# The graph2tab Library, User Guide

*Last Updated: 2 Feb 2012*

graph2tab is a generic implementation of a method for producing spreadsheets out of experimental workflow graph. It can be used to export data from object models like BII, ArrayExpress, FUGE or MAGE-OM to spreadsheet-like formats, such as MAGE-TAB or ISA-Tab. One of the most important procedures in the library computes a minimum set of paths that go from source nodes (e.g., biological sources) to end nodes (e.g., data) and that covers all the nodes and edges in the input graph. This is based on the computation of a particular minimum flow on the graph, by means of the Ford-Fulkerson algorithm (which has good performance when dealing with the typical experimental graph).

The package is designed in a way that allows to adapt a particular object model about experimental work flows and to reuse classes like `TableBuilder` to produce spreadsheets describing instances of such model, according to a given target format.

This document assumes an audience of Java developers, dealing with bioinformatics data management applications and data reporting standards.

## Summary

## Adapting grap2tab to your use case: overview

In short, what you have to do to use graph2tab, in your particular project about a data converter or exporter, typically is:

- define your node wrappers and the attributes to the describe the experimental entities the nodes represent;
- define your node factory;
- define your table builder;
- if needed, define some criteria to sort nodes, e.g., tell the library that ‘Source Name’ comes before ‘Sample Name’;
- invoke your table builder with the right set of nodes coming from your to-be-converted model (such as the sources or the end nodes). Get the results as a matrix, which represent a piece of output in your target format, such as the SDRF-like table.

Details follow below.

## Set up

The graph2tab library is available as a Maven Java project. The quickest way to use it is to include it as a dependency in your Maven project (see Maven documentation for further information):

```
<dependency>
  <groupId>org.isatools</groupId>
  <artifactId>graph2tab</artifactId>
  <version>4.2</version>
</dependency>
```

```
<repository>
  <id>oerc</id>
  <name>Oxford e-Research Center</name>
  <url>http://frog.oerc.ox.ac.uk:8080/artifactory/repo/</url>
</repository>
```

You may want to download the sources too, especially to look at the examples we describe in this guide. The sources are available via GIT at the address:

```
http://github.com/ISA-tools/graph2tab
```

We have never tried it without using Maven. If that is your case, we suggest that you use the Assembly plug-in from the sources to build a Jar, which can then be included in your project:

> TO-DO: this is not ready yet, we need to add the Assembly descriptor for this.

The library contains some logging inside it, which is based on the SLF4J framework, your particular logger should be selected automatically (see the SLF4J documentation for details).

## The basics

graph2tab can be used to convert a graph about an experimental pipeline to a tabular view, similar to the SDRF structure in the MAGE-TAB format, or the Sample/Assay files in the ISA-Tab format.

In order to do that, you must start from a Java object model that is used to represent the experimental pipeline (e.g., the BII model, or the model defined in the Limpopo MAGE-TAB parser). For example, you will have instances of classes like `BioSource` or `Hybridization` and these will have properties like factorValues, characteristics, inputs, outputs.

graph2tab is designed to be generic: apart from few basic elements, the graph/table conversion does not depend neither on the particular graph that is being converted, nor on the particular set of headers that the output consists of. The library ensures this generality by working with a few basic interfaces.

*`Node`* is one of the most important of them, it defines inputs, outputs and attributes for a node. The former point at other instances of the *`Node`* interface itself. By means of the *`getTabValues`* method, a node exposes attributes, which essentially are header/value pairs. We explains the details about node's attributes (the *`TabValueGroup`* data structure) below.

**uk/ac/ebi/tablib/export/graph_algorithm/Node.java**

```
package uk.ac.ebi.tablib.export.graph_algorithm;

public interface Node extends Comparable<Node>
{
   public SortedSet<Node> getInputs ();
   public SortedSet<Node> getOutputs ();

   public List<TabValueGroup> getTabValues ();

   public String getType ();
   public int getOrder ();
}
```

**uk/ac/ebi/tablib/export/graph_algorithm/TabValueGroup.java**

```
public interface TabValueGroup
{
   public String getHeader();
   public String getValue();
   public List<TabValueGroup> getTail ();
}
```

## An example of graph model

An example of how these interface can be used is provided in the package *uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests* (test/ folder). Here, one finds two examples of how a to-be-converted model can be adapted to graph2tab. One is very simple and it is in the package *dummy_graphs_tests*. Here nodes from the input model are directly implementing out *Node* interface. While this is so simple, it is not very realistic, because you will typically have your model and you will not want to derive it from classes in the *grap2tab* package. So, in such cases, what you need is something similar to the example in the package *simple_biomodel_tests*. Here a simple example object model to describe experiments is defined. This is like what your input will be in use case where you want to use the graph2tab library. One class of such model is *BioMaterial*:

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/BioMaterial.java**

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class BioMaterial extends ExperimentNode
{
   public BioMaterial ( String name )
   {
     super ( name );
   }

   /**
    * Biomaterial characteristics are a set of {@link Annotation}s.
    */
   public void addCharacteristic ( String type, String value, String termAcc, String termSrc )
   {
     super.addAnnotation ( type, value, termAcc, termSrc );
   }

   /**
    * Biomaterial characteristics are a set of {@link Annotation}s.
    */
   protected List<Annotation> getCharacteristics () {
     return super.getAnnotations ();
   }

   /*
```

```
    * getInputs() and getOutputs() are defined in the parent class (ExperimentNode)
    */
}
```

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/model/Annotation.java**

```
package uk.ac.ebi.tablib.export.graph_algorithm.simple_biomodel_tests.model;

public class Annotation
{
    ...

    public Annotation ( String type, String value, String termAcc, String termSrc ) {
      ...
    }

    public Annotation ( String type, String value, OntoTerm ontoTerm ) {
      ...
    }

    public Annotation ( String type, String value ) {
      ...
    }

    public String getType () {
      ...
    }

    public String getValue () {
      ...
    }

    public OntoTerm getOntoTerm () {
      ...
    }

}
```

Here, *`Annotation`* is an example of how a node annotation might look like. Similarly to annotations in standards like MAGE-TAB, or ISA-Tab, MAGE-OM, an annotation has a type, a value and possibly an ontology term (identified by a source symbol and an accession). In order to be adaptable to concrete specific situations, the graph2tab library sees the model nodes in an abstract generic form, where the node is able to tell what are its inputs and outputs, plus the list of header/value pairs that needs to be used to represent the node in the final result table, that is, a list like `Source Name = ‘source 1’, Characteristics[Organism] = ‘Mus-musculus’,` … Most of the work that you have to do to use the library for your particular case consists of writing code to convert the attributes from your model, whatever they are, to such string pairs, i.e.: you have to convert them to instances of *`TabValueGroup`*. This is a recursive data structure, defined as in the listing reported above. The reason for that is that certain groups of attribute pairs must be kept always together, in order to produce correct results. For example, `Characteristics[Organism] = ‘Mus-musculus’, Term Source REF = ‘NCBI-Tax’, Term Accession = ‘123’` must be the a successive groups of columns in the resulting table. Hence, a good encoding of this example would be[1]:

```
TabValueGroup (
     header = "Characteristics [ Organism ]"
     value = "Mus-musculus"
     tail = (
       TabValueGroup (
```

1 Note this is pseudo-code, not something defined in a particular real language.

```
        header = "Term Source REF"
        value = "NCBI-Tax"
        tail = (
          TabValueGroup (
            header = "Term Accession"
            value = "123"
          )
        )
      )
    )
  )
```

With this kind of nesting, you are telling grap2tab that the 'NCBI-Tax' term source goes always next to the organism annotation and that, in turn, the 123 accession goes always next to the NCBI-Tax label (i.e., the three of them are kept together in the final result).

You may have noted the existence of the methods `getType()` and `getOrder()`. This is explained in the section 'Uneven graphs and their layering'.

## Model wrapping

The typical approach to implement such an adaptation from your model to the graph2tab interfaces is defining wrappers. The following is an example about the aforementioned *`Biomaterial`*. A Biomaterial wrapper is build, based on a base wrapping class, *`ExpNodeWrapper`*. The listing below is an adaptation from the test code included with the library and available on line.

```
...
public class BioMaterialWrapper extends ExpNodeWrapper
{
   private ExperimentNode base;

   BioMaterialWrapper ( BioMaterial base )
   {
     this.base = base;
   }

   protected List<TabValueGroup> getTabValues ()
   {
     List<TabValueGroup> result = new ArrayList<TabValueGroup> ();
     result.add ( new DefaultTabValueGroup ( “Biomaterial Name”, base.getName () ) );

     for ( Annotation annotation: base.getAnnotations () )
     {
        DefaultTabValueGroup tbg = new DefaultTabValueGroup (
          " Characteristic [ " + annotation.getType () + " ]",
          annotation.getValue () );
        OntoTerm ot = annotation.getOntoTerm ();
        if ( ot != null )
        {
          tbg.append (
             new DefaultTabValueGroup ( "Term Accession Number", ot.getAcc (),
               new DefaultTabValueGroup (
               "Term Source REF", ot.getSource () )
          ));
        }
        result.add ( tbg );
     }
     return result;
   }

