NICOLAS MATENTZOGLU AND IGNAZIO PALMISANO

AN INTRODUCTION TO THE OWL API

UNIVERSITY OF MANCHESTER

PUBLISHED BY UNIVERSITY OF MANCHESTER

TUFTE-LATEX.GOOGLECODE.COM

First printing, April 2016

Contents

Introduction

Version information

This is **Version 1.0** of the OWL API Tutorial handbook.

For contributions we thank:

- Sean Bechhover
- Robert Stevens
- Uli Sattler

This handbook is under constant development. For suggestions and corrections please drop an email to nicolas.matentzoglu@gmail.com.

Requirements

In order to follow this tutorial we expect you to have the following prerequisites:

- Be familiar with Java development.
- Understand the basic concepts of the Web Ontology Language (OWL). This one is crucial, as many of the methods to create class expressions would be otherwise hard to follow.
- Be quite familiar with your IDE of choice.
- You do not have to be an expert at Maven, but a background in using Java build tools (maybe you have used Ant) is certainly helpful.

Welcome

Welcome to our introduction to programming with the OWL API. The OWL API has been around for more than a decade, and is under active development with a team of dedicated software engineers. For people interested in some of the background on design decisions, it is worth it to take a look at the main OWL API publication (Horridge et al. 2011) in the Semantic Web Journal [1].

[1] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semantic Web*, 2(1):11–21, 2011

Setting up

In this chapter you will:

1. Setting up Eclipse for OWL API development
2. Learn to understand the differences between versions

We know from experience that many people struggle with setting up their IDE of choice to develop ontology-based applications using the OWL API. Many developers, especially here at Manchester, like to use Eclipse for developing Java-based software, so we stick to it for the sake of convenience. The default bundle for Java developers also comes with Maven pre-installed, which we will use to manage our application development life-cycle. However, if you prefer to use your own IDE, many of the pitfalls we mention and tips we give in the following are applicable to all environments. We assume that you have the latest *Java 8* Development Kit (JDK) installed on your system. After opening your Eclipse (Version Mars.1 at the time of this writing) workspace of choice, we start by creating a simple Maven project.

Task 1: First steps

1. Create new Maven project (File->New->Other->Maven Project). Check the option "Create simple project". Click "Next".
2. Enter Group Id "owlapi.tutorial" and Artifact Id "msc"; click "Finish".
3. Now we have to make sure that your project is referencing the Java 8 Development Kit rather than the Runtime Environment. Right click on the project, select Build Path->Configure build path... and navigate to the "Libraries" tab. If you see the Java JRE there, select it and click "Remove". Now click Add Libraries->JRE System Library->Alternate JRE. If you can find the JDK in this list, select it. If not, click on Installed JREs->Add..->Standard VM->Directory and navigate to your JDK installation directory. On windows for example, this is typically something like

   "C:\Program Files\Java\jdk1.8.0_51"

   Click finish. Back in the list of installed JRE's, select the JDK version, confirm by clicking "Ok". In the next dialogue, make sure that "Alternate JRE" is pointing to the JRE of the JDK, and click "Finish".
4. You should see now in the list of Libraries that the JRE System Library from the JDK is referenced.
5. In the newly created project, double-click on the "pom.xml", and then click on the little tab at the bottom of the main window called "pom.xml".

   Locate <version></version> , and inject right AFTER it (not in-between!) the snippet you can find in the infobox below.
6. Save the pom file.

Eclipse will provide content assist here - pressing Ctrl-space will propose the tags to insert. Also, the dependency can be added in the Dependencies tab - the UI will guide you and create the XML tags for you.

An alternative way to get the dependency tags for any library is to serch for them on `http://search.maven.org` - this is the Maven Central repository, where a large number of Java libraries is hosted. Once you locate the library and version you are after, the XML fragment required is displayed on the page.

Pay attention to the group id - for OWLAPI, the main libraries all have `net.sourceforge.owlapi` as their group id.

Insert just after <version></version> . In case you already have the <dependencies> element, copy only the dependency elements that are necessary:

```
<dependencies>
<dependency>
<groupId>junit</groupId>
<artifactId>junit</artifactId>
<version>3.8.1</version>
<scope>test</scope>
</dependency>
<dependency>
<groupId>net.sourceforge.owlapi</groupId>
<artifactId>owlapi-distribution</artifactId>
<version>5.0.0</version>
</dependency>
</dependencies>
```

To make sure everything worked out perform the following task.

