Criterion C: Development

Complexities and Programming Methodology used in program:

1. Reading and Creating Text Files
2. Using a Sorting Algorithm (Merge Sort)
3. Data Structures (Traversing Array Lists, Arrays, and Matrices)
4. Comparable Interface
5. Error Handling
6. Encapsulation
7. Modularization
8. Inheritance and Abstraction
9. Dynamic Polymorphism

Reading and Creating Text Files:

At the start of the program, the user is prompted to initialize database with or without data from a text file. If the user chooses to initialize the program using a file, they are then prompted for a file name. After wards the database is instantiated through the loadFile method of the Personal Record Class. This is used to open the Main Menu.

```
private void loadFile(String filename) {
    String name;  // will store a Runner's name
    Scanner inFile; // Scanner object used to traverse file
    String meetName; // Holds name of meet associated with each Runner's PR
    try {
        //fileExists = false;
        System.out.println(filename);
        inFile = new Scanner(new File(filename));
        System.out.println("Found File");
        fileExists = true;
        while (inFile.hasNextLine()) {
            Time fastest;
            inFile.nextLine();
            int temp = inFile.nextInt();            // First Parameter that holds the numeric order that each runner (row) is traversed
            String formattedTime = inFile.next();   // Second Parameter that is read is PR, this is the runner's fastest time
            if(formattedTime.equals("No"))          // Accounts if the time is 0
            {
                 fastest = new Time(0,0,0);
                 inFile.next();
            }
            else
                fastest = new Time(formattedTime);
            int gradeLevel = inFile.nextInt();      // Third Parameter that gives grade level of runner;
            String level = "";
            String next;
            name = inFile.next();                   // Fourth Parameter is level, since level may be empty, the file may skip to reading name
            if (name.equals("JV") || name.equals("V") || name.equals("Frosh/Soph")) {
                level = name;
                name = "";
                name += inFile.next();
                name += " " + inFile.next();
            } else {
                name += " " + inFile.next();       // Fifth Parameter is Name of the runner
            }
            inFile.useDelimiter("\\s");
            for (int i = 0; i < 23 - name.length(); i++)    // moves file traversal point to exactly 2 spaces before next Parameter, allows
            {                                               // a delimiter to be used for next parameter
                inFile.next();
            }
            inFile.useDelimiter("\\s\\s");        // Sixth Parameter is Meet name, a delimiter is used to account for longer names.
            meetName = inFile.next();
            inFile.reset();
            Runner place;
            if (checkRunner(name, gradeLevel, level)) {
                place = getRunner(name, gradeLevel, level);      // runner not created if in database
            } else {
                place = new Runner(name, gradeLevel, level, this); // runner is created
            }
            if (checkMeet(meetName)) {
                getMeet(meetName).addCompetitor(place, fastest);  // meet not created if already in database, runner added to meet
            } else {
                Meet event = new Meet(meetName, this);            // meet is created
                event.addCompetitor(place, fastest);              // runner added to meet
            }
            name = "";
            meetName = "";
        }
        inFile.close();                                           // file closed
    } catch (Exception E) {
        System.out.println(E.getMessage() + " There was an error"); // catches error where File does not exist
        if(fileExists != true)
        fileExists = false;
    }
}
```

The code also writes to files using the writePRSheet() method in the Personal Record class. This is accessed by the “Export PR” button which exports only the PR’s of all runners in the database. File Writing is also displayed in the WriteAllData() method which will export all data including every meet, race, and runner to a text file.

```
/**
 * pre-condition : all data is ready to be exported
 * post-condition: A new file with name "NameOfFile" is created
 *                 or an existing file with the name is updated.
 *                 The file will contain PR's of all runners
 *                 from the database sorted by time.
 */
public void writePRsheet(String NameOfFile) {
    try {
        sortRunnersByTime();
        String[][] mat = getPRMatrix();
        PrintWriter writer = new PrintWriter(NameOfFile);
        String res = " #        Time    Grade     Level                Name"
                + "                  Meet Name    ";
        writer.print(res);
        String temp = "";
        for (int i = 0; i < mat.length; i++) {

            writer.println();
            writer.printf("%-5s", ("   " + mat[i][0]));
            writer.printf("%-12s", ("   " + mat[i][1]));
            writer.printf("%-10s", ("   " + mat[i][2]));
            writer.printf("%-15s", (" " + mat[i][3]));
            writer.printf("%-25s", ("   " + mat[i][4]));
            writer.printf("%-30s", ("   " + mat[i][5]));
        }

        writer.close();
    } catch (IOException e) {
        e.printStackTrace();
    }
}
```

