



Individual Assignment

Name:	Hoh Kok Young
TP Number:	TP046085
Module Name:	Advanced Programming Language Concepts
Module Code:	CT006-3-3-APLC
Intake Code:	UC3F2011SE
Lecturer:	Mr. Lee Kim Keong

Table of Contents

Introduction.....	1
Programming Concepts.....	2
Collections / Tuples	2
Pipelining.....	3
Purity & Side Effect.....	3
Higher-Order Function.....	4
Optional.....	4
Lambda	4
Recursion	5
Currying.....	5
Solutions	6
Task 1.....	6
Read Data from Excel File.....	6
Display Dengue Cases from 2014 – 2019.....	11
Total Dengue Cases Per Year	14
Total Dengue Cases Per Area	17
Area of Lowest Dengue Cases.....	21
Area of Highest Dengue Cases	25
Task 2.....	30
Prolog Rules for Sorting (Quicksort).....	30
Java-Prolog Connectivity.....	31
Sort in Descending Order.....	32
Conclusion	38
References.....	39

Introduction

In software development, there are different programming styles or better known as programming paradigms that defines the overall structure and architecture of the entire system. There are multiple paradigms introduced in the programming environment, imperative and declarative, each possess various benefits and cater to specific use cases. Procedural and object-oriented programming paradigms, classified under imperative programming emphasising on delineating the control flow of the program operation and it promotes state mutability. On the other hand, functional and logic programming paradigms, categorised under declarative programming focuses on expressing the computation logic and it advocates state immutability. Based on the current project objectives described, a system will be built through the incorporation of multiple programming paradigms, primarily the functional and logic paradigm using the Java programming language so as to extract, compute and display the dengue cases statistics that happened in Pahang from 2014 to 2019. The current system is developed in Apache NetBeans IDE 12.2 using jdk-11.0.10.

Programming Concepts

Functional programming is a declarative-based programming paradigm that visualises and approaches software programming to reflect the mathematical concepts and functions as much as possible, meanwhile preventing the state from being altered (Akhil Bhadwal, 2020), therefore ensures data consistency and integrity throughout the system execution. Logic programming is yet another declarative-based programming paradigm wherein the program statements delineate the facts and rules regarding a problem exists within a system (Sindhuja Hari, 2021). Queries are called based on the rules defined to resolve the problem. Sections shown and elaborated below are the programming concepts, mainly functional that have been integrated to the dengue case report system.

Collections / Tuples

Collection represents a group of items that possess the same data type. Tuple is similar to collection except for the fact that it is capable to store a cluster of objects with various data types (Baeldung, 2019). Examples of Java built-in collections framework include lists and sets. Maps, although not under the *Collection* interface in Java collections framework, it is still conceptually a collection as it stores key-value pair of items. Tuples such as pairs and triplets, however, requires an external library to be imported as Java has no built-in support for it. Although the immutable nature of tuples is extremely useful and handy in preventing side effects, the usage of collections was deemed much more reasonable especially in defining the class that is responsible in storing dengue cases-related data comprising the district names, the years as well as its relative number of cases. For example, the *TreeMap* permits storing data of key-value pairs in sorted order, which is perfect in storing the particular district's years of dengue cases as it preserves the relationship between the year and its relative amount of dengue cases for the specific district meanwhile being effortlessly manipulatable. Moreover, side effects can be easily prevented through the implementation of *Cloneable* interface. Apart from that, these collections contain similar methods to add, filter or simply loop through the data with highly optimised implementations and concise method names, making collections exceptionally performant as well as simpler to apply and maintain (HowToDoInJava.com, 2020).

Pipelining

Pipelines or specifically collection pipelines are a pattern for software programming whereby it segregates a computation process with a provided data source into a chain of operations, the output of each completed operation would be transferred as an input to the next operation until the entire computation process has finished. Although this particular concept is also used in object-oriented programming paradigms, pipelining is one of the core concepts in functional programming as methods such as *map()*, *filter()* and *forEach()* describes what to accomplish rather than how to accomplish (Fowler, 2015). To be more specific, these operations can either be intermediate operations or terminal operations. Methods such as *reduce()* or *filter()* are typically intermediate operations as they pass a Stream instance to be processed by the next operation while methods especially *forEach()* are terminal operations because they terminates the pipeline, signifying the end of computation process (Nataraja Gootooru, 2020), although any methods that is located at the end of the pipeline is also considered as terminal operations. As mentioned, these intermediate operations outputs a Stream instance, in actuality, the data source passed are not evaluated until the terminal operation is called. This phenomenon is known as lazy evaluation, which can increase the system efficiency enormously (LogicBig, 2017; Venkat Subramaniam, 2018).

Purity & Side Effect

Purity is the core concept in functional programming. Purity is often associated with pure functions, functions that bears a tight resemblance to mathematical operations, whereby they purely perform computations to produce an output according to the inputs passed, and it specifically does not induce any side effects. The output value retrieved from pure functions is completely dictated by the inputs passed to it. Pure functions are inherently referential transparent, meaning that as long as the input value being passed are the same such as $(2+2)$ or 4, the output value produced will stay the same as well, without affecting the program's behaviour in any way (Saumont, 2017). On the other hand, functions are considered impure if they causes side effects which includes mutating any state that is located outside the scope of the particular function, mutating its input arguments passed, throwing exceptions or executing any I/O processes such as display data into the graphical user interface or reading data from a file (Buonanno, 2017). Through the incorporation of purity concept in functional programming, these pure functions will be intrinsically more testable and maintainable because it is absolutely certain that the function contains zero side effects. Additionally, pure functions are generally

more modular and easier to combine, which allows lazy evaluation to be implemented so that the output will only be computed when it is required (Herring, 2017).

Higher-Order Function

A function that either accepts a function as an argument or returns a function are known as higher-order function. Higher order functions are capable of providing high level abstractions because all the complex internal implementations can be concealed (Normand, 2019). Examples of higher order functions are yet again, *map()*, *filter()*, *reduce()*, *forEach()* and more. These functions inherently promote functional programming concepts because as explained previously in pipelining section, they specify what to perform rather than how.

Optional

Optional is a built-in Java class that acts as a container object that store a possibly null object. Such concept was introduced because it reduces the amount of null checks being written, making the code much more readable and maintainable (Avram, 2020). The implementation of the Optional class could also prevent the program from crashing spontaneously meanwhile providing the developers insightful comments regarding the root problems rather than just printing out NullPointerException in the console log, which can be make the debugging process a hassle especially when the system is huge.

Lambda

Lambdas, or sometimes referred to as anonymous functions are a block of code that can acts as an argument to another function (Thelin, 2021; Fowler, 2004). These nameless functions are capable of representing the instance of a functional interface, as they exhibit only one functionality (GeeksforGeeks, 2019). Examples of lambdas can be frequently seen within methods such as *map()*, *forEach()* and *filter()*.

There are many advantages behind the implementation of lambdas. One of which is that lambdas reduce the lines of code to be written significantly because the need to compose an anonymous class is no more. The curly braces can be neglected for lambdas with single line of statement, therefore further promoting code conciseness and readability. Furthermore, the utilisation of lambdas mitigates the occurrences of shadow variables as well because they are nameless in nature. Shadow variables are variables in two different scope level having the same names (Lowe, 2021). For example, the global variable and the local variable shared the same name of *age*. Such practice can cause major confusion and leads to hard-to-maintain codebase,

hence should be avoided as minimum as possible with various techniques including lambdas. By using lambdas expressions concurrently with the Stream API in Java programming language, the processes can be executed by the collection themselves either sequentially or parallelly by passing the specific function to the collection methods, which in turn can improve the system overall performance tremendously when the processes are performed parallelly (Sahu, 2014).

Recursion

Recursion, or better regarded as recursive functions are functions that invoke itself until a specified condition is fulfilled (Tutorialspoint, 2021). Amongst the different types of recursion, the head and the tail recursions are two of the most renowned recursions discussed. Head recursions are recursions that invoke the recursive call at the initial statement in the function, leaving all the operations to be processed at the returning stage. Tail recursions, however, are recursions that process the operations required during each recursive calling, which leaves no operations to be executed at the end (Tusamma Sal Sabil, 2020). Recursions are generally favourable in functional programming as they are much concise and cleaner compared to iterations, making them easier to understand and apply.

Currying

Currying is a programming technique emphasised in functional programming whereby it converts a function that accepts multiple arguments into a chain of nested individual functions that takes in only one argument at a time. The arguments passed throughout the currying sequences will be executed when the last function in the currying sequences is called (Nnamdi, 2018). It adopts the similar concept of partial application, whereby partial application also breaks down the function of multiple arity (parameter) into individual functions, except these segregated functions may take more than one argument. The application of currying concept promotes the reusability of the function, hence result in much cleaner code without adding any unnecessary complexity, ultimately making the function more expressive and maintainable (Deepak Gupta, 2018).

Solutions

As the section above have detailed the programming concepts in general, this section focuses on explaining how these programming concepts are applied in producing the solutions in both task 1 and task 2. Although multiple programming paradigms will be applied in constructing the solutions, the functional programming paradigm is emphasised in calculating the statistics required by the sub-tasks defined in Task 1. Meanwhile on Task 2, the logic programming paradigm using Prolog (Programming in Logic) will be heavily incorporated instead, specifically to sort the dengue cases per area from highest to lowest. These solutions are then displayed on the graphical user interface (GUI) designed through the usage of Java Swing.

Task 1

Read Data from Excel File

Before the data can be calculated and displayed onto the GUI, the data must be extracted first and foremost. However, it is worth reminding that the data are separated into two excel files, one contains data from 2014 to 2016 meanwhile the other comprises data from 2017 to 2019. On top of that, the formatting of data differs quite notably between these two excel files. Therefore, the code implementation is composed with the intention to extract data from both excel files and consequently store them into the system, so that it can be processed, calculated and displayed neatly.

