

# "Cutting-Edge C# 13"

---

## Lab Manual

Wincubate ApS

22-11-2024



V13.0

## Table of Contents

|   |    |
|---|----|
| Exercise types .....  | 4  |
| Prerequisites.....  | 4  |
| Module 1: "Very Quick Recap of C# 7.x" (Additional Exercises) .....             | 5  |
| Lab 01.1: "Basic Tuples" .....  | 5  |
| Lab 01.2: "Object Deconstruction" (  ) .....                                    | 6  |
| Lab 01.3: "Pattern Matching Shapes" (  ) .....                                  | 8  |
| Lab 01.4: "More Tuples, Pattern Matching, and a Local Function" (  ) .....      | 10 |
| Lab 01.5: "Maximum Subsum Problem" (    ) .....                                 | 12 |
| Lab 01.6: "Avoiding Copying of Value Types" (    ) .....                        | 14 |
| Module 2: "An Introduction to C# 8" .....                                       | 15 |
| Lab 02.1: "Adding Nullability to Reference Types" .....                         | 15 |
| Lab 02.2: "Playing with Pattern Matchings" (  ) .....                           | 16 |
| Write the Code Production Index for each employee.....                          | 16 |
| Find all Student Programmers mentored by a Chief Software Engineer .....        | 16 |
| Lab 02.3: "Pattern Matching Recursive Types" (   ) .....                        | 17 |
| Use Pattern Matching to display expressions .....                               | 17 |
| Use Pattern Matching to evaluate expressions.....                               | 18 |
| Use Positional Pattern Matching to create a better display of expressions ..... | 18 |
| Lab 02.4: "Indices and Ranges for Custom Types" (    ) .....                    | 19 |
| Implement the new C# 8 indices for SequencePacker<T> .....                      | 19 |
| Implement the new C# 8 ranges for SequencePacker<T> .....                       | 19 |
| If you're bored... .....  | 20 |
| Module 3: "An Introduction to C# 9" .....                                       | 21 |
| Lab 03.1: "Employee Records" .....  | 21 |
| Lab 03.2: "Dictionaries and Records" (  ) .....                                 | 22 |
| Lab 03.3: "Url Content Fetching" (    ) .....                                   | 25 |
| Module 4: "An Introduction to C# 10" .....                                      | 27 |
| Lab 04.1: "Refactor and Modernize to C# 10".....                                | 27 |
| Lab 04.2: "LINQ Additions in .NET 6" (  ) .....                                 | 28 |
| Lab 04.3: "Developer-centric Logs" (  ).....                                    | 29 |
| Lab 04.4: "Writing Custom String Interpolation Handlers" (    ).....            | 31 |

|  |    |
|--|----|
| Module 5: "What's New in C# 11?" .....   | 32 |
| Lab 05.1: "Palindromes and List Patterns" (★).....                                 | 32 |
| Lab 05.2: "Required and String Interpolated Members" (★) .....                     | 33 |
| Lab 05.3: "Better Developer-centric Logs" .....                                    | 35 |
| Module 6: "Newest Features in C# 12" .....   | 36 |
| Lab 06.1: "Matching, Merging, and Spreading with Collection Expressions" (★) ..... | 36 |
| Lab 06.2: "Investigating Primary Constructors" .....                               | 37 |
| Module 7: "Brand New C# 13" .....  | 38 |

## 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, 7, 8, and 9 installed on the PC.

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

## Module 1: "Very Quick Recap of C# 7.x" (Additional Exercises)

These labs are not part of the agenda of the course, but merely meant to be used for recapping any missing features of C# 7.x in your own time after the course.

### Lab 01.1: "Basic Tuples"

This exercise implements a simple function using tuples for computing compound values.

- Open the starter project in  
`PathToCourseFiles\Modules\01\Labs\Lab 01.1\Begin` ,  
which contains a project called `BasicTuples`.