Task 2: Running your first OWL API program

1. Create a simple test class by performing a right click on the package "owlapi.tutorial" (underneath src) and selecting New->Class. Call this class OWLAPIFirst and click "Finish".
2. Add the 4 lines of Java code in the info-box below to the main method of your `OWLAPIFirst` class.
3. Right click on the "pom.xml" in the package explorer and click Run as–>Maven build. In the upcoming dialogue, set "install" as a goal and click "Run". You should see Maven downloading a number of dependencies of the OWL API. Note that you have to be connected to the internet in order for this to work.
4. Hover over the red underlined class names and select the first quick fix ("Import...").
5. Click on the little black downwards arrow next to the icon that looks like a green circle with a white arrow inside, select Run As->Java Application to execute your program.

Listing 1: Create OWLOntologyManager

```
1 public static void main(String[] args) {
2         OWLOntologyManager man = OWLManager.createOWLOntologyManager();
3         System.out.println(man.getOntologies().size());
4 }
```

Your Eclipse console should show:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See ... for further details.
0
```

Do not worry about the warnings for now. The warning relates to the OWL API internal logging mechanism and will not affect your program.

Congratulations! You have successfully set up you IDE for OWL API development! In the next chapter, we will learn how to manage ontologies with the `OWLOntologyManager` and introduce you to some of the most fundamental concepts of the API.

Differences between versions

While all versions (3, 4 and 5) are currently maintained, only 4 and 5 are under active development. The latest version 4 releae is 4.2.1 (March 2016), and the latest version 5 release is 5.0.0 (March 2016).

The important thing to know for you is that version 4 involved a major revision of the OWL API that is **not backwards compatible** with version 3 - as with many other libraries, changes in the main version number indicate disruptive changes to the API.

So, why do we still mention version 3 if it is superseded? The problem is that, because of the lack of backwards compatibility, many OWL API-based applications or libraries have not been updated yet. There is usually a lag of anything between a few weeks and a few months between an OWL API release and the corresponding releases of related projects; if you have doubts about which version to use, feel free to email `owlapi-developer@lists.sourceforge.net` or open an issue on `https://github.com/owlcs/owlapi/issues` - there are always people there willing to help.

Protégé and the SKOS API[2] have been updated to version 4 only recently (Protege 5.0.0-beta23 is the most recent Protege build at the moment).

[2] `https://github.com/simonjupp/java-skos-api/pull/2`

Essential tools for working with OWL are **reasoners** - implementations of the `OWLReasoner` interface. There are a few of them available for OWL API 4, and there's work ongoing to release reasoners for OWL API 5. Check the Wiki pages on the OWL API website for the latest updates: https://github.com/owlcs/owlapi/wiki

You can find information on how to make your OWL API 3 application compatible with the later versions here:

List of changes: https://github.com/owlcs/owlapi/wiki/Migrate-from-version-3.4-and-3.5-to-4.0
Additional Tricks: https://github.com/owlcs/owlapi/wiki/MigrateTo4Tricks

In the course of this tutorial, we will deal mainly with the latest generations of the OWL API.

Ontology Management

Key Classes:

- OWLManager
- OWLOntologyManager
- OWLOntology
- IRI
- OWLOntologyFormat

1. Creating a new ontology
2. Loading and saving ontologies
3. Understanding the imports closure
4. Understanding IRIs

Creating a new ontology, adding an axiom and saving it.

In this chapter, we will discuss some of the most fundamental ontology management tasks you may be using the OWL API for. Let's dive right into it. The first thing we will do is create a new ontology. For this we will need the following lines of code:

Listing 2: Create OWLOntology

```
1  public static void main(String[] args) {
2          OWLOntologyManager man = OWLManager.createOWLOntologyManager();
3          OWLOntology o;
4
5          try {
6                  o = man.createOntology();
7                  System.out.println(o);
8          } catch (OWLOntologyCreationException e) {
9                  e.printStackTrace();
10         }
11 }
```

The `OWLOntologyManager` is the central class for managing your ontology: it handles creating, loading and saving ontologies, the application of changes such as annotations or axiom additions and keeping track of your imports closure (in the case that one ontology imports another). `OWLManager.createOWLOntologyManager()` is a utility method to create a new `OWLOntologyManager`–in the five years I

(Nico) have been using the OWL API, I have never had a need to create the manager in any other way. `man.createOntology()` asks the freshly created manager to create a new, empty ontology, that is, an ontology without any axioms and annotations and a default ontology id (typically "Anonymous-0", if this is the first ontology you create with that particular manager). Note that a lot of the functionality of the OWL API will require you to deal with exceptions. We assume the reader to be at least vaguely familiar with exception handling, and will, for the remainder of this tutorial, omit the `try {...} catch{...}` blocks from our code snippets. As most of the this tutorial will take place directly in the `main(...)` method, you may simply add the relevant `throws` declaration to it. We do encourage you however to familiarize yourself a bit with the exceptions, as they will become quite relevant once you are debugging your OWL API based applications. The `OWLOntology` class finally is your main point of access your ontology: looking at the axioms and signature, reading annotations, and more.

Running the code above should produce the following output:

```
Ontology(OntologyID(Anonymous-0)) [Axioms: 0 Logical Axioms: 0] First 20
axioms: {}
```

In the next step, we will discuss some of the different ways we can load an ontology. The first method loads an ontology document directly from your local machine using Java's `File` and works as follows:

Listing 3: Load Ontology From File

```
1 OWLOntologyManager man = OWLManager.createOWLOntologyManager();
2 File file = new File("C:\\pizza.owl.xml");
3 OWLOntology o = man.loadOntologyFromOntologyDocument(file);
4 System.out.println(o);
```

Depending on the operating system you are using, the path to the file may be formatted in a slightly different way. The version above shows how it is done in a Windows environment. I downloaded the Pizza ontology at Stanford (`http://protege.stanford.edu/ontologies/pizza/pizza.owl`) for the purpose of demonstration (make sure the file path points to the ontology you downloaded). Running the above program, your output should be something like (list of axioms omitted):

```
Ontology(OntologyID(OntologyIRI(<http://www.co-ode.org/ontologies/
pizza/pizza.owl>) VersionIRI(<null>))) [Axioms: 940 Logical Axioms:
712] First 20 axioms: {...omitted for brevity...}
```

If you want to directly load an ontology from the web, you can use the following code:

Listing 4: Load Ontology From IRI

```
1 OWLOntologyManager man = OWLManager.createOWLOntologyManager();
2 IRI pizzaontology = IRI.create(``
      http://protege.stanford.edu/ontologies/pizza/pizza.owl
      '');
3 OWLOntology o = man.loadOntology(pizzaontology);
4 System.out.println(o);
```

Running this program should have the exact same output as the previous one. There are various other options to load an ontology. You can use your IDE's autocomplete functionality to look through the `loadOntology...()` methods of your `OWLOntologyManager` to get an idea. The most interesting part of this snippet is the instantiation of an `IRI` object (IRI stands for Internationalised Resource Identifier, a sort of upgraded version of URIs that support unicode characters) that provides us with the link to the ontology on the web. We will be using IRI's quite a bit: many components of OWL are represented in the OWL API in the form of IRI's, from classes and properties to the ontology IRI itself (a part of the ontology identifier).

In order to *save* your ontology, we can use the following lines of code:

Listing 5: Save ontology

```
1 OWLOntologyManager man = OWLManager.createOWLOntologyManager();
2 File fileout = new File("C:\\pizza.func.owl");
3 IRI pizzaontology = IRI.create(``
      http://protege.stanford.edu/ontologies/pizza/pizza.owl
      '');
4 OWLOntology o = man.loadOntology(pizzaontology);
5 man.saveOntology(o, new FunctionalSyntaxDocumentFormat(),
6         new FileOutputStream(fileout));
```

Again, there are various ways to save an ontology. This one is the one I have been using quite a bit, so let's go through it. The `saveOntology(...)` method I am using here has three parameters: the ontology that should be saved, the output format we wish the ontology to be stored in and a reference to the file we want to write to in the form of an `FileOutputStream`. The OWL API supports a large number of document formats such as OWL/XML,

Functional, Manchester, Latex, Turtle, RDF/XML, OBO, DL, KRSS and many more. If you are interested in file formats, it is worth looking at the type hierarchy of `OWLDocumentFormatImpl` in your IDE (Ctrl+T in Eclipse).

The Imports Closure

You will notice that your `OWLOntologyManager` can hold more than one ontology. The most important case where this becomes relevant in my experience is when we are dealing with ontologies that depend on other ontologies. In OWL, we typically make this dependency explicit by adding an owl:imports statement. For example, your Sushi ontology (SO) may depend on a Japanese food ontology (JFO), which in turn depends on a general food ontology (FO) that includes knowledge about basic ingredients, their nutritional values and their origin. If you were to load your ontology using the `OWLOntologyManager`, the JFO and the FO would be represented as separate ontologies in your manager. We call the set of all three ontologies, SO, JFO and FO, the "imports closure" of SO (i.e. the imports closure always includes the ontology itself), the set JFO and FO the "imports" of SO and the JFO all by itself the "direct import(s)" of SO. This is very important to remember when using the OWL API to manage ontologies, as you are allowed to add and remove, count and iterate axioms with respect to all three of these sets. In the following, we discuss one example where this becomes obvious.

Each `OWLOntology` object carries a reference to the `OWLOntologyManager` object that created it. I spend years passing both the manager and the ontology itself as parameters to methods. Now I know that `ontology.getOWLOntologyManager()` would have done the trick.

Task 3: Load an ontology from the Web and save locally

1. Load the ontology located at `http://www.cs.man.ac.uk/~stevensr/ontology/family.rdf.owl`
2. Save the loaded ontology somewhere on your system.
3. Create an NEW `OWLOntologyManager`.
4. Load the ontology you have just saved in the previous steps.
5. Add this line: `System.out.println("Axioms: "+o2.getAxiomCount()+", Format: "+mana.getOntologyFormat(o2));`

If you have successfully completed the task, running it should result in the following output:

```
SLF4J: Failed to load class "org.slf4j.impl.StaticLoggerBinder".
SLF4J: Defaulting to no-operation (NOP) logger implementation
SLF4J: See ... for further details.
Axioms: 2847, Format: Manchester OWL Syntax.
```

Change

Key Classes:

- OWLEntity
  - OWLClass
  - OWLObjectProperty
  - OWLDataProperty
- OWLAxiom
  - OWLSubClassOfAxiom
  - OWLClassAssertion
  - OWLAnnotationAssertion
- OWLDataFactory
- OWLAnnotation
- OWLDatatype
- OWLLiteral
- OWLOntologyChange
  - AddAxiom
  - RemoveAxiom

1. Introducing axioms and entities
2. Adding and removing axioms
3. Changing axioms
4. Annotating the ontology, its entities and axioms

Introducing axioms and entities

There are two categories of building blocks when dealing with OWL ontologies entities and axioms. Entities (`OWLEntity`) can be individuals (`OWLIndividual`), i.e. concrete items in the domain (people, countries, genes), classes (`OWLClass`), i.e. sets of individuals (the class of students), object properties (`OWLObjectProperty`), i.e. relationships between individuals (loves, is part of) or data properties, i.e. relationships between individuals and data values (birthdates, names). Axioms are statements about entities, for example "class `Woman` is a subclass of `Person` ", the "property `isPartOf` is transitive", i.e. each element A that is part of one element B is also part of any element C that is part of element B, or the "individual `Robert` is a member of the class `Person` ".

Axioms are the first class citizens of OWL ontologies. In fact, you should view an ontology as a set of axioms, rather than a set of entities, especially when interacting with the ontology using the OWL API. This means for example that you should not think: "Does the ontology contain class A", but rather "Is class A mentioned in any axiom in the ontology?". The OWL API's `getSignature()` method is merely a convenience method that collects the set of entities mentioned across all the axioms in the ontology, and is not maintained separately. This means you cannot "just add a class" to your ontology; you create a so called `Declaration`, a special kind of axiom, which is added to the ontology as follows:

Listing 6: Add Declaration Axiom

```
IRI IOR = IRI.create("http://owl.api.tutorial");
OWLOntologyManager man = OWLManager.createOWLOntologyManager();
OWLOntology o = man.createOntology(IOR);
OWLDataFactory df = o.getOWLOntologyManager().getOWLDataFactory();
OWLClass person = df.getOWLClass(IOR+"#Person");
OWLDeclarationAxiom da = df.getOWLDeclarationAxiom(person);
o.add(da);
System.out.println(o);
```

This is a relatively complex piece of code for such a small operation, and we will discuss it in detail now. First, we instantiate an `IRI` and the `OWLOntologyManager` in the usual way, and use it to create an empty ontology (i.e., an ontology without axioms). Then, we get an instance of the `OWLDataFactory`, a class that will create our ontology building blocks for us, i.e. entities and axioms. The naming conventions of previous OWL API versions suggested that the OWLDataFactory is where you actually "create" a OWL class (as part of the ontology). It is important to understand that this is **not the case**: you are merely instantiating an `OWLEntity` object (e.g. a class or a property) or an `OWLAxiom`, but you are not adding it to your ontology...yet. In OWL, entities are represented by IRI's (a kind of qualified name)—every class, every property, every individual has one. This is reflected by the "get" methods for entities on the `OWLDataFactory`, which are parametrised with IRI's. Once you have obtained a representation of you person class, you use the same data factory as before to create a statement *about* your class; in this case a simple declaration statement, that merely asserts: "there is a named class called Person". This axiom can then be added to the ontology. There are at least three different ways to add an axiom to your ontology: (1) in the newer version of the OWL API, the `OWLOntology` class comes with the `add` convenience method as described by the code snippet above; (2) another way to add above axiom is to use the `OWLOntologyManager` class: `man.addAxiom(o, da);`–this was the most convenient way to do it in earlier versions of the OWL API; (3) if more control is required, it is possible to wrap the axiom into a change object, which is then applied to the ontology: `AddAxiom ax = new AddAxiom(o, da);` `man.applyChange(ax);` This is how changes are deal with by the OWL API internally. For simplicity, we will resort to the first option for the remainder of this tutorial. Let us now add a proper axiom to the ontology that says "Woman is a kind of Person". This is expressed in the form of a `OWLSubClassOfAxiom` as follows:

Listing 7: Add SubCLassOf Axiom

