seavus education development center

C# PROGRAMMING BASICS

LABS

Document: C# Programming Basics-Labs
Edition: 01
Date: 22.05.2012

Created by: Goce Kalamadevski

Number of Pages: 65

LAB 01-GET TO KNOW VISUAL STUDIO 2010

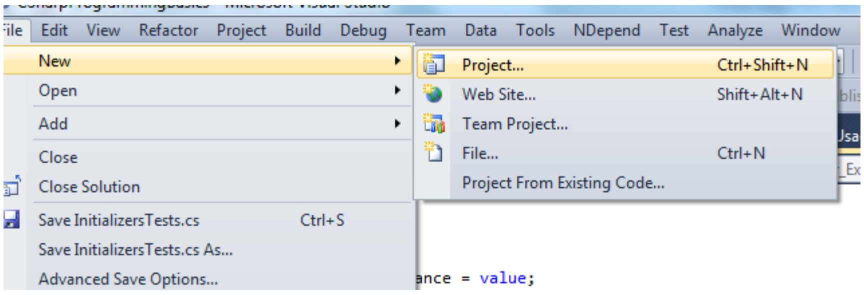
Lab overview:

In this lab you will get to know with the Visual Studio Integrated Development Environment. You will use or navigate through the basic options provided by the VS2010 IDE. For the purpose of this exercise we will use a simple application type (console application) just to illustrate the necessary options.

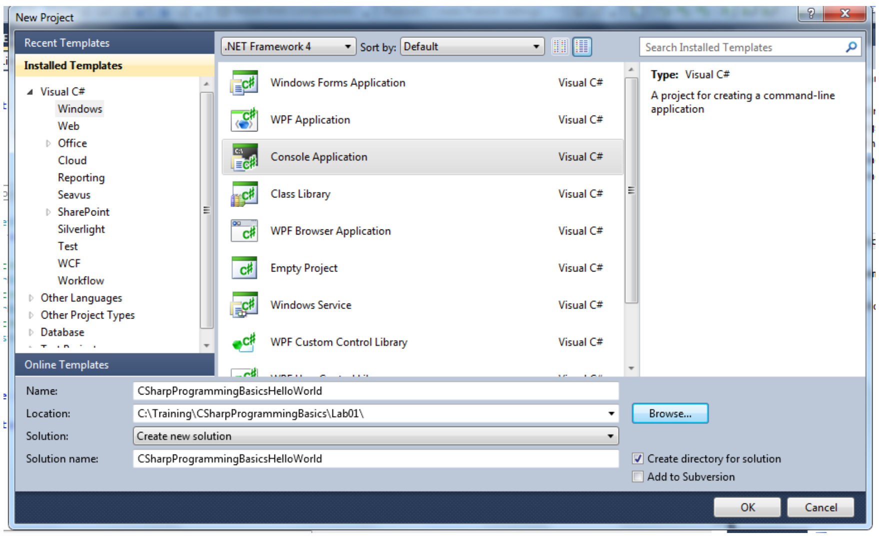
Lab time: 20 minutes

Lab steps:

1. Open ***Visual Studio 2010***
2. Using the file menu create new project (***File->New->Project***)


ile Edit View Refactor Project Build Debug Team Data Tools NDepend Test Analyze Window
New
Open
Add
Close
Close Solution
Save InitializersTests.cs Ctrl+S
Save InitializersTests.cs As...
Advanced Save Options...
Project... Ctrl+Shift+N
Web Site... Shift+Alt+N
Team Project...
File... Ctrl+N
Project From Existing Code...
ance = value;

3. Select the ***Console Application*** project template



Type: Visual C#

A project for creating a command-line application

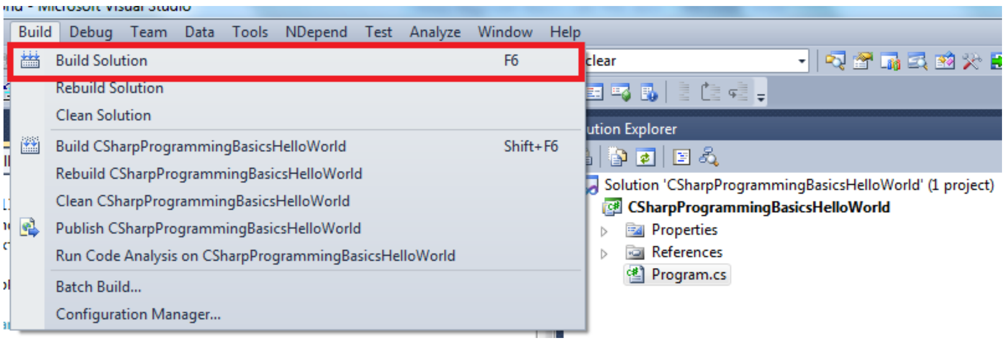
Name:	CSharpProgrammingBasicsHelloWorld
Location:	C:\Training\CSharpProgrammingBasics\Lab01\
Solution:	Create new solution
Solution name:	CSharpProgrammingBasicsHelloWorld

[✓] Create directory for solution

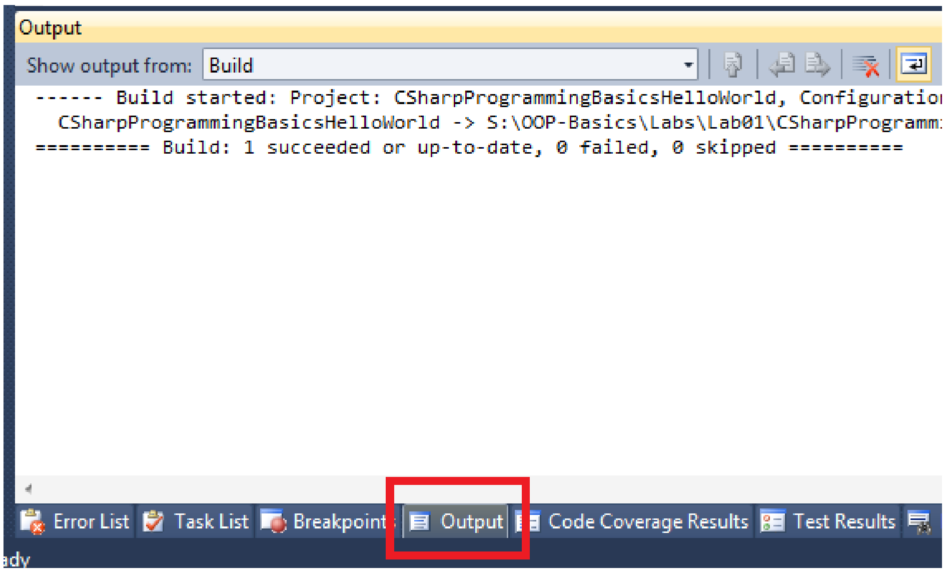
[ ] Add to Subversion

4. Name the project ***CSharpProgrammingBasicsHelloWorld*** and select ***C:\Training\CSharpProgrammingBasics\Labs\Lab01*** folder as the location.
   - Did you notice the ***Solution name*** option?
   - Did you notice the ***Create directory for solution*** option?
   - Did you notice the ***Solution*** option? By default ***Create new solution*** is selected but ***Add to solution*** is also available.
5. A solution should be created with one project in it containing the infrastructure required by the console application.

   Build the solution using the ***Build->Build Solution*** menu option.

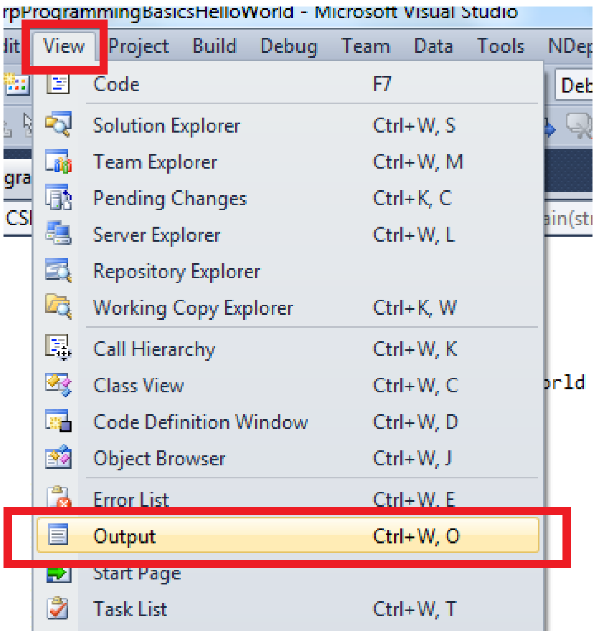

Build
Debug
Team
Data
Tools
NDepend
Test
Analyze
Window
Help
Build Solution F6
Rebuild Solution
Clean Solution
Build CSharpProgrammingBasicsHelloWorld Shift+F6
Rebuild CSharpProgrammingBasicsHelloWorld
Clean CSharpProgrammingBasicsHelloWorld
Publish CSharpProgrammingBasicsHelloWorld
Run Code Analysis on CSharpProgrammingBasicsHelloWorld
Batch Build...
Configuration Manager...
Solution 'CSharpProgrammingBasicsHelloWorld' (1 project)
CSharpProgrammingBasicsHelloWorld
Properties
References
Program.cs

6. Open the ***Output*** results window and rebuild the solution using the ***F6*** key from keyboard.
   1. Click the ***Output tab*** in the bottom of the VS IDE

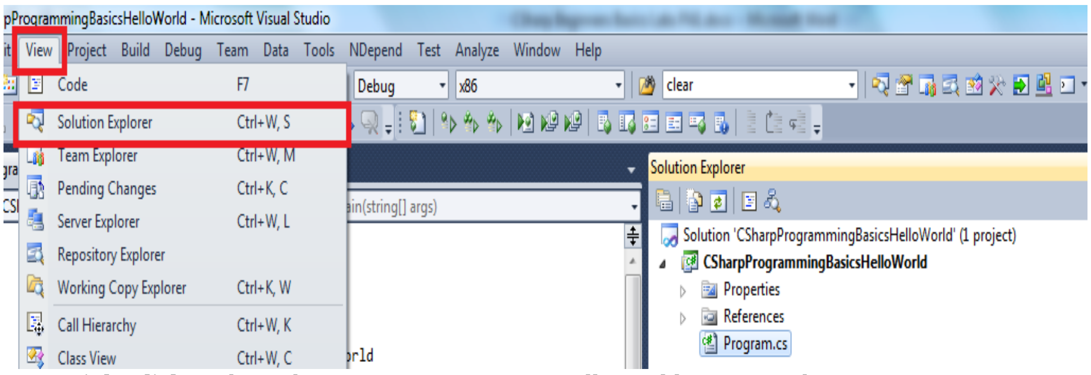


```
------ Build started: Project: CSharpProgrammingBasicsHelloWorld, Configuratio
  CSharpProgrammingBasicsHelloWorld -> S:\OOP-Basics\Labs\Lab01\CSharpProgramm
========== Build: 1 succeeded or up-to-date, 0 failed, 0 skipped ==========
```

2. Use the ***View->Output*** option from the main menu.

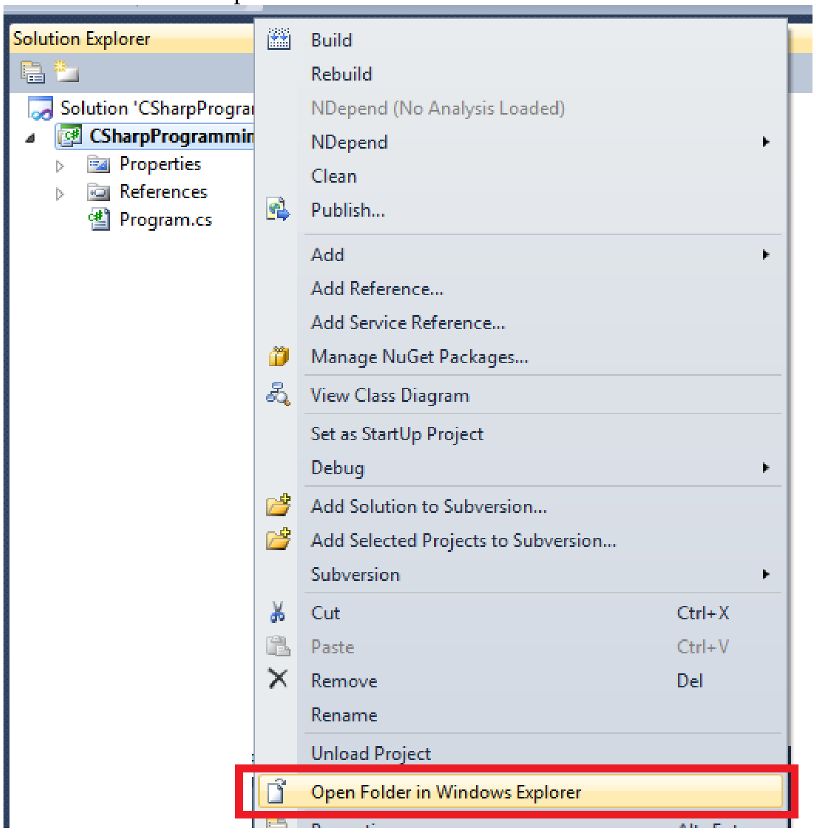


7. Using the ***View->Solution Explorer*** open the solution explorer window.

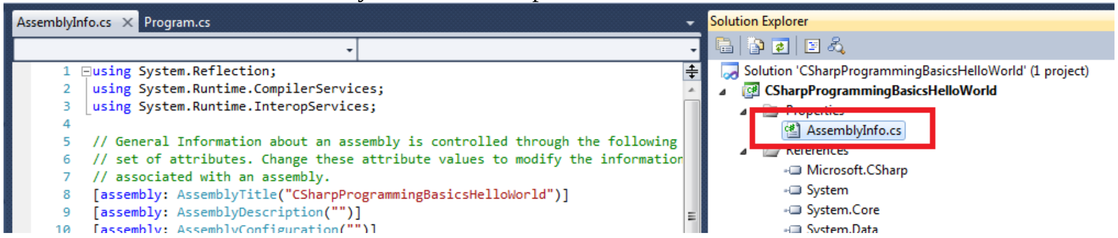

pProgrammingBasicsHelloWorld - Microsoft Visual Studio
View
Project
Build
Debug
Team
Data
Tools
NDepend
Test
Analyze
Window
Help
Code F7
Solution Explorer Ctrl+W, S
Team Explorer Ctrl+W, M
Pending Changes Ctrl+K, C
Server Explorer Ctrl+W, L
Repository Explorer
Working Copy Explorer Ctrl+K, W
Call Hierarchy Ctrl+W, K
Class View Ctrl+W, C
Debug
x86
clear
Solution Explorer
Solution 'CSharpProgrammingBasicsHelloWorld' (1 project)
CSharpProgrammingBasicsHelloWorld
Properties
References
Program.cs

8. ***Right click*** on the CSharpProgrammingBasicsHelloWorld project in the Solution Explorer to open the context menu.

