



"Cutting-Edge C#"

Lab Manual

Wincubate ApS

13-09-2020



Table of Contents

Exercise types	3
Prerequisites.....	3
Module 1: “An Introduction to C# 7”	4
Lab 01.1: “Basic Tuples”	4
Lab 01.2: “Object Deconstruction” (★)	5
Lab 01.3: “Pattern Matching Shapes” (★)	7
Lab 01.4: “More Tuples, Pattern Matching, and a Local Function” (★)	9
Lab 01.5: “Dictionaries and Tuples” (★)	11
Lab 01.6: “Maximum Subsum Problem” (★★★)	14
Module 2: “What’s New in C# 7.1, 7.2, and 7.3?”	16
Lab 02.1: “Upgrading Tuples”	16
Lab 02.2: “Ref Locals and Ref Returns” (★)	17
Lab 02.3: “Enum Constraints” (★★)	19
Lab 02.4: “Avoiding Copying of Value Types” (★★★)	21
Module 3: “An Introduction to C# 8”	22
Lab 03.1: “Playing with Pattern Matchings” (★)	22
Write the Code Production Index for each employee.....	22
Find all Student Programmers mentored by a Chief Software Engineer	22
Lab 03.2: “Pattern Matching Recursive Types” (★★)	23
Use Pattern Matching to display expressions	23
Use Pattern Matching to evaluate expressions.....	24
Use Positional Pattern Matching to create a better display of expressions	24
Lab 03.3: “Indices and Ranges for Custom Types” (★★★)	25
Implement the new C# 8 indices for SequencePacker<T>	25
Implement the new C# 8 ranges for SequencePacker<T>	25
If you’re bored... ..	26

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 16.3 (or later) with .NET Core 3.1 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# 7”

Lab 01.1: “Basic Tuples”

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

- Open the starter project in
PathToCourseFiles\Labs\Module 01\Lab 01.1\Starter ,
which contains a project called BasicTuples.

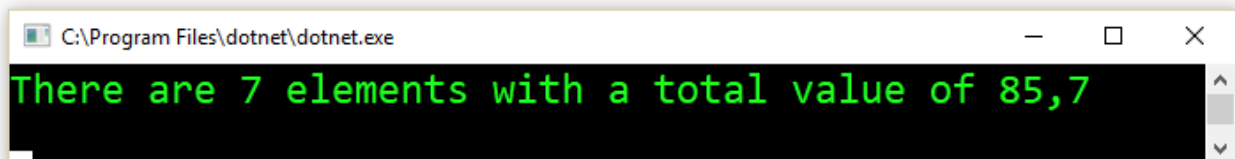
The Main() method contains the following code:

```
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 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\Labs\Module 01\Lab 01.2\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;
    }
}
```

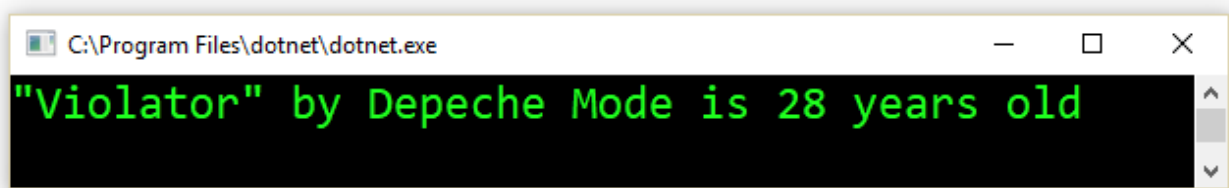
The client project contains a `Main()` method with the following code:

```
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 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\Labs\Module 01\Lab 01.3\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;
    }
}
```

