University of Tübingen
Department of General and Computational Linguistics
Wilhelmstr. 19, 72074 Tübingen, Germany

Tutorial: Java-API to GermaNet

Version 6.0 (April 1, 2011)

Verena Henrich

Acknowledgments go to Marie Hinrichs and Holger Wunsch for their valuable input on both the features and usability of this API.

Table of Contents

1. Introduction

This tutorial is about the Java-API to GermaNet. After an introduction of GermaNet and the API, there is a short overview of the GermaNet XML files (all in subsections of this chapter). Chapter 2 introduces the Java-API to GermaNet by an example tutorial. In the following chapter 3 further methods are explained to finally give a complete overview of the API.

1.1. Basics About GermaNet

GermaNet[1] is a lexical semantic network that partitions the lexical space in a set of concepts that are interlinked with semantic relations. A semantic concept is modeled by a synset (short for *synonymy set*) in GermaNet. A synset is a set of words (called *lexical units*) where all the words are taken to have (almost) the same meaning. Thus a synset is a set-representation of the semantic relation of synonymy.

There are two types of semantic relations in GermaNet: conceptual relations and lexical relations. Conceptual relations hold between two semantic concepts or synsets. They include relations such as hypernymy, part-whole relations, entailment, or causation. Lexical relations hold between two individual lexical units. Antonymy, a pair of opposites, is an example of a lexical relation.

1.2. Java-API to GermaNet

The Java-API to GermaNet represents a programming interface, which means that it provides several methods how GermaNet data can be accessed. The API is located in package `germanet`.

The main class named `GermaNet` serves as a starting point to the API. When a `GermaNet` object is constructed, data is loaded from the GermaNet XML sources. All synsets (class `Synset`) and lexical units (class `LexUnit`) can be obtained through this object, which in turn can be used to examine attributes or find semantic relations, among other things.

This API specifies high-level look-up access to GermaNet data. As it is intended to be a read-only resource, no methods to extend or modify data are provided. All classes and methods are described in the enclosed Java API documentation.

The idea and basic implementation of this API is based on the Java-API to GermaNet by Marie Hinrichs[2]. The main differences between Marie's API and this new version are based on new features in GermaNet, which lead to an adapted and extended XML file design. Marie's API can be used up to version 5.1 of GermaNet (Released 30 April 2008). All later versions of GermaNet should be used with the new version.

Within the Java-API, there is a `GermaNet` class that is a collection of German lexical units (`LexUnit`) organized into synsets (`Synset`). A `GermaNet` object provides methods for

[1] See http://www.sfs.uni-tuebingen.de/GermaNet/

[2] See http://www.sfs.uni-tuebingen.de/GermaNet/GermaNetJavaAPI.zip

retrieving lists of `Synsets` or `LexUnits`, which can be filtered by word category, orthographic form, or some combination.

A `Synset` has a `WordCategory` (adj, nomen, verben) and consists of one or more `LexUnits` and a paraphrase (represented as `Strings`). The list of `LexUnits` for a `Synset` is never empty. A `Synset` object provides methods for retrieving the word category, the paraphrase, and all lexical units as well as methods for retrieving lists of conceptually related synsets.

A `LexUnit` consists of an orthographical form (`orthForms`, represented as a `String`) and has optionally an orthographical variant (`orthVar`), an old orthographical form (`oldOrthForm`) and an old orthographical variant (`oldOrthVar`). Furthermore, a `LexUnit` object can have `Examples` and `Frames`, and it has the following attributes: `sense` (`int`), `source` (`String`), `styleMarking` (`boolean`), `artificial` (`boolean`), and `namedEntity` (`boolean`). A `LexUnit` object provides methods for retrieving any of its properties, as well as methods for retrieving lists of other `LexUnits` lexically related to it.

A `Frame` is simply a container for frame data (`String`).

An `Example` consists of text (`String`) and zero or more `Frames`.

A `ConRel` is a set of possible conceptual relations between `Synsets` (represented as an enum type). A `ConRel` object provides methods for checking if a particular String is a valid conceptual relation, and for determining if a relation is transitive or not. The set consists of the following transitive and non-transitive relations:

- Transitive relations: hypernymy, hyponymy, meronymy, holonymy
- Non-Transitive relations: entailment, entailed, causation, caused, association.

A `LexRel` is a set of possible lexical relations between `LexUnits` (represented as an enum type). A `LexRel` object provides a method for checking if a particular String is a valid lexical relation. Since there is only one transitive lexical relation (synonymy), and no special processing is required by the API to retrieve synonyms, there is no distinction made between transitive and non-transitive lexical relations. The set consists of the following relations:

- synonymy,
- antonymy,
- pertainymy.

A `WordCategory` is a set of possible word categories (represented as an enum type) and contains the values: adj, nomen, verben.

1.3. GermaNet XML Files

The XML files represent the GermaNet data. There are two types of XML files. One type represents all synsets with their lexical units and all other properties. The other type represents all relations, both conceptual and lexical relations.