   /** asks base's inputs and use the node factory to wrap them. Caches the result in an internal class
       field, defined by the DefaultAbstractNode ancestor. */
   @Override
   public SortedSet<Node> getInputs ()
   {
     if ( inputs != null ) return super.getInputs ();
     inputs = new TreeSet<Node> ();
     NodeFactory nodeFact = NodeFactory.getInstance ();
     for ( ExperimentNode in: base.getInputs () ) inputs.add ( nodeFact.getNode ( in ) );
     return super.getInputs ();
   }

   ...
   @Override
   public SortedSet<Node> getOutputs ()  {
     ...
   }
   ...
}
```

## Notes

You're free to directly implement the *`Node`* interface for defining your wrappers. However, for common cases, extending *`DefaultAbstractNode`* is probably quite easier. The latter defines common functionality, such as the collection class members for maintaining input/output nodes.

Every node type in your original model has a corresponding wrapper and the two are linked by the base object received by the constructor.

*`getInputs()`*, *`getOutputs()`*. Your custom implementation will typically behave like (for `getInput()`): if the inputs collection is not null returns it, else initialise it, by getting wrappers for each node that is input of the base node for the current wrapper. Wrappers are usually created via a custom wrapper factory (more later on that). In short, what we are doing is creating a graph of wrappers that is isomorphic to the graph of  nodes from your to-be converted model.

`getTabValues()`. In this particular model (yours may be more complicated), there is only one type of attribute that can be attached to nodes: the `Annotation` class. So the `getTabValues()` method converts instances of this class into the required list of headers/values. In this simple example, each header is typed, i.e.., it is like '`Characteristics[ Organism ]`', every attribute has a string value and can have an optional ontology term, which, in turn is made of an accession and a source. Note that the latter two are nested, as explained in the previous section.

To summarise, the wrapper defines a node identifier as is first *`TableValueGroup`* (i.e., '`BioMaterial Name`'/'`source1`'), followed by one *`TableGroup`* per `Annotation`, which contains the annotation type, its value and, optionally, additional pairs like '`Term Accession Number`'/'`123`', '`Term Source REF`'/'`MO`'. The ontology attributes are nested.

## More notes

- The order the attributes are defined with in the wrappers is reflected in the final output table (this also depends on how nodes of the same types are merged, see below).

- As explained above, those header/value pairs that go together have to be nested, starting from one *`TableGroup`* object and adding other *`TableGroup`* to its tail (and tail of tail, in a tree-like recursive fashion). This way, the headers (and hence columns) that must be reported together in the final table are actually never split apart during the graph-to-table conversion.

- By nesting table value groups this way, two nodes of the same type (e.g., two sample nodes) may have different lists of attributes, e.g., one has a *'`Characteristics[cell line]`'* attribute and the other does not. In such a case, you just omit to define any cell line or value for the second one. The converter takes care automatically of 'merging' the two (or more) samples, creating the right set of columns and populating the lines with proper values or empty cells.

- *`getInputs()`* and *`getOutputs()`* return *`SortedSet`*(s). This has a few implications. First, it is a bonus that allows you to establish some order on the output lines. Typically, nodes are compared on the basis of attributes like sample name or hybridization name. That is up to you to decide, by implementing proper versions of *`compareTo()`*, *`equals()`* and *`hashCode()`* for your implementation of the *`Node`* interface.

*DefaultAbstractNode* has sensible default implementations for these methods that take the value of the value returned by the first `TableGroup` found in *getTabValues()*, i.e., things like the ‘Source Name’ or the ‘Sample Name’. This is usually what you are fine with. The implementation in *DefaultAbstractNode* also bases *equals()* and *hashCode()* on object physical identity. This is usually convenient if you choose to extend graph2tab with the wrappers approach and making use of a node factory too (see next section).

## Node factories

If you adapt graph2tab to your model with the wrappers approach, you typically will use an extension of *AbstractNodeFactory* either. This just keeps a map of base nodes (nodes from your model), mapped to your wrappers. It uses it in the method *NodeWrapper getNode( Base node )*, which creates a new wrapper and links it to the the given base, but only if this does not exist yet and already inside the node/wrapper map. Obviously, it stores the new wrapper in the map, to return it in subsequent calls that have the same base as parameter (instead of creating a new one). This ensures that you create a 1-1 graph of wrappers from the original graph that the to-be-converted model represents. Indeed, you do not need to care about these details, you only need to implement *createNewNode()*, which will be like this:

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/NodeFactory.java**

