”C# and .NET – From 4.8 Framework to .NET 7”

Lab Manual

Wincubate ApS

25-02-2024

V1.1

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 differ 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!) 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:\DotNet7
```

with the latest version of Visual Studio 2022 (or Rider) with .NET 6 and 7 installed on the PC.

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

Lab Session 1: “C# 7 and 8”

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\Part 1\Lab 1.1\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 1.2: "Playing with Pattern Matchings" (★)

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

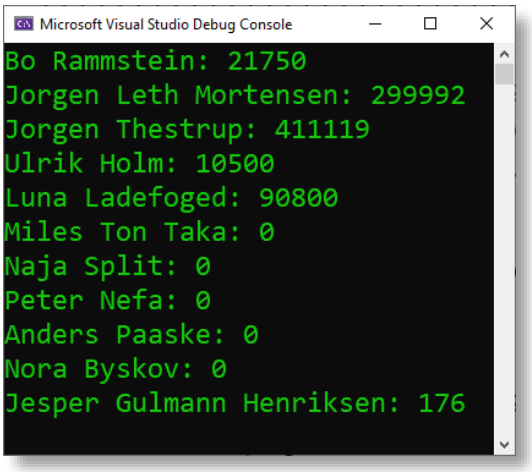
- Open the starter project in
  PathToCourseFiles\Labs\Part 1\Lab 1.2\Starter ,
  which contains a project called `PatternMatching` with `Employee` data supplied in `Data`.

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 Session 2: "C# 9 and 10"

Lab 2.1: "Employee Records"

This exercise investigates the connection between classes and records.

- Open the starter project in
  PathToCourseFiles`\Labs\Part 2\Lab 2.1\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:
  - Student Programmer's 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 2.2: "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\Labs\Part 2\Lab 2.2\Starter ,
  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 Session Extras: "C# 7, 8, 9, 10, and 11"

Lab X.1: "Basic Tuples"

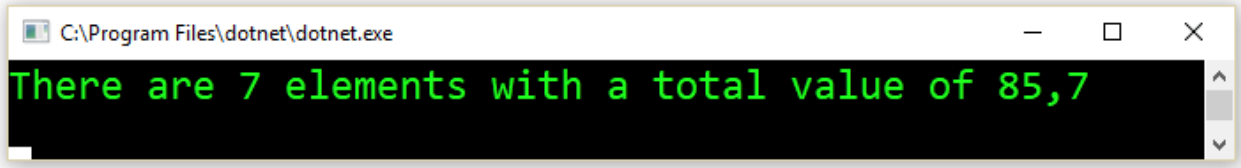
This exercise implements a simple function using tuples for computing compound values.

- Open the starter project in
  PathToCourseFiles\Labs\Extras\Lab X.1\Starter ,
  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:



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 X.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\Labs\Extras\Lab X.2\Starter ,
  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:

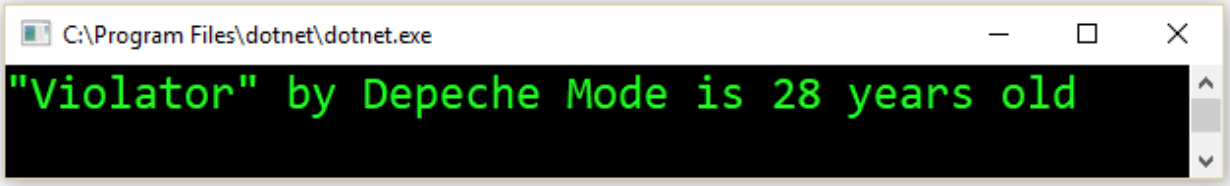
```
public sealed class Album
{
    public Guid Id { get; }
    public string Artist { get; }
    public string AlbumName { get; }
    public DateTime ReleaseDate { get; }

    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:


C:\Program Files\dotnet\dotnet.exe
"Violator" by Depeche Mode is 28 years old

Lab X.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\Labs\Extras\Lab X.3\Starter ,
  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:

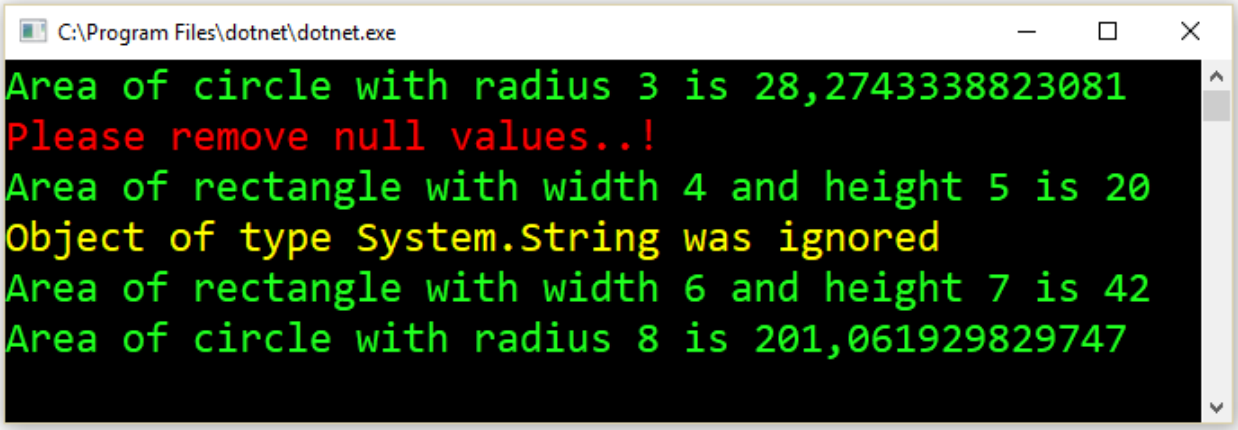
```
struct Rectangle
{
    public double Width { get; }
    public double Height { get; }

    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 X.4: "More Tuples, Pattern Matching, and a Local Function" (★)

This exercise extends the solution of Lab X.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\Labs\Extras\Lab X.4\Starter ,
  which is essentially the solution to Lab X.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.

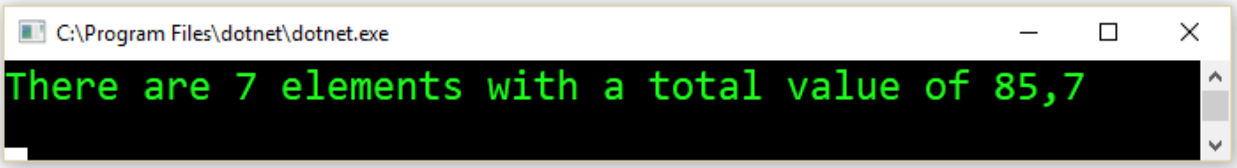
The signature of the `Tally()` method you completed in Lab 01.1 has been generalized accordingly as follows:

```
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 X.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.:

```
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\Extras\Lab X.5\Starter` ,
  which contains a project called `MaximumSubsumProblem`.

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

```
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 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).

Hints:

- Find out what Kadane's Algorithm is
- Use tuples to maintain state (or records, if you want)
- Uncle Google is probably your friend here... 😉

Lab X.6: ”Pattern Matching Recursive Types” (⭐⭐)

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

- Open the starter project in
  PathToCourseFiles\Labs\Extras\Lab X.6\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 X.7: "Url Content Fetching" (★★★)

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

- Open the starter project in
  PathToCourseFiles\Labs\Extras\Lab X.7\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`! 😊


Microsoft Visual Studio Debug Console
=== [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-->

Lab X.8: "Dictionaries and Records" (★)

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

- Open the starter project in
  PathToCourseFiles\Labs\Extras\Lab X.8\Starter ,
  which contains a project called KeyToAwesomeness.

