Programming C# - **Exercises**

Contents

[Notes: 3](#_Toc104766188)

[Sending the solution 4](#_Toc104766189)

[Module 1. Basic data types and basic constucts/statements. 5](#_Toc104766190)

[Task 1. Data types 5](#_Toc104766191)

[Task 2. Casting 5](#_Toc104766192)

[Task 3. Play with Convert, Parse. 6](#_Toc104766193)

[Task 4. Concatenate strings 7](#_Toc104766194)

[Task 5. Implementing Conditional Logic 7](#_Toc104766195)

[Task 6. Loops. 8](#_Toc104766196)

[Task 7. Arrays. 9](#_Toc104766197)

[Module 2. Methods. Logging. Tracing. 9](#_Toc104766198)

[Task 1. Create methods. 9](#_Toc104766199)

[Task 2. Use ref. 9](#_Toc104766200)

[Task 3. Method overloading. 10](#_Toc104766201)

[Task 4. Try catch. 10](#_Toc104766202)

[Task 5. Windows Event Log. 11](#_Toc104766203)

[Task 6. Use Debug class. Use Trace class. 12](#_Toc104766204)

[Module 3. Enums, Structs. Collections. 14](#_Toc104766205)

[Task 1. Create enumeration Day with 7 values. 14](#_Toc104766206)

[Task 2. Create struct Coffee. 14](#_Toc104766207)

[Task 3. Getter/Setter 15](#_Toc104766208)

[Task 4. Indexer. 15](#_Toc104766209)

[Task 5. Use collections. Understand collections. (non generic) 15](#_Toc104766210)

[Task 6. Delegates/Events 16](#_Toc104766211)

[Task 7. Create calculator 16](#_Toc104766212)

[Module 4. Interfaces. Generics. Collections. 18](#_Toc104766213)

[Task 1. Static class. 19](#_Toc104766214)

[Task 2. Write custom generic collection. 19](#_Toc104766215)

[Task 3. Use generic collections. 20](#_Toc104766216)

[**Task 4\*. Implement custom collection "Red-Black Tree".** 21](#_Toc104766217)

[Module 5. Class hierarchies. 23](#_Toc104766218)

[Task 1. Class hierarchy. 23](#_Toc104766219)

[Task 2. Implement custom Exception 24](#_Toc104766220)

[Task 3. Inherit generic collection and specify type 24](#_Toc104766221)

[**Task 4\*. Linq to Objects** 25](#_Toc104766222)

[Module 6. Reading and Writing Local Data 29](#_Toc104766223)

[Task 1. Recursive Search. 29](#_Toc104766224)

[Task 2. File viewer/editor. 30](#_Toc104766225)

[**Task 3\*. Advanced File viewer/editor/finder/explorer.** 30](#_Toc104766226)

[Task 4. Reverse file lines. 32](#_Toc104766227)

[Task 5. Super Encryptor. 33](#_Toc104766228)

[**Task 6\*. Expressions** 33](#_Toc104766229)

[Task 7. Simple Json, XML serilization 35](#_Toc104766230)

[**Task 8\*. Json, XML, Binary serilization to network and file stream** 35](#_Toc104766231)

[Task 10. Most common words. 36](#_Toc104766232)

[Module 7. Accessing a Database 36](#_Toc104766233)

[Task 1. Code first. Console app. 36](#_Toc104766234)

[**Task 2\*. LINQ to EF. Desktop app. Database first. Simple paging, sorting and filtering with LINQ to EF.** 36](#_Toc104766235)

[Module 8. Access Remote Data 38](#_Toc104766236)

[**Task 1\*. Web Api to return data to client apps/systems** 38](#_Toc104766237)

[Module 9. Graphical UI apps. 39](#_Toc104766238)

[**Task 1\*\*. Advanced Calculator.** 39](#_Toc104766239)

[**Task 2\*\*\*. Graphics Calculator** 42](#_Toc104766240)

[**Task 3\*. Data-oriented Desktop App. CRUD operations. Paging, Sorting and Filtering.** 43](#_Toc104766241)

[Module 10. Multitasking and Multithreading. 44](#_Toc104766242)

[Task 1. Using Task<T> 44](#_Toc104766243)

[Task 2. Controlling Task Execution 45](#_Toc104766244)

[Task 3. Use Parallel.Invoke 45](#_Toc104766245)

[Task 4. Creating Awaitable Methods to download an image. 45](#_Toc104766246)

[Task 5\*. Synchronizing Concurrent Access to Data 46](#_Toc104766247)