```
public class NodeFactory extends AbstractNodeFactory<ExpNodeWrapper, ExperimentNode>
{
  private NodeFactory() {}

  private static final NodeFactory instance = new NodeFactory ();

  public static NodeFactory getInstance () {
    return instance;
  }

  @Override
  protected ExpNodeWrapper createNewNode ( ExperimentNode base )
  {
    if ( base instanceof BioMaterial) return new BioMaterialWrapper ( (BioMaterial) base );
    if ( base instanceof Data ) return new DataWrapper ( (Data) base );
    if ( base instanceof ProtocolRef) return new ProtocolRefWrapper ( (ProtocolRef) base );
    throw new IllegalArgumentException (
      "Node of type " + base.getClass ().getSimpleName () + " not supported"
    );
  }
}
```

this method is what the mentioned *getNode()* will call, when it decides it needs a new wrapper.

## Table builder

The `TableBuilder` class coordinates the conversion job. We suggest you extend it and define a method that receives nodes from your input graph, as it is done in the constructor *SimpleModelTableBuilder( Set<ExperimentNode> nodes )*:

**uk/ac/ebi/tablib/export/graph_algorithm/simple_biomodel_tests/node_wrappers/SimpleModelTableBuilder.java**

```
public class SimpleModelTableBuilder extends TableBuilder
{
  public SimpleModelTableBuilder ( Set<ExperimentNode> nodes )
  {
    this.nodes = new HashSet<Node> ();
    NodeFactory nodeFact = NodeFactory.getInstance ();
    for ( ExperimentNode node: nodes ) this.nodes.add ( nodeFact.getNode ( node ) );
  }