9. Select the ***Open Folder in Windows Explorer*** option to open the project folder in Windows explorer.



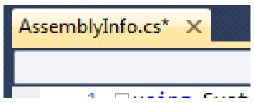
10. Expand the ***Properties*** folder of the CSharpProgrammingBasicsHelloWorld in the solution explorer.
11. Double click the AssemblyInfo.cs file to open the file in the Editor window.



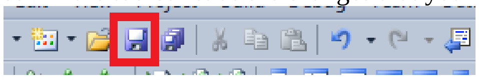
```
using System.Reflection;
using System.Runtime.CompilerServices;
using System.Runtime.InteropServices;

// General Information about an assembly is controlled through the following
// set of attributes. Change these attribute values to modify the informatior
// associated with an assembly.
[assembly: AssemblyTitle("CSharpProgrammingBasicsHelloWorld")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
```

Discuss what is in this file!!!!

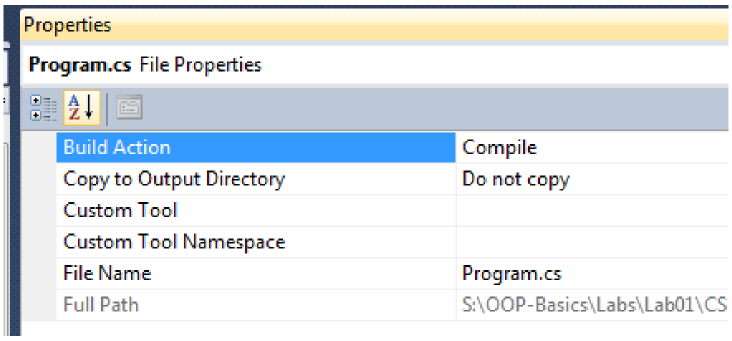
12. Change something in the file.
    Notice the asterisk (*) in the tab after you made the change. This signals that there are unsaved changes in this file.



13. Using the Save icon in the toolbar save the changes that you made.



14. Select the ***Program.cs*** file in the ***Solution explorer*** and using the ***View->Properties window*** option from the ***main menu*** open the Properties window where you can see the file properties.
For displaying the Properties window you can also use the ***F4*** key from the keyboard.



Properties

Program.cs File Properties

Build Action	Compile
Copy to Output Directory	Do not copy
Custom Tool	
Custom Tool Namespace	
File Name	Program.cs
Full Path	S:\OOP-Basics\Labs\Lab01\CS

15. Using the Debug->Start Debugging option from the main menu start the application.

LAB 02-PROJECT REFERENCES

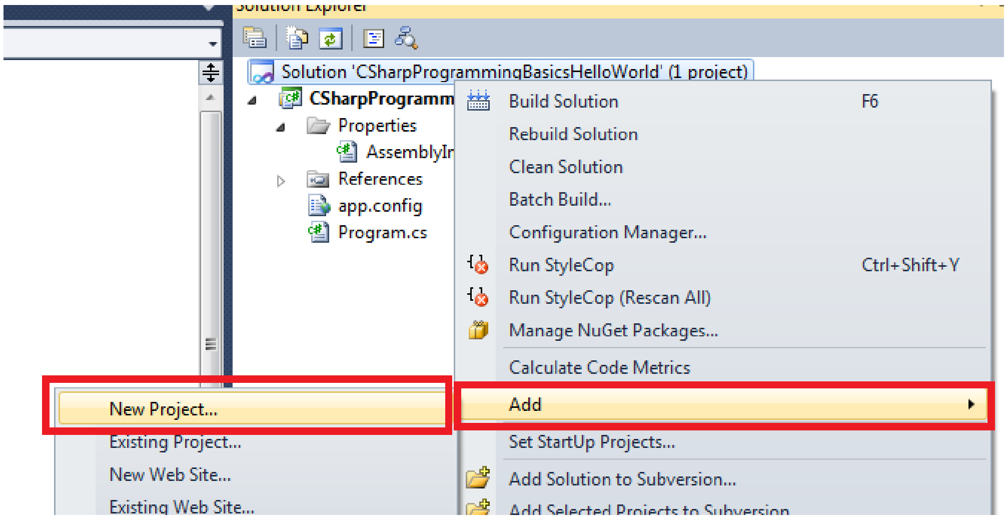
Lab overview:

In this lab you will create will create a new project to an existing solution and reference the new project in another ptoject in the same solution.

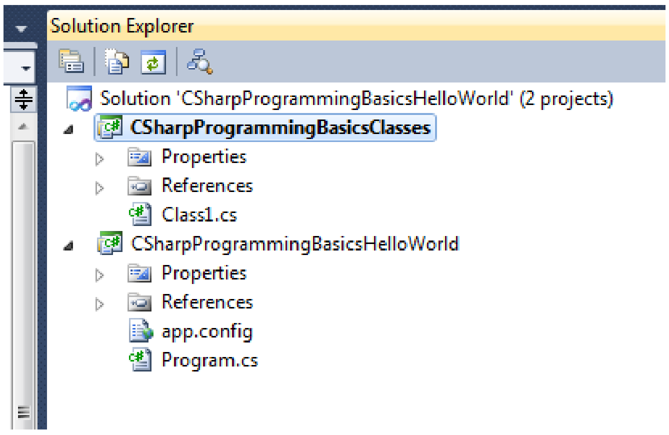
Lab time: 10 minutes

Lab steps:

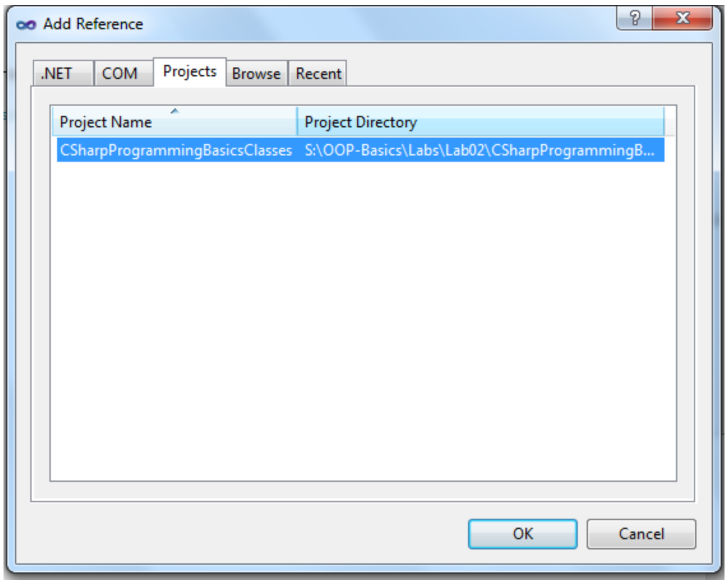
1. ***Right click*** on the solution in the ***Solution Explorer*** and use ***the Add-> New Project*** option to add new project in the solution.



2. Select the ***ClassLibrary*** project template and name it ***CSharpProgrammingBasicsClasses*** and locate it in the ***C:\Training\CSharpProgrammingBasics\Labs\Lab02*** folder. A new project should be added in the solution like in the following picture.


Solution Explorer
Solution 'CSharpProgrammingBasicsHelloWorld' (2 projects)
CSharpProgrammingBasicsClasses
Properties
References
Class1.cs
CSharpProgrammingBasicsHelloWorld
Properties
References
app.config
Program.cs

3. Right click the CSharpProgrammingBasicsHelloWorld project and select the ***Add Reference…*** option from the context menu.This should open a window to select the reference you like to add. The tabs specify different options available for adding references.
   Open the Projects tab and select the CSharpProgrammingBasicsClasses project.



Project Name	Project Directory
CSharpProgrammingBasicsClasses	S:\OOP-Basics\Labs\Lab02\CSharpProgrammingB...

Discuss the other tabs!!!

4. Expand the ***References*** folder of the CSharpProgrammingBasicsHelloWorld in the Solution explorer to verify that the project is added as reference.

LAB 03-PROJECT PROPERTIES

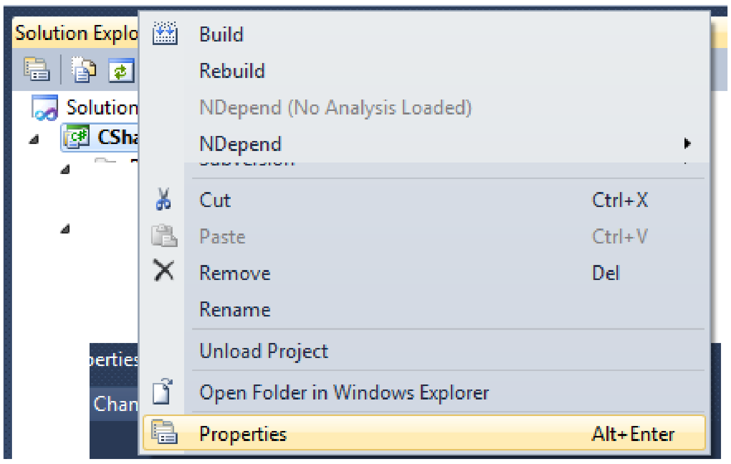
Lab overview:

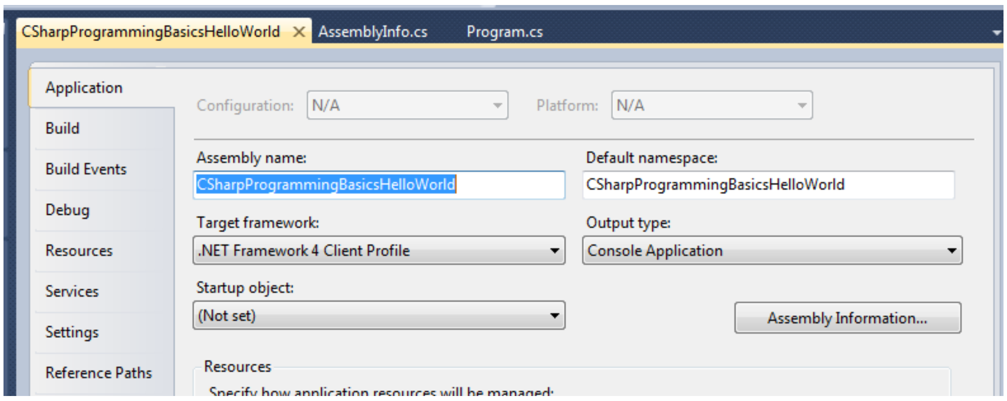
In this lab you will get to know project properties and configurations. Also you will get to know the output folders generated with building the projects.

Lab time: 10 minutes

Lab steps:

1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. Right click the CSharpProgrammingBasicsHelloWorld project and select the ***Properties*** option from the context menu to open the project properties editor window.





Configuration: N/A Platform: N/A

Assembly name: CSharpProgrammingBasicsHelloWorld

Default namespace: CSharpProgrammingBasicsHelloWorld

Target framework: .NET Framework 4 Client Profile

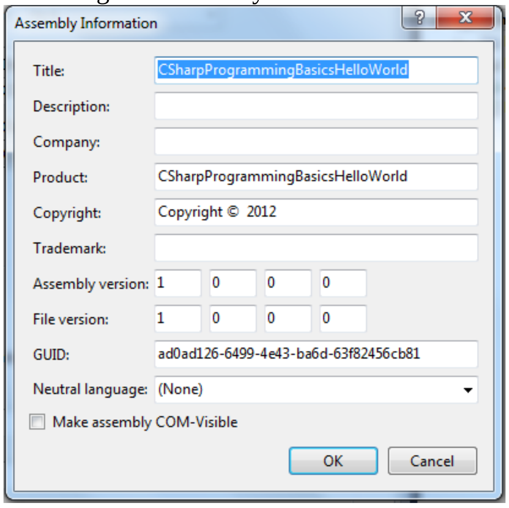
Output type: Console Application

Startup object: (Not set)

Resources

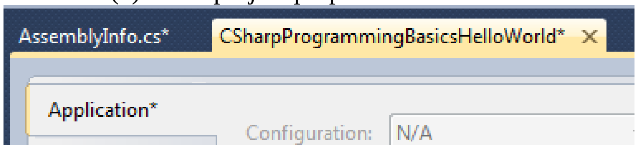
Specify how application resources will be managed:

3. Notice the possible options in the ***Output type*** dropdown.
4. Change the ***Target framework*** to ***.NET Framework 4*** and notice the possible options in the Output type dropdown.
5. Click on the ***Assembly information*** button to open the assembly information editor window and change the assembly version to 1.0.0.1.