```
OWLClass person = df.getOWLClass(IOR+"#Person");
OWLClass woman = df.getOWLClass(IOR+"#Woman");
OWLSubClassOfAxiom w_sub_p = df.getOWLSubClassOfAxiom(woman, person);
```

```
4 o.add(w_sub_p);
```

The OWL 2 specification requires all entities in the ontology to be declared (see `Declaration` axiom above). Conveniently, adding a "proper" (i.e. logically effectual) axiom, such as the `SubClassOf` axiom in the previous snippet, will trigger the OWL API to add the missing `OWLDeclarationAxiom`'s for both `Person` and `Woman` to the ontology (at the latest when you save the ontology to a text file). In other words: it should rarely be necessary to add `Declaration`'s manually. Let us look at a more complex axiom now:

Listing 8: Create a more complex class expression

```
1  OWLClass A = df.getOWLClass(IOR + "#A");
2  OWLClass B = df.getOWLClass(IOR + "#B");
3  OWLClass X = df.getOWLClass(IOR + "#X");
4  OWLObjectProperty R = df.getOWLObjectProperty(IOR + "#R");
5  OWLObjectProperty S = df.getOWLObjectProperty(IOR + "#S");
6  OWLSubClassOfAxiom ax = df.getOWLSubClassOfAxiom(
7  df.getOWLObjectSomeValuesFrom(R, A),
8  df.getOWLObjectSomeValuesFrom(S, B));
9  o.add(ax);
10
11 o.logicalAxioms().forEach(System.out::println);
```

As you can see, we made use of a new OWL construct: `OWLObjectSomeValuesFrom`. This is used to create expressions such as "`isEnrolledIn some University`" or "`R some A`". The key aspect of this code snippet lies in the creation of the `OWLSubClassOfAxiom`. As you can see, it takes two parameters (i.e. sub and super-class), both (potentially) complex class expressions. Class expressions can consist of very deeply nested expressions. In the exercises at the end of this chapter, you will be asked for example to nest `OWLObjectSomeValuesFrom` with `OWLObjectIntersectionOf` expressions. This nesting principle is similar for all axiom types that can be created with the `OWLDataFactory`!

At least for now, it helps to apply the following recipe when creating complex class expressions and axioms:

1. Create all the entities needed for your axiom (with the `OWLDataFactory`).
2. Create all the sub-expressions need for your axiom (intersections, `OWLObjectSomeValuesFrom`, etc).
3. Assemble the sub-expressions into the final axiom.

We might need to delete an axiom that we previously created. Let's add another axiom as follows:

Listing 9: Add buggy SubClassOf Axiom

```
OWLClass mann = df.getOWLClass(IOR+"#Man");
// mann with two "n" to avoid confusion with the OWLOntologyManager
OWLClass woman = df.getOWLClass(IOR+"#Woman");
OWLSubClassOfAxiom m_sub_w = df.getOWLSubClassOfAxiom(mann, woman);
o.add(m_sub_w);
```

The statement is, at least to most of us, obviously wrong, and we might wish to remove it. Deleting works analogous to adding as follows:

Listing 10: Remove Axiom

```
OWLClass mann = df.getOWLClass(IOR+"#Man");
OWLClass woman = df.getOWLClass(IOR+"#Woman");
OWLSubClassOfAxiom m_sub_w = df.getOWLSubClassOfAxiom(mann, woman);
o.remove(m_sub_w);
```

In order to delete an axiom from your ontology, you first create the axiom exactly as it exists in the ontology using the data factory, and them you remove it. Again, `man.removeAxiom(o, m_sub_w);` or `RemoveAxiom ra = new RemoveAxiom(o, m_sub_w); man.applyChange(ra);` can be used for the exact same purpose.

Changing axioms used to be one of the more painful exercises with the OWL API. Previous to OWL API 5, you had to implement your own method to replace sub-expressions in axioms, or alternatively drop the old version of an axiom from the ontology and add the new, updated version of it. Fortunately for us, that changed now. The following code looks extensive, but everyone who ever had to change axioms (replace sub-expressions or literals) will appreciate its flexibility.

Listing 11: Change Axiom (OA5)

```
final Map<OWLClassExpression, OWLClassExpression> replacements = new HashMap
    <>();