	A	B	C	D	E	F
1		KEJADIAN KES DEMAM DENGGI & DEMAM DENGGI BERDARAH TAHUN 2014 HINGGA 2017				
2						
3		Daerah	Tahun 2014	Tahun 2015	Tahun 2016	Tahun 2017
4			KES	KES	KES	KES
5		Kuantan	1,166	1,724	1,684	963
6		Rompin	36	91	46	30
7		Pekan	25	54	76	43
8		Temerloh	310	418	522	284
9		Maran	160	141	197	93
10		Jerantut	236	173	191	72
11		Bera	69	118	71	50
12		Raub	86	191	172	85
13		Bentong	51	51	45	35
14		Lipis	26	38	43	29
15		C.Highlands	5	2	3	6
16		PAHANG	2,170	3,001	3,050	1,690

Figure 1: Excel File – Dengue Cases in Pahang from 2014 to 2016

	A	B	C	D	E	F	G	H	I	J	K
1	KEJADIAN KES DEMAM DENGGI & DEMAM DENGGI BERDARAH TAHUN 2017 HINGGA 2019										
2											
3											
4		Daerah	Bilangan Kes								
5			2017	2018	2019						
6		Kuantan	963	575	1651						
7		Rompin	30	28	101						
8		Pekan	43	56	173						
9		Temerloh	284	107	270						
10		Maran	93	47	160						
11		Jerantut	72	47	125						
12		Bera	50	42	155						
13		Raub	85	28	76						
14		Bentong	35	32	124						
15		Lipis	30	24	36						
16		C.Highlands	6	3	4						
17		Pahang	1691	989	2875						

Figure 2: Excel File – Dengue Cases in Pahang from 2017 to 2019

Source Code

XlsxReader.java

```

15 | import org.apache.poi.ss.usermodel.Row;
16 | import org.apache.poi.xssf.usermodel.XSSFWorkbook;

```

Figure 3:Java Imported Libraries for Excel Operation

The third-party libraries shown above are essential and should be imported into the system so as to enable the system in interacting with excel internal workbooks and rows of data.

```

19 | public class XlsxReader {
20 |     private FileInputStream fileInputStream;
21 |     private XSSFWorkbook workbook;
22 |
23 |     //A constructor to assign and initiate the reading of the excel file paths given.
24 |     public XlsxReader(String filePath) {
25 |         try {
26 |             fileInputStream = new FileInputStream(filePath);
27 |             workbook = new XSSFWorkbook(fileInputStream);
28 |         } catch (Exception ex) {
29 |             ex.printStackTrace();
30 |         }
31 |     }

```

Figure 4: XlsxReader Class

For the purpose extracting the data from excel files, a class known as *XlsxReader* is created under the helper package. It includes all the relevant code implementation necessary including the libraries mentioned above. As shown in the codebase above, particularly from line 24, it is a constructor that accepts a string of file path as an argument. From line 25 to 30, a try catch block are incorporated to catch the errors when reading the excel file path. The try block

contains a *FileInputStream* object to retrieve the input bytes of the excel file passed into it as well as a *XSSFWorkbook* object that accepts the input bytes obtained as an argument to allow the system to read the data from stored in the excel spreadsheets.

```

33 //A function specialised in dissecting the excel file and return java data list.
34 public List<DengueCase> readFile(int fileNth) {
35     int startingIndex = fileNth == 1 ? 4 : 5;
36     List<Integer> years = fileNth == 1
37         ? Arrays.asList(2014, 2015, 2016)
38         : Arrays.asList(2017, 2018, 2019);
39     List<DengueCase> partialDengueCases = new ArrayList<>();
40     for (Row row : workbook.getSheetAt(0)) {
41         if (row.getRowNum() < startingIndex) continue;
42         if (row.getCell(1).getStringCellValue().toLowerCase().contains("pahang")) continue;
43         if (row.getCell(2) == null) continue;
44         TreeMap<Integer, Integer> tempDengueCasesPerYear = new TreeMap<>();
45         tempDengueCasesPerYear.put(years.get(0), (int) row.getCell(2).getNumericCellValue());
46         tempDengueCasesPerYear.put(years.get(1), (int) row.getCell(3).getNumericCellValue());
47         tempDengueCasesPerYear.put(years.get(2), (int) row.getCell(4).getNumericCellValue());
48         partialDengueCases.add(
49             new DengueCase(row.getCell(1).getStringCellValue().trim(), tempDengueCasesPerYear)
50         );
51     }
52     return partialDengueCases;
53 }

```

Figure 5: XlsxReader Class - readFile() method

Aside from the constructor, a method named *readFile()* comprising the parameter *fileNth* that indicates which file that is currently being read, assist in dissecting and pinpointing the rows to start reading the data from, in which the indicator is then stored into the variable *startingIndex*. Moreover, the *years* variable also stores different list of years based on the *fileNth* value accepts, so that the number of dengue cases retrieved from the particular excel file matches to its relative years accordingly. Afterwards, a for loop is constructed starting from the line 40 to commence reading the rows of data stored in the excel workbook specifically on the first sheet. At the beginning of the for loop, there are three if functions created, one to determine which row to start reading the data, one to ignore the rows of the overall dengue cases in Pahang, and the last one to ignore the second column of cells that contains null values. These if functions are created to ensure only relevant data are collected. Consequently, a *TreeMap* object is instantiated and stored into the variable called *tempDengueCasesPerYear* to store the year including its relative dengue cases for the specific district. *TreeMap* is part of the **collection** in programming concept, in which is particularly useful in this context because it shows the relationship between the year and the number of dengue case. It helps storing these data with ease as well meanwhile maintaining them in sorted order. Besides, the variable *partialDengueCases* with the data type of *List<DengueCase>* is utilised to store several *DengueCase* objects, each for each district. It is named *partialDengueCases* because each excel file contains only specific year range of dengue cases data. The internal implementation of the

DengueCase class will be explain in further detail in later stage. The method *readFile()* will then finally return the *partialDengueCases* variable, signifying that one file has been fully read.

```

55 //A function to combine the two partially read data from the two files into one single list.
56 public List<DengueCase> combinePartialDengueCasesData
57     (List<DengueCase> firstPartialDengueCases, List<DengueCase> secondPartialDengueCases) {
58     for(int counter = 0; counter < firstPartialDengueCases.size(); counter++){
59         firstPartialDengueCases.stream().collect(Collectors.toList())
60             .get(counter)
61             .getDengueCasePerYear()
62             .putAll(secondPartialDengueCases
63                 .get(counter)
64                 .getDengueCasePerYear());
65     }
66
67     return firstPartialDengueCases;
68 }

```

Figure 6: *XlsxReader* Class - *combinePartialDengueCasesData()* method

Finally, the *XlsxReader* also contain a method called *combinePartialDengueCasesData()* in which it literally accepts two partially read dengue cases data from each file as arguments and combine them into one single list, hence the return of the *firstPartialDengueCases* of data type *List<DengueCase>*. It achieves such operation by looping through the elements in both lists at the same time while appending the dengue cases from the second list to the respective district in the first list.

DengueCase.java

```

11 public class DengueCase implements Cloneable {
12     private String districtName;
13     private Map<Integer, Integer> dengueCasePerYear;
14
15     public DengueCase(String districtName, TreeMap<Integer, Integer> dengueCasePerYear) {
16         this.districtName = districtName;
17         this.dengueCasePerYear = dengueCasePerYear;
18     }
19
20     public void setDistrictName(String districtName) {
21         this.districtName = districtName;
22     }
23
24     public String getDistrictName() {
25         return districtName;
26     }
27
28     public void setDengueCasePerYear(Map<Integer, Integer> dengueCasePerYear) {
29         this.dengueCasePerYear = dengueCasePerYear;
30     }
31
32     public Map<Integer, Integer> getDengueCasePerYear() {
33         return dengueCasePerYear;
34     }

```

Figure 7: *DengueCase* Class

```

38      @Override
39      public Object clone() {
40          try {
41              return (DengueCase) super.clone();
42          } catch (CloneNotSupportedException ex) {
43              return new DengueCase(
44                  this.districtName, (TreeMap<Integer, Integer>) this.dengueCasePerYear
45              );
46          }
47      }
48  }

```

Figure 8: DengueClass - clone() method

The two figures above show the internal structure of data implemented in storing the dengue cases-related data including the district name, the year and the respective number of cases under the class name called *DengueCase*. Since the districts are names, the *String* are assigned as the data type. However, the data type of *Map<Integer, Integer>* is employed to store the individual years as keys and the respective number of cases as values. The **collection** is used because it represents the relationship between the year and the number of cases with clarity, and the nature of *Map* allowing these key-value pairs to be stored together permits these data to be accessed with ease and less confusing. Due to the fact that these two attributes are private, several getter and setter methods, *getDistrictName()*, *setDistrictName()*, *getDengueCasePerYear()* and *setDengueCasePerYear()* are constructed to set and retrieve dengue case data, which also demonstrate the concept of encapsulation.

Apart from that, the *DengueCase* class has implemented the *Cloneable* interface and overridden the *clone()* method. This is important because the implementation assist in achieving deep cloning, which can be applied to avoid causing side effects in various functions. There are two types of cloning, shallow cloning and deep cloning. Shallow cloning refers to copying the source object, except for the objects the source object has reference to. This implies that the modification of these referred objects in the shallow copy will be reflected in the source object as well, therefore causing side effects. However, deep cloning ensures the changes on the referred objects will not be reflected on the source object (Edureka, 2019), which makes it an ideal method in preventing side effects.

```

26 public class DengueCaseReportSystem extends javax.swing.JFrame {
27     private final String EXCEL_FILE_PATH_2014_To_2016 = "./src/storage/statistik-"
28         + "kes-denggi-di-negeri-pahang-bagi-tempoh-2014-2017.xlsx";
29     private final String EXCEL_FILE_PATH_2017_To_2019 = "./src/storage/statistik-"
30         + "kes-denggi-di-negeri-pahang-bagi-tempoh-2018-2019.xlsx";
31     private final String PROLOG_FILE_PATH = "./src/storage/quicksort.pl";
32
33     private Calculation calculation;
34     private LinkedHashMap<String, Integer> totalDengueCasesPerArea;
35
36     public DengueCaseReportSystem() {
37         initComponents();
38         setLocationRelativeTo(null);
39
40         try {
41             XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42             XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43             List<DengueCase> dengueCasesData =
44                 xlsxReader1.combinePartialDengueCasesData(
45                     xlsxReader1.readFile(1),
46                     xlsxReader2.readFile(2)
47                 );

```

Figure 9: *DengueCaseReportSystem* (Main) Class

The figure above illustrates the main class, named as *DengueCaseReportSystem* where the *main()* method exists. The extraction and the insertion of the excel data into the system memory are performed at the initial stage of the program execution, particularly at the class constructor. Before the extraction of data is executed, the two final variables, *EXCEL_FILE_PATH_2014_To_2016* and *EXCEL_FILE_PATH_2017_To_2019* that stores the file path of each excel file is initialised. Starting from line 41 within the constructor, two *XlsxReader* object are instantiated with each file path passed as argument so as to initialise the data extraction and insertion process. Right afterwards, the method *readFile()* invoked from each *XlsxReader* instance is passed to the *combinePartialDengueCasesData()* so that excel data can be retrieved and combined, which are then finally inserted into the variable called *dengueCasesData* so that it can store the list of dengue cases to be calculated afterwards.

Display Dengue Cases from 2014 – 2019

This is another sub-task that requires all the dengue cases retrieved from the excel file to be displayed on the GUI.