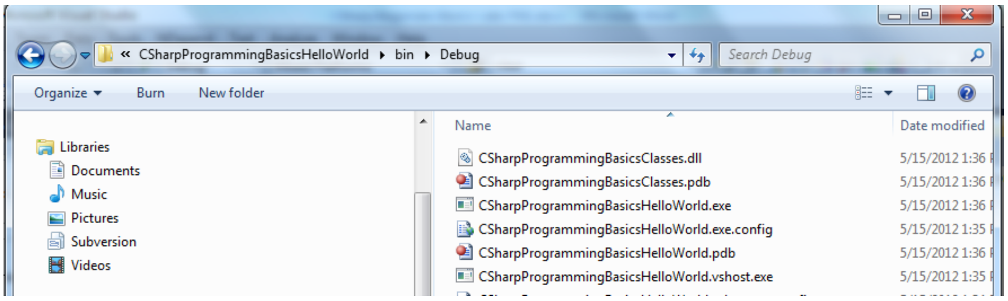


Title: CSharpProgrammingBasicsHelloWorld
Description:
Company:
Product: CSharpProgrammingBasicsHelloWorld
Copyright: Copyright © 2012
Trademark:
Assembly version: 1 0 0 0
File version: 1 0 0 0
GUID: ad0ad126-6499-4e43-ba6d-63f82456cb81
Neutral language: (None)
Make assembly COM-Visible

6. Notice the asterisk (*) in the project properties window tab.



7. Verify that the changes made to the assembly version are saved in the AssemblyInfo.cs file.
8. Find the review the ***Output path*** setting.
9. Rebuild the solution.
10. In the Solution Explorer, right click the CSharpProgrammingBasicsHelloWorld and select the Open folder in Windows Explorer option.
11. Go to ***bin\Debug*** folder and notice the files created.


« CSharpProgrammingBasicsHelloWorld ▸ bin ▸ Debug

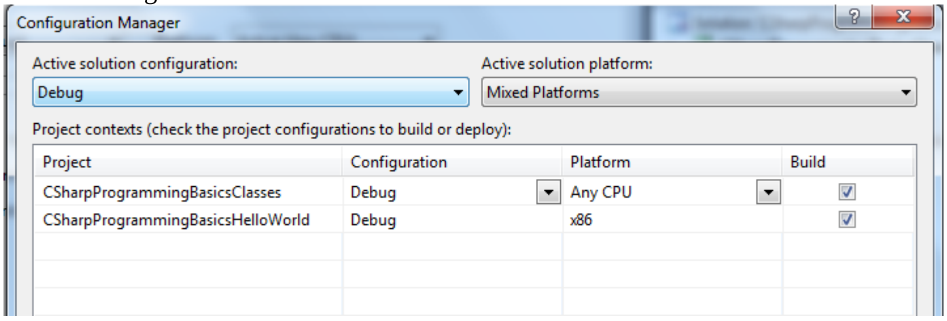
Search Debug

Organize ▾ Burn New folder

Name	Date modified
CSharpProgrammingBasicsClasses.dll	5/15/2012 1:36
CSharpProgrammingBasicsClasses.pdb	5/15/2012 1:36
CSharpProgrammingBasicsHelloWorld.exe	5/15/2012 1:36
CSharpProgrammingBasicsHelloWorld.exe.config	5/15/2012 1:35
CSharpProgrammingBasicsHelloWorld.pdb	5/15/2012 1:36
CSharpProgrammingBasicsHelloWorld.vshost.exe	5/15/2012 1:35

Discuss the folder contents!!!

12. Go back in Visual Studio and in the Solution Explorer right click the solution and select the ***Configuration Manager*** option to open the Configuration Manager window.


Configuration Manager

Active solution configuration: Debug

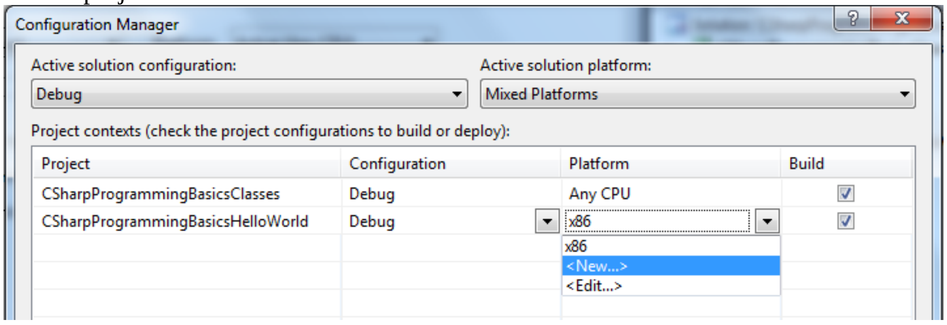
Active solution platform: Mixed Platforms

Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build
CSharpProgrammingBasicsClasses	Debug	Any CPU	✓
CSharpProgrammingBasicsHelloWorld	Debug	x86	✓

Discuss the window!!!

13. Add the Any CPU platform for the CSharpProgrammingBasicsHelloWorld project.


Configuration Manager

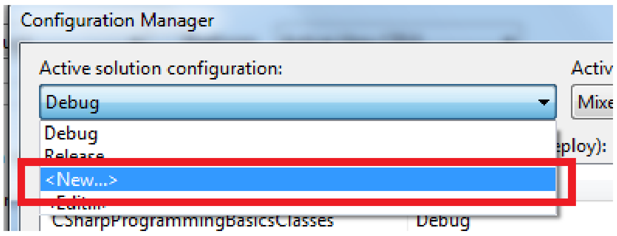
Active solution configuration: Debug

Active solution platform: Mixed Platforms

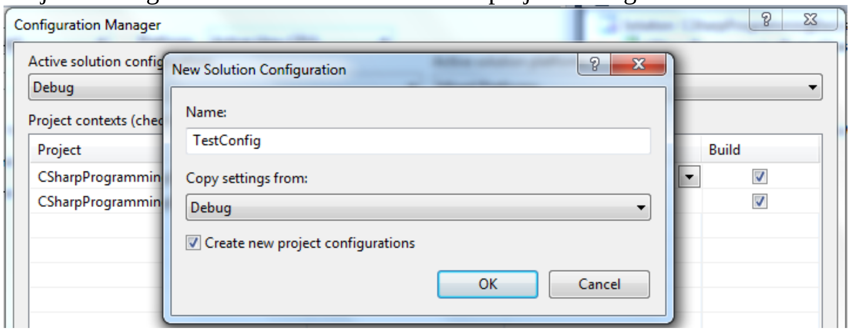
Project contexts (check the project configurations to build or deploy):

Project	Configuration	Platform	Build
CSharpProgrammingBasicsClasses	Debug	Any CPU	✓
CSharpProgrammingBasicsHelloWorld	Debug	x86	✓

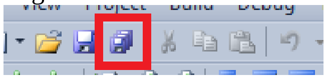
14. Select the ***Any CPU*** platform for the current solution configuration.
15. Add new solution configuration, TestConfig, using the <New...> option in the Active solution configuration dropdown.


Configuration Manager
Active solution configuration:
Debug
Debug
Release
<New...>
<Edit...>
CSharpProgrammingBasicsClasses
Debug

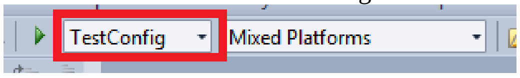
Copy the settings from the Debug configuration and check the Create New Project Configurations check box to add new project configurations as well.



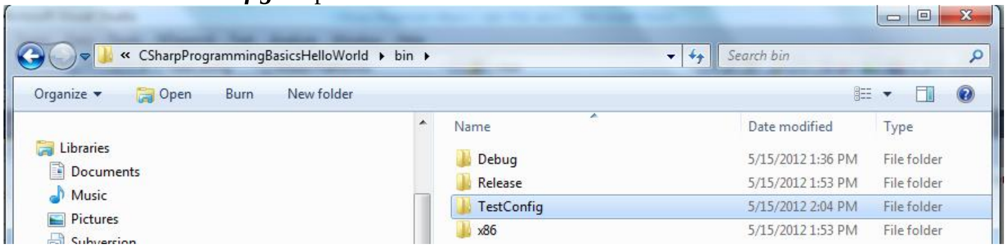
16. After closing the window use the ***Save all*** option in the toolbox to save the changes.



17. Notice the selected solution configuration in the ***Run toolbar***.



18. Rebuild the solution and open the CSharpProgrammingBasicsHelloWorld project folder in the windows explorer. Under the bin folder notice the new folder ***TestConfig*** output folder and in that folder review the created files.



Name	Date modified	Type
Debug	5/15/2012 1:36 PM	File folder
Release	5/15/2012 1:53 PM	File folder
TestConfig	5/15/2012 2:04 PM	File folder
x86	5/15/2012 1:53 PM	File folder

Discuss the solution build configurations!!!

LAB 04-WRITE THE HELLOWORLD

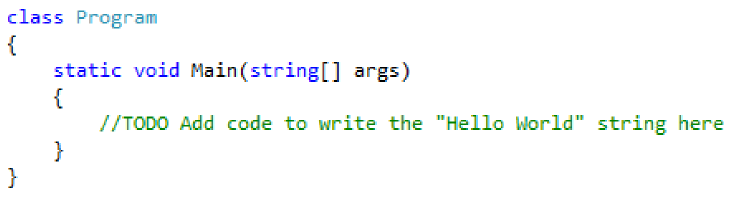
Lab overview:

In this lab you will write a simple C# application that will introduce you to the basic syntax in the C# language in order for you to be able to continue in the more detailed OOP constructs. You will write a simple console application that will take arguments from the command line and make some simple processing of the arguments. According to the provided values the application will output different results.

Lab time: 20 minutes

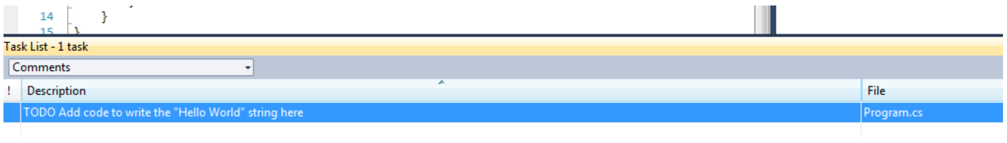
Lab steps:

1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. In the Solution Explorer find the Program.cs file in the CSharpProgrammingBasicsHelloWorlds project and double click it to open the file in the Source Code Editor.
3. In the static void Main method add the line //TODO add code to write Hello World here.



```
class Program
{
    static void Main(string[] args)
    {
        //TODO Add code to write the "Hello World" string here
    }
}
```

4. Open the Task List window to see how this TODO line affects the list. Use the View->Task List option in the main menu or the Task List tab in the buttom of the VS IDE window.

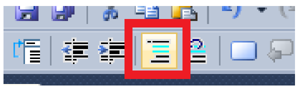


Task List - 1 task

Comments

!	Description	File
	TODO Add code to write the "Hello World" string here	Program.cs

5. Double click the task to take you to the line where the code needs to be added.
6. Add code to write the “Hello World” string in the console output. Use the ***System.Console.WriteLine*** command.
7. In the Solution Explorer right click the CSharpProgrammingBasicsHelloWorld and set the project as startup project using the ***Set as Startup Project*** option in the context menu.
8. Select the ***Debug*** solution configuration in the run toolbar group.

9. Run the project using the ***F5*** key from the keyboard.
10. Add code that will wait for the user to press enter before the output window is closed. Use the ***System.Console.ReadLine*** command.
11. Run the project using the ***F5*** key from the keyboard.
12. Comment the ***System.Console.WriteLine*** command using the single line comments //
13. Write a for loop, before the System.Console.Readline command, that will display the ordinal number before the Hello world message 10 times.
    - Use the for command.
    - Try to generate for loop using the for snippet.
14. Run the application using the ***F5*** key from the keyboard.
15. Comment the for loop using the block comments /**/.
16. Write a while loop that will display the ordinal number before the Hello world message 10 time.
    - Define a variable of type ***int*** and initialize it.
    - Use the while (i<10) { } that will contain the ***System.Console.WriteLine*** command.
    - ***Don't forget to increase/decrease the variable.***
17. Comment the while lopp using the comment option in the toolbar.
    - Select the code you like to comment.
    - Click the Comment code button 
18. Write a foreach loop that will iterate through the arguments provided in the command line and print them in the console output in the format Arg[Index]=Value.
    - Define a variable that will keep the index and initialize it to 0.
    - Use the ***foreach*** command or
    - Use the ***foreach*** snippet.
    - In the foreach body use the System.Console.WriteLine to print the argument values.
19. Run the application using the ***F5*** key. (Doesn't display anything? ☺).
20. Start a command prompt window (Use the search option in the start menu, write cmd and press enter).
21. Change the directory to ***C:\Training\CSharpProgrammingBasics\Labs\Lab01\\CSharpProgrammingBasicsHelloWorld\CSharpProgrammingBasicsHelloWorld\bin\Debug*** (you can use the SolutionExplorer to get the project folder by selecting the project and the press F4 key to display the Properties Window. Copy the path in the

Project Folder property and use it in the command prompt as a base to navigate to bin\Debug folder).

22. In the command prompt call the ***CSharpProgrammingBasicsHelloWorld.exe*** and pass as much as argument as you like separating them with empty space.

    Example: ***CSharpProgrammingBasicsHelloWorld.exe Hello World Called With Arguments***

Note: Don't be alarmed if the previous values are printed out, maybe you forgot to rebuild the project.

Discuss the loop types and command prompt!!!

23. Write a code that will read a configuration setting from the application configuration file and print the value of that setting additional to argument values.
    - In the solution Explorer double click the app.config file to open it in the Editor Window.
    - ***Did you notice how the syntax is highlighted for XML now?***
    - Add the `<appSettings></appSettings>` tag before the `</configuration>` tag. `(Use the intellisense provided by VS).`
    - Between the appSettings tag add the `<add key="MaxNumberOfArguments" value="5"/>` tag that we will use to limit the Maximum Number number of arguments expected in the command line.
    - Add reference to the System.Configuration assembly. This is a .NET assembly and you can find it in the ***.NET*** tab in the ***Add Reference*** window.
    - Declare a variable of type string and name it `_maxNumberOfArguments`
    - Use the `System.Configuration.ConfigurationManager.AppSettings` command to read the setting.
    - Use the System.Console.WriteLine to print the variable in the console output.
    - Run the application using the F5 key.

Discuss the application settings file!!!