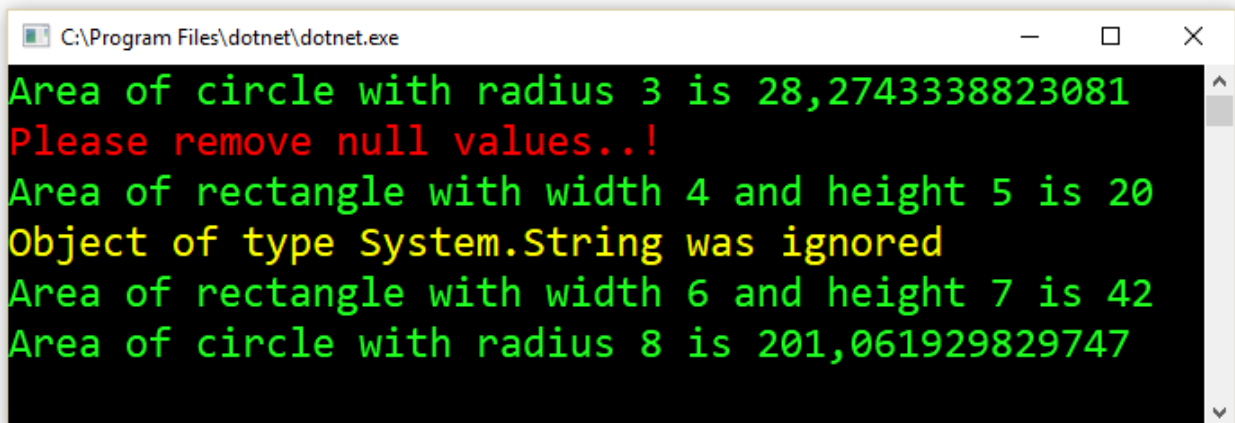
The Main() method contains the following data and method call:

```
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:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Program Files\dotnet\dotnet.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The command prompt area has a black background with text in green and red. The text is as follows:

```
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
```

There is a vertical scrollbar on the right side of the command prompt area.

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\Labs\Module 01\Lab 01.4\Starter ,
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);

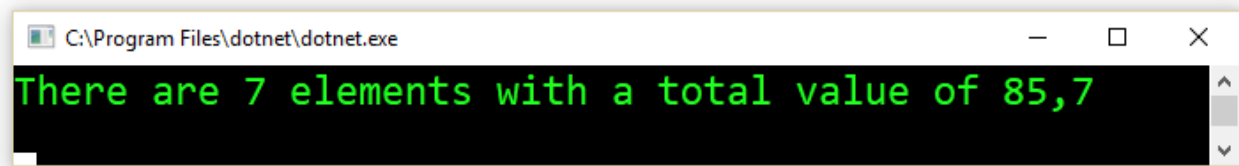
    // TODO

    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:



A screenshot of a Windows command prompt window. The title bar at the top reads "C:\Program Files\dotnet\dotnet.exe". The window has standard Windows window controls (minimize, maximize, close) on the right. The main area of the window is black with green text. The text displayed is "There are 7 elements with a total value of 85,7". A vertical scrollbar is visible on the right side of the text area.

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

Lab 01.5: "Dictionaries and Tuples" (★)

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

- Open the starter project in
PathToCourseFiles\Labs\Module 01\Lab 01.5\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 01.6: "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\Module 01\Lab 01.6\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).

Uncle Google is probably your friend here... 😊

Module 2: “What’s New in C# 7.1, 7.2, and 7.3?”

Lab 02.1: “Upgrading Tuples”

The topic of this exercise is to bring tuples equality to work in an existing C# 7 project.

- Open the starter project in
PathToCourseFiles\Labs\Module 02\Lab 02.1\Starter ,
which contains a project called UpgradingTuples.

The Main() method contains the following code:

```
(int x, int y) tuple = (8, 4);  
var other = (a: 8, b: 4);  
  
WriteLine( tuple == other );
```

Unfortunately, this does not compile in the current project.

- Make this program compile...! 😊

Lab 02.2: "Ref Locals and Ref Returns" (★)

This exercise provides some additional infrastructure to an existing project by using refs.

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

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

```
MeasureUnit unit = new MeasureUnit();  
// TODO 3: initialize currentAverage variable to be read repeatedly  
// in loop  
  
unit.Start();  
  
while (true)  
{  
    _ = Console.ReadLine();  
  
