Informatics Institute of Technology
Department of Computing

Bsc (Hons) Artificial Intelligence and Data Science

Module: CM1601 Programming Fundamentals

Coursework Report

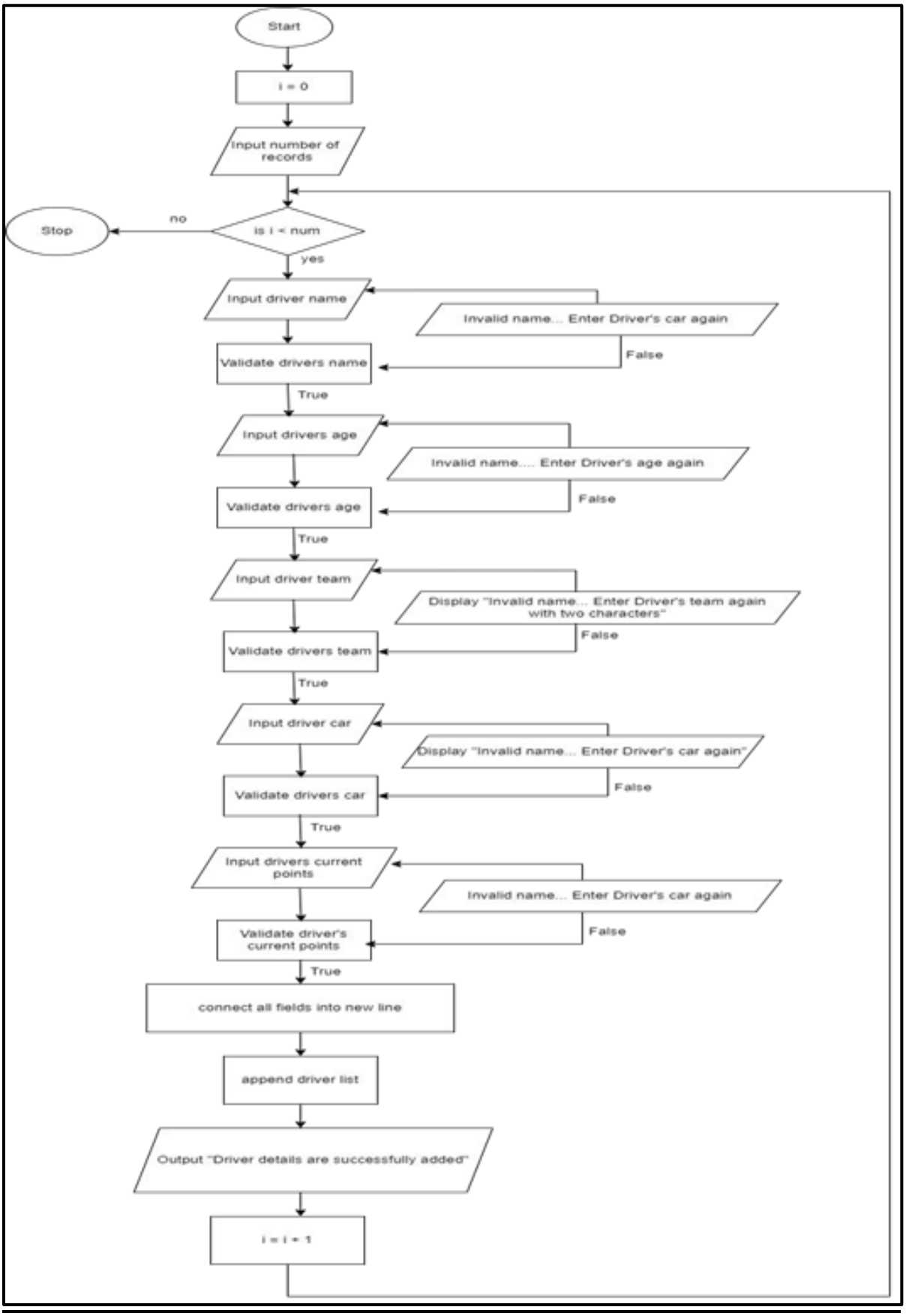
Runisi Nikoya Samaranayake – 2237028

EXECUTIVE SUMMARY

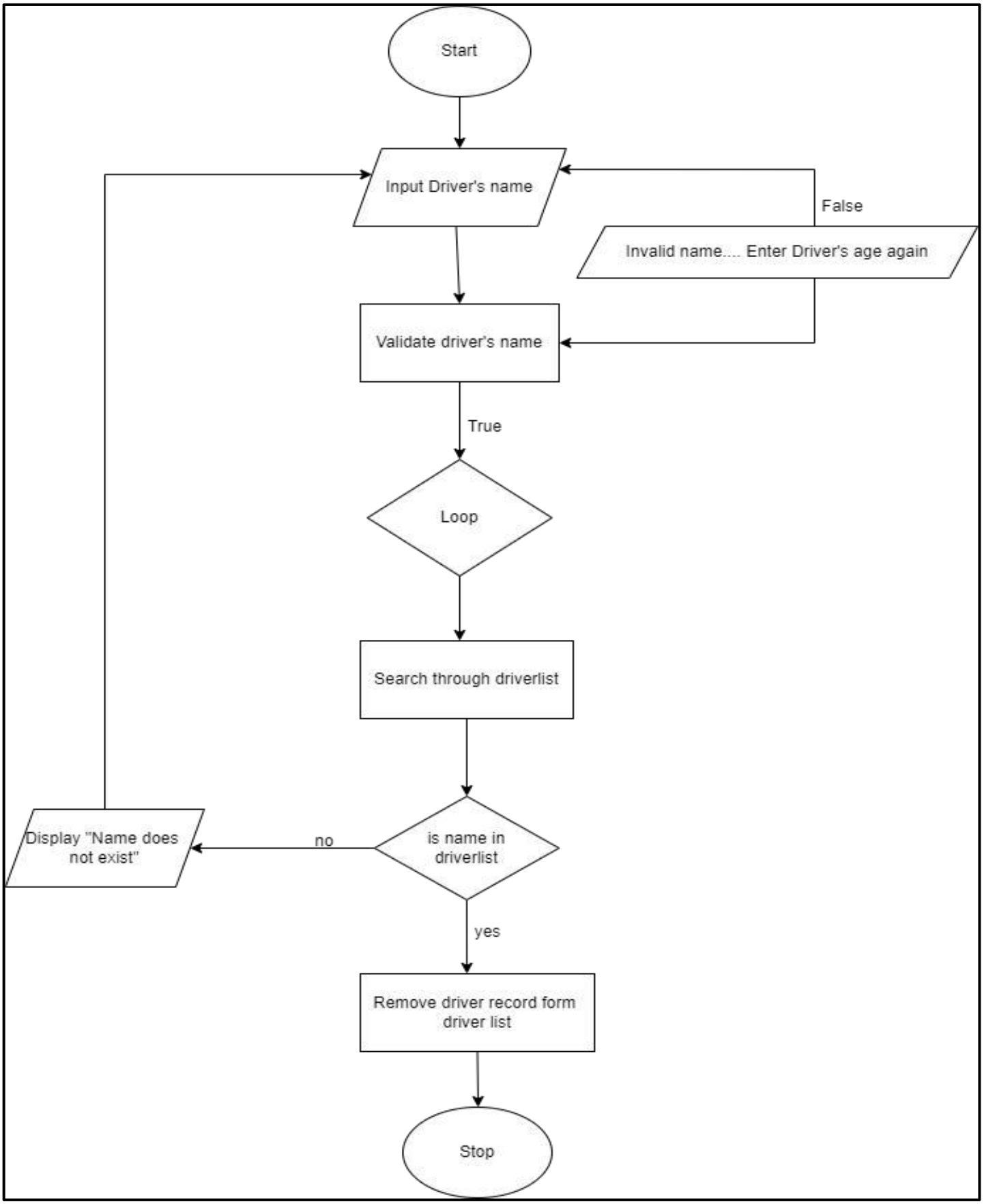
1. First user will allow to enter the following details by prompting the required information such as driver details (Name, age, current points. Team, car) And each field is validated accordingly. And in the Gui interface we can save the drivers by using the save button or you can cancel using the cancel button.
2. User should be able to search the name and user should check the name tered is there in the text fie if it's not driver not found.
3. In the gui interface in the drivers table user can search the name and the name will be deleted and to update the drivers details user can search the name and update the driver details and click save to save the updated details.
4. Then the Standings table with the drivers details and their points sorted in descending order and then display them in a table.
5. Generate a random race and then a position is assigned to each driver and assign points according to their positions and then the points will be updated in the standings table
6. Then the race table with all the races with date and location where the date should be arranged in ascending order
7. The system is able to save the current data to a text file and load the data as well.