DengueCaseReportSystem.java

```

65 public void displayOverallDengueCases (List<DengueCase> dengueCasesData) {
66     DefaultTableModel overallDengueCasesTableModel =
67         (DefaultTableModel) TblOverallDengueCases.getModel();
68     dengueCasesData.forEach(dengueCase -> overallDengueCasesTableModel
69         .addRow(
70             Stream.concat(
71                 Arrays.stream(new Object[]{dengueCase.getDistrictName()}),
72                 Arrays.stream(dengueCase.getDengueCasePerYear().values().toArray()))
73                 .toArray(Object[]::new)
74             );
75     );
76 }

```

Figure 10: DengueCaseReportSystem (Main) Class - displayOverallDengueCases() method

The display of the dengue cases data from 2014 to 2019 revolves around this method called *displayOverallDengueCases()*. This method receives a parameter *dengueCasesData* of data type *List<DengueCase>* that contains the entire list of dengue cases from individual district of different years. In this sub-task, JTable component with the variable name, *TblOverallDengueCases* supported in Java Swing is utilised to display the dengue cases in two-dimensional view. Afterwards, the method *getModel()* was called and the value was assigned to another variable called *overallDengueCasesTableModel* for the purpose of populating the table with dengue cases data. To begin the process of the data population, each instance of *DengueCase* is iterated through the use of *forEach()* method. Consequently, the table model retrieved will then start adding rows of data based on the instance of *DengueCase* retrieved. Due to the fact that the method *addRow()* only supports argument of data type *Object[]*, the code from line 70 till line 73 is responsible in converting the individual district name and its respective dengue cases per year into *Object[]* data type. The concept of **higher-order function and lambda** is manifested here as a lambda function is passed to the method *forEach()*. The programming concept of **pipelining** is also demonstrated here wherein pipeline occurs when the *Stream.concat().toArray()* is invoked to concatenate two streams together and convert the stream back to array object.

```

36 public DengueCaseReportSystem() {
37     initComponents();
38     setLocationRelativeTo(null);
39
40     try {
41         XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42         XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43         List<DengueCase> dengueCasesData =
44             xlsxReader1.combinePartialDengueCasesData(
45                 xlsxReader1.readFile(1),
46                 xlsxReader2.readFile(2)
47             );
48
49         calculation = new Calculation();
50
51         List<DengueCase> newDengueCasesData =
52             calculation.getTotalCasesPerYear(dengueCasesData);
53
54         displayOverallDengueCases(newDengueCasesData);

```

Figure 11: DengueCaseReportSystem (Main) Class - Constructor

To display the dengue cases in the tabular format as defined in the function mentioned, *displayOverallDengueCases()*, the function is invoked at the main method specifically at line 54.

Screenshot

Dengue Case 2014 - 2019 Report							
<div>Home</div> <div>Statistics</div> <div>Exit</div>	District	2014	2015	2016	2017	2018	2019
	Kuantan	1166	1724	1684	963	575	1651
	Rompin	36	91	46	30	28	101
	Pekan	25	54	76	43	56	173
	Temerloh	310	418	522	284	107	270
	Maran	160	141	197	93	47	160
	Jerantut	236	173	191	72	47	125
	Bera	69	118	71	50	42	155
	Raub	86	191	172	85	28	76
	Bentong	51	51	45	35	32	124
	Lipis	26	38	43	30	24	36
	C.Highlands	5	2	3	6	3	4
	Pahang	2170	3001	3050	1691	989	2875

Figure 12: Graphical User Interface - Overall Dengue Cases in Pahang from 2014 to 2019

The diagram above displays the tabular form of the dengue cases happened throughout the years for each Pahang district in the user interface.

Total Dengue Cases Per Year

Based on the tabular data shown above, it can be seen that the last row of the table contains the total dengue cases that occurred for each year, which should be calculated within this sub-task. The code implementation for the calculation regarding this criterion is written with full intention of purity and no side-effects.

Source Code

Calculation.java

```
20 public class Calculation {
21
22     //A currying function to summate the dengue cases for each year of all the districts.
23     Function<Integer, Function<List<DengueCase>, Integer>> dengueCasesSummation =
24         year ->
25             dengueCasesData ->
26                 dengueCasesData
27                     .stream()
28                     .mapToInt(dengueCase ->
29                         dengueCase
30                             .getDengueCasePerYear()
31                             .get(year))
32                     .sum();
33
34     //A function to calculate the total dengue cases of all the districts per year.
35     public List<DengueCase> getTotalCasesPerYear(List<DengueCase> dengueCasesData) {
36         //Avoid side effect.
37         List<DengueCase> totalDengueCasesByYear = CloneList.cloneList.apply(dengueCasesData);
38         Map<Integer, Integer> dengueCasesByYear = new TreeMap<>();
39         List<Integer> years = Arrays.asList(2014, 2015, 2016, 2017, 2018, 2019);
40         years.stream().forEach(year -> dengueCasesByYear
41             .put(year, dengueCasesSummation.apply(year).apply(dengueCasesData)));
42         totalDengueCasesByYear.add(new DengueCase("Pahang", (TreeMap) dengueCasesByYear));
43         return totalDengueCasesByYear;
44     }
```

Figure 13: Calculation Class

Before discussing about the calculation procedures, it was worth mentioning that the calculation-related functions are all stored within the class called *Calculation*, primarily for the purpose to promote greater modularity and high cohesiveness, which translates to better code readability and maintainability. Based on the figure displayed above, there are two functions defined and implemented, both functions are related in solving the calculation. Since the method *getTotalCasesPerYear()* should and will be called in the *main()* method, the method will be explained priorly. First and foremost, the method accepts a single argument of data type *List<DengueCase>* under the name *dengueCasesData*, so that the list of *DengueCase*

instances can be iterated and perform the calculation necessary. However, due to the fact that the intended means to display the total dengue cases after the calculation has been performed is by appending the new *DengueCase* instance into the current existing list, the *dengueCasesData* that holds the original list of *DengueCase* instances are deep cloned using the static method *cloneList()* defined in the *CloneList* class. The deep clone is absolutely necessary to ensure the function itself is **pure** and possess **zero side effects**. Without deep cloning, the addition of the new *DengueCase* instance would alter the original data, *dengueCasesData* as well, which would then violate to the philosophy of functional programming.

The actual calculation of the total dengue cases per year is located at line 40 and 41, wherein it involves the iteration of the list of years initialised at line 39 and the currying function *dengueCasesSummation()*. Each year iteration would calculate the overall dengue cases occurred for that particular year, which would then be stored in variable *dengueCasesByYear* with instantiation of *TreeMap* data type, a **collection** that maintains a sorted group of items based on the keys. The hard-coded district name “Pahang” (although it is technically a state) including the maps of data in the variable *dengueCasesByYear* will be instantiated as a *DengueCase* instance and subsequently be added to the cloned list, *totalDengueCasesByYear*. The function in the end will return the cloned list so that it can be display onto the GUI. Before explaining the currying function, there are several concepts adapted in this very function, which includes **pipelining** for *dengueCasesData.stream().forEach()*, **lambda** expression in *forEach()* method, making it a **higher-order function** and lastly, **purity** as the referred objects, *DengueCase* are not altered thanks to deep cloning.

From line 23 till 32, the **currying** concept is demonstrated as the *dengueCasesSummation()* function has chain of functions that accepts only one argument each. This function is liable to calculate all the dengue cases happenings on the particular year that was passed as the first argument. Curried functions are inherently **higher-order functions** as it returns a function and they contain **lambdas** as well, *dengueCasesData -> ...* as shown in the code implementation.

CloneList.java

```
13 public class CloneList {
14
15     //A function specialised in cloning a list of DengueCase object,
16     //so as to prevent side effect.
17     static Function <List<DengueCase>, List<DengueCase>> cloneList = list -> {
18         List<DengueCase> clonedList = new ArrayList<>();
19         list.stream().forEach(element -> clonedList.add((DengueCase) element.clone()));
20         return clonedList;
21     };
22 }
```

Figure 14: CloneList Class

As shown above, *CloneList* class contains a static method purely to deep clone the list of *DengueCase* instances it accepts as an argument. This method iterates the list of *DengueCase* instances as shown in line 19. For each instance iterated, the instance will be cloned and added to the new *ArrayList* instantiation under the variable named *clonedList*. After all the instances have been deep cloned, the method will then return the cloned list.

DengueCaseReportSystem.java

```
33 private Calculation calculation;
34 private LinkedHashMap<String, Integer> totalDengueCasesPerArea;
35
36 public DengueCaseReportSystem() {
37     initComponents();
38     setLocationRelativeTo(null);
39
40     try {
41         XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42         XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43         List<DengueCase> dengueCasesData =
44             xlsxReader1.combinePartialDengueCasesData(
45                 xlsxReader1.readFile(1),
46                 xlsxReader2.readFile(2)
47             );
48
49         calculation = new Calculation();
50
51         List<DengueCase> newDengueCasesData =
52             calculation.getTotalCasesPerYear(dengueCasesData);
53
54         displayOverallDengueCases(newDengueCasesData);
55     } catch (Exception e) {
56         // Handle exception
57     }
58 }
```

Figure 15: DengueCaseReportSystem (Main) Class - Constructor

Since the planned architecture of the system is to display both the individual dengue cases happened in each district throughout years as well as the total dengue cases in Pahang per year, the method invocation was not properly explained in the first sub-task. In actuality, the *Calculation* class must first be declared and initialised as shown in the line 33 and line 49.

After the instantiation, the method *getTotalCasesPerYear()* can then be invoked with the argument of original list of dengue cases *dengueCasesData* being passed into it, so that the calculation of the total dengue cases per year can be performed. The results which is a new list of *DengueCase* instances are then be stored into the new variable called *newDengueCasesData()*. The new variable is then be passed into the method *displayOverallDengueCases()* for the purpose of displaying the overall dengue cases in tabular form, as explained in previous section.

Screenshot



District	2014	2015	2016	2017	2018	2019
Kuantan	1166	1724	1684	963	575	1651
Rompin	36	91	46	30	28	101
Pekan	25	54	76	43	56	173
Temerloh	310	418	522	284	107	270
Maran	160	141	197	93	47	160
Jerantut	236	173	191	72	47	125
Bera	69	118	71	50	42	155
Raub	86	191	172	85	28	76
Bentong	51	51	45	35	32	124
Lipis	26	38	43	30	24	36
C.Highlands	5	2	3	6	3	4
Pahang	2170	3001	3050	1691	989	2875

Figure 16: Graphical User Interface - Overall Dengue Cases in Pahang from 2014 to 2019

As shown above, the overall dengue cases occurred throughout the years for each district including the total dengue cases happened in Pahang each year is displayed in tabular form, same as section 1.

Total Dengue Cases Per Area

This sub-task involves the calculation of the total dengue cases for each district throughout the years.

