# .NET Exercises

The first few steps of the exercises are just to get started with Visual Studio .NET. The later exercises are to create an application to enter/display passenger details. The application developed for the majority of exercises gradually evolves as new language/framework features are introduced.

## Create a Solution

1) Go to the File menu File|New|Project… . Within the New Project dialog select 'Other Project Types'|'Visual Studio Solutions' and 'Blank Solution' from the Templates window. Browse to a suitable location for the exercises folder and name the solution 'Exercises'. Click OK. An empty Solution should appear in the Solution Explorer.

## Simple .NET Application

2) Right click on the new Solution within the Solution Explorer and from the context menu select Add|New Project… . The New Project dialog will appear. Within the Project Types select "C#" and within the Templates window select Console Application. Name the project "Hello" and click OK. The project will be added to both the "Solution Explorer" and the "Class view".

   The Program.cs file will open within the editor. Add the following line to the Main() method:

```
Console.WriteLine( "Hello Universe");
```

   Build the project and test.

## Input and Output Exercise (Flight Application)

This application will be extended in further exercises. Create an application for handling Passenger Details for tracking passengers. However, only a simple representation of a passenger is required for the purpose of this appliction.

3) Add a new Console project to the solution called "FlightConsole". Within the Main() method declare variables for a 'name' (String) and 'weight' (int). Use the Console.WriteLine() method to prompt the user and Console.ReadLine() method to input the name and weight (use int.Parse()). Output the name and weight.

Note: If using C# 6, string interpolation $"" can be used.

   Build the project and test.

*Debugging: Stepping and Locals Window*

   Step through the program with the debugger. Use the "Locals" window to see the values of variables change.

 

**Method Exercise**

4) Separate the functionality, from the previous exercise, by writing public 'static' methods called InputPassengerDetails() and OutputPassengerDetails() to perform the input and output respectively. InputPassengerDetails() requires the values to be passed by reference (in order for Main() to access them). Call these methods from within the Main() method.

Note: Quick Actions and Refactoring option can be used to extract these methods. There is no need to write these methods from scratch.

Note: These early exercises methods will need to be 'static' in order that they can be called from Main(), which is also a static method.

Build the project and test.

*Debugging: Stepping and Changing Values*

Step through the program with the debugger. Step into the methods and watch the values of variables change.

Option:

Try using the watch window to modify the value of the weight after it has been entered.

**For Exercise**

5) Within the Main() method add a "for" loop to repeatedly input and output a number of passenger details. Define a constant for the maximum number of passenger details to input and output (initialize to 4).

Build the project and test.

**Loop Exercise**

6) Change the code in the Main() method to allow the user to select an option for the required action. Use a do-while loop to perform the iteration. Display a menu of options to the user (i - input, o - output and x – exit) and then request an option (String). Use a switch statement to call the appropriate functionality; using the default case to inform the user they have entered an invalid option. Exit the loop when the user enters 'x'; set a Boolean variable to indicate when 'x' is entered.

Build the project and test.

Does the program work if the user enters upper case letters? How can the program be modified to handle alternative cases?

 

**Array Exercise**

7) Declare arrays to hold names and weights. Initialize a counter to zero. Within the 'i' option copy the passenger details into the arrays, subscripted by the counter. Increment the counter when passenger details are added. Allow the 'o' option to output the last entered passenger details. Add a new option 'a' to output all of the passenger details; implement this by iterating over the arrays filled so far. Output a message to the user if the arrays are full when the user is attempting to enter another passenger details.

Build the project and test.

*Debugging: Locals Window and Conditional Break Points*

Step through the program with the debugger. Use the Locals window to see the arrays and watch the values change.

Set a breakpoint within the loop. Select "Breakpoint | Condition…" from the context menu. Add an expression for the counter equaling 2. Run the program in the debugger and observe the program stop.

**Class Exercise**

8) Use a Visual Studio Class Diagram to add a new class called PassengerDetails to the project. Add fields for name and weight.

Note: Use Quick Action and Refactoring option 'Encapsulate Field' to provide properties for these fields.

Move the methods InputPassengerDetails() and OutputPassengerDetails() from the Program class to the PassengerDetails class. Modify these methods to be public non-static methods called Input() and Output() respectively with no parameters. These Input() and Output() methods should now access the name and weight fields. Within the Main() method allocate a PassengerDetails object. Call the appropriate Input() and Output() methods on the object within the Main() method.

Build the project and test.

Add a constructor to PassengerDetails, which takes the name and weight, to initialize the fields. Do we also need to add a default constructor to PassengerDetails? Add code to try out the constructor(s).

Build and test.

 

*Debugging: Locals Window "this"*

Set a breakpoint where the Input() method is called. Step into the Input() method. Within the Locals window observe the values of the members of "this".