Synset Files

The XML files contatin all synsets in separated files. These files are named `wordcategory.wordclass.xml`, e.g. `adj.Allgemein.xml`. Each synset starts with a `synset` tag and contatins at least one lexical unit (encoded with the tag `lexUnit`) with its properties and frames and examples.

```
<synsets>
  <synset id="s[0-9]" wordCategory="{adj|nomen|verben}">
    <lexUnit id="l[0-9]" sense="[0-9]" source="STRING"
     namedEntity="{yes|no}" artificial="{yes|no}" styleMarking="{yes|no}">
      <orthForm>STRING</orthForm>
      <orthVar>STRING</orthVar>
      <oldOrthForm>STRING</oldOrthForm>
      <oldOrthVar>STRING</oldOrthVar>
      <example><text>STRING</text><exframe>STRING</exframe></example>
      <frame>STRING</frame>
    </lexUnit>
    <paraphrase>STRING</paraphrase>
  </synset>
  ...
</synsets>
```

Relation Files

The relations are stored within a separate XML file. Both kinds of relations are encoded: conceptual (tag `con_rel`) and lexical (tag `lex_rel`) relations.

```
<relations>
  <con_rel dir="{one|both|revert}" from="s[0-9]" to="s[0-9]"
       name="{has_hypernym|has_component_meronym|causes|is_related_to...}"
       inv="{has_hyponym|is_entailed_by|has_member_meronym...}"/>
  <lex_rel dir="{one|both|revert}" from="l[0-9]" to="l[0-9]"
       name="{has_antonym|has_pertainym|has_participle}"/>
  ...
</relations>
```

2. Tutorial

In this tutorial, we will develop a Java program that makes use of the most important part of the GermaNet-API. Once it is finished, your program will even be useful – it generates a description of a graph that shows a concept and all its hypernyms and hyponyms up to a certain distance from the concept, which is specified by the user. The file `HypernymGraph.java` contains the source code for this tutorial, and is included in the GermaNet distribution.

The final output of the tutorial program will look somewhat like the graph in Figure 1.