Calculation.java

```

46 //A recursive function to calculate the dengue case per area.
47 BiFunction<Map<Integer, Integer>, Integer, Integer> calculateCasePerAreaRecursively
48     = (dengueCasePerYear, year) ->
49         year <= (int) dengueCasePerYear
50             .keySet()
51             .toArray()
52             [dengueCasePerYear.size() - 1]
53             ? this.calculateCasePerAreaRecursively
54                 .apply(dengueCasePerYear, year + 1)
55                 + dengueCasePerYear.get(year)
56             : 0;
57
58 //A function to calculate the total dengue cases per district throughout the years.
59 public LinkedHashMap<String, Integer> getTotalCasesPerArea(List<DengueCase> dengueCasesData) {
60     LinkedHashMap<String, Integer> totalDengueCasesPerArea = new LinkedHashMap<>();
61     dengueCasesData
62         .stream()
63         .forEach((dengueCaseData) -> totalDengueCasesPerArea
64             .put(dengueCaseData.getDistrictName(), calculateCasePerAreaRecursively
65                 .apply(dengueCaseData.getDengueCasePerYear(),
66                     (int) dengueCaseData.getDengueCasePerYear().keySet().toArray()[0])));
67     return totalDengueCasesPerArea;

```

Figure 17: Calculation Class - `getTotalCasesPerArea()` method & `calculateCasePerAreaRecursively` function

As shown in the code implementation above, there are two functions written to perform the calculation of the total dengue cases occurred throughout the years for each individual district. Since the method `getTotalCasesPerArea()` will be invoked by the main method, the method will first be thoroughly scrutinised and discussed. Firstly, it again accepts the list of *DengueCase* instances as an argument under the name *dengueCasesData*, therefore permitting the data extracted from the excel files to be read, iterated and perform the calculation required. However, rather than returning another list of *DengueCase* instances, the return data type here is another **collection** known as *LinkedHashMap*<String, Integer>, a map that maintains the insertion order without performing sorting operations internally like *TreeMap* or randomise the order or elements like *HashMap*. *LinkedHashMap* is employed in this specific scenario mainly because the year values are not relevant to be displayed and it provides a clearer relationship between the district and the total dengue cases compared to using the instances of *DengueCase*. Moreover, the reason as to why *LinkedHashMap* is chosen compared to other maps is because it maintains the order of key-value pairs as previous tabular data, sorting or randomising the elements could cause confusion to the system user.

From line 61 till 66, the list of *DengueCase* instances are iterated through the **pipelining** sequences of methods *dengueCasesData.stream().forEach()* in order to insert the total dengue cases calculated for each district into the *LinkedHashMap*<String, Integer> variable

totalDengueCasesPerArea initialised in line 60. Once the calculation is completed, the *getTotalCasesPerArea()* method will return the variable, which will be used to display the data onto the GUI. Another example of **pipelining** that exists within in function is the chain of methods *getDengueCasePerYear().keySet().toArray()*. Furthermore, a **lambda** expression was applied in the *forEach((dengueCaseData) -> ...)* method. Not to mention, this function is **pure** and causes **no side effects** because the function does not modify any global variables nor the input parameters. The input *dengueCasesData* is only accessed to iterate the data, no modifications towards it were performed. The function does not throw exception and execute any I/O operations as well.

Onto the **recursive** function, *calculateCasePerAreaRecursively* starting from the line 47 till line 56, this function is defined, implemented and invoked in the *getTotalCasesPerArea()* method to calculate the dengue cases for the specific district recursively. Based on the differentiation between the types of recursion discussed earlier, this recursive function would be head recursion as it invokes the recursive calls at the initial statement of the function. This recursive function calls itself for every year argument passed into it until it reaches the last year stored in the *dengueCasePerYear* map variable, which is 2019. Afterwards, the respective value based on the key passed to the map will be accumulated until all the operations have been executed. Aside from the **recursion** concept, another **pipeline** exists in this function as well which is *dengueCasePerYear.keySet().toArray()*.

DengueCaseReportSystem.java

```

78 public void displayTotalDengueCasesPerArea() {
79     DefaultTableModel totalDengueCasesPerAreaTableModel =
80         (DefaultTableModel) TblTotalDengueCasesPerArea.getModel();
81     pnlSort.setBackground(Color.decode("#D23743"));
82     pnlReset.setBackground(Color.decode("#96222A"));
83     totalDengueCasesPerAreaTableModel.setRowCount(0);
84     totalDengueCasesPerArea
85         .entrySet()
86         .stream()
87         .forEach(dengueCasesPerArea -> totalDengueCasesPerAreaTableModel
88             .addRow(new Object[] {
89                 dengueCasesPerArea.getKey(),
90                 dengueCasesPerArea.getValue()
91             }
92         )
93     );
94 }

```

Figure 18: DengueCaseReportSystem (Main) Class - displayTotalDengueCasesPerArea() method

To actually display the total dengue cases occurred for each district onto the GUI, the method named *displayTotalDengueCasesPerArea()* is implemented in the main class. Although this method does not receive any argument, it still uses a global variable exists in the main class to iterate the data that is pre-calculated in the constructor to be displayed. Much like the first sub-task, this sub-task displays the necessary data into another JTable component named *TblTotalDengueCasesPerArea*. The method yet again necessitates the invocation of *getModel()* method so that table can be manipulated. The two methods called from lines 81 till lines 82 are purely for aesthetic purposes meanwhile the method invoked in line 83 is to ensure the table will not get overpopulated with duplicate data because the task 2 involves the usage of the same table again. This method incorporates the concept of **pipelining** in *entrySet().stream().forEach()*, **lambdas** and **higher-order functions** in *dengueCasesPerArea* -> ... as well.

```

33     private Calculation calculation;
34     private LinkedHashMap<String, Integer> totalDengueCasesPerArea;
35
36     public DengueCaseReportSystem() {
37         initComponents();
38         setLocationRelativeTo(null);
39
40         try {
41             XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42             XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43             List<DengueCase> dengueCasesData =
44                 xlsxReader1.combinePartialDengueCasesData(
45                     xlsxReader1.readFile(1),
46                     xlsxReader2.readFile(2)
47                 );
48
49             calculation = new Calculation();
50
51             List<DengueCase> newDengueCasesData =
52                 calculation.getTotalCasesPerYear(dengueCasesData);
53
54             displayOverallDengueCases(newDengueCasesData);
55
56             totalDengueCasesPerArea = calculation.getTotalCasesPerArea(dengueCasesData);
57             displayTotalDengueCasesPerArea();

```

Figure 19: *DengueCaseReportSystem* (Main) Class - Constructor

As mentioned earlier, the *Calculation* class are necessary to be declared and initialised as shown in line 33 and line 49 so that the calculation methods can be accessed and applied. For the purpose of storing the data retrieved from the calculation produced in the method *getTotalCasesPerArea()*, the variable *totalDengueCasesPerArea* with data type of *LinkedHashMap<String, Integer>* is pre-declared globally in line 34 as well. The variable *totalDengueCasesPerArea* is a global variable because it allows other subsequent functions (sub-tasks) to read the same data. However, it should be notified that in line 56, wherein the

method *getTotalCasesPerArea()* was called, the data passed to its parameter is the *dengueCasesData*, which is the original data that contains no “Pahang” *DengueCase* instance instead of the *newDengueCasesData*. This is because the requirement of the sub-task only specifies the area, not including the state itself. Thence, the total dengue cases that occurred throughout the years for each district will only be calculated and displayed. Speaking of display, the display method, *displayTotalDengueCasesPerArea()* is invoked as well at the subsequent line.

Screenshot

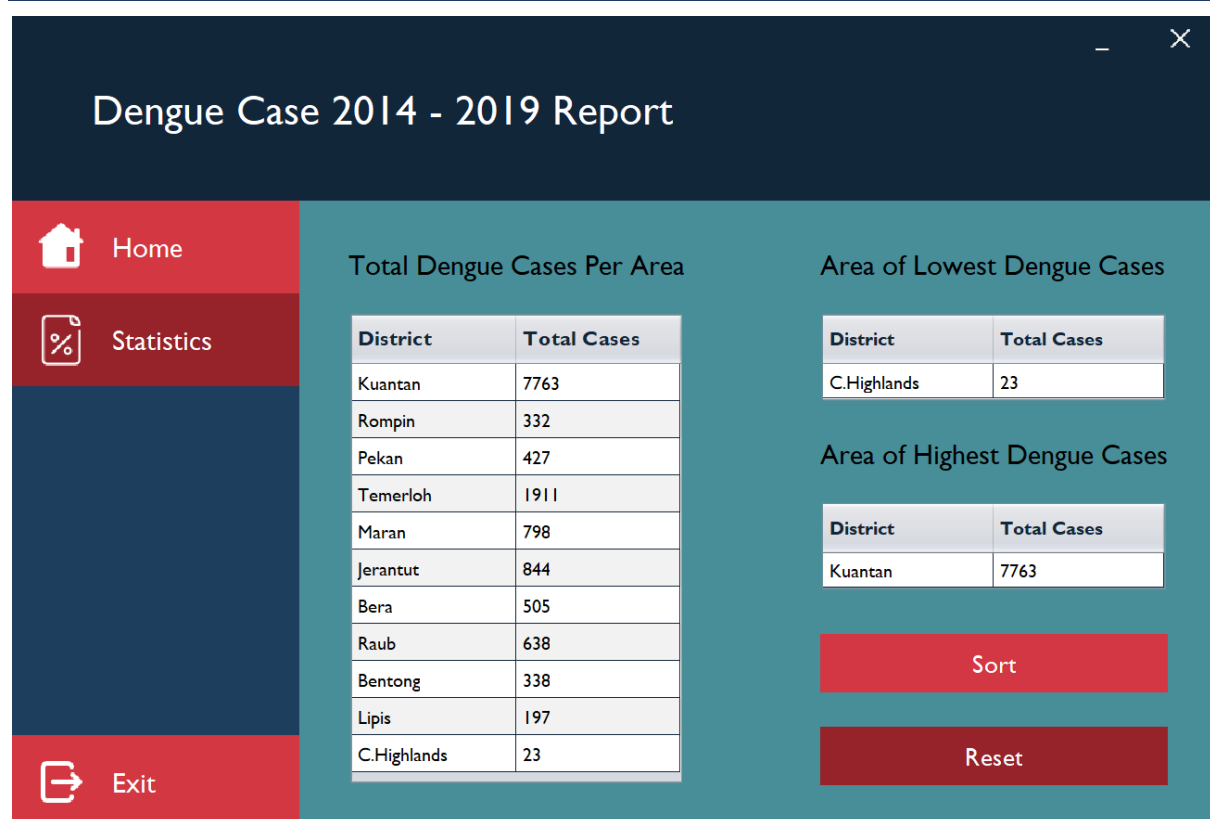


Figure 20: Graphical User Interface - Total Dengue Cases Per Area

As illustrated above, the total dengue cases occurred throughout the years for all the individual districts are displayed in tabular form.

Area of Lowest Dengue Cases

This sub-task involves the computation of the dengue cases data to search for the Pahang district that has the lowest dengue cases occurrences throughout the years from 2014 till 2019.