    // TODO 4: Comment in the line below to read currentAverage  
    Console.WriteLine($"Average read: {currentAverage:f2}"  
        + Environment.NewLine);  
}
```

Essentially, this code instantiates a *MeasureUnit* instance which at random intervals produces a reading between 0 and 100 when started. The main loop – from now to eternity – awaits the user pressing ENTER to force a reading of the *MeasureUnit*'s current average.

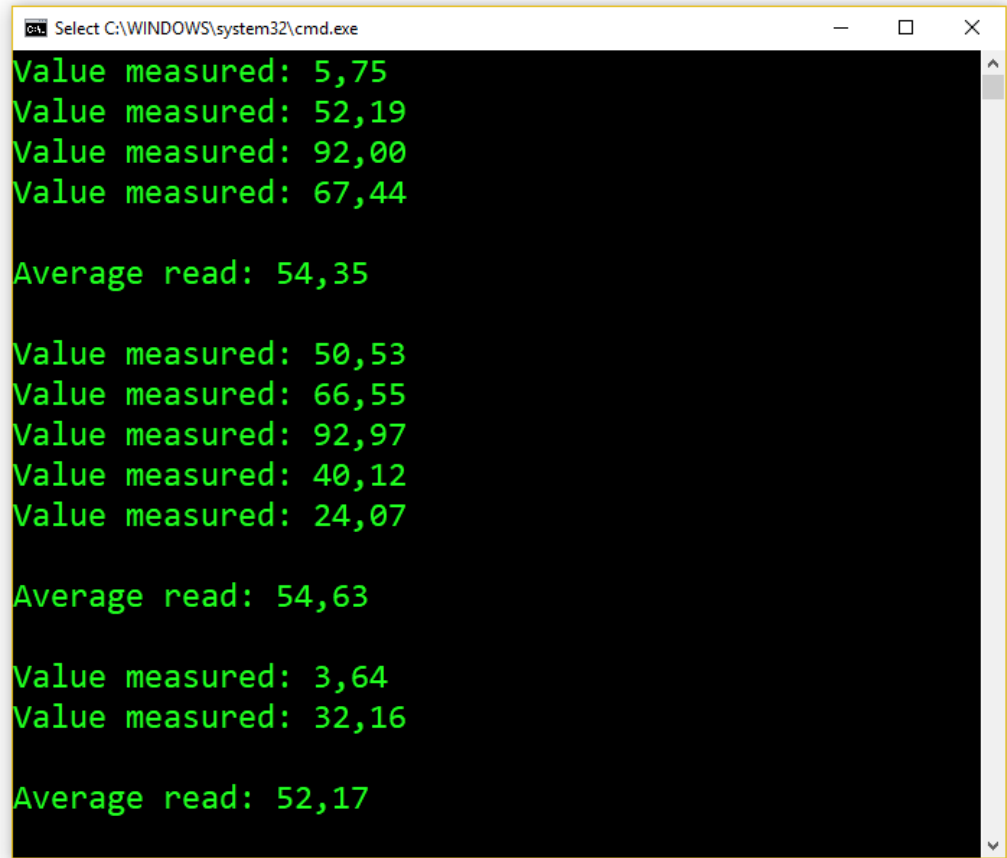
The current average should consist of the latest 10 measured values of the *MeasureUnit*. It should be exposed as an appropriate property of the class to be invoked only once – and then subsequently consulted for the current average whenever the user presses ENTER.

Your task is now to use ref locals and ref returns appropriately to fill in all the pieces missing from the *MeasureUnit* class.

- Inspect the *MeasureUnit* class to familiarize yourself with the existing code in it.
- Locate **TODO 1**:
 - Define an appropriate *CurrentAverage* property to be continuously kept up-to-date
 - Define the data storage needed for the storage of the latest 10 measurement values read.
- Locate **TODO 2**:
 - Implement the *Add()* method appropriately
 - Calculate the average of the last 10 values and keep *CurrentAverage* up-to-date
 - Note: Do not care about thread-safety when updating.
- Locate **TODO 3**:
 - Declare and initialize an appropriate *currentAverage* local variable to be read repeatedly within the loop.
- Locate **TODO 4**:

- Comment in the line reading the currentAverage local variable and printing the result
- Don't change anything else within Main()..!

When your task is completed, a successful execution could result in a screen dump similar to the following:



```
Select C:\WINDOWS\system32\cmd.exe

Value measured: 5,75
Value measured: 52,19
Value measured: 92,00
Value measured: 67,44

Average read: 54,35

Value measured: 50,53
Value measured: 66,55
Value measured: 92,97
Value measured: 40,12
Value measured: 24,07

Average read: 54,63

Value measured: 3,64
Value measured: 32,16

Average read: 52,17
```

Lab 02.3: "Enum Constraints" (☆☆)

This exercise is concerned with the new generic Enum constraint in C# 7.3.

- Open the starter project in
PathToCourseFiles\Labs\Module 02\Lab 02.1\Starter ,
which contains a project called *RelaxingConstraints*.

The Starter project already contains two enum types:

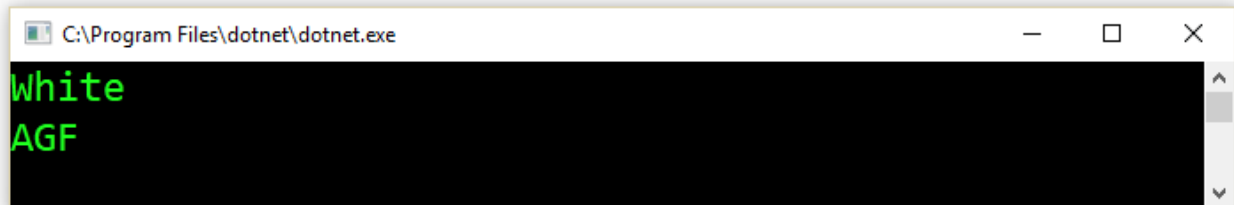
```
enum Color
{
    White,
    Red,
    Green,
    Blue,
    Yellow,
    Gray
}

enum Team
{
    AGF,
    Brøndby,
    FCK,
    OB,
    AaB,
    Horsens,
    Lyngby,
    Randers,
    Helsingør,
    Silkeborg,
    Hobro,
    Sønderjyske,
    FCM,
    FCN
}
```

Your task is to implement a method supplying a random member of a specified concrete enumeration type, such as `Color` or `Team`.

- Write a method `GetRandomMember()` such that
 - the following lines of code (and any other concrete `enum` types) compile:
`Console.WriteLine(GetRandomMember<Color>());`
`Console.WriteLine(GetRandomMember<Team>());`
 - these lines however should not compile:
`Console.WriteLine(GetRandomMember<Enum>());`
`Console.WriteLine(GetRandomMember<string>());`
`Console.WriteLine(GetRandomMember<int>());`
`Console.WriteLine(GetRandomMember<object>());`

A typical output of the program lines within Main() could be:

A screenshot of a Windows command prompt window. The title bar at the top reads 'C:\Program Files\dotnet\dotnet.exe'. The window has standard minimize, maximize, and close buttons. The main area is black with green text. The first line of output is 'White' and the second line is 'AGF'. A vertical scrollbar is visible on the right side of the window.

```
C:\Program Files\dotnet\dotnet.exe  
White  
AGF
```

Lab 02.4: "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\Labs\Module 02\Lab 02.4\Starter ,
which contains a project called *EffectiveValueTypes*.

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 actually 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 in *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 3: “An Introduction to C# 8”

Lab 03.1: “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 03\Lab 03.1\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.



```
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 `StudentProgrammers` who are mentored by a `Chief SoftwareEngineer`.

Lab 03.2: "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 03\Lab 03.2\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:

- SimpleExpression*
- Integer*
- Negative*
- Add*
- 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 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 03.3: "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 03\Lab 03.3\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.