySolutions (rows) that the RowPrinter consumes one by one. The method name *accept* is defined by the *Consumer*-class. We cannot chose another name (and, actually, *Consumer* is a Java interface, not a class).

We *can* pass an object of this class to forEachRemaining - this is exactly what it expects (**this works!**):

```
	table.forEachRemaining(new RowPrinter());
```

But it is pretty cumbersome to write a new class like RowPrinter every time we want to do a forEachRemaining-call - or some other streaming call. Therefore Java 8 has introduced a shorthand, a *lambda expression*:

```
	table.forEachRemaining(row -> System.out.println(row.toString());
```

Whenever Java 8 or later sees code like this, it behaves as if we had explicitly written a RowPrinter or similar Consumer-class like the one above.

Lecture 4: TDB and Fuseki

Creating a dataset

```
        Dataset dataset = TDBFactory.createDataset();
        Model defaultModel = dataset.getDefaultModel();

        ...

        dataset.close()
```

This creates an in-memory dataset, which is not persistent.

Creating/loading a TDB-backed dataset

```
        Dataset dataset = TDBFactory.createDataset("TDBTest");
        Model defaultModel = dataset.getDefaultModel();

        ...

        dataset.close()
```

The first time it is run, this creates a persistent dataset, backed by a TDB triple store located in the directory "TDBTest" inside your Eclipse project. Refresh the project to see it (or F5).

When re-run later, this loads the dataset from the TDB store.

It is important to close a TDB-backed dataset before the program terminates. Otherwise, you need to go into the database folder (for example the "TDBTest" directory inside your Eclipse project) and delete the file named "tdb.lock". (Do a refresh with F5 if you do not see it in Eclipse.)

Fuseki

When you get started, it is easiest to run Fuseki from the directory where you unpacked it along with the other Jena downloads, for example:

```
cd C:\Programs\Jena\apache-jena-fuseki-3.6.0
```

or

```
cd /opt/Jena/apache-jena-fuseki-3.6.0
```

Start the Fuseki server with this command on Windows:

```
fuseki-server --localhost --loc=C:\...\your\Eclipse\workspace\INFO216\TDBTest /tdb
```

On Linux:

```
sh fuseki-server --localhost --loc=C:\...\your\Eclipse\workspace\INFO216\TDBTest /tdb
```

Here, TDBTest is the name of the triple store, INFO216 is the name of the Eclipse project, located inside your Eclipse workspace. Use the **--help** option to see what the other options do.

Open a web browser and go to **localhost:3030** to run queries/updates and otherwise explore and use the TDB-backed dataset.

You can also start Fuseki without the "--loc" option:

```
fuseki-server --localhost
```

or

```
sh fuseki-server --localhost
```

When you go to **localhost:3030** in your web browser, Fuseki will now appear empty at first, but you can create new datasets and load triples into them from files or from the web. If you choose to create datasets that are persistent, they will not end up in the Eclipse project folder, but in a subfolder of the Fuseki-installation folder, for example:

```
C:\Programs\Jena\apache-jena-fuseki-3.6.0\runatabases\TDBTest
```

Lecture 5: RDFS

Creating an RDFS model

```
	Model rdfModel = ModelFactory.createDefaultModel();
	InfModel rdfsModel = ModelFactory.createRDFSModel(rdfModel);
```

RDFS entailment: subClassOf

```
	String iriBase = "http://no.uib.infomedia.info216/";

	Resource resUCB = rdfsModel.createResource(iriBase + "UCB");
	Resource resUniversity = rdfsModel.createResource(iriBase + "University");
	resUCB.addProperty(RDF.type, resUniversity);

	Resource resHEI = rdfsModel.createResource(iriBase + "HEI");
	resUniversity.addProperty(RDFS.subClassOf, resHEI);

	rdfsModel.write(System.out, "TURTLE");
```

The output will show that University of Califoria, Berkeley (UCB) is a Higher-Education Institution (HEI), even though we did not assert that explicitly.

