# Fractions V3 - Summary

In this assignment, you will improve your Fractions V2 program in a multitude of ways. The program will accept four different sorting commands (described below). The program will be even more object oriented in its organization. It will be similar to Fractions V2 in that it will still read a file of Fractions and count the number of fractions of the same value. It will continue to reduce the fraction and have the denominator never be negative. And, the program will handle cases when the fractions file is not found.

The program should be commented with a focus on two types of commenting (described below).

You will be given template files. (See the **Files** section below) `Main.java` will provide a sentinel loop that manages integration with Unit Test engine called, `UnitTestRunner`. While you will not change the code provided, you will add code and helper methods. The place to start your coding is in the method `processCommand`. See the section on `Main.java` for details.

The goals of the assignment is improve the student's ability to implement `CompareTo` and to leverage `Comparable` when sorting. See the section **Comparable** for details.

**Files**:

- `Main.java` : Gets commands from the user and processes them by using the other classes.
- `Fraction.java` : Encapsulates a fraction object, a numerator and denominator.
- `FractionContainer.java` : Holds a single Fraction and how many times the Fraction appears in the file.
- `FractionTracker.java` : Maintains a collection of `FractionContainer` objects. It will sort the list in four different ways. It will display itself as the sorted list in a special way (described below).
- `UnitTestRunner.java` : A helper file that enables unit tests to run.
- `tests_checkpoint1.txt` : A data file that supports the unit tests.
- `fractions#.txt` : 5 different text files with fractions in them that can be used in your tests and are used in the provided unit tests.

## Commands

Your program will process the following commands:

- `sortaf` : Sorts the current list of Fraction Counts in **ascending** order using the Fraction value. (i.e. from small to large fractions)
- `sortdf` : Sorts the current list of Fraction Counts in **descending** order using the Fraction value. (i.e. from large to small fractions)
- `sortac` : Sorts the current list of Fraction Counts in **ascending** order using the **COUNT** of Fractions. Where the counts are identical, it will sort secondarily by the Fraction value.
- `sortdc` : Sorts the current list of Fraction Counts in **descending** order using the **COUNT** of Fractions. Where the counts are identical, it will sort secondarily by the Fraction value.
- <*filename*> : Read the file of fractions in a similar way done in prior Fraction assignments.

The `UnitTestRunner.java` and other provided code enables a special command that will work automatically. You don't need to do anything to support the command.

```
test 1
```

## Main.java

You must keep the following code.

```
public static void main(String[] args) {
      Main main = new Main();
      main.run();
   }

   public Main() {
      tracker = new FractionTracker();
   }

   public void run() {
      boolean done = false;
      Scanner console = new Scanner(System.in);
      while (!done) {
         System.out.print("Command: ");
         String filename = console.nextLine().strip();
         if (filename.equalsIgnoreCase("quit")) {
            System.out.println("All done!");
            done = true;
         } else  if (filename.length() > 0 &&
!UnitTestRunner.processCommand(filename, this::processCommand)) {
            System.out.println(processCommand(filename));
         }
      }
      console.close();
   }
```

You must keep the method `processCommand` with the signature provided. This method will **RETURN** a String. It will **NOT** print anything. You must implement the method.

```
   public String processCommand(String cmd) {
      if (cmd.equalsIgnoreCase("sortaf")) {
         // TODO: ask tracker to sort ascending by Fraction
         return tracker.toString();
      }
      if (cmd.equalsIgnoreCase("sortdf")) {
         // TODO: ask tracker to sort descending by Fraction
         return tracker.toString();
      }
      // TODO: complete the remaining commands

      return "TODO: return output. Don't print!";
   }
```

## Reading the Fraction File

You will read the file and process it in a very similar way that you did in the assignment `FractionsV2`. However, it will be completely refactored to use the classes described in this handout.

You will still have Fraction class that should be virtually unchanged.

Fractions will be a simple numerator and denominator. The denominator will never be negative. The fraction will be automatically reduced.

