KGraph Text (KGT)

The KGraph is the basic data structure used by the Eclipse Layout Kernel (ELK) to describe and work with graphs. While developing layout algorithms, it is often necessary to assemble very specific graphs to see what the algorithm does with them. This is what the KGraph Text language was designed for: to be a simple language to assemble KGraphs for testing purposes.

This short tutorial will first introduce you to the KGraph and then walk you through writing your first KGT file. Grab a cup of tea and a few biscuits, slip into something more comfortable and get ready!

Preliminaries

There's a few things to do before we dive into the tutorial itself. For example, to do Eclipse programming, you will have to get your hands on an Eclipse installation first. Read through the following sections to get ready for the tutorial tasks.

Required Software

You will first need an Eclipse installation to hack away on layout algorithms with. Since we have a shiny Oomph setup available, this turns out to be comparatively painless:

1. Go to this site and download the Eclipse Installer for your platform. You will find the links at the bottom of the *Try the Eclipse Installer* box.
2. Start the installer. Click the Hamburger button at the top right corner and select Advanced Mode. Why? Because we're computer scientists, that's why!
3. Select *Eclipse Modelling Tools* from the *Eclipse.org* section and click Next.
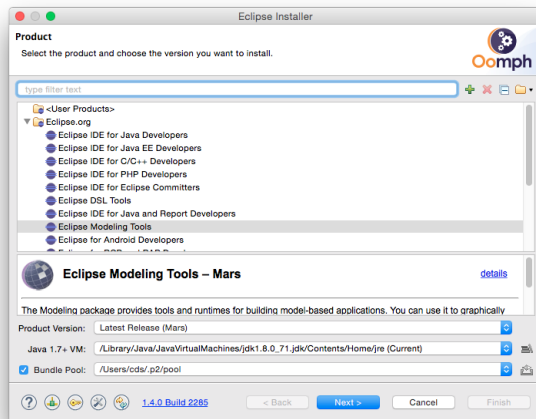
4. Next, we need to tell Oomph to get everything ready for layout algorithm development. Download our Oomph setup file, click the Plus button at the top right corner and add the setup file to the Github Projects catalog. Select the new Graph Drawing entry by clicking the check box to its left. This will cause an item to appear in the table at the bottom of the window. Once you're done, click Next.
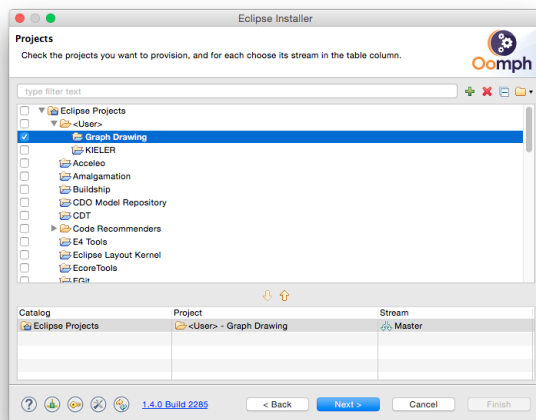
5. Oomph now asks you to enter some more information. You can usually leave the settings as is, except for the Installation folder name. This will be the directory under which all your Eclipse installations installed with Oomph will appear, each in a separate sub-directory. Select a proper directory of your choice and click Next.

Finding Documentation

During the tutorial, we will cover each topic only briefly, so it is always a good idea to find more information online. Here's some more resources that may prove helpful:

- Java™ Platform, Standard Edition 8 API Specification
  As Java programmers, you will already know this one, but it's so important and helpful that it's worth repeating. The API documentation contains just about everything you need to know about the API provided by Java.
- Eclipse Help System
  Eclipse comes with its own help system that contains a wealth of information. You will be spending most of your time in the *Platform Plug-in Developer Guide*, which contains the following three important sections:
  - Programmer's Guide
    When you encounter a new topic, such as SWT or JFace, the Programmer's Guide often contains helpful articles to give you a first overview. Recommended reading.
  - References -> API Reference
    One of the two most important parts of the Eclipse Help System, the API Reference contains the Javadoc documentation of all Eclipse framework classes. Extremely helpful.
  - References -> Extension Points Reference
    The other of the two most important parts of the Eclipse Help System, the Extension Point Reference lists all extension points of the Eclipse framework along with information about what they are and how to use them. Also extremely helpful.
- Eclipsepedia
  The official Eclipse Wiki. Contains a wealth of information on Eclipse programming.
- Eclipse Resources
  Provides forums, tutorials, articles, presentations, etc. on Eclipse and Eclipse-related topics.
- Eclipse Layout Kernel 👍 ⚠️
  Documentation on how the layout infrastructure works and on how to write your own layout algorithms. This is our project, so if you find that something is unclear or missing, tell us about it!

