Part C – Development – N.R – Computer Science Wordcount: 1084

TABLE OF CONTENTS

Concept PAGE

Intuitive GUI – Additional Libraries (JavaFX) 2

Calculations 4

Object Orientation 7

Graphing 8

File Reading / Writing 9

Sources 11

Preposition:

In this writeup, aspects such as error handling have not been treated individually as they are evident in most methods and have the purpose of preventing the application from crashing in the unlikely event. Furthermore, the relationship between object orientation and most other aspects, particularly the Intuitive GUI, are expected to be noticed due to the frequent calling of methods from other classes. The application/solution on a whole is strongly intertwined with most classes being used in every scene.

N.R

Intuitive GUI – Additional Libraries (JavaFX)

I made the decision to use JavaFX rather than Swing as a GUI library as {Client} and I agreed that it a. looked better, and b. allowed for more flexibility when dealing with user-inputs. This would all aid in fulfilling (primarily) criterion A as the user would find the GUI far more intuitive. By using fx:id's, specific containers (e.g. Labels) could be altered individually.

Imports and containers for score calculation scene.

```java
import java.io.IOException;
import java.net.URL;
import java.util.ResourceBundle;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.Initializable;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableColumn;
import javafx.scene.control.TableView;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.layout.GridPane;
import java.text.SimpleDateFormat;
import java.util.ArrayList;
import java.util.Optional;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.control.Alert;
import javafx.scene.control.ButtonType;
import javafx.stage.Stage;
```

```java
@FXML Button mmBtn;
@FXML Label totalNettoLabel;
@FXML Label totalBruttoLabel;
@FXML Label totalToParLabel;
@FXML Label totalScoreLabel;
@FXML GridPane numGirdPane;
@FXML Button btn1;
@FXML Button btn2;
@FXML Button btn3;
@FXML Button btn4;
@FXML Button btn5;
@FXML Button btn6;
@FXML Button btn7;
@FXML Button btn8;
@FXML Button btn9;
@FXML Button btn0;
@FXML Button btnNA;
@FXML Button btnDEL;
@FXML Button btnNEXT;
@FXML Label outputNumber;
@FXML TableView calculationTable;
@FXML TableColumn<PredefinedColumns, Integer> hole;
@FXML TableColumn<PredefinedColumns, Integer> par;
@FXML TableColumn<PredefinedColumns, Integer> hdc;
@FXML TableColumn<PredefinedColumns, Integer> shots;
@FXML TableColumn<PredefinedColumns, Integer> score;
@FXML TableColumn<PredefinedColumns, Integer> netto;
@FXML TableColumn<PredefinedColumns, Integer> brutto;
@FXML TableColumn<PredefinedColumns, Integer> toPar;
@FXML Label oldHDCLabel;
@FXML Label newHDCLabel;
@FXML Button finishBtn;
@FXML Button continueBtn;
```

N.R

Showing how all of these containers work together to produce a simple and in particular easy to use interface is shown by using the final application below. Only buttons on the numpad are necessary to be pressed even though 32 methods work together to produce accurate results. Ingenuity is evident as multiple complicated and extensive methods work 'behind the scenes' to meet the requirements.

Please Enter your Score

Hole	Par	Stroke Index	Strokes	Score	Netto	Brutto	To Par
1	4	5	2	5	3	1	1
2	3	13	2	0	0	0	0
3	4	3	2	0	0	0	0
4	4	9	2	0	0	0	0
5	4	17	1	0	0	0	0
6	5	11	2	0	0	0	0
7	4	1	2	0	0	0	0
8	3	15	2	0	0	0	0
9	5	7	2	0	0	0	0
10	4	2	2	0	0	0	0
11	5	18	1	0	0	0	0
12	3	14	2	0	0	0	0
13	4	12	2	0	0	0	0
14	4	6	2	0	0	0	0
15	3	16	1	0	0	0	0
16	5	8	2	0	0	0	0
17	4	10	2	0	0	0	0
18	5	4	2	0	0	0	0

Total Score:	Total Netto:	Total Brutto:	Overall to Par:
5	3	1	1

Please use the NumPad Below:

1	2	3
4	5	6
7	8	9
	0	
DEL	N/A	NEXT

Old HDC: New HDC

28.4

The following is an example of all the logic that is behind the simple pressing of the "next" button, which confirms the entry of a score for a hole. This is one many examples where dozens of lines of code are executed without the user realising it, further illustrating the intuitive GUI, which is exactly what {Client} required. Creating such a solution was very complicated and required ingenuity as this is an obvious constraint and it not being possible/ feasible to freely program.

```
private void setNEXT(ActionEvent event) throws IOException {
    if (outputNumber.getText().length() > 0) {
        int index = calculationTable.getSelectionModel().selectedIndexProperty().get();
        int scoreInput = Integer.parseInt(outputNumber.getText());
        int par = data.get(index).getHolePar();
        int shot = data.get(index).getShots();
        PlayerController pc = new PlayerController(); //Player controller object
        //Score input
        PredefinedColumns selectedColumn = data.get(index);
        selectedColumn.setScore(scoreInput);
        data.set(index, selectedColumn);
        score.setCellValueFactory(new PropertyValueFactory<>("score"));
        //Netto
        int netto = pc.calculateSingleNetto(shot, scoreInput, par);
        selectedColumn.setNetto(netto);
        data.set(index, selectedColumn);
        this.netto.setCellValueFactory(new PropertyValueFactory<>("netto"));
        //Brutto
        int brutto = pc.calculateSingleBrutto(scoreInput, par);
        selectedColumn.setBrutto(brutto);
        data.set(index, selectedColumn);
        this.brutto.setCellValueFactory(new PropertyValueFactory<>("brutto"));
        //To Par
        int toPar = pc.calculateToSinglePar(scoreInput, par);
        selectedColumn.setToPar(toPar);
        data.set(index, selectedColumn);
        this.toPar.setCellValueFactory(new PropertyValueFactory<>("toPar"));
        calculationTable.setItems(data);
        outputNumber.setText("");
        //Total calculations
        int totalNetto = 0;
        int totalBrutto = 0;
        int totalToPar = 0;
        int totalScore = 0;
        for (PredefinedColumns p : data) {
            totalNetto += p.getNetto();
            totalBrutto += p.getBrutto();
            totalScore += p.getScore();
            totalToPar += p.getToPar();
        }
        totalBruttoLabel.setText(Integer.toString(totalBrutto));
        totalNettoLabel.setText(Integer.toString(totalNetto));
        totalScoreLabel.setText(Integer.toString(totalScore));
        totalToParLabel.setText(Integer.toString(totalToPar));
        if (index == 17) {
            newHDCLabel.setText(Double.toString(pc.calculateNewHDC(oldHDC, totalNetto)));
            finishBtn.setVisible(true);
            if (isGuest == false) {
                continueBtn.setVisible(true);
            }
        }
        calculationTable.getSelectionModel().select(index + 1); //Move to next row
    }
}
```

Index (where in the table we are) is determined and used to retrieve relevant values.

Each of the 4 rows which are calculated are populated following the pressing of the button.

Totals are summed and updated/ outputted.