OWLClass A = df.getOWLClass(IOR + "#A");
OWLClass B = df.getOWLClass(IOR + "#B");
OWLClass X = df.getOWLClass(IOR + "#X");
OWLObjectProperty R = df.getOWLObjectProperty(IOR + "#R");
OWLObjectProperty S = df.getOWLObjectProperty(IOR + "#S");
OWLSubClassOfAxiom ax = df.getOWLSubClassOfAxiom(
df.getOWLObjectSomeValuesFrom(R, A),
df.getOWLObjectSomeValuesFrom(S, B));
o.add(ax);

o.logicalAxioms().forEach(System.out::println);

replacements.put(df.getOWLObjectSomeValuesFrom(R, A), X);
```

```
16
17 OWLObjectTransformer<OWLClassExpression> replacer =
18         new OWLObjectTransformer<>((x) -> true, (input) -> {
19                 OWLClassExpression l = replacements.get(input);
20                 if (l == null) {
21                         return input;
22                 }
23                 return l;
24         } , df, OWLClassExpression.class);
25
26 List<OWLOntologyChange> results = replacer.change(o);
27 o.applyChanges(results);
```

We will not get into the details of the above code in this tutorial, but we want to instead focus your attention on the Java `Map` called `replacements`. The keys in this map correspond to the `OWLClassExpression`'s to be replaced. The values are the values that these expresssions should be replaced with. The `OWLObjectTransformer` takes in this map, and performs the replacement, which results in a set of `OWLOntologyChange` objects. These can then simply be applied to your ontology, and you are done. Note that the `OWLObjectTransformer` is generic: You can perform replacements of virtually everything you can think of, not only class expressions, for example `OWLNamedIndividual`, if you need to rename some or all of the individuals in your ontology.

Traversing the ontology

One of the recurring tasks when dealing with your ontology using the OWL API is iterating over the signature (the set of classes, properties and individuals across all axioms in your ontology) or over the set of axioms. In the older versions of the OWL API, you would have done that in the following way (you can still do it like that):

Listing 12: Iterate over Axioms

```
1 for(OWLAxiom ax:o.getLogicalAxioms()) {
2         System.out.println(ax);
3 }
```

If you use Eclipse or similar IDE's for Java development, you will notice that the `getLogicalAxioms()` method is deprecated now. This should not worry you; it is perfectly fine to use it anyways. The OWL API 5 makes heavy use of Java 8's streams to replace the practice of iterating through the signature using the traditional `get` methods. Explaining streams is beyond this short introduction, but the interested reader can take a look

for example at this tutorial: http://winterbe.com/posts/2014/07/31/java8-stream-tutorial-examples/. In a nutshell, a stream represents a sequence of elements, such as the entities in the signature, or the axioms in an ontology, on which you can perform operations in a very convenient, functional-style manner. For example, the above iteration can be performed using the stream analog as follows:

Listing 13: Iterate over Axioms (OA5)

```
1 o.logicalAxioms().forEach(System.out::println);
```

This one-line program has the exact same effect as the previous code snippet, but is terser and often more performant. Using streams can be a bit of a challenge for programmers without a great deal of java experience, but they are a powerful tool to master.

Let's try another example. This time, we want to iterate through all `OWLEntity`'s in the signature (that do not belong to the built-in vocabulary), and for each of them check whether their name starts with a "P".

Listing 14: Filter during iteration (OA5)

```
1 o.signature().filter((e->(!e.isBuiltIn()&&e.getIRI().getFragment().startsWith("P")))).
      forEach(System.out::println);
```

`o.signature()` again offers us the ontologies signature as a stream. Note that by default, this gives us **all elements in the signature**, even some of the built-in vocabulary (such as `rdfs:label` for annotations).

Again, I am a rebel and use the deprecated `getFragment()` method here. Ideally, you should start using the brand new and shiny `getRemainder()` method instead, but it returns an `Optional`, another Java 8 thing that I have not yet fully adapted as a programming style. Essentially, an `Optional` is a clean way to avoid returning null values. So, instead of checking whether a return value is null (or worse, possibly forget checking it in the first place), we now get these `Optional` objects back, whose values we can obtain for example by calling `ifPresent()` on them. The "OWL API 5" way of doing the same thing using `getRemainder()` is as follows:

Listing 15: The IRI Remainder as a Java Optional (OA5)

```
1 o.signature().filter(e->!e.isBuiltIn()&&e.getIRI().getRemainder().orElse("").startsWith
      ("P")).forEach(System.out::println);
```

As this gets a little bit hard to read, here an alternative way, defining a suitable method to check whether the name of the class starts with "P":

Listing 16: Filter with function (OA5)

