# Root Code Sample

## Development Platform

Visual Studio 2019 Enterprise  
Windows 10 Enterprise

## Project Type

C# Console project

## Running the application

In windows explorer, navigate to '<Project Root>\Drive\bin\Debug' folder.

Double-click Drive.exe to run the application. At the console, type 'trip.txt' for the file name.

> Enter file name: trip.txt

## Project Rationale

I decided to go with a C# console application as I figured it would be the quickest to get up and running. However, the project is set up in such a way that it would be UI independent (i.e, the input file could be uploaded via the web or even a Windows desktop application).

Almost all objects in the project implement interfaces. This would allow me to use dependency injection as well as allowing me to utilize mocking frameworks to make unit testing easier (I use [NSubstitute](https://nsubstitute.github.io/) for my mocking library).

I also use [LINQ](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/) to more easily calculate aggregate results, such as averages and sums. It allows me to use sql type aggregate functions on collections of objects.

For example, in the TripLogic class line 94-100, I use the Sum method to calculate the total miles driven and total duration for each driver.

IEnumerable<DriverTotals> driverTotals = trips.GroupBy(trip => trip.Driver.Name)

.Select(tripGroup => new DriverTotals()

{

Name = tripGroup.Key,

TotalMilesDriven = tripGroup.Sum(x => x.MilesDriven),

TotalMinutesDriven = tripGroup.Sum(x => x.Duration.TotalMinutes)

});

I first created the two main business objects (models) along with their respective interfaces (IDriver, Driver, ITrip, Trip). The driver has one property of Name. The Trip class includes properties for Driver, StartTime, EndTime, MilesDriven, & also an additional property, Duration, that calculates the trip duration based on StartTime and EndTime. I also created 2 additional business objects (DriverAverages, DriverTotals) to store aggregate result for each driver to be used to calculate the final output.

The main logic is contained in the TripLogic class. This class includes methods for creating the list of valid drivers, the list of trips, and for calculating the final results.

I also created a couple of wrapper classes and interfaces that would allow me to inject the native C# Console object and the StreamReader object. This also allowed me to mock these objects during unit testing.

All the logic for calculating the result are in TripLogic.CalculateResulst(). Below is a explanation of what was done:

(Lines 94-100): Get total miles driven and total duration for each driver and map to a collection of DriverTotals objects.

IEnumerable<DriverTotals> driverTotals = trips.GroupBy(trip => trip.Driver.Name)

.Select(tripGroup => new DriverTotals()

{

Name = tripGroup.Key,

TotalMilesDriven = tripGroup.Sum(x => x.MilesDriven),

TotalMinutesDriven = tripGroup.Sum(x => x.Duration.TotalMinutes)

});

(Lines 102-107): Use the DriverTotals list created above to calculate the average mph for each driver. Map the results to a collection of DriverAverages objects.

IEnumerable<DriverAverages> driverAverages = driverTotals

.Select(x => new DriverAverages

{

Name = x.Name,

TotalMilesDriven = Math.Round(x.TotalMilesDriven),

AverageMph = Math.Round((double)x.TotalMilesDriven / (x.TotalMinutesDriven / 60))

});

(Lines 109-110): Removes any average mphs that are less that 5mph and greater than 100 mph. Sort the result in descending order.

List<DriverAverages> validAverages = driverAverages

.Where(x => x.AverageMph >= 5 && x.AverageMph <= 100).OrderByDescending(x => x.TotalMilesDriven).ToList();

(Lines 112-115): Add these to a string list of calculated results.

validAverages.ForEach(x =>

{

driverResults.Add($"{x.Name}: {x.TotalMilesDriven} miles @ {x.AverageMph} mph");

});

(Lines 117-119): For any driver that does not have any associated trips, their results is 0 miles. Add this at the end of the existing string list.

driverResults.AddRange(drivers.OrderBy(x => x.Name)

.Where(driver => trips.All(x => x.Driver != driver))

.Select(driver => $"{driver.Name}: 0 miles"));

## Unit Tests ([NUnit](https://nunit.org/) Test Project)

I created 2 Unit Test classes. The first is to test the Program class. The second class tests the TripLogic class.

#### ProgramTests:

* Test to see if the file name entered is a valid file. If a file is not found, it displays the message in the console, "File not found. Please try again.".

public void Run\_writes\_FileNotFoundMessage\_to\_console\_when\_FileNotFound()

* Test to see if 3 invalid attempts have been made when entering the file name. If so, it displays the message, "You have exceeded the maximum number of tries.".

public void Run\_writes\_MaxNumberExceeded\_to\_console\_when\_FileNotFound3Times()

* Test to make sure that the Data directory has been correctly appended to the file name to get the actual file path as that is the expected directory of the data file.

public void PromptForFile\_return\_FilePath()

* Test to make sure the calculated results are written to the console.

public void Run\_writes\_DriverResults\_to\_console\_when\_DriverResultsExist()

#### TripLogicTests

* Test to make sure that list of valid drivers does not return null or empty

public void RegisterDriver\_returns\_NonEmptyDrivers()

* Test to make sure that the list of trip entries does not return null or empty

public void RegisterTrips\_returns\_NonEmptyTrips()

* Test to make sure a trip entry included a valid driver. If not, it will skip that trip entry

public void RegisterTrips\_skips\_TripEntry\_if\_DriverDoesNotExist()

* Test to make sure that start time is less than end time. If not, it will skip the entry.

public void RegisterTrips\_skips\_TripEntry\_if\_StartTimeGreaterEndTime()

* Test to make sure the RegisterTrips method creates the correct Driver object based on the Driver entry.

public void RegisterTrips\_returns\_CorrectTripDataForDriver()

* Test to make sure that calculated results do not return empty or a null list of results.

public void CalculateResults\_returns\_NonEmptyResults()

* Test to make to make sure the calculations are correct for a driver

public void CalculateResults\_returns\_CorrectResultForDriver()

* Test to make sure that 0 miles driven is returned if the driver averaged less than 5 miles.

public void CalculateResults\_LessThanFiveTotalMiles\_returns\_ZeroMilesDriven()

* Test to make sure that 0 miles driven is return if the driver averaged more than 100 mph.

public void CalculateResults\_GreaterThanOneHundredTotalMiles\_returns\_ZeroMilesDriven()

* Test to make that the results are sorted by miles driven in descending order.

public void CalculateResults\_returns\_ResultsSortedByMilesDrivenDesc()