Ahmed El-Hawat
21831369

University Of Brighton

Reflective Report

Student Management System

Table of Contents

Introduction

The student management system I have produced for this assignment was developed in Java using mostly its functional programming paradigm incorporating JavaFX and SceneBuilder to produce a user-friendly Graphical User Interface (GUI), it is a CRUD (Create Read Update Delete) application with some additional features to manipulate data and to benchmark the performance of the different streams.

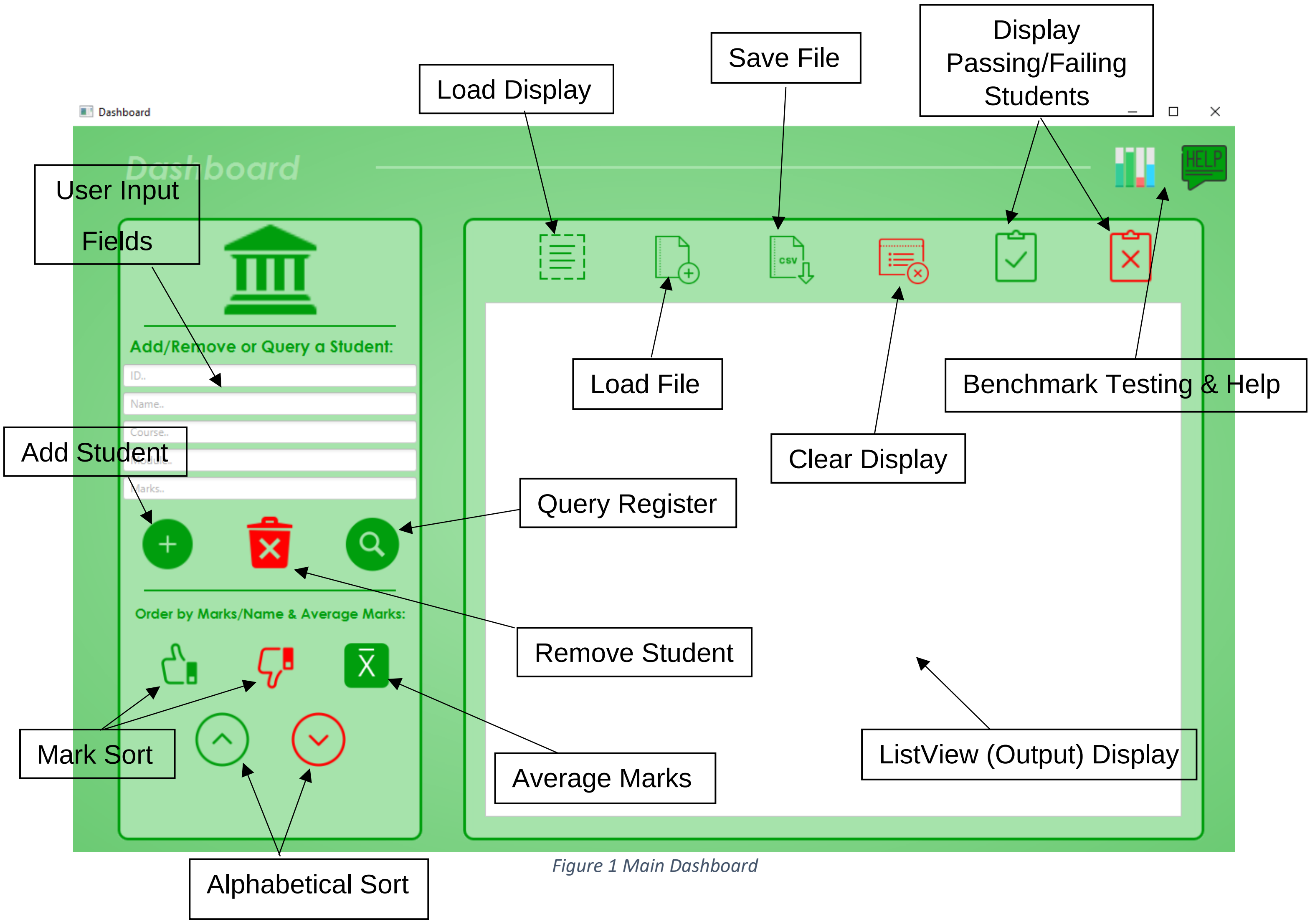


Figure 1 Main Dashboard

To get started you can either load a previous register in the form of a .txt file or you can start a new register that you will be able to save to your hard drive, once you have loaded a file to the application it registers the file into a ConcurrentHashMap saving the details of the students in memory allowing you to clear or load the display as well as manipulate the data without necessarily changing the original file/map – there is a save button which will open a directory browser allowing you to save a new file or overwrite an existing one, in a real-world situation this would allow different tutors/staff to see different views of students from one dataset.

The basic design incorporates input fields that allow the user to enter the details of a student they would like to add, remove or update while simultaneously allowing for search functionality.

There are accessibility features by providing sorting options that will sort the data either by name or marks in ascending or descending order, for a more streamlined user experience.

Furthermore, there are features that allow you to manipulate the data by calculating the average marks of students as well as all those who have passed and all those who have failed of a queried data set again without altering the original ConcurrentHashMap.

There is also a brief help section outlining the basic operations of the application as well as a BenchMark test where I compare sequential and parallel streams when filtering different sized data sets using more complex conditions.

Capabilities and Features:

→ Data Entry and Management – The system allows for addition and removal of student records.
→ Data Persistence and Concurrency - The system allows for users to open and save files in a .txt format, when loaded the files data will be stored into a ConcurrentHashMap ensuring efficient retrieval and update operations in constant or *O(1)* time and should also be capable of handling multiple threads or users concurrently.
→ Interactive GUI and Metrics – A user-friendly GUI that will display data into a ListView. Allows the student data to be manipulated into ordered data by their marks or names, and features to display all passing and failing students as well as the average marks from a queried dataset allowing the user to view data in various formats.
→ Searching and Filtering – The system also supports a search functionality using Java's Stream API that allows users to filter student records based on multiple criteria such as the student's ID, name, course, module and marks then displaying the filtered data to the ListView.
→ Data Validation and Error Handling – The system provides extensive mechanisms for handling invalid inputs and other errors that can occur, it provides user-friendly messages in the event of issues.
→ Stream Performance Testing – The system includes a benchmarking class that compares the performance of Stream vs ParallelStream when processing different sized datasets.

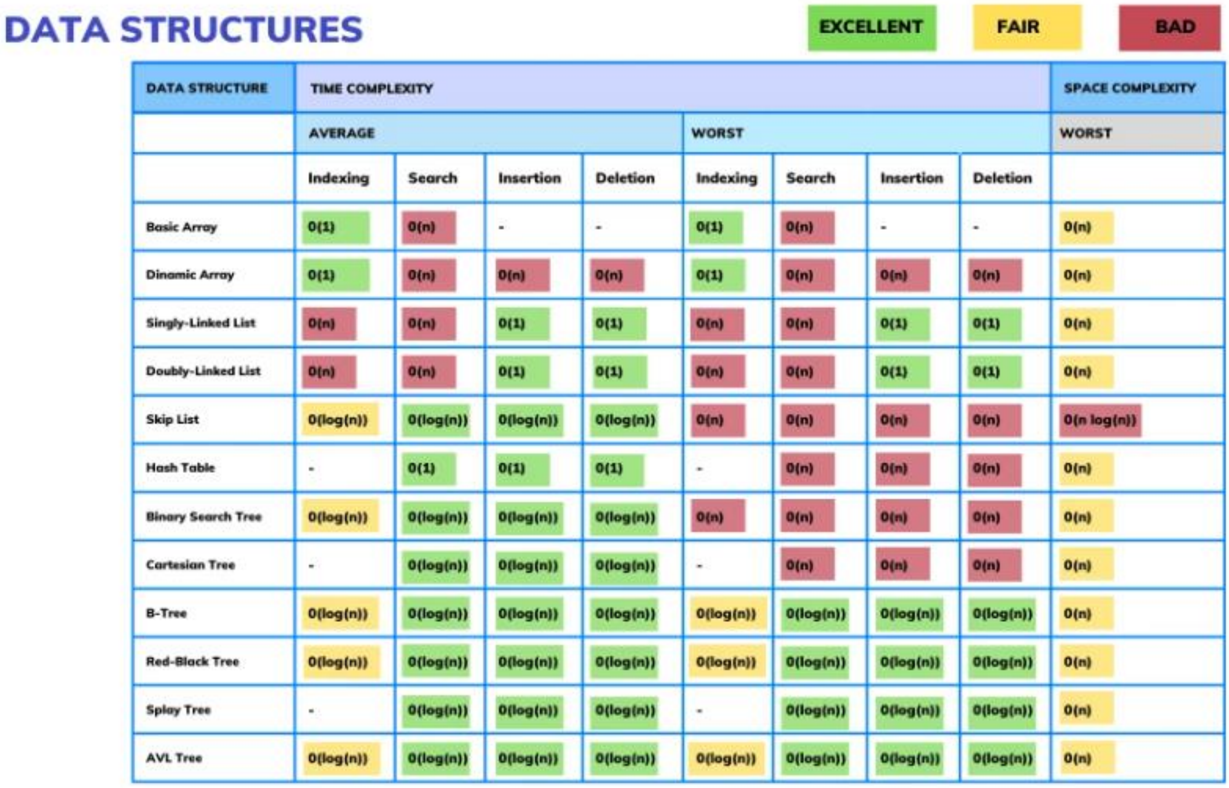
This introductory summary provides an overview of the systems features and capabilities, in the following section we will delve a bit deeper into each aspect of the system.