CONTENTS

FLOWCHART - FUNCTION ADD



FLOWCHART - FUNCTION DDD



INTRODUCTION TO FUNCTIONS WITH CODE

FUNCTION ADD

NewDriverController

```
package com.Controllers;

import com.Models.Driver;
import com.Utilities.Constants;
import com.Utilities.DriversManager;
import com.Utilities.Messages;
import com.Utilities.Resources;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.TextField;
import javafx.scene.control.TextFormatter;
import javafx.stage.Stage;

import java.io.IOException;

public class NewDriverController {
   public Button cancelButton;
   @FXML
   private TextField nameField;
   @FXML
   private TextField ageField;
   @FXML
   private TextField teamField;
   @FXML
   private TextField carField;
   @FXML
   private Button saveButton;

   private DriversController;
   private Driver selectedDriver;
   private boolean isUpdate;

   public void initialize() {
      TextFormatter<String> numericOnlyFormatter = new TextFormatter<>(change ->
change.getControlNewText().matches("\\d*") ? change : null);
      ageField.setTextFormatter(numericOnlyFormatter);
   }
```

```
@FXML
public void onSaveButtonClicked(ActionEvent actionEvent) throws IOException {
    String name = nameField.getText();
    int age = 0;
    try { age = Integer.parseInt(ageField.getText()); } catch (Exception e) { e.printStackTrace(); }
    String team = teamField.getText();
    String car = carField.getText();

    if (name.equals("")) {
        Messages.error("Invalid Name", "Please enter a valid name!");
        return;
    }

    if (team.equals("")) {
        Messages.error("Invalid Team", "Please enter a valid team name!");
        return;
    }

    if (car.equals("")) {
        Messages.error("Invalid Car", "Please enter a valid car name!");
        return;
    }

    if (age < 20) {
        Messages.error("Invalid Age", "Please enter an age greater than or equal to 20!");
        return;
    }

    Driver newDriver = new Driver(name, age, team, car, isUpdate ? selectedDriver.getPoints() : 0);
    if (isUpdate) DriversManager.updateDriver(selectedDriver, newDriver);
    else DriversManager.addDriver(newDriver);
    Messages.information("Success", "Driver saved successfully!");

    nameField.setText("");
    ageField.setText("");
    teamField.setText("");
    carField.setText("");
    onCancel(actionEvent);
}

public void setDriver(Driver driver) {
    nameField.setText(driver.getName());
    ageField.setText(driver.getAge() + "");
    teamField.setText(driver.getTeam());
    carField.setText(driver.getCar());
    this.isUpdate = true;
    this.selectedDriver = driver;
}

public void onCancel(ActionEvent actionEvent) {
    Stage = (Stage) cancelButton.getScene().getWindow();
    FXMLLoader loader = Resources.getScreen("home.fxml");
```

```
        Scene = null;
        try {
            scene = new Scene(loader.load());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        HomeController = loader.getController();
        stage.setScene(scene);
        stage.show();
    }
}
```

This is a code snippet for a Java application that allows users to add driver details. It prompts the user to enter driver information such as name, age, team, car, and current points.

The code uses JavaFX to create a GUI with several text fields for the user to input data. The onSaveButtonClicked method is called when the user clicks the "Save" button, and it validates the user input to make sure the required fields are not empty and the age is greater than or equal to 20.

If the input is valid, the driver details are stored in a Driver object and either added to the list of drivers or updated if the user is editing an existing driver. A success message is then displayed, and the text fields are cleared.

The setDriver method is used to populate the text fields with the details of an existing driver if the user is editing a driver, while the onCancel method is called when the user clicks the "Cancel" button to return to the home screen.

Encapsulation: To prevent direct access from outside the class, the nameField, ageField, and teamField fields of the NewDriverController class are designated as private. The NewDriverController object is instead accessible to other classes via public methods (like setDriver and onSaveButtonClicked).

Inheritance: The NewDriverController class inherits properties from the Driver class.

Polymorphism: The initialize method, a callback method that is called by the JavaFX framework when the FXML file is loaded, is implemented by the NewDriverController class. When a user clicks the corresponding buttons in the GUI, callback methods called onSaveButtonClicked and onCancel are also triggered.

Abstraction -: Managing driver objects in the GUI is an abstraction. The specifics of how drivers are added, updated, or removed from the user interface