The project defines two enumeration types

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

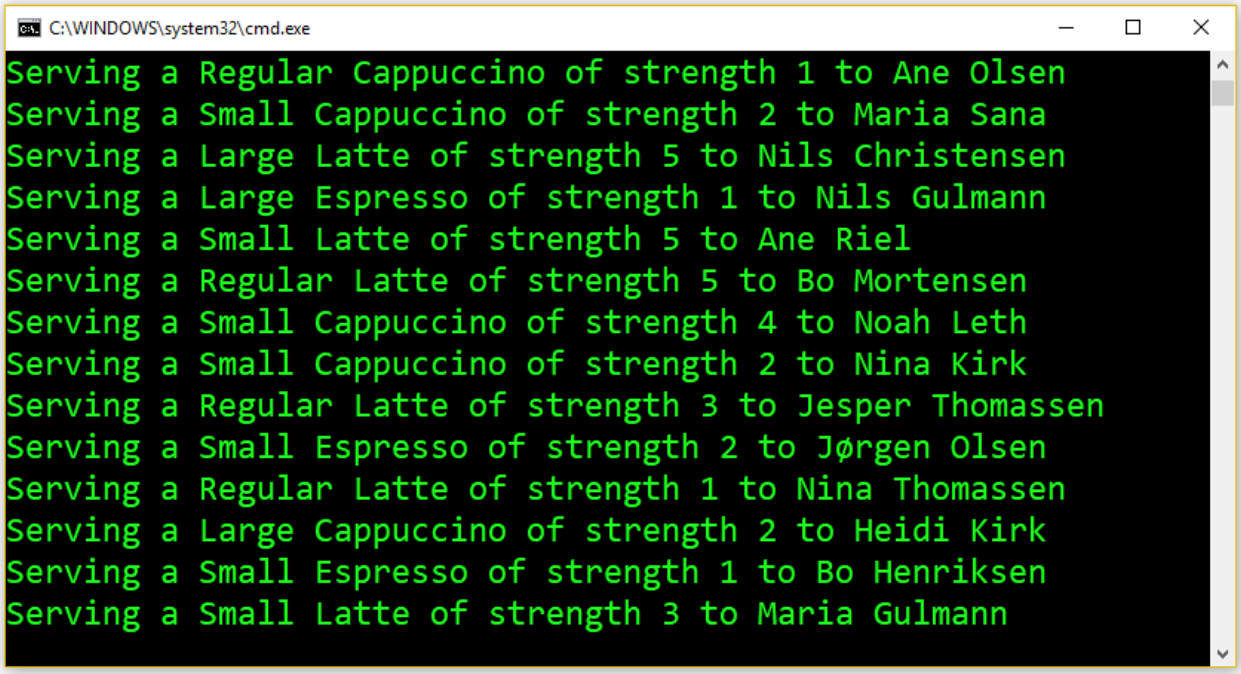
and

```
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:

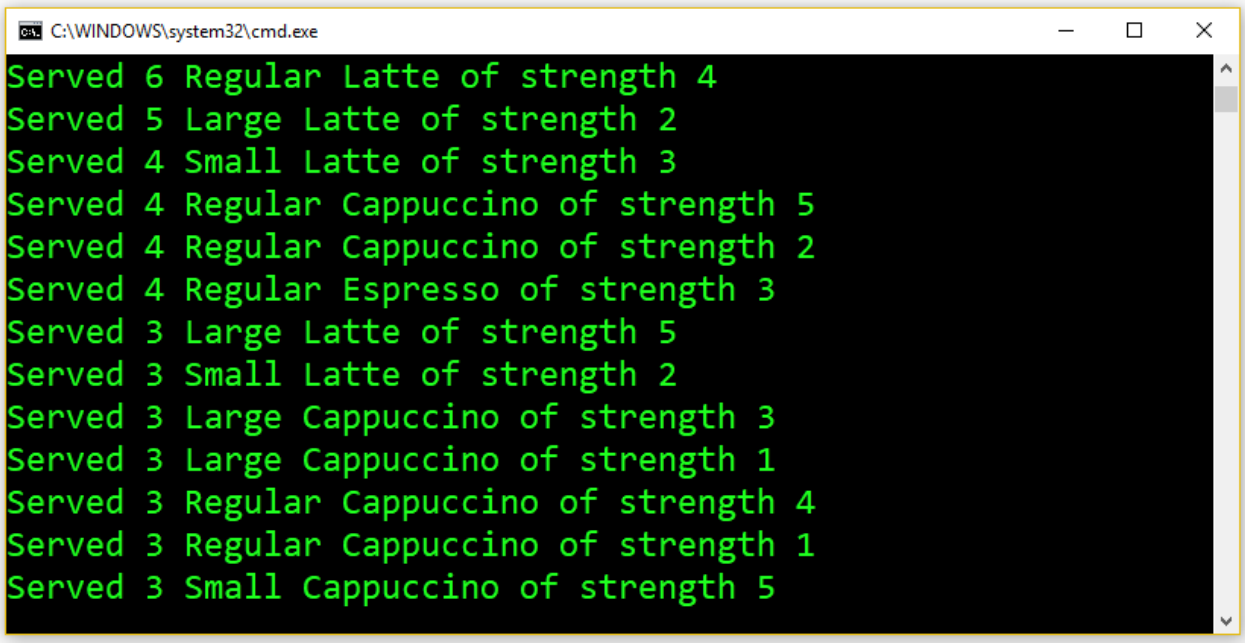
```
void Serve( string customerName, CoffeeKind kind, CoffeeSize size,
            int strength )
{
    Console.WriteLine($"Serving a {size} {kind} of strength {strength}
         to {customerName}");
}

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:



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:



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 X.9: ”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\Extras\Lab X.9\Starter` ,
  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 X.10: "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`\Labs\Extras\Lab X.10\Starter` ,

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 = "VoksneIrereDividererIEnSkov";
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.