```
o.signature().filter(MyClass::owlClassNameStartsWithP)\\.forEach(System.out::println);
/**
* Ignoring inessential code
*/

private static boolean owlClassNameStartsWithP(OWLEntity e) {
        return !e.isBuiltIn()&&e.getIRI().getRemainder().orElse("").startsWith("P");
}
```

This code can be read like this: get the signature stream, then filter it down. The method (`owlClassNameStartsWithP(OWLEntity e)`) that is passed to the `filter()` method returns a boolean, takes exactly one parameter (`OWLEntity`) and checks (1) whether the `OWLEntity` is built-in and then (2) checks whether the remainder, a Java `Optional`, starts with "P". The `orElse(T t)` method returns either the remainder (if the entity has one), or t, which is an object of the class held by the `Optional`, in our example an empty string.

Annotations

There are three important aspects of ontologies we can annotate in OWL: (1) the ontology itself can be annotated for example with provenance information such as authors, date of creation or versioning information; (2) every axiom in the ontology can be annotated (I used this feature in the past for example to give every axiom an id); (3) every entity in the ontology can be annotated, for example with a human readable label (typically, but not necessarily, using `rdfs:label` . Annotations do not have any effect on the logical structure of the ontology, but provide a way to enrich the ontology with meta-data. One of the most prominent use cases for annotations is the use of human-readable labels on classes, properties and individuals to render them in the ontology engineering environment Protégé. In the following, we will show one simple example for each of the three annotation types.

Listing 17: Add Annotation

```
OWLClass student = df.getOWLClass(IRI.create(IOR + "#ID879812719872"));
OWLAnnotation commentAnno = df.getOWLAnnotation(df.getRDFSComment(), df.
    getOWLLiteral("Class representing all Students in the University", "en"));
OWLAnnotation labelAnno = df.getOWLAnnotation(df.getRDFSLabel(), df.
    getOWLLiteral("Student", "en"));
OWLAxiom ax1 = df.getOWLAnnotationAssertionAxiom(student.getIRI(), labelAnno);
man.applyChange(new AddAxiom(ont, ax1));
OWLAxiom ax2 = df.getOWLAnnotationAssertionAxiom(student.getIRI(),
    commentAnno);
```

```
7 man.applyChange(new AddAxiom(ont, ax2));
```

In this example, we can see that annotations are created in quite a similar fashion as *logical* axioms. We first get hold of the components we need to describe the annotation axiom (in the case `ax1`, this is the `OWLClass` we want to annotate, the property we want to annotate with (`rdfs:label` and the actual data value, in our case a string).

Task 4: Building your first complex expression

1. Create a new empty ontology.
2. Add the axiom `Student EquivalentTo: Person and isEnrolledIn some University and attends some Course` .
3. Create a named individual (like yourself, we will call her/him person X from now on), assert her or him as a Person (not Student!), and a named individual ManchesterUniversity, asserted as an instance of the class University.
4. Assert that the person X you just created is enrolled in the ManchesterUniversity. This task requires you to create an object property assertion!
5. Assert that the person you just created attends a course, but without specifying which one. In Manchester syntax that looks like this:

   Individual: Nico

   Types: attendsCourse some Course, Person

   It is a bit tricky to wrap your head around that one, but you have the tools already to succeed at this.
6. Print all axioms in your ontology to check whether they look complete according to the task. You might even want to save the ontology like we discussed in the previous exercise, and inspect the result in Protégé.

In order to complete the task above, you will need to dig deep into the `OWLDataFactory` . The in order to create an existential restriction, you will for example need `datafactory.getOWLObjectIntersectionOf(..)` and `datafactory.getOWLObjectSomeValuesFrom(...)`.

If you did everything as discussed, your output should look something like this (ignoring warnings and the likes):

```
EquivalentClasses(<http:\\owl.org\o#Student>
ObjectIntersectionOf(<http:\\owl.org\o#Person>
ObjectSomeValuesFrom(<http:\\owl.org\o#attends> <http:
owl.org\o#Course>)
ObjectSomeValuesFrom(<http:\\owl.org\ o\#isEnrolledIn>
<http:\\owl.org\o#University>)) )
ClassAssertion(<http:\\owl.org\ o\#Person> <http:\\owl.org\ o\#Nico>)
ClassAssertion(ObjectSomeValuesFrom(<http:\\owl.org\ o\#attends>
<http:\\owl.org\ o\#Course>) <http:\\owl.org\ o\#Nico>)
```

Inference and reasoning

Key Classes:

- OWLReasoner
- OWLReasonerFactory

1. Adding reasoner packages to your project
2. Running the reasoner and inspecting the inferences
3. Appreciating the implementational differences between reasoners

Setting your project up for reasoning

In much the same way as we integrated the OWL API itself with our application using Maven, we will now integrate one of the reasoners that are implementing the `OWLReasoner` interface. In order to achieve that, we will simply inject the following XML snippet into our pom.xml file:

Insert between <dependencies></dependencies> :

<dependency> <groupId>net.sourceforge.owlapi</groupId> <artifactId>org.semanticweb.hermit</artifactId> <version>1.3.8.500</version> </dependency>

This integrates a version of HermiT [3], a popular (and reliable) OWL reasoner, into our project. This version is a special port of HermiT that is compatible with the OWL API version 4 and 5. Note that *HermiT is one of the few reasoners ported so far to OWL API 5.*

[3] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014

If we now update our Maven (if it does not happen all by itself, right click on project in eclipse->Maven->Update project), we will be able to write and execute the following code snippet:

Listing 18: Instantiate Reasoner