Function ADD,DDD,UDD

DriversController

```
package com.Controllers;

import com.Models.Driver;
import com.Utilities.Constants;
```

```
import com.Utilities.DriversManager;
import com.Utilities.Messages;
import com.Utilities.Resources;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.fxml.FXMLLoader;
import javafx.geometry.Pos;
import javafx.scene.Node;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.stage.Stage;
import java.io.IOException;

public class DriversController {
    public Button newDriverBtn;
    private ObservableList<Driver> drivers = FXCollections.observableArrayList();
    @FXML
    public Button searchBtn;
    @FXML
    public TextField searchField;
    @FXML
    private TableView<Driver> driversTable;

    public void loadTable() {
        driversTable.setItems(DriversManager.retrieveAllDrivers());
    }

    public void initialize() {
        driversTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
        loadTable();

        TableColumn<Driver, Void> deleteColumn = new TableColumn<>("Delete");
        deleteColumn.setCellFactory(param -> new TableCell<Driver, Void>() {
            private final Button deleteButton = new Button("Delete");

            {
                deleteButton.setOnAction(event -> {
                    Driver = getTableView().getItems().get(getIndex());
                    boolean result = Messages.confirmation("Confirm Deletion",
                            "Are you sure you want to delete this item?");
                    if (result) {
                        DriversManager.removeDriver(driver);
                        loadTable();
                    }
                });
                deleteButton.prefWidthProperty().bind(deleteColumn.widthProperty());
                deleteButton.setAlignment(Pos.CENTER);
            }

            @Override
```

```
    protected void updateItem(Void item, boolean empty) {
        super.updateItem(item, empty);

        if (empty) {
            setGraphic(null);
        } else {
            setGraphic(deleteButton);
        }
    }
});
deleteColumn.setStyle("-fx-alignment: CENTER;");

TableColumn<Driver, Void> updateColumn = new TableColumn<>("Update");
updateColumn.setCellFactory(param -> new TableCell<Driver, Void>() {
    private final Button updateButton = new Button("Update");

    {
        updateButton.setOnAction(event -> {
            Driver = getTableView().getItems().get(getIndex());
            // create a new screen and pass the selected race object to its constructor
            Stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
            FXMLLoader loader = Resources.getScreen("new-driver.fxml");
            Scene = null;
            try {
                scene = new Scene(loader.load());
            } catch (IOException e) {
                throw new RuntimeException(e);
            }
            NewDriverController controller = loader.getController();
            controller.setDriver(driver);
            stage.setScene(scene);
            stage.show();
        });
        updateButton.prefWidthProperty().bind(updateColumn.widthProperty());
        updateButton.setAlignment(Pos.CENTER);
    }

    @Override
    protected void updateItem(Void item, boolean empty) {
        super.updateItem(item, empty);

        if (empty) {
            setGraphic(null);
        } else {
            setGraphic(updateButton);
        }
    }
});
updateColumn.setStyle("-fx-alignment: CENTER;");

// add the delete and update columns to the table view
driversTable.getColumns().addAll(updateColumn, deleteColumn);
```

```
    }

    public void onSearch(ActionEvent actionEvent) {
        String searchText = searchField.getText().toLowerCase();
        if (searchText.isEmpty()) driversTable.setItems(DriversManager.retrieveAllDrivers());
        else driversTable.setItems(DriversManager.searchDrivers(searchText));
    }

    public void onDelete(ActionEvent actionEvent) {
    }

    public void onUpdate(ActionEvent actionEvent) {
    }

    public void onNewDriverClick(ActionEvent actionEvent) {
        // create a new screen and pass the selected race object to its constructor
        Stage = (Stage) ((Node) actionEvent.getSource()).getScene().getWindow();
        FXMLLoader loader = Resources.getScreen("new-driver.fxml");
        Scene = null;
        try {
            scene = new Scene(loader.load());
        } catch (IOException e) {
            throw new RuntimeException(e);
        }
        NewDriverController controller = loader.getController();
//        controller.setDriver();
        stage.setScene(scene);
        stage.show();
    }

    public void onRefresh(ActionEvent actionEvent) {
        searchField.setText("");
        driversTable.setItems(DriversManager.retrieveAllDrivers());
    }
}
```

This DriversController class has the functionality of adding, deleting and updating the driver details. This class is responsible for handling the UI and user interaction with the table of drivers.

This code is for a program that keeps track of data on race car drivers. Users of the system can add, update, and delete driver details using its user interface. A table listing a list of drivers with columns for each driver's name, age, team, vehicle, and current points is part of the user interface.

The user interface behavior is managed by a JavaFX controller class included in the code. Many methods in the controller class react to user activities like clicking a button or entering text into a search field.

When the user interface is initialized, the loadTable() method is invoked, and it fills the driver table with information obtained from a class named DriversManager. The list of drivers is managed by the DriversManager class.

The driver table's custom "Delete" and "Update" columns are added as part of the initialize() method's setup of the table's columns. The user can change or delete a driver record from the database by using the buttons in these custom columns. When a user selects the "Delete" button, a confirmation box appears. If the user clicks "Yes," the driver is deleted from the table and the DriversManager class is alerted to do so. A new window that enables editing of the selected driver's details is opened when the user hits the "Update" button.

The driver database is updated by the onSearch() method in response to user input in the search field, and only the drivers whose names match the search text are displayed. The search field is cleared and the entire driver database is reloaded by the onRefresh() method.

Adding a new driver to the system is made possible by a new window that is opened when the onNewDriverClick() method is called. The driver's name, age, team, car, and current points are all listed in the fields of this window. Data is entered into the driver table and the DriversManager class is informed to add a new driver object when the user submits fresh driver information.

Classes – This code uses classes such as Driver, DriversManager, and Resources, to manage the drivers' information.

Inheritance: The NewDriverController class inherits properties from the Driver class.

Encapsulation: Logic for handling and managing drivers is encapsulated in the class DriversController.

Polymorphism: To alter the size of the table, the DriversController class overrides the initialize() method of the Controller class.

Abstraction: The code makes use of abstraction by defining a class called Driver, which hides the specifics of a driver's information and offers a straightforward interface for accessing and changing it.

FUNCTION VCT

StandingsController

```
package com.Controllers;

import com.Models.Driver;
import com.Utilities.DriversManager;
import javafx.collections.FXCollections;
import javafx.collections.ObservableList;
import javafx.event.ActionEvent;
import javafx.fxml.FXML;
import javafx.scene.control.*;

public class StandingsController {
    private ObservableList<Driver> drivers = FXCollections.observableArrayList();
    @FXML
    public Button searchBtn;
    @FXML
    public TextField searchField;
    @FXML
    private TableView<Driver> driversTable;
```

```
    public void initialize() {
        driversTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
        driversTable.setItems(DriversManager.retrieveAllStandings());
    }

    public void onSearch(ActionEvent actionEvent) {
        String searchText = searchField.getText().toLowerCase();
        if (searchText.isEmpty())
driversTable.setItems(DriversManager.retrieveAllStandings());
        else driversTable.setItems(DriversManager.searchStandings(searchText));
    }

    public void onRefresh(ActionEvent actionEvent) {
        searchField.setText("");
        driversTable.setItems(DriversManager.retrieveAllStandings());
    }
}
```

The championship standings of a car racing competition are shown using the JavaFX controller in this code. The list of drivers who can be seen has the championship standings kept in it. In a table layout, the drivers are shown according to decreasing point totals.

Three methods—initialize(), onSearch(), and onRefresh—are available in the controller class (). The controller is initialized after using the initialize() method. The items of the driversTable are set to the standings obtained from the DriversManager class, and the column resize policy of the driversTable is set.

In this code, the JavaFX controller is used to display the championship standings for a race car competition. The championship rankings are maintained in the list of drivers who are visible. The drivers are listed in descending order of point totals in a table format.

There are three methods in the controller class: initialize(), onSearch(), and onRefresh (). The initialize() method is used, and then the controller is initialized. The driversTable's column resize policy is established, and the items are set to the rankings obtained from the DriversManager class.

Encapsulation: The StandingsController class is the only place where the private drivers variable can be accessed.

Abstraction: To work with drivers' standings, the StandingsController class offers a streamlined interface. The DriversManager class abstracts away the implementation specifics of how the data is retrieved and displayed.

Inheritance- By inheriting the properties and methods of the javafx.scene.control.TableView class, the StandingsController class extends it.

Polymorphism: The driversTable variable is declared as a TableView, but because of polymorphism, it can hold any kind of Driver object.