Using a Sorting Algorithm (Merge Sort):

A Sorting algorithm is used in the Runner class in order to sort the private ArrayList<Race> myRaces. Each Race object has a private time, and name. This code chooses to sort Races by their time as this is the predominant way that runners prefer to view the meets in a season. Merge Sort splits the data into two halves, and then recursively calls the method on each half. Once the recursion is complete and all the arrays have a length of 1 or 2, the merge method is called to recreate a sorted array of the original size of myRaces. The Merge Sort Algorithm is a recursive and highly efficient algorithm with an O(n * log(n)). Since this method is called every time a race is manipulated (added, deleted, time changed), this is the most effective method of sorting through the data.

```java
// Algorithm based on generic merge sort found on geeksforgeeks.org
// Sorting method that sorts arr[beg..r] using
// merge()
private void mergeSort(Time arr[], int beg, int end) {
    if (beg < end) {
        // Find the middle point
        int m = (beg + end) / 2;

        // Sort first and second halves
        mergeSort(arr, beg, m);
        mergeSort(arr, m + 1, end);

        // Merge the sorted halves
        merge(arr, beg, m, end);
    }
}
```

```java
private void merge(Time arr[], int beg, int mid, int end) {
    // Find sizes of two subarrays to be merged
    int size1 = mid - beg + 1;
    int size2 = end - mid;

    /* Create temp arrays */
    Time first[] = new Time[size1];
    Time second[] = new Time[size2];

    /* Copy data to temp arrays */
    for (int i = 0; i < size1; ++i)
        first[i] = arr[beg + i];
    for (int j = 0; j < size2; ++j)
        second[j] = arr[mid + 1 + j];

    /* Merge the two arrays */

    // Initial indexes of first and second subarrays
    int i = 0, j = 0;

    // Initial index of merged subarry array
    int k = beg;
    while (i < size1 && j < size2) {
        if (first[i].compareTo(second[j]) <= 0) {
            arr[k] = first[i];
            i++;
        } else {
            arr[k] = second[j];
            j++;
        }
        k++;
    }

    /* Copy remaining elements of first[] if any */
    while (i < size1) {
        arr[k] = first[i];
        i++;
        k++;
    }

    /* Copy remaining elements of second[] if any */
    while (j < size2) {
        arr[k] = second[j];
        j++;
        k++;
    }
}
```

Data Structures (Array List, Array, Matrices):

- Array List: Main method of storing data in the program, Personal Record class has an Array List of type Meet and type Runner. This is traversed through numerous methods. Screenshot displays getRunner() methods

```java
/**
 * pre-condition : String name, int gradeLevel are all parameters passed in
 * post-condition: Returns the runner with matching inputs or null
 */
public Runner getRunner(String name, int gradeLevel) {
    if (allRunners.size() > 0) {
        for (int i = 0; i < allRunners.size(); i++) {
            if (allRunners.get(i).getName().equals(name) && allRunners.get(i).getGradeLevel() == gradeLevel) {
                return allRunners.get(i);
            }
        }
    }
    return null;
}

/**
 * pre-condition : String name, int gradeLevel, String level are all parameters passed in
 * post-condition: Returns the runner with matching inputs or null
 */
public Runner getRunner(String name, int gradeLevel, String level) {
    if (level == null || level.equals(""))
        return getRunner(name, gradeLevel);
    else if (allRunners.size() > 0) {
        for (int i = 0; i < allRunners.size(); i++) {
            if (allRunners.get(i).getName().equals(name) && allRunners.get(i).getGradeLevel() == gradeLevel
                    && allRunners.get(i).getTeamLevel().equals(level)) {
                return allRunners.get(i);
            }
        }
    }
    return null;
}
```

