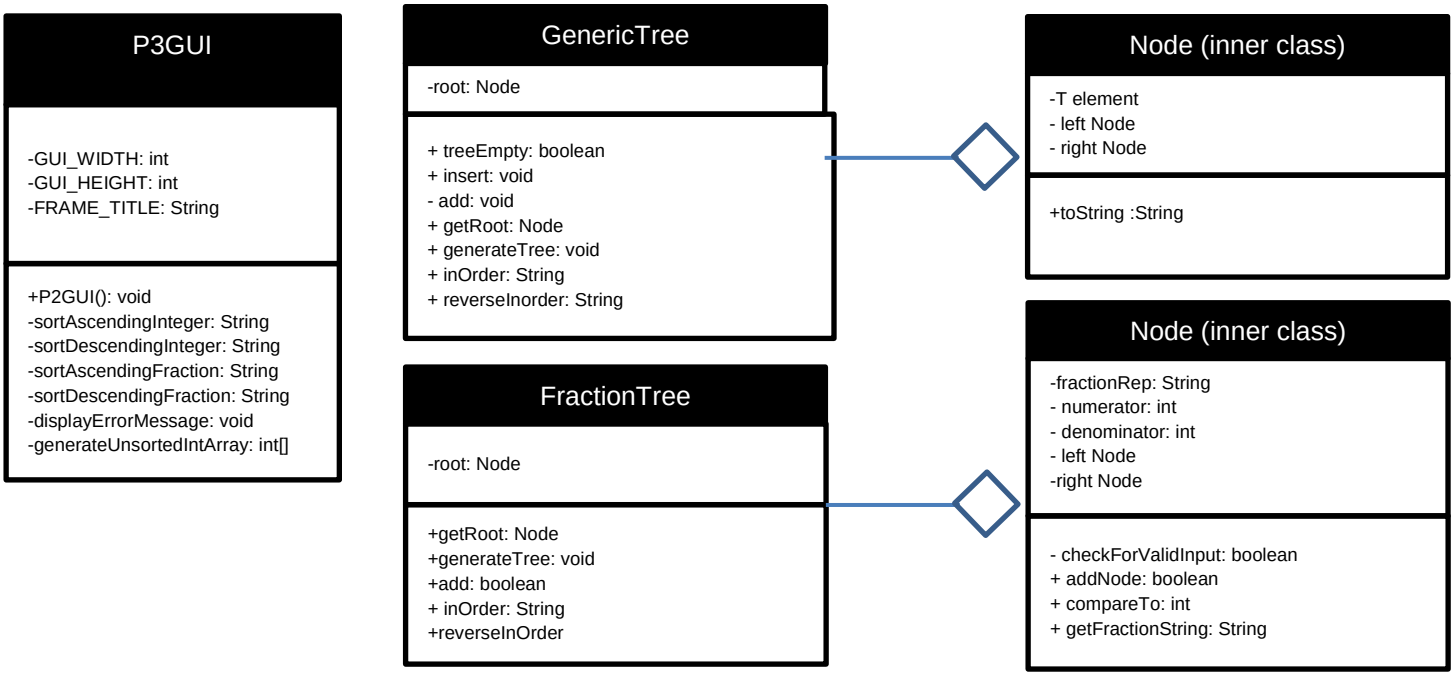
Eric Olsen

Assumptions:

The task for this assignment is to create a Graphical User Interface (GUI) that performs a sorting operation on an unsorted list of either integers or fractions. To narrow the scope of this assignment we must assume a few things. First, our sorting algorithm will utilize an in-order traversal of a tree structure. Second, our GUI will utilize spaces as the delimiters between our unsorted elements. Additionally, we will only utilize integers and fractions in our sorting; we will not utilize decimal numbers. Finally, we must allow duplicate elements to exist in the data tree structure and we do not need to re-balance the tree when inserting its nodes.

Main Design Decisions:

This program consists of three primary classes: P3GUI, GenericTree, and FractionTree. Additionally, both the GenericTree and FractionTree classes have inner classes which define the individual nodes of the data tree structure. To aid in this description I will include the UML diagram below before any further description.



