**Data Structures and LINQ**

**Activity 4.01: Treasury Flight Data Analysis**

**Solution:**

1. Create a new folder called **Activities** in the **Chapter04** folder.
2. Add a new folder called **Activity01** to that folder.
3. Add a new class file called **Flight.cs**:

namespace Chapter04.Activities.Activity01

{

internal record Flight (string Agency, double PaidFare,

string TripType, string RoutingType, string TicketClass,

string DepartureDate, string Origin, string Destination,

string DestinationCountry,

string Carrier);

}

This will be a **Record** class with fields that match those in the flight data. A **record** type is used as it offers a simple type purely to hold data rather than any form of behavior.

1. Add a new class file called **FlightLoader.cs**. This class is responsible for downloading or importing data.
2. Add an interface that exposes the **Import** and **Download** methods, as follows:

using System;

using System.Collections.Generic;

using System.IO;

using System.Linq;

using System.Net;

namespace Chapter04.Activities.Activity01

{

internal interface IFlightLoader

{

IList<Flight> Import(string filePath);

IList<Flight> Download(string url, string filePath);

}

6. Now for the **FlightLoader** implementation, use a **static** class to define the index of known field positions in the data file:

internal class FlightLoader : IFlightLoader

{

private static class ImportFieldIndex

{

public const int Agency = 0;

public const int PaidFare = 1;

public const int TripType = 2;

public const int RoutingType = 3;

public const int TicketClass = 4;

public const int DepartureDate = 5;

public const int Origin = 6;

public const int Destination = 7;

public const int DestinationCountry = 8;

public const int Carrier = 9;

}

This makes it easier to handle any futures changes in the layout of the data.

1. The **Import** interface must be passed the name of the local file to import. So,

**Skip** the first line in the file as that is a header.

1. Split each line into a **string** array, one element for each field:

public IList<Flight> Import(string filePath)

{

var flights = new List<Flight>();

// Skip the header line

foreach (var line in File.ReadLines(filePath)

.Skip(1)

.Where(ln => !string.IsNullOrWhiteSpace(ln)))

{

var fields = line.Split(",");

Once each line is split, the number of elements is expected to be exactly nine, and you must ensure that **Agency** (at element **0**) is not a null or empty string.

1. If that is the case, use the **continue** statement to skip the current line and move onto the next, as follows:

if (fields.Length < ImportFieldIndex.Carrier ||

string.IsNullOrEmpty(fields[ImportFieldIndex. Agency]))

continue;

1. Parse the **PaidFare** element from a string into a **double** value and create a new **Flight** record:

**FlightLoader.cs**

double.TryParse(fields[ImportFieldIndex.PaidFare],

out double paidFare);

var flight = new Flight(

fields[ImportFieldIndex.Agency],

paidFare,

fields[ImportFieldIndex.TripType],

fields[ImportFieldIndex.RoutingType],

fields[ImportFieldIndex.TicketClass],

fields[ImportFieldIndex.DepartureDate],

fields[ImportFieldIndex.Origin],

fields[ImportFieldIndex.Destination],

fields[ImportFieldIndex.DestinationCountry],

fields[ImportFieldIndex.Carrier]);

1. Next, pass a URL and destination file to the **Download** method.
2. Use **WebClient.DownloadFile** to download the data file and then defer to **Import** to process the downloaded file, as follows:

public IList<Flight> Download(string url, string filePath)

{

using var client = new WebClient();

client.DownloadFile(url, filePath);

return Import(filePath);

}

}

}

13. Add a new class file called **FilterCriteria.cs** as follows:

using System;

namespace Chapter04.Activities.Activity01

{

internal enum FilterCriteriaType

{  
Class,

Origin,

Destination

}

internal record FilterCriteria(FilterCriteriaType Filter,

string Operand);

}

This class contains an **enum** and a **string** value, **Operand**. Use a **List** of these to define the user's filter criteria. The **enum** determines the type of filter that has been requested, and **Operand** represents the **string** to search for, such as **Economy**.

14. Now, for the main filtering class, add a new class file called **FlightQuery.cs**:

using System;

using System.Collections.Generic;

using System.Linq;

namespace Chapter04.Activities.Activity01

{

internal class FlightQuery

{

private readonly List<Flight> \_flights = new();

private readonly List<FilterCriteria> \_filters = new();

private readonly IFlightLoader \_loader;

public FlightQuery(IFlightLoader loader)

{

\_loader = loader;

}

Here the constructor will be passed a **FlightLoader** instance (declared using the **IFlightLoader** interface). The **\_flights** list contains the data imported via the **FlightLoader** instance. **\_filters** is a list **List<FilterCriteria>** instance that represents each of the criteria items that are added every time the user specifies a new filter condition.