[Module 11. IDisposable 46](#_Toc104766248)

[Task 1. Word count 46](#_Toc104766249)

[Module 12. Reusable Types and Assemblies - Reflection 46](#_Toc104766250)

[Task 1. Reflection investigation 46](#_Toc104766251)

[**Task 2\*. Advanced method invoker/reflector** 46](#_Toc104766252)

[Module 13. Encrypting and Decrypting Data (this module will not be covered in some courses). 47](#_Toc104766253)

[Task 1. Symmetric Encryption. Hashing. 47](#_Toc104766254)

[Module 14. SQL 48](#_Toc104766255)

[Task1. Querying. 48](#_Toc104766256)

# Notes:

1. Tasks.

All tasks below (in this document) should be implemented with Visual Studio 2017-2022, using language **C#**.

You can use .Net Framework 4.0+ (better .Net Framework 4.7/4.8) or .Net Core 2.1+ (better .Net 6.0).

Where ‘print’ is mentioned in tasks, it means writing in console, not using printing in physical printer; in other words: Print = Console.WriteLine()

If not specified, the task is to develop/create a console application. Otherwise, it is a desktop app (wpf or windows forms). But note that some tasks are web app or web services.

When you finish tasks for Module X after that you should finish the lab for Module X. The implementation order should be:

* Module 1 tasks (this document)
* Module 1 Lab
* Module 2 tasks (this document)
* Module 2 Lab
* …

For labs (official labs), see point B. (below)

Tasks marked with star(\*) like in:

Task 6\*. Expressions

are more difficult task. You may/should start them only if you complete the others (more easier) tasks for the current module.

1. Labs.

For each module, first finish the tasks, after that finish the official labs in the course:

Lab files:

<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Allfiles>

Lab Instructions:

<https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Instructions>

For example for lab’s instructions in Module 2 use: [20483C\_MOD02\_LAB\_MANUAL.md](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/blob/master/Instructions/20483C_MOD02_LAB_MANUAL.md) and use lab files [Mod02](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Allfiles/Mod02)/**Labfiles, where** [Starter](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Allfiles/Mod02/Labfiles/Starter) is lab files when you start, [Solution](https://github.com/MicrosoftLearning/20483-Programming-in-C-Sharp/tree/master/Allfiles/Mod02/Labfiles/Solution) is target/completed lab files.