When a file is read, the order of the fractions will be the same as in prior assignments. This means the first fraction in the list will be the first fraction in the file.

Whenever a file is read, the list of fractions will be returned in the specified format. (See `Sample input/output` below.)

## Miscellaneous

If the user types in the name of a file that is not found, then `processCommand` should return `Error reading file`.

Do not have any `package` usage in your files.

## Comments

This project will focus on two types of comments.

1. Inline comments of value that explain why not what
2. Class responsibilities

### Commenting Why not What

The best comments explain **why** code is present. Secondarily, explanations of **how** are helpful. The idea is to let the code speak for itself when the code is obvious. For example, below is sample code that is **HORRRIBLE!!**

```
// declare an integer and set it to one
int x = 1;
// add it to our sum
sum += x;
// print it out
System.out.println(sum);
```

Instead, we want to have comments of **VALUE**. And, if there is no need for any explanation, then we omit the comment because a useless comment is just noise and annoying!!

Here is the same code with helpful comments:

```
// our first iteration deals with a 1-based indexed matrix.
// Future versions may require that we update this to 0.
int x = 1;
// x will be subracted at the end, so we need to offset
// it here at the start to counter balance the extraneous subtraction.
sum += x ;
// This print is for debugging only.
// TODO: delete me before submitting the code.
System.out.println(sum);
```

> Important: Good inline commenting does **NOT** mean that you need to comment every line. Use comments sparingly. Use comments where they add value and ONLY where they add value.

### Commenting Class Responsibilities

In this project, there are 4 classes. For each class you must provide comments about the responsibilities of the class. Detail important information about what the class does. Don't explain the methods. Simply state what the class is responsible for doing and any other important information for readers of the code. For example:

```
/*
* DiscreteBooger is intended to offer the ability to sneeze and wipe
* before a Booger is picked. We simply override the onBefore() and onAfter()
* methods so that this behavior is offered. This saves the Human class
* a lot of time and embarrassment.
*
* This class will not have any extra instance fields because it
* needs to serialize identically to Booger.
*
* DiscreteBooger knows how to compare itself to any Booger and subclass
* of Booger which allows for powerful Booger sorting.
*/
pubic class DiscreteBooger extends Booger implements Comparable<Booger> {
    // code not shown
}
```

## Comparable

Fractions V3 offers four different types of sorting. This will be accomplished by using `List.sort` or `Collections.sort`. These methods require some assistance in order to sort correctly.

You will store information about the count of equivalent fractions in objects of type, `FractionContainer`. In other words, the class will have a fraction and a count. If you were to sort a list of FractionContainer objects, there are two ways to do this.

**Way #1**

The `FractionContainer` class will implement `Comparable`.

```
public class FractionContainer implements Comparable<FractionContainer>
```

When the `Comparable` interface is implemented, the `FractionTracker` class can sort the array as follows:

```
myFractionContainers.sort(null);
```

When the above `sort` method is called with `null`, the sorting algorithm will ask the FractionContainer objects to sort themselves. This means that the sorting algorithm will call `compareTo` to determine the order.

```
// return 0                  if this == other
// return negative integer   if this < other
// return positive integer   if this > other
public int compareTo(FractionContainer other) { }
```

**Way #2**

The class `FractionContainer` can offer only one implementation of Comparable; there can be only one way to sort itself. To sort in other ways, you will need to provide a `Comparator` interface. There are lots of ways to do this and we simply do not have enough time to learn all the various ways.

To sort backwards, use the "reverse" interface provided by the Collections framework: `Collections.reverseOrder()`. When sort is called with this reverse order inteface, the sorting algorithm will sort the list in the opposite order. It's pretty magical!

