Advanced Object Oriented Programming

MOD004883

Student Id: 2112465

# Contents

[**Contents 2**](#_hncdrqrbi2vj)

[**Table of Figures 2**](#_kyupl8y4x9y1)

[**1. Introduction 4**](#_egai642pjavp)

[**2. Application Model 4**](#_7js6j8fb8wu)

[2.1 Requirements 4](#_zcimwekwsyrf)

[2.2 Software Architecture 5](#_ze4umaigtdrp)

[2.3 Flow Diagram 6](#_6cxbac8nt75j)

[2.4 Sequence Diagram 7](#_wfr7ly4hrar)

[2.5 Initial Plan Class Diagram 8](#_to9samvhklqb)

[**3. Implementation 9**](#_pgdi2drnak6r)

[3.1 Final Class Diagram 9](#_joxbzsl13dqi)

[**4. Testing and results 11**](#_6rn8re5ltqb8)

[4.1 Unit Testing 11](#_2ajdjqy0j1op)

[4.2 Functional Testing 19](#_x0tr9rb0ktyt)

[Post Correction Functional Tests 31](#_c1fm7us8rmpq)

[**5. Conclusion 33**](#_qn5mlfxshwd)

[**Appendix - A - Code Samples 34**](#_utukt4vjjpte)

[**References 45**](#_lk2yviomssfj)

# Table of Figures

[Figure 1: Flow Diagram 6](#_bb9tghzrkxf)

[Figure 2: Sequence Diagram 7](#_lh5dad6otoxv)

[Figure 3: Initial Class Diagram 8](#_d58gcofy9ovc)

[Figure 4: Final Class Diagram 9](#_kkysmyo0xt0m)

[Figure 5: Initial output of the method “getFilteredStudentData” from the Class “DataProcessor” 11](#_unrit46bkto)

[Figure 6: After code correction output of the method “getFilteredStudentData” from the Class “DataProcessor” 12](#_ii6xyz6t195p)

[Figure 7: Output of the method “processSubjectData” from the Class “DataProcessor” 13](#_gkgcl2hy0799)

[Figure 8: The amount of students per subject as shown in Numbers 13](#_jlz9stppnqwt)

[Figure 9: Amount of History students as filtered with the search function in IntelliJ Idea 14](#_m20f28hvyv3k)

[Figure 10: The results of the output in Figure 7, in Numbers 14](#_vz82eokszg73)

[Figure 11: The output of the initial tests of the “filterStudentsFromYear” methods from all of the Subject subclasses 15](#_1q39o8a8at5c)

[Figure 12: An extension of Figure 11 Maths student portion 15](#_lynvztrcqdnm)

[Figure 13: Students from the specified year in the test, filtered in Numbers in order to cross-validate the outputted data 15](#_ctdzwf11hjwr)

[Figure 14: The output of the post code corrected tests of the “filterStudentsFromYear” methods from all of the Subject subclasses 16](#_xoaql3n9828i)

[Figure 15: The output of the tests of the “filterFailedStudents” methods from all of the Subject subclasses, displaying all Students who ever failed 17](#_aojo5i266dl6)

[Figure 16: An extension of Figure 15 displaying the final outputted Geography Student 17](#_nhxrhlico3px)

[Figure 17: Filtered students who failed ever within Numbers software 17](#_urov2n7acqbg)

[Figure 18: Output of the tests of the “getGrade” method from the Enums class 18](#_hqns5xrdbxkx)

[Figure 19: Output of initial functional test 1 (entering an invalid option) 20](#_plsndua77s51)

[Figure 20: Output of functional test 2 (entering a capitalised option) 21](#_tzr7rblw4vb7)

[Figure 21: Output of initial functional test 3 (Display any student who ever failed, with final results in results format and with the highest mark first) 21](#_cigy1umcpo6y)

[Figure 22: Snippet of the output of functional test 4 (Maths students who graduated in 2003, displayed in surname alphabetical order with final results in grade format) 22](#_79vqf711et2e)

[Figure 23: The equivalent snippet of Figure 22, filtered through Numbers for cross validating the output 23](#_gums9wnp2c67)

[Figure 24: Snippet of the output of functional test 5 (Display History students from every year, starting with the highest final result first, in result format) 24](#_yrtsafkieayd)

[Figure 25: Results of a student in figure 24 for data validation 24](#_rk93n2t4ca1g)

[Figure 26: Results of a student in figure 24 for data validation 25](#_c59h6hhnzxfv)

[Figure 27: Snippet of the History output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format) 25](#_94r90i9x39tf)