Development

Data Structure Selection and Concurrency

Selecting a data structure for my project was easy as we spent the whole of the first semester learning about data structures and, in particular HashMaps.

HashMaps run at a time complexity of *O(1)* or constant time making it a very efficient data structure for CRUD applications of search, insertion and deletion. HashMaps also excel at large datasets as they store data in key-value pairs and assuming the hash function uniformly distributes the student objects across the buckets it should stay in constant time no matter the size of the dataset.



DATA STRUCTURES

EXCELLENT FAIR BAD

DATA STRUCTURE	TIME COMPLEXITY								SPACE COMPLEXITY
	AVERAGE				WORST				WORST
	Indexing	Search	Insertion	Deletion	Indexing	Search	Insertion	Deletion	
Basic Array	0(1)	0(n)	-	-	0(1)	0(n)	-	-	0(n)
Dinamic Array	0(1)	0(n)	0(n)	0(n)	0(1)	0(n)	0(n)	0(n)	0(n)
Singly-Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Doubly-Linked List	0(n)	0(n)	0(1)	0(1)	0(n)	0(n)	0(1)	0(1)	0(n)
Skip List	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)	0(n)	0(n)	0(n)	0(n log(n))
Hash Table	-	0(1)	0(1)	0(1)	-	0(n)	0(n)	0(n)	0(n)
Binary Search Tree	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)	0(n)	0(n)	0(n)	0(n)
Cartesian Tree	-	0(log(n))	0(log(n))	0(log(n))	-	0(n)	0(n)	0(n)	0(n)
B-Tree	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)
Red-Black Tree	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)
Splay Tree	-	0(log(n))	0(log(n))	0(log(n))	-	0(log(n))	0(log(n))	0(log(n))	0(n)
AVL Tree	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(log(n))	0(n)

Figure 2 Data Structure Time Complexities

A ConcurrentHashMap in Java is essentially a thread-safe version of a normal HashMap, it’s part of the standard Java library that provides thread-safe operations without needing to synchronise the whole map at a slight cost to performance when compared to a normal HashMap.

A ConcurrentHashMap provides;

→ Thread Safety – Allows multiple users in a multi-threaded environment to safely share and manipulate the data across all threads whereas a HashMap will need to be manually synchronised.

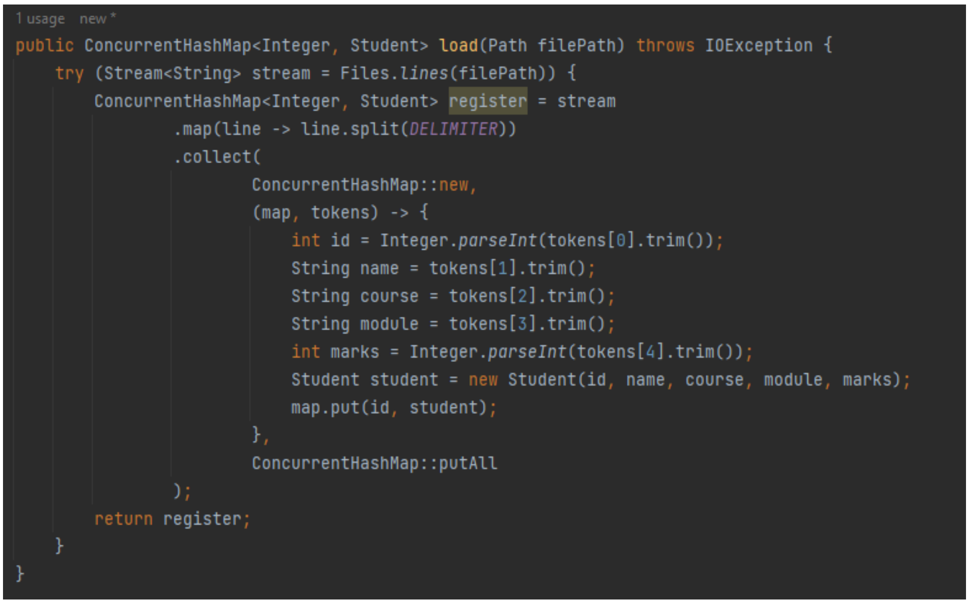
→ High Concurrency – Doesn't lock the whole map while performing operations, it instead divides the map into different segments and locking is done on those individual segments improving concurrency.

(Medewar , S. *Data Structures,* 2023 & *www.javatpoint.com, Difference between HashMap and ConcurrentHashMap* (no date))

File Handling

In a separate class called StudentRegisterFileHandler I have my file handling methods used to load from a hard drive and store it in a ConcurrentHashMap or save which will save the ConcurrentHashMap to a file on the hard drive.

A. Loading



```
1 usage  new *
public ConcurrentHashMap<Integer, Student> load(Path filePath) throws IOException {
    try (Stream<String> stream = Files.lines(filePath)) {
        ConcurrentHashMap<Integer, Student> register = stream
                .map(line -> line.split(DELIMITER))
                .collect(
                        ConcurrentHashMap::new,
                        (map, tokens) -> {
                            int id = Integer.parseInt(tokens[0].trim());
                            String name = tokens[1].trim();
                            String course = tokens[2].trim();
                            String module = tokens[3].trim();
                            int marks = Integer.parseInt(tokens[4].trim());
                            Student student = new Student(id, name, course, module, marks);
                            map.put(id, student);
                        },
                        ConcurrentHashMap::putAll
                );
        return register;
    }
}
```

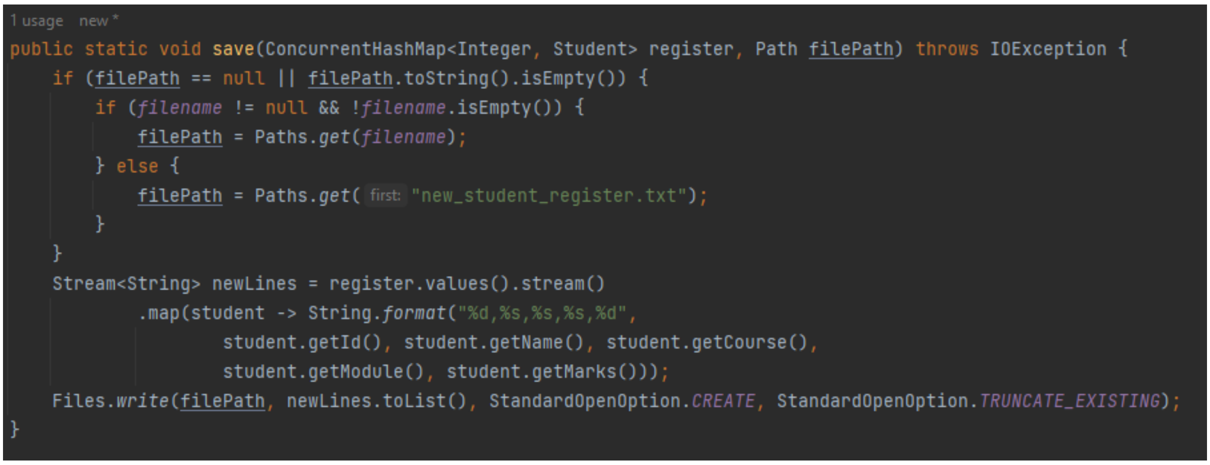
We start with a try/catch statement that ensures the stream will be closed automatically at the end of the block of code, **'Files.lines(filePath)'** reads all lines from the file and returns a **'Stream<String>'** where each 'String' is a line from the file. Now each line is a student object that has its information separated by commas, to transform each line into a student object we use the **'.map'** function that maps each piece of a students information that is separated by the commas to a student him or herself.