I will first begin by describing P3GUI. Like in the previous assignments, most of the java swing methods are self-explanatory. This particular GUI utilizes radio buttons in order to specify whether our unsorted list is a fraction or integer and whether the sorted list will be sorted in ascending or descending order. The GUI also catches both the NumberFormatException and InputMismatchException from integer parsing operation and FractionTree class. The GUI also removes extra spaces between elements in the unsorted list to alleviate some error control. Finally, it was necessary to make the GUI generate an integer array BEFORE passing it to the Generic tree. Previous versions of the GenericTree class attempted to parse the unsorted String with some very inconsistent results.

The GenericTree class started as an IntegerTree class before I realized that I misread the assignment. The current GenericTree class accepts the type parameter 'T.' GenericTree contains a single field which is the tree's root node. Its inner class, Node, contains three fields: the node's type element, its left pointer, and its right pointer. It also contains a single constructor which assigns only the element value to the node; the left and right pointer assignments occur during the GenericTree's add method.

The add method is worth some description. It is a private method that is called from the public insert method. The insert method exists to first check if there is a root value. If there is no root, it creates one, otherwise it sends all new nodes to the add method. Since this tree accepts duplicate values, some extra consideration is needed. Normally a Binary Search Tree adds values by comparing the inserted value to the root. When the value is greater than the root, we compare the inserted value to the right child. If the inserted value is less than the root, we follow the same process to the left. For duplicates values we need to consider what happens when the compared value equals the compared root or child. For this assignment I chose to send all duplicate values to the left. This method also utilizes recursion and not iteration. More specifically, when the compared node has a null left or right pointer, we instantiate a new Node object at that the appropriate null location. If we need to compare the new value to a child node, we recursively call the add method again until we discover a null node.

The remainder of the GenericTree is not overly complex. The inOrder method returns a String value of the tree structure. Just like Project2, the inOrder method utilizes recursion. It first checks to see if the root value has an element. If it does not, it returns a blank value which is also this method's exit criteria. If the root value has an element, this method recursively calls itself. The inOrder method returns the left, root, and right values to show an ascending order. The reverseInOrder method functions in the same manner, but returns a right, root, and left String value instead. The majority of issues I had with this class was initially in the String parsing operation. This class now accepts an array of type objects and works fine. Consequently, I put the responsibility of String parsing on the GUI rather than this generic class.

Our fraction tree class requires a little extra consideration. This class also contains a single field as the root Node and contains an inner Node class. This inner class contains five fields: a String representation, an integer numerator, an integer denominator, a left Node pointer, and a right Node pointer. This class utilizes handles two data parsing operations. The Fraction class parses the space delimited Strings and the inner class Node constructor parses the numberator and denominator. The addNode method is nearly identical to my approach in the GenericTree class; it utilizes the compareTo method but carries a specific operation. More specifically, the compareTo method in the FractionTree Node class accepts an Object and immediately casts it to a Node object. Using double values within the compareTo method created some very strange results. What eventually worked quite well is multiplying the inserted fractions numerator by the compared fraction's denominator and vice versa. An integer comparison produces the correct results.

Aside from the error control which I will describe momentarily, there are not too many differences between the FractionTree class and GenericTree class outside of their node data structure and data parsing operations. One significant difference, however, is when we instantiate the inner node object. When inserting elements into the GenericTree, we instantiate the new tree nodes after completing our node comparisons in the add method. The fraction nodes, on the other hand, are instantiated before being added because of the inner class's added complexities.

Error Handling:

There are many ways that this program can go wrong. Most of the runtime exceptions that I encountered were either NumberFormatException or InputMismatchException. My early attempts at making my own private methods to detect errors did not end well; most of my character checks on the unsorted String did not sufficiently overlook spaces and divisor signs. Instead, throwing the two exceptions listed above to the GUI and displaying a private error message identified all the errors I could think of.

