”What’s New in C# 11 and 12 (with Additional Tips ‘n Tricks)”

Lab Manual

Wincubate ApS

08-04-2024

V1.0

Table of Contents

Exercise types

The exercises in the present lab manual differs in type and difficulty. Most exercises can be solved by applying the techniques from the presentations in the slides in a more or less direct manner. Such exercises are not categorized further.

However, the remaining exercises differs slightly in the sense that they are not necessarily easily solvable. These are categorized as follows:

Labs marked with a single star denote that the corresponding exercises are a bit more loosely specified.

Labs marked with two stars denote that the corresponding exercises contain only a few hints (or none at all!) or might be a bit more difficult or nonessential. They might even require additional searches for information elsewhere than in the slide presentations.

Labs marked with three stars denote that the corresponding exercises are not expected in any way to be solved. These are difficult, tricky, or mind-bending exercises for the interested participants – mostly for fun! ☺

Prerequisites

The present labs require the course files accompanying the course to be extracted in some directory path, e.g.

```
C:\Wincubate\CS12
```

with Visual Studio 2022 with .NET 6, 7, and 8 installed on the PC.

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

Module 2: "What's New in C# 11?"

Lab 02.1: "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\02\Labs\Lab 02.1\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 = "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.

Lab 02.2: "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\02\Labs\Lab 02.2\Begin ,

which contains a project called `Required`.

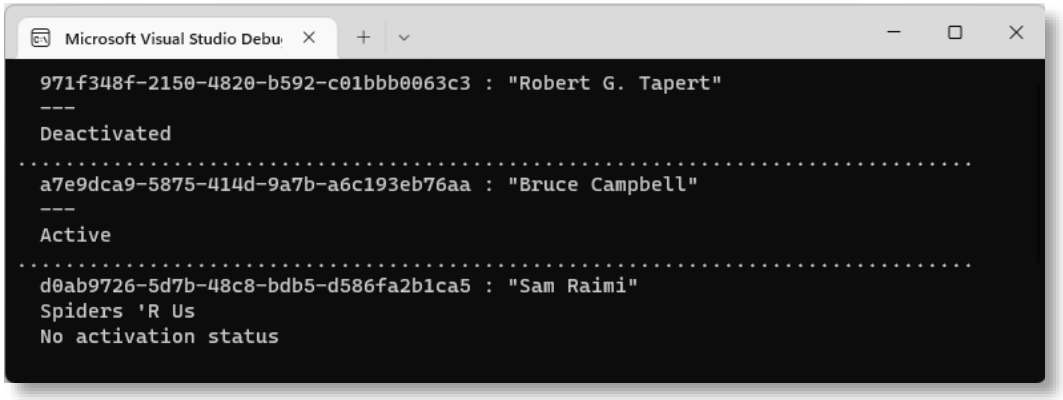
The project contains a class `User` defined as follows:

```
class User
{
    // TODO: a1)
    public Guid UserId { get; init; }
    public string FullName { get; set; }
    public string Company { get; set; }
    public bool IsActive { get; set; }

    public User()
    {
    }

    // TODO: a2)
    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:



- 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.

Module 3: "Newest Additions to C# 12"

Lab 03.1: "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\03\Labs\Lab 03.1\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 03.2: "Investigating Primary Constructors"

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

- Open the starter project in
  PathToCourseFiles\Modules\03\Labs\Lab 03.2\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 4: “Code Better C#”

Lab 04.1: ”Exception Taxonomy”

Inspect your own code bases that you work on regularly.

Find real-world examples of

- Fatal Exceptions,
- Boneheaded Exceptions,
- Vexing Exceptions, and
- Exogenous Exceptions.

Discuss whether you handle these appropriately due to Lippert’s Exception Taxonomy.

Lab 04.2: "A Value Object for Emails" (⭐⭐ / ⭐⭐⭐)

This lab will investigate how to write a value object for email addresses in a manner reminiscent of what we investigated when discussing primitive obsession.

- Open the starter project in
  PathToCourseFiles\Modules\04\Labs\Lab 04.2\Begin ,
  which contains the end result of our primitive obsession example from the presentation.

The code contains the following

```
class Customer
{
    public required CustomerId Id { get; init; }
    public required string FirstName Id { get; init; }
    public required string LastName Id { get; init; }
    public string? Email { get; set; }
    public required PhoneNumber PhoneNumber { get; set; }

    ...
}
```

which you need to improve even further by creating a proper value object for the `Email` property.

When validating the email address string, you are free to use whichever means you find appropriate. For instance, you could either of:

i. The new .NET 7 `[GeneratedRegex]` approach described here: https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-source-generators
   a. Don't worry toooooooo much about finding a completely bullet-proof regular expression! 😊
ii. The **FluentValidation** nuget package
iii. Your own pure code.

Now;

- Create a proper type for `EmailAddress` and apply it to the example.
- Which you would classify the exception type when validating the email address?
  - Make Lippert happy: Also offer a non-vexing alternative…!
  - If you want to do this absolutely perfectly, this requires a small "trick" (⭐⭐⭐)

If time permits:

- Consider whether this could be generalized even further for general string-based value objects…