FUNCTION VRL

RaceController Class

```
package com.Controllers;

import com.Models.Race;
import com.Models.Result;
import com.Utilities.Constants;
import com.Utilities.Resources;
import com.Utilities.ResultsManager;
import javafx.event.ActionEvent;
import javafx.fxml.FXMLLoader;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TableView;
import javafx.stage.Stage;

import java.io.IOException;
import java.text.SimpleDateFormat;

public class RaceController {
    public Button backButton; //This is the button back to go to the race history table
    public TableView<Result> resultsTable;
    public Label title;

    public void setRace(Race selectedRace) { //race object as parameter
        SimpleDateFormat sdf = new SimpleDateFormat(" (E, dd MMM yyyy HH:mm:ss z)");
        // use it to format the date of the selectedRace object
        String formattedDate = sdf.format(selectedRace.getDate()); //formatted date
stored as string
        String screenTitle = selectedRace.getName() + " - " + selectedRace.getLocation()
+ formattedDate;
        //name, location, and formatted date of the selectedRace object
        title.setText(screenTitle);
        resultsTable.setItems(ResultsManager.retrieveAllResults(selectedRace));
    }

    public void initialize() {

        resultsTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
        //columns will automatically resize to fit the available space.
    }

    public void onBack(ActionEvent actionEvent) {
        Stage = (Stage) backButton.getScene().getWindow(); //gets the Stage object
associated
        // with the backButton Button.
        FXMLLoader loader = Resources.getScreen("home.fxml"); //load the user interface
defined in the home.fxml file.
        Scene = null;
        try {
            scene = new Scene(loader.load()); // creates a new Scene object from the FXML
file that was loaded previously
        } catch (IOException e) {
            throw new RuntimeException(e);
```

```
        }
        HomeController = loader.getController();
        homeController.tabPane.getSelectionModel().select(3);
        //select the fourth tab in the home page which is to display the races
        stage.setScene(scene);
        stage.show();
    }
}
```

Sets the text of the title label to a string consisting of the name, location, and date of the selectedRace object, formatted using a SimpleDateFormat. This function takes a Race object as an argument. Furthermore, using ResultsManager, it sets the entries of the resultsTable to the outcomes of the chosen race. retrieveAllResults(selectedRace).

initialize() sets TableView as the resultsTable's column resizing policy.CONSTRAINED RESIZE POLICY, meaning that columns will automatically resize to meet the available space.

When the backButton is clicked, the method onBack(ActionEvent actionEvent) is triggered. It obtains the Stage object linked to the backButton button, loads the home.fxml FXML file using Resources.getScreen("home.fxml"), creates a new Scene object from the loaded FXML file, selects the fourth tab in the home page's tabPane, sets the Stage object's Scene to the newly created Scene, and displays the Stage.

Objects - This code makes use of a number of objects to represent the user's actual screens.Race, result, and resource-related objects

Encapsulation: The class makes use of encapsulation to shield other program elements from the implementation specifics of its properties and methods. For instance, the setRace method uses a Race object as a parameter to set the user interface screen's title and table data.

Polymorphism: The initialize method makes use of polymorphism to set the resultsTable object's column resize policy to TableView.CONSTRAINED_RESIZE_POLICY.This works because setColumnResizePolicy is a method in TableView that takes an argument of type TableView, and resultsTable is of type TableViewResult>.ResizePolicy.

Abstraction: The RaceController class abstracts the specifics of how to display a race's details and results from implementation. To display the specifics and results, the caller of this class only needs to call the setRace() method with a Race object.

RaceHistory controller

```
public class RaceHistoryController {
    public TableView<Race> racesTable;  //display the table
    public TextField searchField;
    public Button searchBtn; //search button to search for specific races
    private final ObservableList<Race> races = FXCollections.observableArrayList();

    public void initialize() {
        racesTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY); //columns
```

```
will be resized to fill the available space.
        racesTable.setItems(RaceManager.retrieveAllRaces()); //race objects received from
a tables class

        racesTable.setOnMouseClicked(event -> { //it is an event listener to click the
button so u can retrieve more details of the race
            if (event.getClickCount() == 2) { // so u can retrieve more details of the
race
                Race = racesTable.getSelectionModel().getSelectedItem();
                goToRaceViewScreen(null, event, race);
            }
        });

        TableColumn<Race, Void> seeMoreColumn = new TableColumn<>("");
        seeMoreColumn.setCellFactory(param -> new TableCell<Race, Void>() {
            //seeMoreColumn is colum which refers to another
            private final Button seeMoreButton = new Button("See more..."); //this is the
button for see more

            {
                seeMoreButton.setOnAction(event -> {
                    Race = getTableView().getItems().get(getIndex());
                    goToRaceViewScreen(event, null, race);
                });
                seeMoreButton.prefWidthProperty().bind(seeMoreColumn.widthProperty());
                seeMoreButton.setAlignment(Pos.CENTER);
            }
```

The table of races will be displayed by a TableView widget, which is what the racesTable variable refers to. A text field and a button that can be used to search for specific races are identified by the variables searchField and searchBtn.

When the screen first loads, the initialize() method is invoked, which configures the racesTable with the following settings:

The table's resizing policy is set to TableView using the setColumnResizePolicy() method.CONSTRAINED RESIZE POLICY designates that columns will be proportionally enlarged to fill the available space.

The table's data source is changed via the setItems() function to an ObservableList of Race objects fetched from a RaceManager class.

A listener that will be activated whenever a mouse click happens is attached to the table using the setOnMouseClicked() method. If there are two clicks, or a double-click, the procedure retrieves the chosen Race object and switches to a more in-depth view of the race.

The TableColumn object, which stands for the extra column in the table, is the object to which the seeMoreColumn variable refers. The setCellFactory() function establishes a factory that produces TableCell objects specifically for this column. The TableCell objects produced by this factory include a Button that, when clicked, navigates to a more thorough view of the relevant race.

The table uses the updateItem() method to modify each cell's contents in the additional column. The method makes the cell's graphic null if the cell is empty. Unless something unusual happens, it sets the Button produced in the TableCell

object's constructor as the cell's graphic.

```
@Override
        protected void updateItem(Void item, boolean empty) { //update the content of
each cell in the additional column
            super.updateItem(item, empty);

            if (empty) {
                setGraphic(null);  //if the cell is empty it will set to null
            } else {
                setGraphic(seeMoreButton);
            }
        }
    });
    seeMoreColumn.setStyle("-fx-alignment: CENTER;");
    racesTable.getColumns().addAll(seeMoreColumn);
}

public void goToRaceViewScreen(ActionEvent event, MouseEvent mEvent, Race race) {
    Stage;
    if (event != null) {
        stage = (Stage) ((Node) event.getSource()).getScene().getWindow();
    } else {
        stage = (Stage) ((Node) mEvent.getSource()).getScene().getWindow();
    }

    FXMLLoader loader = Resources.getScreen("race.fxml");
    Scene = null;
    try {
        scene = new Scene(loader.load());
    } catch (IOException e) {
        throw new RuntimeException(e);
    }
    RaceController controller = loader.getController();
    controller.setRace(race);
    stage.setScene(scene);
    stage.show();
}
```

A nested class that extends TableCell contains the updateItem method definition. The extra column that has been added to the racesTable's additional column now has its contents updated using this approach. It accepts item and empty as parameters and overrides the updateItem function of the TableCell class. When the empty argument is true, the procedure changes the cell's graphic to zero. Instead, it sets the graphic to the seeMoreButton button, which enables the user to view further race-related information.

The goToRaceViewScreen method is used to navigate to the screen where additional racing information can be seen. It requires the three arguments race, mEvent, and event. If the event parameter contains a value other than null, the function retrieves the Stage object related to the event. In the absence of that, it receives the Stage object connected to the mEvent. The race.fxml file is then loaded using an FXMLLoader object, and a new Scene object is created by loading the FXML file for the race view screen. Finally, the RaceController object connected to the scene is retrieved. Last but not least, it prepares the Race object to appear on the screen, creates the stage's backdrop, and displays the stage.