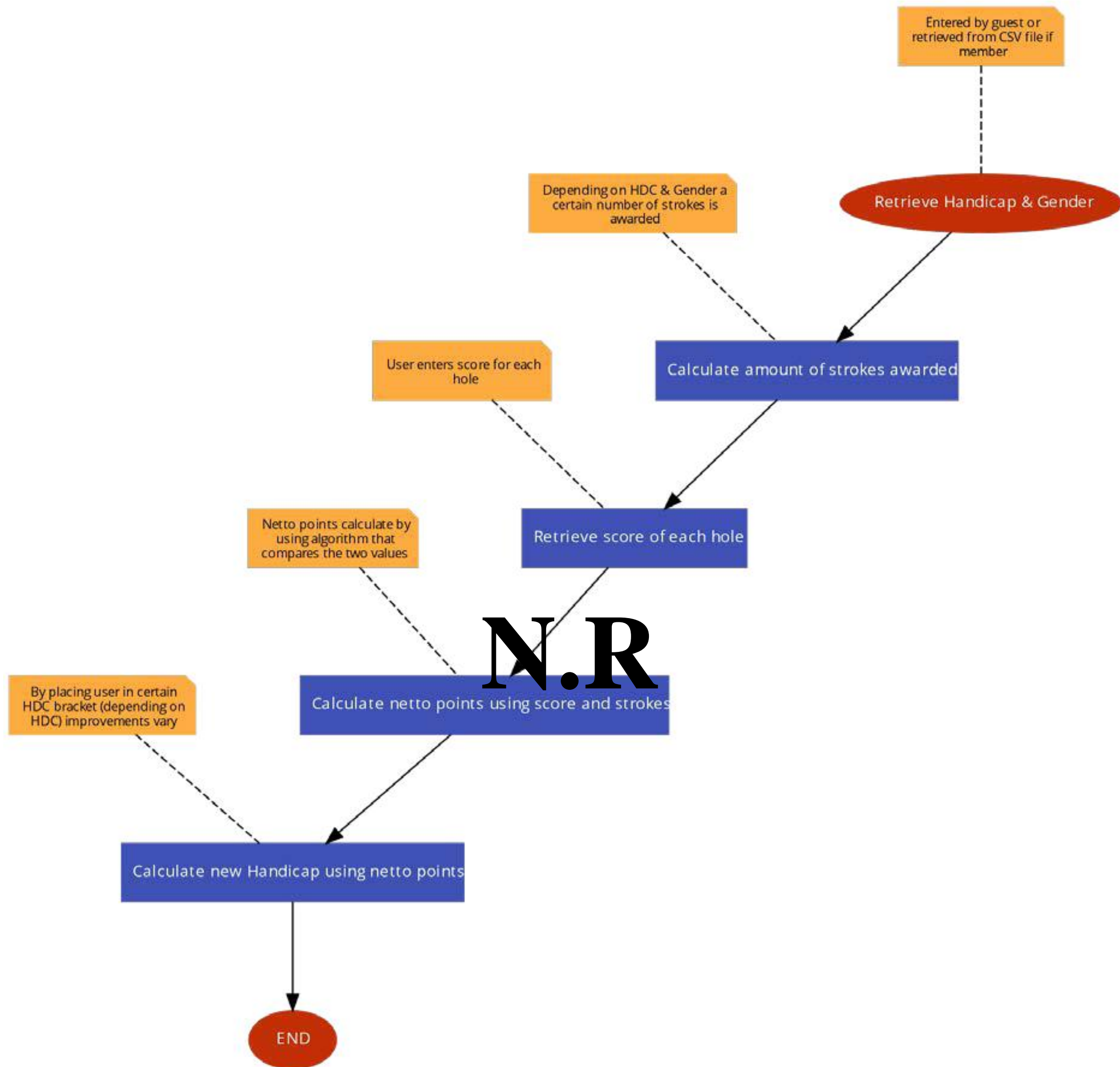
If "next" is pressed on the last hole, calculateNew HDC method is called from PlayerControll er class.

"Continue" button only appears if the user is a member, further simplifying the GUI for the user and making it more intuitive.

N.R

Calculations

Lying at the heart of the application, as the calculations are responsible for creating most values (new HDC, netto, etc.) I soon realized that to do certain calculations, previous calculations were necessary, which in turn required further previous calculations. The following flowchart illustrates this.



N.R

As the flowchart shows, a unique series of algorithms are required to fulfil Mr. Williams' requirements. Said algorithms and their complexity are outlined below:

As the flowchart shows, we have to begin with calculating the correct amount of strokes for each hole.

```
1  public int[] calculateVorgabe(double handicap, int slope, double courseRating) {
2
3   int[] arrayToReturn = new int[18];
4   int[][] par = getPar();
5   int length = arrayToReturn.length;
6   int vorgabe = 0;
7   if (handicap <= 36.0) {
8    double dbVorgabe = (double)(-handicap * (double)(slope) / 113) - courseRating + 73;
9    vorgabe = abs((int) Math.round(dbVorgabe));
10  } else if (handicap > 36) {
11   vorgabe = (int) handicap + 6;
12  }
13  //Vorgabe <18 implies at max one shot on each hole
14  if (vorgabe <= 18) {
15   for (int i = 0; i < length; i++) {
16    if (par[i][0] <= vorgabe) {
17     arrayToReturn[i] = 1;
18    } else {
19     arrayToReturn[i] = 0;
20    }
21   }
22   //Vorgabe >18 and <37 implies at least 1 shots on each hole, 2 on some
23  } else if (vorgabe > 18 && vorgabe < 37) {
24   for (int i = 0; i < length; i++) {
25    if (par[i][0] <= vorgabe - 18) {
26     arrayToReturn[i] = 2;
27    } else {
28     arrayToReturn[i] = 1;
29    }
30   }
31   //Vorgabe >36 implies at least 2 shots on each hole
32  } else if (vorgabe > 36) {
33   for (int i = 0; i < length; i++) {
34    if (par[i][0] <= vorgabe - 36) {
35     arrayToReturn[i] = 3;
36    } else {
37     arrayToReturn[i] = 2;
38    }
39   }
40  }
41
42  return arrayToReturn;
43
44 }
```

N.R

Ingenuity required to customize formula to {Client}'s golf course, "slope" and "courseRating" are specific for the GC.

Custom nested if statements required to fulfil {Client}'s requirements. 3 different HDC 'categories' to correctly award strokes.

arrayToReturn contains correct distribution of strokes for the array of size [18] (18 holes on a golf course), which can then be used for further calculations.

Hole	Par	Stroke Index	Strokes	Score	Netto	Brutto	To Par
1	4	5	2				
2	3	13	2				
3	4	3	2				
4	4	9	2				
5	4	17	1				
6	5	11	2				
7	4	1	2				
8	3	15	2				
9	5	7	2				
10	4	2	2				
11	5	18	1				
12	3	14	2				
13	4	12	2				
14	4	6	2				
15	3	16	1				
16	5	8	2				
17	4	10	2				
18	5	4	2				

Here we can see this algorithm in action. Using an inputted HDC, strokes have been calculated a correctly allocated. 33 Strokes have been awarded meaning that 2 strokes should be awarded on the 15 hardest holes (those with Stroke Index 1-15) and 1 Stroke should be awarded on the 3 easiest holes (SI 16-18). This has been done correctly proving that the algorithm works successfully.

Now that strokes have been calculated, we can calculate the respective netto points depending on the hole's score and par.

As will be explained later, this method is created so it can be called from the central calculation class in order to maintain modularity. A custom formula has been created to use "shots" (strokes), score and par.

```
1  public int calculateSingleNetto(int shots, int score, int par) {
2          //"shots" is really "strokes"
3          int nettoToReturn = 0;
4          nettoToReturn = ((par + shots) - (score)) + 2;
5          if (nettoToReturn > 0) {
6              return nettoToReturn;
7          } else {
8              return 0;
9          }
10     }
```

Having calculated netto points, the new Handicap can be calculated by using a custom and complex categorical system, which is conform with the relevant logic.