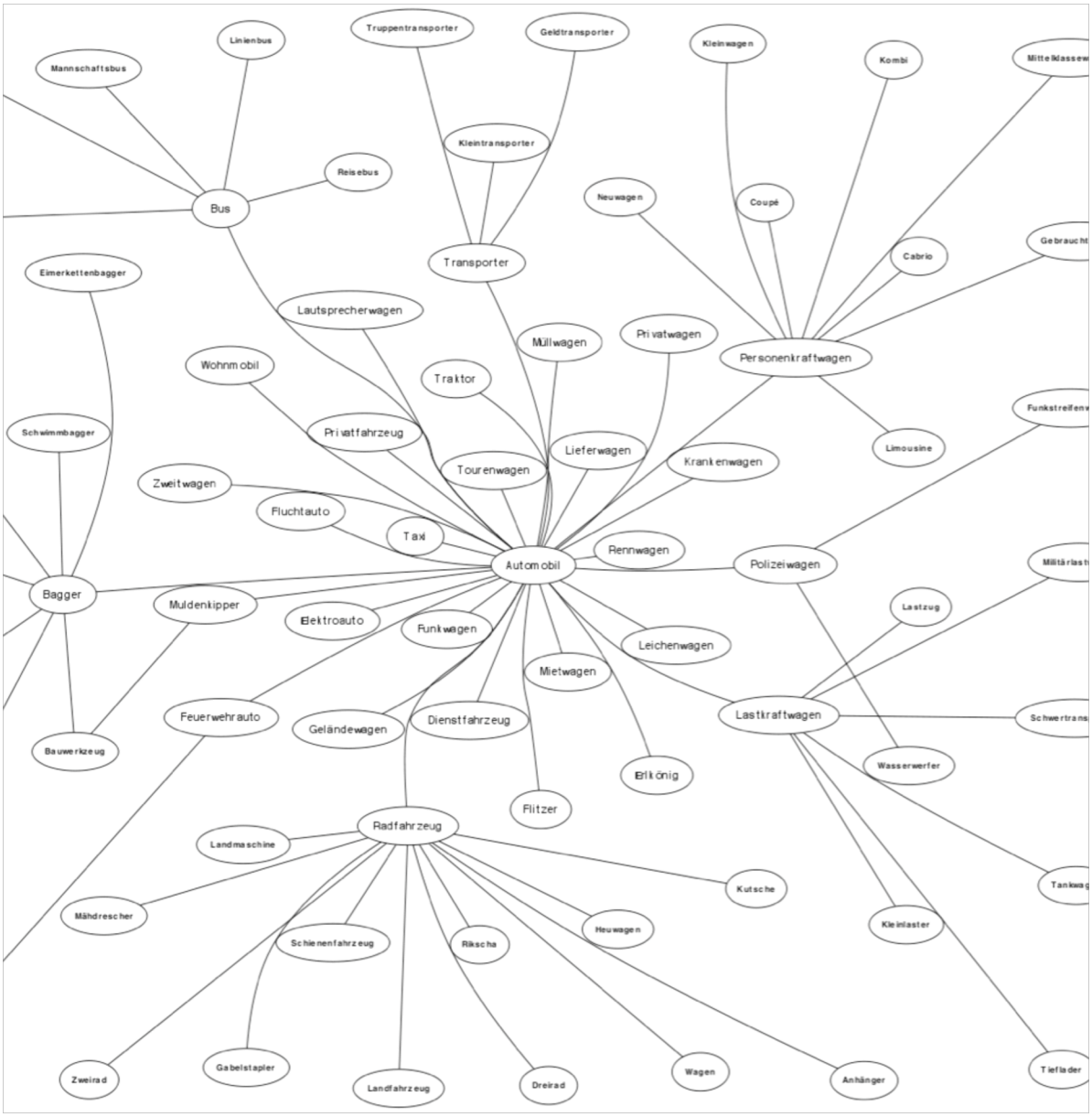

Figure 1: Output of this tutorial

2.1. Before You Start

If you haven't done so already, you will need to obtain:

1. The GermaNet data (unpacked to a directory typically named *GN_Vxx*).
2. The GermaNet Java library, called *GermaNetApi2.0.3.jar*.
3. In order to turn the graph description into an actual image, you will need the GraphViz Tools[3]. Now would be a good time to download and install them.

All of the classes described previously are defined in the package `germanet` within the *GermaNetApi2.0.3.jar* file. You do not need to unpack the jar file.

Classpath

If you are working from the command line, you will need to add *GermaNetApi2.0.3.jar* to your `CLASSPATH` environment variable[4].

If you are working within an IDE (such as NetBeans or Eclipse), add *GermaNetApi2.0.3.jar* to the classpath for any project, which uses GermaNet.

Important Note on Memory

Loading GermaNet requires more memory than the JVM allocates by default. Any application that loads GermaNet will most likely need to be run with JVM options that increase the memory allocated, like this:

```
java -Xms128m -Xmx128m MyApplication
```

These options can be added to your IDE's VM options so that they will be used automatically when your application is run from within the IDE.

Depending on the memory needs of the application itself, the 128's may need to be changed to a higher number. Be careful not to allocate too much memory for the JVM, though, as this may cause other running programs (like your windowing environment) to crash.

2.2. Step 1: Importing Libraries

Before we can create a `GermaNet` object, which loads the XML data and provides methods for looking up synsets and lexical units, we need to import the germanet library and several other necessary libraries.

The box below shows the first lines of the program. If you plan to type the program yourself along with the tutorial, create a file called *HypernymGraph.java*.

[3] The GraphViz Tools are freely available from www.graphviz.org

[4] See http://faq.javaranch.com/java/HowToSetTheClasspath for help with setting your classpath on various operating systems.

```
import germanet.*;
import java.io.*;
import java.util.*;
public class HypernymGraph {
  public static void main(String[] args) {
    // to be filled in...
  }
}
```

2.3. Step 2: Getting User Input

The program needs some information to do its job that the user must supply:

- The word (i.e. orthographic form that represents a lexical unit) whose hypernyms and hyponyms should be displayed (to be accurate, it is not a lexical unit whose relations are to be displayed, but rather the synset that the lexical unit is a member of). In fact, a lexical unit could be a member of more than one synset if it is ambiguous, in which case the program will print the hypernyms and hyponyms for all of the synsets.
- The maximum distance up to which hypernyms and hyponyms are to be displayed.
- The name of the file to write the output to.

```
import germanet.*;
import java.io.*;
import java.util.*;

public class HypernymGraph {
  public static void main(String[] args) {
    Scanner keyboard = new Scanner(System.in);
    String destName;
    File gnetDir;
    String word;
    int depth;
    Writer dest;
    System.out.println("HypernymGraph creates a GraphViz graph " +
            "description of hypernyms and hyponyms of a GermaNet" +
                "concept up to a given depth.");
    System.out.println("Enter <word> <depth> <outputFile> " +
                "[eg: Automobil 2 auto.dot]: ");
    word = keyboard.next();
    depth = keyboard.nextInt();
    destName = keyboard.nextLine().trim();
    // to be continued...
  }
}
```