VS = Visual Studio 2017/2019/2022

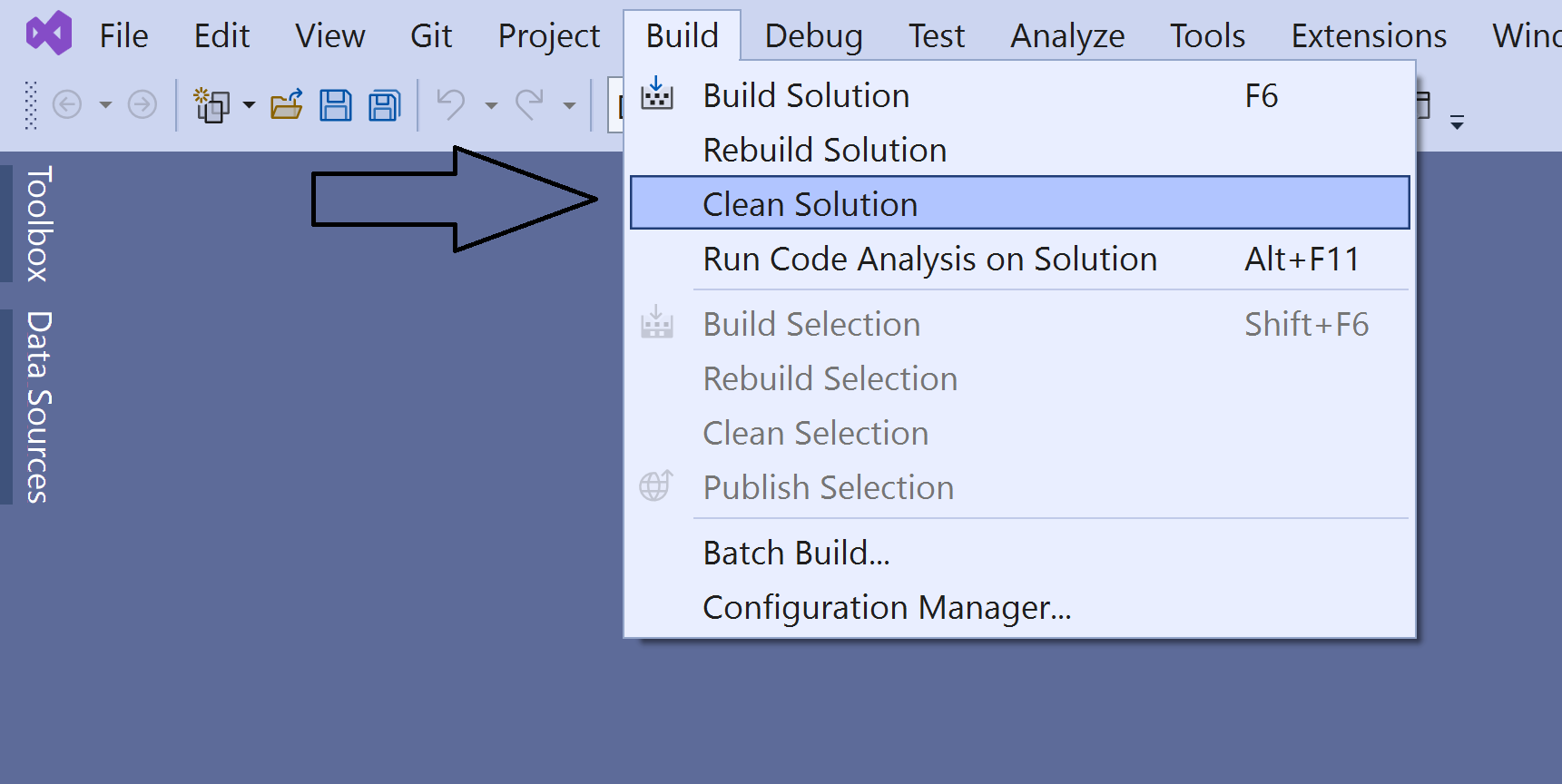
# Sending the solution

You can send the solutions in my email ([hristo256@gmail.com](mailto:hristo256@gmail.com); [hristo.yonchev@outlook.com](mailto:hristo.yonchev@outlook.com) ), google drive, dropbox or other location for which you can send me a link.

You can use single VS solution per module with multiple projects in it, or single solution per task. It is up to you.

Note that before you send me the solutions you need to clean the solution in Visual Studio - this will remove the executable files from your Debug/Release folder.

How to clean the Visual Studio solution:



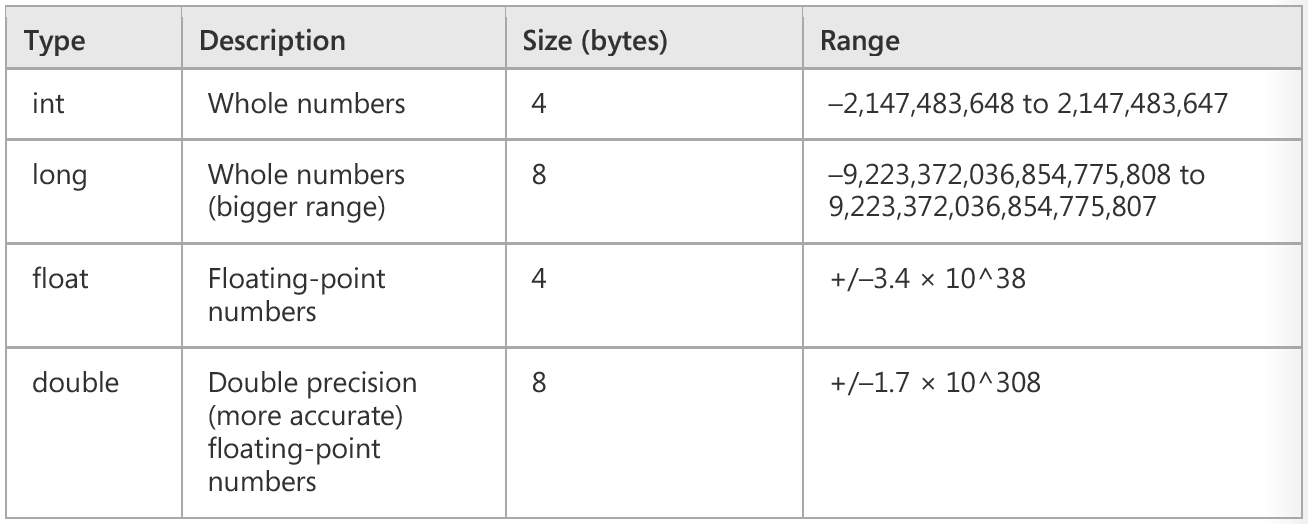
Exercises/Tasks

# Module 1. Basic data types and basic constucts/statements.

## Task 1. Data types

Write console application.