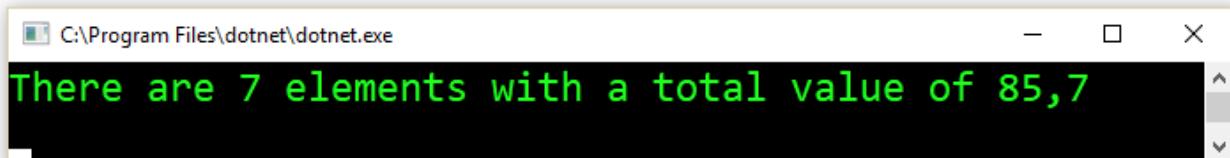
The `Main()` method contains the following code:

```
IEnumerable<decimal> numbers = new List<decimal>
{
    4.2m, 8.7m, 17.6m, 11.2m, 25.5m, 7.5m, 11.0m
};

// TODO

Console.WriteLine( $"There are {count} elements with a total value
of {total}");
```

You should use tuples to complete the TODO above such that the program will print the following when run:



A screenshot of a terminal window titled "C:\Program Files\dotnet\dotnet.exe". The window displays the text "There are 7 elements with a total value of 85,7".

You should complete your task by

- Defining a method `Tally()`
  - accepting an argument of `IEnumerable<decimal>` and
  - returning an appropriate tuple type
- Replace the TODO with just a single line invoking the `Tally()` method and handling the return values appropriately with changing any other line in `Main()`.

## Lab 01.2: "Object Deconstruction" (★)

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

- Open the starter project in  
*PathToCourseFiles\Modules\01\Labs\Lab 01.2\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 class `Album`. That class is part of an externally supplied API and cannot be modified or derived from:

```
public sealed class Album
{
    public Guid Id { get; }
    public string Artist { get; }
    public string AlbumName { get; }
    public DateTime ReleaseDate { get; }

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

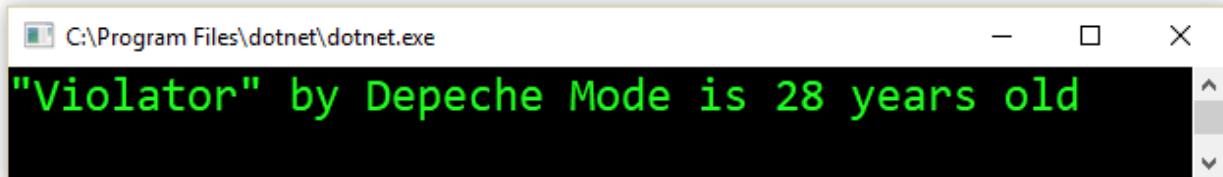
The client project contains a `Main()` method with the following code:

```
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 `Main()` method or the `Album` class.
- When completed, your `Main()` method should produce the following output:



## Lab 01.3: "Pattern Matching Shapes" (★)

The purpose of this exercise is to use pattern matching for distinguishing shapes and computing their respective areas.

- Open the starter project in  
`PathToCourseFiles\Modules\01\Labs\Lab 01.3\Begin`,  
which contains a project called MatchingShapes.

The project contains two predefined shape structs. Circle.cs contains

```
struct Circle
{
    public double Radius { get; }

    public Circle( int radius ) => Radius = radius;
}
```

and Rectangle.cs has this definition:

```
struct Rectangle
{
    public double Width { get; }
    public double Height { get; }

    public Rectangle( int width, int height )
    {
        Width = width;
        Height = height;
    }
}
```

The Main() method contains the following data and method call:

```
List<object> objects = new List<object>
{
    new Circle( 3 ),
    null,
    new Rectangle( 4, 5 ),
    "Not really a shape",
    new Rectangle( 6, 7 ),
    new Circle( 8 )
};

objects.ForEach(ComputeArea);
```

Your task is to use pattern matching to figure out which area calculation is to be employed for each particular object.