Outputting the RDFS axioms

```
	ModelFactory
	    .createRDFSModel(ModelFactory.createDefaultModel())
	    .write(System.out, "TURTLE");
```

Removing axioms from RDFS outputs

Here, we write the triples in rdfsModel to the console, after eliminating all triples that are axioms:

```
        InfModel axiomModel = ModelFactory.createRDFSModel(ModelFactory.createDefaultModel());
        ModelFactory
            .createDefaultModel()
            .add(rdfsModel)
            .remove(axiomModel)
            .write(System.out, "TURTLE");
```

RDFS entailment: subPropertyOf

```
        Resource resCadeTracy = rdfsModel.createResource(rdfsModel.getNsPrefixURI("") + "Cade_Tracy"
        Property propHasBScFrom = rdfsModel.createProperty(rdfsModel.getNsPrefixURI("") + "hasBScFrc
        resCadeTracy.addProperty(propHasBScFrom, resUCB);

        Property propGraduatedFrom = rdfsModel.createProperty(rdfsModel.getNsPrefixURI("") + "gradua
        propHasBScFrom.addProperty(RDFS.subPropertyOf, propGraduatedFrom);

        rdfsModel.write(System.out, "TURTLE");
```

The output will show that Cade graduated from University of Califoria, Berkeley (UCB), even though we did not assert that explicitly.

Chained RDFS entailment: subPropertyOf and domain

```
        propGraduatedFrom.addProperty(RDFS.domain, FOAF.Person);

        rdfsModel.write(System.out, "TURTLE");
```

The output will show that Cade is a FOAF person, even though we did not assert that explicitly.

The Reasoner object

This outputs the name of the Reasoner's class:

```
        System.out.println(rdfsModel.getReasoner().getClass().toString());
```

Adding namespace prefixes

```
        rdfsModel.setNsPrefix("", iriBase);
        rdfsModel.setNsPrefix("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#");
        rdfsModel.setNsPrefix("rdfs", "http://www.w3.org/2000/01/rdf-schema#");
```

We can do this in a single call too, by first creating a map:

```
        rdfsModel.setNsPrefixes(new HashMap() {{
            put("", iriBase);
            put("rdf", "http://www.w3.org/1999/02/22-rdf-syntax-ns#");
```

```
            put("rdfs", "http://www.w3.org/2000/01/rdf-schema#");
        }});
```

(The "double-brace notation" *{{ ... }}* lets us add code to initialise the new HashMap object at construction, before we pass it as a parameter to setNsPrefixes.)

Instead of

```
        Resource resUCB = rdfsModel.createResource(iriBase + "UCB");
```

we can now write

```
        Resource resUCB = rdfsModel.createResource(rdfsModel.getNsPrefixURI("") + "UCB");
```

to reduce the need for global strings in large programs and to eliminate inconsistencies by keeping all prefixes in the same place.

Lecture 6: RDFS Plus

Here and below we use this main method, so we can place the Jena code in the constructor for the HelloRDFSPlus class (this is a little bit tidier than putting everything straight into the *static main()* method :-)).

```
    public static void main(String[] args) {
        new HelloRDFSPlus();
    }
```

Minimal OWL model example

A minimal working *OntModel* example:

```
    HelloRDFSPlus() {

        String base = "http://ex.org/info216#";
        String prefixes = ""
            + "PREFIX ex: <" + base + "> "
            + "PREFIX owl: <" + OWL.getURI() + "> ";

        OntModel owlModel = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_RULE_INF);

        UpdateAction.parseExecute(prefixes
            + "INSERT DATA {"
            + "    ex:INFO310 ex:buildsOn ex:INFO216 . "
            + "    ex:INFO216 ex:buildsOn ex:INFO116 . "
            + "    ex:INFO116 ex:buildsOn ex:INFO100 . "
            + "    ex:buildsOn a owl:TransitiveProperty . "
            + "}", owlModel);

        owlModel.getWriter("TURTLE").write(owlModel, System.out, "base");
    }
```

The last line is important: you cannot write out an OntModel using the owlModel.write(...) method that we used for RDF and RDFS models.