- Arrays: Used exclusively for preparing row data for matrices and Merge Sort algorithm. Screen shot displays a helper method to returning a matrix.

```java
/**
 * pre-condition : All parameters refer to corresponding runner data
 * post-condition: Returns an String[] with runner data to be used as a row in a matrix
 */
private String[] getSingleData(String number, String time, String grade,
        String level, String name,String meetName) {
    String[] data = new String[6];
    data[0] = number;
    data[1] = time;
    data[2] = grade;
    data[3] = level;
    data[4] = name;
    data[5] = meetName;
    return data;
}
```

- Matrix: used to return runner data that is formatted to be used in JTables in both the mian menu and the meet menu. JTables require a parameter of type array for the headers and a matrix for the data that is displayed in rows and columns.

```java
/**
 * pre-condition : Method is called from the MainMenu class
 * post-condition: Returns a String[][] with data that is traversable and displayed
 *                 in a JTable
 */
private String[][] getDataMatrix() {
    String grade;
    String number;
    String time;
    String level;
    String name;
    String meetName;
    int place = 0;
    String[][] data = new String[allRunners.size()][6];
    for (int i = 0; i < allRunners.size(); i++) {
        number = i + 1 + "";

        time = allRunners.get(i).getFastestTime().toString() + "";
        grade = allRunners.get(i).getGradeLevel() + "";

        name = allRunners.get(i).getName();
        if (allRunners.get(i).getTeamLevel() != null) {
            level = allRunners.get(i).getTeamLevel();
        } else
            level = "";
        if (!allRunners.get(i).getFastestTime().isEmpty())
            meetName = allRunners.get(i).getRace(allRunners.get(i).getFastestTime()).getName();
        else
            meetName = "";
        data[i] = getSingleData(number, time, grade, level, name, meetName);
        place = i;
    }
    return data;
}
```

Comparable Interface:

The Comparable Interface is used in this program in order to define a compareTo() method for the Time class. Only two forms of comparison are necessary for this program, for String and for Time objects, and the default compareTo() method for the String class is suitable. The method needed to be defined based for the Time class on the parameters of minutes, seconds, and milliseconds.

```
package InternalCode;

public class Time implements Comparable {
    private int myMin;
    private int mySec;
    private int myMillisec;
```

```
/**
 * pre-condition : An object is passed in to compare with this
 * post-condition: Returns a negative number if this < time
 *                 Returns a 0 if this = time
 *                 Returns a positive number if this > time
 */
public int compareTo(Object time) {
    Time newTime = (Time) time;
    int compare = 0;

    if (myMin < newTime.getMin()) {
        compare = 6000 * (myMin - newTime.getMin());
    } else if (myMin > newTime.getMin()) {
        compare = 6000 * (myMin - newTime.getMin());
    }
    if (mySec < newTime.getSec())
        compare += 100 * (mySec - newTime.getSec());
    else if (mySec > newTime.getSec())
    {
        compare += 100 * (  mySec - newTime.getSec());
    }
    if (myMillisec < newTime.getMillisec())
    {
        compare += myMillisec - newTime.getMillisec();
    }
    else if (myMillisec > newTime.getMillisec())
        compare +=  myMillisec - newTime.getMillisec();

    return compare;

}
```

Error Handling:

Error Handling is used throughout the file reading process. This includes the loadFile() method which accounts for if there is no data file matching the inputted filename. Additionally, error handling is used in the sortRunnersByTime() method. This method specifically handles the error of having No Time for a Runner. It is also noted that this sorting algorithm is called only one time when data is exported to a PR sheet. Because of this the sorting algorithm is of my own design to display algorithmic knowledge at the cost of the efficiency of merge sort. In the GUI interface, error handling is addressed in the main method of the InitializeProgram class and when setting the look and feel of the program.

```
/**
 * Launch the application.
 */
public static void main(String[] args) {
    EventQueue.invokeLater(new Runnable() {
        public void run() {
            try {
                InitializeProgram frame = new InitializeProgram();
                frame.setVisible(true);
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    });
}

/**
 * Create the frame with all buttons, fields, labels, and panes
 */
public InitializeProgram() {
    try
    {
        UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());

    }
    catch (Exception ex)
    {
        ex.printStackTrace();
    }
```

```
try {
    compare = toSort.get(i).getFastestTime().compareTo(temp.get(j).getFastestTime());
    if (compare > 0) {
        temp.add(j + 1, toSort.get(i));
        j = 0;

    } else if (j == 0) {
        temp.add(0, toSort.get(i));
    }
} catch (Exception e) {
    // Runner has competed in no races and has no PR
}
```