- Complete the ComputeArea() method in Program.cs appropriately such that the program will produce the following output:

```
C:\Program Files\dotnet\dotnet.exe
Area of circle with radius 3 is 28,2743338823081
Please remove null values..!
Area of rectangle with width 4 and height 5 is 20
Object of type System.String was ignored
Area of rectangle with width 6 and height 7 is 42
Area of circle with radius 8 is 201,061929829747
```

## Lab 01.4: "More Tuples, Pattern Matching, and a Local Function" (★)

This exercise extends the solution of Lab 01.1 to implement processing of recursive sequences of numbers using a pattern matching technique and a local function to assist it.

- Open the starter project in  
*PathToCourseFiles\Modules\01\Labs\Lab 01.4\Begin*,  
which is essentially the solution to Lab 01.1. containing a project called MoreTuples.

However, now the Main() method contains the following code:

```
IEnumerable<object> numbers = new List<object>
{
    4.2m, 8.7m, new object[]{ 17.6m, 11.2m, 25.5m },
    7.5m, new List<object>{ 11.0m }
};

var (count, total) = Tally(numbers);
Console.WriteLine($"There are {count} elements with a total value
of {total}");
```

The data has now been generalized from `IEnumerable<decimal>` to `IEnumerable<object>`, and the data is now “recursive” in the sense that some elements of the object sequence are – in turn – an object sequence. Apart from that, Main() has not been modified.

The signature of the Tally() method you completed in Lab 01.1 has been generalized accordingly as follows:

```
static (int count, decimal total) Tally( IEnumerable<object> data )
{
    (int count, decimal total) tuple = (0, 0);

    // TODO

    return tuple;
}
```

Your task is now to use pattern matching and a local helper function to complete the updated Tally() method to correctly recursively compute the tally of the sequence.

- Within Tally() introduce a local helper function called Update() which updates the local tuple variable by adding a subcount and a subtotal to the constituent tuple values.
- Use a switch statement with pattern matching to process the sequence (and subsequence ) elements
  - Use Update() in each of the cases to update tuple.

As before, your completed program should produce the following output:

```
C:\Program Files\dotnet\dotnet.exe
There are 7 elements with a total value of 85,7
```

## Lab 01.5: "Maximum Subsum Problem" (★★★)

In this brain teaser you will employ tuples 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.:

$$\begin{array}{rcl} 2 + (-3) + 7 + 1 + 4 & = & 11 \\ -3 + 7 & = & 4 \\ -6 & = & -6 \end{array}$$

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\Modules\01\Labs\Lab 01.5\Begin* ,  
which contains a project called MaximumSubsumProblem.

In the starter project you will find the following code inside of `Main()`:

```
IEnumerator<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 containing tuples to compute 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 tuples 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... 😊

## Lab 01.6: "Avoiding Copying of Value Types" (★★★)

This exercise is concerned with using the new features for achieving performance for value types C# 7.x.

- Open the starter project in  
*PathToCourseFiles\Modules\01\Labs\Lab 01.6\Begin* ,  
which contains a project called EffectiveValueType.

In the supplied solution, somebody has implemented an Abstract Factory pattern for `Coffee` instances as follows:

```
interface ICoffeeFactory
{
    Coffee CreateCoffee( CoffeeType coffeeType );
}
```

However, these `Coffee` elements are value types, so while the factory creates and caches these values, it produces a large number of copying of value types.

- Please take a second to ponder: Why is that?

Your task is to

- modify the factory (and if necessary also Program.cs) to eliminate as much unnecessary copying of values as you can
  - You can use any of the new C# 7.x features that you see fit.

Feel free to eliminate any number of allocations as well. 😊

## Module 2: “An Introduction to C# 8”

### Lab 02.1: “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\Modules\02\Labs\Lab 02.1\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 02.2: "Playing with Pattern Matchings" (★)

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

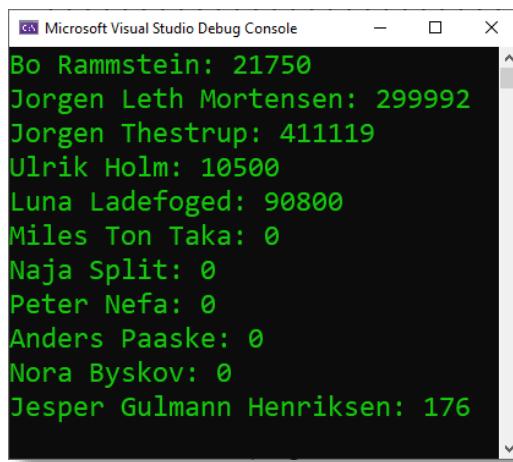
- Open the starter project in  
`PathToCourseFiles\Modules\02\Labs\Lab 02.2\Begin` ,  
which contains a project called `PatternMatching` with `Employee` data supplied in `Data`.

### 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.



Microsoft Visual Studio Debug Console

```
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 02.3: "Pattern Matching Recursive Types" (★★)

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

- Open the starter project in  
*PathToCourseFiles\Modules\02\Labs\Lab 02.3\Begin* ,  
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 endeavour).

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 02.4: "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\Modules\02\Labs\Lab 02.4\Begin* ,  
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 3: “An Introduction to C# 9”

### Lab 03.1: "Employee Records"

This exercise investigates the connection between classes and records.

- Open the starter project in  
*PathToCourseFiles\Modules\03\Labs\Lab 03.1\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 `StudentProgrammers`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 03.2: "Dictionaries and Records" (★)

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

- Open the starter project in  
*PathToCourseFiles\Modules\03\Labs\Lab 03.2\Begin* ,  
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:\ 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:\ 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 03.3: "Url Content Fetching" (★★★)

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

- Open the starter project in  
*PathToCourseFiles\Modules\03\Labs\Lab 03.3\Begin* ,  
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! 😊



## Module 4: “An Introduction to C# 10”

### Lab 04.1: “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\Modules\04\Labs\Lab 04.1\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.

## Lab 04.2: "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\Modules\04\Labs\Lab 04.2\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 04.3: "Developer-centric Logs" (★)

This lab will investigate how the latest addition to the caller info attributes; [CallerArgumentExpression](#).

- Open the starter project in  
*PathToCourseFiles\Modules\04\Labs\Lab 04.3\Begin* ,  
which contains a project called LINQ Extension.

The project contains a sequence of integer numbers already defined along with three queries over these numbers, which are to eventually produce the three distinct results as described in the comments.

- Your task is to implement a proper LINQ-style extension method called `Sample<T>` which
  - Extends `IEnumerable<T>` (as the LINQ methods all do)
  - Accepts a frequency parameter indicating how often to sample the sequence
  - Returns the input sequence sampled at every frequency element
  - Throws an `ArgumentException` in case input sequence contains less than frequency elements
    - The exception message should contain the expression producing the too short input sequence being erroneously sampled.

Some specification examples are specified below.

1. The following code produces a copy of the entire sequence:

```
var query = numbers
    .Sample(1)
    ;
```

2. The following code produces the subsequence 42, 176, 176, 112, 24, 42:

```
var query = numbers
    .Where(i => i % 2 == 0)
    .Sample(5)
    ;
```

3. The following code throws `ArgumentException` when evaluated with the exception message specified above, e.g.

```
System.ArgumentException: Expression doesn't have enough
elements:
numbers
    .Where(i => i < 0) (Parameter 'sequence')

var query = numbers
    .Where(i => i < 0)
    .Sample(5)
    ;
```

These code snippets are included in the Starter project for you to comment in and run.

- Test your implementation on these code snippets.

## Lab 04.4: "Writing Custom String Interpolation Handlers" (★★★)

In the presentation we touched upon how string interpolation has been improved in C# 10 and .NET 6.