[Figure 28: Snippet of the History output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format) 26](#_g6hqp9erl3g)

[Figure 29: Snippet of the Maths output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format) 26](#_j19rcjinuayy)

[Figure 30: Snippet of the Maths subject output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format) 27](#_mc6d2mxnv8yj)

[Figure 31: Snippet of the Geography output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format) 27](#_j6o2hkeapfjr)

[Figure 32: Snippet of the Geography subject output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format) 28](#_6b6jv15276rb)

[Figure 33: Output of functional test 7 (Close the programme) 29](#_m3wht4iz0her)

[Figure 34: Output of functional test 7 (loop back to the initial menu) 30](#_iz05xccyo8as)

[Figure 35: Output of the repeated functional test 1 (Enter an invalid option) 32](#_r626zactjd9k)

[Figure 36: Output of the repeated functional test 3 (Display any student who ever failed, with final results in results format and with the highest mark first) 32](#_ga8h3aotmw22)

# Introduction

This is the documentation of a console based student result managing application. The premise of the application is to process a comma separated value file and allow the user to filter and process the student result data. For more information on the features please see section 2.1.

Within the documentation the architecture, models, diagrams, implementation and testing is included. The intended user base for this application would be teachers within educational facilities that utilise result based curriculums, as it quickly filters large quantities of student data.

# Application Model

## 2.1 Requirements

Basic Requirements

* Read in a text file of student results.
* Print out three lists for each year – one for each type of student. The lists should contain the student’s name (First name, then Surname) along with their final result.

Further Requirements

* A menu based system to choose options
* The option to print out student results for one subject and one year
* The option to print results in alphabetical order of student surname for each subject and year
* The option to print results in descending order (Highest first) for each subject and year
* The option to include “Grades” - <40% Fail, 40%-59% Pass, 60%-69% Merit, >69% Distinction
* The option to find any students who have failed an individual module (Less than 40%)

## 2.2 Software Architecture

The software architecture that I have opted for is Model View Presenter (MVC). This is because most models, such as MVVM (Model View View-Model) and Model View Controller (MVP) have caveats which are not suited to this project or are not preferable. MVVM is just too complicated and implementing it would be inefficient since it is more suited to complex user interfaces, whereas this is a simple menu within the terminal. MVC and MVP are more applicable, however MVP is usually better for User Interface designs and as previously stated this is a simple command line interface.

Utilising this architecture means that there is a level of encapsulation that isolates errors into one “section” for instance, an error with data processing where incorrect student results are returned, would be related to the “model” (DataProcessor object). Thus making debugging easier.

Furthermore, using a preset architecture means that in most cases it will be easier to pass the code to another team or to work with others, as an architecture that other developers are accustomed to will make for faster understanding of the code. So if the application was to be expanded for usage on a large scale with many features, it would be much easier than not using an architecture.

## 2.3 Flow Diagram

###### 

###### Figure 1: Flow Diagram

This flow model was opted for due to it possessing user input in what is believed to be the most efficient yet easy to understand selection style. Every decision or selection is built upon the previous one, ultimately resulting in the relevant data in accordance to the specified filters and/or order. It may not be deemed the most efficient model, as it only asks for one specific filter per question, however this way was opted for so that the user is not overwhelmed with options.

Every requirement ranging from the further requirements to the basic ones are covered in the model. In order to put into perspective the merging of this flowchart with the proposed architecture, the sequence diagram in figure 2 was created.

## 2.4 Sequence Diagram

###### 

###### Figure 2: Sequence Diagram

This diagram displays the architecture in further detail, the GuiController will handle all user interactions, whereas the “main” class (the “controller” in the MVC architecture) is in charge of the main code logic. The dataProcessor (the “model” in MVC architecture) only being utilised, for the initial data reading and when required by the Controller. This level of separation allows for easier understanding if the software was to be created by a team but also it has decomposed the initial problem to the extent to which implementation should not be viewed as an impossible or even difficult task.

## 2.5 Initial Plan Class Diagram

###### 

###### Figure 3: Initial Class Diagram

Figure 3 is documented as the initial class diagram, due to the fact that in implementation problems may arise or further analysis could potentially lead to tweaks within classes and thus the class diagram.

Object oriented programming (OOP) principles have been implemented throughout the design. For instance both the Student and Subject classes are Superclasses to three subclasses, which inherit attributes and methods from the Superclass, making it so that the methods and attributes are not required to be placed in each class separately. Encapsulation is also present consistently with separate classes for separate use cases, in which the attributes are only externally available through getters and setters. Elements of abstraction are also present throughout, most prevalent in the “DataProcessor” where the user is completely unaware of all of the filtering. This is completed without any need for the user to interfere and all they are required to do is choose their parameters.