24. Write a code that will check if the number of arguments passed in the command line call exceeds the defined maximum and if this is the case throw an ***ApplicationException*** with message saying the the Maximum number of arguments is exceeded.
    - Use the ***Int32.Parse*** command to parse the application setting for the maximum number of arguments and store the value in an int variable called _maxNumberOfArguments.
    - Use the ***if*** command to check if the length of the arguments array is bigger then the value in the _maxNumberOfArguments variable.

- If this is the case use the throw new ApplicationException("Maximum number of arguments exceeded!").

Discuss the code so far!!!

LAB 05-DEBUG THE HELLOWORLD

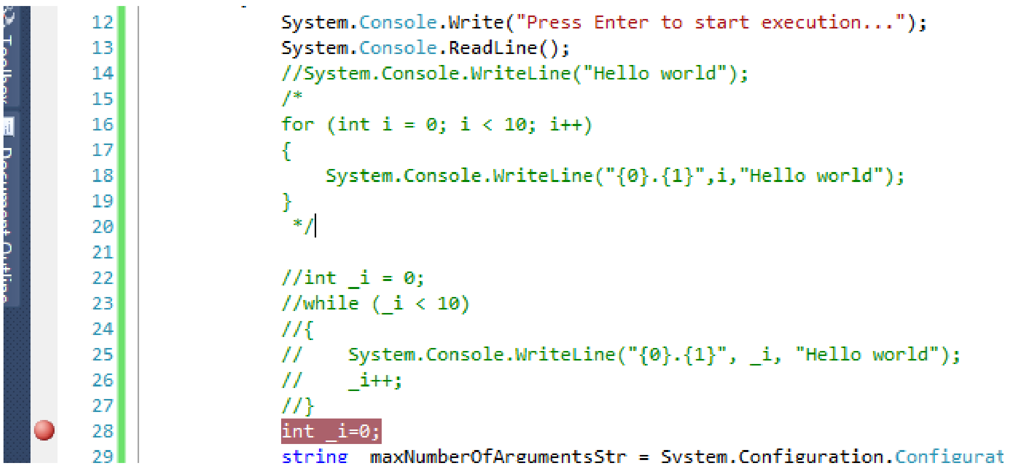
Lab overview:

In this lab you will get to know with tha basic actions when debugging an application. You will put breakpoints, step through the code while it executes, inspect and change values as well as attach the debugger to an already running process.

Lab time: 30 minutes

Lab steps:

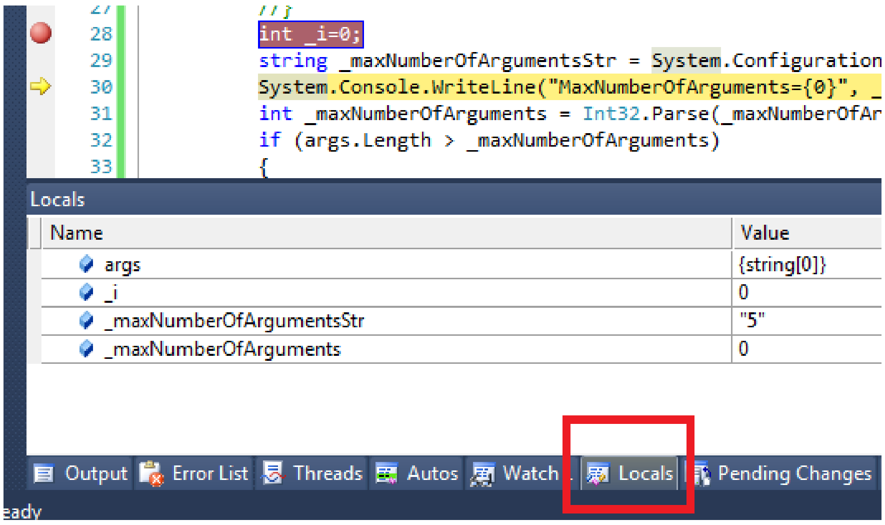
1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. In the Solution Explorer double click the Program.cs file to open it in Code Editor.
3. In the static main(..) method add code that will expect the user to press enter before the other code is executed:
   - Add System.Console.Write("Press Enter to start execution…");
   - Add System.Console.ReadLine();
4. Add a breakpoint on first command after the the System.Console.ReadLine() command:
   - You add a breakpoint by pressing the left button on the mouse over the left strip of the Code Editor window outside the code window, on the line you want to add the breakpoint. This should result in a red dot being displayed on the strip as on the following picture



```
            System.Console.Write("Press Enter to start execution...");
            System.Console.ReadLine();
            //System.Console.WriteLine("Hello world");
            /*
            for (int i = 0; i < 10; i++)
            {
                System.Console.WriteLine("{0}.{1}",i,"Hello world");
            }
             */

            //int _i = 0;
            //while (_i < 10)
            //{
            //    System.Console.WriteLine("{0}.{1}", _i, "Hello world");
            //    _i++;
            //}
            int _i=0;
            string  maxNumberOfArgumentsStr = System.Configuration.Configurat
```

5. Start the program using the F5 key. (Open the ***Debug*** menu and view what is the F5 shortcut).

6. After the program starts, press Enter.
7. The VS will stop the execution at the breakpoint you specified.
8. ***Open the Debug menu and discuss the menu options.***
9. Step through the code using the F10 key.
10. Stop after the line that reads the MaxNumberOfArgument setting and assigns it to a variable.
11. Review the values for the variables in the Locals window:

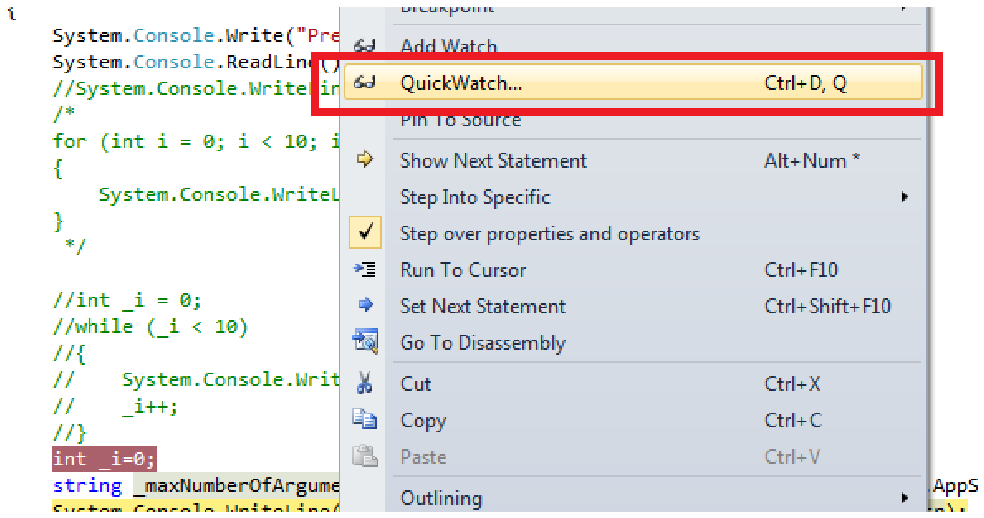


```
            //}
            int _i=0;
            string _maxNumberOfArgumentsStr = System.Configuration
            System.Console.WriteLine("MaxNumberOfArguments={0}", _
            int _maxNumberOfArguments = Int32.Parse(_maxNumberOfAr
            if (args.Length > _maxNumberOfArguments)
            {
```

Locals

Name	Value
args	{string[0]}
_i	0
_maxNumberOfArgumentsStr	"5"
_maxNumberOfArguments	0

12. Double click in the value column at the _maxNumberOfArgumentsStr and change the value for the variable to 3. (Don't delete the quotations since this is a string).
13. In the Code Editor, position the mouse over the _maxNumberOfArgumentsStr. The VS will display a popup with the variable value.
14. In the Code Editor right click on the _maxNumberOfArgumentsStr variable and use the option Quick Watch to open the watch window that is mostly used to inspect the value of a variable. Use the Quick Watch window to change the variable back to 5.



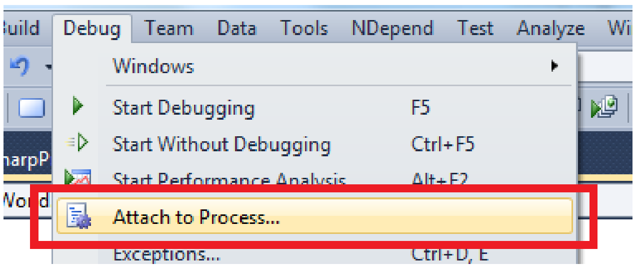
```
    System.Console.Write("Pre
    System.Console.ReadLin
    //System.Console.Write
    /*
    for (int i = 0; i < 10; i
    {
        System.Console.WriteL
    }
     */

    //int _i = 0;
    //while (_i < 10)
    //{
    //    System.Console.Writ
    //    _i++;
    //}
    int _i=0;
    string _maxNumberOfArgume
```

Discuss other debug windows.

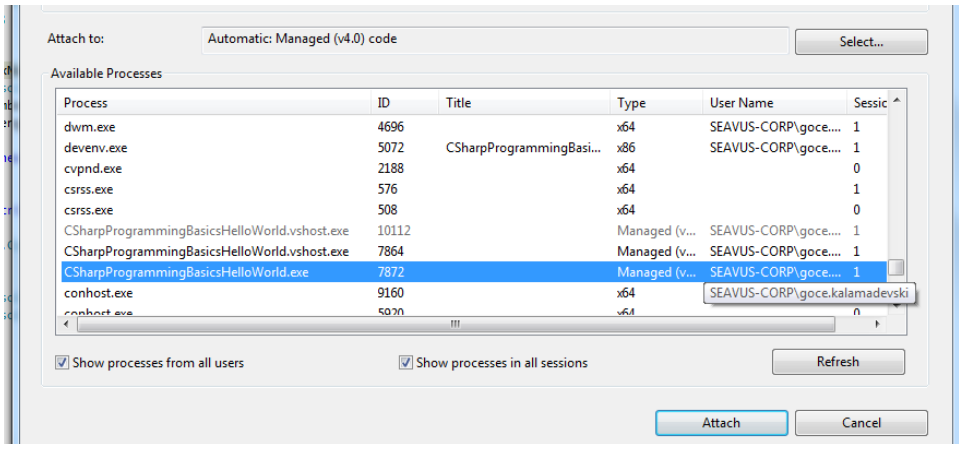
15. Continue the program execution using the F5 key.
16. Close the application.
17. Change the configuration to Release in the run toolbox group.
18. Start the application using the F5 key.
19. Press Enter when prompted to in order to stop the execution at the breakpoint.
    - Did the program stop at the breakpoint?
    - Did you notice that the dot for the breakpoint changed?
    - Did you notice that the breakpoint is not available?

Discuss the situation!!!

20. Change the build configuration back to Debug.
21. Rebuild the solution using the F6 key.
22. Open command prompt window and navigate to the CSharpProgrammingBasicsHelloWorld\bin\Debug folder.
23. Execute the CSharpProgrammingBasicsHelloWorld.exe with some arguments in the command line.
24. While the program waits for you to press enter go back in the VS and in the ***Debug*** menu click on the ***Attach to process…*** option



25. In the next window find the ***CSharpProgrammingBasicsHelloWorld.exe*** process and click ***Attach.***



Attach to: Automatic: Managed (v4.0) code

Available Processes

Process	ID	Title	Type	User Name	Sessic
dwm.exe	4696		x64	SEAVUS-CORP\goce....	1
devenv.exe	5072	CSharpProgrammingBasi...	x86	SEAVUS-CORP\goce....	1
cvpnd.exe	2188		x64		0
csrss.exe	576		x64		1
csrss.exe	508		x64		0
CSharpProgrammingBasicsHelloWorld.vshost.exe	10112		Managed (v...	SEAVUS-CORP\goce....	1
CSharpProgrammingBasicsHelloWorld.vshost.exe	7864		Managed (v...	SEAVUS-CORP\goce....	1
CSharpProgrammingBasicsHelloWorld.exe	7872		Managed (v...	SEAVUS-CORP\goce....	1
conhost.exe	9160		x64	SEAVUS-CORP\goce.kalamadevski	
conhost.exe	5920		x64		0

☑ Show processes from all users ☑ Show processes in all sessions

26. Go back in the command prompt and press Enter.
27. The program should stop execution at the breakpoint after the ReadLine() like in the previous scenario.
28. Review the variables or step through the code.
29. Press the ***Shift+F5*** keys to stop debugging.
30. Go back to the command prompt and review the results.

Discuss the Attach to Process option!!!

Home work:

Create a windows application that will have one form with a text box, label and a button on it. When the user clicks on the button display a message box containing the text that the user entered in the text box.

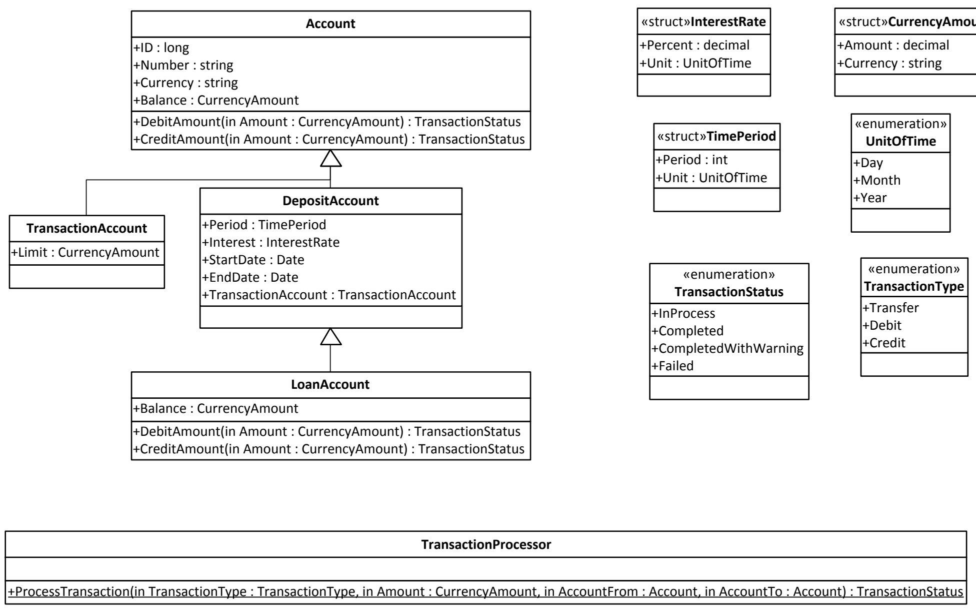
Hints:

- The code that displays the message should be in the implementation of the ***OnClick*** button event.
- To display message use the ***MessageBox.Show*** method of the ***MessageBox*** system class.