Encapsulation:

Encapsulation is used to gain access to private variables throughout the code. This is most important in the runner class and the get methods are used throughout the Personal Record class.

```
// Changes grade level
public void changeGradeLevel(int gradeLevel) {
    myGradeLevel = gradeLevel;
}
// Changes Team level
public void changeTeamLevel(String teamLevel) {
    myTeamLevel = teamLevel;
}
// Returns myName
public String getName() {
    return myName;
}
// Returns myTeamLevel
public String getTeamLevel() {
    return myTeamLevel;
}
// Returns myGradeLevel
public int getGradeLevel() {
    return myGradeLevel;
}
//Returns fastest time
public Time getFastestTime() {
    updateFastestTime();
    return myFastestTime;
}
```

An Example of how this encapsulation is utilized is shown in the sortRunnersByName() method which uses the name field as a comparison parameter for the runners in the array list.

```
/**
 * pre-condition : allRunners is an ArrayList<Runner>
 *                 method called from MainMenu
 * post-condition: allRunners is sorted by name
 */
private void sortRunnersByName() {
    ArrayList<Runner> temp = new ArrayList<Runner>();
    int compare;
    if (allRunners.size() > 1) {

        temp.add(allRunners.get(0));
        compare = allRunners.get(0).getName().compareTo(allRunners.get(1).getName());
        if (compare > 0) {
            temp.add(0, allRunners.get(1));
        } else
            temp.add(allRunners.get(1));

    }
    if (allRunners.size() > 2) {
        for (int i = 2; i < allRunners.size(); i++) {

            for (int j = i - 1; j >= 0; j--) {
                compare = allRunners.get(i).getName().compareTo(temp.get(j).getName());
                if (compare > 0) {
                    temp.add(j + 1, allRunners.get(i));
                    j = 0;

                } else if (j == 0) {
                    temp.add(0, allRunners.get(i));
                }

            }
        }
        allRunners = temp;
    }
}
```

Modularization:

The code also extensively uses modularization to simplify many complex methods. The modularization serves to prevent duplications of similar code in multiple methods that would waste memory and add unnecessary complication. Modularization makes the code easier to follow when proofing and checking for errors. The example below shows the use of modularization in the updateComboBox() method.

```java
/**
 * pre-condition : contentPane has been initialized as a JPanel
 * post-condition: Updates an existing or creates a new comboBox,
 *                 if selected is the name of an object  in the
 *                 comboBox, the corresponding object will be selected
 */
private void updateComboBox(String selected) {
    if (comboBox != null) {
        contentPane.remove(comboBox);
    }
    initializeComboBox(selected);

}
```

```java
/**
 * pre-condition : contentPane has been initialized as a JPanel
 * post-condition: Creates a new comboBox, if selected is the name of an object
 *                 in the comboBox, the corresponding object will be selected
 */
private void initializeComboBox(String selected) {

    System.out.println(PR.getNumMeets());
    ArrayList<Meet> meets = PR.getMeets();
    String[] meetNames = new String[meets.size()];
    int index = -1;
    for (int i = 0; i < meetNames.length; i++) {
        String temp = meets.get(i).getName();
        meetNames[i] = temp;
        if (selected != null && temp.equals(selected)) {
            index = i;
        }
    }

    comboBox = new JComboBox(meetNames);
    comboBox.setEditable(true);
    if (index == -1)
        comboBox.setSelectedItem("Select a Meet");
    else
        comboBox.setSelectedItem(selected);
    comboBox.setEditable(false);

    comboBox.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent arg0) {
            Object o = (comboBox.getSelectedItem());
            String meetName = ((String) o);
            if (!meetName.equals("Create New Meet")) {
                if (PR.getMeet(meetName) != null)
                    System.out.println(PR.getMeet(meetName).getName() + "  this is curent");
                current = PR.getMeet(meetName);
            }
            updateScrollPanes();
        }
    });

    contentPane.add(comboBox, "cell 1 1 2 1,growx");
    contentPane.revalidate();
    contentPane.repaint();

}
```

