University of Tuebingen
Department of Computational Linguistics
Wilhelmstr. 19
72074 Tuebingen, Germany

Java[1]-API to GermaNet, Version 1.0

Marie Hinrichs

University of Tuebingen

1 This document, as well as the API itself, is based on the Perl-API to GermaNet by Holger Wunsch: http://www.sfs.uni-tuebingen.de/~wunsch/germanet_api.html

Manual Version 1.0 (31 July 2008)

Acknowledgments go to Pilku Gupta, Lothar Lemnitzer, Holger Wunsch, and Thomas Zastrow for their valuable input on both the features and usability of this API.

What is GermaNet?

GermaNet 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** 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. The term synset means "set of synonyms".
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 hyperonymy, 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.

Java API to GermaNet

The Java API to GermaNet provides high-level look-up access to GermaNet data for use within Java programs. When a GermaNet object is constructed, data is loaded from the GermaNet XML sources. This object can then be used to extract Synsets and LexUnits, which in turn can be used to examine attributes or find semantic relations, among other things.

This API is intended as a read-only resource: no public methods are provided for changing or adding data.

GermaNet Data Organization and Objects

This section describes the various types of objects contained in the GermaNet API.

GermaNet is a collection of German lexical units (LexUnits) organized into sets of synonyms (Synsets). A GermaNet object provides methods for retrieving Lists of Synsets or LexUnits which can be filtered by word class, orthographic form, or some combination.

A **Synset** has a WordClass (adj, nomen, verben) and consists of Lists of LexUnits, Frames, paraphrases (represented as Strings), and Examples. The List of LexUnits for a Synset is never empty, but any of the others may be. 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.

A **LexUnit** consists of one or more orthForms (represented as a List of Strings), and has the following attributes: markedStyle (boolean), sense (int), orthVar (boolean), artificial (boolean), properName (boolean), status (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.

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

An **Example** consists of text (String) and 0 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:	hyperonymy, 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, pertonymy,
arg1, arg1_pred, arg2, arg2_pred, participleOf

A **WordClass** is a set of possible word classes (represented as an enum type) and contains the values:
adj, nomen, verben

Tutorial

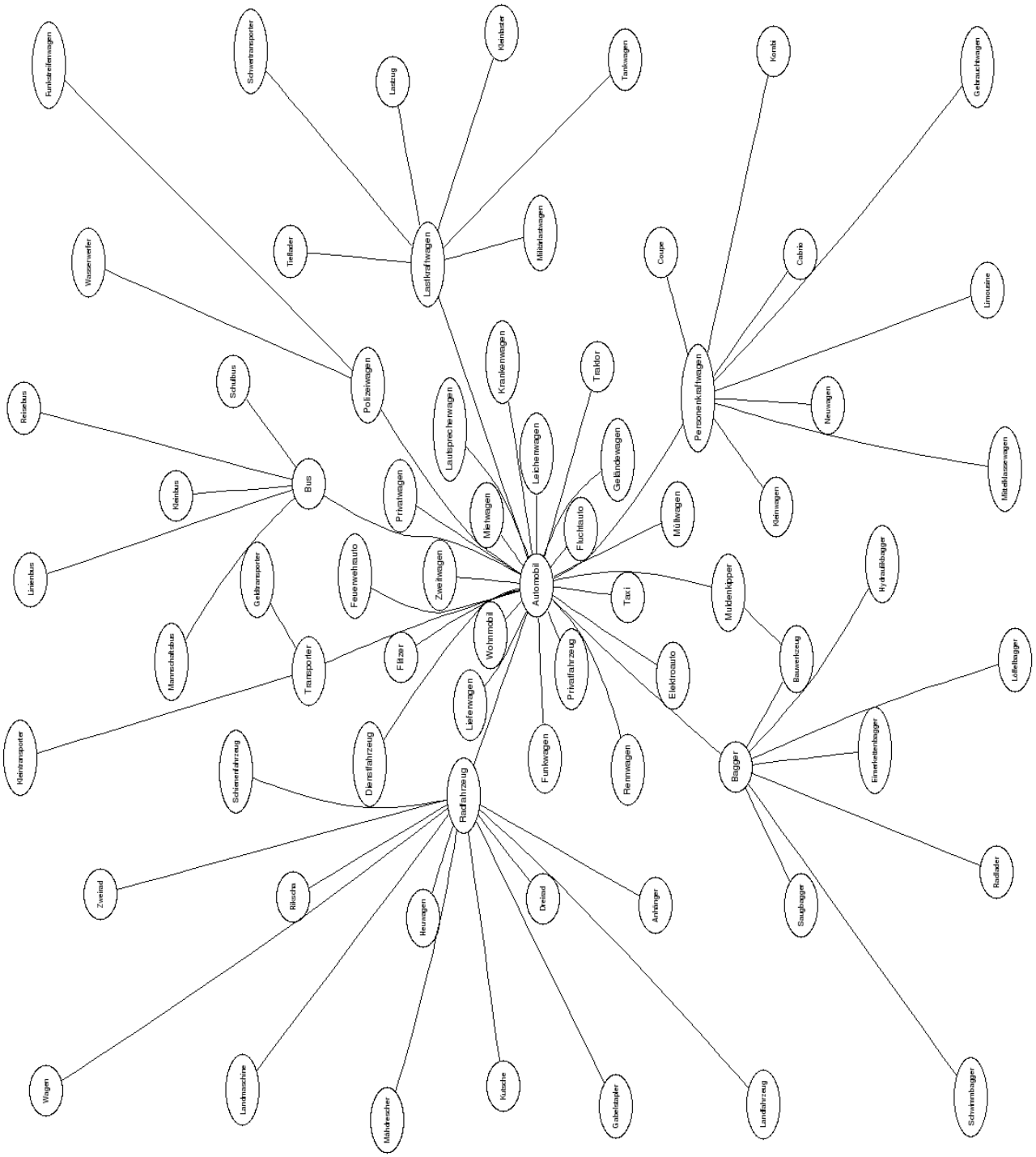
In this tutorial, we will develop a Java program that makes use of the most important part of the GermaNet-API. Once it is finished, your program will even be useful – it generates a description of a graph that shows a concept and all its hyperonyms and hypernyms up to a certain distance from the concept,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

