”Cutting-Edge C# 8+9”

Lab Manual

Wincubate ApS

02-10-2021

V2.0

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 2019 version 16.8 (or later) with .NET Core 3.1 and .NET 5 or later installed on the PC.

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

Module 1: "An Introduction to C# 8"

Lab 01.1: "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\Module 01\Lab 01.1\Starter` ,

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 01.2: "Playing with Pattern Matchings" ()

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

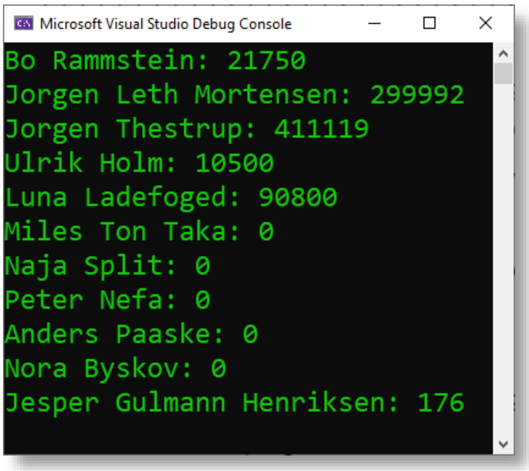
- Open the starter project in
  PathToCourseFiles`\Labs\Module 01\Lab 01.2\Starter` ,
  which contains a project called `PatternMatching` with `Employee` data supplied in `Data`.

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 01.3: ”Pattern Matching Recursive Types” (⭐⭐)

This lab extends our treatment of pattern matchings to processing of recursive types.

- Open the starter project in
  PathToCourseFiles`\Labs\Module 01\Lab 01.3\Starter` ,
  which contains a project called `PatternMatchingExpressions`.

The project contains a set of simple integer expression types for producing abstract syntax trees for simple arithmetic expressions over integers. More precisely, the following types are defined:

i. `SimpleExpression`
ii. `Integer`
iii. `Negative`
iv. `Add`
v. `Multiply`.

Firstly;

- Inspect the types in the source code and get a feeling for the connection between the various types.

The Program.cs file contains an expression of type `SimpleExpression` initialized as follows:

```
SimpleExpression expression = new Add(
    new Negative(
        new Integer(-176)
    ),
    new Add(
        new Integer(-42),
        new Multiply(
            new Integer(1),
            new Integer(87)
        )
    )
);
```

Use Pattern Matching to display expressions

Unfortunately, we have no way of displaying such an expression to the console. So we need to complete the extension method `Display()` in the `SimpleExpressionExtensions` class. A simple way of outputting the expression above to the console would be to compute and print the following display string:

```
(-(-176))+((-42)+((1)*(87)))
```

With this definition in mind;

- Locate the `TODO: Complete Display()` in the code.
- Use pattern matching to complete the `Display()` method.
- Test that it produces the output above.

It turns out that the solution can be expressed quite neatly using pattern matching.

Use Pattern Matching to evaluate expressions

In a similar vein, let's produce an evaluation method for `SimpleExpression`. Using standard arithmetic rules we would expect the expression printed above to evaluate to:

```
221
```

Consequently;

- Locate the `TODO: Complete Evaluate()` in the code.
- Use pattern matching to complete the `Evaluate()` method.
- Test that it produces the output above.

Use Positional Pattern Matching to create a better display of expressions

While the display string

```
(-(-176)))+((-42)+((1)*(87)))
```

is simply to produce, it does lend itself to a number of rather trivial optimizations. For instance, we could probably eliminate a few of the unnecessary parenthesis:

a) `-(-176))` could be reduced to `176` .
b) `(87) * (`*`expression`*`)` could be reduced to `87*(`*`expression`*`)` (or in the opposite order)

Furthermore;

c) `0 +` *`expression`* could be reduced to *`expression`* (or in the opposite order)
d) `0 *` *`expression`* could be reduced to `0` (or in the opposite order)
e) `1 *` *`expression`* could be reduced to *`expression`* (or in the opposite order)

As an example, one might reduce

```
(-(-176)))+((-42)+((1)*(87)))
```

to something along the lines of

```
(176)+(-42+87)
```

or perhaps even simpler (depending upon exactly how much effort you put into this endavour).