Inheritance and Abstraction:

Inheritance is used throughout the GUI classes. The parent class is TableFormatter which contains methods that are used in the Main Menu, Meet Menu, and Race Menu windows in order to create and format JTables. The methods inside this class mainly function to initialize the Table, format the table, create a table listener (right click option menu), and read fields (name, grade, and level).

```
public abstract class TableFormatter extends JFrame {
```

```
public class MeetMenu extends TableFormatter {
public class MainMenu extends TableFormatter {
public class RaceMenu extends TableFormatter {
```

Additionally the class has an abstract updateScrollPanes() method that is implemented in all three of these classes that defines how the JTable will be updated whenever the data is altered. The examples shown below are all 3 implementations of updateScrollPanes()

```
/**
 * pre-condition : contentPane is initialized as a JPanel
 * post-condition: Updates the data inside the runner list
 *                 , updates the JTable, updates contentPane
 */
public void updateScrollPanes() {
    runnerList = PR.getDataMatrix();
    runnerTable = initializeLog(runnerList, headers);
    updateInterfaceTab(runnerTable);
}
```

```
/**
 * pre-condition : contentPane is initialized as a JPanel
 * post-condition: Updates the data inside the alphabetic
 *                 list , updates the JTable, updates
 *                 contentPane
 */
public void updateScrollPanes() {

    if (scrollPane != null)
        updateMenus();
    raceData = myRunner.getRaceDataMatrix();
    raceTable = initializeLog(raceData, headers);
    contentPane.remove(scrollPane);
    popup = intializePopup(raceTable, popup);
    createTableListener(raceTable, popup);
    scrollPane = new JScrollPane(raceTable);
    scrollPane.setViewportBorder(new CompoundBorder());
    contentPane.add(scrollPane, "cell 1 3 2 14,grow");
    contentPane.revalidate();
    contentPane.repaint();

}
```

```
/**
 * pre-condition : contentPane is initialized as a JPanel
 * post-condition: Updates the data inside the alphabetic
 *                 list , updates the JTable, updates
 *                 contentPane
 */
public void updateScrollPanes() {
    if (current != null) {
        data = current.getDataMatrix();
        meetTable = initializeLog(data, headers);
    }
    else
    {
        data = new String[0][0];
        meetTable = initializeLog(data,headers);
    }


    if (scrollPane != null)
        contentPane.remove(scrollPane);
    popup = intializePopup(meetTable, popup);
    createTableListener(meetTable, popup);
    scrollPane = new JScrollPane(meetTable);

    contentPane.add(scrollPane, "cell 1 3 2 14,grow");
    contentPane.revalidate();
    contentPane.repaint();

    System.out.println("called");
    mainMenu.updateScrollPanes();
}
```

Dynamic Polymorphism:

Polymorphism is used to make the process of keeping data updated through multiple menus much simpler. The abstract updateScrollPanes() method was introduced in the previous section and is also an example of how the program utilizes polymorphism. The parameter *parent* in the constructor of the NewTime class is of the type TableFormatter. This window is created either from the MeetMenu or from the RaceMenu as both give the option to insert a new time for a race. Since either of these classes can be the *parent*, and both are child classes of TableFormatter, a TableFormatter object is used instead of a specific type. Since the abstract method updateScrollPanes() is defined in both classes, calling this method will update the correct menu through polymorphism.

```
/**
 * Create the frame.
 */
public NewTime(PersonalRecord database, MainMenu main, TableFormatter Parent, Race race) {

    mainMenu = main;
    PR = database;
    myParent = Parent;
    myRace = race;
```

```
// Changes time for selected runner and returns to parent window
btnDone.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        Time time = readTime();
        if(time == null || time.isEmpty())
        {
            JOptionPane.showMessageDialog(new JFrame(),
                    "At least one field (minutes, seconds, milliseconds) must have a non-zero value"
                    + " and seconds must be below 60", "Error",
                    JOptionPane.WARNING_MESSAGE);
        }
        else
        {
        myRace.changeTime(time);
        mainMenu.updateLog("Time is changed");
        myParent.updateScrollPanes();
        myParent.setVisible(true);
        dispose();
    }
    }
});
contentPane.add(btnDone, "cell 5 6,growx");
```

Word Count: 978 words