The seeMoreColumn variable, which represents the extra column in the racesTable where the seeMoreButton is presented,

is an instance of the TableColumn class. It centers the alignment of each cell in the column. The getColumns() method of the TableView class is used to add the seeMoreColumn to the racesTable.

```
public void onSearch(ActionEvent actionEvent) { //this method is called when the user
clicks the search
        String searchText = searchField.getText().toLowerCase(); //It gets the text from
a search field and converts it to lowercase.
        if (searchText.isEmpty()) racesTable.setItems(RaceManager.retrieveAllRaces());
        //If the search text is empty, it sets the table items to all races using this
method
        else racesTable.setItems(RaceManager.searchRaces(searchText)); // sets the table
items to a
        // filtered list of races that match the search tex
    }

    public void refreshTable() { //method is called to refresh the table data.
        searchField.setText("");  //clears the search field by setting its text to an
empty string.
        racesTable.setItems(RaceManager.retrieveAllRaces()); //sets the table items to
all races
    }

    public void onRefresh(ActionEvent actionEvent) { //this method is called when user
clicks the refresh button

        refreshTable(); //refreshes the table
    }
}
```

OnSearch is the first technique (ActionEvent actionEvent). The user's click on the search button invokes this procedure. It retrieves the text that the user supplied in a search field and lowercases it. The RaceManager arranges the entries in a table to display all the races if the search term is empty. the retrieveAllRaces() function. If not, it uses the RaceManager to filter a list of races that match the search term before setting the table's items to those races. using the searchRaces(searchText) function.

RefreshTable is the second technique (). To update the table data, this procedure is used. By setting the search field's text to an empty string, it clears it, and it changes the table's elements to show all races using the RaceManager. the retrieveAllRaces() function.

the onRefresh technique is the third (ActionEvent actionEvent). When the user presses the refresh button, this function is invoked. To update the table data, it merely invokes the refreshTable() method.

Classes and Objects -: This code uses the concepts of classes such as Race,Race manager and resources and uses the concept of objects such as raceTable,searchBtn etc..

Encapsulation – In order to restrict access to the class's data and features, the code makes use of both public methods and private instance variables and uses access modifiers such as getters and setters.

Polymorphism - This code makes use of polymorphism, specifically method overriding. The TableCell subclass's updateItem method is overridden in order to alter how the cells in the extra column are displayed.

The Race class, which represents a race and has properties like race name, distance, date, etc., is an illustration of abstraction. It also has methods for retrieving and changing these properties, as well as ways to compute specific values like the race's average pace. The RaceHistoryController class doesn't need to be aware of how the races are actually stored because the RaceManager class abstracts the storage and retrieval of Race objects.

FUNCTION SRR

RandomRaceController

```
package com.Controllers;

import com.Models.*;
import com.Utilities.*;
import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.application.Platform;
import javafx.event.ActionEvent;
import javafx.event.EventHandler;
import javafx.fxml.FXML;
import javafx.scene.control.*;
import javafx.scene.control.cell.CheckBoxTableCell;
import javafx.scene.control.cell.PropertyValueFactory;
import javafx.scene.image.ImageView;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.text.TextAlignment;
import javafx.util.Duration;
import java.util.Date;

public class RandomRaceController {
    public TextField raceNameField; //text field for entering the race name.
    public TextField raceLocationField;  //text field for entering the race location.
    public Button startBtn; //button to start the race
    public TableView<Result> resultsTable; //display the results after the race
    public Label resultsLabel; //table view to represent the results
    public Pane randomRacePane;// pane containing race components
    public ImageView trackImageView; //represents the image view to display the race
track.
    public TableView driversTable; //represents the table view to display the list of
drivers.
    public Button finishBtn; //button to finish the race
    public Label countLabel; //represents the label to display the countdown.

    private LinkedList<ImageView> carImageViews;//private LinkedList of ImageView
instances
    // named carImageViews that represents the list of car image views.
    private LinkedList<ImageView> treeImageViews; //represents the list of tree image
views.
    private ImageView finishLine;//represents the image view for the finish line.

    private LinkedList<Driver> selectedDrivers; //represents the list of selected
drivers.
    private TableColumn<Driver, Boolean> checkboxColumn; // table column for the driver
    // selection checkbox.
```

```
    @FXML
    public void initialize() {
        resultsTable.setVisible(false);
        resultsLabel.setVisible(false);
        resultsTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
//columns resize to fill the available space.
        trackImageView.setImage(Constants.Images.TRACK_2);
        trackImageView.setPreserveRatio(false);
        trackImageView.setVisible(false);
        finishBtn.setVisible(false);
        selectedDrivers = new LinkedList<Driver>(); // hold instances of Driver.
        countLabel.setTextFill(Color.RED); //set the text fill, visibility, text
alignment,
        // and text of countLabel.
        countLabel.setVisible(false);
        countLabel.setTextAlignment(TextAlignment.CENTER);
        countLabel.setText("");

        driversTable.setItems(DriversManager.retrieveAllDrivers());
        //ets the items of driversTable to a list of drivers retrieved using the
        // retrieveAllDrivers() method of the DriversManager class.
        driversTable.setColumnResizePolicy(TableView.CONSTRAINED_RESIZE_POLICY);
        checkboxColumn = new TableColumn<>();
        checkboxColumn.setCellFactory(CheckBoxTableCell.forTableColumn(checkboxColumn));
//checkbox column
        checkboxColumn.setText("Select");
        checkboxColumn.setCellValueFactory(new PropertyValueFactory<>("selected"));
        checkboxColumn.setCellFactory(column -> new TableCell<Driver, Boolean>() {
            private final CheckBox = new CheckBox();
            {
                checkBox.setOnAction(event -> {
                    if (selectedDrivers.size() >= Constants.MAX_DRIVERS_PER_RACE &&
checkBox.isSelected()) {
                        checkBox.setSelected(false);
                        Messages.warning("Maximum limit reached", "You can select only up
to " + Constants.MAX_DRIVERS_PER_RACE + " drivers!");
                        return;
                    }
                    Driver = getTableView().getItems().get(getIndex());
                    driver.setSelected(checkBox.isSelected());
                    if (checkBox.isSelected()) selectedDrivers.add(driver);
                    else selectedDrivers.remove(driver);
                });
            }

            @Override
            protected void updateItem(Boolean item, boolean empty) {
                super.updateItem(item, empty);

                if (empty) {
                    setGraphic(null);
                } else {
                    checkBox.setSelected(item);
                    setGraphic(checkBox);
                }
            }
        });
        driversTable.getColumns().add(checkboxColumn);

    }
```

```
    public void onStart(ActionEvent actionEvent) {
        String btnText = startBtn.getText();
        if (btnText.equals("Start")) {
            String name = raceNameField.getText();//get the text in the raceNameField and
store
            // in name variable
            String location = raceLocationField.getText(); //gets the text of the
            // raceLocationField and store in name variable

            if (name.equals("")) { //checks if name is empty
                Messages.error("Invalid Name", "Please enter a valid name!");
                return;
            }

            if (location.equals("")) { // checks if location is empty
                Messages.error("Invalid Location", "Please enter a valid location!");
                return;
            }

            if (selectedDrivers.size() < 2) {//checks amount of drivers
                Messages.error("Insufficient Drivers", "Please select at least two
drivers!");
                return;
            }

            Race newRace = new Race(name, location, new Date(),
DriverSerialization.load());
            //create a new Race object using the name, location, the current date, and a
            // list of Driver objects loaded using the DriverSerialization class
            LinkedList<Result> results = newRace.simulateRace(selectedDrivers);
            //method is called on the newRace object, passing in the selectedDrivers
linked list.
            LinkedList<Car> cars = newRace.setCars(results);
            //his method returns a linked list of Result objects, which are then used to
            // create a linked list of Car objects using the setCars method on the
newRace object.
            displayRace(newRace, cars, results);
            startBtn.setText("Processing...");
            driversTable.setVisible(false); //hides the drivers table
        } else if (btnText.equals("Reset")) {
            startBtn.setText("Start");
            resultsTable.setVisible(false);
            resultsLabel.setVisible(false);
            raceNameField.setText("");
            raceLocationField.setText("");
            trackImageView.setVisible(false);
            driversTable.setVisible(true);
            DriversManager.resetSelectedStatus();
            driversTable.setItems(DriversManager.retrieveAllDrivers());
            driversTable.refresh();
        }
    }

    public void displayRace(Race race, LinkedList<Car> cars, LinkedList<Result> results)
{
        finishLine = new ImageView(Constants.Images.FINISH_LINE);
        finishLine.setFitWidth(850);
        finishLine.setFitHeight(40);
        finishLine.setLayoutX(210);
        finishLine.setLayoutY(80);
```

```
        finishLine.setVisible(false);
        carImageViews = new LinkedList<ImageView>(); //the car objects
        trackImageView.setVisible(true);
        finishBtn.setOnAction(e -> {  //action button when the button is clicked race
ends
            onRaceEnd(race, results);
            race.finish();
        });

        randomRacePane.getChildren().add(finishLine);

        int i = 0;
        double spacing = cars.size() == 5 ? 140 : cars.size() == 2 ? 320 : cars.size() ==
3 ? 230 : 180;
        //space between cars depending on number
        double leftSpace = cars.size() == 5 ? 340 : cars.size() == 2 ? 450 : cars.size()
== 3 ? 395 : 365;
        //space from the car to the side track depending on number of cars.
        for(Car : cars) {
            ImageView carImageView = car.getImage();
            carImageView.setLayoutX(leftSpace + spacing * i);
            carImageView.setLayoutY(700);
            randomRacePane.getChildren().add(carImageView);
            carImageViews.add(carImageView);
            i++;
        }

        i = 1;
        for(SideObject : race.getSideObjects()) { //sets the position of the trees along
            //the side track
            ImageView treeImageView = sideObject.getImage();
            if (i % 2 == 1) treeImageView.setLayoutX(20); //sets x coordinate of image to
20
            else treeImageView.setLayoutX(1150); //or else sets to 1150
            treeImageView.setLayoutY(sideObject.getPosition());
            //Sets the Y coordinate of the image to the position property of the
SideObject.
            randomRacePane.getChildren().add(treeImageView);
            //Adds the treeImageView to the randomRacePane container.
            carImageViews.add(treeImageView);//Adds the treeImageView to the
carImageViews list.
            i++;
        }

        finishBtn.toFront();

        Runnable onRaceFinish = () -> { //defines what should happen when the race
finishes.
            Platform.runLater(() -> {
                onRaceEnd(race, results);
            });
        };
        countLabel.setVisible(true); //makes count label visible
        startRace(race, onRaceFinish, cars); //calls startRace method passing race object
    }

    public void onRaceEnd(Race race, LinkedList<Result> results) {
        RaceManager.addRace(race, results); //dds the current Race object and its Result
objects to the RaceManager.
        resultsTable.setVisible(true); //in this the results table will be visible
        resultsLabel.setVisible(true);
```

```
        trackImageView.setVisible(false); //And the racetrack will not be visible
        resultsTable.setItems(ResultsManager.transformToObservableList(results));
        //This line sets the items of the resultsTable to an observable list generated
        // by the ResultsManager.
        if (carImageViews != null) for (ImageView carImageView : carImageViews)
carImageView.setVisible(false);
        startBtn.setText("Reset");
        finishBtn.setVisible(false);
        if (finishBtn != null) finishLine.setVisible(false); //finish button will not be
visible
        selectedDrivers = new LinkedList<Driver>();
    }

    public void startRace(Race race, Runnable onRaceFinish, LinkedList<Car> cars) {
        countLabel.toFront(); //makes the countlabel visible
        Timeline = new Timeline( //to create the animation
                new KeyFrame(Duration.seconds(1), new EventHandler<ActionEvent>() {
                    int count = 0;
                    @Override
                    public void handle(ActionEvent event) {
                        count++;
                        countLabel.setText(Integer.toString(count));
                        //Sets the text of the countLabel to the current value of the
counter variable.
                        if (count == 4) countLabel.setText("GO"); //so after 3 the 4th
will be GO
                        if (count == 5) {
                            race.startRace(onRaceFinish, finishLine, randomRacePane,
cars);
                            //calls the startRace method on race object passing these
objects
                            countLabel.setVisible(false);//count label will be not
visible
                            countLabel.setText("");
                            finishBtn.setVisible(true); //finish button will be visible
                        }
                    }
                })
        );
        timeline.setCycleCount(5);
        timeline.play();
        //lays the Timeline, which will execute the event handler every second for 5
seconds.
    }
}
// Runnable onRaceFinish, ImageView finishLine, Pane randomRacePane
```

