Documentation Overview for jmodev

Jmodev is a complicated application, so to ensure your complete understanding of it, I recommend that you proceed down the following course of action IN ORDER:

1. Read the README.md file on the GitHub repository fully and completely FIRST.
2. Read through and understand the How it Works guide in this document.
3. Read through the example config file provided in the documentation folder as well as the config file guide provided in this document.
4. Read the Intro to JFreeChart section in this document.
5. Read through the package guide in this document.
6. Briefly read through the top comment in every class that describes the purpose of the class and what it does.
7. Take a deeper look into each class, ensuring you understand not only the purpose, but the actual implementation of each class (and why it is the way it is). Read the helpful tips section of this document if you get stuck. I recommend leaving ODEView for last, as it is the most complicated class.
8. Read the What Should I Do Next? Section of this document.

I would anticipate that it would take a dedicated student 4-5 hours of time to fully become comfortable with this application.

How it Works

Jmodev has a complicated structure. It itself takes an input of a config file like the example provided in the documentation folder. This config file itself contains a script that is run every time the ODEView updates. The script is denoted by the lines beginning with “run”, and the files included in the “run script” also need to be in the same directory as the config file and the .jar file itself. This means that makefiles and full on programs that we use to generate our output are necessary to be in the same directory as the jar file.

In other words, in order for our application to run, we need any of the dependencies of the “run script” in the config file to be readily available, and we need the config file itself.

Intro to JFreeChart

JFreeChart is a core dependency to this project, as it is what we use to generate charts.   
  
The general workflow to making a chart in JFreechart is to first create a Dataset. There are multiple classes that extend Dataset, but the one we will focus on is the DefaultXYDataset, as it is the one we use in this project. Once you create your DefaultXYDataset, you will call the addSeries command on it. The addSeries command takes 2 arguments, a key for keeping track of the series, and a 2d array of doubles which represent the actual datapoints themselves.

DefaultXYDataset dataset = **new** DefaultXYDataset();  
**double**[][] data = **new** **double**[2][50];  
loadData(data); // psuedocode  
dataset.addSeries(“Data1”, data);  
data = **new** **double**[2][50];  
loadData(data); // pseudocode  
dataset.addSeries(“Data2”, data);

As an important note, our 2d array of doubles must be length 2 in the first dimension. This creates 2 arrays each of our second dimension in length. The first array (0 index) is the x values, and the second array (1 index) is the y values, which pair off to form (x,y) coordinates.

Once you create your dataset, you can start making your plot. There are many different types of plots in JFreeChart, but here we will use the XYPlot, as it correctly represents the data we are trying to plot. An XYPlot takes 4 arguments, one of them being the dataset we mentioned earlier, and the others being 2 ValueAxis parameters and 1 XYItemRenderer. For our current implementation, a ValueAxis can either be a LogAxis or a NumberAxis. A LogAxis is an axis that follows a logarithmic scale, and a NumberAxis is just a plain old number Axis. Both of these Axises can be named, and their names will appear on the chart. An example construction is below. The XYItemRenderer is an interface in JFreeChart that contains all the methods a Plot needs to tell it how to render its data. The implementing class that we use with our XYPlot regularly in this project is the XYLineAndShapeRenderer.

LogAxis yAxis = **new** LogAxis(“Height”);  
NumberAxis xAxis = **new** NumberAxis(“Time”);  
XYLineAndShapeRenderer renderer = **new** XYLineAndShapeRenderer(**true**, **false**); // lines on shapes off  
XYPlot plot = **new** XYPlot(dataset, xAxis, yAxis, renderer);

Once you have your plot constructed, all that’s left is to place the plot in a JFreeChart and mount it to a ChartPanel, which behaves just like a normal JPanel except it houses a singular object being the JFreeChart object.

JFreeChart jChart = **new** JFreeChart(plot);  
ChartPanel cPanel = **new** ChartPanel(jChart);

There are more methods and other things that are used by our project in JFreeChart, but with this information you should be able to piece together what those do, and be able to expand upon it by reading the official JFreeChart documentation: <https://www.jfree.org/jfreechart/javadoc/index.html>

Package Guide

Jmodev is currently split into 7 packages. Each of these packages is organized according to their structure in the application. Below is a rundown of the packages and what they do:

* actions: Contains classes that define java actions primarily for use in the menuBar present in the application. If you are unfamiliar with how actions work, I recommend doing your own research on it or talking to Professor Bernstein, who is very familiar with them.
* app: Contains the main class.
* data: Contains classes that define collections of data necessary for building our charts with JFreeChart
* error: Contains classes that allow us to better keep track of and manage errors due to user input. Very bare right now, but ideally should be built up so that the user knows what to correct in the event that they make a mistake.
* gui: Contains elements for general use in the GUI, such as our JSnapSlider
* settings: Contains classes pertaining to the feature of user manipulation of chart settings. If you want to see what the GUI looks like, click View > Chart Settings on the menu bar.
* view: Contains the most complicated class right now, ODEView, which does too much on its own to be placed in one of the above packages. I anticipate that if you do refactor this class, like I recommend later, that you would move it into the GUI package and create a new “Parser” package.

Helpful Tips

As a preface, like many undergrad research projects, this is largely unstructured. I did not have a comprehensive idea on how I wanted the project to look at the start of my research, nor did I really know what the people I was working for would want me to for it. This means that the project is way less organized than it could be, and although I have put time and effort into refactoring things and trying to make them more clean and understandable (and better to code with too), schoolwork and life both are real things that took away from my time to do this. There are obvious improvements to make, but they take time. If you see an obvious improvement, please go ahead and make it, as I hope that this will be a research project that interests JMU students to come.

* ODEView has altogether too much functionality, so when it comes to understanding how it works, I would start by reading the constructor and understanding what’s going on there, then reading each method that it calls after to understand that.
* Let it be known that event handling in this project is altogether inconsistent. If you find an easy way to improve this, please do.
* If you’re still confused about ODEView after reading through everything, please talk to Professor Lam or reach out to me to understand what’s going on.
* If for whatever reason your program isn’t running, check your dependencies.
* Like mentioned earlier, if you are unfamiliar with using actions in menuBars talk to professor Bernstein or read up on them.

What should I do Next?

There is a lot to improve with this project. Currently the parsing capabilities are put into ODEView. If you wanted to separate that into a different class hierarchy so students could easily define new parsers for existing ODEs that would be an admirable goal.  
  
One of the most important things with this project is SPEED. It has not been explored as to whether or not parallelism could be used to speed this up.

Another good idea is improving precision. This would probably require extending one of jFreeChart’s classes to implement, but creating high precision charts can be very important, especially for doing precise things like displaying errors.

jFreeChart’s UI is currently quite lackluster. It has some basic features in context menus, but it could really use more to interact with. If you wanted to get more into UI design, you could try to clean up the GUI and make it more accessible. Also, the java swing GUIs are by default, run on a single thread. This means that they can slow down when there are a lot of things interacting with it at the same time. Reaching out to use a different GUI could be worth doing if that is something that you think of.