"ConcurrentHashMap::new" is an example of method referencing where we can collect the results into a new ConcurrentHashMap, after creating a new ConcurrentHashMap for the loaded data we then use an accumulator function for the **'.collect'** method which takes 2 arguments: the current map and the current String array or token. The function collects the tokens and creates a new student object and finally again using method referencing we can add everything that we have loaded – **'ConcurrentHashMap::putAll'**.

B. Saving

‘register.values().stream()’ collects the student register values/objects from the register and converts it into a stream for processing which is followed by **‘.map(student -> String.format(“%d, %s. %s. %s. %d” …’** which transforms each student object into a string using a lambda expression, the string itself is formatted to be separated by commas when saved.

The final line simple writes the data to file with **‘StandardOpenOption.CREATE’** option that will create the file if it doesn’t exist and **‘StandardOpenOption.TRUNCATE_EXISTING’** meaning it will delete the contents of the file before writing, i.e. overwriting the file.



```
1 usage   new *
public static void save(ConcurrentHashMap<Integer, Student> register, Path filePath) throws IOException {
    if (filePath == null || filePath.toString().isEmpty()) {
        if (filename != null && !filename.isEmpty()) {
            filePath = Paths.get(filename);
        } else {
            filePath = Paths.get( first: "new_student_register.txt");
        }
    }
    Stream<String> newLines = register.values().stream()
            .map(student -> String.format("%d,%s,%s,%s,%d",
                    student.getId(), student.getName(), student.getCourse(),
                    student.getModule(), student.getMarks()));
    Files.write(filePath, newLines.toList(), StandardOpenOption.CREATE, StandardOpenOption.TRUNCATE_EXISTING);
}
```

Both the load and the save methods utilise the functional programming paradigm in Java as both methods use multiple stream operations like **‘.map’** and **‘.collect’** and both incorporate lambda expressions when processing data with method referencing in the load method.

(Globant YouTube Channel. 2023)

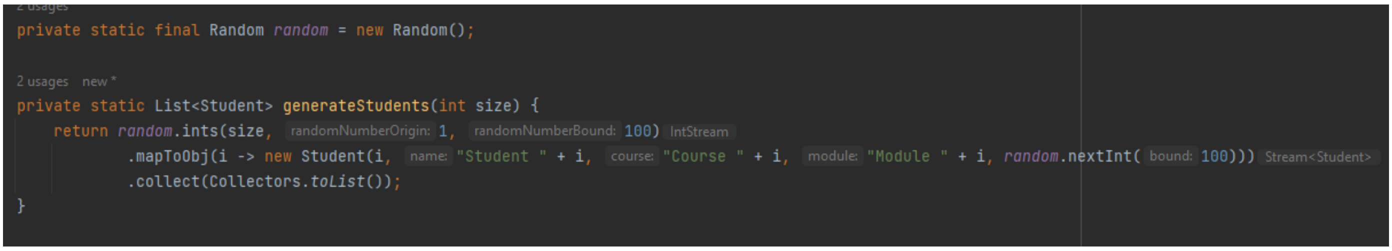
<u>BenchMarking Streams</u>

This class was designed to showcase the performance differences between Streams and ParallelStreams on different sizes of data.

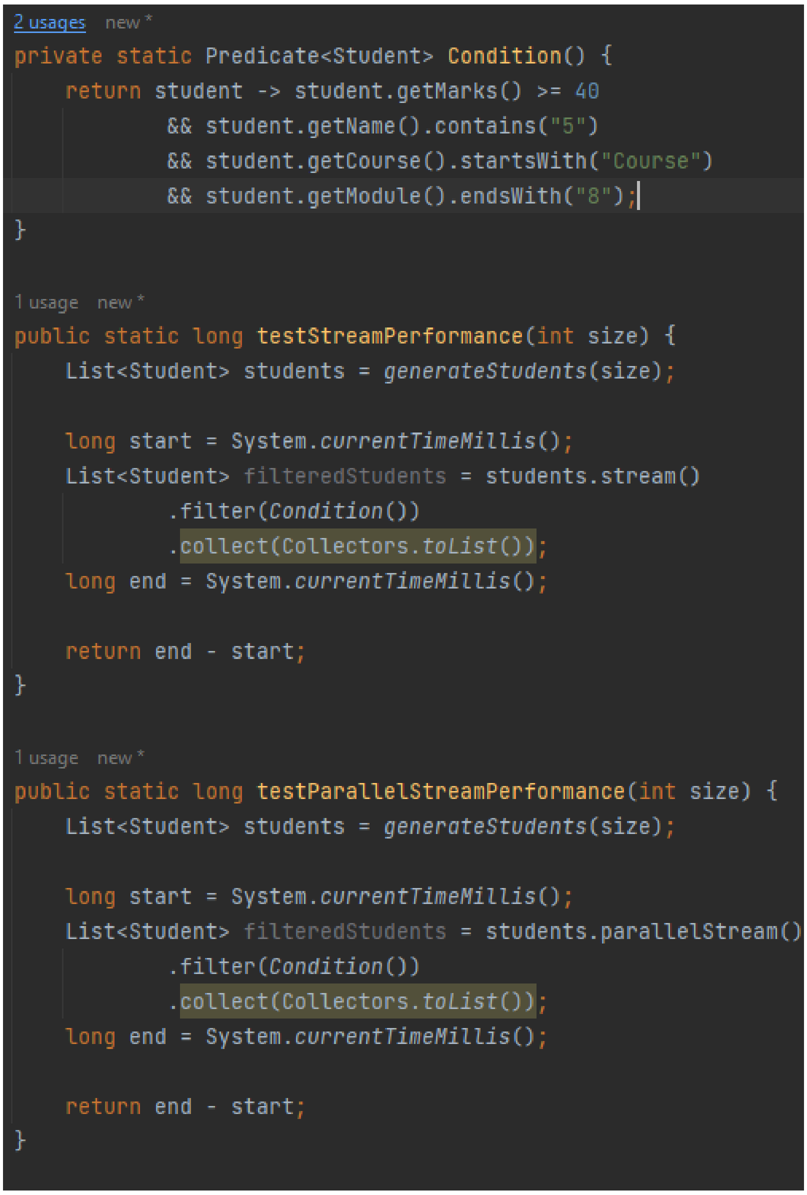
The method, **‘generateStudents’** takes an integer **‘size’** as a parameter and generates a list of student objects of the given size. **‘random.ints(size, 1, 100)’** generates a stream of random integers ranging from 1 and 100 and the amount of random integers will be determined by the **‘size’**.

‘.mapToObj’ uses these random integers to generate new student objects, where each integer **‘i’** will be used as part of the students ID, name, course and module with the marks also being applied randomly.

‘.collect’ then collects the generated students into a list.



```
2 usages
private static final Random random = new Random();

2 usages   new *
private static List<Student> generateStudents(int size) {
    return random.ints(size,  randomNumberOrigin: 1,  randomNumberBound: 100) IntStream
            .mapToObj(i -> new Student(i,  name: "Student " + i,  course: "Course " + i,  module: "Module " + i, random.nextInt( bound: 100))) Stream<Student>
            .collect(Collectors.toList());
}
```



```
2 usages  new *
private static Predicate<Student> Condition() {
    return student -> student.getMarks() >= 40
            && student.getName().contains("5")
            && student.getCourse().startsWith("Course")
            && student.getModule().endsWith("8");
}

1 usage  new *
public static long testStreamPerformance(int size) {
    List<Student> students = generateStudents(size);

    long start = System.currentTimeMillis();
    List<Student> filteredStudents = students.stream()
            .filter(Condition())
            .collect(Collectors.toList());
    long end = System.currentTimeMillis();

    return end - start;
}

1 usage  new *
public static long testParallelStreamPerformance(int size) {
    List<Student> students = generateStudents(size);

    long start = System.currentTimeMillis();
    List<Student> filteredStudents = students.parallelStream()
            .filter(Condition())
            .collect(Collectors.toList());
    long end = System.currentTimeMillis();

    return end - start;
}
```

The **'Condition'** method simply creates a Predicate for the generated students, I tried to make the condition as complex as possible with multiple checks to ensure a greater load on the streams.