2.4. Step 3: Create a GermaNet Object

To construct a `GermaNet` object, provide the location of the GermaNet data. This can be done with a `String` representing the path to the directory containing the data, or with a `File` object. Generally speaking, file locations should never be hardcoded, but for the sake of simplicity, this code assumes that the GermaNet data files are in a directory called */germanet/GN_V53*. Please change the line:

```
      gnetDir = new File("/germanet/GN_V53");
```

to reflect the actual location of the GermaNet data files on your computer.

```
import germanet.*;
import java.io.*;
import java.util.*;

public class HypernymGraph {
  public static void main(String[] args) {
    try {
      Scanner keyboard = new Scanner(System.in);
      String destName;
      String word;
      int depth;
      Writer dest;
      System.out.println("HypernymGraph creates a GraphViz graph " +
                  "description of hypernyms and hyponyms of a GermaNet" +
                  "concept up to a given depth.");
      System.out.println("Enter <word> <depth> <outputFile> " +
                  "[eg: Automobil 2 auto.dot]: ");
      word = keyboard.next();
      depth = keyboard.nextInt();
      destName = keyboard.nextLine().trim();
      gnetDir = new File("/germanet/GN_V53");
      GermaNet gnet = new GermaNet(gnetDir);
      // to be continued...
    } catch (Exception ex) {
       ex.printStackTrace();
       System.exit(0);
    }
  }
}
```

Notice that we need to enclose the call to the constructor in a try/catch block. This is because the `GermaNet` object cannot be created if the data files are not found or are corrupted. If something goes wrong, an exception is thrown. We just print the stack trace and exit if this happens.

2.5. Step 4: Finding All Synsets

We can now find all the synsets in GermaNet that the word `orthForm` is a member of. Recall that words may be ambiguous, which means that a word (or lexical unit) may occur in more than one synset.

```
      List<Synset> synsets;
      synsets = gnet.getSynsets(word);
      if (synsets.size() == 0) {
        System.out.println(word + " not found in GermaNet");
        System.exit(0);
      }
      // to be continued...
```

The method `getSynsets(orthForm)`, which is defined in the class `GermaNet`, returns a `List` containing all of the `Synsets` that the word occurs in. If the size of this list is zero, then no synsets were found with a lexical unit containing the orthographic form `orthForm`, and we exit the program.

Each element of the `List synsets` is a `Synset` object. A `Synset` object has methods to retrieve all the lexical units that are members of the synset, and to find out about what other synsets are related to it with respect to a specific kind of conceptual relation. We will use some of the methods that are implemented in the `Synset` class in the next step.

2.6. Step 5: Generating the Hypernym Graph

We are now ready to generate the output, which is first stored in a `String` called `dotCode`, then written to the output file. As mentioned before, our program does not directly create images, but rather textual descriptions of graphs in the GraphViz graph definition language. These can later be turned into images using the GraphViz tools.

```
String dotCode = "";
dotCode += "graph G {\n";
dotCode += "overlap=false\n";
dotCode += "splines=true\n";
dotCode += "orientation=landscape\n";
dotCode += "size=\"13,15\"\n";
HashSet<Synset> visited = new HashSet<Synset>();
for (Synset syn : synsets) {
  dotCode += printHypernyms(syn, depth, visited);
}
dotCode += "}";
dest = new BufferedWriter(new OutputStreamWriter(
                new FileOutputStream(new File(destName)), "UTF-8"));
dest.write(dotCode);
dest.close();
```

The first line of the `dotCode String` opens a GraphViz graph-statement. The following four lines then define the basic layout of the graph. Please refer to the GraphViz manual if you want to find out what exactly these statements do.

The algorithm that traverses the network to find the hypernyms and hyponyms is not very complicated. It works as follows:

- Start with a `Synset` that the lexical unit the user requested is a member of (called `Synset syn`). This becomes the center node of the graph.
- Look up all hypernyms of `syn` and add them to the graph as neighbor nodes of `syn`.
- Look up all hyponyms of `syn` and add them to the graph also.
- For each hypernym and hyponym found, recursively find and add their hypernyms and hyponyms to the graph, up to the maximum distance to the center node, as specified by the user.

To sum up, the algorithm finds all hypernyms and hyponyms of a given `Synset syn`, adds them to the graph, and then in turn does exactly the same it did with `syn` with all of its hypernyms and hyponyms.

There is one catch, however, that we must pay attention to: Assume the algorithm looks at some `Synset s`. It finds all hypernyms of `s` and adds them to the graph. Then it recursively repeats all its steps for each hypernym `h` it found: That is, it first finds all hypernyms of `s`, then it finds all hyponyms of `h`. At this point, we must be careful, since the `Synset s` the algorithm looked at in the previous recursive step is, of course, a hyponym of `h`! We must make sure that the algorithm does not consider `Synsets` it already looked at over and over

again. In our program, we use the `HashSet visited` for this: For each `Synset` the algorithm finds, we add the `Synset` to the visited set. Any `Synset` that is in the visited set is not considered any further by the algorithm in subsequent recursive steps.

The program proceeds by calling the static `printHypernyms()` method for each `Synset` in the `synsets` list. In the next step, we will turn to `printHypernyms()`, which is the implementation of the recursive algorithm sketched above.

We then finish up by adding a closing brace to the GraphViz description, write the code to the output file, and close the file.

