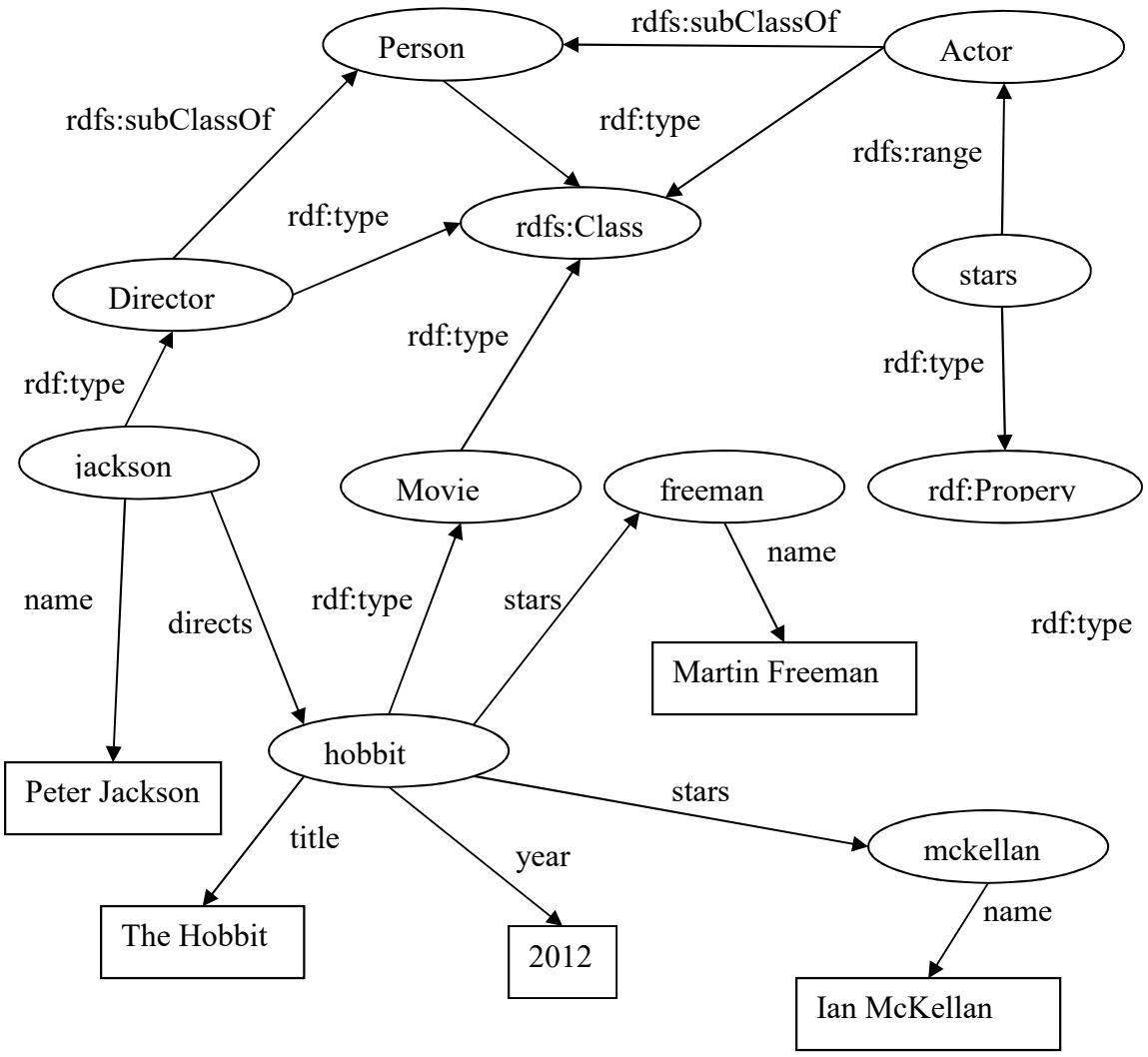
Homework #1: RDF

The following exercises are due at the beginning of class on Monday, Feb. 10. There are two sections: written exercises and electronic exercises. This will count for 10% of your overall grade.