DOMAIN MODEL

For the purpose of the following labs you will work on a scenario related to banking and bank transactions. You will create the necessary classes for the bank entities and use them in implementing different scenarios that are common in the everyday banking.

Domain model:


Account
+ID : long
+Number : string
+Currency : string
+Balance : CurrencyAmount
+DebitAmount(in Amount : CurrencyAmount) : TransactionStatus
+CreditAmount(in Amount : CurrencyAmount) : TransactionStatus
TransactionAccount
+Limit : CurrencyAmount
DepositAccount
+Period : TimePeriod
+Interest : InterestRate
+StartDate : Date
+EndDate : Date
+TransactionAccount : TransactionAccount
LoanAccount
+Balance : CurrencyAmount
+DebitAmount(in Amount : CurrencyAmount) : TransactionStatus
+CreditAmount(in Amount : CurrencyAmount) : TransactionStatus
«struct»InterestRate
+Percent : decimal
+Unit : UnitOfTime
«struct»CurrencyAmount
+Amount : decimal
+Currency : string
«struct»TimePeriod
+Period : int
+Unit : UnitOfTime
«enumeration» UnitOfTime
+Day
+Month
+Year
«enumeration» TransactionStatus
+InProcess
+Completed
+CompletedWithWarning
+Failed
«enumeration» TransactionType
+Transfer
+Debit
+Credit
TransactionProcessor
+ProcessTransaction(in TransactionType : TransactionType, in Amount : CurrencyAmount, in AccountFrom : Account, in AccountTo : Account) : TransactionStatus

LAB 06-DEFINING CLASSES

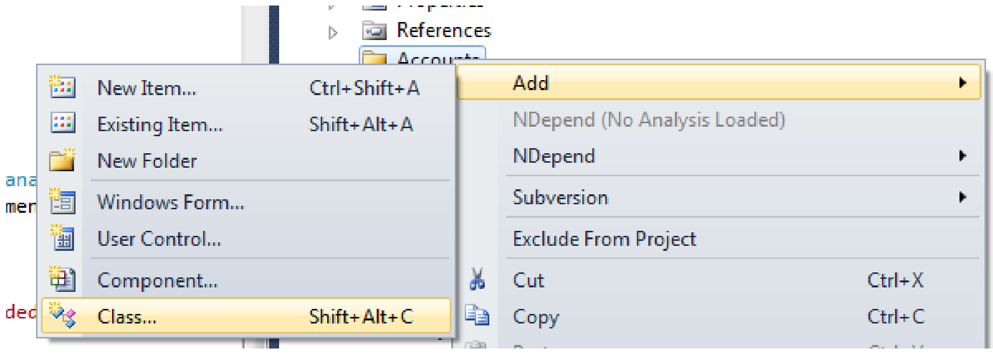
Lab overview:

In this lab you will learn how to create classes and later use these classes in your application. You will add fields and properties to the classes and create constructors. You will also comment the class and class members and use some of the snippets that VS offers to speed up development

Lab time: 40 minutes

Lab steps:

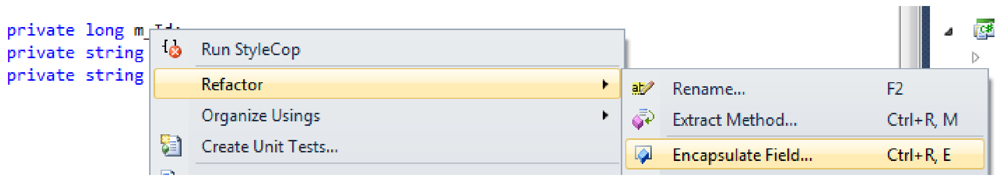
1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. Right click the CSharpProgrammingBasicsClasses project add select the Properties option.
3. In the properties window change the default namespace to CSharpProgrammingBasics.Classes.
4. Create new folder in the CSharpProgrammingBasicsClasses project called Accounts.
5. Create new folder in the CSharpProgrammingBasicsClasses project called Common.
6. In the Accounts folder add new.
   - Right click the Accounts folder
   - Use the Add->Class option from the context menu.



   - Name the class ***Account*** and click ***Add.***
   - ***Notice the namespace!!!***
   - ***Notice the access modifier!!! Is this class public or not?***
7. Change the access modifier of the class to be public.
8. Change the class to be abstract

- Do you remember the difference between regular and abstract class?
- Change the places between public and abstract and try to build the project? Can they be changed?

9. Add fields for Id, Number. (***Before you do, do you remember the difference between field and property?***)
   - The types of the fields are specified in the domain model;
   - Declare the fields like private;
   - The names of the variables should start with m_ . (***Do you know what does the m_ signify?***)
10. Right click on each defined variable and use the Refactor->Encapsulate field option to generate a property.



```
private long m_Id;
private string
private string
```

   - ***Did you notice how VS removes the m_ prefix when creating the property?***
11. Add an auto-implemented public property for Currency. (***Do you remember what an auto-implemented property is?***)
12. Mark the setters as private and the getters as public.
13. Add a constructor for the Account class that will have three parameters, for the Id, Number and Currency.
   - Note: try adding constructor manually or by using the ***ctor*** snippet.
14. Set the values of the properties from the corresponding parameter.
   - ***Did you use the properties to set the values or the fields directly?***
   - ***Did you use the this keyword to access the properties?***
15. Add another constructor for the Account class that will have only one parameter for the Currency. (***Do you remember how is the process of definition multiple methods or constructors with different parameters called?***)
   - Call the constructor with three parameters from this one? Id=-1, Number=”X”
16. Add summary for the class, each property and the constructors.
   - Hint: Use the /// notation and see what VS will generate automatically.
   - ***Did you notice how VS generates the <summary> tag automatically?***
17. Rebuild the solution.

Discuss what you did so far!!!

LAB 07-DEFINING STRUCTURES

Lab overview:

In this lab you will continue with creation of the necessary entities from the above mentioned domain model. More specific the focus will be on creation of the necessary structures and enumerations and also using these structures in the previously created class.

Lab time: 15 minutes

Lab steps:

1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. In the Common folder of the CSharpProgrammingBasicsClasses project add new structure ***CurrencyAmount***
   - Use the Add->Class and after the class is generated change to structure.
3. Add the structure properties.
   - The structure as well as the structure properties should be public.
4. Add comments (summary) for the structure and the structure properties.
5. In the Common folder of the CSharpProgrammingBasicsClasses project add new enumeration ***UnitOfTime***
   - Use the Add->Class and after the class is generated change to enumeration.
6. Add the enumeration elements.
   - Hint: The enumeration can contain only elements and each elements is separated with comma.
7. Add comments (summary) for the enumeration.
8. Add the other structures and enumerations specified in the domain model in the same folder and in the same way as in the previous steps.

Discuss the structures and enumerations code?

- ***Did you notice any difference between defining classes and structures?***
- ***Do you remember of the basic difference between classes and structures?***

LAB 08-ACOUNT DETAILS APPLICATION

Lab overview:

In this lab you will continue with the implementation of the rest of the classes specified in the domain model you will add methods, inherit classes and also implement some functionalities. At the end you will create a Windows Forms Application which will display the details of a created account.

Lab time: 1 Hour

Lab steps:

1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. In the Account class add a field and a property Balance of type CurrencyAmount.

- Note: The CurrencyAmount type will not be highlighted until you add the ***CSharpProgrammingBasics.Classes.Common*** namespace.
- Again define the getter as public and setter as private
- In the constructor taking three parameters initialize the Balance (Amount=0, Currency=CurrencyParameter).
- Since ***CurrencyAmount*** is a structure you can use the new keyword but it is not necessary.

3. Define a region for the public methods called Public methods
4. Add the methods ***DebitAmount*** and ***CreditAmount*** that accept one parameter, the amount, and return a TransactionStatus result.

- Declare the methods as public.
- The Debit operation should decrease the balance of the account.
- The Credit operation should increase the balance of the account.
- Define a region for the private methods called Private Methods
- Add a private method that will check if the amount provided in the DebitAmount and CreditAmount method has the same currency as the account currency. If this is not the case the transaction should not be executed and the Failed TransactionStatus should be returned.
- Don't forget to comment the created methods (public as well as private).

5. In the Accounts folder create the ***TransactionAccount*** and the ***DepositAccount*** classes and inherit them from the Account class.

- Hint: To specify the base class, in the class definition after the class name put ***:*** and after that put the name of the base class.

- These classes should not be abstract and should be public.
- The properties should have public getters and private setters
- For the TransactionAccount class define one constructor that will accept two parameters ***string currency*** and ***decimal:limitAmount.***
- Call the base class constructor with the appropriate parametes and initialize the additional properties in the TransactionAccount class.
- For the DepositAccount class define one constructor that will accept the following parameters ***string currency, TimePeriod depositPeriod, InterestRate interestRate, DateTime startDate, DateTime endDate, TransactionAccount transactionAccount***.
- Call the base class constructor with the appropriate parameters and initialize the additional properties in the DepositAccount class.

Discuss the defined classes so far!!!

- ***What OOP concepts did you use so far?***

6. Add new project called CSharpProgrammingBasicsTransactionApp in the solution and locate it in the ***C:\Training\CSharpProgrammingBasics\Labs\Lab06*** folder.

- In the add new project dialog select the Windows Forms Application template

7. Add reference to the CSharpProgrammingBasicsClasses project.
8. Rename the Form1 form to frmMain.

- Hint: In the designer click on the form and press F4 to display the Properties Window. In this window change the (Name) property.

9. Change the Text property to Account Details Form.
10. From the Toolbox drag and drop two buttons on the form.

- Name one of the buttons btnCreateTransactionAccount and set the Text property to Create Transaction Account (resize the button and the form if necessary)
- Name the second button btnCreateDepositAccount and set the Text property to Create Deposit Account.

11. From the toolbox add two text boxes, one for the account currency and the other for the account limit.

- Add labels for each text box to specify the purpose of each text box and put them near each text box.
- Name one of the text boxes txtCurrency and set the Text property to MKD.
- Name the second text box txtLimit and set the Text property to 10000.
- Set the Text property accordingly.

12. Add text boxes to specify values for the additional properties necessary to create an instance of the DepositAccount class.

- Hint: For the StartDate and EndDate you can use the DateTimePicker control instead the TextBox.
- Hint: For the properties Period and InterestRate use two text boxes one for the amount and one for the unit.
- Follow the ***txt*** prefix to name the text boxes.
- For the DateTimePicker use the ***dtp*** prefix.

13. Add labels to display the common account properties (Id, Number, Currency, Balance).

- From the toolbox drag and drop Label controls on the form.
- Use the ***lbl*** prefix to name the labels and clear the Text property.

14. Add separate labels to display the transaction account specific properties (Limit and Limit currency).
15. Add separate labels to display the deposit specific properties (Period, Interest, StartDate, EndDate and skip the TransactionAccount property for now).
16. Implement a logic that will create a new instance of the TransactionAccount class when the btnCreateTransactionAccount is clicked.

- Use the values from the text boxes to populate the necessary parameters in the TransactionAccount constructor.

17. Define a private method that will receive a parameter of type Account and will populate the Account common labels.
18. Call this method in the OnButtonClick event handler for the btnCreateTransactionAccount to display the details for the created transaction account.
19. Define a private method that will receive a parameter of type Account and will check if this is a TransactionAccount and if this is true will populate the TransactionAccount specific label otherwise it will clear the TransactionAccount specific labels.
20. Implement a logic that will create a new instance of the DepositAccount class when the btnCreateDepositAccount is clicked.

- Use the values from the deposit account specific text boxes to populate the necessary parameters in the DepositAccount constructor.
- Hint: Parse the values from the text boxes to the appropriate types.

21. Define a private method that will receive a parameter of type Account and will check if this is a DepositAccount and if this is true will populate the DepositAccount specific label otherwise it will clear the DepositAccount specific labels.

22. Call this method from the above mentioned method that displayes the Account common details.
23. Set the CSharpProgrammingBasicsTransactionApp as startup project.
24. Start the application using the F5 key.

Did you notice how you can benefit from the abstraction OOP concept (when displaying the common account properties)?

Did you notice how you can benefit from inheritance to specialize the base account class?

LAB 09-INTERFACES AND OVERRIDING

Lab overview:

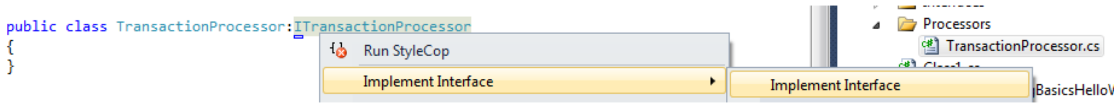
In this lab you will improve the desing of our classes by introducing interfaces. You will implement these interfaces in your classes where necessary and refactor the code in the application to use the interfaces instead of classes. You will also override some of the methods defined in the base classes to implement specific behaviour. You will extend the possibilities of your application to create the LoanAccount entity and also to perform some transactions.