2.7. Step 6: Recursively Printing Hypernyms and Hyponyms

The `printHypernyms()` method, which recursively adds all hypernyms and hyponyms of a synset to the hypernym graph, expects three arguments:

- The synset whose hypernyms and hyponyms are to be added next (the argument `synset`)
- The remaining distance from the center node of the graph to the last hypernym or hyponym to be added (argument `depth`)
- The set of synsets already visited (argument `visited`)

```
  static String printHypernyms(Synset synset, int depth,
                               HashSet<Synset> visited) {
```

Now declare the variables we will need later:

```
    String rval = "";
    List<LexUnit> lexUnits;
    String orthForm = "";
    List<Synset> hypernyms = new ArrayList<Synset>();
    List<Synset> relations;
    String hypOrthForm;
    visited.add(synset);
    // to be continued...
  }
```

The synset is added to the `visited` set (to make sure the algorithm does not run in an infinite loop; see step 4).

We have already seen that the GermaNet-API contains a special class, `Synset`, that represents the properties of a synset. There is also a class `LexUnit` that represents the properties of a lexical unit. Both classes provide methods to obtain information about other objects in GermaNet the synset or lexical unit is related to. A lexical unit may contain multiple orthographic forms (i.e. `orthForm` (main orthographic form), `orthVar` (a variant of the main form), `oldOrthForm`, (main orthographic form in the old German orthography), and `oldOrthVar` (a variant of the old form)), which represent different spellings of the same word. If there are several spellings of a word, for example *Schloß* and *Schloss* in the old and new German spelling, *Schloss* represents `orthForm` and *Schloß* represents `oldOrthForm`.

We will use the main orthographical form of the `LexUnit` that is first returned by `synset.getLexUnits()` as a representative for the concept the `Synset` represents. So we must first retrieve all lexical units that are a member of the synset:

```
    lexUnits = synset.getLexUnits();
```

As you can see, this works very much the same as retrieving all synsets in GermaNet. `getLexUnits()`, which is a method of the `Synset` class, returns a `List` of `LexUnit` objects.

We now fetch the first orthographic form of the first `LexUnit` and add it to the graph description, along with some formatting information:

```
orthForm = lexUnits.get(0).getOrthForm();
rval += "\"" + orthForm + "\" [fontname=Helvetica,fontsize=10]\n";
```

Again, you can see that the way orthographic forms are retrieved is extremely similar to the way synsets and lexical units are accessed. Of course, since orthographic forms are plain strings, the `List` returned is of type `String`.

It is now time to collect all hypernyms and hyponyms and add them to the graph. Since we will make no difference in the graphical output between hypernyms and hyponyms we will store them (a little sloppily) in one list called hypernyms.

```
relations = synset.getRelatedSynsets(ConRel.has_hypernym);
hypernyms.addAll(relations);
relations = synset.getRelatedSynsets(ConRel.has_hyponym);
hypernyms.addAll(relations);
```

`ConRel` is an `enum` class defined in GermaNet. Enums are special constructs in Java for storing constants. The `ConRel` class provides a way of telling the `getRelatedSynsets(conRel)` method which relation is being requested so that an invalid relation cannot be requested.

`ConRel.has_hypernym` and `ConRel.has_hyponym` are conceptual relations that apply between synsets. The complete list of conceptual realations are: hypernymy, hyponymy, meronymy, holonymy, entailment, entailed, causation, caused, and association.

Similarly, the `LexUnit` class contains a `getRelatedLexUnits(lexRel)` method which accepts a `LexRel` object as a parameter.

```
01     for (Synset syn : hypernyms) {
02       if (!visited.contains(syn)) {
03         hypOrthForm = syn.getLexUnits().get(0).getOrthForm();
04         rval += "\"" + orthForm + "\" -- \"" + hypOrthForm + "\";\n";
05
06         if (depth > 1) {
07           rval += printHypernyms(syn, depth - 1, visited);
08         } else {
09           rval += "\"" + hypOrthForm +
10                   "\"[fontname=Helvetica,fontsize=8]\n";
11         }
12       }
13     }
14     // return the graph string generated
15     return rval;
```

For each hypernym and hyponym we found, we first check if we have visited it before (line 2). If so, we skip it. Otherwise, we fetch the first orthographic form of the first lexical unit (line 3) and use it in line 4 to add an edge to the graph description between the node that represents the current synset and the node that represents the hypernym or hyponym (edges in GraphViz syntax are expressed by two node labels that are separated by --).

If the maximum distance to the center node has not yet been reached (line 6), we add the hypernyms and hyponyms of the current hypernym or hyponym by recursively calling

`printHypernyms()` with a decremented depth. Otherwise, we add some formatting information for the hypernym or hyponym node.