Such optimizations lend themselves to the use of *Positional Pattern Matching*.

- Figure out how to extend the type hierarchy to enable Positional Pattern Matching.

With that is in place, we can proceed to:

- Locate the `TODO: Complete BetterDisplay()` in the code.
- Use Positional Pattern Matching to complete the `BetterDisplay()` method.
    - Note: You are free to implement as many additional optimizations as you like! 😊

Lab 01.4: "Indices and Ranges for Custom Types" (★★★)

This advanced lab will investigate how to add indices and range manipulation of our own types.

- Open the starter project in
  PathToCourseFiles`\Labs\Module 01\Lab 01.4\Starter` ,
  which contains a project called `CustomIndicesAndRanges`.

The project contains a custom generic class called `SequencePacker<T>`, which stores sequences of elements of type T in a compressed form. More precisely, the sequence

```
42 87 87 87 87 11 22 22 87 99
```

is stored internally as a list of `Node<T>` elements as follows:

(42,1) (87,4) (11,1) (22,2) (87,1) (99,1).

`SequencePacker<T>` also implements `IEnumerable<T>` such the original uncompressed sequence is produced whenever iterating over the packed sequence.

- Inspect the source code and investigate how the `SequencePacker<T>` works.

Implement the new C# 8 indices for SequencePacker<T>

You have fallen in love with the shiny new way of using indices in C# and would like to extend the syntax to `SequencePacker<T>`. For instance, you would want the following to compile, run, and produce the expected results:

```
Console.WriteLine(sp[4]);  // == 87
Console.WriteLine(sp[^4]); // == 22
```

- Implement `get` Index support for `SequencePacker<T>`
  - **Don't implement the set!**

Note: You might obtain interesting information here: https://docs.microsoft.com/en-us/dotnet/csharp/tutorials/ranges-indexes#type-support-for-indices-and-ranges 😊

Implement the new C# 8 ranges for SequencePacker<T>

You're on a roll! Now implement support for ranges as well…

More precisely, with the following definition in place:

```
SequencePacker<int> sp =
  new SequencePacker<int>{ 42, 87, 87, 87, 87, 11, 22, 22, 87, 99 };
```

in place, you want to have the following results:

```
sp[..]    // == 42 87 87 87 87 11 22 22 87 99
sp[2..^3] // ==       87 87 87 11 22
sp[2..]   // ==       87 87 87 11 22 22 87 99
```

- Implement `get` Range support for `SequencePacker<T>`
    - **Don't implement the `set`!**

If you're bored…

How difficult is it to implement the setters for Index and Range?

If you do implement it, you should probably also include a `Remove()` method on the `SequencePacker<T>`itself.

Module 2: "What's New in C# 9?"

Lab 02.1: "Employee Records"

This exercise investigates the connection between classes and records.

- Open the starter project in
  PathToCourseFiles`\Labs\Module 02\Lab 02.1\Starter` ,
  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.

Lab 02.2: "Dictionaries and Records" (★)

This exercise illustrates a simple, but neat trick for composite keys in dictionaries.

- Open the starter project in
  PathToCourseFiles`\Labs\Module 02\Lab 02.2\Starter` ,
  which contains a project called `KeyToAwesomeness`.

The project defines two enumeration types

```
enum CoffeeKind
{
    Latte,
    Cappuccino,
    Espresso
}
```

and

```
enum CoffeeSize
{
    Small,
    Regular,
    Large
}
```

A coffee consists of a `CoffeeKind`, a `CoffeeSize`, and a strength between 1 and 5. The `Main()` method contains some very simple code serving 100 random coffees to random customers:

```
void Serve( string customerName, CoffeeKind kind, CoffeeSize size,
            int strength )
{
    Console.WriteLine($"Serving a {size} {kind} of strength {strength}
         to {customerName}");
}

RandomHelper helper = new RandomHelper();

for (int i = 0; i < 100; i++)
{
    CoffeeKind kind = helper.GetRandomCoffeeKind();
    CoffeeSize size = helper.GetRandomCoffeeSize();
    int strength = helper.GetRandomCoffeeStrength();

    Serve(helper.GetRandomName(), kind, size, strength);
}

Console.WriteLine();
```