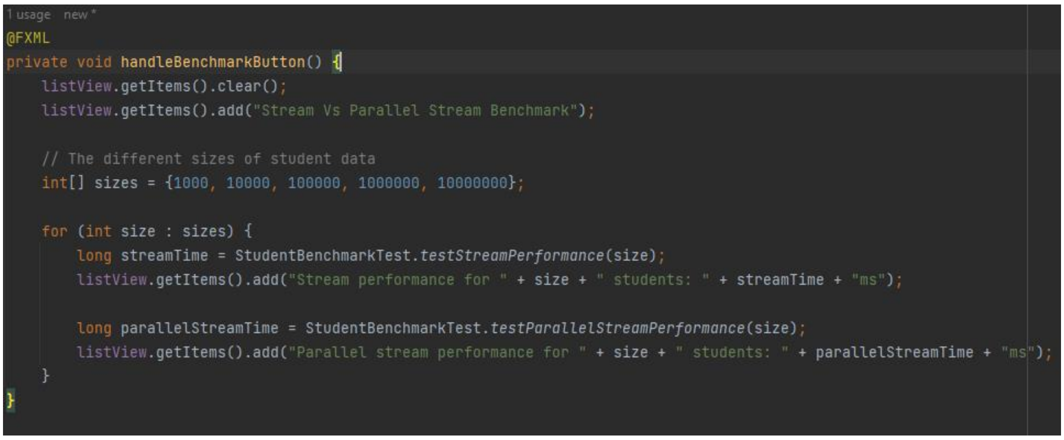
Now the two test methods record the length of time in m/s that each operation takes.

First line generates the students for testing, second line records the start time, third line proceeds to filter the student objects according to the predicate **'Condition'**, finally the method records the end time and works out the difference giving the overall run time of each operation.

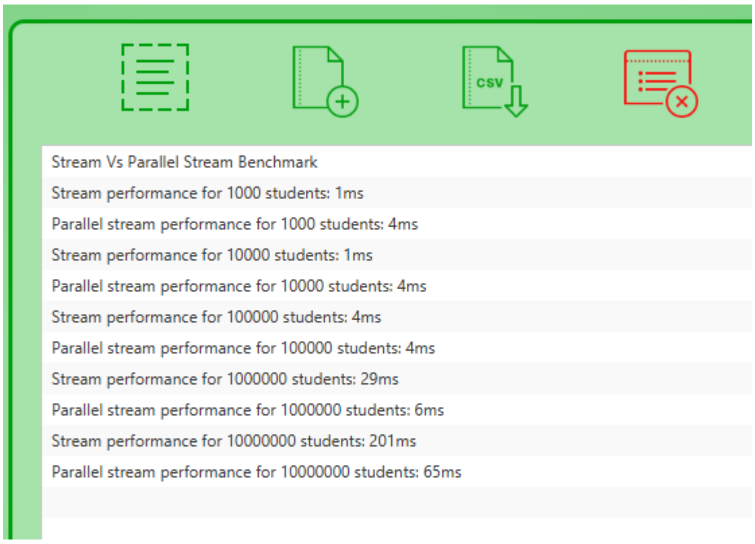
To connect it all together I linked a button on my GUI (icon that resembles a graph in the top right corner) to a method in my controller called **'handleBenchMarkButton'**.

This method starts by clearing the ListView and then adding a title to the operation.

Next, we create an array called **'sizes'** that will store the different sizes of student data, the for loop then iterates over the different input sizes and performs a benchmark test for each size and adds the results to the ListView.



```
1 usage  new *
@FXML
private void handleBenchmarkButton() {
    listView.getItems().clear();
    listView.getItems().add("Stream Vs Parallel Stream Benchmark");

    // The different sizes of student data
    int[] sizes = {1000, 10000, 100000, 1000000, 10000000};

    for (int size : sizes) {
        long streamTime = StudentBenchmarkTest.testStreamPerformance(size);
        listView.getItems().add("Stream performance for " + size + " students: " + streamTime + "ms");

        long parallelStreamTime = StudentBenchmarkTest.testParallelStreamPerformance(size);
        listView.getItems().add("Parallel stream performance for " + size + " students: " + parallelStreamTime + "ms");
    }
}
```



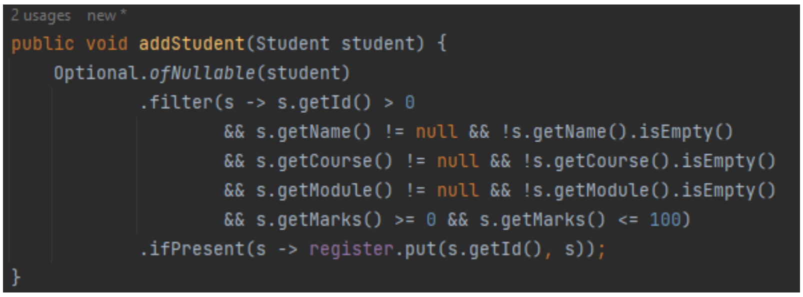
```
Stream Vs Parallel Stream Benchmark
Stream performance for 1000 students: 1ms
Parallel stream performance for 1000 students: 4ms
Stream performance for 10000 students: 1ms
Parallel stream performance for 10000 students: 4ms
Stream performance for 100000 students: 4ms
Parallel stream performance for 100000 students: 4ms
Stream performance for 1000000 students: 29ms
Parallel stream performance for 1000000 students: 6ms
Stream performance for 10000000 students: 201ms
Parallel stream performance for 10000000 students: 65ms
```

Interestingly a normal stream will outperform a parallel stream up to a certain point and then falls off greatly in comparison to the parallel stream, it seems parallel streams would be ideal for big data sets with multiple users working concurrently while a normal stream may be more suited to smaller operations whilst being cost/energy efficient.

Note: Ran into **'java.lang.OutOfMemoryError'** when testing on datasets above 20-30 million, exceeding the heap size limit of the JVM.