15. Next, let the **Import** and **Download** methods be called by the console at startup, allowing previously downloaded data to be processed via the **\_loader** instance:

public void Import(string path)

{

\_flights.Clear();

\_flights.AddRange(\_loader.Import(path));

}

public void Download(string url, string filePath)

{

\_flights.Clear();

\_flights.AddRange(\_loader.Download(url, filePath));

}

The **Count** variable simply returns the number of flight records that have been imported. This is defined as follows:

public int Count => \_flights.Count;

16. When you specify a filter to add, the console should call **AddFilter**, passing an **enum** to define the criteria type and the **string** value being filtered for. The code for this can be added as follows:

public void AddFilter(FilterCriteriaType filter, string operand)

{

\_filters.Add(new FilterCriteria(filter, operand));

Console.WriteLine($"Added filter: {filter}={operand}");

}

public void ClearFilters()

{

\_filters.Clear();

Console.WriteLine("Cleared filters");

}

17. Add **RunQuery** as the main method that returns those flights that match the user's criteria.

18. Use the built-in **StringComparer.InvariantCultureIgnoreCase** comparer to ensure string comparison ignores any case differences.

19. Also, define a query variable that calls **Select** on the flights. At present, this results in a filtered result set:

public IList<Flight> RunQuery()

{

var comparer = StringComparer.InvariantCultureIgnoreCase;

var query = \_flights.Select(f => f);

All types of available filters are string-based, so you must extract all string items.

20. If there are any items to filter, add an extra **Where** call to the query for each type

(**Class**, **Destination**, or **Origin**).

21. Use a **Contains** predicate for each **Where** clause, which examines the associated property. Add the following code for this:

**FlightQuery.cs**

var filterClasses = GetFiltersByType(FilterCriteriaType.Class);

if (filterClasses.Any())

{

query = query.Where(flt =>

filterClasses.Contains(flt.TicketClass, comparer));

Console.WriteLine($"Classes: {FormatFilters(filterClasses)}");

}

var filterDestinations = GetFiltersByType(FilterCriteriaType.Destination);

if (filterDestinations.Any())

{

query = query.Where(flt =>

filterDestinations.Contains(flt.Destination, comparer));

Console.WriteLine($"Destinations:

{FormatFilters(filterDestinations)}");

22. Next, add the two helper methods used by **RunQuery**:

private IList<string> GetFiltersByType(FilterCriteriaType

filter)

=> \_filters

.Where(f => f.Filter == filter)

.Select(f => f.Operand)

.ToList();

In the preceding snippet, **GetFiltersByType** is passed each of the **FilterCriteriaType** enums that represent a known type of criteria type and finds any of these in the list of filters using the **.Where** method. For example, if the user added two **Destination** criteria such as **India** and **Germany**, this would result in the two strings **India** and **Germany** being returned.

1. Now add **FormatFilters** as shown in the next snippet:

private string FormatFilters(IEnumerable<string> filterValues)

=> string.Join(" OR ", filterValues);

}

}

This simply joins a list of **filterValues** strings into a user-friendly string with the word **OR** between each item, such as **London OR Dublin**.

1. Now create the main console app. To do so, add a new class called **Program. cs**, which allows the user to input requests and process their commands,  
   as follows:

using System;

using System.IO;

using System.Linq;

namespace Chapter04.Activities.Activity01

{

class Program

{

public static void Main()

{

25. Hardcode the download URL and destination filename as follows:

const string FlightDataUrl = "https://www.gov.uk/

government/uploads/system/uploads/attachment\_data/file/245855/HMT\_-

\_2011\_Air\_Data.csv";

const string FlightDataFilePath = "hm-treasury-flight-

data-2011.csv";

26. Create the main **FlightQuery** class, passing in a **FlightLoader** instance:

var flightQuery = new FlightQuery(new FlightLoader());

if (File.Exists(FlightDataFilePath))

{

Console.WriteLine($"Importing {FlightDataFilePath}");

flightQuery.Import(FlightDataFilePath);

Console.WriteLine();

} else {

Console.WriteLine($"Downloading {FlightDataUrl}");

flightQuery.Download(FlightDataUrl,

FlightDataFilePath);

Console.WriteLine();

Console.WriteLine($"Downloaded to

{FlightDataFilePath}...");

}

If the app has been run before, you can **Import** the local flight data, or use **Download** otherwise.