2.8. Step 7: Trying It Out

This is it! We are now ready to test our program. Compile the source code using Java JDK 6.0 or above:

```
javac HypernymGraph.java
```

Then run the program:

```
java –Xms256m –Xmx256m HypernymGraph
```

Let's create a graph that shows the concept *Automobil* in the center and the hypernyms and hyponyms up to a distance of two. When asked to enter the data, type `Automobil 2 auto.dot`:

HypernymGraph creates a GraphViz graph description of hypernyms and hyponyms of a GermaNet concept up to a given depth.

```
Enter <word> <depth> <outputFile> [eg: Automobil 2 auto.dot]:
Automobil 2 auto.dot
```

This creates the graph description file *auto.dot* in the current working directory. The first few lines should look like this:

```
graph G {
overlap=false
splines=true
orientation=landscape
size="13,15"
"Automobil" [fontname=Helvetica,fontsize=10]
"Automobil" -- "Muldenkipper";
"Muldenkipper" [fontname=Helvetica,fontsize=10]
"Muldenkipper" -- "Bauwerkzeug";
"Bauwerkzeug" [fontname=Helvetica,fontsize=8]
"Automobil" -- "Bagger";
...
```

We can now use one of the GraphViz tools to create a visual representation of the graph from the graph description file in a PNG file called *auto.png*:

```
neato –Tpng auto.dot –o auto.png
```

The GraphViz tools provide many more output formats and ways of influencing the layout of the graph, which are described in the GraphViz manuals[5].

This finishes the tutorial. Please see the GermaNet javadoc documentation, viewable in your web browser, for a complete list of methods, including descriptions, available for each class within the `germanet` package.

[5] See www.graphviz.org

3. Code Snippets and Samples

This section contains code snippets and samples that demonstrate how to use the GermaNet library objects and their methods.

3.1. Creating a GermaNet Object

Before you can construct a `GermaNet` object, you need to make sure that the *GermaNetApi2.0.3.jar* file is on your classpath, then import the library:

```
import germanet.api.*;
```

When a `GermaNet` object is created, it needs to know where to find the XML-formatted GermaNet data files. The location of the directory containing the data files is sent as a parameter to the `GermaNet` constructor either as a `String` object:

```
GermaNet gnet = new GermaNet("/germanet/GN_V53/");
```

or a `File` object:

```
File gnetDir = new File("/germanet/GN_V53");
GermaNet gnet = new GermaNet(gnetDir);
```

To ignore case when getting `Synsets` and `LexUnits`, set the `ignoreCase` flag in the constructor:

```
GermaNet gnet = new GermaNet("/germanet/GN_V53/", true);
```

or:

```
File gnetDir = new File("/germanet/GN_V53");
GermaNet gnet = new GermaNet(gnetDir, true);
```

Unless otherwise stated in the javadoc documentation, all methods in all objects will return an empty `List` rather than `null` to indicate that no objects exist for a given request.

3.2. Getting Synsets from a GermaNet Object

A `Synset` has a `WordCategory` (i.e. adj, nomen, verben), a paraphrase (represented as a `String`), and a `List` of `LexUnits`. The `List` of `LexUnits` for a `Synset` is never empty.

A `Synset` object provides methods for retrieving any of its properties as well as methods for retrieving `Lists` of other `Synsets` conceptually related to it. Once you have constructed a `GermaNet` object (called `gnet` in the examples below), you can retrieve `Lists` of `Synsets`, using orthographical form or word category filtering, if desired.

Get a `List` of all `Synsets`:

```
List<Synset> allSynsets = gnet.getSynsets();
```

Get a `List` of all `Synsets` containing a lexical unit with `orthForm` Bank (Note: if `gnet` was constructed with the `ignoreCase` flag set, then the following method call will return the same list with parameters such as *bank*, *BANK* or *BaNK*):

```
List<Synset> synList = gnet.getSynsets("Bank");
```

Get a `List` of all `Synsets` with word category adjective (`WordCategory.adj`, other options are `WordCategory.nomen` and `WordCategory.verben`):

```
List<Synset> adjSynsets = gnet.getSynsets(WordCategory.adj);
```

3.3. Working with Synsets

Once you have obtained a `List` of `Synsets`, you can start processing them. A `Synset` object has methods for retrieving its word category, lexical units (or just the orthographic forms of the lexical units), and paraphrases, as well as methods for retrieving synsets that are related to it.

To get a synset's word category and do further processing in case of an adjective:

```
WordCategory wc = aSynset.getWordCategory();
if (wc == WordCategory.adj) {
    // do something
}
```

Retrieving the paraphrase is done in a similar way:

```
String paraphrase = aSynset.getParaphrase();
```

To get a synset's orthographic forms (retrieves a `List` of all orthographic forms in all the `LexUnits` that belong to this `Synset`):

```
List<String> orthForms = aSynset.getAllOrthForms();
```

To get a list of all lexical units of a synset and iterate through them:

```
List<LexUnit> lexList = aSynset.getLexUnits();
for (LexUnit lu : lexList) {
    // process lexical unit
}
```