See Appendix C.1 to understand different "handicap categories", the term "pufferzone" (buffer zone) and "geht hoch" (increase in HDC). These are differentiated here by using cascading if-statements. "Vorgabenklasse" is simply the German equivalent word. In each case, an improvement in HDC occurs if nettoPoints>36. However, the HDC improves (decreases) by a different margin per point for every HDC bracket. Therefore, complex algorithmic thinking was necessitated; in order to account for all scenarios and therefore complete calculations correctly and thus reliably for Mr. Williams.

```
1  public double calculateNewHDC(double currentHDC, int nettoPoints) {
2
3          //double newHDC = 0.0;
4          BigDecimal newHDC = null;
5
6          int finalNetto = nettoPoints - 36;
7          //Vorgabenklasse 1:
8          if (currentHDC >= 0.0 && currentHDC <= 4.4) {
9              if (nettoPoints > 36) {
10                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 0.1));
11             } //Pufferzone
12             else if (nettoPoints == 35 || nettoPoints == 36) {
13                 newHDC = BigDecimal.valueOf(currentHDC);
14             } //Geht Hoch
15             else {
16                 newHDC = BigDecimal.valueOf(currentHDC + 0.1);
17             }
18         } //Vorgabenklasse 2:
19         else if (currentHDC >= 4.5 && currentHDC <= 11.4) {
20             if (nettoPoints > 36) {
21                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 0.2));
22             } //Pufferzone
23             else if (nettoPoints == 34 || nettoPoin[illegible]5 || [illegible]to[illegible]ints == 36) {
24                 newHDC = BigDecimal.valueOf(current[illegible]DC[illegible]
25             } //Geht Hoch
26             else {
27                 newHDC = BigDecimal.valueOf(currentHDC + 0.1);
28             }
29         } //Vorgabenklasse 3:
30         else if (currentHDC >= 11.5 && currentHDC <= 18.4) {
31             if (nettoPoints > 36) {
32                 //Verbesserung
33                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 0.3));
34             } //Pufferzone
35             else if (nettoPoints == 33 || nettoPoints == 34 || nettoPoints == 35 || nettoPoints == 36) {
36                 newHDC = BigDecimal.valueOf(currentHDC);
37             } //Geht Hoch
38             else {
39                 newHDC = BigDecimal.valueOf(currentHDC + 0.1);
40             }
41         } //Vorgabenklasse 4:
42         else if (currentHDC >= 18.5 && currentHDC <= 26.4) {
43             if (nettoPoints > 36) {
44                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 0.4));
45             } //Pufferzone
46             else if (nettoPoints == 32 || nettoPoints == 33 || nettoPoints == 34 || nettoPoints == 35 || nettoPoints == 36) {
47                 newHDC = BigDecimal.valueOf(currentHDC);
48             } //Geht Hoch
49             else {
50
51                 newHDC = BigDecimal.valueOf(currentHDC + 0.1);
52             }
53         } //Vorgabenklasse 5:
54         else if (currentHDC >= 26.5 && currentHDC <= 36.0) {
55             if (nettoPoints > 36) {
56                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 0.5));
57             } //Pufferzone
58             else if (nettoPoints == 31 || nettoPoints == 32 || nettoPoints == 33 || nettoPoints == 34 || nettoPoints == 35 || nettoPoints == 36) {
59                 newHDC = BigDecimal.valueOf(currentHDC);
60             } //Geht Hoch
61             else {
62                 newHDC = BigDecimal.valueOf(currentHDC + 0.2);
63             }
64         } //Vorgabenklasse 6:
65         else if (currentHDC <= 54 && currentHDC >= 37.0) {
66             if (nettoPoints > 36) {
67                 newHDC = BigDecimal.valueOf(currentHDC - (finalNetto * 1));
68             }
69
70         }
71         double HDCRounded = (double) Math.round(newHDC.doubleValue() * 100) / 100;
72         return HDCRounded;
73     }
```

N.R

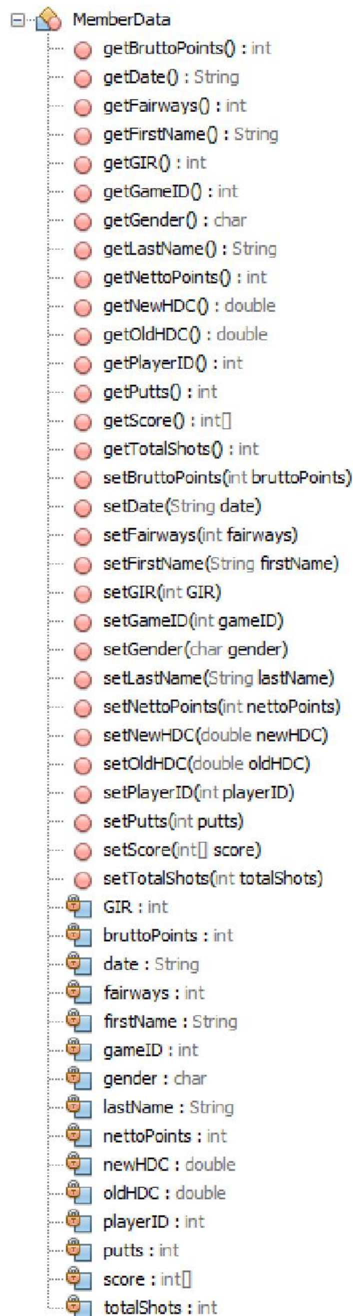
Math functions used to create usable, accurate double. BigDecimal limits value to one decimal place, assuring conformity. Again, an original solution was required.