Build the project and test.

**Structure Exercise**

9) Try swapping the PassengerDetails from a class to structure. Change back afterwards.

Build the project and test.

**Collection Exercise**

10) Replace the array with a generic List of PassengerDetails. In the Main() method within the 'i' option add the PassengerDetails to the List. Within the 'a' option add a 'foreach' loop to output the passenger details in the list.

Build the project and test.

**Inheritance Exercise**

Airlines often have frequent flier clubs and therefore it may be necessary to distinguish between different types of passenger.

11) Create a new class called SilverPassengerDetails, which inherits PassengerDetails (the class diagram can be used). Add the virtual keyword to the Input() and Output() methods of the PassengerDetails class. Override the Input() and Ouput() methods within the SilverPassengerDetails class. Within its Input() and Output() methods call the 'base' class Input() and Output() methods respectively. Precede these calls with the prompt to the user with "Silver club passenger".

Within the Main() method add a new option 's' within the switch statement for entry of SilverPassengerDetails. The 's' option functionality should be similar to that of the 'i' option. Instantiate a SilverPassengerDetails object within the 's' option.

Add a constructor for SilverPassengerDetails, taking both the name and weight. The SilverPassengerDetails constructor should call the constructor for its base class. Add an appropriate default constructor.

Build the project and test.

 

**Optional:**

**Static Member Exercise**

11a) Define another class called DiagnosticCount. Add two fields (one static field) to provide the count and the other to uniquely number the objects created (readonly field). Initialise the static field to zero.

Add a constructor and within the constructor increment the count and assign to the number field of the class. Output a message indicating the number of objects created.

PassengerDetails should now inherit from the DiagnosticCount class.

Build and test.

**Exception Handling Exercise**

12) Add exception handling to the application to handle an exception thrown by the int.Parse() method. Exception handling could be added either within the Input() method, or where the Input() method is called.

Build and test.

**Interfaces Exercise**

Currently passenger details are output in the order they are entered. This exercise allows sorting of the List (sort by Name).

13) For the class PassengerDetails implement the generic interface IComparable<PassengerDetails>. Define the CompareTo() method to return the value for comparing the name fields (alphabetic order). Within the Main() method modify the 'a' option of the switch statement; sort the List of PassengerDetails before the passenger details are output (use the Sort method).

Build and test.

**Option:**

**IComparer Interface**

13a) Add a new public nested class, within the PassengerDetails class, called CompareByNameReverse which implements IComparer<PassengerDetails>. Define the Compare() method (to implement the interface) to compare the names of PassengerDetails (to be in reverse alphabetic order). Modify the 'a' option and instantiate a CompareByNameReverse object. Pass this object to the Sort() method of the List.

Build and test.

 

**Find Exercise**

14) To the switch statement add an 'f' option for "find". Within this option prompt the user for a name and instantiate a PassengerDetails with this name. Use the BinarySearch() method of the List to find the index for an object.

If this index is not -1 then output the passenger details.

Build and test. If this does not work why?

Call the Sort() method (no parameters).

Build and test.

**Method Overriding Exercise**

To complete the definition of a class implement the overridable methods from object class.

15) Override the Equals() method for PassengerDetails class. Define the Equals method to compare the members of the class.

Build and test.

Note: The compiler should give a warning if 'GetHashCode' is not defined.

To complete the PassengerDetails class add override methods for ToString() and GetHashCode().

Build and test.

**Library Exercise**

16) Add a new Class Library project to the solution called 'FlightHandling'. Cut the files PassengerDetails.cs and SilverPassengerDetails.cs (and possibly DiagnosticCount.cs) from the application project and paste into the 'FlightHandling' project. Within these files change the namespace to 'FlightHandling'. Check that access for the PassengerDetails and SilverPassengerDetails classes are public.

Build the project FlightHandling.

To the application project add a reference to FlightHandling. To the top of the file containing the Main() method add the following:

```
using FlightHandling;
```

Build and test.

 

*Debugging:*

Set a breakpoint within the Input() method. Run the application to observe the breakpoint within the library.

**Write to File Exercise**

17) To the switch statement add a ‘w’ option for “write to file”. Within this option add a using statement, instantiating a ‘StreamWriter’. Open this ‘StreamWriter’ on a text file (within temp folder or desktop). Use a ‘foreach’ loop to iterate over the List of PassengerDetails to output to the file using ‘WriteLine’ method.

Build and test.

**Linq Exercise**

18) Modify the ‘f’ option to find a passenger details using the extension method ‘FirstOrDefault’. Use a lambda function to find by the Name property.

If the returned values is not null, output the passenger details, otherwise output a message indicating it is not found.

Build and test.

18a) Modify the ‘f’ option to find a passenger details using a LINQ statement.

Build and test.

 