Suppose you want to find all of the member meronyms of a synset:

```
List<Synset> meronyms =
aSynset.getRelatedSynsets(ConRel.has_member_meronym);
```

Sometimes you may have a conceptual relationship represented as a `String`. The following code can be used to validate the `String` and retrieve the relations:

```
String aRel = "has_hypernym";
List<Synset> relList;
if (ConRel.isRel(aRel)) {// make sure aRel is a valid conceptual relation
    relList = aSynset.getRelatedSynsets(ConRel.valueOf(aRel));
}
```

The following are all valid calls to `getRelatedSynsets()`:

```
aSynset.getRelatedSynsets(ConRel.has_hypernym);
aSynset.getRelatedSynsets(ConRel.has_hyponym);
aSynset.getRelatedSynsets(ConRel.has_component_meronymy);
aSynset.getRelatedSynsets(ConRel.has_component_holonymy);
aSynset.getRelatedSynsets(ConRel.has_member_meronymy);
aSynset.getRelatedSynsets(ConRel.has_member_holonymy);
aSynset.getRelatedSynsets(ConRel.has_substance_meronymy);
```

```
aSynset.getRelatedSynsets(ConRel.has_substance_holonymy);
aSynset.getRelatedSynsets(ConRel.has_portion_meronymy);
aSynset.getRelatedSynsets(ConRel.has_portion_holonymy);
aSynset.getRelatedSynsets(ConRel.is_related_to;
aSynset.getRelatedSynsets(ConRel.causes);
aSynset.getRelatedSynsets(ConRel.entails);
aSynset.getRelatedSynsets(ConRel.is_entailed_by); // and so on...
```

Suppose you are not interested in any particular relation, but want a `List` of all `Synsets` that are related to `aSynset` in any way:

```
List<Synset> allRelations = aSynset.getRelatedSynsets();
```

For transitive relations (hypernymy, hyponymy, meronymy, holonymy), there is a method that retrieves a `List` of `Lists` of `Synsets`, where the `List` at position 0 contains the originating `Synset`, the `List` at position 1 contains the relations at depth 1, the `List` at position 2 contains the relations at depth 2, and so on up to the maximum depth. Using this data structure, some information cannot be included – namely, for any synset at depth n, you cannot determine which synset at depth n-1 it is a relation of. Nonetheless, you may find the method useful.

The following code prints the orthographic forms of each synset at every depth of the hyponyms of *Decke*:

```
List<List<Synset>> transHyponyms;
synList = gnet.getSynsets("Decke");
String spaces;
for (Synset s : synList) {
    spaces = "";
    transHyponyms = s.getTransRelatatedSynsets(ConRel.has_hyponym);
    for (List<Synset> listAtDepth : transHyponyms) {
        for (Synset synAtDepth : listAtDepth) {
            System.out.println(spaces + synAtDepth.getAllOrthForms());
        }
        spaces += "     ";
    }
}
```

Two `Synsets` are found containing the `orthForm` *Decke*. For each of them, we retrieve the hyponyms using the `getTransRelatedSynsets()` method, store the result in the `List` of `Lists` of `Synsets` called `transHyponyms`, and then print `transHyponyms`. The output looks like this:

```
[Decke]
     [Bettdecke]
     [Wolldecke]
     [Kuscheldecke]
     [Altardecke]
     [Satteldecke]
     [Plane]
     [Löschdecke]
          [Plastikplane]
[Decke, Zimmerdecke]
     [Kuppel]
     [Beleuchtungsdecke]
     [Hängedecke]
     [Stuckdecke]
          [Zirkuskuppel]
```

3.4. Getting LexUnits from a GermaNet Object

A `LexUnit` may consist of multiple orthographic forms (represented as a `String`), which represent different spellings of the same word:

- The main orthographic form `orthForm` is always set.
- A variant of the main form `orthVar` (optional).
- The main orthographic form in the old German orthography `oldOrthForm` (optional).
- A variant of the old form `oldOrthVar`.

If there are several spellings of a word, for example *Schloß* and *Schloss* in the old and new German spelling, *Schloss* represents `orthForm` and *Schloß* represents `oldOrthForm`.

Lexical units can have `Frames` and `Examples`. Further attributes of a `LexUnit` are the following: `styleMarking` (`boolean`), `sense` (`int`), `orthVar` (`boolean`), `artificial` (`boolean`), `namedEntity` (`boolean`), and `source` (`String`). A `LexUnit` object provides methods for retrieving any of its properties, as well as methods for retrieving `Lists` of other `LexUnits` lexically related to it. Once you have constructed a `GermaNet` object (called `gnet` in the examples below), you can retrieve `Lists` of `LexUnits`, using orthographic form or word category filtering, if desired.