The RaceNameField and RaceLocationField are used to store strings that identify the race and its location, respectively. The startBtn is a button that will be used to start the race. The code checks to see if the button's wording is "Start" in the first place. If it is not, the text from the raceNameField and racelocationField is read and checked to see if they are not empty. If they are, an error message is displayed and the method ends. If there are fewer than two selected drivers in the selectedDrivers linked list, a warning message is displayed instead.

When the startBtn's text is changed from "Reset" to "Start," a number of GUI elements are also changed back to their initial states, including the resultsTable being hidden, the text fields being reset, the driversTable being displayed, the

selected status of the drivers being reset, and the driversTable being refreshed. This program creates a new Race object by loading a list of Driver objects and the race's name, location, date, and time. It then uses the new Race object's simulateRace() method to run the race. A linked list of Result objects is returned, and the new Race object's setCars() method is used to generate a linked list of Car objects from those results. When the race is finished, the newRace, cars, and results linked lists are passed as parameters to the displayRace() method, which displays the results on the GUI. "Processing..." is set as the startBtn's text, and the driversTable is hidden.

The SideObject objects in the Race object are used to position the trees along the side of the track, and the carImageViews list and randomRacePane container are both added to with the tree pictures. The randomRacePane container receives the finishLine object as an addition. The spacing and leftSpace variables are used to determine the layout positions of the cars based on the number of automobiles. The car images are also added to the list of carImageViews and the randomRacePane container. A list of Result objects and a Race object are passed to the onRaceEnd function. It populates a RaceManager with the Race and any associated Result objects and toggles the visibility of several UI elements depending on whether the race has finished or not. In contrast to the trackImageView, which is concealed, the resultsTable and resultsLabel are now visible. The ResultsManager class additionally adds the results to the resultsTable. Finally, the selectedDrivers list is reset and the finishBtn is hidden. The finishBtn is brought to the foreground of the display.When the race is over, a Runnable with the name onRaceFinish is generated and called.The countLabel's visibility has been set.

Classes and Objects - : The code uses classes such as Race, Driver, Result, Car, and DriverSerialization

Encapsulation is the technique used to keep implementation specifics out of other code's view. We can see from this code that the class' instance variables are set to private and that getter and setter methods are the only ways to access them.

Polymorphism - Depending on the situation, objects can take on various forms thanks to polymorphism. The function toString() { method of the Object class is overridden by the Result class in this code to provide a unique string representation of its instance variables.

Inheritance - A class's ability to inherit traits and behaviors from its parent class is known as inheritance. We can see from this code that some classes, such as the Result and Driver classes, extend the Model class.

Abstraction -A race is abstracted by the class RandomRaceController, which hides the specifics of how the race is run and displayed.