# Implementation

The application was implemented utilising Java and the IntelliJ Idea development environment.

## 3.1 Final Class Diagram

###### 

###### Figure 4: Final Class Diagram

Upon implementation a few changes arose. One was to implement the “filterSubjectData” methods within the “DataProcessor”. The job of the methods is to decrease the amount of code within the “main” class and to keep to the MVC architecture, as otherwise data processing would occur outside of the specified class since the data would have to be retrieved from the subject class and then combined with the other user specified filters within the main class. The three methods have the same name implementing polymorphism to make the usage exactly the same across each subject while entering different parameters. This will make it easier for code usage and readability within a team or even for just one person.

The second implemented change was the movement of “returnStudentsInResultDescendingOrder” and “returnStudentsInAlphabeticalOrder” from the “Subject” subclasses to the Superclass. The reason this was changed is because in the original class diagram (Figure 3) it was planned to have them within every subclass of “Subject”. The logic being that each input would be a different Student subclass so you were unable to place it within the Superclass. However, upon research into the matter, java wildcards were seen and attention to the Liskov substitution principle was taken into account (GeeksforGeeks, 2022). This meant that these methods were able to be moved into the Superclass, with the usage of upcasting to return the inputs and wildcards to accept the inputs. This is deemed as completely acceptable since the fields that become inaccessible are no longer required by the time ordering data is required as seen in Figure 2.

Finally the nested classes “StudentDescendingFinalResultComparator” and

“StudentSurnameAlphabeticalOrderComparator” were added to the “Student” Superclass. This is because during research comparators were stated as a pre-existing way to compare values within classes and create a sorted data structure. This was exactly what was required for ordering the students and required much less code than implementing a merge sort or any other sorting algorithm from scratch (Baeldung, 2023).

# Testing and results

## 

## 4.1 Unit Testing

#### DataProcessor - getFilteredStudentData

#### 

This method processes the data in the CSV file and returns an Array that holds two smaller arrays. Each entry is one student, the first Array being student metadata and the second Array being their results.

##### Initial Method Code

SEE APPENDIX - A CODE SAMPLE 1

##### Initial Method Testing Code

SEE APPENDIX - A CODE SAMPLE 2

##### Initial Method Testing Output

###### 

###### Figure 5: Initial output of the method “getFilteredStudentData” from the Class “DataProcessor”

From the output it becomes apparent that the file with the real data functions correctly outputting one array holding two smaller arrays within it. One smaller array holding the students’ results, the other the students’ metadata.

The test in which an invalid path was provided also was successful, returning the correct error message and not printing the data since null was to be returned by the method. However, with the blank file it is seen that the method still returns null, as the “BLANK:[ARRAY VALUES]” is not printed, however it does not print the relevant error message, meaning that it is not performing as expected. Upon closer inspection it was found that the blank error was due to a logic error, which was promptly fixed.

This is also the point at which the realisation of adding a method to the “GUIController” class to display error messages arrived, since printing to the terminal though effective to the developer, dependent on the front end may not ever inform the user.

##### Post Testing Method Code

SEE APPENDIX - A CODE SAMPLE 3

##### Post Method Testing Output

###### 

###### Figure 6: After code correction output of the method “getFilteredStudentData” from the Class “DataProcessor”

Different from the initial output in Figure 5, an extra line is printed stating “BLANK”. This is the expected output and the logic error that was present was simply that a blank file does not trigger errors, so a blank check must be executed with a code block rather than attempting to catch an error.

#### DataProcessor - processSubjectData

#### 

This method takes the filtered student data returned by the “getFilteredStudentData” method and creates the relevant objects. Logic wise, it will go through each array, create the student object, insert the relevant data into the object and add the student into the subject that they take.

##### Method Code

SEE APPENDIX - A CODE SAMPLE 4

##### Method Testing Code

SEE APPENDIX - A CODE SAMPLE 5

##### Method Testing Output

###### 

###### Figure 7: Output of the method “processSubjectData” from the Class “DataProcessor”

Here the output based on the code, means that Braulio is a history student from the year 2000 and that there are 320 geography, 349 history and 331 maths students . Now to ensure that the data is correct at least amount wise, other software such as “Numbers” can be utilised to find out these values.

###### 

###### Figure 8: The amount of students per subject as shown in Numbers

While cross checking data from Figure 7 and Figure 8, it seems as though we have an issue as the programme's History value does not match with the value in numbers. However upon closer inspection it seems as though this is Numbers software related as, when utilising the find and replace function and other software, it becomes apparent that there are in fact 349 History students. To further reiterate this, Numbers was utilising the first entry as “Headers” so therefore the value was not counted.

