The Shipment Tracker App

The project for week 12 was to create a shipment tracker from specifications given for the 6 different classes and interfaces and implement a collection (of shipments) that can be sorted or ordered. The exact design specs for Box, Item, Return and Shipment, and interfaces Weightable and Post (with generic interfaces) were provided.

We learned that by getting interfaces “up” (or running) immediately it made programming classes more consistent by reassuring instances of other classes like Boxes, Returns or whatever would include “weightable” Items that helps for later comparisons. The instances of Shipment implement Post. The places in code where generics were included in: method return types, fields types, and with two generic list implementations (Shipment and Box).

Part two of the exercise asked to extend features of the program in Shipments methods for addPackage and sortShipment. The real secret to accomplishing the task was to not jump ahead into attempting to make object comparisons *and also taking some of the difficult thought process out of the programming* by relying on our already implemented ArrayList for the shipments in the class. Therefore these methods just call add and sort methods of ArrayList, simply or Iterable’s iterator method.

The challenging work was in the details of comparator found in the Driver code class, where an anonymous class could be used to set up an Iterator that compares using a Comparator as follows,

Iterator<Post> **it** = shipment.sortShipment(**new** Comparator<Post>() {

@Override

**public** **int** **compare**(Post o1, Post o2) {

**if**(o1.getWeight() < o2.getWeight())

**return** -1;

**if**(o1.getWeight() > o2.getWeight())

**return** 1;

**return** 0;

}

});

Iterator<Post> **it2** = shipment.sortShipment(**new** Comparator<Post>() {

@Override

**public** **int** **compare**(Post o1, Post o2) {

**return** o1.getDay().compareTo(o2.getDay());

}

});

**while**(it.hasNext()) {

System.***out***.println(it.next());

}

NOTE: The iterators seem to create issues when trying to run both of them at the same time

That’s it!

The Graph Structure visualization Test Application

The second project for week 12 was more of an individual focused effort to work with a fully packaged 3rd party application that we downloaded the jar files, for a tool that analyzes and presents graphing structures of the app developer’s implementations of graph structures, SingleGraph and Graph.

We were actually the “user” of the data structures this time, and had to learn to correctly save the packaged jar files into a project folder, create other folders to store data to be accessible from within our IDE, including two data files (csv files) that we downloaded for nodes and edges to read into the program, using the text in the files and following instructions to create new object instances for FileReader in the BufferedReader, and then read one line at a time, with br.readline().

The data from the files were used with some slightly abnormal accessing issue to handle the comma separated values by calling line.split(“,”) that returns an array with the 2 elements that happen to be both the arguments we need for the graph.addEdge method. The values from split alongside the line being read in from file together are the 3 pieces needed to complete the arguments for this, which for some reason works like this,

graph.addEdge(String.***valueOf***(line2), values[0], values[1]);