We also mentioned that the API for creating string interpolation handlers is now open and can be extended by anyone willing to dig into the API. Microsoft has supplied a hands-on guide illustrating how to do so.

If you're interested,

- Browse to <https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/tutorials/interpolated-string-handler>
- Follow the tutorial step-by-step to develop and activate a custom string interpolation handler for a logger.

## Module 5: “What’s New in C# 11?”

### Lab 05.1: “Palindromes and List Patterns” (★)

This lab investigates how to use list patterns with recursive methods to compute properties of strings in a functional programming manner.

- Open the starter project in  
*PathToCourseFiles\Modules\05\Labs\Lab 05.1\Begin* ,

which contains a project called `List Patterns`.

A palindrome is a string of characters which reads identically both forwards and backwards, i.e. the string is “identical when mirrored”. Just to make it more fun, we will disregard the casing of the characters.

The project defines three strings which are to be tested for palindrome-ness:

```
string s1 = "VoksneIrererDividererIEnSkov";
string s2 = "Otto";
string s3 = "NotAPalindrome";
```

With the definition outline above for these string definitions, we would expect the following results when tested:

- ❖ `s1` is a palindrome
- ❖ `s2` is a palindrome
- ❖ `s3` is not a palindrome

Your task is now:

- Locate the TODO in the source code in `Program.cs`
- Implement the `IsPalindrome` method correctly to produce the above results by applying various list and slice patterns appropriately.

## Lab 05.2: "Required and String Interpolated Members" (★)

This lab dives into both required members and the new string interpolation syntax.

- Open the starter project in  
*PathToCourseFiles\Modules\05\Labs\Lab 05.2\Begin* ,

which contains a project called Required.

The project contains a class **User** defined as follows:

```
class User
{
    public Guid UserId { get; init; }
    public string FullName { get; set; }
    public string Company { get; set; }
    public bool IsActive { get; set; }

    public User()
    {
    }

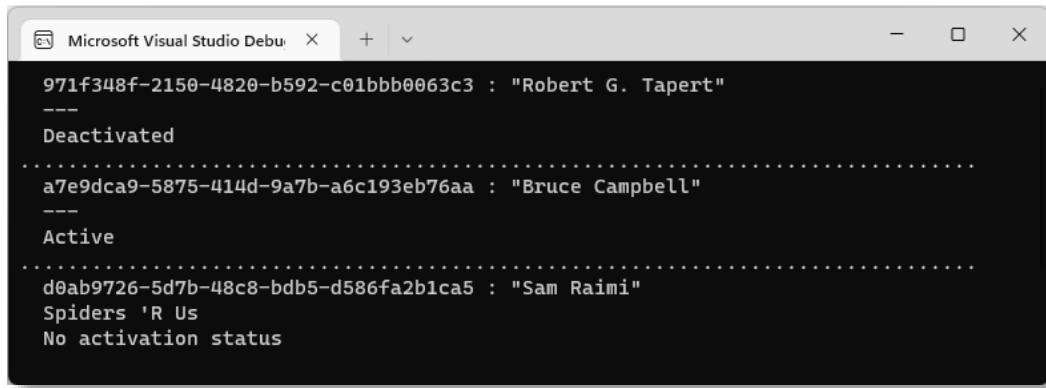
    // TODO: a)
    public User(string fullName)
    {
        UserId = Guid.NewGuid();
        FullName = fullName;
    }

    // TODO: b)
    public override string ToString() => string.Empty;
}
```

- Locate the two **TODO: a1**) and **TODO: a2**) in the code.
  - Decorate and/or change the members to make the correct properties required (or not) and nullable (or not) so that it fits the following specification:
    - In the code there are three **User** object instantiations which should compile: **user1**, **user2**, and **user3**.
    - Moreover, there are three other **User** object instantiations commented out which should **not** compile: **user4**, **user5**, and **user6**.
  - Note: You should not change the contents or parameters of the constructors, nor add any more constructors 😊 .