###### 

###### Figure 9: Amount of History students as filtered with the search function in IntelliJ Idea

###### 

###### Figure 10: The results of the output in Figure 7, in Numbers

Cross checking data from Figure 7 and Figure 10, the values match and are therefore correct, meaning that this method functions as it should.

#### All Subject Subclasses - filterStudentsFromYear

#### 

The purpose of these methods is to take a graduation year as its parameter and then return only the students who graduated within that year, using the student list that exists within the class. All of the methods should function exactly the same with the only difference being the type of student returned.

##### Initial Method Code

SEE APPENDIX - A CODE SAMPLE 6

##### Method Testing Code

SEE APPENDIX - A CODE SAMPLE 7

##### Initial Method Testing Output

###### 

###### Figure 11: The output of the initial tests of the “filterStudentsFromYear” methods from all of the Subject subclasses

###### 

###### Figure 12: An extension of Figure 11 Maths student portion

In Figure 11 it can be seen that Maths functions as it should, outputting matching values to the ones in Figure 13. It can also be observed in Figure 11 that empty arrays are returned instead of a null alternative. The required action so that the method functions as it should, is to implement an if statement to check whether or not there are values within the data object, before returning it.

###### 

###### Figure 13: Students from the specified year in the test, filtered in Numbers in order to cross-validate the outputted data

##### Method Code

SEE APPENDIX - A CODE SAMPLE 8

##### Method Testing Code

SEE APPENDIX - A CODE SAMPLE 7

##### Method Testing Output

###### 

###### Figure 14: The output of the post code corrected tests of the “filterStudentsFromYear” methods from all of the Subject subclasses

This is the expected output now, since it returns a null alternative instead of an empty array and the correct values are still outputted.

#### All Subject Subclasses - filterFailedStudents

#### This method should take a parameter which is a list of Students, then return a list of Students, who have failed a single module.

#### Method Code

SEE APPENDIX - A CODE SAMPLE 9

##### Method Testing Code

SEE APPENDIX - A CODE SAMPLE 10

##### Method Testing Output

###### Figure 15: The output of the tests of the “filterFailedStudents” methods from all of the Subject subclasses, displaying all Students who ever failed

###### 

###### 

###### Figure 16: An extension of Figure 15 displaying the final outputted Geography Student

###### 

###### Figure 17: Filtered students who failed ever within Numbers software

The output in Figure 15 and Figure 16, displays an overall amount of 3 students who failed a subject. When checked against the same filters in Numbers, the output in Figure 17 matches to the output of the programme, thus meaning that the methods are working as they should.

#### Enums - getGrade

This method simply takes an integer as a parameter and returns the predefined grade equivalent of the number.

#### Method Code

SEE APPENDIX - A CODE SAMPLE 11

##### Method Testing Code

SEE APPENDIX - A CODE SAMPLE 12

##### Method Testing Output

###### 

###### Figure 18: Output of the tests of the “getGrade” method from the Enums class

Requirement Grades : <40% = Fail, 40%-59% = Pass, 60%-69% = Merit, >69% = Distinction

Comparing the stated grades in the requirements to the output, the getGrades method is working correctly since the numbers within or on the boundaries within the programme output, correspond with the predefined boundaries .

###### 

## 

## 

## 4.2 Functional Testing

| Test Number | Test  Case | Test  Data | Expected  Result | Date | Success Rate |
| --- | --- | --- | --- | --- | --- |
| 1 | Enter invalid option | q | Output error message “invalid option”  Allows another input | 08/12/2023 | Fail |
| 2 | Enter capitalised valid option | A | Should continue to the next question “Select a year” | 08/12/2023 | Success |
| 3 | Criteria:  All students who failed any module  In results format  In highest final result order | D  G  B  B  B | Should output all students who failed any module, for all subjects in three separate lists.  With the final grade in number format and the students ordered with the highest final grade first | 09/12/2023 | Fail |
| 4 | Criteria:  Maths students  Graduated in 2003  Alphabetical order  Grade format | A  D  A  A  A | Output the data with the same criteria specified in the test case | 09/12/2023 | Success |
| 5 | Criteria:  History students  All years  Descending order  Result format | c  g  a  b  b | Output the data with the same criteria specified in the test case | 09/12/2023 | Success |
| 6 | Criteria:  All students  All years  Descending order  Result format | D  G  A  B  B | Output the data with the same criteria specified in the test case | 09/12/2023 | Successful |
| 7 | Closing the programme loop | D  D  B  B  B  A | The initial menu should reappear | 11/12/2023 | Successful |
| 8 | Continuing the programme loop | D  D  B  B  B  B | The programme should end and not show the menu again | 11/12/2023 | Successful |

#### Test 1

##### Outcome

###### 

###### Figure 19: Output of initial functional test 1 (entering an invalid option)

The programme does allow you to re-input a value, however it does not display the “invalid option” message.

