”Modern C# for Python Developers”

Lab Manual

Jesper Gulmann Henriksen

30-09-2025

v1.2

Table of Contents

Exercise types

The exercises in the present lab manual differ in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:

Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.

Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.

Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! ☺

Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Code\ModernCSforPyDevs
```

with .NET 8 and a compatible IDE or tools installed on the machine.

We will henceforth refer to the chosen installation path containing the lab files as *PathToCourseFiles* .

Session 1

None. 😊

Session 2

Lab 2.1: "Inheritance"

The purpose of this exercise is to get hands-on experience with the unfamiliar syntax of C# for concepts that are very familiar to you in Python.

- Open the starter project in
  PathToCourseFiles`\Session 2\Labs\Lab 2.1\Begin` ,
  which contains a project called `Inheritance`.

This is essentially the example from the presentation cleaned up slightly. Each class has been moved to a separate file and provided a namespace definition.

a) Make it compile

For some syntactic reason the program doesn't compile now.

- Fix the syntactic error and compile and run the program.

b) Extend the textual presentation

When the program runs it currently prints the following output to the console:

```
John Doe [25763 C# lines]
John Doe [26250 C# lines]
```

All employees have an employee number which is not printed.

- Make the output appear as

```
100000: John Doe [25763 C# lines]
100000: John Doe [26250 C# lines]
```

c) Another Employee Type

The fictitious company OODev has several other employee types other than `SoftwareEngineer`.

- Create an appropriate type `SoftwareArchitect` creating Visio drawings at work.

Adjust the main program such that it produces the following output:

```
100001: Jane Doe [176 drawings]
100000: John Doe [25763 C# lines]
100001: Jane Doe [178 drawings]
100000: John Doe [26250 C# lines]
```

Lab 2.2: "Exception Handling" ( )

The purpose of this exercise is to map a Python program to C# and discover the similarities and problems.

- Open the starter project in
  PathToCourseFiles`\Session 2\Labs\Lab 2.2\Begin` ,

which contains a project called `ExceptionHandling`.

The project contains the following Python function:

```
def divide(x,y):
    try:
        result = x/y
    except ZeroDivisionError:
        print("Please make 'y' non-zero")
    except:
        print("Something went wrong")
    else:
        print(f"Your answer is {result}")
    finally:
        print("Aaaaand... We're done!")
```

Your task is now to

- Rewrite it to an equivalent C# method.
- Run the program and test it.

Did you encounter any problems or surprises?

Lab 2.3: "Object Deconstruction" (⭐⭐)

The purpose of this exercise is to implement object destruction to tuples of a preexisting class.

- Open the starter project in
  PathToCourseFiles`\Session 2\Labs\Lab 2.3\Begin` ,

which contains a project called `ObjectDestruction`.

The starter solution consists of two projects – a client project and a class library called `DiscographyLab`.

The class library contains an existing class `Album`. That class is part of an externally supplied API and cannot be modified or derived from:

```
public sealed class Album(string artist, string albumName,
                          DateTime releaseDate)
{
    public Guid Id { get; } = Guid.NewGuid();
    public string Artist { get; } = artist;
    public string AlbumName { get; } = albumName;
    public DateTime ReleaseDate { get; } = releaseDate;
}
```

The client project contains top-level statements with the following code:

```
Album album = new Album(
    "Depeche Mode",
    "Violator",
    new DateTime( 1990, 3, 19 )
);

(_, string summary, int age) = album;
Console.WriteLine( $"{summary} is {age} years old");
```

Currently this code does not compile. You need to fix that.

- Your task is to add an appropriate class and method to make the code compile.
  - Note: You should **not** change anything in the top-level statements or the `Album` class.

When completed, your `Program.cs` should produce the following output:

```
"Violator" by Depeche Mode is 35 years old
```

Session 3

Lab 3.1: "Expression-bodied Members" ( )

- Open your completed project in
  PathToCourseFiles`\Session 2\Labs\Lab 2.1\Complete` ,
  which contains your `Inheritance` project completed earlier.

Let's update existing methods and properties to be expression-bodied and more functional and "modern".

- Make as much of the solution as you like expression-bodied.
- Are there areas of the code which you would prefer not to be expression-bodied?

If time permits:

- What happens if you make the `Number` property of `Employee` expression-bodied?

Lab 3.2: "Playing with Pattern Matchings" (★)

In this exercise we will see several different ways of using the new patterns for processing employees.

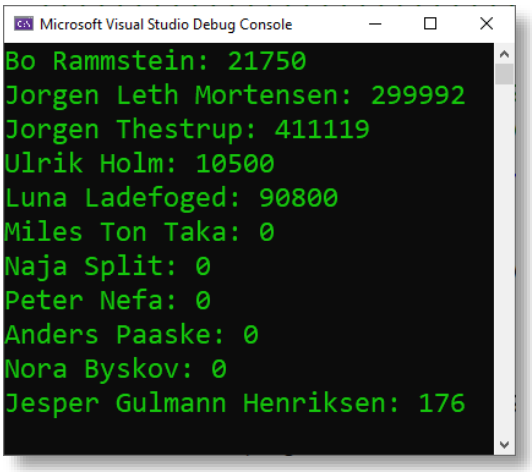
- Open the starter project in
  PathToCourseFiles`\Session 3\Labs\Lab 3.2\Begin` ,
  which contains a project called `PatternMatching` with `Employee` data supplied in `Data`.

Write the Code Production Index for each employee

The fictitious *Code Production Index* for an `Employee` is defined as

- the number of code lines produced (for `SoftwareEngineer`)
- for `SoftwareArchitect`, each Visio drawing produced corresponds to 250 code lines produced
- any other employee has a code production index of 0.

Use appropriate pattern matching to list all employees along with their code production index, e.g.



Find all Student Programmers mentored by a Chief Software Engineer

Construct a LINQ expression capturing a sequence of `StudentProgrammer`s who are mentored by a `Chief SoftwareEngineer`.

Lab 3.3: "LINQ Additions in .NET 6" (★)

This lab investigates the new methods and overloads to methods, which .NET 6 has added to LINQ.

- Open the starter project in
  PathToCourseFiles`\Session 3\Labs\Lab 3.3\Begin` ,
  which contains a project called `DotNet6 LINQ`.

The project already contains a simple `Movie` type as well as a hardcoded data set of such instances:

```
IEnumerable<Movie> movies =
[
    new("Total Recall", 2012, 6.2f),
    new("Evil Dead", 1981, 7.5f),
    new("The Matrix", 1999, 8.7f),
    new("Cannonball Run", 1981, 6.3f),
    new("Star Wars: Episode IV - A New Hope", 1977, 8.6f),
    new("Don't Look Up", 2021, 7.3f),
    new("Evil Dead", 2013, 6.5f),
    new("Who Am I", 2014, 7.5f),
    new("Total Recall", 1990, 7.5f),
    new("The Interview", 2014, 6.5f)
];
```

The program compiles but contains 6 queries which are currently empty. You will need to fill out the code of these queries located at 6 different TODOs in the code – perhaps using some of the updated LINQ methods added in .NET 6.

You can choose to use include one or more of these newly added methods:

- `ElementAt<T>` and `ElementAtOrDefault<T>`
  - New support for `Index`
- `Take<T>`
  - New support for `Range`
- `XxxOrDefault<T>`
  - New support for supplying default
- New `Chunk<T>` method
- New `DistinctBy<T>`, `MinBy<T>` and `MaxBy<T>` methods
- New `UnionBy<T>`, `IntersectBy<T>`, and `ExceptBy<T>`.

Now go ahead and solve the following tasks:

- Locate `TODO: a)` in the code and make `queryA` produce
  - the movie which premiered first
    - without using the `OrderBy()` method!
- Locate `TODO: b)` in the code and make `queryB` produce
  - the first movie with a rating above 9.0 (or just the first movie if no such high-rated movie exists)

- Locate `TODO: c)` in the code and make `queryC` produce
  - the second-to-last movie of the list (if it exists)
- Locate `TODO: d)` in the code and make `queryD` produce
  - all the movies except the first and last to premiere.
- Locate `TODO: e)` in the code and make `queryE` produce
  - the sequence of all movies with the remakes removed.
- Locate `TODO: f)` in the code and make `queryF` produce
  - A grouping of the movies into groups of 4 movies each
    - With the last group potentially containing fewer than 4 elements, if 4 does not divide the total number of movies.

Lab 3.4: "Matching, Merging, and Spreading Collection Expressions" ()

This lab will investigate collection expressions and spread operator as well as illustrating the syntax equivalences to list patterns.

- Open the starter project in
  PathToCourseFiles`\Session 3\Labs\Lab 3.4\Begin` ,
  which contains some setup code defining two ordered lists.

The code contains a method

```
List<int> Merge(List<int> list1, List<int> list2) =>
    (list1, list2) switch
    {
        // TODO
    };
```

which you need to complete.

- Complete the implementation to merge the two ordered lists into a resulting ordered list using the list patterns of C# 11 along with the corresponding syntax of collection expressions and the spread operator of C# 12.

Your code should produce the following resulting ordered list:

```
11 22 33 44 44 55 66 77 88 99 99
```

Session 4

Lab 04.1: "Injecting Composites and Proxies" ()

We will now consider variations of using dependency injection containers, in particular the composites and proxies that we have seen earlier.

- Open the starter project in
  PathToCourseFiles`\Session 4\Labs\Lab 4.1\Begin` ,
  which contains a project called `Injection Fun`.

The existing code contains the combined solutions for the "Living SOLID-ly" discussions which we used in Session 4.

Injecting Composites

Remember the `CompositeWriteStorage` write storage from before?

- Can you modify the dependency injection code to make the project write to a `CompositeWriteStorage` composed of both of
  - `ConsoleStorage`
  - `FileStorage`?

Hint: There are a number of (good and bad) ways of setting this up, but Keyed Services which became available with .NET 8 is probably the least ugly way. Read more about those here: https://learn.microsoft.com/en-us/dotnet/core/extensions/dependency-injection#keyed-services .

Injecting Proxies

Now… What about the `RetryWriteStorageProxy` from earlier?

- Could you further modify the dependency injection configuration to retry the `CompositeWriteStorage`?