You will find that despite of all of these resources Eclipse is still not as well commented and documented as we'd like it to be. Finding out how stuff works in the world of Eclipse can thus sometimes be a challenge. However, you are not alone: this also applies to many people who are conveniently connected by something called *The Internet*. It should go without saying that if all else fails, Google often turns up great tutorials or solutions to problems you may run into. And if it doesn't, your advisers will be happy to help.

As far as KIELER documentation is concerned, you will find documentation at the KIELER Confluence. The documentation is not as complete as we (and especially everyone else) would like it to be, however, so feel free to ask those responsible for help if you have questions that the documentation fails to answer.

The KGraph

Take a look at the meta model on the KGraph Meta Model page. As you can see there, graphs consist of *nodes* and directed *edges* that connect the nodes. Nodes can declare special connection points called *ports* that edges can, but don't have to connect to. Nodes, ports, and edges can have *labels* that describe them. Since we don't want to pass lists of nodes around, each graph has a top-level node that contains the other nodes. This kind of parent-child relationship also enables us to describe *nested graphs*: graphs where nodes can contain further nodes themselves. So far, this is not all that surprising.

Our main goal with the KGraph is to have a data structure for layout algorithms. To compute a layout, we need to be able to specify the size and coordinates of each graph element. This is where the KLayoutData Meta Model comes into play. Each graph element can have arbitrarily many `KGraphData` elements attached to it. The KLayoutData meta model describes a number of such types that go with different graph elements:

Graph Element	KGraphData Type
KNode	KShapeLayout
KPort	KShapeLayout
KLabel	KShapeLayout
KEdge	KEdgeLayout

A shape has a size and width as well as coordinates and, possibly, insets. An edge has a starting point, an end point, and a list of bend points. Note that different coordinates have different frames of reference, as shown in the diagram below the KLayoutData meta model.

Layout algorithms will usually provide different options to configure them to your needs. To this end, each KGraphData class has a list of properties associated with it. Layout algorithms will go and evaluate properties as they compute layouts.

KGraph Text

1. When we cobble together our KGraph, it would be nice to see if what we type in makes any sense. As it turns out, we have actually built an Eclipse view that shows the graph as you type it in, but you will have to open that view first. From the *Window* menu, select *Show View - Other....* In the dialog, select *Diagram* from the *KIELER Lightweight Diagrams* category. Your Eclipse window should look something like this:
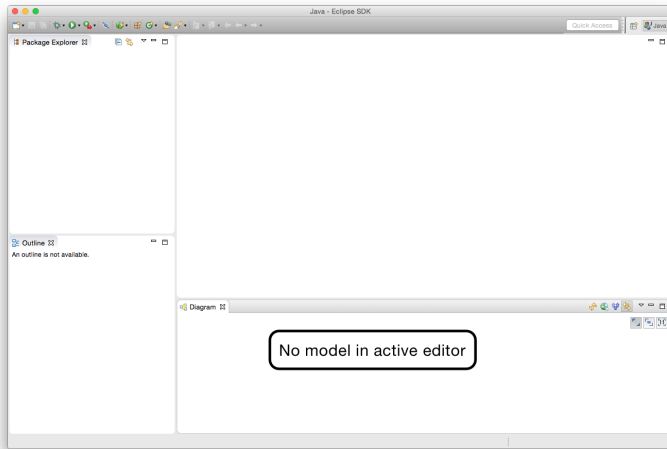
2. Create an empty project in your Eclipse workspace. Right-click the project and select *New -> Other...* In the dialog that pops up, select *Empty KGraph* from the *KGraph* category. Give the new file a proper name and click *Finish* to create it. Eclipse will ask you if you want to add the Xtext nature to your project. Click *Yes.* Don't worry about it.
3. The KGraph editor should open and the Diagram view should update itself. Both should be empty, but the Diagram view should now show a handy sidebar that you can use to influence how your graph will be displayed. If the KGraph editor does not open, open the file yourself by double-clicking it in the *Project Explorer.*
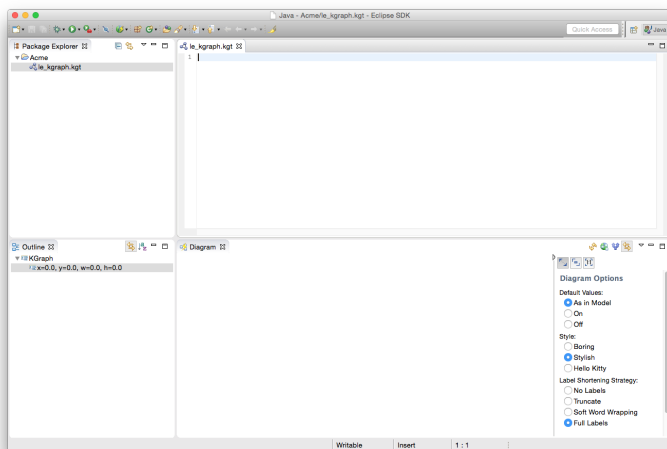