To sort primarily by the count of fractions, we need to provide an implemenation of `Comparator`. You will create a `static` method and use *Method Pointers* to fulfill the interface. For example, the below code is a single method that accepts two FractionContainer objects and returns an integer. This meets the requirements necessary to fulfill the `Comparator` interface.

```
   // This method is a gift from Mr. Stride to the student.
   // It is a static method that implements Comparator<> that can be used
   // to help FractionTracker sort in ascending order by the count.
   public static int compareAscendingByCount(FractionContainer fc1,
FractionContainer fc2) {
      // if our count is equal, then use the fraction value
      if (fc1.count == fc2.count) {
         return fc1.compareTo(fc2);
      }

      // just use the count
      return fc1.count - fc2.count;
   }
```

The implementation is short and effective. It first examines the count of fractions of each object. If they are identical, then our primary method of sorting isn't enough. We need to use the secondary method of sorting,

which is the fraction's value. Well, that is easy. We simply use the `compareTo` method.

If the counts are different, we can subtract fc2 from fc1 to get the desired result. If there are more fractions in fc1, then fc1 > fc2 and the result will be positively. Conversely, a negative value is returned when the counts in fc1 < fc2.

Now all we need to do is to sort using the method pointer as follows:

```
myFractionContainers.sort(FractionContainer::compareAscendingByCount);
```

## Sample Input/Output

First, recall that all the printing will happen in the method `main`. You will not code any printing. You will instead create a multi-line String that is returned from `processCommand`.

Here is the input/output you should see when processing the file *fractions5.txt*. Note that the order of the fractions is concruent to the order found in the file.

```
Command: fractions5.txt
2/1 has a count of 3
5/9 has a count of 1
8/9 has a count of 2
3/5 has a count of 1
1/1 has a count of 4
1/10 has a count of 1
1/100 has a count of 1
1/1000 has a count of 1
1/3 has a count of 4
1/2 has a count of 2
1/4 has a count of 2
1/5 has a count of 1
1/6 has a count of 1
1/7 has a count of 1
1/8 has a count of 1
1/9 has a count of 1
2/3 has a count of 1
2/5 has a count of 1
2/7 has a count of 1
2/9 has a count of 1
Total Count = 31
```

Below you'll see how reading a file will immediately output the list of fractions in the order found in the file. The command to sort the list will output the list of fractions in the desired order. In this example, we ask that the list be sorted in descending order by the fraction value (sortdf).

```
Command: fractions3.txt
1/3 has a count of 2
```

```
1/2 has a count of 2
Total Count = 4
Command: sortdf
1/2 has a count of 2
1/3 has a count of 2
Total Count = 4
```

The built-in Unit Tests will result in the following when the user types in `test 1`.

```
Command: test 1
Running Test [fractions1.txt] passed  (+1.0 pts)
Running Test [fractions2.txt] passed  (+1.0 pts)
Running Test [fractions3.txt] passed  (+1.0 pts)
Running Test [not_found file with spaces] passed  (+1.0 pts)
Running Test [fractions4.txt] passed  (+1.0 pts)
Running Test [sortdf] passed  (+1.0 pts)
Running Test [sortac] passed  (+1.0 pts)
Running Test [sortdc] passed  (+1.0 pts)
Running Test [sortaf] passed  (+1.0 pts)
Running Test [fractions5.txt] passed  (+1.0 pts)
Running Test [sortac] passed  (+1.0 pts)
Running Test [sortdf] passed  (+1.0 pts)
Running Test [sortaf] passed  (+1.0 pts)
Running Test [sortdc] passed  (+1.0 pts)
*************************
TOTAL SCORE: 14.0 / 14.0
*************************
```

## Code Submission

You must submit four files to https://css.mrstride.com.
Your 4 files must be named:

- `Main.java`
- `Fraction.java`
- `FractionContainer.java`
- `FractionTracker.java`

## Rubric

> IMPORTANT:
>
> - If your code does not compile, you get 0 points.
> - Do not use Packages.
> - Comments are graded as detailed above.

```
Points     Description
------     ---------------------------
```

```
     14     Pass functional requirements
      6     Language, Library, Design
      5     Commenting (Valuable inline & class responsibilities)
      5     Conventions & style
 ------
     30     TOTAL
```

See UW Quality 142 for Java Coding Conventions.
https://courses.cs.washington.edu/courses/cse142/21au/quality-142.html