System Requirements

A. Add new or update existing student.

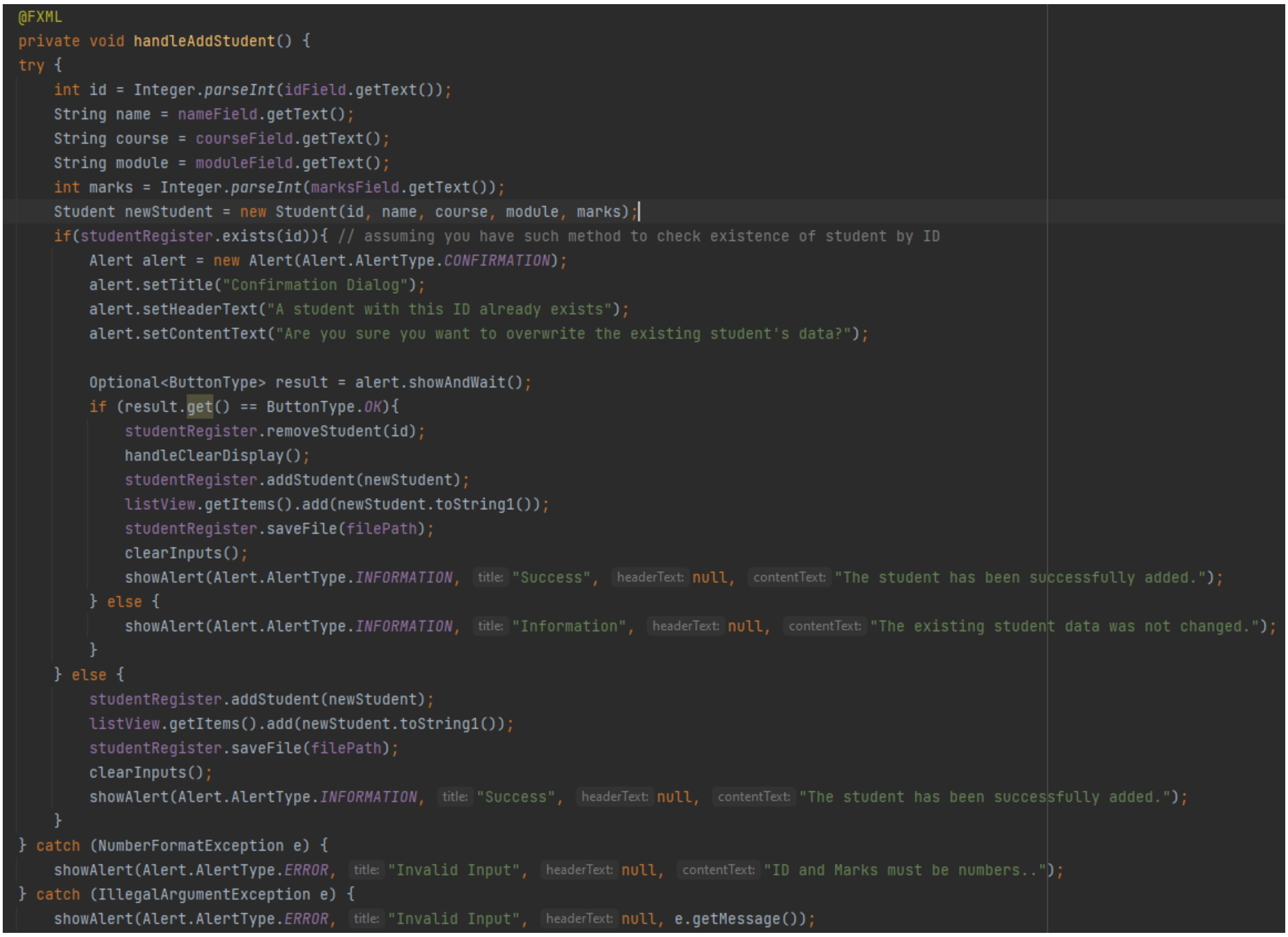


```java
public void addStudent(Student student) {
    Optional.ofNullable(student)
            .filter(s -> s.getId() > 0
                    && s.getName() != null && !s.getName().isEmpty()
                    && s.getCourse() != null && !s.getCourse().isEmpty()
                    && s.getModule() != null && !s.getModule().isEmpty()
                    && s.getMarks() >= 0 && s.getMarks() <= 100)
            .ifPresent(s -> register.put(s.getId(), s));
}
```

This is my **'addStudent'** method in my **StudentRegister** class that adds a new student object to the ConcurrentHashMap register.

It uses a container object **'Optional'** which is used to avoid **'NullPointerExceptions'** by wrapping the student objects – so it either contains a student or it is null.

The **'.filter'** method takes a Predicate (A functional interface that takes an input and returns a Boolean) and checks the students ID is positive, the name, course and module are not null or empty and finally checks whether the marks being inputted are between the ranges of 0 and 100 returning the optional with the student object that matches the predicate or simply returns an empty optional. Lastly the **'.ifPresent'** method is called and if the Optional is not empty the student is then added to the ConcurrentHashMap register.



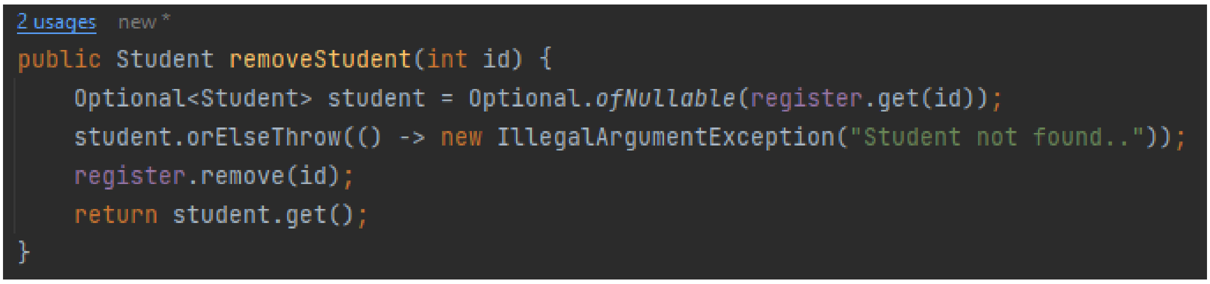
```
@FXML
private void handleAddStudent() {
try {
    int id = Integer.parseInt(idField.getText());
    String name = nameField.getText();
    String course = courseField.getText();
    String module = moduleField.getText();
    int marks = Integer.parseInt(marksField.getText());
    Student newStudent = new Student(id, name, course, module, marks);
    if(studentRegister.exists(id)){ // assuming you have such method to check existence of student by ID
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION);
        alert.setTitle("Confirmation Dialog");
        alert.setHeaderText("A student with this ID already exists");
        alert.setContentText("Are you sure you want to overwrite the existing student's data?");

        Optional<ButtonType> result = alert.showAndWait();
        if (result.get() == ButtonType.OK){
            studentRegister.removeStudent(id);
            handleClearDisplay();
            studentRegister.addStudent(newStudent);
            listView.getItems().add(newStudent.toString1());
            studentRegister.saveFile(filePath);
            clearInputs();
            showAlert(Alert.AlertType.INFORMATION, title: "Success", headerText: null, contentText: "The student has been successfully added.");
        } else {
            showAlert(Alert.AlertType.INFORMATION, title: "Information", headerText: null, contentText: "The existing student data was not changed.");
        }
    } else {
        studentRegister.addStudent(newStudent);
        listView.getItems().add(newStudent.toString1());
        studentRegister.saveFile(filePath);
        clearInputs();
        showAlert(Alert.AlertType.INFORMATION, title: "Success", headerText: null, contentText: "The student has been successfully added.");
    }
} catch (NumberFormatException e) {
    showAlert(Alert.AlertType.ERROR, title: "Invalid Input", headerText: null, contentText: "ID and Marks must be numbers..");
} catch (IllegalArgumentException e) {
    showAlert(Alert.AlertType.ERROR, title: "Invalid Input", headerText: null, e.getMessage());
```

The method in my controller class to handle the operation is as follows, it first retrieves input from the text fields in the GUI and attempts to create a new student object, it checks whether the ID already exists using a custom method in my StudentRegister (.exists(id)), if it doesn't exist it is simply added to the register ConcurrentHashMap, added to the ListView and also saved to file.

If the ID already exists a pop-up box appears prompting the user to either overwrite the existing student or to cancel and return.

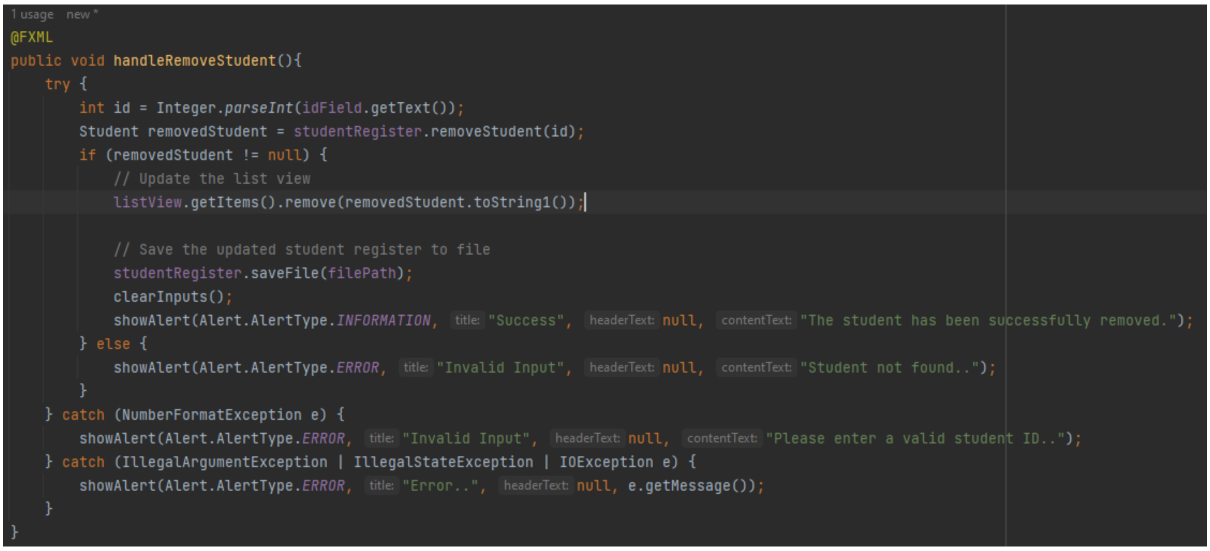
B. Remove an existing student.