Calculation.java

```

70      //A function to identify the district with lowest overall dengue cases.
71      public Optional<Entry<String, Integer>> getLowestCasesArea
72      (LinkedHashMap<String, Integer> totalDengueCasesPerArea) {
73          return totalDengueCasesPerArea
74              .entrySet()
75              .stream()
76              .min((firstElement, secondElement) ->
77                  firstElement.getValue() < secondElement.getValue()
78                      ? -1
79                      : 1
80              );
81      }

```

Figure 21: Calculation Class - getLowestCasesArea()

To compute and determine the district that contains the least dengue cases throughout the years, the method *getLowestCasesMap()* is defined and implemented. This method accepts one argument, which is the data with data type *LinkedHashMap<String, Integer>* that is computed in the previous sub-task. This is because the previous sub-task calculates the total dengue cases occurred throughout the years for all districts. Through this data, the search operation on the district with the least dengue cases can be found with ease, especially with the help of *min()* method, wherein as long as the first element has lower value to the second element compared, it will return the first element, the comparison is performed across the entire lists of values. This function is yet again adopted the discussed programming concept including **pipelining** in *entrySet().stream().min()*, **lambda** and **higher-order function** in *(firstElement, secondElement) -> ...*. Moreover, the function is a **pure** function and imposes **no side effects**. The argument passed, *totalDengueCasesPerArea* is only used in iterating and finding the minimum value, the original source object will not be affected in whatsoever way. Not to mention, it can be seen that the return type of this function is an *Optional* type, which is another programming concept. Although very unlikely in the current scenario, returning an *Optional* data type helps the developer to write code in verifying whether the function will return a value or not because the data passed into the parameter could be empty. Furthermore, though the **optional** concept, the program will still continue execution despite discovering null values.


```

96  public void displayAreaOfLowestDengueCases() {
97      DefaultTableModel areaOfLowestDengueCasesTableModel =
98          (DefaultTableModel) TblAreaOfLowestDengueCases.getModel();
99      calculation.getLowestCasesArea(totalDengueCasesPerArea)
100          .ifPresent(dengueCasesPerArea -> areaOfLowestDengueCasesTableModel
101              .addRow(new Object[]{
102                  dengueCasesPerArea.getKey(),
103                  dengueCasesPerArea.getValue()
104              })
105          );
106  }

```

Figure 22: DengueCaseReportSystem (Main) Class - displayAreaOfLowestDengueCases() method

To display the district with the least dengue cases occurrences, the method *displayAreaOfLowestDengueCases()* is implemented in the main class. Since the data to be utilised is stored in the global variable, *totalDengueCasesPerArea*, the input parameter is not needed in this method. The specific district with the respective total number of cases is again, displayed through the JTable component with the assigned variable name called *TblAreaOfLowestDengueCases*. Similar to other tables, the method *getModel()* is invoked and stored into the *areaOfLowestDengueCasesTableModel* variable so that the table can be populated with the data of the district with lowest dengue cases calculated from the invocation of *getLowestCasesArea()* method. Due to the reason that the method *getLowestCasesArea()* returns an *Optional* type object, the method *ifPresent()* will become available to be implemented. Any code exists within the *ifPresent()* will only be executed if the value is present. If there is data exist, the table model will then add a row of data based on the key-value pair retrieved. Within this function, programming concepts like **pipelining** in *getLowestCases().ifPresent()*, **lambda** and **higher-order function** in *dengueCasesPerArea -> ...* wherein the lambda expression are passed into the *ifPresent()* method are demonstrated.

```

33     private Calculation calculation;
34     private LinkedHashMap<String, Integer> totalDengueCasesPerArea;
35
36     public DengueCaseReportSystem() {
37         initComponents();
38         setLocationRelativeTo(null);
39
40         try {
41             XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42             XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43             List<DengueCase> dengueCasesData =
44                 xlsxReader1.combinePartialDengueCasesData(
45                     xlsxReader1.readFile(1),
46                     xlsxReader2.readFile(2)
47                 );
48
49             calculation = new Calculation();
50
51             List<DengueCase> newDengueCasesData =
52                 calculation.getTotalCasesPerYear(dengueCasesData);
53
54             displayOverallDengueCases(newDengueCasesData);
55
56             totalDengueCasesPerArea = calculation.getTotalCasesPerArea(dengueCasesData);
57             displayTotalDengueCasesPerArea();
58             displayAreaOfLowestDengueCases();

```

Figure 23: *DengueCaseReportSystem (Main) Class - Constructor*

To actually display the data into the GUI, only the method *displayAreaOfLowestDengueCases()* is required to be invoked, provided that the *totalDengueCasesPerArea* variable is storing values retrieved from the calculation executed.

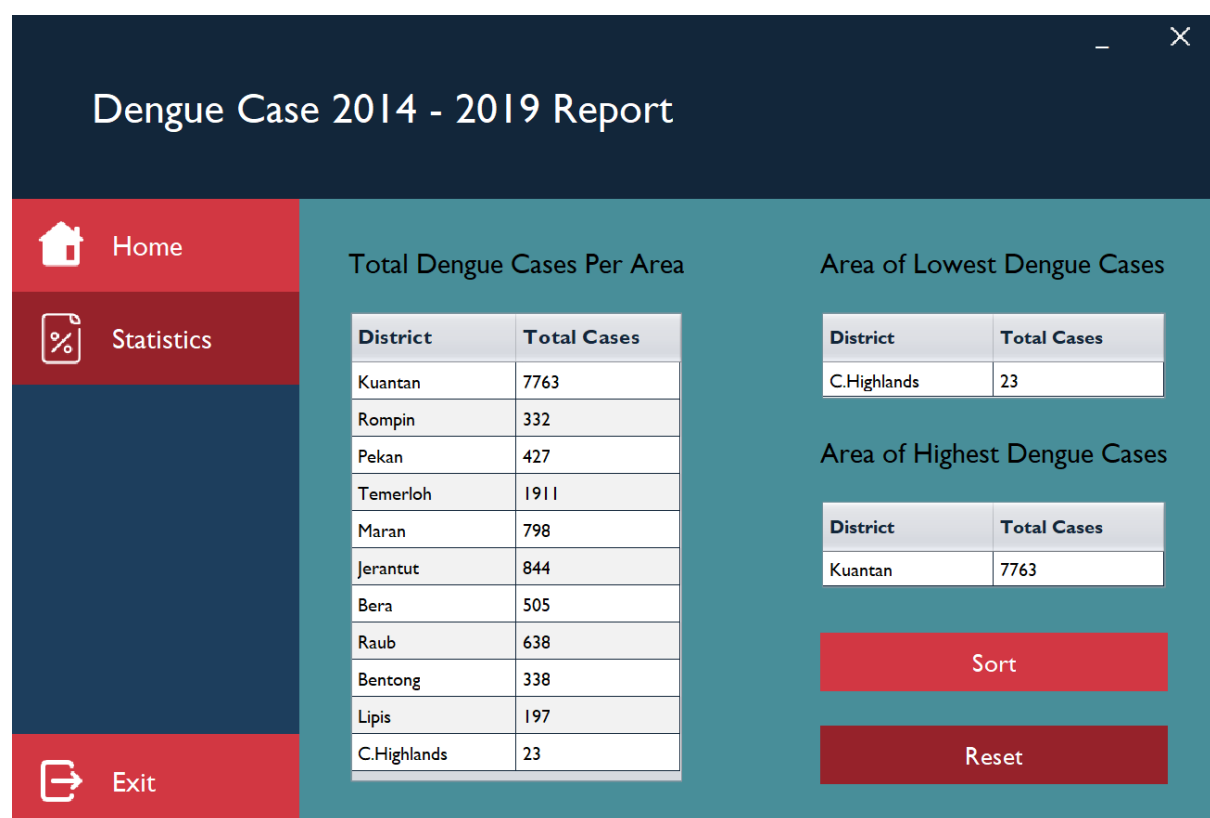


Figure 24: Graphical User Interface - Area of Lowest Dengue Cases

As shown above, the district with the least dengue cases occurred throughout the years are again displayed on a tabular form in the user interface.

Area of Highest Dengue Cases

This sub-task involves the computation of the dengue cases data to search for the Pahang district that has the most dengue cases occurrences throughout the years from 2014 till 2019.

Calculation.java

```
83 //A function to identify the district with highest overall dengue cases.
84 public Map<String, Integer> getHighestCasesArea
85     (LinkedHashMap<String, Integer> totalDengueCasesPerArea) {
86     return totalDengueCasesPerArea
87         .entrySet()
88         .stream()
89         .max((firstElement, secondElement) ->
90             firstElement.getValue() > secondElement.getValue()
91                 ? 1
92                 : -1
93         )
94         .stream()
95         .collect(Collectors.toMap(
96             element -> element.getKey(),
97             element -> element.getValue()
98         ));
99 }
```

Figure 25: Calculation Class - getHighestCasesArea()

This method `getHighestCasesArea()` is constructed in order to iterate through the `LinkedHashMap<String, Integer>` data that is passed as an argument for the purpose of finding the district that has the most dengue cases occurrences from 2014 to 2019. Similar to the previous sub-task, the argument is also taken from the data calculated in the sub-task that computes the total dengue cases occurred throughout the years for all Pahang districts. However, this method returns a `Map<String, Integer>` type rather than an *Optional* type because as seen from the code starting at line 94 till line 98 that collects the stream of data into *Map* of key-value pairs. Moreover, this method incorporates the `max()` method instead to find the highest dengue cases happening in a district. Based on the written manner within the `max()` method, as long as the first element has higher value to the second element compared, it will return the first element, the comparison is also performed across the entire lists of values. The `getHighestCasesArea()` method has a longer instance of **pipelining** concept, which is `entrySet().stream().max().stream().collect()`. Aside from pipelining, there is also the **lambda** expression, `(firstElement, secondElement) -> ...` within the `max()` method. Since the `max()` method accepts the lambda function as an argument, `max()` method is a **higher-order function**. Moreover, this method upholds the main principle in functional programming as it is **pure** and causes **no side effects**.

```
108 public void displayAreaOfHighestDengueCases() {
109     DefaultTableModel areaOfHighestDengueCasesTableModel =
110         (DefaultTableModel) TblAreaOfHighestDengueCases.getModel();
111     calculation.getHighestCasesArea(totalDengueCasesPerArea)
112         .entrySet()
113         .stream()
114         .forEach(dengueCasesPerArea -> areaOfHighestDengueCasesTableModel
115             .addRow(new Object[]{
116                 dengueCasesPerArea.getKey(),
117                 dengueCasesPerArea.getValue()
118             }));
119 }
120 }
```

Figure 26: DengueCaseReportSystem (Main) Class - displayAreaOfHighestDengueCases() method