For all types below, try to print min and max values allowed. (for example int.MinValue). Add a number to the max value and remove number from the min value; print the results. Do this with or without **checked** statement.

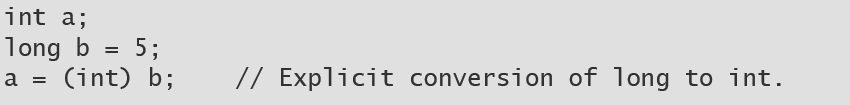


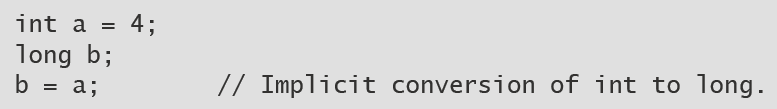


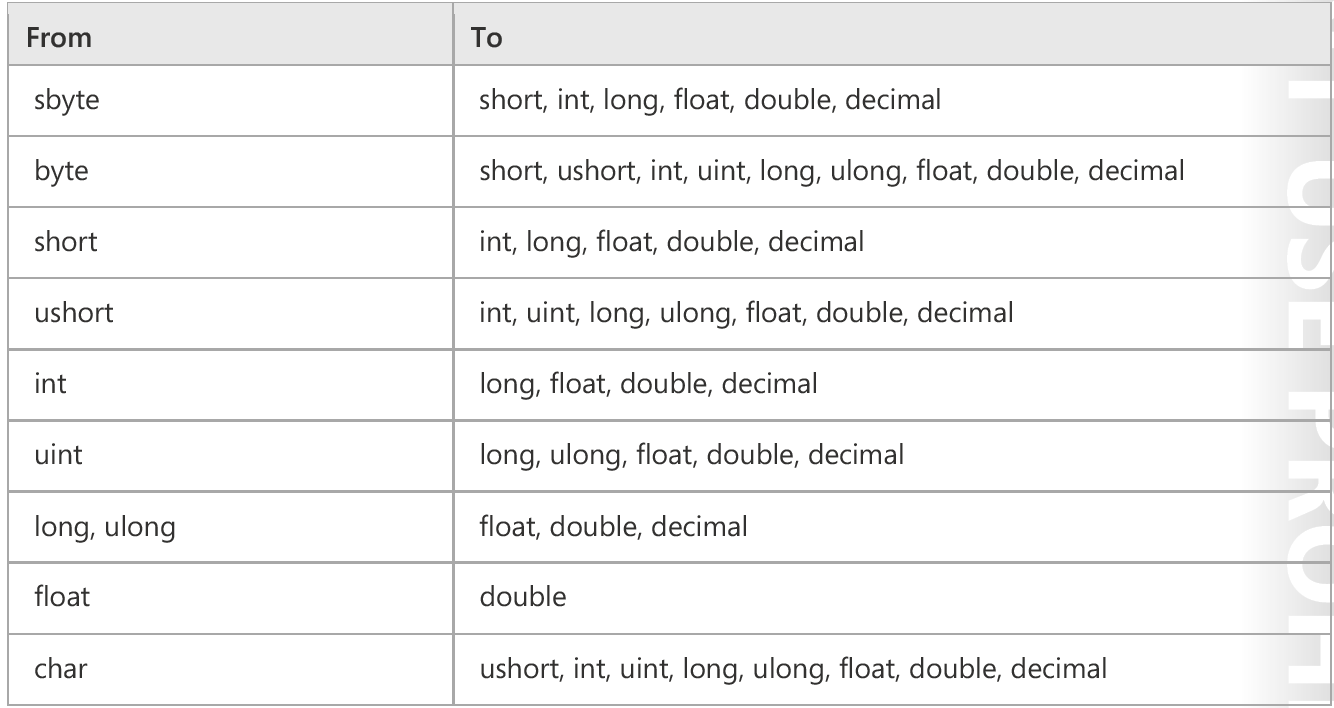
## Task 2. Casting

Write console application.

1. Try implicit and explicit conversions between one type from ‘From’ column and one type from ‘To’ column.
2. Cast between floating point numbers and whole numbers.







## Task 3. Play with Convert, Parse.

Write console application.

A.