Outputting an OWL model without the axioms

You can also output the OntModel without axioms:

```
        OntModel emptyOwlModel = ModelFactory.createOntologyModel(OntModelSpec.OWL_MEM_RULE_INF);
        ModelFactory
            .createDefaultModel()
            .add(owlModel)
            .remove(emptyOwlModel)
            .write(System.out, "TURTLE");
```

ASK query

A more direct way to test that reasoning over transitive properties works:

```
        System.out.println("Transitive properties work: " +
            QueryExecutionFactory
                .create(prefixes
                    + "ASK {"
                    + "    ex:INFO310 ex:buildsOn ex:INFO100 . "
                    + "}", owlModel)
                .execAsk());
```

Listing OWL statements

Or you can output just the triples that have buildsOn as their predicate:

```
        ObjectProperty buildsOn = owlModel.createObjectProperty(base + "buildsOn");
        List<Statement> stmts = owlModel
                .listStatements((Resource)null, buildsOn, (RDFNode)null)
                .toList();
        for (Statement stmt : stmts)
            System.out.println(stmt.toString());
    }
```

Adding OWL triples through method calls

Instead of using SPARQL Update, you can populate the OntModel as follows:

```
        Individual info100 = owlModel.createIndividual(base + "INFO100", OWL.Thing);
        Individual info116 = owlModel.createIndividual(base + "INFO116", OWL.Thing);
        Individual info216 = owlModel.createIndividual(base + "INFO216", OWL.Thing);
        Individual info310 = owlModel.createIndividual(base + "INFO310", OWL.Thing);

        ObjectProperty buildsOn = owlModel.createObjectProperty(base + "buildsOn");
        buildsOn.addProperty(RDF.type, OWL.TransitiveProperty);
        info310.addProperty(buildsOn, info216);
        info216.addProperty(buildsOn, info116);
        info116.addProperty(buildsOn, info100);
```

(We pass OWL.Thing to the createIndividual method to indicate the class of the individual we create. Because we have not created a specific ex:Course class, we just use OWL.Thing, which is the most general of all classes in OWL.)

Lectures 7-9: Vocabularies and Linked Open Datasets

No code examples from these lectures.

Lecture 10: JSON and JSON-LD

JSON-LD web API to Dataset

The following code loads the result of a JSON-LD web API call straight into a Jena dataset:

```
String uri = "http://www.europeana.eu/api/v2/record/92065/"
        + "BibliographicResource_1000056084136.jsonld?wskey=tX3Zstfo2";
Model model = RDFDataMgr.loadModel(uri, Lang.JSONLD);
model.write(System.out, "TURTLE");
```

If you are going to run this more than once, please register at [1] (https://pro.europeana.eu/get-api) to get your own API key, instead of using my personal key *tX3Zstfo2*. You can find more JSON-LD web APIs at [2] (https://github.com/json-ld/json-ld.org/wiki/Users-of-JSON-LD).

Web API to JSON Object

In newer versions of Jena (at least since 3.2.0), you can load the results of regular JSON web API call into a Java object as follows:

```
String url = "http://api.geonames.org/postalCodeLookupJSON?postalcode=46020&country=ES&username=
Object jsonObj = JsonUtils.fromURL(new URL(url), JsonUtils.getDefaultHttpClient());
```

(The user name *demo* is only good for a small number of daily calls. You will need to register your own user name at api.geonames.org to call GeoNames many times in a day. See the top bullet points here: [3] (http://www.geonames.org/export/web-services.html).)

Prettyprint a JSON Object

```
System.out.println(JsonUtils.toPrettyString(jsonObj));
```

Read JSON-LD string into a Jena model

When you have a JSON-LD string, you can read it into a Jena model as follows (replace "" with a base IRI if needed):

```
String jsonStr = JsonUtils.toPrettyString(jsonObj);

Model model = ModelFactory.createDefaultModel();
RDFDataMgr.read(model, new StringReader(jsonStr), "", Lang.JSONLD);

model.write(System.out, "TURTLE");
```

or as follows (again, replace "" with a base IRI if needed):

```
String jsonStr = JsonUtils.toPrettyString(jsonObj);

Model model = ModelFactory.createDefaultModel();
model.read(IOUtils.toInputStream(jsonStr, "UTF-8"), "", "JSON-LD");

model.write(System.out, "TURTLE");
```