##### Action

Implement the “invalid option” message.

#### Test 2

##### Outcome

###### 

###### Figure 20: Output of functional test 2 (entering a capitalised option)

##### Action

None required.

#### Test 3

##### Outcome

###### 

###### Figure 21: Output of initial functional test 3 (Display any student who ever failed, with final results in results format and with the highest mark first)

##### Action

Implement a catch to process the null error, or prevent it even reaching the error by checking if the parameter entered is null. The desired outcome is to have null returned.

#### Test 4

##### Outcome

###### 

###### Figure 22: Snippet of the output of functional test 4 (Maths students who graduated in 2003, displayed in surname alphabetical order with final results in grade format)

###### 

###### Figure 23: The equivalent snippet of Figure 22, filtered through Numbers for cross validating the output

The outputted grades in Figure 22 seem suspicious at first due to the same grade being stated for every seen value, however upon filtering the data set within Numbers software, it matches the data that is present from the output so this test is successful.

##### Action

None required.

#### Test 5

##### Outcome

###### 

###### Figure 24: Snippet of the output of functional test 5 (Display History students from every year, starting with the highest final result first, in result format)

###### 

###### Figure 25: Results of a student in figure 24 for data validation

###### 

###### Figure 26: Results of a student in figure 24 for data validation

Final Result of Figure 26: Ceiling of (87+87+86+79+78+79+81+75+86+76)/10 = 82.

Final Result of Figure 25: Ceiling of (89+72+87+75+88+86+89+89+84+71)/10 = 83.

Checking the data in Figure 24 against the Numbers software data in Figures 25 and 26 the final result values are correct and the observation is that Figure 24 has ranked the students highest first, just as requested, thus this is a successful test.

##### Action

None required.

#### Test 6

##### Outcome

###### 

###### Figure 27: Snippet of the History output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format)

###### 

###### Figure 28: Snippet of the History output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format)

###### 

###### 

###### Figure 29: Snippet of the Maths output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format)

###### 

###### Figure 30: Snippet of the Maths subject output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format)

###### 

###### Figure 31: Snippet of the Geography output of functional test 6 (Display all students from every year, starting with the highest final result first, in result format)

###### 

###### Figure 32: Snippet of the Geography subject output of from Numbers using the test 6 criteria (Display all students from every year, starting with the highest final result first, in result format)

After checking the criteria output in Numbers versus the programme output, it is evident that the rankings are the same, thus proving a successful test. Figures 27,28,29,30,31 and 32 are identical in output except the programme provides abstracted output.

##### Action

None required

#### Test 7

##### Outcome

###### 

###### Figure 33: Output of functional test 7 (Close the programme)

Process finished with exit code 0 in Figure 33, indicates the programme ended with no errors.

##### Action

None required, the test was successful.

#### Test 8

##### Outcome

###### 

###### Figure 34: Output of functional test 7 (loop back to the initial menu)

###### 

This was completely expected behaviour in Figure 34, the menu has in fact reappeared, so no action is required.

##### Action

None required.

## 

## 

## 

## 

## 

## 

## Post Correction Functional Tests

| Test Number | Test  Case | Test  Data | Expected  Result | Date | Success Rate |
| --- | --- | --- | --- | --- | --- |
| 1 | Enter invalid option | q | Output error message “invalid option”  Allows another input | 12/12/2023 | Success |
| 3 | Criteria:  All students who failed any module  In results format  In highest final result order | D  G  B  B  B | Should output all students who failed per subject in three separate lists.  With the final grade in number format and the students starting with the highest final grade first | 12/12/2023 | Success |

#### Repeated Test 1

##### Outcome

###### 

###### Figure 35: Output of the repeated functional test 1 (Enter an invalid option)

In comparison to Figure 19, there is now an “invalid answer” message present, just as expected.

#### Repeated Test 3

###### 

###### Figure 36: Output of the repeated functional test 3 (Display any student who ever failed, with final results in results format and with the highest mark first)

###### 

In Figure 36 instead of outputting a null pointer exception as the previous test did, the output is now “No Students”, which is much better as it does not halt the programme and is how the programme should function.