The FractionTree class has one additional error handling consideration. While the GenericTree class uses no characters except single space delimiters the FractionTree class uses the additional divisor sign. To ensure that we do not have multiple denominator values, the inner Node class of FractionTree has a private boolean method called checkForValidFraction. It returns true if a single divisor sign is present. It otherwise returns false and assists with catching any improper character within the parsed fraction.

Program Testing:

Test Case #	Input	Description	Expected Output	Actual Output	Pass / Fail
1	4 8 2 1 23 16 8 16 3 14 2 10 24	Integer Best case (example)	Ascending: 1 2 2 3 4 8 8 10 14 16 16 23 24 Descending: 24 23 16 16 14 10 8 8 4 3 2 2 1	Ascending: 1 2 2 3 4 8 8 10 14 16 16 23 24 Descending: 24 23 16 16 14 10 8 8 4 3 2 2 1	Pass
2	½ ¾ 5/8 ½ 4/9 7/16 5/32 1/8	Fraction Best case (example)	Ascending: 1/8 5/32 7/16 4/9 1/2 5/8 3/4 3/2 Descending: 3/2 3/4 5/8 1/2 4/9 7/16 5/32 1/8	Ascending: 1/8 5/32 7/16 4/9 1/2 5/8 3/4 3/2 Descending: 3/2 3/4 5/8 1/2 4/9 7/16 5/32 1/8	Pass
3	4 8 6 9 10005 54 64 1 1 100000	Larger Integers	Ascending: 1 1 4 6 8 9 54 64 10,005 100,000 Descending: 100,000 10,005 64 54 9 8 6 4 1 1	Ascending: 1 1 4 6 8 9 54 64 10,005 100,000 Descending: 100,000 10,005 64 54 9 8 6 4 1 1	Pass
4	19/1005 99/3 501/1000 99/201 798/100602	Complex fractions	Ascending: 798/100602 19/1005 99/201 501/1000 99/3 Descending: 99/3 501/1000 99/201 19/1005 798/100602	Ascending: 798/100602 19/1005 99/201 501/1000 99/3 Descending: 99/3 501/1000 99/201 19/1005 798/100602	Pass
5	4 8 2 1 23 16 8 16 3 14 2 10 24	Integer Extra spaces	Ascending: 1 2 2 3 4 8 8 10 14 16 16 23 24 Descending: 24 23 16 16 14 10 8 8 4 3 2 2 1	Ascending: 1 2 2 3 4 8 8 10 14 16 16 23 24 Descending: 24 23 16 16 14 10 8 8 4 3 2 2 1	Pass
6	1/2 3/4 3/2 5/8 4/9 7/16 5/32 1/8	Fraction extra spaces	Ascending: 1/8 5/32 7/16 4/9 1/2 5/8 3/4 3/2 Descending: 3/2 3/4 5/8 1/2 4/9 7/16 5/32 1/8	Ascending: 1/8 5/32 7/16 4/9 1/2 5/8 3/4 3/2 Descending: 3/2 3/4 5/8 1/2 4/9 7/16 5/32 1/8	Pass
7	1 5 7 c w 4 5	Integer non-numeric	Caught exception/error message	Caught exception/error message	Pass
8	a/b 2/7 3/2 c/7	Fraction non-numeric	Caught exception/error message	Caught exception/error message	Pass
9	101, 4, 7, 2, 56, 12, 7	Integer non-space delimit	Caught exception/error message	Caught exception/error message	Pass
10	1/8;7/9;1/1;65/78	Fraction non-space delimit	Caught exception/error message	Caught exception/error message	Pass
11	2/9/7 5/8 7/2 1/2	Double divisor	Caught exception / error message	Caught exception / error message	Pass

Test Case 6:

Test Case 11:

Lessons Learned:

I am not very strong or familiar with the Generics class, so creating the GenericTree was actually quite difficult for me. The FractionTree Class, on the other hand, had so many specific considerations I was able to complete it without too many issues. Type casting was the biggest error that took a large chunk of time to correct. I definitely learned that Generic classes and data parsing do not go well together. More specifically, the class that instantiates an Integer type cast should also ensure that it's passing the same data type as a parameter. Relying on a generic class structure to differentiate between data primitives is risky business.