To display the district with the highest dengue cases occurrences, the method *displayAreaOfHighestDengueCases()* is defined and implemented in the main class. This method does not need any input parameter as well because the variable *totalDengueCasesPerArea* is a global variable, making it accessible to perform the computations without the need to pass as argument. Similar to other implementations, this data as well will be displayed onto a JTable component named *TblAreaOfHighestDengueCases*. Moreover, the *getModel()* method is called so as to allow the table to be populated with data regarding the district with the highest dengue cases calculated from the invocation of *getHighestCasesArea()* method. This method has demonstrated several programming concepts as well, which includes **pipelining** in *.getHighestCasesArea().entrySet().stream.forEach()*, **lambda**, *dengueCasesPerArea -> ...* and **higher-order function** since the *forEach()* method accepts a lambda expression as an argument.

```

33     private Calculation calculation;
34     private LinkedHashMap<String, Integer> totalDengueCasesPerArea;
35
36     public DengueCaseReportSystem() {
37         initComponents();
38         setLocationRelativeTo(null);
39
40         try {
41             XlsxReader xlsxReader1 = new XlsxReader(EXCEL_FILE_PATH_2014_To_2016);
42             XlsxReader xlsxReader2 = new XlsxReader(EXCEL_FILE_PATH_2017_To_2019);
43             List<DengueCase> dengueCasesData =
44                 xlsxReader1.combinePartialDengueCasesData(
45                     xlsxReader1.readFile(1),
46                     xlsxReader2.readFile(2)
47                 );
48
49             calculation = new Calculation();
50
51             List<DengueCase> newDengueCasesData =
52                 calculation.getTotalCasesPerYear(dengueCasesData);
53
54             displayOverallDengueCases(newDengueCasesData);
55
56             totalDengueCasesPerArea = calculation.getTotalCasesPerArea(dengueCasesData);
57             displayTotalDengueCasesPerArea();
58             displayAreaOfLowestDengueCases();
59             displayAreaOfHighestDengueCases();
60         } catch (Exception ex) {
61             ex.printStackTrace();
62         }
63     }

```

Figure 27: *DengueCaseReportSystem (Main) Class - Constructor*

Similar to the previous sub-task, the data will be displayed onto the GUI by invoking the method *displayAreaOfHighestDengueCases()* in the main class constructor, provided that the *totalDengueCasesPerArea* variable is storing values retrieved from the calculation executed.

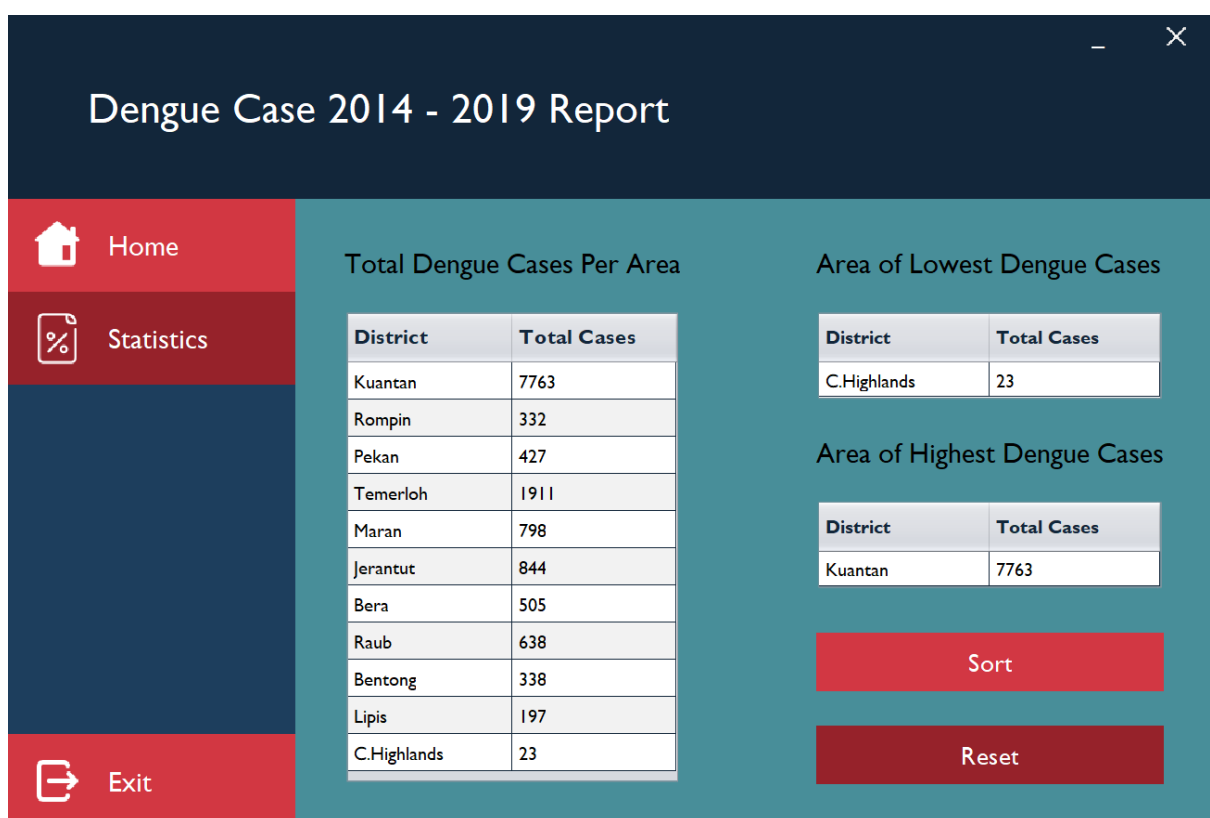


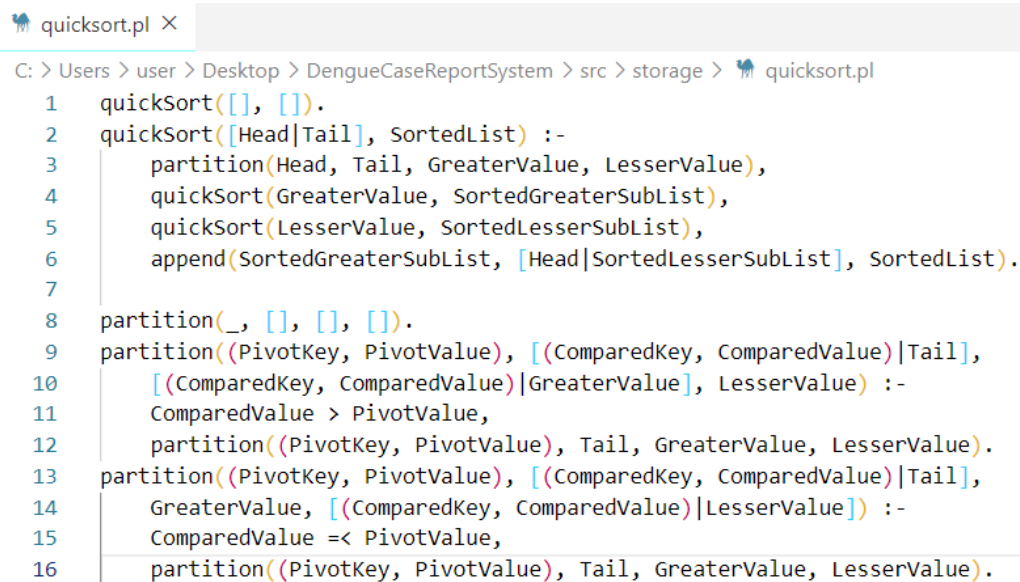
Figure 28: Graphical User Interface - Area of Highest Dengue Cases

As shown above, the district with the highest dengue cases occurred throughout the years are yet again displayed on a tabular form in the user interface.

Task 2

Prolog Rules for Sorting (Quicksort)

Before the operation for sorting the total dengue cases per area from highest to lowest (descending order) can be performed, the sorting algorithm and its respective prolog rules must first be defined and written in a so-called database. For this particular task, the sorting algorithm, Quicksort is applied, particularly using the first element as pivot. The prolog rules to perform the Quicksort algorithm on the data given is written and stored into the *quicksort.pl* file. Applying prolog to define complex sorting problems is very appropriate because the algorithm can be written with great compactness and compactness. Apart from that, it allows the algorithm written to be easily understandable as well.



```
1 quickSort([], []).
2 quickSort([Head|Tail], SortedList) :-
3     partition(Head, Tail, GreaterValue, LesserValue),
4     quickSort(GreaterValue, SortedGreaterSubList),
5     quickSort(LesserValue, SortedLesserSubList),
6     append(SortedGreaterSubList, [Head|SortedLesserSubList], SortedList).
7
8 partition(_, [], [], []).
9 partition((PivotKey, PivotValue), [(ComparedKey, ComparedValue)|Tail],
10    [(ComparedKey, ComparedValue)|GreaterValue], LesserValue) :-
11    ComparedValue > PivotValue,
12    partition((PivotKey, PivotValue), Tail, GreaterValue, LesserValue).
13 partition((PivotKey, PivotValue), [(ComparedKey, ComparedValue)|Tail],
14    GreaterValue, [(ComparedKey, ComparedValue)|LesserValue]) :-
15    ComparedValue <= PivotValue,
16    partition((PivotKey, PivotValue), Tail, GreaterValue, LesserValue).
```

Figure 29: quicksort.pl - Quicksort Rules in Prolog

Prior to explaining the rules, it must be noted that the total dengue cases per area data passed to the prolog file are formatted as such:

`[(Kuantan, 7763), (Rompin, 332), ...]`

While the entirety of the query is as such:

`?-quickSort([(Kuantan, 7763), (Rompin, 332), ...], SortedDengueCasesPerArea).`

Since the Quicksort algorithm adapts the divide and conquer principle, the first line of the prolog clause, *quickSort([], [])*. is to stop the sorting operation when there are no more elements left to be sorted. However, if there are elements to be sorted, the unsorted list of elements, or in this context, unsorted list of *DengueCase* instances, the list will be passed into

the first parameter of *quickSort()* rule in line 2, which is segregated into the variable *Head* and *Tail*. *Head* stores the first element to be used as pivot within the list while the *Tail* stores the rest of the elements. Moreover, the variable to store the sorted list of elements is passed into the second parameter variable. Within the *quickSort()* rule, there are several operations to be called, including *partition()* that divides the list of elements, two *recursive* calls of *quickSort()* rule to sort and store the sub-list of greater and lower values partitioned into either the variable *SortedGreaterSubList* or *SortedLowerSubList*, and lastly, *append()* that combines the divided sorted sub-list of greater and lower values into one variable, *SortedList*. The *partition()* rule within the *quickSort()* rule accepts the argument *Head* and *Tail* to partitioned into either the variable *GreaterValue* or *LesserValue* depending on the *Tail* value compared to the pivot, which is the *Head* value.

From line 8 till line 16 outlines the definition of the *partition()* rule. Again, since there could be no more elements to be partitioned, hence the clause in line 8 helps stopping the partition operation. However, the *partition()* rule from line 9 till line 12 will store the element with greater value compared to the pivot value into the *GreaterValue* variable while the *partition()* rule from line 13 till line 16 stores the element with lesser or equal to the pivot value into the *LesserValue* variable. Due to the fact that within *quickSort()* rule, the element in the *Head* and *Tail* variable passed to the *partition()* rule are in the format of (*Key*, *Value*), the *partition()* rule extracts the format of the element within the parameters so that the pivot value can be compared to the individual value stored in the *Tail* variable passed from the *quickSort()* rule.