The final output will look somewhat like the graph on the next page.

Obtaining the GraphViz Tools

In order to turn the graph description into an actual image, you will need the GraphViz Tools, which are freely available on the web. Now would be a good time to download and install them from the GraphViz website at www.graphviz.org

The output of the tutorial program:



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 V*xx*_UTF)
2. The GermaNet java library, called GermaNet1.0.jar

All of the classes described previously are defined in the package *germanet* within the GermaNet1.0.jar file. You do not need to unpack the jar file.

If you are working from the command line, you will need to add GermaNet1.0.jar to your CLASSPATH environment variable. See http://faq.javaranch.com/java/HowToSetTheClasspath for help with setting your classpath on various operating systems.

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

Important Note:

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.

Step 1: Importing Libraries

Before we can create a GermaNet object, which loads the XML data and provides methods for looking up Synsets and LexUnits, 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 HyperonymGraph.java.

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

public class HyperonymGraph {
  public static void main(String[] args) {

    // to be filled in...

  }
}
```

Step 2: Getting User Input

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

- The word whose hyperonyms and hyponyms should be displayed (to be accurate, it is not a *word* whose relations are to be displayed, but rather the *synset* that the word is a member of. In fact, a word could be a member of more than one synset if it is ambiguous, in which case the program will print the hyperonyms and hyponyms for all of the synsets.
- The maximum distance up to which hyperonyms 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 HyperonymGraph {
  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("HyperonymGraph creates a GraphViz graph " +
                "description of hyperonyms 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...
  }
}
```

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/V51_UTF. Please change the line:

```
      gnetDir = new File("/germanet/V51_UTF");
```

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

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

public class HyperonymGraph {
  public static void main(String[] args) {
```

```
    try {
      Scanner keyboard = new Scanner(System.in);
      String destName;
      String word;
      int depth;
      Writer dest;

      System.out.println("HyperonymGraph creates a GraphViz graph " +
                  "description of hyperonyms 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/V51_UTF");
      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.

Step 4: Finding All Synsets

We can now find all the synsets in GermaNet that the word `word` 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 `get_synsets`, 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 LexUnit containing the orthForm `word`, 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.