FUNCTION STF

```
package com.Utilities;

import com.Models.Driver;
import com.Models.LinkedList;

import java.io.*;
```

```
public class DriverSerialization {
    private static final String FILENAME = "drivers.txt";

    public static LinkedList<Driver> load() {
        LinkedList<Driver> drivers = new LinkedList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(FILENAME))) {
            String line;
            while ((line = reader.readLine()) != null) {
                if (!line.equals("")) drivers.add(Driver.fromCsvString(line));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return drivers;
    }

    public static void save(LinkedList<Driver> drivers) {
        try (PrintWriter writer = new PrintWriter(new FileWriter(FILENAME))) {
            for (Driver : drivers) {
                writer.println(driver.toCsvString());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Two static methods—load and save—are included in the class. To create a LinkedList of Driver objects, the load function deserializes the serialized Driver objects it reads from the drivers.txt file. Opening a BufferedReader on the file, reading each line, and adding non-empty lines to a LinkedList of Driver objects are the steps used to accomplish this. The LinkedList is what is returned.

In contrast to load method, save method operates. A LinkedList of Driver objects are used, and they are serialized to a file called drivers.txt. To do this, a PrintWriter is opened on the file, and then each Driver object's CSV representation is written to a separate line by iterating over the LinkedList of Driver objects.

The name of the file in which the serialized data will be stored is stored in a private static String variable called FILENAME that is a part of the class.

FUNCTION RFF

```
package com.Utilities;

import com.Models.LinkedList;
import com.Models.Race;
import java.io.*;
```

```
public class RaceSerialization {
    private static final String FILENAME = "races.txt"; //Name of the file data is stored

    public static void save(LinkedList<Race> results) {
        try (PrintWriter writer = new PrintWriter(new FileWriter(FILENAME))) {
            //This method takes a LinkedList of Race objects as an argument, and writes
the
            // serialized data to the file specified by the FILENAME variable
            //PrintWriter object to write the data to the file,
            for (Race : results) {
                writer.println(race.toCsvString());
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static LinkedList<Race> load() {
        //s method reads the serialized data from the file specified by the FILENAME
variable
        // and returns a LinkedList of Race objects.
        LinkedList<Race> races = new LinkedList<>();
        try (BufferedReader reader = new BufferedReader(new FileReader(FILENAME))) {
            String line;
            while ((line = reader.readLine()) != null) {
                // For each non-empty line of data, it calls the fromCsvString() method
to
                // deserialize the data into a Race object, and adds the object to the
LinkedList.
                if (!line.equals("")) races.add(Race.fromCsvString(line));
            }
        } catch (IOException e) {
            e.printStackTrace();
        }
        return races;
    }
}
```

This method load reads the serialized data from the file with the given name and deserializes it into a LinkedList of Race objects. The try-with-resources block ensures that the BufferedReader is properly closed after the deserialization is complete. The method reads each line from the file, deserializes it into a Race object using the fromCsvString method, and adds it to the LinkedList. This method save takes a LinkedList of Race objects as a parameter and serializes the data into a file with the given name. The try-with-resources block ensures that the PrintWriter is properly closed after the serialization is complete. The method iterates through the LinkedList and writes each object's serialized data to a new line in the file. This line declares a Java class called RaceSerialization with a private static final field FILENAME that represents the name of the file where the serialized data will be stored.

TEST PLAN AND TEST CASES

ADD

NORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Driver's name – Nikoya Age – 29 Team – Avengers Car – Bugati	"Driver saved successfully"	"Driver saved successfully"

ABNORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Drivers name - ?(left blank) Age – 30 Team – fighters Car – Lamborghini	"Driver saved successfully"	"Invalid name", please enter a valid name

ABNORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Drivers name - Nick Age – 15 Team – fighters Car – Lamborghini	"Driver saved successfully"	"Invalid age", please enter an age greater than 20

ABNORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Drivers name - Nick Age – 25 Team – ?(left blank) Car - Lamborghini	“Driver saved successfully”	“Invalid team”, please enter a valid team name

ABNORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Drivers name - Nick Age – 25 Team – fighters Car - ? (left blank)	“Driver saved successfully”	“Invalid car”, please enter a valid car name

```
if (name.equals("")) {   //validates name
    Messages.error("Invalid Name", "Please enter a valid name!");
    return;
}

if (team.equals("")) {  //validates team
    Messages.error("Invalid Team", "Please enter a valid team name!");
    return;
}

if (car.equals("")) {   //validates car
    Messages.error("Invalid Car", "Please enter a valid car name!");
    return;
}

if (age < 20) {    //validates age
    Messages.error("Invalid Age", "Please enter an age greater than or equal to 20!");
    return;
}

Driver newDriver = new Driver(name, age, team, car, isUpdate ? selectedDriver.getPoints()
: 20);
//all the fields are valid, a new Driver object is created with the entered information.
if (isUpdate) DriversManager.updateDriver(selectedDriver, newDriver);
else DriversManager.addDriver(newDriver);
Messages.information("Success", "Driver saved successfully!");
```

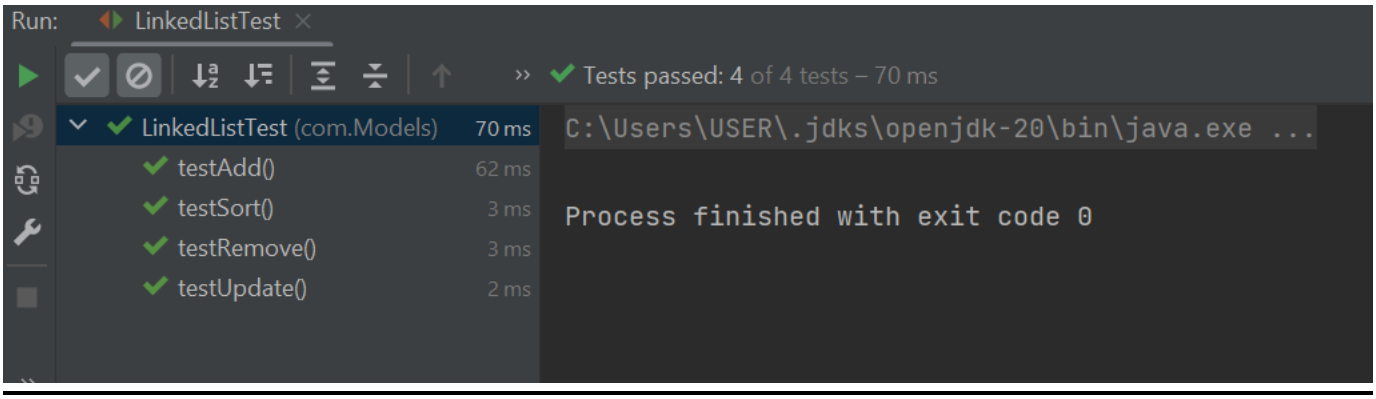
DDD

NORMAL TEST CASE	EXPECTED OUTPUT	ACTUAL OUTPUT
Driver's name – Nikoya	"Confirm deletion", Are you sure Want to delete this item	"Confirm deletion",Are you sure you want to delete this item.
Age – 29		
Team – Avengers		
Car – Bugati		

JUNITS

```
package com.Models;

import org.junit.jupiter.api.Test;

import java.util.Comparator;
import java.util.Iterator;

import static org.junit.jupiter.api.Assertions.*;
class LinkedListTest {
    @Test
    public void testAdd() {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        assertEquals(3, list.size());
        assertEquals(Integer.valueOf(1), list.iterator().next());
    }