Java-Prolog Connectivity

Apart from defining the rules, the prolog file written must be connected to the Java program developed, so as to allow queries to be passed to the prolog file, *quicksort.pl* containing the rules and perform the sorting operation.

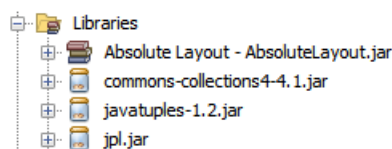


Figure 30: Java Imported Libraries

Aside from installing the SWI-Prolog file and setting the system environment variable path directed to the bin folder, the *jpl.jar* file located in the lib folder must be imported into the Java software program.

PrologConnection.java

```
15 public class PrologConnection {
16     private String filePath;
17
18     //A constructor to assign the file path retrieved.
19     public PrologConnection(String filePath) {
20         this.filePath = filePath;
21     }
22
23     //A function to check if the connection to the prolog file succeeded or not.
24     public void consult() throws Exception {
25         Query connectionQuery = new Query("consult('\" + this.filePath + '\"");
26         if (!connectionQuery.hasSolution())
27             throw new Exception("The connection to the prolog file has failed.");
28         connectionQuery.close();
29     }
```

Figure 31: PrologConnection Class

Afterwards, in order to establish a connection between the Java and the specific prolog file, *quicksort.pl* created, the method *consult()* is constructed and placed under the *PrologConnection* class for modularity purpose. Within the *consult()* method, a *Query* instance containing the consult query embedded with the file path to the *quicksort.pl* prolog file is instantiated. From line 26, an if statement containing the invocation of the *hasSolution()* method from the *Query* instance stored is created to check whether that the prolog file can be loaded into the Java program. If not, an *Exception* instance is thrown in line 27. After the prolog file has been loaded, the *Query* instance is closed with the *close()* method.

Sort in Descending Order

PrologConnection.java

```
31 //A function that can be called to sort the passed java data map.
32 public Map<String, Integer> sort(LinkedHashMap<String, Integer> totalDengueCasesPerArea) {
33     Term term = new Query(formulateQuery(totalDengueCasesPerArea))
34                 .oneSolution()
35                 .get("SortedDengueCasesPerArea");
36     return convert(term, new LinkedHashMap<>());
37 }
```

Figure 32: PrologConnection Class - sort() method

Once the connection between the prolog file and the Java program has been established, few functions were created in *PrologConnection* class in order to sort the passed total dengue cases per area data from Java in descending order, highest to lowest cases, using the Quicksort rules written previously. First and foremost, the *sort()* method that will be invoked from the main class is defined and implemented. This method takes the *LinkedHashMap<String, Integer>*

data calculated from task 1 as an argument under the name *totalDengueCasesPerArea*. From line 33, the *term* variable is declared to store the *Term*-typed data generated from the **pipeline** of methods, *Query().oneSolution().get()*. The *Query* instance contains the method *formulateQuery()* with the *totalDengueCasesPerArea* data passed in so that the data can be converted into a format that is readable in the prolog file. The method *oneSolution()* will then execute the query while the method *get()* is called to retrieve the *Term*-typed sorted data based on the variable name passed to the query. The *Term* is a data type applied in prolog, which will need to be converted and returned back to *Map<String, Integer>* by the method so that the sorted total dengue cases per area can be displayed with ease, which is demonstrated in line 36 with the *convert()* method that was pre-defined under the *PrologConnection* class as well.

```

39 //A function to formulate the prolog query based on the java data map passed.
40 public String formulateQuery(LinkedHashMap<String, Integer> totalDengueCasesPerArea) {
41     List<String> totalDengueCasesPerAreaList = new ArrayList<>();
42     totalDengueCasesPerArea
43         .entrySet()
44         .stream()
45         .forEach(dengueCasesPerArea -> totalDengueCasesPerAreaList
46             .add("'" +
47                 dengueCasesPerArea.getKey()
48                 + "'," +
49                 dengueCasesPerArea.getValue()
50                 + ")");
51     );
52 };
53 return "quickSort("
54     + totalDengueCasesPerAreaList.toString()
55     + ", SortedDengueCasesPerArea).";
56 }
57
58 //A function to convert the retrieved data from prolog to java data map.
59 public Map<String, Integer> convert(Term term, Map<String, Integer> unformattedDengueCasesPerArea) {
60     Map<String, Integer> formattedDengueCasesPerAreaMap =
61         new LinkedHashMap<>(unformattedDengueCasesPerArea);
62     formattedDengueCasesPerAreaMap
63         .put(term.arg(1).arg(1).toString().replace("'", ""),
64             term.arg(1).arg(2).intValue());
65
66     return ((term.arg(2).arity() != 0)
67         ? convert(term.arg(2), formattedDengueCasesPerAreaMap)
68         : formattedDengueCasesPerAreaMap);
69 }

```

Figure 33: PrologConnection Class - *formulateQuery()* and *convert()* method

The figure shown above are the detailed implementation of the two methods, *formulateQuery()* and *convert()* that are applied in the *sort()* method. Firstly, the *formulateQuery()* accepts the total dengue cases per area calculated as an argument under the name *totalDengueCasesPerArea* and returns a string of query that will be processed by the prolog file. From line 42 till 52, the *totalDengueCasesPerArea* data is iterated through the **pipeline** of methods, *totalDengueCasesPerArea.entrySet().stream().forEach()* so that each key-value pair is stored into the *List<String>* variable initialised in line 41 with the name *totalDengueCasesPerAreaList*. It should be noted that in line 55, the query contains the

variable named *SortedDengueCasesPerArea* which helps storing and retrieving the data sorted within the prolog file.

convert() method on the other hand, is a **recursive** method implemented to convert and format the parameter *term* that contains sorted data of *Term* data type into *Map<String, Integer>* data type. From line 62 till line 64, the first element returned from the invocation of *term.arg(1)* is formatted into key-value pair and appended into the *formattedDengueCasesPerAreaMap* variable that is initialised with the *Map*-typed data gathered from previous recursive calls in line 60 till 61. Since *term.arg(2)* represents the *Tail* variable such as in prolog, which means it contains the rest of the list of the elements. As long as the rest of the elements are not empty wherein was check using the method *arity()*, the *convert()* method will recursively call itself, each time passing the rest of the elements stored in *term* as well as the map of formatted data into the new invocation of *convert()* method until the there are no more elements to be formatted, which will then return the *Map*-typed *formattedDengueCasesPerAreaMap* that contains the formatted and sorted total dengue cases per area.

```

542 private void displaySortedDengueCasesPerArea(java.awt.event.MouseEvent evt) {
543     try {
544         DefaultTableModel totalDengueCasesPerAreaTableModel =
545             (DefaultTableModel) TblTotalDengueCasesPerArea.getModel();
546         PnlSort.setBackground(Color.decode("#96222A"));
547         PnlReset.setBackground(Color.decode("#D23743"));
548         totalDengueCasesPerAreaTableModel.setRowCount(0);
549         PrologConnection prologConnection = new PrologConnection(PROLOG_FILE_PATH);
550         prologConnection.consult();
551         Map<String, Integer> sortedDengueCasesPerAreaMap =
552             prologConnection.sort(totalDengueCasesPerArea);
553         sortedDengueCasesPerAreaMap
554             .entrySet()
555             .stream()
556             .forEach(dengueCasesPerArea ->
557                 totalDengueCasesPerAreaTableModel.addRow(
558                     new Object[]{
559                         dengueCasesPerArea
560                             .getKey()
561                             .substring(0, 1)
562                             .toUpperCase()
563                             + dengueCasesPerArea
564                                 .getKey()
565                                 .substring(1),
566                         dengueCasesPerArea.getValue()
567                     }
568                 );
569     } catch (Exception ex) {
570         ex.printStackTrace();
571     }
572 }

```

Figure 34: DengueCaseReportSystem (Main) Class - displaySortedDengueCasesPerArea() method

This method, *displaySortedDengueCasesPerArea()* is a method that will establish the prolog connection and perform the sorting operation on the pre-calculated *totalDengueCasesPerArea* data through the invocation of *consult()* and *sort()* method from the instantiation of *PrologConnection* class. The *displaySortedDengueCasesPerArea()* is invoked every time the system user presses the “Sort” button from the user interface. From line 543 and 544, it can be seen that the *TblTotalDengueCasesPerArea* table is used again when invoking *getModel()* method, same table as one of the sub-task in task 1. This is because it was intended to display the sorted data on the same table. Hence the reason why the method *setRowCount(0)* is called in line 547 so that the table is emptied and repopulated with the sorted data. After the sorted data was retrieved and stored into the new variable declared in line 550, the data is iterated through **pipelining** of several methods, *sortedDengueCasesPerAreaMap.entrySet().stream().forEach()* so that the table can be populated with the sorted data. Another instance of **pipeline** concept was displayed from line 558 till line 561, *dengueCasesPerArea.getKey().substring(0, 1).toUpperCase()* in order to

capitalise the first letter of every district, which in overall is for uniformity and aesthetic purpose in the user interface. **Lambda** expression was utilised in this method as well, which is *dengueCasesPerArea* -> Since the *forEach()* method accepts function as argument, which implies that it is a **higher-order function**.

```

575 private void displayOriginalTotalDengueCasesPerArea(java.awt.event.MouseEvent evt) {
576     displayTotalDengueCasesPerArea();
}

```

Figure 35: *DengueCaseReportSystem (Main) Class - displayOriginalTotalDengueCasesPerArea () method*

This is another method specifically created for when the button “Reset” is pressed on the user interface wherein it will empty and repopulate the same table *TblTotalDengueCasesPerArea* with the unsorted total dengue cases per area data.