Now having adjusted the types and decorations to meet the specifications above, we should go ahead and print the **user1**, **user2**, and **user3** nicely by adjusting the implementation of **ToString()** inside of **User**.

We would like the printout to look as follows in the console window:



The screenshot shows a Microsoft Visual Studio Debug window with the title bar "Microsoft Visual Studio Debug". The content area displays the following text:

```
971f348f-2150-4820-b592-c01bbb0063c3 : "Robert G. Tapert"
---
Deactivated
-----
a7e9dca9-5875-414d-9a7b-a6c193eb76aa : "Bruce Campbell"
---
Active
-----
d0ab9726-5d7b-48c8-bdb5-d586fa2b1ca5 : "Sam Raimi"
Spiders 'R Us
No activation status
```

- Locate the **TODO: b)** in the code.
  - Use string interpolation to implement `ToString()` appropriately
  - Note: Pay particular attention to the **two-space indentation** in each line of the printout.

## Lab 05.3: "Better Developer-centric Logs"

This lab will investigate how the solution to Lab 04.3 can be made a bit nicer in terms of type-safety and rename-maintainability of the code.

- Open the starter project in  
*PathToCourseFiles\Modules\05\Labs\Lab 05.3\Begin* ,  
which contains the solution code from Lab 04.3.

C# 11 offers a more powerful **nameof** construction, which we really should apply here...! 😊

- Locate the **Sample<T>()** method in the code and change it appropriately.
- Run the program and test that the program has the same functionality.

## Module 6: “Newest Features in C# 12”

### Lab 06.1: ”Matching, Merging, and Spreading with 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\Modules\06\Labs\Lab 06.1\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
```

## Lab 06.2: "Investigating Primary Constructors"

This lab will illustrate primary constructors and different use cases for them,

- Open the starter project in  
*PathToCourseFiles\Modules\06\Labs\Lab 06.2\Begin* ,  
which contains some bank account-related types.

The code contains the following type which you should use and potentially extend:

```
readonly record struct AccountNumberType(string AccountNumber)
{
    public override string ToString() =>
        AccountNumber.ToString();
}

[Serializable]
public abstract class BankException : Exception
{
    public BankException(
        string? message = null, Exception? inner = null)
        : base(message, inner)
    {
    }
}

abstract class BankAccount
{
    public AccountNumberType Number { get; }
    public decimal Balance { get; private set; }

    public void Deposit(decimal amount)
    {
        Balance += amount;
    }
}
```

Your tasks are now:

- Complete the **BankAccount** as follows:
  - Define a primary constructor which accepts
    - **string accountNumber**
    - **decimal initialBalance**
- The **Balance** property should throw a **BalanceException** when the initial balance is negative:
  - Create the **BalanceException** class deriving from **BankException**
    - using primary constructors for both classes
  - Make sure the new exception is thrown when negative
- Define a new class **CheckingAccount** deriving from **BankAccount** as follows:
  - Define a primary constructor which accepts
    - **decimal initialBalance**

- The account numbers should be autogenerated in the class:
  - The first account should have account number 10000000,
  - The second account should have account number 10000001,
  - Etc.
- Add a method `public void Withdraw(decimal amount)`
  - The method should throw a `BalanceException` when `Balance` is less than the amount – and otherwise update `Balance`.

The `Program.cs` file contains

```
var checking1 = new CheckingAccount(42);
Console.WriteLine(checking1.Number);           // Should print
"10000000"
var checking2 = new CheckingAccount(0);
Console.WriteLine(checking2.Number);           // Should print
"10000001"

var checking3 = new CheckingAccount(-87);      // Should throw
                                              // BalanceException
```

- Finally, when you run the program, make sure that it behaves as described in the comments above.

## Module 7: “Brand New C# 13”

None.