Play with methods in Convert class, for example Convert.ToInt32():

string possibleInt = "1234";

int count = Convert.ToInt32(possibleInt);

Convert to/from:

* String <-> bool
* int <-> bool
* string <-> int
* floating point number <-> string
* Datetime <-> string

B.

Use primitive types Parse method for all numbers, bools and Datetime values.

## Task 4. Concatenate strings

Write console application.

A.

Use strifng and StringBuilder to concatate strings. Print the result to the console.

Example:

StringBuilder address = new StringBuilder();

address.Append("23");

address.Append(", Main Street");

address.Append(", Buffalo");

string concatenatedAddress = address.ToString();

B.

Compare the performance with regular string concatenation when you need to join 1000 strings.

## Task 5. Implementing Conditional Logic

Write console application.

Write sample application (up to you) and try IF and Switch/Case statements; use at least 4 branches.

Examples:

if (response == "connection\_failed") {. . .}

else if (response == "connection\_error") {. . .}

else { }

switch (response)

{

case "connection\_failed":

. . .

break;

case "connection\_success":

. . .

break;

default:

. . .

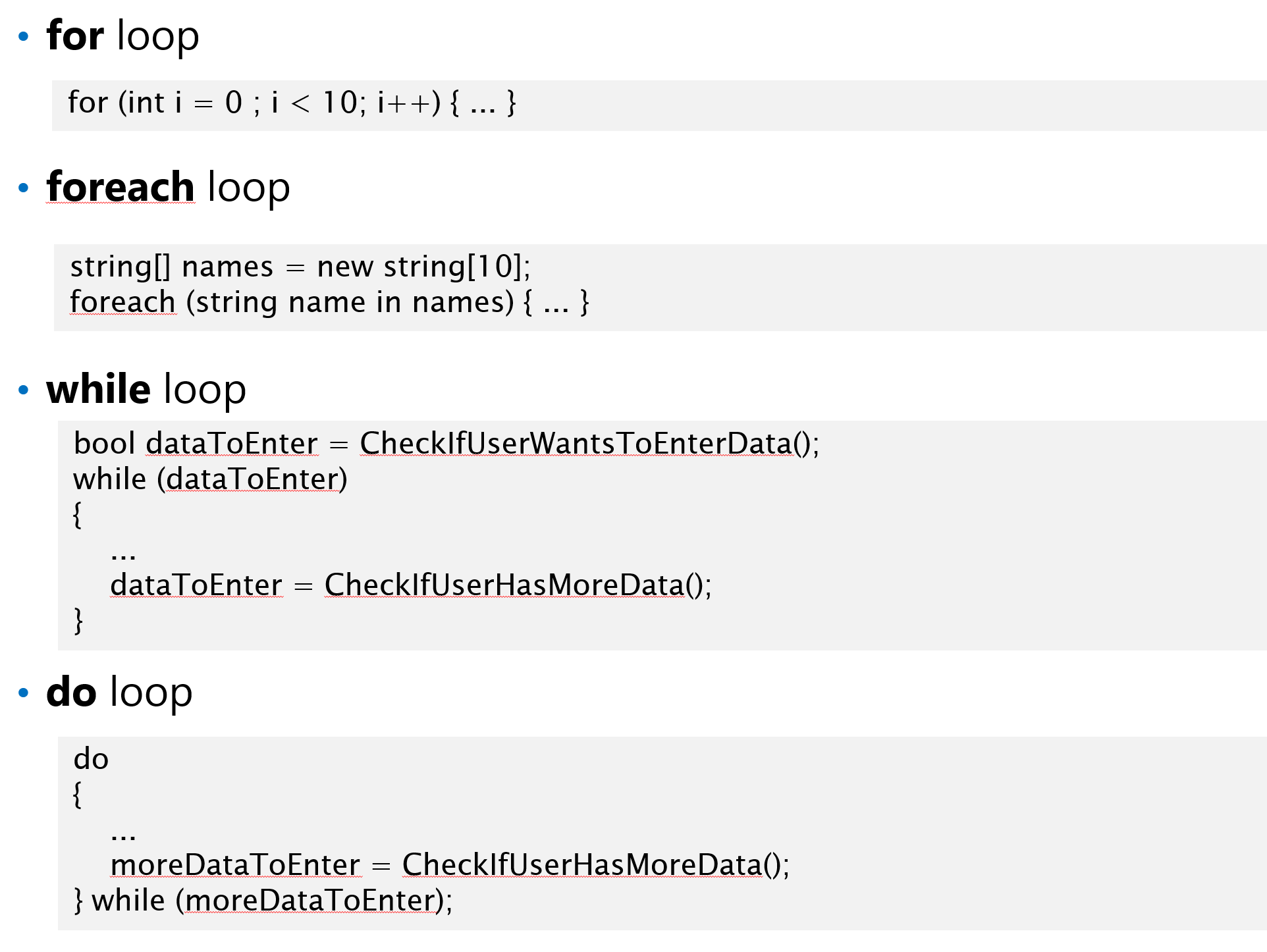
break;

}

## Task 6. Loops.

1.