    @Test
    public void testRemove() {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.remove(2);
        assertEquals(2, list.size());
        Iterator<Integer> iterator = list.iterator();
        assertEquals(Integer.valueOf(1), iterator.next());
        assertEquals(Integer.valueOf(3), iterator.next());
    }

    @Test
```

```java
    public void testUpdate() {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(1);
        list.add(2);
        list.add(3);
        list.update(2, 4);
        assertEquals(3, list.size());
        Iterator<Integer> iterator = list.iterator();
        assertEquals(Integer.valueOf(1), iterator.next());
        assertEquals(Integer.valueOf(4), iterator.next());
        assertEquals(Integer.valueOf(3), iterator.next());
    }

    @Test
    public void testSort() {
        LinkedList<Integer> list = new LinkedList<>();
        list.add(5);
        list.add(1);
        list.add(3);
        list.add(2);
        list.add(4);
        list.sort(Comparator.naturalOrder());
        assertEquals(5, list.size());
        Iterator<Integer> iterator = list.iterator();
        assertEquals(Integer.valueOf(1), iterator.next());
        assertEquals(Integer.valueOf(2), iterator.next());
        assertEquals(Integer.valueOf(3), iterator.next());
        assertEquals(Integer.valueOf(4), iterator.next());
        assertEquals(Integer.valueOf(5), iterator.next());
    }
}
```




```java
package com.Models;

import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

import static org.junit.jupiter.api.Assertions.*;

class DriverTest {

    private Driver;
```

```
    @BeforeEach
    void setUp() {
        driver = new Driver("Lewis Hamilton", 36, "Manly", "W12", 573);
    }

    @Test
    void getName() {
        assertEquals("Lewis Hamilton", driver.getName());
    }

    @Test
    void getAge() {
        assertEquals(36, driver.getAge());
    }

    @Test
    void getTeam() {
        assertEquals("Manly", driver.getTeam());
    }

    @Test
    void getCar() {
        assertEquals("W12", driver.getCar());
    }

    @Test
    void getPoints() {
        assertEquals(573, driver.getPoints());
    }

    @Test
    void setName() {
        driver.setName("Max Verstappen");
        assertEquals("Max Verstappen", driver.getName());
    }

    @Test
    void setAge() {
        driver.setAge(24);
        assertEquals(24, driver.getAge());
    }

    @Test
    void setTeam() {
        driver.setTeam("Red Bull Racing");
        assertEquals("Red Bull Racing", driver.getTeam());
    }

    @Test
    void setCar() {
        driver.setCar("RB16B");
        assertEquals("RB16B", driver.getCar());
    }

    @Test
    void setPoints() {
        driver.setPoints(368);
        assertEquals(368, driver.getPoints());
    }
```

```
    @Test
    void isSelected() {
        assertFalse(driver.isSelected());
    }

    @Test
    void setSelected() {
        driver.setSelected(true);
        assertTrue(driver.isSelected());
    }

    @Test
    void testToString() {
        String expected = "Driver{name='Lewis Hamilton', age=36, team='Manly', car='W12',
points=573}";
        assertEquals(expected, driver.toString());
    }

    @Test
    void toCsvString() {
        String expected = "Lewis Hamilton,36,Manly,W12,573";
        assertEquals(expected, driver.toCsvString());
    }

    @Test
    void fromCsvString() {
        String csvString = "Max Verstappen,24,Red Bull Racing,RB16B,368";
        Driver expectedDriver = new Driver("Max Verstappen", 24, "Red Bull Racing",
"RB16B", 368);
        assertEquals(expectedDriver.getName(),
Driver.fromCsvString(csvString).getName());
        assertEquals(expectedDriver.getAge(), Driver.fromCsvString(csvString).getAge());
        assertEquals(expectedDriver.getTeam(),
Driver.fromCsvString(csvString).getTeam());
        assertEquals(expectedDriver.getCar(), Driver.fromCsvString(csvString).getCar());
        assertEquals(expectedDriver.getPoints(),
Driver.fromCsvString(csvString).getPoints());
    }

    @Test
    void selectedProperty() {
        assertFalse(driver.selectedProperty().getValue());
    }
}
```

ROBUSTNESS AND THE MAINTAINABILITY

DRIVERS CONTROLLER(ADD)

In order to ensure that the user enters accurate data for the driver's name, age, team, and car, the code employs input validation. Additionally, a try-catch block is used to deal with any errors that might arise while parsing the age field as an integer. Error messages are also included in the code to let the user know if they've entered any invalid data. Overall, these precautions work to make sure that the code can process a variety of inputs without crashing or yielding inaccurate results.

The code has a clean and well-organized structure with appropriately named variables, methods, and classes, making it maintainable. The MVC (Model-View-Controller) pattern, which separates the application logic from the user interface, is one of the common design patterns it also adheres to. Additionally, comments are used in the code to describe its purpose and record any significant information. By making the code simple to comprehend, alter, and troubleshoot, all of these elements help to maintain its maintainability.

NEW DRIVERS CONTROLLER(ADD,DDD,UDD)

In this code, several features improve its robustness. For instance, the code includes error-handling mechanisms such as try-catch statements that prevent the program from crashing in case an error occurs. Additionally, the code uses confirmation messages before deleting a driver to prevent unintentional deletions. Moreover, the code implements a well-organized architecture that promotes modularity, making it easier to modify or extend the code without breaking the entire system.

Best practices like separation of concerns, modularity, and appropriate naming conventions are used in this code's design to make it maintainable. By using a different class to manage drivers, the code, for instance, divides the user interface (UI) logic from the business logic. This strategy makes it simpler to change the business logic or user interface without affecting the other. The code also adheres to a naming convention that facilitates understanding of each variable's, method's, or class's function by developers, making it simpler to maintain and modify.

RANDOMRACE DRIVERS CONTROLLER(SRR)

The RandomRaceController class seems to be built to handle a variety of error scenarios to guarantee that the application continues to run under various conditions. For instance, the class verifies that the user entered accurate data in the text fields and that the required minimum number of drivers have been chosen before the race can start. The class displays an error message to alert the user of the problem if they attempt to select more drivers than allowed or enter invalid data.

By utilizing modular and reusable elements like table views, image views, and text fields, the RandomRaceController class seems to have been created with maintainability in mind. Encapsulation is another technique that the class uses to keep the inner workings of its parts hidden from other classes. This can help to simplify the codebase and make it simpler to modify or update in the future. Additionally, the class complies with coding standards by giving each component a clear purpose in the form of comments and using meaningful variable and method names.

RACE HISTORY DRIVERS CONTROLLER(VRL)

The goToRaceViewScreen() method and the onSearch() method both check to see if the search text is empty before filtering the race list, which in this context gives the code the appearance of being robust. The error handling for loading the FXML file for the Race view screen is included in the goToRaceViewScreen() method.

Due to the use of meaningful variable and method names and other sound coding principles, such as encapsulation, the code appears to be maintainable. The RaceManager class, which separates the GUI logic from the business logic and uses the FXCollections library to manage the ObservableList of Race objects, gives the code another appearance of modularity.

CONCLUSIONS AND ASSUMPTIONS

- For the driver details I have put points as 0 as the points are not taken as a part of the driver details so the points gets updated to the table using the random race function. They ask at the end for the total points for each driver
- For the SRR function I have used only 5 drivers per Race so the points will be given to five drivers in one race.

REFERENCES

Animation using javafx

https://youtu.be/HKOJXlEJy6U

https://youtu.be/-zQzsBUHMvg