```
1 OWLReasonerFactory rf = new ReasonerFactory();
2 OWLReasoner r = rf.createReasoner(o);
3 r.precomputeInferences(InferenceType.CLASS_HIERARCHY);
```

Line 1 first instantiates an `OWLReasonerFactory`, which allows us to create the reasoner (Line 2). Line 3 finally classifies the ontology, which is for some tasks not strictly necessary, but convenient with respect to this introduction.

Querying the reasoner

Through the `OWLReasoner` interface, we can do a number of interesting things. In the following, you will learn who do query for sub-classes, equivalent classes and instances.

One of the most important tasks we ask our reasoners to perform is to classify, and then obtain the sub- and superclasses **inferred** for a particular class. For example, when querying the pizza ontology, we might be interested to query for all `VegetarianTopping` 's or `InterestingPizza` 's. We can obtain the subclasses of a class as follows:

Listing 19: Get all Inferred Subclasses

```
1  OWLOntology o;
2  OWLDataFactory df = man.getOWLDataFactory();
3  IRI pizzaontology = IRI.create("http://protege.stanford.edu/ontologies/pizza/pizza.owl"
       );
4  try {
5          o = man.loadOntology(pizzaontology);
6          OWLReasonerFactory rf = new ReasonerFactory();
7          OWLReasoner r = rf.createReasoner(o);
8          r.precomputeInferences(InferenceType.CLASS_HIERARCHY);
9          r.getSubClasses(df.getOWLClass("http://www.co-ode.org/ontologies/pizza/
               pizza.owl#RealItalianPizza"), false).forEach(System.out::println);;
10 } catch (OWLOntologyCreationException e) {
11         e.printStackTrace();
12 }
```

Line 9 is responsible for the subclass query. The `getSubClasses(..)` method takes as an input to parameters: a named `OWLClass`, such as a kind of Pizza, and a boolean, indicating whether you want only the direct (true) subclasses, or also the indirect ones, i.e. the subclasses of the subclasses of the pizza. If done correctly, you should see the following in your output:

```
Node( <http:\\www.co-ode.org\ontologies\pizza\pizza.owl#Veneziana> )
Node( <http:\\www.co-ode.org\ontologies\pizza\pizza.owl#IceCream>
<http:\\www.co-ode.org\ontologies\pizza\pizza.owl#CheeseyVegetableTopping>
owl:Nothing )
Node( <http:\\www.co-ode.org\ontologies\pizza\pizza.owl#Napoletana> )
```

As we can see, the reasoner returns a set of `Node` objects, each containing a set of classes that are logically equivalent to each other. If you play around with the `Node` object, you will see that there are ways to turn them into lists of `OWLClass` objects. To explain the second node is beyond the scope of this tutorial. Suffice it to say: `owl:Nothing` is a subclass of everything *by definition of the language*, hence HermiT correctly returns it, along with all the unsatisfiable classes in the ontology that are equivalent to `owl:Nothing`.

Task 5: Unsatisfiable Classes

1. Create a new ontology with at least two satisfiable classes and one unsatisfiable class. Tip: the easiest way to do that is to make a class a subclass of the intersection of two disjoint classes. One suggestion (the thing that will be demonstrated later on) is to use a Student and Teacher class, make them disjoint and make a class like Demonstrator a subclass of their intersection.
2. Instantiate a reasoner and determine whether your ontology is consistent. Should it be?
3. Print the list of unsatisfiable classes to the console.
4. Create an individual and make it an instance of the unsatisfiable class in the way we discussed it before.
5. Check whether the ontology is consistent and explain why.

Task 6: Querying Subclasses

1. Instantiate the reasoner again in the same way as we did for the previous task.
2. The question is: Is the person X you created a student or not? Try to find this out using the OWLOntology interface alone. DonâĂŹt spend more than 7 minutes on this and move on.
3. Ask the reasoner the same question (use the OWLReasoner interface).

Bibliography

[1] Birte Glimm, Ian Horrocks, Boris Motik, Giorgos Stoilos, and Zhe Wang. HermiT: An OWL 2 Reasoner. *Journal of Automated Reasoning*, 53(3):245–269, 2014.

[2] Matthew Horridge and Sean Bechhofer. The OWL API: A Java API for OWL Ontologies. *Semantic Web*, 2(1):11–21, 2011.

Index