Use all 4 loop statements in sample application (up to you):



2. Create a collection with N random items and use the all 4 loop statements to iterate:

* all elements
* every element in even position (where index%2==0).
* elements in reverse order

(Random class: <https://docs.microsoft.com/en-us/dotnet/api/system.random?view=net-6.0>)

## Task 7. Arrays.

Write application where you are using:

* + -dimensional arrays
  + Multidimensional arrays Single
  + Jagged arrays

Put break point inside the loop body. Debug the iterations.



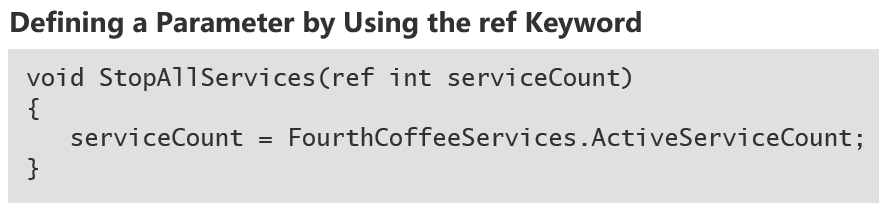
# Module 2. Methods. Logging. Tracing.

## Task 1. Create methods.

Create method with 2 params -> int, bool; the method should return 2 values: string, string.

## Task 2. Use ref.

Use ref with int param, check if value is changed in the method after method finish:



## Task 3. Method overloading.

Use method overloading to create 3 methods with the same name.

A)

* 0 params
* 1 param
* 2 param -> int, string
* 2 param -> string, int

B)

Use named and optional params. Example:

void StopService(

bool forceStop,

string serviceName = null,

int serviceId =1)

{

...

}

var forceStop = true;

StopService(forceStop);

// OR

var forceStop = true;

var serviceName = "FourthCoffee.SalesService";

StopService(forceStop, serviceName);

## Task 4. Try catch.

Use try catch block with at least 3 catch clauses and 1 finally.

Example:

try

{

}

catch (NullReferenceException ex)

{

// Catch all NullReferenceException exceptions.

}

catch (Exception ex)

{

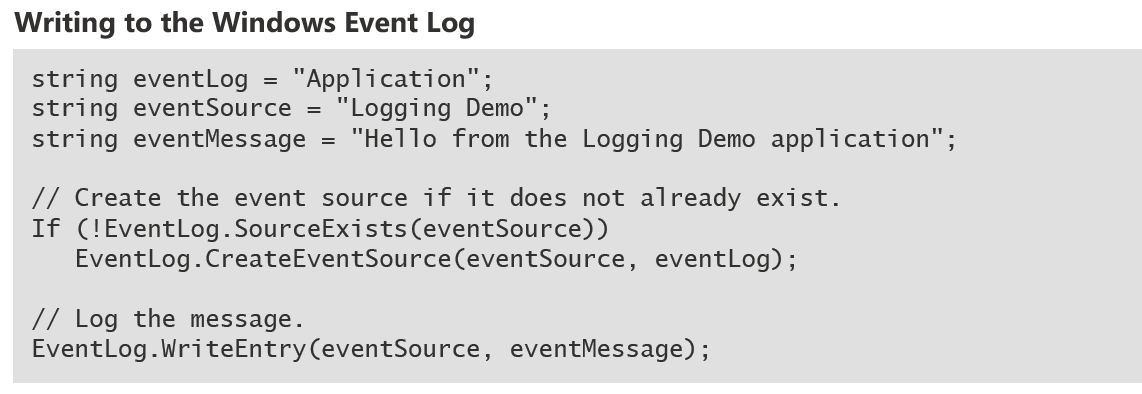
// Catch all other exceptions.

