TEKNOLOGISK
INSTITUT

”C# and .NET”

Lab Manual

Wincubate ApS

10-01-2023

V1.0

Table of Contents

Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less 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:\Wincubate\90383
```

with Visual Studio 2022 with .NET 6 or later installed on the PC.

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

Part 1: “New Features of C# 8 and 9”

Lab 1.1: ”Adding Nullability to Reference Types”

In this exercise we will retrofit an existing sequence type with the nullable reference operators ? and `!` to express and check the intent of the various aspects of the type.

- Open the starter project in
  PathToCourseFiles`\Labs\Lab 1.1\Begin` ,

which contains a project called `DataStructures` with `Sequence` and `Node` types representing a generic linked list implementation.

Your task is to make it compliant with the new nullable standard for reference types.

- Enable nullability checks for the project and activate *“Treat warnings as errors”* for all in the project properties.
- Try to figure out how the incurring types are thought to work internally.
- Decorate the types appropriately with ? and `!`to make it compile and run correctly.

Lab 1.2: "Playing with Pattern Matchings" (★)

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

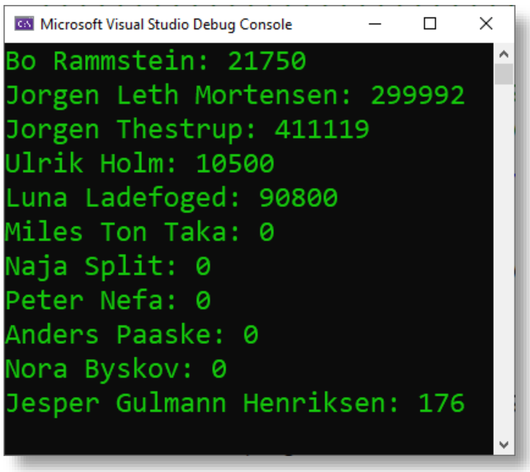
- Open the starter project in
  PathToCourseFiles`\Labs\Lab 1.2\Begin` ,
  which contains a project called `PatternMatching` with `Employee` data supplied in `Data`.

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.



```
Bo Rammstein: 21750
Jorgen Leth Mortensen: 299992
Jorgen Thestrup: 411119
Ulrik Holm: 10500
Luna Ladefoged: 90800
Miles Ton Taka: 0
Naja Split: 0
Peter Nefa: 0
Anders Paaske: 0
Nora Byskov: 0
Jesper Gulmann Henriksen: 176
```

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 1.3: ”Employee Records”

This exercise investigates the connection between classes and records.

- Open the starter project in
  PathToCourseFiles`\Labs\Lab 1.3\Begin` ,
  which contains a project containing the well-known `Employee` classes.

The task at hand is to convert this class hierarchy to records instead of classes.

- Convert all the classes of the `Employee` hierarchy to records.
  - Maintain the conceptual intent of records by making the records immutable even if the corresponding class is not.
- Locate the first TO-DO in `Program.cs` and use pattern matching to populate `search` with all `StudentProgrammer`s mentored by a `SoftwareEngineer` with these constraints:
  - Own first name contains at least 4 characters
  - Mentor has not written between 100.000 and 400.000 lines of code.

Finally;

- Locate the second TO-DO in `Program.cs` and populate `haveNewMentor` with the above `StudentProgrammer`s where they have their mentor changed to Bo Rammstein.

Part 2: “Removing Overhead and Adding Completion in C# 9 and 10”

Lab 2.1: ”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`\Labs\Lab 2.1\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 List<Movie>
{
    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.

- 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 2.2: ”Refactor and Modernize to C# 10”

We will start by applying the newly discovered syntax improvements to simplify an existing C# 9 program.

- Open the starter project in
  PathToCourseFiles`\Labs\Lab 2.2\Begin` ,
  which contains a project called `Modernizing`.

It contains what is apparently a C# 9 project with a number of files, types, namespaces, and other syntactic constructs all written in an old-school and often inappropriate manner. It does not make much use of the C# 7, 8, 9, and 10 ways of making the programs safer, better, and more readable. Moreover, while it seems to be functional, it still produces some warnings.

- Inspect and run the code to figure out what the program does.

Your task is now to correct, improve, beautify etc. the program using the state-of-the-art features that you know, but the original developer probably didn’t.

Do your best to fix the warnings and use techniques from C# 7.x, 8, 9, and 10 such as

- Global and implicit usings
- File-scoped namespaces
- Records
- ...

to improve the program, while maintaining its functionality.

Extra Labs: “If You Feel You Need More Challenge…”

Lab 01.5: ”Maximum Subsum Problem” (★★★)

In this brain teaser you will employ a crucial new C# feature inside a LINQ statement to solve the maximum subsum problem, which is a thoroughly studied algorithmic problem within the theory of computer science.

Any finite sequence, *s*, of integers of length *n*, has a number of distinct subsequences of length at most *n*.

As an example, consider the sequence

```
2, -3, 7, 1, 4, -6, 9, -8
```

Here, the subsequences include (among many, many others) e.g.

```
2, -3, 7, 1, 4
-3, 7
-6
…
```

Note that the empty sequence as well as the entire original sequence are both legal subsequences.

Each such subsequence can be viewed as defining a subsum of *s* obtaining by adding all the integers of the subsequence. The subsums for the example subsequences above are, resp.:

```
2 + (-3) + 7 + 1 + 4  = 11
-3 + 7                =  4
-6                    = -6
```

Note: The sum of the empty sequence is 0.

The maximum subsum for the example sequence above is 15 – illustrated by the sequence highlighted in green.

And with that, let’s finally get on to the exercise itself..!

- Open the starter project in
  PathToCourseFiles`\Labs\Lab X.1\Begin` ,
  which contains a project called `MaximumSubsumProblem`.

In the starter project you will find the following code:

```
IEnumerable<int> sequence = new List<int> { 2, -3, 7, 1, 4, -6, 9, -8 };

// TODO
int result = ...;

Console.WriteLine( $"Maximum subsum is {result}");
```

Your task is to replace the “`...`“ with a **single** (but relatively complex) LINQ statement computing the maximum subsum of the sequence.

In more detail,

- Create a single LINQ query computing the maximum subsum of a specific sequence supplied as a `IEnumerable<int>`
- Use a new C# type feature inside of the single LINQ query as computational state
- Make sure you compute the maximum subsum in linear time (in the length of the sequence).

Uncle Google is probably your friend here... 😉