Lab time: 1 hour 30 minutes

Lab steps:

1. Open the CSharpProgrammingBasicsHelloWorld solution.
2. Add folder ***Interfaces*** in the CSharpProgrammingBasicsClasses project.
3. In the ***Interfaces*** folder add interface ***IAccount*** that will contain all the properties from the Account class (only the get) and the DebitAmount and CreditAmount methods.
4. Implement the interface in the Account class.
5. Save and Build the solution.
    - Hint: If the IAccount is not highlighted try adding the namespace (try using the Shift+Alt+F10 key combination while the cursor is at the IAccount).
    - ***Was there a need to change something in the Account class?***
6. In the Interfaces folder add ***ITransactionAccount*** and ***IDepositAccount*** that will contain all the properties and methods from the TransactionAccount and DepositAccount class and implement them in the corresponding class.
    - Hint: Right click on the class name and try using the Refactor->Extract Interface option from the context menu to see how VS can extract interfaces from existing classes.
    - If you used the Extract Interface option then move the file to the Interfaces folder, change the namespace to correspond to the folder structure and apply these changes in the namespace in the class implementing the interface as well.
    - Inherit the ITransactionAccount and IDepositAccount interface from IAccount interface.
    - In the IDepositAccount interface change the type for the TransactionAccount property to ITransactionAccount and change the type in the DepositAccount class as well.

7. Save and Build the solution.
8. In the Interfaces folder add interface ***ITransactionProcessor***.
   - Hint: Use the Right click->Add-> New item.. option.
   - Define one method ***ProcessTransaction*** with the appropriate parameters and return type in this interface. Use the ***IAccount*** interface in the method parameters instead of Account class.
9. Add new folder Processors in the CSharpProgrammingBasicsClasses and add class TransactionProcessor that will implement the ITransactionProcessor interface.



```
public class TransactionProcessor:ITransactionProcessor
{
}
```

10. In the implementation of the ProcessTransaction method implement the transaction types specified in the TransactionType enumeration..
    - Hint: Use the ***switch*** command instead of ***if*** to test for transaction type.
    - For ***Transfer*** type of transaction you should call the ***DebitAmount*** method on the account passed in the ***accountFrom*** parameter and call the ***CreditAmount*** method on the account passed in the ***accountTo*** parameter.
    - For ***Debit*** type of transaction you should call the ***DebitAmount*** method on the account passed in the AccountFrom parameter.
    - For ***Credit*** type of transaction you should call the ***CreditAmount*** method on the account passed in the AccountFrom parameter.
11. Save and Build the solution.
12. In the CSharpProgrammingBasicsTransactionApp copy the already created labels for displaying account details in order to be able to display details for one more account and rename them using the ***lbl__To*** syntax.
    - Refactor the implementation of the private methods that display account details to use the IAccount, ITransactionAccount and IDepositAccount interfaces instead of the classes.
13. In the CSharpProgrammingBasicsTransactionApp add additional button btnMakeTransaction for making a Transfer transaction.
    - In the implementation of the OnClick event, create one instance that implements the ITransactionAccount and one instance that implements the IDepositAccount. Also create an instance that implements the ITransactionProcessor and transfer 20000 MKD from the ITransactionAccount to the IDepositAccount.
    - After the transaction is processed display details for the ITransactionAccount and IDepositAccount instances in the appropriate labels.
14. Run the application using the F5 key.

Discuss what you did so far!!!

- ***Did you notice how interface implementation is specified in the classes?***
- ***Did you notice how inherited classes inherit interface implementation as well?***
- ***Did you notice how you used the Abstraction OOP principle through interfaces?***

15. In the Interfaces folder add interface ***ILoanAccount*** that will inherit from ***IDepositAccount*** and it will have no additional members.

Why are we doing this?

16. In the Account folder add LoanAccount class that will inherit the DepositAccount class, will implement the ILoanAccount interface and it will be sealed.
17. Rebuild the solution.
    - Hint: An error should appear saying that the DepositAccount does not contain a default constructor. To resolve this error add a constructor for the LoanAccount class that will take the necessary parameters to call the DebitAccount constructor.
18. Go in the Account class and prepare the DebitAmount and CreditAmount method for overriding, by marking them as ***virtual***.
19. Go in the LoanAccount class and override the DebitAmount method.
    - Hint: You override the method by specifying the method again in the inherited class and adding the override keyword.
    - For the LoanAccount the DebitAmount method should increase the balance since more money are taken from this loan account.
20. Go in the LoanAccount class and override the CreditAmount method.
    - For the LoanAccount the CreditAmount method should decrease the balance since money are returned to the laon account.
21. Create a private method ***CreateLoanAccount***, analog to the CreateDebitAccount method that will create a new instance of the LoanAccount class.
22. In the OnClick event handler for the btnMakeTransaction button instead of using IDebitAccount refactor the code in order to make a transfer from ITransactionAccount to ILoanAccount.
    - ***Hint: Don't delete the line where the IDepositAccount is created you will need it later.***
23. Run the application and test the behaviour.

24. In the OnClick event handler for the btnMakeTransaction button instead of using the ITransactionAccount refactor the code to make a transfer from ILoanAccount to IDepositAccount.

Do you know which OOP principle did you use to implement this different behaviour? Discuss!!!

LAB 10-STATIC CLASSES AND METHODS

Lab overview:

In this lab you will continue with the enrichment of the functionalities of the domain model classes by including some static members that will be used as helpers in the classes. You will also define some abstract members and see how are they used and implemented and what can we benefit from them.

In the first part of the lab you will use static classes to generate sequential Id’s for the accounts .

In the second part of the lab you will use abstract method to implement the functionality of generation of account numbers and also use static utility classes to generate these numbers.

Lab time: 40 minutes

Lab steps:

1. In the CSharpProgrammingBasicsClasses project add folder ***Helpers***.
2. In this folder add new class called ***AccountHelper***.
3. Mark this class public static.
4. Add a public static method called GenerateAccountId withouth parameters that will return integer values.
5. Add a private static variable of type integer in the AccountHelper class called s_AccountId.
   - ***Do you know what does the s_ stand for?***
6. Add a static constructor for the class AccountHelper and initialize the varianle s_AccountId=0.
7. Implement the GenerateAccountId method by adding code that will increase the s_AccountId variable and return the new value.
8. Go in the Account class and in the constructor taking only currency parameter add line that will call the AccountHelper.GenerateAccountId() method and store the returned value in the Id property.
9. Run the application and test the behaviour by creating multiple accounts or making transactions.

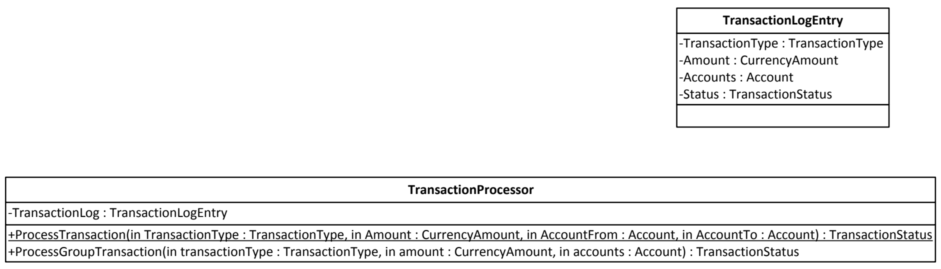
Do you know what OOP principle you used?

- ***Do you think that with this implementation it will be relatively easy to change the way account id’s are generated?***

- ***How many classes will be affected if you decide to change the ID generation?***

10. Stop debugging and go back to the Account class.
11. Add an abstract method called GenerateAccountNumber() that will have no parameters and return string. The method should be protected.
12. In the constructor taking only the currency as parameter call this method and store the returned value in the Number property.
13. Rebuild the solution.
    - It should fail because none of the inherited classes don't implement the abstract method.
14. In the AccountHelper class add a public static method called GenerateAccountNumber that has a parameter called ***accountType*** of type ***Type*** and a parameter called ***accountId*** of type ***long***.
15. Implement the method by adding code that will check what is the type provided in the parameter and according to the type generate an account number in the following format:
    - When type = TransactionAccount, TR0000AccountId
    - When type = DepositAccount, DP0000AccountId
    - When type = LoanAccount, LN0000AccountId.
16. Implement the GenerateAccountNumber abstract method in TransactionAccount, DepositAccount and LoanAccount class to call the AccountHelper.GenerateAccountNumber method and pass the type and Id as parameter.
17. Run the application and test the behaviour.
    - ***Did you notice how polymorphism is achieved using abstract methods?***
    - ***Did you notice how the account number assignement is encapsulated in the Account class and how it helped in delegation of responsibilities through the abstract method?***
    - ***How many classes will need to be changed if the format for the TransactionAccount should be changed? This is the benefit from encapsulation principle.***

DOMAIN MODEL EXTENSION 01


TransactionLogEntry
-TransactionType : TransactionType
-Amount : CurrencyAmount
-Accounts : Account
-Status : TransactionStatus
TransactionProcessor
-TransactionLog : TransactionLogEntry
+ProcessTransaction(in TransactionType : TransactionType, in Amount : CurrencyAmount, in AccountFrom : Account, in AccountTo : Account) : TransactionStatus
+ProcessGroupTransaction(in transactionType : TransactionType, in amount : CurrencyAmount, in accounts : Account) : TransactionStatus

LAB 11-PRACTISE COLLECTIONS AND INDEXERS

Lab overview:

In this lab you will work with arrays and collections. You will extend the capabilities of the TransactionProcessor to enable us to make group transactions and also to keep a log of all executed transactions.

In the first part of the lab you will add additional method to the TransactionProcessor in order for us to be able to make group debits and credits.

In the second part of this lab you will add the functionality for keeping a transaction log for all executed transactions as well as the ability to retrieve these transaction later on using indexers.

Lab time: 45 minutes

Lab steps:

1. In the CSharpProgrammingBasicsClasses project open the ***ITransactionProcessor*** interface.
2. Add new method ***ProcessGroupTransaction*** to the ITransactionProcessor interface accepting the following parameters:
   - transactionType of type TransactionType
   - amount of type CurrencyAmount
   - array of IAccount instances (***IAccount[]***).
3. Implement the ProcessGroupTransaction method in the following way:
   - First check if the provided transaction type is Credit or Debit and if it is not return TransactionFailed.
   - For the Credit/Debit transaction type itterate through the accounts in the array and for each account call the CreditAmount/DebitAmount appropriately.
4. Save and Build the solution.
5. In the CSharpProgrammingBasicsTransacionApp, in the main form add additional button called ***btnMakeGroupTransaction*** for initiating group transactions.
6. In the button’s OnClick event add code that will:
   - Create an array of IAccount and initialize it to have 2 elements
   - Create one IDepositAccount and store it in the first position of the array.
   - Create one ILoanAccount and store it in the second position of the array.
   - Create an instance of the class implementing the ITransactionProcessor interface.

- Call the ProcessGroupTransaction to create a Debit transaction on the accounts.
- Display the details using the DisplayAccountDetails/To method.

7. Run the application.
8. Refactor the code to call the ProcessGroupTransaction method
   - using null array;
   - using initialized array but with null values on each position.
9. Run the application and test the behaviour.

Discuss the last two scenarios!!!

10. In the Processors folder add new class called TransactionLogEntry.
11. Add the properties defined in the domain model for this class.
12. In the TransactionProcessor add a field of type IList called _transactionLog.
13. Add a parametarless constructor for the TransactionProcessor class and initialize this field to a new ArrayList.
14. Create a private method called LogTransaction that will create an instance of the TransactionLogEntry class and add this instance in the _transactionLog collection.
    - This method will have the same parameters as the ProcessGroupTraansaction method and additionally it will have the transactionStatus parameter of type TransactionStatus.
15. Add logic in the ProcessGroupTransaction and ProcessTransaction methods that will call the LogTransaction private method to add new entry in the transaction log.
16. In the ITransactionProcessor add a property called LastTransaction that will have only getter and return TransactionLogEntry.
17. In the ITransactionProcessor add a property called TransactionCount that will have only getter and return int.
18. In the ITransactionProcessor add an indexer property that will have only getter and return ***TransactionLogEntry*** and have ***int*** indexer parameter.
19. Implement the LastTransaction property in the TransactionProcessor class
    - If the transactionlog does not contain entries then return null
20. Implement the TransactionCount property that will count how many transactions were executed so far.
21. Implement the indexer property that will return the TransactionLogEntry at the specified index by the indexer parameter. Check if the index is out of bounds of the collection and if this is the case return null.

22. In the TransactionProcessor add public static method ***GetTransactionProcessor*** that will create an instance that implements the ITransactionProcesor interface using the Singleton pattern.
23. Mark the default constructor in the TransactionProcessor class as private.
24. Change the creation of ITransactionProcessor instance in both OnClick event handler methods.
25. Add additional labels in the form that will display the TotalTransactionCount.
26. Add labels in the form that will display TransactionLogEntry details
    - For displaying account details use the existing labels and methods.
27. Add private method DisplayLastTransactionDetails that will get the last TransactionLogEntry from the transaction processor using the LastTransaction property and display the details in the appropriate labels as well as the transaction count.
28. Call this method in both of the OnClick handlers instead of the call to DisplayAccountDetails/To methods.
29. Run the application and test the behaviour.
    - Validate that after multiple transactions the transaction count is increased.
30. Refactor the implementation of the DisplayLastTransactionDetails method to use the indexer to get the last transaction instead the LastTransaction property.
31. Run the application and test the behaviour.

Discuss the code!!!

- ***Did you notice how you had to cast the object taken from the ArrayList in the getters?***
- ***Why is this necessary?***

LAB 12-PRACTISE GENERICS

Lab overview:

In this lab you will refactor the code in the transaction processor to use generic collections in order to see the benefits from generics. Also you will extend the AccountHelper with a generic method that will overload the account number generation and you will see how you can use generic methods outside the generic collections classes and how to specify constraint on generic type parameters.

Lab time: 10 minutes

Lab steps:

1. Open the TransactionProcessor class.
2. Change the type of the _transactionLog field from IList to IList<TransactionLogEntry>
3. Remove the casting from the LastTransaction and the indexer.
4. Run the application and test the behaviour.

Discuss the differences!!!

- What are the benefits from using generic IList?
- Did you notice that we benefited from the good encapsulation once again? Were there any changes necessary outside the TransactionProcessor class?