# Conclusion

The application functions correctly with all specified requirements met. This however does not mean that there are no improvements to be made, for example the current method requires a series of “if” statements within the main logic. Given enough subjects, that series of statements could become extremely lengthy and hard to work with. Thus other methods may be more effective such as storing subjects within data structures or having factory design patterns for subjects and maybe even students.

In the future if this programme was to be utilised within a school or University, it could potentially benefit from creating a separate class, for organising students, in which comparators are stored. Currently there are only two comparators so having them nested within the relevant Student class is viable, however if the data is wanting to be sorted by other values then a separate class would be a better approach, abiding to the Single Responsibility Principle and keeping cleaner code. In regards to useability it could also benefit from cleaner command line output or a whole different user interface, as for the less tech literate teachers the software may prove difficult to interpret or use.

# Appendix - A - Code Samples

Code Sample 1:

public ArrayList<List<List<String>>> getFilteredStudentData(String filepath){

//using bufferedReader because most efficient way to read file

try {

BufferedReader bufferedReader = new BufferedReader(new FileReader(filepath));

//1 dimensional array for the data

ArrayList<List<List<String>>> filteredStudentData = new ArrayList<>();

//while there is a new line in the csv, split the line at the comma

String currentLine;

while ((currentLine = bufferedReader.readLine()) != null){

//data manipulation(separation)

String unfilteredStudentRecord = currentLine;

//remove unnecessary spaces

String filteredStudentRecord = unfilteredStudentRecord.replaceAll("\\s", "");

//split data into a list

List<String> commaSplitStudentRecord = Arrays.asList(filteredStudentRecord.split(","));

//split list into two parts, meta data and student marks

List<String> studentMetaData = commaSplitStudentRecord.subList(0,4);

List<String> studentMarks = commaSplitStudentRecord.subList(4,commaSplitStudentRecord.size());

//create array lists

List<List<String>> filteredRecordData = new ArrayList<>();

filteredRecordData.add(studentMetaData);

filteredRecordData.add(studentMarks);

//place the lists inside the dataslot

filteredStudentData.add(filteredRecordData);

}

//finished with the file, close the stream

bufferedReader.close();

//return relevant data

return filteredStudentData;

}

catch (FileNotFoundException exception){

//file error

return null;

}

catch (

IOException e) {

//no line to print

System.out.println("BLANK FILE");

return null;

}

}

Code Sample 2:

//csv file path for usage by the DataProcessor

String filepath = "src/Resources/Data\_File\_For\_Assignment.csv";

//class used to filter and read the csv

DataProcessor dataProcessor = new DataProcessor();

//get the filtered data from designated file path

List<List<List<String>>> filteredStudentData = dataProcessor.getFilteredStudentData(filepath);

if (filteredStudentData != null) {

System.out.println("REALDATA: " + filteredStudentData);

}

else{

//data error

}

//csv file path for usage by the DataProcessor

String filepath2 = "src/Resources/nil.csv";

//get the filtered data from designated file path

List<List<List<String>>> filteredStudentData2 = dataProcessor.getFilteredStudentData(filepath2);

if (filteredStudentData2 != null) {

System.out.println("nil: "+filteredStudentData2);

}

else{

//data error

}

//csv file path for usage by the DataProcessor

String filepath3 = "src/Resources/blank.csv";

//get the filtered data from designated file path

List<List<List<String>>> filteredStudentData3 = dataProcessor.getFilteredStudentData(filepath3);

if (filteredStudentData2 != null) {

System.out.println("BLANK: " + filteredStudentData3);

}

else{

//data error

}

Code Sample 3:

public ArrayList<List<List<String>>> getFilteredStudentData(String filepath){

//using bufferedReader because most efficient way to read file

try {

BufferedReader bufferedReader = new BufferedReader(new FileReader(filepath));

//1 dimensional array for the data

ArrayList<List<List<String>>> filteredStudentData = new ArrayList<>();

//while there is a new line in the csv, split the line at the comma

String currentLine;

while ((currentLine = bufferedReader.readLine()) != null){

//data manipulation(separation)

String unfilteredStudentRecord = currentLine;

//remove unnecessary spaces

String filteredStudentRecord = unfilteredStudentRecord.replaceAll("\\s", "");

//split data into a list

List<String> commaSplitStudentRecord = Arrays.asList(filteredStudentRecord.split(","));

//split list into two parts, meta data and student marks

List<String> studentMetaData = commaSplitStudentRecord.subList(0,4);

List<String> studentMarks = commaSplitStudentRecord.subList(4,commaSplitStudentRecord.size());