Expand a JSON Object

Proxy object: The next few examples will assume the following proxy JSON object, which we pretend has been returned from a JSON web API call:

```
// we pretend this string comes from a regular JSON web API call
String proxyJsonReponse = ""
        + "{"
        + "    \"name\" : \"Markus Lanthaler\", "
        + "    \"workplaceHomepage\" : \"http://www.homepage.com/ML\", "
        + "    \"address\" : {"
        + "        \"streetAddress\" : \"Somestreet 123\", "
        + "        \"cityAddress\" : \"ZIP-4567 Acity\" "
        + "    }"
        + "}";

// parse the string into a JSON object
Object jsonObj = JsonUtils.fromString(proxyJsonReponse);
System.out.println(JsonUtils.toPrettyString(jsonObj));
```

Create context object: First create a context object:

```
// create a context object
String baseIRI = "http://ex.org/base#";

Map contextObj = new LinkedHashMap();
contextObj.put("name", FOAF.name.getURI());
contextObj.put("workplaceHomepage", "@id");
contextObj.put("address", baseIRI + "address");
contextObj.put("streetAddress", baseIRI + "streetAddress");
contextObj.put("cityAddress", baseIRI + "cityAddress");
```

This is a more advanced way to do the same thing, which some people think looks clearer:

```
// create a context object
String baseIRI = "http://ex.org/base#";

Map contextObj = new LinkedHashMap() {{
        put("name", FOAF.name.getURI());
        put("workplaceHomepage", "@id");
        put("address", baseIRI + "address");
        put("streetAddress", baseIRI + "streetAddress");
        put("cityAddress", baseIRI + "cityAddress");
}};
```

(The latter example uses an *anonymous inner class* with an *instance initialization block*. This explains the double-nested { { and } }-braces: the outer {...} pair is for the anonymous class and the inner {...} pair is for its initialization block, which is the only thing the inner class defines.)

Note that without a mapping for "address" in the context, expansion will not work for "street-" and "cityAdress", because they are nested inside "address" in the JSON object.

We have two ways to expand a JSON object: either with or without an explicit options object.

Simple compacting: We first try expansion without an explicit options object. This is the simplest way to do it:

```
((Map) jsonObj).put("@context", contextObj);
Object expandedObj = JsonLdProcessor.expand(jsonObj);
System.out.println(JsonUtils.toPrettyString(expandedObj));
```

To avoid the type-unsafe casting, you can do:

```
Map json = new LinkedHashMap();
if (jsonObj instanceof Map)
    json = (Map) jsonObj;
json.put("@context", contextObj);
Object expandedObj = JsonLdProcessor.expand(json);
System.out.println(JsonUtils.toPrettyString(expandedObj));
```

Compacting with options object: We now look at expansion with an explicit options object, which can give more control:

```
// create and set an options object
JsonLdOptions expandOptions = new JsonLdOptions(baseIRI);
expandOptions.setExpandContext(contextObj);

// expand the JSON object
Object expandedObj = JsonLdProcessor.expand(jsonObj, expandOptions);
System.out.println(JsonUtils.toPrettyString(expandedObj));
```

Flatten an expanded JSON-LD object

```
Object flattenedExpandedObj = JsonLdProcessor.flatten(expandedObj, new JsonLdOptions());
System.out.println(JsonUtils.toPrettyString(flattenedExpandedObj));
```

Compact a JSON-LD Object

You can run the code below on the following proxy string, or you can use it on the outputs from a semantic JSON-LD web API.

```
// we pretend this string comes from a semantic JSON-LD web API call
String proxyJsonLdReponse = ""
        + "{"
        + "    \"@id\" : \"http://www.homepage.com/ML\", "
        + "    \"http://xmlns.com/foaf/0.1/name\" : [ { "
        + "        \"@value\" : \"Markus Lanthaler\" "
        + "    } ], "
        + "    \"http://ex.org/base#address\" : [ { "
        + "        \"http://ex.org/base#cityAddress\" : [ { "
        + "            \"@value\" : \"ZIP-4567 Acity\" "
        + "        } ], "
        + "        \"http://ex.org/base#streetAddress\" : [ { "
        + "            \"@value\" : \"Somestreet 123\" "
        + "        } ] "
        + "    } ] "
        + "}";

// create a JSON object
Object jsonObj = JsonUtils.fromString(proxyJsonLdReponse);
System.out.println(JsonUtils.toPrettyString(jsonObj));
```