4. Start by adding a first node. Enter the following text into the editor:

```
knode node1
```

The KLighD view should update itself, but you won't know since your node has no size yet and KLighD doesn't know how to draw it. Yep, that's not quite as helpful as you would have hoped. But alas, help is on its way!
5. In the sidebar of the KLighD view, you can enable default values. Do so. (You can also enable stylish styling, which will improve the styling of the drawing style used to style nodes.)
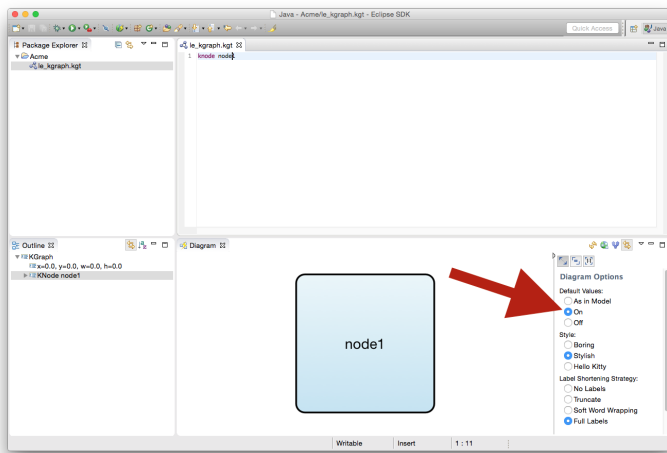
Your node now has a default label, a default size, and a default way to render it (a simple or a stylish rectangle).

6. Add two further nodes, `node2` and `node3`:

```
knode node2
knode node3
```

7. Uniform sizes are boring, so let's give more individuality to `node3`:

```
knode node3 {
    size: width=80 height=40
}
```

8. Let's add connection points to the nodes. Add two ports to `node1`:

```
knode node1 {
    kport port1_1
    kport port1_2
}
```

Two black rectangles should now appear at `node1`. If we had turned off default values, the ports would not have a proper size and would miss their labels.

9. Add a port `port2_1` and `port2_2` to `node2` and a port `port3_1` to `node3`.

10. It's now time to connect the nodes. Add two edges to the graph that originate at `node1` by adding the following lines under the port definitions of `node1`:

```
kedge (:port1_2 -> node2:port2_1)
kedge (:port1_1 -> node3:port3_1)
```