//create array lists

List<List<String>> filteredRecordData = new ArrayList<>();

filteredRecordData.add(studentMetaData);

filteredRecordData.add(studentMarks);

//place the lists inside the dataslot

filteredStudentData.add(filteredRecordData);

}

//finished with the file, close the stream

bufferedReader.close();

//return relevant data

if(filteredStudentData.size() > 0){

return filteredStudentData;

}

else{

//blank file

return null;

}

}

catch (FileNotFoundException exception){

//file error

return null;

} catch (IOException e) {

return null;

}

}

Code Sample 4:

public boolean processSubjectData(List<List<List<String>>> filteredStudentData){

boolean processedData = false;

//loop through each filtered data record if the filtered student data exists

if (filteredStudentData != null) {

processedData = true;

for (int recordIndex = 0; recordIndex < filteredStudentData.size(); recordIndex++) {

List<List<String>> dataRecords = filteredStudentData.get(recordIndex);

List<String> metaData = dataRecords.get(0);

List<String> marksList = dataRecords.get(1);

int[] marks = new int[marksList.size()];

//cast list array to array to store in the student subclasses

for(int markListIndex=0;markListIndex<marksList.size();markListIndex++){

marks[markListIndex] = Integer.valueOf(marksList.get(markListIndex));

}

//assigning each piece of data to easy to understand values

String year = metaData.get(0), subjectName = metaData.get(1), lastName = metaData.get(2),

firstName = metaData.get(3);

//Create the relevant student object for the correct subject and append it to the subject's list of students

if(subjectName.equals("History")){

HistoryStudent student = new HistoryStudent(lastName,firstName,year,marks);

history.addStudent(student);

}

else if(subjectName.equals("Maths")){

MathsStudent student = new MathsStudent(lastName,firstName,year,marks);

maths.addStudent(student);

}

else if(subjectName.equals("Geography")) {

GeographyStudent student = new GeographyStudent(lastName,firstName,year,marks);

geography.addStudent(student);

}

}

}

return processedData;

}

Code Sample 5:

//csv file path for usage by the dataInterpreter

String filepath = "src/Resources/Data\_File\_For\_Assignment.csv";

//class used to filter and read the csv

DataProcessor dataProcessor = new DataProcessor();

//get the filtered data from designated file path

List<List<List<String>>> filteredStudentData = dataProcessor.getFilteredStudentData(filepath);

boolean processedData = dataProcessor.processSubjectData(filteredStudentData);

if (processedData) {

for (int i=0;i<filteredStudentData.size();i++){

System.out.println(i+1+": "+filteredStudentData.get(i));

}

String firstName = dataProcessor.getHistory().getStudents().get(0).getFirstName();

Integer studentYear = dataProcessor.getHistory().getStudents().get(0).getStudentYear();

System.out.println("History Student: " + firstName +" " + studentYear);

System.out.println("Amount of geog students: "+ dataProcessor.getGeography().getStudents().size());

System.out.println("Amount of history students: "+ dataProcessor.getHistory().getStudents().size());

System.out.println("Amount of maths students: "+ dataProcessor.getMaths().getStudents().size());

}

Code Sample 6:

/\*\* Returns students who graduated maths in the provided year \*/

public List<MathsStudent> filterStudentsFromYear(int year) {

//holder to place the students from a certain year in

List<MathsStudent> filteredStudents = new ArrayList<>();

//loop through each student and if it is within the year then add it to the filtered list

for(MathsStudent student : this.students){

if(student.getStudentYear() == year){

filteredStudents.add(student);

}

}

return filteredStudents;

}

Code Sample 7:

//class used to process and read the csv

DataProcessor dataProcessor = new DataProcessor();

//prepare the GUI for the user

GuiController commandLineGui = new GuiController();

//csv file path for the dataProcessor to find the right file

String filepath = "src/Resources/Data\_File\_For\_Assignment.csv";

//get the student data from designated file path

List<List<List<String>>> filteredStudentData = dataProcessor.getFilteredStudentData(filepath);

//place the student data into subjects

boolean processedData = dataProcessor.processData(filteredStudentData);

//if there is student data and subjects to process further then trigger the user menu

// else print an error with the data

if (processedData) {

List<MathsStudent> students = dataProcessor.getMaths().filterStudentsFromYear(2003);

System.out.println("Maths:2003 ");

System.out.println(students);

List<HistoryStudent> student2 = dataProcessor.getHistory().filterStudentsFromYear(2006);

System.out.println("history:2006 ");

System.out.println(student2);