Written Exercises:
The exercises in this section should be completed and turned in on paper.

1. *[10 pts.]* Translate the following RDF Graph into the Turtle syntax. For nodes and arcs labeled with qnames, assume the standard namespaces apply. For other names, assume they are all local to the document you are writing. Note. in Turtle, relative URIs are writing in angle brackets, just like other URIs.



2. *[10 pts.]* Using the RDFS entailment rules rdfs2, rdfs3, rdfs5, rdfs7, rdfs9, and rdfs11 (see the RDF Semantics recommendation [https://www.w3.org/TR/rdf11-mt/], Section 9.2), determine what triples can be inferred from the example in Problem #1. You may ignore the other entailment rules because they do not add anything particularly interesting. Give your answer by listing additional triples in Turtle.

3. *[10 pts.]* Consider the following RDF document using the XML syntax. Draw the equivalent graph. For convenience, you may use qnames for your node and edge labels.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
         xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
         xmlns:u="http://www.example.org/uni#"
         xml:base="http://www.example.org/uni">
<rdfs:Class rdf:ID="Person" />
<rdfs:Class rdf:ID="Student">
   <rdfs:subClassOf="#Person" />
</rdfs:Class>
<rdfs:Class rdf:ID="Professor">
   <rdfs:subClassOf="#Person" />
</rdfs:Class>
<rdfs:Class rdf:ID="Course" />
<rdf:Property rdf:ID="advises">
   <rdfs:domain rdf:resource="#Professor" />
   <rdfs:range rdf:resource="#Student" />
   <rdfs:subPropertyOf="#knows">
</rdf:Property>
<rdf:Property rdf:ID="takes">
   <rdfs:domain rdf:resource="#Student" />
   <rdfs:range rdf:resource="#Course" />
</rdf:Property>
<rdf:Property rdf:ID="teaches">
   <rdfs:domain rdf:resource="#Professor" />
   <rdfs:range rdf:resource="#Course" />
</rdf:Property>
<rdf:Property rdf:ID="knows" />
<u:Professor rdf:ID="alan">
   <u:teaches rdf:resource="#cs100" />
   <u:advises rdf:resource="#rob" />
   <u:advises rdf:resource="#sarah" />
</u:Professor>
<u:Student rdf:ID="rob">
   <u:takes rdf:resource="#cs100" />
   <u:takes rdf:resource="#cs200" />
</u:Student>
</rdf:RDF>
```

4. *[10 pts.]* According to the RDFS Schema recommendation, a property can have multiple rdfs:range statements. When this occurs the range of the property is treated as the intersection of the individual ranges. What advantage does this have over using the union of the individual ranges to determine the actual range? Hint: Consider the distributed design of RDF and the inferences that are sanctioned by rdfs:range.

Electronic Exercises:

For these two problems, you must use Jena 3.14.0 to solve two basic tasks in RDF. I expect these two programs to total somewhere between 250 and 300 lines of code, so plan your time accordingly. Note, the Jena distribution includes a number of JAR files, and many (but not all) of these will need to be in your classpath for your programs to compile and run. I give command-line examples for each program, where *jenaLoc* stands for the location where you installed Jena. Note, in the examples below, I use Linux style file paths; if you are working in Windows, be sure to replace the forward slashes with back slashes. You will need to import classes from the **org.apache.jena.rdf.model** package, and may find the **org.apache.vocabulary.RDF** class useful as well. All of your classes should be placed in a Java package named with your Lehigh LTS account user name (e.g., aaa120), referred to as *userId.* . in the descriptions below.

5. *[25 pts.]* Consider the file **top20albums.txt** that is available on the course web page. This file includes a list of albums, one per line. Each line has the form:
   rank artist album year
   where fields are separated by tabs. Decide on an approach of representing this data as an RDF graph, choosing URIs for classes and properties as appropriate. You may create new URIs or reuse ones defined by existing schemas; you do not need to create an explicit RDF schema. Choose a scheme for generating a unique URI for each album. Then use Jena to write a Java class *userId*.**AlbumsToRdf** that can read in any file of the same form, and write out an equivalent RDF/XML file.

   From the command-line, your program should run as:

   `java -cp` *jenaLoc*`/lib/*;.` *userId.*`AlbumsToRdf` *input output*

   where *input* is the name of the text file to read from and *output* is the name of the resulting RDF file. File names will be specified as paths relative to the JVM's working directory.

6. *[35 pts.]* Using Jena, write a Java class *userId*.**ReadPubRdf** that can read in an RDF file describing a set of publications, and then output a condensed list of these publications to the screen, one per line. The input file will use the vocabulary from the RDF Schema **swpub.rdf,** which is available on the course web page. From the command-line, your program should run as:

   `java -cp` *jenaLoc*`/lib/*;.` *userId.*`ReadPubRdf` *rdf-filename*

   where *rdf-filename* is a relative path to the input RDF. Each entry in the output should be of the form:

   pub-type: **"***title***" by** *author*$_1$**,** *author*$_2$**, ... and** *author*$_n$ **(***year***)**

   where each italicized symbol represents information from a triple. The *pub-type* placeholder should be one of: **Article In Periodical, In Proceedings, Book Chapter,** or **Tech Report.** You can ignore any resources that are not instances of one of the corresponding classes. I have provided the file **reasoning.rdf** to help you test your program. If we ignore the linebreaks below, the output line corresponding to the first publication in this file might look like:

   ```
   In Proceedings: "SAOR: Template Rule Optimisations for
   Distributed Reasoning over 1 Billion Linked Data Triples" by
   Aidan Hogan, Jeff Z. Pan, Axel Polleres, and Stefan Decker (2010)
   ```

   Note, the input files will use the <rdf:Seq> and <rdf:li> elements to provide ordered lists of authors. Your code must appropriately navigate the resulting Jena model constructs.

   On the course website, I provide a file **SwPub.java** that includes useful constants for the various classes, properties and other resources, defined in **swpub.rdf**. If you want to use this file, add the line "package *userId;*" to the top of it and then include it in your directory with your other Java files.

The program should terminate successfully for any syntactically correct RDF/XML file. You should ignore any triples that do not match the expected vocabulary, and you should ignore any publications that are missing the required **rdf:type**, **title**, **authorList** or **publishedYear** triples or that have authors that are missing **fullName** triples.

Submission:

These exercises require both submission of files and hardcopies of these files. Create a zip file *userId*-prog.zip that contains your source code (.java) files, organized according to Java's file structure (i.e., classes in packages are in subdirectories corresponding to the package naming structure. Do not include any .class or Jena files in your submission, and do not put the code in a "src" directory. The contents of your zip file should be something like:

/*userId*/AlbumsToRdf.java
/*userId*/ReadPubRdf.java
…

Note, I will unzip your file directly into the working directory I will use. I will then run the commands as described for each exercise above. Thus it is important the your submission be organized exactly as I have described. I strongly recommend that before submitting, you unzip your files in a fresh directory, copy the input files, and try to run them using the commands I give above. Once you are confident your zip file will work as expected, submit it via CourseSite.

Print out your .java file(s) and turn in the hardcopy with the rest of your written answers.

Grading:

Your programs will be graded on functionality (90%) and style (10%). Style includes modularity (avoid repeated code when possible, keep methods under ~60 lines, use multiple classes when appropriate), commenting (all of your files should be reasonably commented, including an initial comment that identifies you as the author and descriptive comments for each class and method), proper indentation, clear names, and use of standard naming conventions.

If I cannot immediately compile your program or run it using the procedure above, it will be returned to you to fix, and then a late penalty will be assessed depending on how long it takes you to fix it.