This is my **‘removeStudent’** method in my StudentRegister class, again I used an Optional to wrap **‘register.get(id)’** and if the optional returns empty then **‘.orElseThrow’** throws an ‘IllegalArgumentException’ else the student object is removed using **‘register.remove(id)’**, we also return the student object that was removed.



```
2 usages   new *
public Student removeStudent(int id) {
    Optional<Student> student = Optional.ofNullable(register.get(id));
    student.orElseThrow(() -> new IllegalArgumentException("Student not found.."));
    register.remove(id);
    return student.get();
}
```

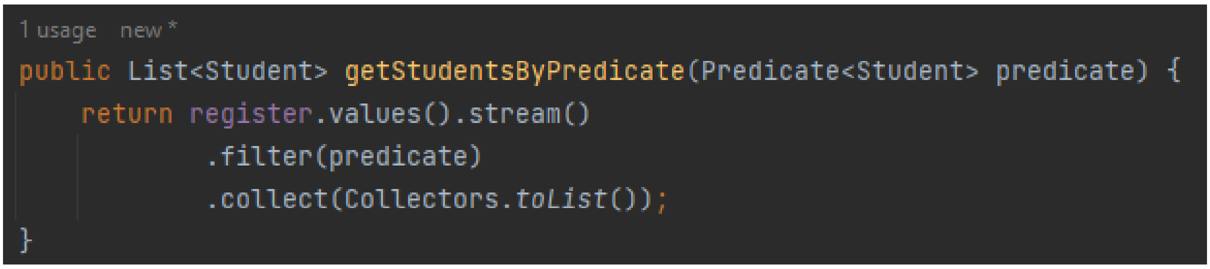
In my controller class we have the method **‘handleRemoveStudent’** which first attempts to parse the provided ID as an integer, if successful it attempts to remove the student with that ID from the register, if removed the ListView is updated with confirmation and saved to file – if no student was found an error message is displayed.



```
1 usage   new *
@FXML
public void handleRemoveStudent(){
    try {
        int id = Integer.parseInt(idField.getText());
        Student removedStudent = studentRegister.removeStudent(id);
        if (removedStudent != null) {
            // Update the list view
            listView.getItems().remove(removedStudent.toString1());

            // Save the updated student register to file
            studentRegister.saveFile(filePath);
            clearInputs();
            showAlert(Alert.AlertType.INFORMATION, title: "Success", headerText: null, contentText: "The student has been successfully removed.");
        } else {
            showAlert(Alert.AlertType.ERROR, title: "Invalid Input", headerText: null, contentText: "Student not found..");
        }
    } catch (NumberFormatException e) {
        showAlert(Alert.AlertType.ERROR, title: "Invalid Input", headerText: null, contentText: "Please enter a valid student ID..");
    } catch (IllegalArgumentException | IllegalStateException | IOException e) {
        showAlert(Alert.AlertType.ERROR, title: "Error..", headerText: null, e.getMessage());
    }
}
```

C. Query existing students by name, ID, course and module.

To begin with in my StudentRegister class I have this method that takes in a predicate as an argument returns a list, it does this by filtering the register according to the predicate then collecting the filtered results/stream to a new list.

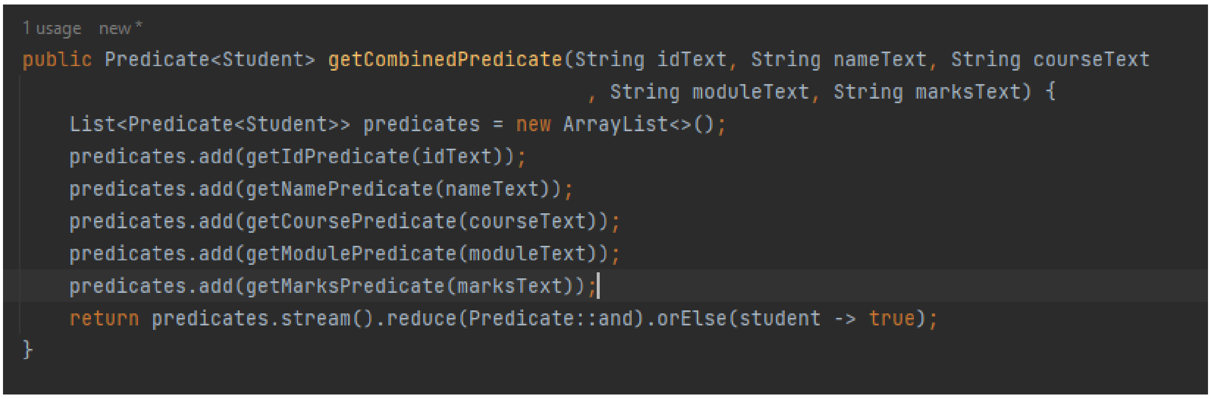


```java
public List<Student> getStudentsByPredicate(Predicate<Student> predicate) {
    return register.values().stream()
            .filter(predicate)
            .collect(Collectors.toList());
}
```

The **‘getCombinedPredicate’** method generates a combined predicate that is used to filter students based on multiple criteria. It creates individual predicates for each field for the student objects and then adds each individual predicate to an ArrayList which can be used to filter students based on more than one criteria if required.

It uses **‘.stream().reduce(Predicate::and).orElse(student -> true)’** to combine the predicates, the **‘.reduce’** operation applies the **‘Predicate::and’** function which means it generates a new predicate that checks all the individual predicates and only returns true if all return true.

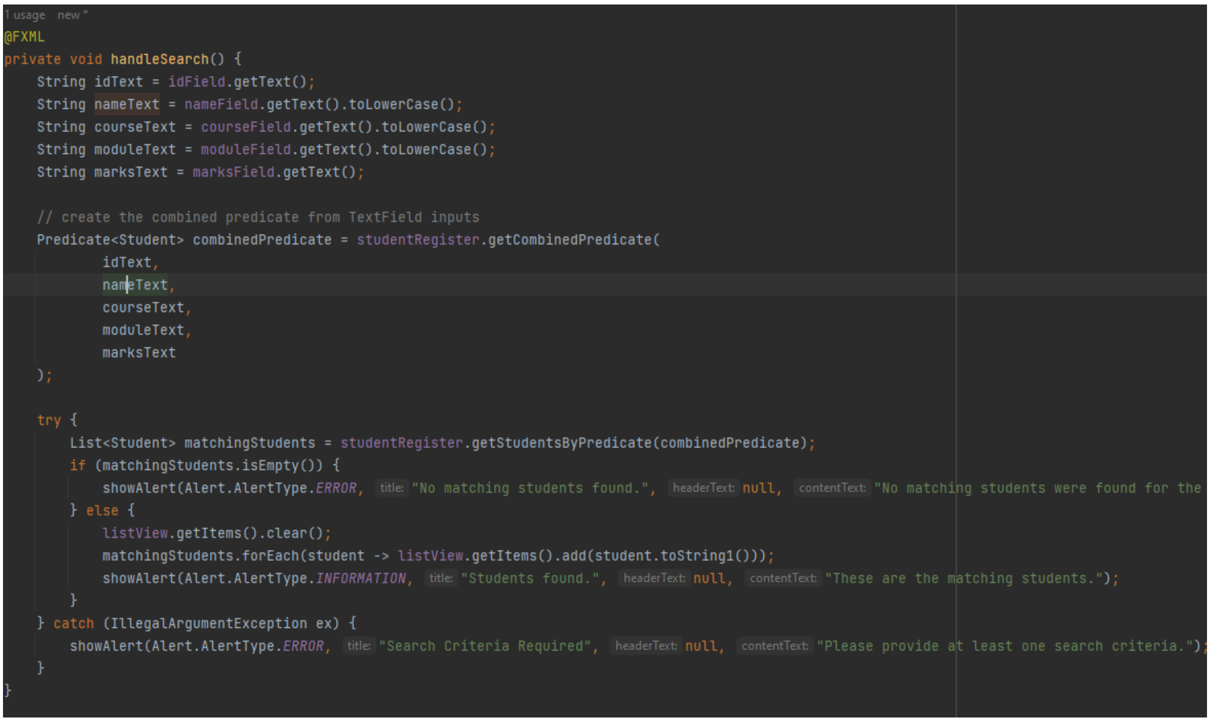
Note: You don’t have to search on all criteria, you can search just for 1 individual predicate just leave the other inputs empty.



```java
public Predicate<Student> getCombinedPredicate(String idText, String nameText, String courseText
                                              , String moduleText, String marksText) {
    List<Predicate<Student>> predicates = new ArrayList<>();
    predicates.add(getIdPredicate(idText));
    predicates.add(getNamePredicate(nameText));
    predicates.add(getCoursePredicate(courseText));
    predicates.add(getModulePredicate(moduleText));
    predicates.add(getMarksPredicate(marksText));
    return predicates.stream().reduce(Predicate::and).orElse(student -> true);
}
```