List<GeographyStudent> student3 = dataProcessor.getGeography().filterStudentsFromYear(2012);

System.out.println("geography:2012 ");

System.out.println(student3);

} else {

commandLineGui.errorMessage("Error with the data file");

}

Code Sample 8:

/\*\* Returns students who graduated maths in the provided year \*/

public List<MathsStudent> filterStudentsFromYear(int year) {

//holder to place the students from a certain year in

List<MathsStudent> filteredStudents = new ArrayList<>();

//loop through each student and if it is within the year then add it to the filtered list

for(MathsStudent student : this.students){

if(student.getStudentYear() == year){

filteredStudents.add(student);

}

}

//return the filtered list if it has items in, otherwise return null

if(filteredStudents.size() > 0){

return filteredStudents;

}

return null;

}

Code Sample 9:

/\*\* Returns students who failed a single module in maths \*/

public List<MathsStudent> filterFailedStudents(List <MathsStudent> studentList){

//used to store the filtered data

List<MathsStudent> filteredList = new ArrayList<>();

//loop through each student looking for a failed module

for (MathsStudent student : studentList){

boolean failedAModule = false;

//loop through each mark to see if the student failed a module

for(int mark : student.getMarks()){

if(mark < 40){

//if they did the flag the earlier boolean and break the loop since they failed

failedAModule = true;

break;

}

}

//if the student failed a module add them to the filtered list since they fit criteria

if(failedAModule){

filteredList.add(student);

}

}

//if there are objects in the filtered list then return the list, otherwise null for easy usage

if(filteredList.size() > 0){

return filteredList;

}

return null;

}

Code Sample 10:

//class used to process and read the csv

DataProcessor dataProcessor = new DataProcessor();

//prepare the GUI for the user

GuiController commandLineGui = new GuiController();

//csv file path for the dataProcessor to find the right file

String filepath = "src/Resources/Data\_File\_For\_Assignment.csv";

//get the student data from designated file path

List<List<List<String>>> filteredStudentData = dataProcessor.getFilteredStudentData(filepath);

//place the student data into subjects

boolean processedData = dataProcessor.processData(filteredStudentData);

//if there is student data and subjects to process further then trigger the user menu

// else print an error with the data

if (processedData) {

List<MathsStudent> students = dataProcessor.getMaths().filterFailedStudents(dataProcessor.getMaths().getStudents());

System.out.println("Maths:All ");

System.out.println(students);

List<MathsStudent> students4 = dataProcessor.getMaths().filterFailedStudents(dataProcessor.getMaths().filterStudentsFromYear(2003));

System.out.println("Maths:2003 ");

System.out.println(students4);

List<HistoryStudent> student2 = dataProcessor.getHistory().filterFailedStudents(dataProcessor.getHistory().getStudents());

System.out.println("history:All ");

System.out.println(student2);

List<HistoryStudent> students5 = dataProcessor.getHistory().filterFailedStudents(dataProcessor.getHistory().filterStudentsFromYear(2004));

System.out.println("Maths:2004 ");

System.out.println(students5);

List<GeographyStudent> student3 = dataProcessor.getGeography().filterFailedStudents(dataProcessor.getGeography().getStudents());

System.out.println("geography:All ");

System.out.println(student3);

List<GeographyStudent> students6 = dataProcessor.getGeography().filterFailedStudents(dataProcessor.getGeography().filterStudentsFromYear(2001));

System.out.println("Maths:2001 ");

System.out.println(students6);

} else {

commandLineGui.errorMessage("Error with the data file");

}

Code Sample 11:

/\*\* Returns the grade that is equal to the entered result \*/

public grades getGrade(int averageMark){

if (averageMark < 40){

return grades.Fail;

}

else if(averageMark > 39 && averageMark < 60){

return grades.Pass;

}

else if(averageMark > 59 && averageMark < 70){

return grades.Merit;

}

else if(averageMark > 69 ){

return grades.Distinction;

}

return null;

}

Code Sample 12:

Enums enumClass = new Enums();

System.out.println("40:" + enumClass.getGrade(40));

System.out.println("39:" +enumClass.getGrade(39));

System.out.println("70:" +enumClass.getGrade(70));

System.out.println("60:" +enumClass.getGrade(60));

System.out.println("90:" +enumClass.getGrade(90));

# References

Baeldung. (2023). Comparator and Comparable in Java. Baeldung. Available at:<https://www.baeldung.com/java-comparator-comparable>.

GeeksforGeeks. (2022). Wildcards in Java. GeeksforGeeks. Available at:<https://www.geeksforgeeks.org/wildcards-in-java/>.