Step 5: Generating the Hyperonym 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 += printHyperonyms(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 hyperonyms and hyponyms is not very complicated. It works as follows:

- Start with a synset that the word the user requested is a member of (called Synset `syn`). This becomes the center node of the graph.
- Look up all hyperonyms 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 hyperonym and hyponym found, recursively find and add their hyperonyms 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 hyperonyms 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 hyperonyms and hyponyms.
There is one catch, however, that we must pay attention to: Assume the algorithm looks at some synset `s`. It finds all hyperonyms of `s` and adds them to the graph. Then it recursively repeats all its steps for each hyperonym `h` it found: That is, it first finds all hyperonyms of `h`, 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 doesn't 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 `printHyperonyms` method for each synset in the `synsets` list. In the next step, we will turn to `printHyperonyms`, 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.

Step 6: Recursively Printing Hyperonyms and Hyponyms

The `printHyperonyms` method, which recursively adds all hyperonyms and hyponyms of a synset to the hyperonym graph, expects three arguments:

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

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

Now declare the variables we'll need later:

```
    String rval = "";
    List<LexUnit> lexUnits;
    String orthForm = "";
    List<Synset> hyperonyms = 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*, which represent different spellings of the same word. In the current version of GermaNet, however, a lexical unit never contains more than one orthographic form. If there are in fact several spellings of the same word (such as Schloß vs Schloss in the old and new German spelling), two lexical units exist instead in a synset.
We will use the first lexical form in a synset 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(0);
      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 hyperonyms and hyponyms and add them to the graph. Since we will make no difference in the graphical output between hyperonyms and hyponyms we will store them (a little sloppily) in one list called `hyperonyms`.

```
    relations = synset.getRelations(ConRel.hyperonymy);
    hyperonyms.addAll(relations);
    relations = synset.getRelations(ConRel.hyponymy);
    hyperonyms.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 `getRelations` method which relation is being requested so that an invalid relation cannot be requested.
`ConRel.hyperonymy` and `ConRel.hyponymy` are conceptual relations that apply between synsets. The complete list of conceptual realations are: hyperonymy, hyponymy, meronymy, holonymy, entailment, entailed, causation, caused, and association.
Similarly, the `LexUnit` class contains a `getRelations` method which accepts a `LexRel` object as a parameter.

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

For each hyperonym 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 hyperonym 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 hyperonyms and hyponyms of the current hyperonym or hyponym by recursively calling `printHyperonyms` with a decremented `depth`. Otherwise, we add some formatting information for the hyperonym or hyponym node.

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 HyperonymGraph.java
```

Then run the program:

```
java -Xms128m -Xmx128m HyperonymGraph
```

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

```
HyperonymGraph creates a GraphViz graph description of hyperonyms and hyponyms
of a GermaNetconcept 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.
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.

Code Snippets and Samples

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

Creating a GermaNet object

Before you can contruct a GermaNet object, you need to make sure that the GermaNet1.0.jar file is on your classpath, then import the library:

```
import germanet.*;
```

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 or a File object:

```
GermaNet gnet = new GermaNet("/germanet/V51_UTF/");
```
or:
```
File gnetDir = new File("/germanet/V51_UTF");
GermaNet gnet = new GermaNet(gnetDir);
```

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.

Getting Synsets from a GermaNet object

A Synset has a WordClass (adj, nomen, verben) and consists of Lists of LexUnits, Frames, paraphrases (represented as Strings), and Examples. The List of LexUnits for a Synset is never empty, but any of the others may be. 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 orthForm or WordClass 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 "gehen":
```
List<Synset> synList = gnet.getSynsets("gehen");
```

Get a List of all Synsets with a word class of adj (other options are nomen and verben):
```
List<Synset> adjSynsets = gnet.getSynsets(WordClass.adj);
```

Getting LexUnits from a GermaNet object

A LexUnit consists of one or more orthForms (represented as a List of Strings), and has the following attributes: markedStyle (boolean), sense (int), orthVar (boolean), artificial (boolean), properName (boolean), status (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 orthForm or WordClass filtering, if desired.