To bring these two methods together, we have another method called **‘handleSearch’** in the controller class, this method handles the search event by first registering the text from each input field, then uses these inputs to generate a combined predicate using the **‘getCombinedPredicate’** method.

Finally, we can use an example of a higher order function by passing the **‘combinedPredicate’** as an argument to the StudentRegisters **‘getStudentByPredicate’** method and if there are any matching students they are displayed to the ListView.



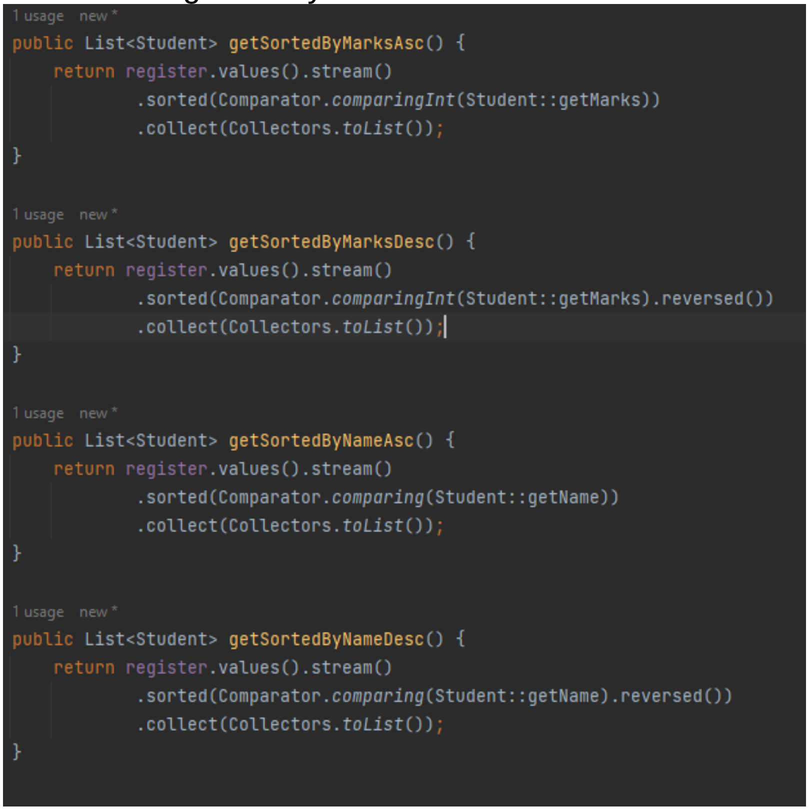
```java
@FXML
private void handleSearch() {
    String idText = idField.getText();
    String nameText = nameField.getText().toLowerCase();
    String courseText = courseField.getText().toLowerCase();
    String moduleText = moduleField.getText().toLowerCase();
    String marksText = marksField.getText();

    // create the combined predicate from TextField inputs
    Predicate<Student> combinedPredicate = studentRegister.getCombinedPredicate(
            idText,
            nameText,
            courseText,
            moduleText,
            marksText
    );

    try {
        List<Student> matchingStudents = studentRegister.getStudentsByPredicate(combinedPredicate);
        if (matchingStudents.isEmpty()) {
            showAlert(Alert.AlertType.ERROR, title: "No matching students found.", headerText: null, contentText: "No matching students were found for the
        } else {
            listView.getItems().clear();
            matchingStudents.forEach(student -> listView.getItems().add(student.toString1()));
            showAlert(Alert.AlertType.INFORMATION, title: "Students found.", headerText: null, contentText: "These are the matching students.");
        }
    } catch (IllegalArgumentException ex) {
        showAlert(Alert.AlertType.ERROR, title: "Search Criteria Required", headerText: null, contentText: "Please provide at least one search criteria.");
    }
}
```

This search is also able to get all students whose names begin with a certain letter, all students on a given module and all students on a given course whose names match a given text meeting some of the requirements of the application.

(Amigoscode YouTube Channel, *Java Functional Programming,* 2020*)*

Additional Queries

A. Sorting Data By Names and Marks



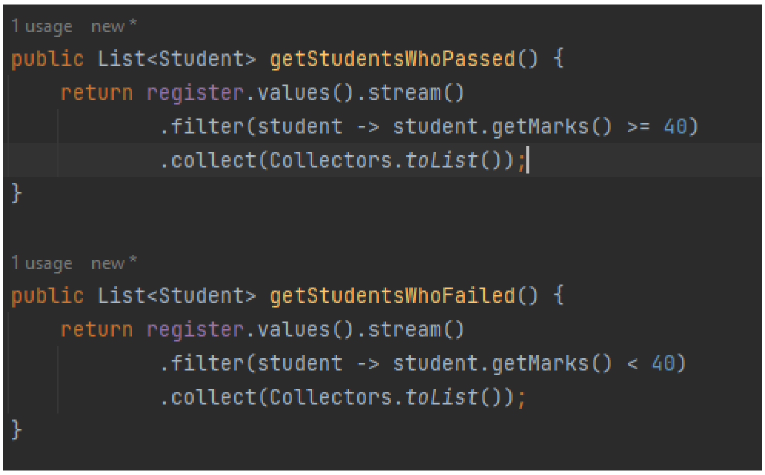
```
1 usage  new *
public List<Student> getSortedByMarksAsc() {
    return register.values().stream()
            .sorted(Comparator.comparingInt(Student::getMarks))
            .collect(Collectors.toList());
}

1 usage  new *
public List<Student> getSortedByMarksDesc() {
    return register.values().stream()
            .sorted(Comparator.comparingInt(Student::getMarks).reversed())
            .collect(Collectors.toList());
}

1 usage  new *
public List<Student> getSortedByNameAsc() {
    return register.values().stream()
            .sorted(Comparator.comparing(Student::getName))
            .collect(Collectors.toList());
}

1 usage  new *
public List<Student> getSortedByNameDesc() {
    return register.values().stream()
            .sorted(Comparator.comparing(Student::getName).reversed())
            .collect(Collectors.toList());
}
```

These 4 methods in my StudentRegister return lists of student objects sorted by different criteria, each using Javas Stream API.

They are very similar methods that create streams of values from the register ConcurrentHashMap, we then sort them using the **'.sort'** operator and **'Comparator.comparing'** methods which will by default sort in ascending order based on either their marks or their names.

Finally **'.collect(Collectors.toList())'** gathers the sorted stream elements into a new list.