}
```

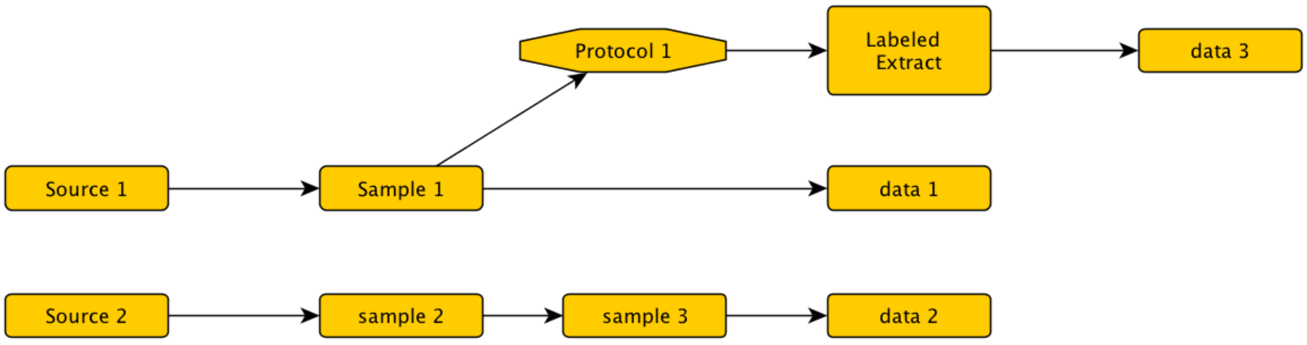
As one can see, this method just populates the *nodes* class field, containing node wrappers, and does it by using your node factory. As for the nodes that should be passed to the table builder, any set of nodes that allow one to reach all the graph by walking from such nodes upstream (following inputs up to the sources) and downstream (following outputs up to the sinks/end nodes) is fine. Actually, passing source nodes, i.e., those without any input, will make the conversion slightly faster (see below in this document).

## Invoking the conversion

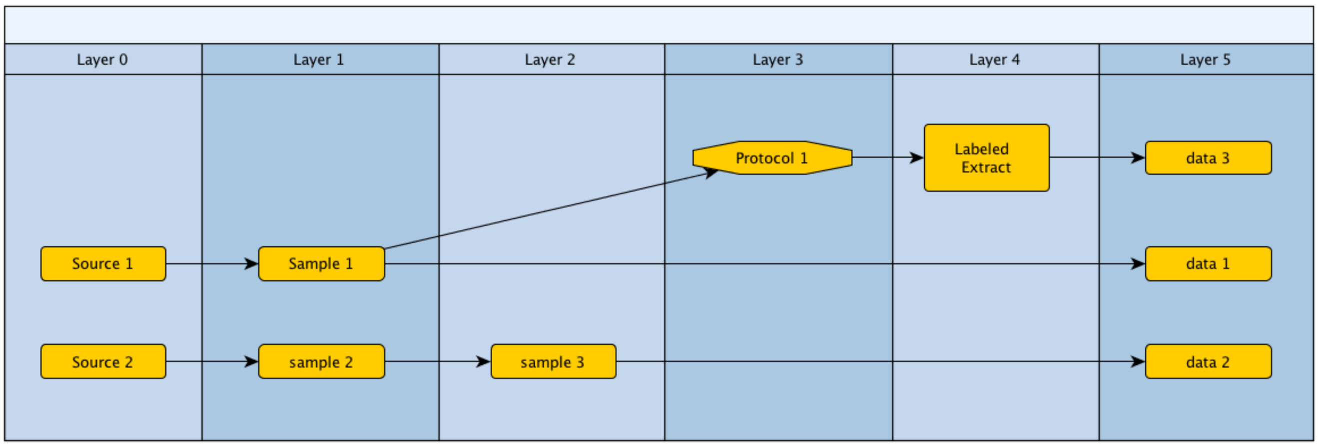
Once all is in place, you can go with the conversion, basically invoking *getTable()* from your table builder. An invocation example can be found in *SimpleBioModelTest* in the source code.

## Uneven graphs and their layering

What we have explained so far works correctly only upon the condition that the graph to be converted is 'even', which means that all the paths from the sources (e.g., bio-sources) to the end nodes (e.g., data) have the same length. graph2tab supports 'uneven' graphs as well, e.g., a graph where you may have paths of different lengths, with some experimental workflow stages omitted. An example is this:



A graph of this type is converted into tables by means of the same approach described above, which transforms the initial graph into a set of chains. However, in order to cope with such uneven paths, the input graph is pre-processed by a 'layering' component (if you are curious, this is implemented by the `LayersBuilder` class, but you do not need to know anything about it in order to use the library to convert your experimental workflows). This component distributes the graph nodes into a set of layers, where every layer reflects the walk distances of the nodes it contains from left-most side of the graph and also contains nodes all of the same type. In the example above, this would produce:



Once the graph is re-arranged this way (i.e., every node gets a layer index), it is easy to scan the graph and to generate a table that has gaps:

| **Source Name** | **Sample Name** | **Sample Name** | **Protocol REF** | **Labeled Extract Name** | **Data File** |
|---|---|---|---|---|---|
| Source 1 | Sample 1 | | Protocol 1 | Labeled Extract | Data 3 |
| Source 1 | Sample 1 | | | | Data 1 |
| Source 2 | Sample 2 | Sample 3 | | | Data 2 |

The layering algorithm implemented in the library is essentially based on the method briefly described in the MAGE-TAB specification[2], with several necessary details (not explicitly considered in that description) added by us. Once a graph is adapted to the *Node* interface, there isn't much else that you have to care about in order to make graph2tab to compute the node layers. However, because of the generic nature of the library, there is some information that you are required to provide with. This is done via the *Node.getOrder()* method. It works like this: layer indexes are initially assigned to the graph nodes based on the highest topological distance from their sources (e.g.: 'Protocol 1' and 'Sample 3' in the example above would get 2). After that, for each layer, all the nodes in that layer are compared each other. If two nodes in the same layer are of a different type, getOrder() is invoked to decide which one is supposed to shift on the right (the one having a bigger or equal order). If you expect to work with uneven graphs, you do have to provide an implementation of getOrder() in your custom implementation. Two typical ways to do this are:

- An hash table (i.e., Java Map) of header types is defined (ie, an array of of strings), such as 'Source Name', 'Sample Name', 'Extract Name' … and the order of a node is computed by considering its first header (actually *Node.getType()*, see below) and taking the index of this header in such predefined array. For example, if the first header returned by a node is 'Sample', its order is 2, because in the predefined array headers it has that index. This approach is equivalent to providing a list of types which of order is known in advance because of the experimental stage they represent.
- An alternative is possible when the graph to be exported was initially generated from an existing tabular format (e.g.: loaded from a MAGE-TAB submission) and stored to a database. In such a case it is possible to track and store the column index the initial header of a node comes from and use such index as return value for *getOrder()*. This kind of solution is interesting for those

2 Section 3.3.2 in http://annotare.googlecode.com/files/MAGE-TABv1.1.pdf

who want to be able to re-export originally submitted data and reflect the initial submission as much as possible. For instance, in the example in the figure above, it may be that `'Protocol REF'` nodes are not associated to a particular order (i.e., *`getOrder()`* returns -1) and the layering component places such nodes close to their output (as it is the case more often than not). In a different scenario, `'Protocol 1'` could be between the two columns about samples (i.e., its output is an omitted sample) and this could be defined by the column occupied in the submission that generated this graph.

The use of the order approach is very flexible and can be used for many different custom cases. For example, you may decide to combine the two criteria above, using a predefined array for all the fields except the protocol references, for which you may want to apply the column position. Another more advanced example consists of considering the type of protocol a 'Protocol REF' node refers to and assigning the order on the basis of this. Then additional fields would be configured in the list of possible fields, e.g.: 'Sampling Protocol', 'Extraction Protocol' etc. This is the reason why the *`Node`* interface has the method `getType()`. Normally the type can be associated to the first header (and this is the default implementation in `DefaultAbstractNode`). However, in a case like the one at issue, it may be that the first header is `'Protocol REF'`, while the node type is something like `'Extraction Protocol'`, achieved from the referred protocol. `getType()` could be useful when you have entities reported with the same header, but of slightly different subtype, e.g., you want to distinguish 'treatment protocol' and 'extraction protocol'.

### Headers with Undefined order

As mentioned above, there will be certain headers that will have an undefined order (which corresponds to -1 in the graph2tab library), for instance `'Protocol REF'`. Where one of such fields have to be shifted, the layering procedure considers the order of the nodes on its left and on its right and the other node being compared, if this has an order different than -1. For instance, in the example above, the `Protocol 1` is shifted on the right, since `Sample 3`, the node that initially shares the layer with such protocol reference, has an order that is closer to the one of `Sample 1` and `Sample 2` than the lowest order found on its right, which is the one of `Labelled Extract` (order(Sample 3) – order(Sample 1/2) is 0, being the nodes of the same type). The rationale for this is to prefer a right shift of nodes like protocol references in such situations, because in real biological cases it is more frequent to report a protocol direct output together with a named protocol and omit its direct input, rather than the opposite. See the source of `LayersBuilder` for details about the implemented shifting criteria.

### Layering example in the source code

The graph2tab package contains an example of 'layered implementation' in the package `layering_tests`. Please see the Junit tests in the class *`LayeringTests`*. There, the class *`ExpNodeWrapper`* can be found too, which was reported above. In the real implementation, this contains an implementation of the *`getOrder()`* method that uses the first approach described before (the array of possible headers). The package *`layering_tests.model`* extends *`simple_biomodel_tests.model`* with a few nodes that makes it possible to build examples of uneven graphs.

This is a fragment from *`ExpNodeWrapper`*, showing how *`getOrder()`* is implemented there:

```
public abstract class ExpNodeWrapper extends DefaultAbstractNode
{
   ...
   public static final Map<String, Integer> TYPE_ORDER = new HashMap<String, Integer> ()
   {{
      put ( "Source Name", 0 );
      put ( "Sample Name", 1 );
      put ( "Extract Name", 2 );
      put ( "Labeled Extract Name", 3 );
      put ( "Assay Name", 4 );
      put ( "Data File Name", 5 );
   }};
   ...
   public int getOrder ()
   {
      String header = getType();
      Integer order = TYPE_ORDER.get ( header );
      return order == null ? -1 : order;
   }
   ...
}
```

### *Disabling the layering stage for even graphs*

By default the *TableBuilder* class creates and instance of the *LayersBuider* class and invokes the layering procedure, before doing the the rest of the conversion process. If you are absolutely sure that your custom implementation will deal with even graphs only, this layering step, while not affecting the results in such a case, is not necessary. Therefore we have provided an option to disable it, so that you can save some processing time (and a little of RAM too) if you do not need this processing. This is available via the version of *TableBuilder* constructor that accepts the parameter *isLayeringRequired*. Such parameter is true by default.

### *Choosing the initial node set to be passed to the table builder*

We have already shown that *TableBuilder* needs a certain subset of nodes from your input graph, in order to reach all the graph nodes and work out the conversion to a table. A simple working choice for this consists of all the nodes the graph is composed of. However, the conversion can be speed up a little by passing the set of source nodes, i.e., nodes that have no input. These must still have the general property of being connected with the rest of the graph that is to be converted.