Edges can start and end at a node or at a port. The source node does not need to be explicitly specified since it is clear from the context (the edges are defined in the body of the source node, after all). Thus, if an edge connects directly to the source node (that is, not through a port), the part before the arrow (->) will be empty. The target needs the node to be specified, with an optional target port. Add another edge that starts at `port2_2` and ends at `port3_1`. By now, the KLighD view should show something like this (with stylish styling enabled, of course):
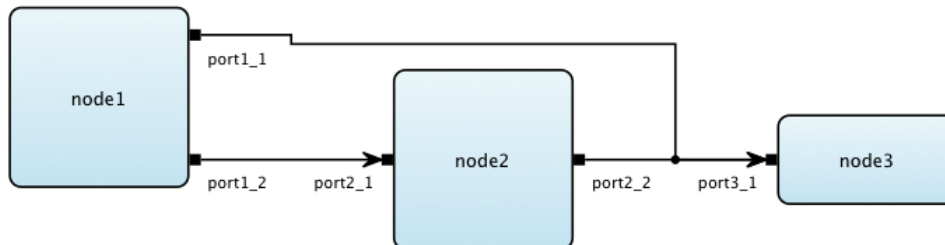
11. Let's add a final touch to the graph. Currently, the edges are routed orthogonally. If we want to change that, we need to tell the layout algorithm to use another edge routing algorithm. This can be done by attaching a layout option to our graph. Add a new `properties` section to the beginning of the file:

```
properties:
    org.eclipse.elk.edgeRouting=POLYLINE
```

Your result could look something like this:
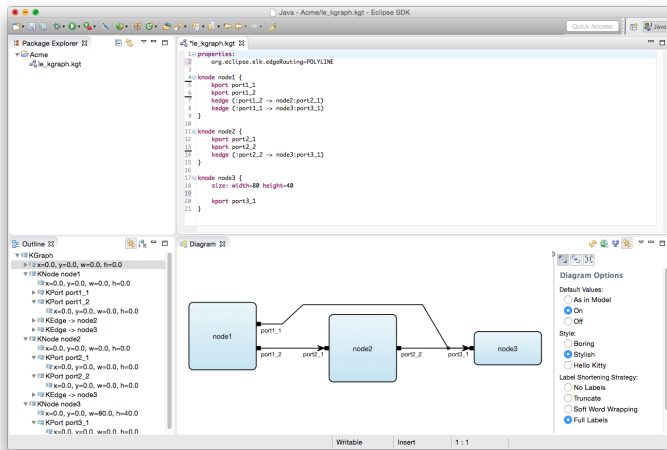
> ✅ **Properties**
>
> Properties can be attached to just about anything in a KGraph: the graph itself, nodes, ports, labels, ...

12. Okay, how about a final final touch. The node labels are currently centered inside their nodes. You can change that by adding the following property to, say, `node1`:

```
org.eclipse.elk.nodeLabels.placement="OUTSIDE V_BOTTOM H_CENTER"
```

This will center its label below the node.

Right, that concludes our little tutorial. If you want to go further, read more on the KGT syntax and look at a bigger example. Also, solve the following assignment.

Assignment

Graphs can also be nested. Child nodes are added to a parent node just like you added ports to parent nodes. Try to extend the graph above such that it looks like this:
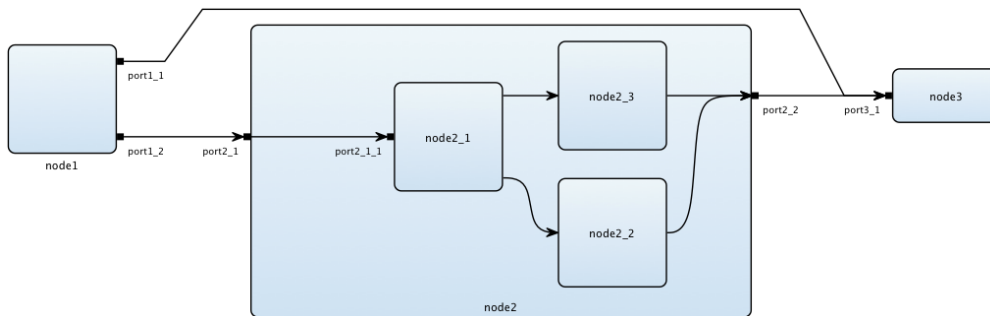
Note that the edges inside `node2` are routed with smooth, stylish splines. To achieve that, set edge routing to `SPLINES`.