Get a List of all LexUnits:
```
List<LexUnit> allLexUnits = gnet.getLexUnits();
```

Get a List of all LexUnits with orthForm "Bank":
```
List<LexUnit> lexList = gnet.getLexUnits("Bank");
```

Get a List of all LexUnits with a word class of nomen (other options are adj and verben):
```
List<LexUnit> nomLexUnits = gnet.getLexUnits(WordClass.nomen);
```

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 class, LexUnits (or just the orthographic forms of the LexUnits), Exmaples, Frames, and paraphrases, as well as methods for retrieving Synsets that are related to it.

To get a Synset's word class:
```
WordClass wc = aSynset.getWordClass();
if (wc == WordClass.adj) {
    // do something
}
```

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

Finding the Examples, Frames, and paraphrases is done in a similar way:
```
List<Example> exList = aSynset.getExamples();
List<Frame> frameList = aSynset.getFrames();
List<String> paraphraseList = aSynset.getParaphrases();
```

Suppose you want to find all of the meronyms of a Synset:
```
List<Synset> meronyms = aSynset.getRelations(ConRel.meronymy);
```

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 = ”hyperonymy”;
List<Synset> relList;
if (ConRel.isRel(aRel)) {// make sure aRel is a valid conceptual relation
    relList = aSynset.getRelations(ConRel.valueOf(aRel));
}
```

The following are all valid calls to `getRelations`:
```
aSynset.getRelations(ConRel.hyperonymy);
aSynset.getRelations(ConRel.hyponymy);
aSynset.getRelations(ConRel.meronymy);
aSynset.getRelations(ConRel.holonymy);
```

```
aSynset.getRelations(ConRel.association;
aSynset.getRelations(ConRel.causation);
aSynset.getRelations(ConRel.caused);
aSynset.getRelations(ConRel.entailed);
aSynset.getRelations(ConRel.entailment);
aSynset.getRelations(ConRel.valueOf(“hyperonymy”));
aSynset.getRelations(ConRel.valueOf(“hyponymy”)); // 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.getRelations();
```

For transitive relations (hyperonymy,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 can't 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.getTransRelations(ConRel.hyponymy);
    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 `getTransRelations` 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]
```

Working with LexUnits

Once you have obtained a List of LexUnits, you can start processing them. A LexUnit object has methods for retrieving its word class, parent Synset, orthographic forms, attributes (including word sense number, status, properName, orthVar, artificial, and markedStyle), as well as methods for retrieving LexUnits that are related to it.

To get a LexUnit's word class:

```
WordClass wc = aLexUnit.getWordClass();
if (wc == WordClass.verben) {
    // do something
}
```

To get a LexUnit's orthographic forms:

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

Since the current versin of the germanet data does not contain any LexUnits with multiple orthForms, you may prefer to just retrieve the first orthForm:

```
String orthForm = aLexUnit.getOrthForm(0);
```

Suppose you want to generate a List of LexUnits with wordclass nomen, but you are not interested in proper nouns 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(WordClass.nomen);
LexUnit aLexUnit;
Iterator<LexUnit> iter = lexList.iterator();
while (iter.hasNext()) {
    aLexUnit = iter.next();
    if (aLexUnit.isProperName() || aLexUnit.isArtificial()) {
        iter.remove();
    }
}
// ... process lexList ...
```

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

```
List<LexUnit> antonyms = aLexUnit.getRelations(LexRel.antonymy);
```

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 = ”antonymy”;
List<LexUnit> relList;
if (LexRel.isRel(aRel)) {// make sure aRel is a valid lexical relation
    relList = aLexUnit.getRelations(LexRel.valueOf(aRel));
}
```

The following are all valid calls to `getRelations`:

```
aLexUnit.getRelations(LexRel.synonymy);
aLexUnit.getRelations(LexRel.antonymy);
aLexUnit.getRelations(LexRel.pertonymy);
aLexUnit.getRelations(LexRel.participleOf);
aLexUnit.getRelations(LexRel.arg1);
aLexUnit.getRelations(LexRel.arg1_pred);
```

```
aLexUnit.getRelations(LexRel.arg2);
aLexUnit.getRelations(LexRel.arg2_pred);

aLexUnit.getRelations(LexRel.valueOf(“synonymy”));
aLexUnit.getRelations(LexRel.valueOf(“antonymy”)); // 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.getRelations();
```

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 the 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());
         }
      }
   }
}
```