27. Show a summary of the records imported and the available commands:

Console.WriteLine($"Found {flightQuery.Count} flight

records");

const string GoCommand = "go";

const string ClearCommand = "clear";

const string ClassCommand = "class";

const string OriginCommand = "origin";

const string DestinationCommand = "destination";

Console.WriteLine($"Commands: {GoCommand} |

{ClearCommand} | {ClassCommand} value | {OriginCommand} value |

{DestinationCommand} value");

1. Use a **do** loop to allow the user to enter their command and any required arguments:

string input;

do

{

Console.Write("Enter a command:");

input = Console.ReadLine().ToLower();

When the user enters a command, you can expect that this may also have specified an argument, such as **destination united kingdom**, where **destination** is the command and **united kingdom** is the argument.

1. In order to determine this, use the **IndexOf** method to find the location of the first space character in the input (if any):

string command;

string argument;

var spaceIndex = input.IndexOf(' ');

if (spaceIndex == -1)

{

command = input;

argument = null;

}

else {

command = input[..spaceIndex].Trim();

argument = input[spaceIndex..].Trim();

}

This allows a range to extract the first letters into a **command** variable and the last characters into an **argument** variable.

1. Add a **switch** statement to process the commands.
2. For the **GoCommand**, call **RunQuery** and use various aggregation operators on the results returned, as follows:

switch (command)

{

case GoCommand:

var flights = flightQuery.RunQuery();

if (flights.Any())

{

var average = flights.Average(fl=>fl. PaidFare);

var min = flights.Min(fl => fl.PaidFare);

var max = flights.Max(fl => fl.PaidFare);

Console.WriteLine($"Results:

Count={flights.Count}, Avg={average:N2}, Min={min:N2}, Max={max:N2}");

}

else

{

Console.WriteLine("No matching flights. found");

}

break;

32. For the remaining commands, clear or add filters as requested.

33. If **ClearCommand** is specified, call the query's **ClearFilters** method, which clears the list of criteria items:

case ClearCommand:

flightQuery.ClearFilters();

break;

34. If a **class** filter command is specified, call **AddFilter** specifying the **FilterCriteriaType.Class** enum and the string **argument**:

case ClassCommand:

flightQuery.AddFilter(FilterCriteriaType.Class, argument);

break;

35. Use the same pattern for the **Origin** and **Destination** commands. 36. Call **AddFilter**, passing in the required **enum** value and the argument:

case OriginCommand:

flightQuery.AddFilter(FilterCriteriaType.Origin, argument);

break;

case DestinationCommand:

flightQuery.AddFilter(FilterCriteriaType. Destination, argument);

break;

}

} while (input != string.Empty);

}

}

}

37. Run the console app by adding various filter criteria. This produces the following output:

Importing hm-treasury-flight-data-2011.csv

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

.....................................................................

..........................

Found 716 flight records

Commands: go | clear | class value | origin value | destination value

1. Enter **go** to see the unfiltered results. You should get a result similar to the following:

Enter a command:go

Results: Count=716, Avg=693.58, Min=-1,678.19, Max=5,669.23

1. Next, add two class filters, for **economy** or **Business Class**:

Enter a command:class economy

Added filter: Class=economy

Enter a command:class Business Class

Added filter: Class=business class

As you can see, the output shows the user input in lower case, although the flight query class is case insensitive when searching.

1. Now, enter **go** to show the filtered results:

page23image16084560

Enter a command:go

Classes: economy OR business class

Results: Count=695, Avg=685.74, Min=-1,678.19, Max=5,669.23

1. Add **London** as an origin and **Zurich** as a destination:

Enter a command:origin london

Added filter: Origin=london

Enter a command:destination zurich

Added filter: Destination=zurich

Again, the output shows the user input in lowercase.

42. Finally, enter **go** once more to show the filtered aggregate values:

Enter a command:go

Classes: economy OR business class

Destinations: zurich

Origins: london

Results: Count=16, Avg=266.92, Min=-74.71, Max=443.49

You have now created a console app that allows the user to download publicly available flight data files and apply statistical analysis to the files.

In this activity, you created a console app to download publicly available flight data files and apply statistical analysis to the files. This data file contained details of flights made by the UK's HM Treasury department between January 1 to December 31, 2011 (there are 714 records.)