Get a `List` of all `LexUnits`:

```
List<LexUnit> allLexUnits = gnet.getLexUnits();
```

Get a `List` of all `LexUnits` with `orthForm` *Bank* (Note: if `gnet` was constructed with the `ignoreCase` flag set, then the following method call will return the same list with parameters such as *bank*, *BANK* or *BaNK*):

```
List<LexUnit> lexList = gnet.getLexUnits("Bank");
```

Get a `List` of all `LexUnits` with word category nomen (`WordCategory.nomen`, other options are `WordCategory.adj` and `WordCategory.verben`):

```
List<LexUnit> nomLexUnits = gnet.getLexUnits(WordCategory.nomen);
```

3.5. Working with LexUnits

Once you have obtained a `List` of `LexUnits`, you can start processing them. A `LexUnit` object has methods for retrieving its `WordCategory`, `Synset`, orthographic forms (`orthForm`, `orthVar`, `oldOrthForm`, and `oldOrthVar`), and further attributes including word sense number (`sense`), `source`, `namedEntity`, `artificial`, and `styleMarking`, as well as methods for retrieving `LexUnits` that are lexically related to it.

To get the word category of a lexical unit and do further processing in case of a verb:

```
WordCategory wc = aLexUnit.getWordCategory();
if (wc == WordCategory.verben) {
    // do something
}
```

To get the orthographic forms of a lexical unit:

```
List<String> orthForms = aLexUnit.getOrthForms();
```

You may prefer to retrieve the main orthographic form:

```
String orthForm = aLexUnit.getOrthForm();
```

You may prefer to just retrieve the variant of the main orthographic form:

```
String orthVar = aLexUnit.getOrthVar();
```

To get the main orthographic form in the old German orthography:

```
String oldOrthForm = aLexUnit.getOldOrthForm();
```

Retrieving the variant of the old orthographic form:

```
String oldOrthVar = aLexUnit.getOldOrthVar();
```

Suppose you want to generate a `List` of `LexUnits` with word category *nomen*, but you are not interested in named entities or artificial nouns. You could generate such a `List` with the following code (note that we use a real `Iterator` object here instead of just a simple for-loop because it is the only safe way to remove elements from a `List` while iterating):

```
List<LexUnit> lexList = gnet.getLexUnits(WordCategory.nomen);
LexUnit aLexUnit;
Iterator<LexUnit> iter = lexList.iterator();
while (iter.hasNext()) {
    aLexUnit = iter.next();
    if (aLexUnit.isNamedEntity() || aLexUnit.isArtificial()) {
        iter.remove();
    }
}
// ... process lexList ...
```

Suppose you want to find all of the antonyms of a `LexUnit`:

```
List<LexUnit> antonyms = aLexUnit.getRelatedLexUnits(LexRel.has_antonym);
```

Sometimes you may have a lexical relationship represented as a `String`. The following code can be used to validate the `String` and retrieve the relations:

```
String aRel = "has_antonym";
List<LexUnit> relList;
if (LexRel.isRel(aRel)) {// make sure aRel is a valid lexical relation
    relList = aLexUnit.getRelatedLexUnits(LexRel.valueOf(aRel));
}
```

The following are all valid calls to `getRelatedLexUnits()`:

```
aLexUnit.getRelatedLexUnits(LexRel.has_synonym);
aLexUnit.getRelatedLexUnits(LexRel.has_antonym);
aLexUnit.getRelatedLexUnits(LexRel.has_pertainym); // and so on ...
```

Suppose you are not interested in any particular relation, but want a `List` of all `LexUnits` that are related to aLexUnit in any way:

```
List<LexUnit> allRelations = aLexUnit.getRelatedLexUnits();
```

Finding the `Examples` and `Frames` is done as follows:

```
List<Example> exList = aSynset.getExamples();
List<Frame> frameList = aSynset.getFrames();
```

3.6. Working with Frames and Examples

A `Frame` is simply a container for frame data, which can be retrieved with the `getData()` method. `Frames` occur in two contexts within GermaNet:

1. A `List` of `Frames` may be present within a `Synset` object. You could print the `orthForms` of verb `Synsets` containing a `Frame` that begins with *NN* like this:

```
synList = gnet.getSynsets(WordClass.verben);
List<Frame> frameList;
boolean printIt;
for (Synset syn : synList) {
  printIt = false;
  frameList = syn.getFrames();
  for (Frame f : frameList) {
    if (f.getData().startsWith("NN")) {
      printIt = true;
    }
  }
  if (printIt) {
    System.out.println(syn.getAllOrthForms());
  }
}
```

2. A `List` of `Frames` may be present within an `Example` (which in turn is part of a `Synset`). We could print the `Examples` with `Frames` containing the substring *AN* of verb `Synsets` with the following code:

```
synList = gnet.getSynsets(WordClass.verben);
List<Example> exList;
List<Frame> frameList;
for (Synset syn : synList) {
  exList = syn.getExamples();
  for (Example ex : exList) {
    frameList = ex.getFrames();
    for (Frame f : frameList) {
      if (f.getData().contains("AN")) {
        System.out.println(f.getData() + " : " +ex.getText());
      }
    }
  }
}
```