Object Orientation

One key aspect of the entire application, which was crucial to its success in terms of usability and extensibility, was the usage of Object Orientation in the creation process. Methods, to e.g. append information could now be called across multiple scenes and restrictions were removed.

By using OOP I found that it is useful to deal with multiple classes and it made my program more modular. Being more modular made it easier to version the application, meaning that consequently, extensibility is a lot easier, and so is maintenance. Moreover, due to the size and the number of classes, OOP made the programming substantially easier and allowed for the many complicated methods to be used in various scenarios.

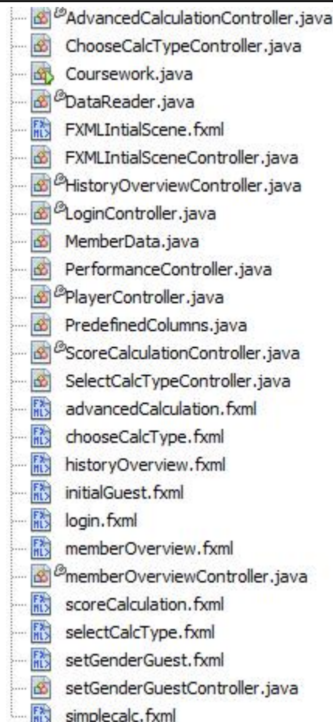
Encapsulation



In most cases I used encapsulation to ensure that the variables could not be accessed directly from another class to maintain data conformity. Each variable has its own getter and setter by which data can be retrieved and defined/altered. This increases usability and makes it possible to separate an object's implementation from its behaviour to restrict access to its internal data meaning that possible errors are averted. This will make it easy to expand Mr. Williams' application in future patches and it has made the creation process easier as issues could be fixed faster.

N.R

The fact that every JavaFXML file has its own Controller class required due to OOP, this can be seen in the file structure below. Every scene could now be fully extended on its own, which would not have been possible otherwise.



Graphing

Correct, readable and clear graphing was imperative to the fulfilment of success criteria C and the general functionality of the product (one of {Client}'s main issues that this solution should solve was looking at how his students developed over time).

I decided to use JavaFX LineCharts, which turned out to be extremely complicated to populate as I had to first retrieve data from the CSV file. Once I managed to populate them, I realised that an even more complicated aspect was to make them comprehendible and therefore useful.

```java
1   @Override
2       public void initialize(URL url, ResourceBundle rb) {
3           //ArrayList containing all of the relevant Data from DataReader class
4           ArrayList<MemberData> data = DataReader.membersData;
5           //Making sure that only the values for that specific player are outputted
6           int playerID = FXMLDocumentController.playerID;
7           //Initializing XYChart series for each value
8           XYChart.Series<String, Number> series = new XYChart.Series<String, Number>();
9           XYChart.Series<String, Number> seriesBrutto = new XYChart.Series<String, Number>();
10          XYChart.Series<String, Number> seriesNetto = new XYChart.Series<String, Number>();
11          XYChart.Series<String, Number> seriesTotalScore = new XYChart.Series<String, Number>();
12          XYChart.Series<String, Number> seriesPutts = new XYChart.Series<String, Number>();
13          XYChart.Series<String, Number> seriesFairways = new XYChart.Series<String, Number>();
14          XYChart.Series<String, Number> seriesGIR = new XYChart.Series<String, Number>();
15          ArrayList<MemberData> playerGameData = new ArrayList<>();
16          //Consider adding the first ever HDC
17          for (MemberData m : data) { //Memberdata is really gameData
18              //for every value (also the ones not required
19              if (m.getPlayerID() == playerID) {
20                  playerGameData.add(m);
21                  //specific values retrieved using getters and populated using .add
22                  //always retrieved with date to simplify analysis and readability
23                  //GRAPH 1: Handicap
24                  series.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getNewHDC()));
25                  //GRAPH 2: Brutto and Netto Points and total shots
26                  seriesBrutto.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getBruttoPoints()));
27                  seriesNetto.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getNettoPoints()));
28                  seriesTotalScore.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getTotalShots()));
29                  //GRAPH 3: Putts, Fairways and GIR
30                  seriesPutts.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getPutts()));
31                  seriesFairways.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getFairways()));
32                  seriesGIR.getData().add(new XYChart.Data<String, Number>(m.getDate(), m.getGIR()));
33              }
34          }
35          //Setting Names for each Value
36          series.setName("Handicap");
37          hdcDev.getData().add(series);
38          seriesBrutto.setName("Brutto");
39          snb.getData().add(seriesBrutto);
40          seriesNetto.setName("Netto");
41          snb.getData().add(seriesNetto);
42          seriesTotalScore.setName("Total Score");
43          snb.getData().add(seriesTotalScore);
44          seriesPutts.setName("Putts");
45          pfg.getData().add(seriesPutts);
46          seriesFairways.setName("Fairways");
47          pfg.getData().add(seriesFairways);
48          seriesGIR.setName("GIR");
49          pfg.getData().add(seriesGIR);
50
51      }
```