Screenshot

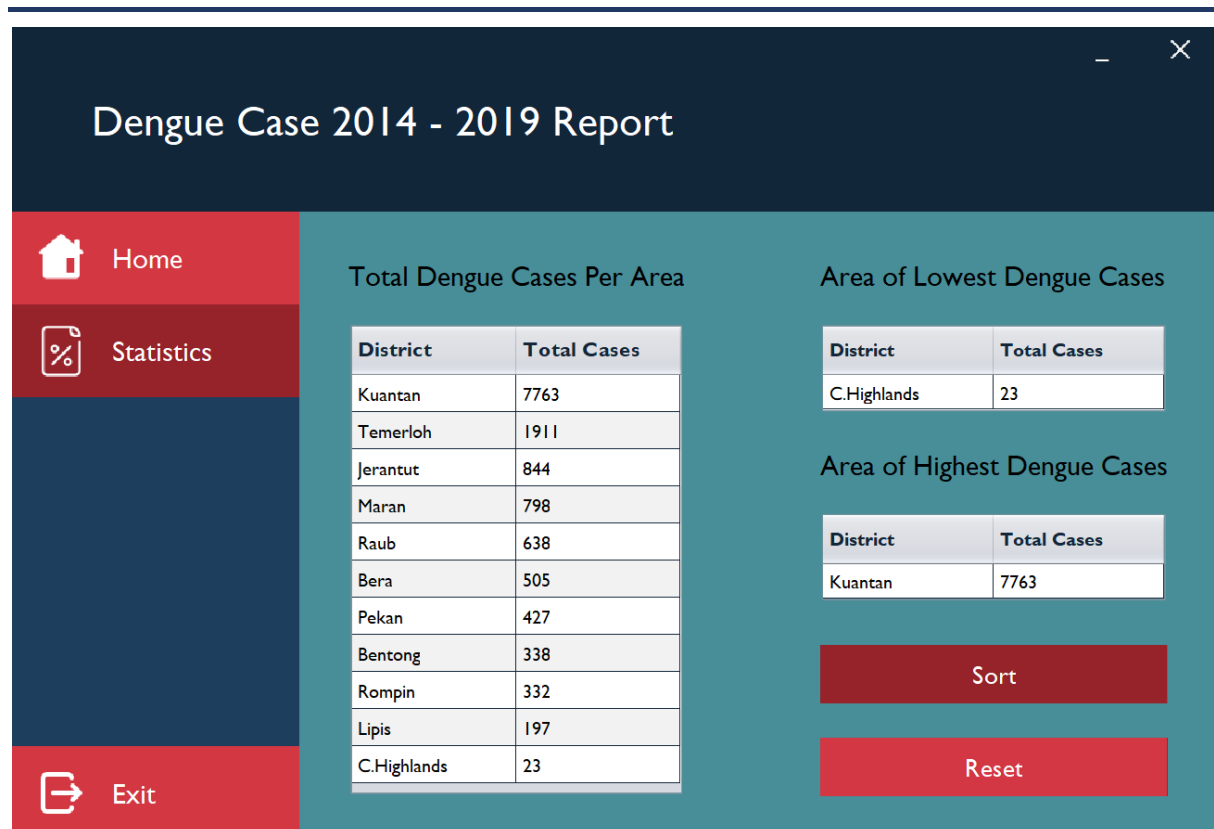


Figure 36: *Graphical User Interface - Sorted Total Dengue Cases Per Area*

As shown above, the district with its respective total dengue cases occurred throughout the years are displayed in the table in descending order, from highest to lowest cases, after the button “Sort” was pressed.

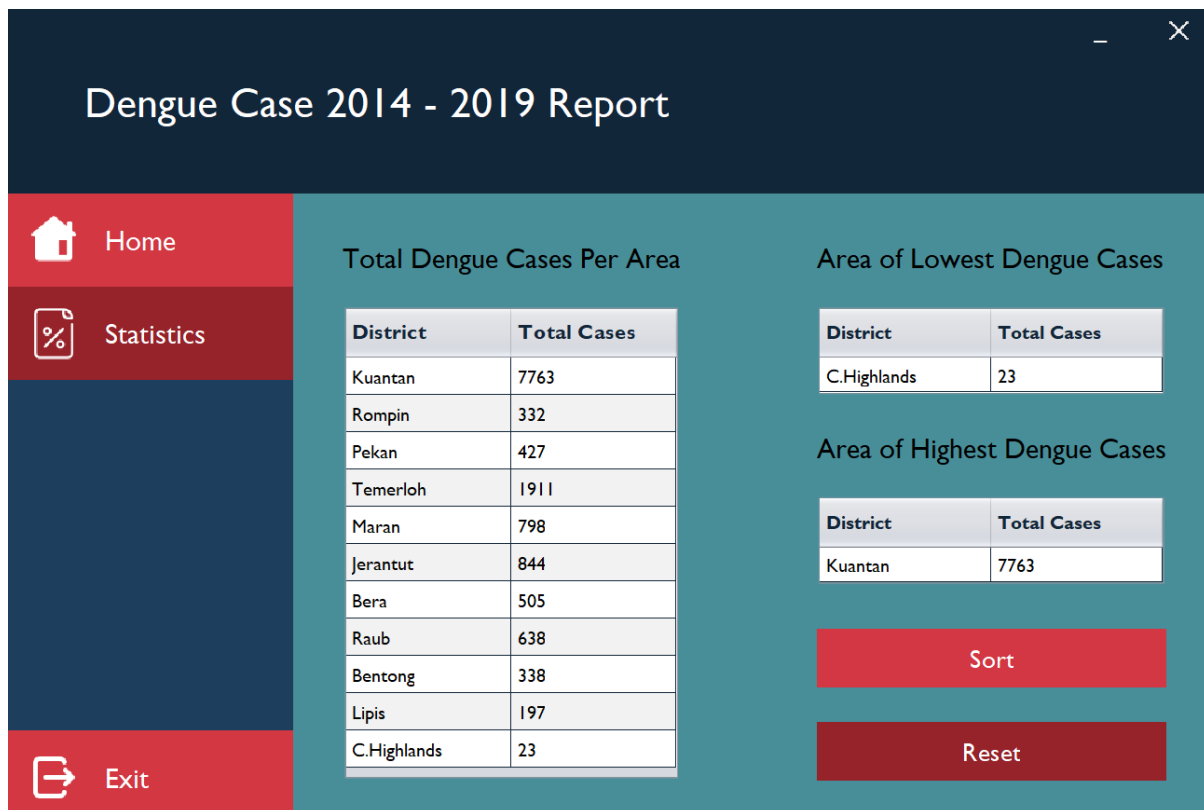


Figure 37: Graphical User Interface - Total Dengue Cases Per Area

As shown above, the total dengue cases per district data are arranged back to its original arrangement in the table after the button “Reset” was pressed.

Conclusion

In overall, the dengue cases report system was successfully implemented through incorporating multiple programming paradigms, primarily the functional and logic paradigm. Different programming concepts were adapted into the system, including collections, pipelining, purity, higher-order function, optional, lambda, recursion, currying. The importance and benefits of each concepts were heavily discussed and greatly understood as well. Moreover, the usage of Prolog was further comprehended from the implementation of Quicksort algorithm. From the development, the strengths and weaknesses of each programming paradigm were realised as well.

References

Akhil Bhadwal, 2020. *Functional Programming: Concepts, Advantages, Disadvantages, and Applications*. [Online]

Available at: <https://hackr.io/blog/functional-programming>
[Accessed 19 March 2021].

Avram, A., 2020. *Functional Programming in Java: Optional*. [Online]
Available at: <https://www.cognizantsoftvision.com/blog/functional-programming-in-java-optional/>

[Accessed 19 January 2021].

Baeldung, 2019. *Introduction to Javatuples*. [Online]

Available at: <https://www.baeldung.com/java-tuples>

[Accessed 19 January 2021].

Buonanno, E., 2017. *Functional Programming in C#: How to write better C# code*. 1st ed. New York City: Manning Publications.

Deepak Gupta, 2018. *Javascript- Currying VS Partial Application*. [Online]

Available at: <https://towardsdatascience.com/javascript-currying-vs-partial-application-4db5b2442be8>

[Accessed 19 January 2021].

Edureka, 2019. *How to Implement Shallow Copy and Deep Copy in Java*. [Online]

Available at: <https://www.edureka.co/blog/shallow-and-deep-copy-java/>

[Accessed 20 January 2021].

Fowler, M., 2004. *Lambda*. [Online]

Available at: <https://martinfowler.com/bliki/Lambda.html>

[Accessed 19 January 2021].

Fowler, M., 2015. *Collection Pipeline*. [Online]

Available at: <https://martinfowler.com/articles/collection-pipeline/#WhenToUseIt>

[Accessed 19 January 2021].

GeeksforGeeks, 2019. *Functional Interfaces In Java*. [Online]

Available at: <https://www.geeksforgeeks.org/functional-interfaces-java/>

[Accessed 19 January 2021].

Herring, M., 2017. *Why Pure Functions? - 4 benefits to embrace*. [Online]
Available at: <https://www.deadcoderising.com/2017-06-13-why-pure-functions-4-benefits-to-embrace-2/>

[Accessed 19 March 2021].

HowToDoInJava.com, 2020. *Collections in Java*. [Online]
Available at: <https://howtodoinjava.com/java-collections/>

[Accessed 19 January 2021].

LogicBig, 2017. *Java 8 Streams - Lazy evaluation*. [Online]
Available at: <https://www.logicbig.com/tutorials/core-java-tutorial/java-util-stream/lazy-evaluation.html>

[Accessed 19 January 2021].

Lowe, D., 2021. *Shadowed Classes or Variables in Java*. [Online]
Available at: <https://www.dummies.com/programming/java/shadowed-classes-or-variables-in-java/>

[Accessed 19 January 2021].

Nataraja Gootooru, 2020. *What is stream pipelining in Java 8?*. [Online]
Available at: <https://www.java2novice.com/java-interview-questions/java-8-stream-pipelining/>

[Accessed 19 January 2021].

Nnamdi, C., 2018. *Understanding Currying in JavaScript*. [Online]
Available at: <https://blog.bitsrc.io/understanding-currying-in-javascript-ceb2188c339>

[Accessed 19 January 2021].

Normand, E., 2019. *WHAT IS A HIGHER-ORDER FUNCTION?*. [Online]
Available at: <https://lispcast.com/what-is-a-higher-order-function/>

[Accessed 19 January 2021].

Sahu, J., 2014. *Lambda Expressions in Java 8: Why and How to Use Them*. [Online]
Available at: <https://www.nagarro.com/en/blog/post/26/lambda-expressions-in-java-8-why-and-how-to-use-them>

[Accessed 19 January 2021].

Saumont, P.-Y., 2017. *What Is Referential Transparency?*. [Online]
Available at: <https://www.sitepoint.com/what-is-referential-transparency/>
[Accessed 19 January 2021].

Sindhuja Hari, 2021. *Programming Paradigms: A must know for all Programmers*. [Online]
Available at: <https://hackr.io/blog/programming-paradigms>
[Accessed 19 January 2021].

Thelin, R., 2021. *What are lambda expressions?*. [Online]
Available at: <https://www.educative.io/blog/java-lambda-expression-tutorial#what>
[Accessed 19 January 2021].

Tusamma Sal Sabil, 2020. *Tail Recursion and Head Recursion*. [Online]
Available at: <https://medium.com/@tssovi/tail-recursion-and-head-recursion-d9aeb8478b12>
[Accessed 19 January 2021].

Tutorialspoint, 2021. *Functional Programming - Recursion*. [Online]
Available at: https://www.tutorialspoint.com/functional_programming/functional_programming_recursion.htm
[Accessed 19 January 2021].

Venkat Subramaniam, 2018. *Functional purity*. [Online]
Available at: <https://developer.ibm.com/languages/java/articles/j-java8idioms11/>
[Accessed 19 January 2021].