B. Sorting Data By Students that Passed or Failed

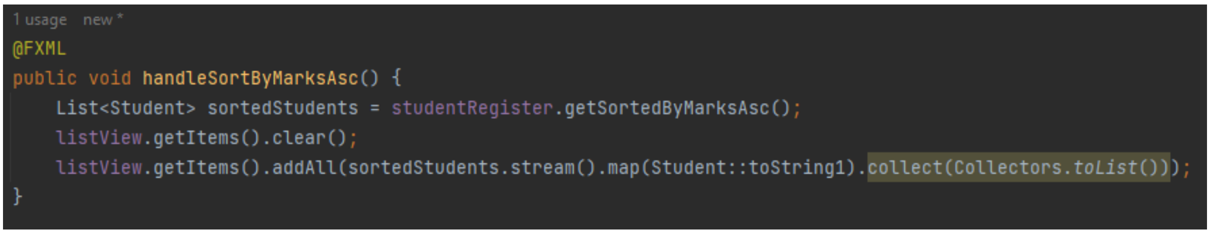


```
1 usage  new *
public List<Student> getStudentsWhoPassed() {
    return register.values().stream()
            .filter(student -> student.getMarks() >= 40)
            .collect(Collectors.toList());
}

1 usage  new *
public List<Student> getStudentsWhoFailed() {
    return register.values().stream()
            .filter(student -> student.getMarks() < 40)
            .collect(Collectors.toList());
}
```

These 2 methods are used to separate passing students from failing students.

In both methods a stream of students objects is created from the register map, then **'.filter'** proceeds to take a predicate as a higher order function argument that will filter the student data by their marks, anyone with marks of 40 or above have passed and anyone below 40 will have failed.

And to link these methods to an EventListener in my controller class I've used a simple method that is used by all 6 methods above:



```
1 usage  new *
@FXML
public void handleSortByMarksAsc() {
    List<Student> sortedStudents = studentRegister.getSortedByMarksAsc();
    listView.getItems().clear();
    listView.getItems().addAll(sortedStudents.stream().map(Student::toString1).collect(Collectors.toList()));
}
```

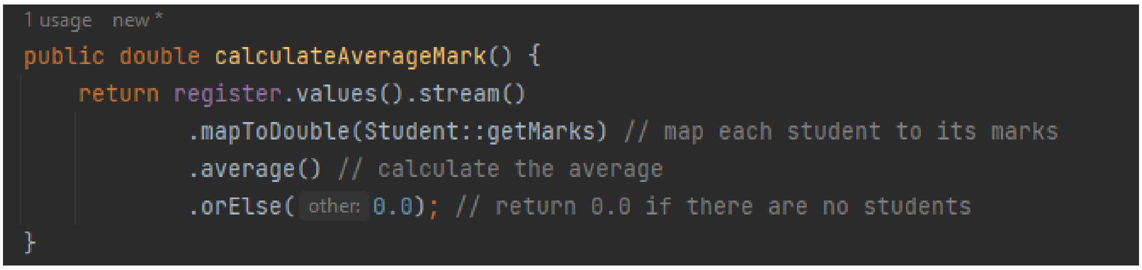
This method will take the ListView and clear it and then using the **'.addAll'** function we can add the entire list once converted to a stream, next the **'.map'** operation maps each student object to a string representation of it by calling the **'toString1'** method.

Finally, we collect the transformed student objects to a new list and then this newly formed list is added to the ListView.

C. Calculating an Average Mark for a given Dataset

Once again we convert the collection of values (student objects) from the register map into a Java Stream using **'register.values().stream()'**, next using the mapping operation **'.mapToDouble(Student::getMarks)'** and method referencing we can map each student object into a double value.

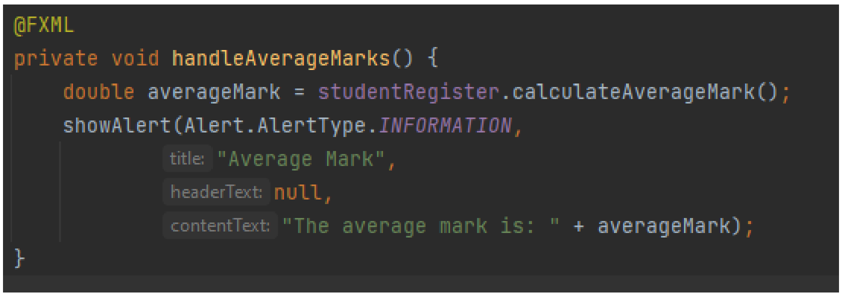
Next using the **'.average()'** terminal operation we can calculate the average of all the elements in the stream



```
1 usage  new *
public double calculateAverageMark() {
    return register.values().stream()
            .mapToDouble(Student::getMarks) // map each student to its marks
            .average() // calculate the average
            .orElse( other: 0.0); // return 0.0 if there are no students
}
```

'.orElse' will produce a result of '0.0' if the stream is null or empty.

Possibly the most declarative method in the system which does not explicitly describe how to perform the computation rather what the result should be.



```
@FXML
private void handleAverageMarks() {
    double averageMark = studentRegister.calculateAverageMark();
    showAlert(Alert.AlertType.INFORMATION,
            title: "Average Mark",
            headerText: null,
            contentText: "The average mark is: " + averageMark);
}
```

When clicking on the average marks button located to the right of the sorting buttons, it will pop-up an alert box with the requested information.

Critical Review

Three reasons why the design and implementation of the project are good:

1. Immutability – immutable student class, which is fundamental in functional programming, it reduces any side effects while also making code easier to follow and helps with thread safety in concurrent programs.

2. Use of Java Streams functions and operators – I tried to use purely java functional programming where appropriate, using a vast array of different functions and intermediary/ternary operators. The use of higher order functions throughout especially in my predicate logic which is also a fundamental aspect to functional programming.

3. The GUI and EventListeners – I was happy with the GUI overall as it provides a nice, clean user interface, the EventListeners themselves and methods in my controller class extensively used functional programming with the use of lambda expressions and other operators.

4. The BenchMark Class – Really enjoyed testing out different streams and working out how to time different methods, think it really adds to the application and could be used in scalability and sizing up.

Three reasons where the implementation could be improved and how:

1. File I/O – My file I/O methods themselves produce side effects, **'handleFileOpen'** method alters the state of the studentRegister object then updates the UI, where in functional programming it is ideal to avoid all side effects. I'm not sure how to produce a purely functional I/O system especially when incorporating a GUI – maybe separating out the logic a bit more but it seems extremely difficult to produce an application with a GUI using purely functional programming.

2. Error Handling – Although I used extensive input validation and error handling I feel there is room for improvement as I could have utilised **'Optionals'** a bit more wisely – this would improve maintainability of the program as well as readability.

3. I wanted to improve the amount of queries and features available, wanted to expand on the student object itself, include age, gender and attendance which would allow more useful data breakdowns and I also wanted to include a separate window that could be opened and the student data could be put into different bar chats and pie graphs to better represent the data visually.

Conclusion

I feel I have learnt a great deal during this project and vastly improved my programming skills and problem solving, the key concepts and main take-aways from the project:

1. Functional programming – The usefulness of the functional programming paradigm, the ease at which you can produce and manipulate data, the use of predicates, lambda expressions and the Stream API itself to manipulate and process data. Feels like it produces much simpler and easier to follow code.

2. Immutability – The importance of making objects immutable to reduce potential side effects, making code safer and reliable.

3. JavaFX – The project introduced me to more JavaFX features and tools including project setups with maven and the use of SceneBuilder to streamline the building of the GUI.

4. File I/O – Reinforced previous knowledge on file I/O although I didn’t produce a purely functional I/O it still helped improve my knowledge.

5. Java 8 Features – Introduced to many different Java 8 features including streams, lambda expressions, predicates, optionals and higher order functions helping me improve my code writing abilities.

Estimated grade

Functional Principles: A – A+ (80-90)
Report: A – A+ (80-90)

References

Amigoscode YouTube Channel (2020) *Java Functional Programming | Full Course*.. Available at: https://www.youtube.com/watch?v=VRpHdSFWGPs.

Difference between HashMap and ConcurrentHashMap - (no date) *www.javatpoint.com*. Available at: https://www.javatpoint.com/hashmap-vs-concurrenthashmap-in-java#:~:text=In%20HashMap%2C%20if%20one%20thread,the%20other%20thread%20is%20running.

Globant YouTube Channel. (2023), *Functional Programming for All | Java Code Camp🏕 | Live Coding Sessions YouTube*. Available at: https://www.youtube.com/watch?v=ett_gWVpUJ0.

Medewar , S. (2023) *Data Structures*, *hackr.io*. <image> Available at: https://cdn.hackr.io/uploads/posts/attachments/1650358110m7fPqMdxs5.webp. https://hackr.io/blog/big-o-notation-cheat-sheet