5. Stop debugging the application.
6. Open the AccountHelper class.
7. Add an override to the public static method GenerateAccountNumber that will be generic and will have only the accountId parameter.
8. Add constraint on the generic type to be inherited from IAccount.
9. Implement the logic to generate the account number:
    - Use the generic type provided in the method call to find the type of the account for which the number should be generated.
    - Call the previous GenerateAccountNumber method to generate the actual number.
10. Call the generic GenerateAccountNumber from the TransactionAccount, DepositAccount and LoanAccount class.

11. Run the application and test the behaviour.

Discuss the benefits from using the generic method!!!

- What is the process of specifying multiple methods with different signatures called?

Home work:

Comment [GK1]: Remove the homework fron this document and specify it somewhere else.

Refactor the implementation of the CreateDepositAccount and CreateLoanAccount methods in the CSharpProgrammingBasicsTransactionApp.

1. Implement one generic method that will have one parameter of type TransactionAccount.
2. The generic type specified in the method will define what type of account will be created.
3. Define constraints on the generic type (Only DepositAccount and LoanAccount can be created).
4. If the passed type is not DepositAccount or LoanAccount it should return the default value for that type.
5. Call this method from CreateDepositAccont and CreateLoanAccount and test the CSharpProgrammingBasicsTransactionApp.

LAB 13- PRACTISE DELEGATES AND EVENTS

Lab overview:

In the first part of this lab you will refactor the implementation of the TransactionProcessor to implement a scenario where because of a legal regulation while making a transactions each transaction with amount bigger then 20000 MKD needs to be registerd in a special log. If the amount is above 25000 MKD the national bank needs to be informed. Because these actions are separate from the responsibility of the TransactionProcessor we will delegate them to different classes that will handle them. For this purpose we will use delegates.

In the second part of the lab you will change the implementation of the Account class to implement notifications whenever the AccountBalance is changed. First you will implement these notifications using a simple multicast delegate and later you will refactor the code to use an event and see what are the benefits from defining an event over a delegate. You will also implement an event that will be used for notifications whenever a TransactionAccount's limit is exceeded using the existing EventHandler delegate.

Lab time: 1 hour

Lab steps:

1. In the Common folder create a code file Delegates.
2. Add a delegate called TransactionLogger that has three parameters:
    - Parameter account of type IAccount.
    - Parameter transactionType of type TransactionType
    - Parameter amount of type CurrencyAmount.
    - Hint: Delegates are defined using the delegate keyword.
3. Add a property in the ITransactionProcessor called ExternalLogger of type TransactionLogger with public getter and setter and implement this property in the TransactionProcessor class.
4. In the TransactionProcessor add a private method called CallExternalLogger that will have the necessary parameters in order to call the ExteranlLogger delegate.
5. In the ProcessTransaction and ProcessGroupTransaction method after the calls to DebitAmount or CreditAmount call the CallExternalLogger method.
6. Build the solution.
7. In the AccountHelper add a static method called LogTransaction that has the same parameters as the defined delegate.

8. Implement the method with logic that will check whether the amount exceeds 20000 MKD and if this is the case then write the accountNumber, transactionType, amount and amountCurrency to the console output.
9. In the AccountHelper class add a static method called NotifyNationalBank that has the same parameters as the defined delegate.
10. Implement the method with logic that will check whether the amount exceeds 25000 MKD and if this is the case write a notification message in the console output that should mimic the national bank notification.
11. In the TransactionProcessor class, in the static constructor, attach the two methods from the AccountHelper class to the ExternalLogger property.
12. Run the application and initiate some transactions.
    - Hint: After the transaction is initiated go back to VS and open the Output window to check for messages.
    - Add breakpoints in the AccountHelper methods to check if the methods are called.

Did you notice how the delegate can encapsulate multiple methods hence MulticastDelegate?

- Did both of the methods execute as they should?
- Did you use += to assign methods to the delegate?

13. Go back in the TransactionProcessor class and in the static constructor instead += assigne the methods using the =.
14. Run the application and initiate some transactions. ***Were both methods being called***?
15. Stop the applicaton.
16. Go in the Delegates.cs file and define an event arguments class:.
    - Add a public class called BalanceChangedEventArguments that will inherit System.EventArgs class.
    - Add two properties IAccount Account, CurrencyAmount Change that will have public getter and private setter.
    - Add constructor that will have two parameters that will be used to set the above mentioned properties.
    - ***Discuss why only public getters and not setters for the properties?***
17. Define a delegate BalanceChanged that will accept methods that have void return type and accepts two parameters Object sender, BalanceChangedEventArguments eventArgs.

18. In the IAccount interface add event OnBalanceChanged that will be able to accept references to methods that have the above defined signature and implement the event in the Account class.
19. In the setter for the Balance property add logic that will check if the new value for the balance is different from the old and if this is the case call the event.
20. Go in the CSharpProgrammingBasicsTransactionApp and add a handler for the OnBalanceChanged that will print a message with the event arguments details to the console output.
21. Attach this handler to the OnBalanceChanged event that you just added in one of the account factory methods (example:for the DepositAccount).
22. Run the application and initiate a transaction. Was the event hander called? So far you have the same effect as using a multicast delegate directly.
23. Stop the application and try to assign a method to the event using the equal (=) operator.
24. Build the application. What happened? ***Discuss the difference!!!***
25. You will now refactor the code that defines the event using the provided system classes. For that purpose go back in the IAccount interface and instead of using your delegate definition (BalanceChanged) refactor the code to use the EventHandler<> generic delegate.
26. Go back in the Account class and apply the changes.
27. Run the application and initiate a transaction. ***Discuss the change and the benefits!!!***

Comment [GK2]: Include flags attribute. Leav
for the end

LAB 14-PRACTISE ATTRIBUTES

Lab overview:

In this lab you will use attributes. You will create an enumeration and use it as a flag enumeration. This is one of the most common attributes used during development.

You will also create a custom attribute and use it later on in the classes.

Lab time: 15 minutes

Lab steps:

1. In the Common folder add new code file called CreateAccountType.
2. Create an enumeration called CreateAccountType that will derive from int, with the following values:
    - None = 0
    - TransactionAccount = 1
    - DepositAccount = 2
    - LoanAccount = 4
3. Add the Flags attribute to the enumeration.
4. Add new private method in the form called CreateAccounts that will return a Dictionary<CreateAccountType, IAccount> and accept two parameters:
    - CreateAccountType accountTypesToCreate
    - ITransactionAccount transactionAccount
5. Implement the method to check which flags are set in the provided parameter and create the appropriate accounts by calling the Create methods.
6. Call the new method in the btnMakeGroupTransaction click handler.
    - First call it to create only the transaction account.
    - And then with one call to create Deposit and Loan account.
7. Run the application and test the group transaction.
8. In the Common folder add new code file called Attributes.
9. Add new class called AccountMetadataAttribute that will inherit from System.Attribute class.
10. Add two properties in this class:
    - AccountDescription
    - AccountLimitations

11. Limit the usage of this attribute to classes only.
    - Hint: use the AttributeUsage attribute.
12. Add this attribute in the TransactionAccount, DepositAccount, LoanAccount class and set the description and limitations.
    - Hint: Try only AccountMetadata and omit the Attribute suffix.
13. Try to add this attribute on some property and test if the code will compile.
14. Define another attribute called FormatRestrictionAttribute and add two properties:
    - string FormatString
    - int MaxLength
15. Limit the usage of this attribute to properties and fields.
16. In the AccountClass add this attribute to the Number field and limit the length to 16 and the format string to the credit card format (XXXX-XXXX-XXXX-XXXX).
17. Run the application. Nothing should change.

Discuss the attribute usage!!!

- Did you notice how you can specify attributes withouth the Attribute suffix?
- Open the AssemblyInfo.cs file and see the assembly attributes?

LAB 15-PRACTISE EXCEPTIONS

Lab overview:

In this lab you will get to know with using exceptions and exception handling. You will refactor the Account class and add checks in the DebitAmount and CreditAmount methods if the amount provided for the transaction has the same currency as the account currency. You will handle these errors in tha windows application.

Lab time: 30 minutes

Lab steps:

1. Go in the Account class and in the DebitAmount and CreditAmount method add code that will check if the amount provided for the transaction is with the same currency as the account amount. If this is not the case then throw an ApplicationException with message that shows the difference.
   - ***Will the check apply for the LoanAccount class? Discuss!!!***
2. Go in the CSharpProgrammingBasicsTransactionApp and add a text box for the transaction amount called txtTransactionAmount and a text box for the transaction currency called txtTransactionCurrency.
3. Refactor the code to use these text boxes when initiating transactions.
4. Run the application and initiate a transaction with EUR currency.
   - ***Did the application throw the exception?***
   - ***What happened with the application?***
5. Go back in the button click handler and wrap the call to ProcessTransaction within a try/catch/finally block.
   - In the catch block catch an ApplicationException.
6. Add a variable bool _errorOccurred and in the catch block set it to true. Also add a string _errorMsg variable and in the catch block set it to the exception message.
7. Add code in the finally block that will show a message box if an error occurred with the exception message.
8. Run the application and initate a transaction with EUR currency again.
   - ***Did the application crash again?***
   - ***Was the code in the finally executed?***
9. In the following few steps you will create a custom exception. For that purpose in the CSharpProgrammingBasicsClasses project, in the Common folder add a class CurrencyMismatchException that will inherit from ApplicationException.

10. In the CreditAmount and DebitAmount throw the new exception.
11. Run the application and test with EUR transaction.

    - ***Did everything work as before? Discuss!!!***

12. In the button click handler, add another catch (before the catch for ApplicationError) that will catch the new CurrencyMismatchException exception and in the catch cut the code from the ApplicationError catch.
13. In the catch for ApplicationError rethrow the exception.
14. Run the application and initiate an EUR transaction.

Discuss the code!!!

Did you code in the catch (ApplicationError) execute?

Did the finally statement execute?

Did you notice how you can structure you exception handling by using inheritance?

Home work:

Add logic in the TransactionAccount class that will check if the limit is reached. When a transaction is initiated that will overflow the limit throw an exception.

For the purpose of this check define a new exception that will inherit from ApplicationExcpetion and will have and ErrorCode property that will identify the exception.

In the Common folder add an ExceptionLogger class that will be used for logging exceptions (use the Console.Output for now).

In the windows application add two exception classes ***UserInterfaceException*** and ***BusinessLogicException*** that you will use to wrap exceptions. Use the UserInterfaceException to wrap exceptions that come from the UI (example invalid decimal format, or invalid enumeration value) and use the BusinessLogicException to wrap the exceptions inititated from the business operations (such as the example in the previous lab). In both of the exceptions populate the Message property with some “user friendly” message and store the original exception in the InnerException property available in the System.Exception class. Use te InnerException property in the ExceptionLogger class to display exception details.

LAB 16-PRACTISE EXTENSION METHODS

Lab overview:

In this lab you will extend the functionality of the TransactionProcessor withouth changing the class itself. You will use extension method. The extension mehods are a common mechanism for such extensions and are extensively used in the LINQ namespace, where almost all of the functionalities are implemented this way.

Lab time: 15 minutes

Lab steps:

1. Add new folder in the CSharpProgrammingBasicsClasses called Extensions.
2. In this folder add a new static class called ***ProcessorExtensions***.
3. Add a public static method ***ChargeProcessingFee*** that will have the following parameters:
   - this ITransactionProcessor processor,
   - CurrencyAmount amount,
   - IEnumerable<IAccount> accounts
   - and will return TransactionStatus.
4. Implement the method to call the ProcessGroupTransaction method available in the ITransactionProcessor with a Debit transaction type.
5. In the CSharpProgrammingBasicsTransactionApp add a button called btnChargeFee.
6. Implement the OnClick handler and acc code that will
   - Call the new method to charge a 15 MKD fee on a Deposit and Loan account.
   - Display the last transaction details.
7. Run the application and test the new button.

Discuss the code!!!

- ***Did you have any problems with include?***
- ***Did you use the ToArray() method? This is also an extension method. ☺***
- ***Did you notice that you can call the new method just like a regulare static method as well as like an instance method on ITransactionProcessor?***

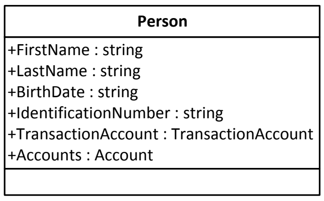
8. Close the application.

9. In the ITransactionProcessor add a method called ***ChargeProcessingFee*** with the same parameters as the previous method, but skip this ITransactionProcessor processor parameter.
10. In the TransactionProcessor class implement this method and add code that will throw NotImplementedException.
11. Run the application and try the button again.

Discuss the results!!!

- ***What method was called, the extension or the instance? (The drawback of using extension methods).***
- ***How can you now call the extension method?***

DOMAIN MODEL EXTENSION 02



LAB 17-PRACTISE INITIALIZERS

Lab overview:

In this lab you will use object and collection initializers. You will first extend the domain model with additional class called Person that will be used to map an individual person that is the banks customer and has one or multiple accounts. You will extend the windows application with additional elements for displaying details abouth the account owner.

In creation of new persons you will use initializers to set values for the Person's properties. You will also use initializers to initialize collections as well as populate them with members.

Lab time: 15 minutes

Lab steps:

1. In the CSharpProgrammingBasicsClasses project add folder Party and in this folder create the Person class.
   - Make the properties in the person class public for both get and set.
2. In the CSharpProgrammingBasicsTransactionApp add group for displaying Person details as well as a button btnCreatePerson to create a new instance of type person.
3. Add a private method CreatePerson that will return a new instance of Person class.
4. In the button Click event handler add code that will call this method to create a Person instance and then display the Person details in the corresponding labels.
5. In the CreatePerson method initialize the properties of the Person class using the initializer syntax.
   - Use your First, Last name as well as birth date.
   - Leave the TransactionAccount and Accounts properties empty
6. Build and run the application.

Did you notice which fields can be accessed using initializers?