First create a context object:

```
// create a context object
String baseIRI = "http://ex.org/base#";
Map contextObj = new HashMap() {{
    put("@context", new HashMap() {{
        put("name", FOAF.name.getURI());
        put("workplaceHomepage", "@id");
```

```
            put("address", baseIRI + "address");
            put("streetAddress", baseIRI + "streetAddress");
            put("cityAddress", baseIRI + "cityAddress");
        }});
    }};
```

Then compact the JSON-LD object using the context object:

```
    Object compactedObj = JsonLdProcessor.compact(jsonObj, contextObj, new JsonLdOptions());
    System.out.println(JsonUtils.toPrettyString(compactedObj));

    // if you want pure JSON, you can also remove the context object:
    ((Map) compactedObj).remove("@context");
```

Flatten a compacted JSON object

Flatten the compacted JSON object:

```
    Object flattenedCompactedObj = JsonLdProcessor.flatten(compactedObj, defaultOptions);
    System.out.println(JsonUtils.toPrettyString(flattenedCompactedObj));
```

Web API to String

If you want full control of your HTTP connection, this is a fairly simple and straightforward way to call a web API and return the results as a String. It is independent of Jena and Json. (There are many APIs available that do this in more advanced ways, but it will get you started.)

```
    // calls a Web API and returns the result as a string
    // this method is not necessary, but included because it may be useful for some of you
    String getResponseBody(URL serverAddress) {
        String responseBody = null;

        HttpURLConnection connection = null;

        BufferedReader rd  = null;
        StringBuilder sb = null;
        String line = null;

        try {
            // send GET request
            connection = null;
            connection = (HttpURLConnection)serverAddress.openConnection();
            connection.setRequestMethod("GET");
            // connection.setDoOutput(true);
            connection.setReadTimeout(10000);
            connection.connect();

            // receive response
            rd  = new BufferedReader(new InputStreamReader(connection.getInputStream()));
            sb = new StringBuilder();

            // turn response into a string
            while ((line = rd.readLine()) != null)
            {
                sb.append(line + '\n');
            }
            responseBody = sb.toString();

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (ProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
```

```
        } finally {
            // close the connection
            connection.disconnect();
            rd = null;
            sb = null;
            connection = null;
        }

        return responseBody;
    }
```

Web API to JSON Object

Builds on the above full-control example, but assumes the result of the web API call is JSON and parses it.

```
    static Object getJsonBody(URL serverAddress) {
        Object jsonObject = null;
        HttpURLConnection connection = null;

        try {
            // send GET request
            connection = null;
            connection = (HttpURLConnection)serverAddress.openConnection();
            connection.setRequestMethod("GET");
            // connection.setDoOutput(true);
            connection.setReadTimeout(10000);
            connection.connect();

            // parse JSON reponse
            jsonObject = JsonUtils.fromInputStream(connection.getInputStream());

        } catch (MalformedURLException e) {
            e.printStackTrace();
        } catch (ProtocolException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        } finally {
            //close the connection
            connection.disconnect();
            connection = null;
        }

        return jsonObject;
    }
```

You call this like we did in one of the first examples, but with the call-line changed:

```
    String url = "http://api.geonames.org/postalCodeLookupJSON"
            + "?postalcode=46020&country=ES&username=demo";
    Object jsonObj = getJsonBody(new URL(url));
```

Web API call proxy

While you develop and test, you will fire off the same API calls many times. This will usually be unproblematic, but it can cause problems if you exceed hourly/daily/etc. API call limits. In such cases you can write API call proxies, which look like real API calls to the rest of the program, and return JSON strings or object, but which return the same test string/object every time (without actually making an API call).

For example, the below method seems to behave like the previous **getJsonBody** method, but returns a fixed JSON object:

```
    static Object getJsonBodyProxy(URL url) {

        String jsonBody = "{\"postalcodes\":[{\"adminCode2\":\"V\",\"adminCode1\":\"VC\","
                + "\"adminName2\":\"Valencia\",\"lng\":-0.377386808395386,"
                + "\"countryCode\":\"ES\",\"postalcode\":\"46020\","
                + "\"adminName1\":\"Comunidad Valenciana\",\"placeName\":\"Valencia\","
                + "\"lat\":39.4697524227712}]}";
        try {
            return JsonUtils.fromString(jsonBody);
        } catch(Exception ex) {
            return null;
        }
    }
```