When run the program produces a number of output lines like the following:

```
C:\WINDOWS\system32\cmd.exe
Serving a Regular Cappuccino of strength 1 to Ane Olsen
Serving a Small Cappuccino of strength 2 to Maria Sana
Serving a Large Latte of strength 5 to Nils Christensen
Serving a Large Espresso of strength 1 to Nils Gulmann
Serving a Small Latte of strength 5 to Ane Riel
Serving a Regular Latte of strength 5 to Bo Mortensen
Serving a Small Cappuccino of strength 4 to Noah Leth
Serving a Small Cappuccino of strength 2 to Nina Kirk
Serving a Regular Latte of strength 3 to Jesper Thomassen
Serving a Small Espresso of strength 2 to Jørgen Olsen
Serving a Regular Latte of strength 1 to Nina Thomassen
Serving a Large Cappuccino of strength 2 to Heidi Kirk
Serving a Small Espresso of strength 1 to Bo Henriksen
Serving a Small Latte of strength 3 to Maria Gulmann
```

However, the coffee shop would like to print a summary of all the coffee served, i.e. how many coffees were served of each combination of a `CoffeeKind`, a `CoffeeSize`, and a strength.

They would like to augment the program with a `PrintSummary()` method which provides a summary like:

```
C:\WINDOWS\system32\cmd.exe
Served 6 Regular Latte of strength 4
Served 5 Large Latte of strength 2
Served 4 Small Latte of strength 3
Served 4 Regular Cappuccino of strength 5
Served 4 Regular Cappuccino of strength 2
Served 4 Regular Espresso of strength 3
Served 3 Large Latte of strength 5
Served 3 Small Latte of strength 2
Served 3 Large Cappuccino of strength 3
Served 3 Large Cappuccino of strength 1
Served 3 Regular Cappuccino of strength 4
Served 3 Regular Cappuccino of strength 1
Served 3 Small Cappuccino of strength 5
```

Your task will be to produce this result in a simple manner.

- Augment the `Serve()` method with a means for counting how many coffees were served for each combination of kind, size, and strength

- Define a `PrintSummary()` method outputting a number of strings to the console as illustrated
  - Sort first by the count of coffees served (from high to low)
  - Sort secondly by kind (from first to last)
  - Sort thirdly by size within that kind (from largest to smallest)
  - Use the strength as the final sort criterion (from strongest to weakest).

Lab 02.3: "Url Content Fetching" (★★★)

This advanced lab will investigate how to add advanced functionality to plain, old lists.

- Open the starter project in
  PathToCourseFiles`\Labs\Module 02\Lab 02.3\Starter` ,
  which contains a project called `Extension Async Enumerables`.

The project contains top-level statements which do **not** currently compile:

```
using System;
using System.Collections.Generic;
using Wincubate.CS9.ExtensionAsyncEnumerableLab;

List<string> urls = new()
{
    "http://www.dr.dk",
    "http://www.jp.dk",
    "http://www.bold.dk"
};

await foreach (var urlResult in urls)
{
    Console.ForegroundColor = ConsoleColor.Yellow;
    Console.WriteLine($"=== [{urlResult.Url}] ===");
    Console.ResetColor();

    Console.WriteLine(urlResult.Contents.Substring(0, 240));
}
```

Your task is now to

- Locate the TO-DO in the ListExtensions.cs file.
- Replace the TO-DO with code to make the program compile and produce a result similar to the screenshot shown below
    - Do not change anything in `Program.cs`! 😊

```
=== [http://www.dr.dk] ===

    <!DOCTYPE html>
    <html dir="ltr" lang="da"><head><meta charSet="utf-8"/><meta name="viewport" conte
nt="width=device-width, initial-scale=1.0"/><script>
              (function() {
                function logElementsError(event) {

=== [http://www.jp.dk] ===


<!DOCTYPE html>
<html lang="da">
    <head>

<script class="consent-independent" type="application/javascript">
    window.commonJpData = JSON.parse("\u007b\u0022anonId\u0022\u003a\u002299999999\u00
2d9999\u002d9999\u002d9999\u002d
=== [http://www.bold.dk] ===
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
        "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">

<html xmlns="http://www.w3.org/1999/xhtml">
<head>

<!--Pubstack-->
```