7. Extend the CreatePerson metod and create a TransactionAccount as well as a DepositAccount and LoanAccount.
8. Create a new List<IAccount> and use an initializer to populate this list with elements.
9. In the initializer for the Person instance add the initializetion of the TransactionAccount property and Accounts property.

Did you notice how collections can be populated with members withouth using .Add method?

LAB 18-PRACTISE TYPE INVARIATION

Lab overview:

In this lab you will get to know with the invariant typed variables. This is short exercise in order for you to get to know the syntax and use it in the next couple of labs. You will refactor an existing code which will be semantically identical with the previus but with different syntax.

Lab time: 5 minutes

Lab steps:

1. Go to the CreatePerson method that you created in the previous lab.
2. Remove all the type declarations in this method and refactor the code to use the var keyword for all variables.
3. Run and test the application. The application should behave the same.
4. Stop the application.
5. Go to the TransactionProcessor class.
6. In the ProcessGroupTransaction method find the loop that iterates the through the accounts parameter.
7. Remove the type declaration in the loop variable and use the var keyword instead.
8. Run and test the group transaction button. The application should behave the same.

Discuss the ability of the compiler to infere the type of the variable?

- ***What will happen if you try to assign values of different types to a type invariant variable?***

LAB 19-PRACTISE ANONYMOUS TYPES AND METHODS

Lab overview:

In this lab you will use anonymous types and anonymous methods and see how you can benefit from their usage. You will create anonymous types to store some readonly data and use the data later in your code. You will also implement an anonymous method and use it as an event handler for the previous defined events.

You will use the knowledge practised in the previous labs for invariant types as well as for initializers since these combined are used for anonymous types.

Lab time: 15 minutes.

Lab steps:

1. Open the code behind for the Windows application form and locate the handler method for the btnMakeTransaction click event.
2. After the finally block aggregate the variables that you used in the method in a single anonymous typed variable the will have the following properties FromAccount, ToAccount, Amount, HasError and ErrorMessage.
3. Comment the code that shows and error message in the finally and add code after the finally block that will check if the boolean property HasError in the new anonymous variable and if this is the case show the error message else show a success message displaying an information about the FromAccount, ToAccount and the transaction amount for the transfer.
4. Run and test the transfer operation.
5. After the ***if*** statement try to change the value for some of the properties.

   -Were you able to set the properties? Is the application compiling?

   -Try to initialize the variable to null.

   -Discuss the behaviour!!!
6. In the CreateLoanAccount method create an anonymous method that will handle the OnBalanceChanged event to display a message in the console output saying that the balance of the loan account was changed.
   - Hint use the ***delegate (type variable, type variable) {…code};*** syntax.
7. Run and test the transfer operation.
8. Declare a variable in the form that will be incharge to count the number of times the anonymous method is called.
9. Refactor the anonymous method to increase the local variable and display its value in the console as well.

Discuss the benefits of anonymous methods and anonymous types!!!

- ***Did you notice how variables outside the methods scope can be captured? What will happen if the variable is declared in the CreateLoanAccount method?***
- ***Can you say how are the anonymous types and delegates handled in .NET?***
- ***Who is handling them?***

LAB 20-PRACTISE LAMBDA EXPRESSIONS

Lab overview:

In this lab you will use lambda expressions and practise using both types of lambda expresions, the expression lambdas and statement lambdas.

For expression lambdas you will refactor the TransactionProcessor class and add some checks in the ProcessGroupTransaction method as well as extend the TransactionProcessor with an additional method that can be used for searching the transaction log by providing lambda expressions for condition.

For statement lambdas you will refactor the implementation of the CreateLoanAccount method and instead of using the anonymous method for an event handler you will replace this by a statement lambda expression.

Lab time: 30 minutes

Lab steps:

1. In the Common folder in CSharpProgrammingBasicsClasses project add a new enumeration for AccountStatus with the following elements:
    - Inactive,
    - Active
    - Blocked
2. Extend the IAccount interface with additional property ***AccountStatus Status*** that will have a public get and set.
3. Implement this property in the Account class and in the constructor initialize the value to Status=Inactive.
4. In the ProcessGroupTransaction method add code that will check if all of the accounts provided in the transaction are Active. If this is not the case then raise and exception that states that there are accounts that are not active.
    - ***Hint: include the System.Linq namespace and try to use the extension methods provided for the IEnumerable interface (over the IList<IAccount> accounts property).***
    - Try using the Any<> metod.
    - Try other methods as well.
5. Create new exception AccountStatusInvalidException that will be used in the ProcessGroupTransaction.
6. In the windows form handle the exception and display a message box with the error message if an exception occurs.

7. Run the application and initiate a group transaction.
8. Stop the application and refactor the code in the btnMakeGroupTransaction event handler to set the status to Active on the created Loan and Deposit account.
9. Run the application and initiate a group transaction to verify that this solved the problem.

Discuss the code!!!

- ***Did you notice the signature for the extension operations (Any) ?What was the type for the input parameter?***
- ***What kind of a lambda expression did you create?***
- ***Did you specify the input parameter name and type?***
- ***Can you compare the lambda expressions with anonymous methods? Can you specify how and who is handling them?***

10. Open the ITransactionProcessor interface and add a SearchTransactinLog method that will return IEnumerable<TransactionLogEntry> and will accept one parameter of type Func<TransactionLogEntry,bool> as search criteria.
11. Implement the method in the TransactionProcessor class.
    - Hint: use the Where<> extension for the implementation.
12. In the windows form add additional button btnSearch and implement the Click event.
13. Define a private method called HasInactiveAccount that will return bool and accept a parameter of type TransactionLog.
14. Implement the method to return true if there are accounts in the Accounts property of the input parameter with status Inactive.
15. In the btnSearch event handler add code that will get all of the TransactionLogEntry that have inactive accounts using the SearchTransactionLog method from the TransactionProcessor and the HasInactiveAccount method as search criteria.
16. Count the returned entries and display a message box with the count.
17. Run the application, create some transactions, transfer and group transactions, and test the search button.
18. Stop the application and go in the btnSearch handler. Refactor the code that calls the SearchTransactionLog method to use lambda expression instead of the HasInactiveAccount.
19. Run the application and test the search button.

Discuss the code!!!

- ***Did you notice the Func type? Do you know what is this type and how it is defined?***

- ***Dicuss the similarities between the lambda expression and the HasInactiveAccounts method, do you have a picture of what a lambda is?***

20. Next in this lab you will continue with creation of statement lambda expression. For that purpose go in the CreateLoanAccount method and comment the anonymous method assignment to the OnBalanceChanged event.
21. Define new variable of type EventHandler<BalanceChangedEventArgument> called _lambdaHandler.
22. Assign a statement lambda block to the _lambdaHandler variable that will do the same functionality as the anonymous method (increment a variable and write something in the console output) and have two parameters of type object and type BalanceChangedEventArguments.
23. Assign this variable as a handler to the OnBalanceChanged event.
24. Run the application and test some transactions. Verify that your lambda expression is executed.
25. Go back in the handler and remove the type definition from the lambda expression.
26. Run the application and test some transactions. Verify that your lambda expression is executed.
27. Go back in the handler and comment the definition of the variable and assigne the previous lambda expression (withouth type specification) directly to the OnBalanceChanged event.
28. Run the application and test some transactions. Verify that you lambda expression is executed.

Discuss the statement lambdas!!!

- ***Did you notice how the compiler can determine the type based in the context?***
- ***Do you now have a more clear view of what a lambda expression is?***
- ***Do you remember how to define a lambda withouth parameters?***
- ***Do you remember how is the => operator called?***

LAB 21-PRACTISE LINQ TO OBJECTS

Comment [GK3]: Try to define simulation extension methods just like those from LINQ, for where, group etc.

Lab overview:

In this lab you will use LINQ to query data from collections.

You will use the most common LINQ operations that are applicable in LINQ to SQL as well as LINQ to XML. You will query objects, apply filters, group, order and other common operations on object datasources.

After you get to know how to create LINQ queries you will try to figure out how LINQ works under the hood and you will attempt to implement similar logic ot the one already implemented in System.Linq namespace.

Lab time: 1 hour

Lab steps:

1. Before you start with the lab do you remember what does the abbreviation LINQ mean and what are the basic elements of LINQ query. (which keywords are used).
2. Go in the ITransactionProcessor interface and add a method SearchTransactionLog that accepts one parameter of type TransactionStatus and returns an IEnumerable<TransactionLogEntry>.
3. Implement this method in the TransactionProcessor class using LINQ.Query the _transactionLog collection for all TransactionLogEntries that have status the same as the status provided by the parameter.
4. In the windows forms application, in the handler for btnSearch, additional to the count of transactions with inactive accounts, perform a count for transactions that are failed and count for transactions that are succesfull using the new method. Display the numbers in the message box.
5. Run the application, initiate some transactions and test the search button.
6. Change the method SearchTransactions to accept one more parameter CurrencyAmount amount, that can accept null value as well. If this parameter is not null that additional to the filter by status apply the filter for the Amount as well.
   - Hint: LINQ query is not executed until an enumeration operation is performed on the result. You can apply as much as you like additional filters before the result (***IEnumerable<TransactionLog>***) is not enumerated (by a for, foreach, count etc).
7. In the windows forms application, in the handler for the btnSearch, add code that will count all the transactions that are succesfull and have 20000 MKD as amount. Also count succesfull transactions with 150000 MKD. Display this numbers in the message box as well.

8. Run the application, initiate some transactions to 20000MKD and some to 150000MKD and try the search button.

Discuss the LINQ queries that you wrote!!!

- Did you notice how filters are used?
- Did you notice how you can combine filters?

9. Add new class called TransactionHelper in the Helpers folder of the CSharpProgrammingBasicsClasses project.
10. Add a public static method GetMaxMinAmounts that will accept an IEnumerable<TransactionLogEntry> and a string variable for the currency and have two output parametes for the minimum and maximum amount of type CurrencyAmount found in the list of those transactions made with that currency.
    - In the method use a LINQ query to filter the transactions with the specified currency.
    - Use the order by LINQ keyword to order the result by the amount.
    - From the TransactionLogEntry class take only the amount property.
    - From the resulting list get the First element, the minimum, and assign it to the output parameter.
    - Using a LINQ query reorder the elements in the resulting list by amount descending.
    - From the resulting list get the First element, the maximum, and assign it to the output parameter.
11. Call this method in the btnSearch event handler to find the min and max amount from the Completed transactions and display the results in the message box.
12. Run the application, create some transactions and test the search button.

Discuss the code!!!

- Did you notice how ordering is done with LINQ?
- Did you notice how LINQ queries can operate on any IEnumerable collection of objects?
- Did you notice how you can make projections and return different type that the one in the source collection?
- Did you notice how the compiler infers the type of the the result?

13. In the TransactionHelper class add a public static method called GroupByCurrency that will accept IEnumerable<TransactionLogEntry> and return IEnumerable<IGrouping<string,TransactionLogEntry>>.
    - Hint: In the implementation use:

from _t in ….

group _t by …

into _group

select _group

14. In the handler for the btnSearch call the method with the result list for the completed transactions, loop through the grouped results and display the count of transactions for each currency in the message box.
15. Run the application, create some transactions with MKD and some with EUR and test the search button.

Discuss the code!!!

- Did you notice the return type from the grouping operation?

16. In the common folder add new class called CurrencyProjection with the following properties:
    - string Currency
    - decimal Amount,
    - string FromAccount
    - string ToAccount
17. Add another public static method called JoinByCurrency in the TransactionHelper that will have IEnumerable<TransactionLogEntry> as parameter and return IEnumerable<CurrencyProjection>.
18. In the JoinByCurrency method body, declare a variable of type string array called _currencies and add two members to it “MKD” and “EUR”.
    - Hint: use initializers to add the members in the array.
19. Join the _currencies array with the IEnumerable<TransactionLogEntry> parameter and filter only the ***Transfer*** operations. In the select part of the LINQ query “project” the result in a new instances of type CurrencyProjection. Return the result back from the method.
20. Call this method in the btnSearch click handler and loopt through the result. Display the CurrencyProjection properties in the message box.
21. Run the application, initiate some transactions with MKD, EUR and USD and test the search button.

Discuss the join and projection LINQ syntax!!!

- ***Did you try to make composite condition in the join ? Is that possible? How are composite keys handled?***

- ***Can you use anonymous type for the projection instead the CurrencyProjection?***
- ***Did you use initializer syntax to specify the values for the CurrencyProjection properties?***
- ***Did you notice where you lost the USD transactions?***

22. In the following few steps you will try to "reinvent" the LINQ to Objects extensions. You will make your own implementation of the Where and Select LINQ extensions. For that purpose in the ***Extensions*** folder add another class called ***LinqCustomImplementation*** and make it public static.
23. Declare a static method called Select which will be generic with two type parameters TSource and TResult and return IEnumerable<TResult> and have the following input parameters:
    - this IEnumerable<TSource> collection,
    - Func<TSource,TResult> selector
24. Implement the method by calling the System.Linq.Enumerable.Select static method with the appropriate parameters.
25. Declare a static method called Where which will be generic and return IEnumerable<T>, where T is the generic type and have the following input parameters:
    - this IEnumerable<T> collection
    - Func<T,bool> filter
26. Implement the method to call the System.Linq.Enumerable.Where static method and provide the appropriate parameters.
27. In the Helpers folder add a new class called CustomLinqGateway and define a static method FilterByCurrency.
28. Comment the using System.Linq and add using to CSharpProgrammingBasics.Classes.Extensions namespace.
29. Implement the FilterByCurrency method and add a LINQ query that will filter the provided collection for all the TransactionLogEntry's that have the same currency as provided in the parameter.
30. Call this method in the btnSearch click handler to filter the list with transactions for transactions done in USD. Count the items returned by the method and display the result in the message box.
31. Add a breakpoint in the ***LinqCustomImplementation.Where*** static method.
32. Run the application, make some transactions in EUR, MKD and USD and test if your method is called.

Discuss the custom LINQ implementation!!!

- ***Do you see know how the compiler translates the syntax into code?***
- ***Did you notice that we removed the System.Linq namespace? Why did we do that?***