Brutto Netto Total Score

Creating comprehendible charts/results was the complex aspect of this task. By adding the dates on the x-axis, I give {Client} a clear indication of the time period the improvements took place, which directly helps him and his goals.

File Reading and Writing

Accessing and altering the CSV file that contains the information of members is necessary to fulfil Success Criteria A, B, C and D, as they all directly and indirectly rely on the data contained within the file and on being able to update/edit the file. A custom and extendable solution was required for Mr. Williams to be able to do said actions for all variables that he required in his application.

This method takes the parameters as input and inserts them to the end of the csv file (appending).

```java
public void appendToCSVFile(int gameID, int playerID, char gender, String firstName, String lastName, int totalShots,
int[] score, double oldHDC, double newHDC, int nettoPoints, int bruttoPoints, int putts, int fairways, int GIR, String date) throws IOException {
  //set path to file
 final String csvPath = "enter path to PlayerDataCSV.csv";
 Path p = Paths.get(csvPath);
 //do not follow symbolic links
 if (!Files.exists(p, LinkOption.NOFOLLOW_LINKS)) {
  Files.createFile(p);
 }
 PrintWriter pw = null;
 try {
  pw = new PrintWriter(new BufferedWriter(new FileWriter(csvPath, true)));
 } catch (FileNotFoundException e) {
  e.printStackTrace();
 }
 // use StringBuilder to append values to CSV file.
 StringBuilder sb = new StringBuilder();
 sb.append(gameID);
 sb.append(",");
 sb.append(playerID);
 sb.append(",");
 sb.append(gender);
 sb.append(",");
 sb.append(firstName);
 sb.append(",");
 sb.append(lastName);
 sb.append(",");
 sb.append(totalShots);
 sb.append(",");
 // use loop for score (because 18 holes)
 for (int i = 0; i < score.length; i++) {
  sb.append(score[i]);
  sb.append(",");
 }
 sb.append(oldHDC);
 sb.append(",");
 sb.append(newHDC);
 sb.append(",");
 sb.append(nettoPoints);
 sb.append(",");
 sb.append(bruttoPoints);
 sb.append(",");
 sb.append(putts);
 sb.append(",");
 sb.append(fairways);
 sb.append(",");
 sb.append(GIR);
 sb.append(",");
 sb.append(date);
 sb.append(System.getProperty("line.separator"));
 pw.write(sb.toString());
 pw.close();
}
```

Below, we can see the resulting layout of the CSV file, which contains all required information.

If the file does not exist, csv file is created.

Try and catch block initialized, it tries to open a PrintWriter, BufferedWrtier and FileWriter to read the contents of the file.

N.R

Building the String (line) that is appended.

Is appended and PrintWriter is closed.

I had to create an original method that would never throw an error and could easily be extended by simply copying-and-pasting existing conventions.

An ArrayList of the type MemberData is defined, this ArrayList will hold all the data from the player data .csv file. The reason it is defined as static is because it needs to be accessed directly from other classes, in other words the keyword ‘new’ does not have be used when accessing from another class. Creating this complex method resulted in me having very easy access to every single piece of data. Due to the variance in data Types (gender = Char/HDC = double/ FirstName = String/ #Putts = Integer etc.) I had to come up with a specific and original solution to make every datatype usable wherever it was needed.

```
1  public static ArrayList<MemberData> membersData;
2
3  public ArrayList<MemberData> getMembersData() {
4          return membersData;
5      }
6  public void loadCSV(String csvPath) throws IOException {
7          membersData = new ArrayList<MemberData>();
8          BufferedReader br = null;
9          String line = "";
10         String csvSplitBy = ",";
11
12         try {
13             br = new BufferedReader(new FileReader(csvPath));
14             while ((line = br.readLine()) != null) {
                   // use comma as separator
                   String[] lineValues = line.split(csvSplitBy);
                   for (int i = 0; i < lineValues.length; i++) {
                   }
                   MemberData md = new MemberData();
                   try {
                       md.setGameID(Integer.parseInt(lineValues[0]));
                       md.setPlayerID(Integer.parseInt(lineValues[1]));
                       md.setGender(lineValues[2].charAt(0));
                       md.setFirstName(lineValues[3]);
                       md.setLastName(lineValues[4]);
                       md.setTotalShots(Integer.parseInt(lineValues[5]));

                       int[] score = new int[18];
                       int size = score.length;
                       String[] memberScore = Arrays.copyOfRange(lineValues, 6, 24);
                       for (int i = 0; i < size; i++) {
                           score[i] = Integer.parseInt(memberScore[i]);
                       }
                       md.setScore(score);
                       md.setOldHDC(Double.parseDouble(lineValues[24]));
                       md.setNewHDC(Double.parseDouble(lineValues[25]));
                       md.setNettoPoints(Integer.parseInt(lineValues[26]));
38                     md.setBruttoPoints(Integer.parseInt(lineValues[27]));
39                     md.setPutts(Integer.parseInt(lineValues[28]));
40                     md.setFairways(Integer.parseInt(lineValues[29]));
41                     md.setGIR(Integer.parseInt(lineValues[30]));
42                     md.setDate((lineValues[31]));
43                 } catch (NumberFormatException e) {
44                     e.printStackTrace();
45                 }
46                 membersData.add(md);
47             }
48         } catch (FileNotFoundException e) {
49             e.printStackTrace();
50         }
51         br.close();
52     }
```

N.R

BufferedReader and FileReader are initialised for accessing the content of the .csv file.

A while loop is used to loop through every line of the .csv file.

A String array is initialised containing the data for that specific line, the split method under the String class is used to separate each value of that line for storing into the array.

An object of MemberData is created inside the while loop which will hold that particular Game’s data.

The setters from the MemberData class are used to set each value of the ‘md’ object using the data that was previously split into the String array. All primitive data type conversions are considered.

Finally, the MemberData object (‘md’) is added to the global variable ‘membersData’.

Bibliography

Graphing:

Redko, Alla. "Using Javafx Charts: Line Chart | Javafx 2 Tutorials And Documentation". Docs.oracle.com. N.p., 2014. Web. 12 Dec. 2016. http://docs.oracle.com/javafx/2/charts/line-chart.htm

"Javafx Tutorial - Javafx Linechart". Java2s.com. Web. 6 Dec. 2016. *http://www.java2s.com/Tutorials/Java/JavaFX/0820__JavaFX_LineChart.htm*

File Reading/ Writing:

Sarhan, Ashraf. "Write/Read CSV Files In Java Example". Examples Java Code Geeks. N.p., 2014. Web. 5 Dec. 2016. *https://examples.javacodegeeks.com/core-java/writeread-csv-files-in-java-example/*

"How To Read And Parse CSV File In Java". Mkyong.com. N.p., 2013. Web. 14 Dec. 2016. *https://www.mkyong.com/java/how-to-read-and-parse-csv-file-in-java/*

JavaFX in General – shout out to Bucky:

"Javafx Java GUI Design Tutorials - Youtube". YouTube. N.p., 2017. Web. Dec. 2016. https://www.youtube.com/playlist?list=PL6gx4[illegible]l9DGB[illegible]XL[illegible]VLSYVy8EbTdpGbUIG

N.R