}

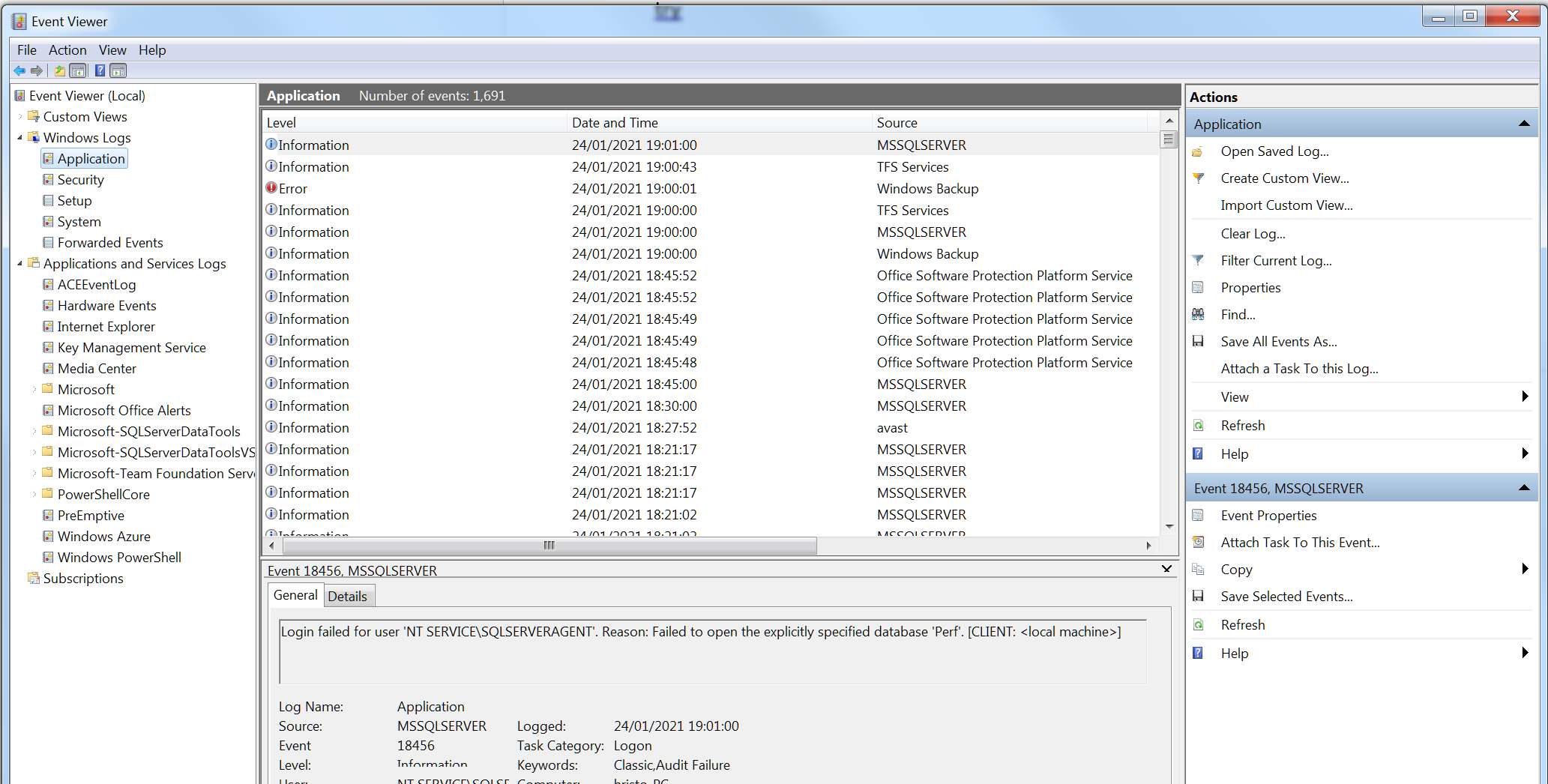
In “try” block ask the user which exception to throw, and in catch clause print the exception.

## Task 5. Windows Event Log.

Write messages to Windows Event Log. Example:

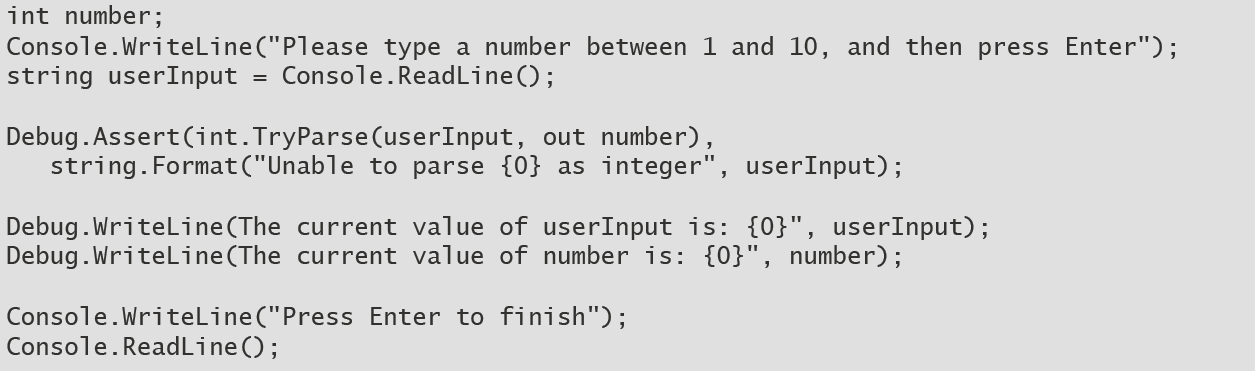


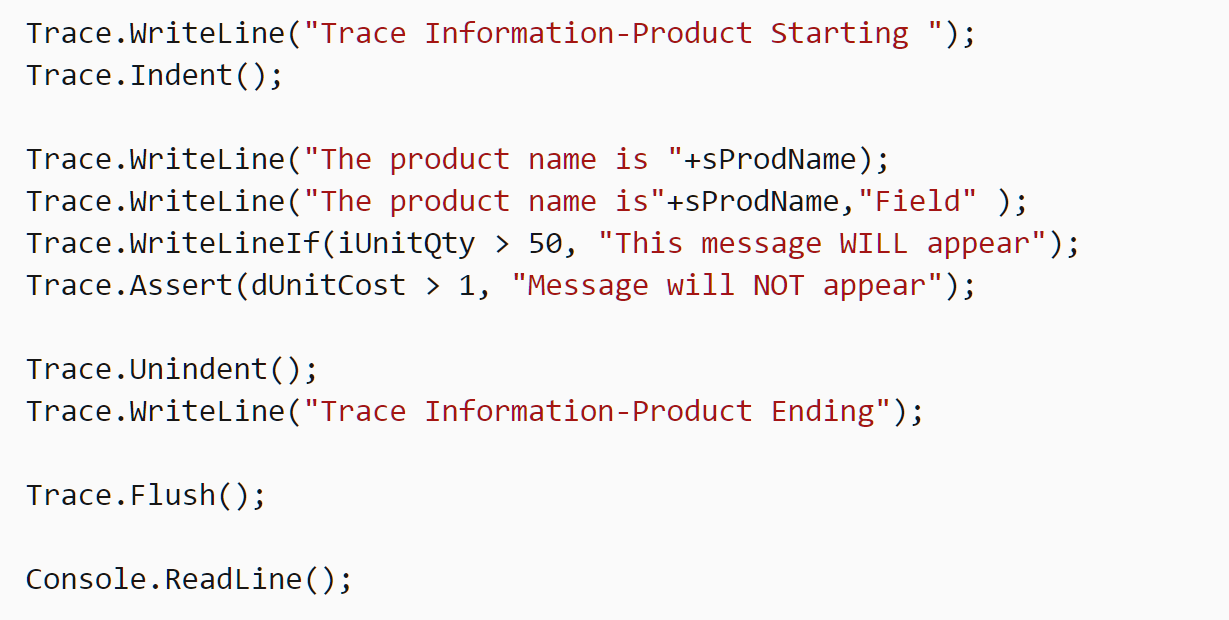
Check the messages in Windows Event Viewer:



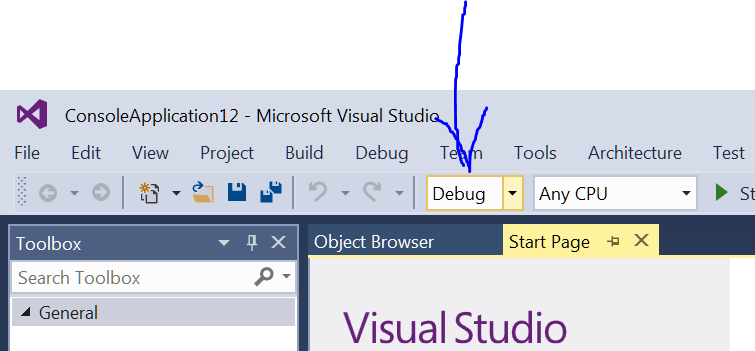
## Task 6. Use Debug class. Use Trace class.

Check which messages appear in Debug and which in Release mode.



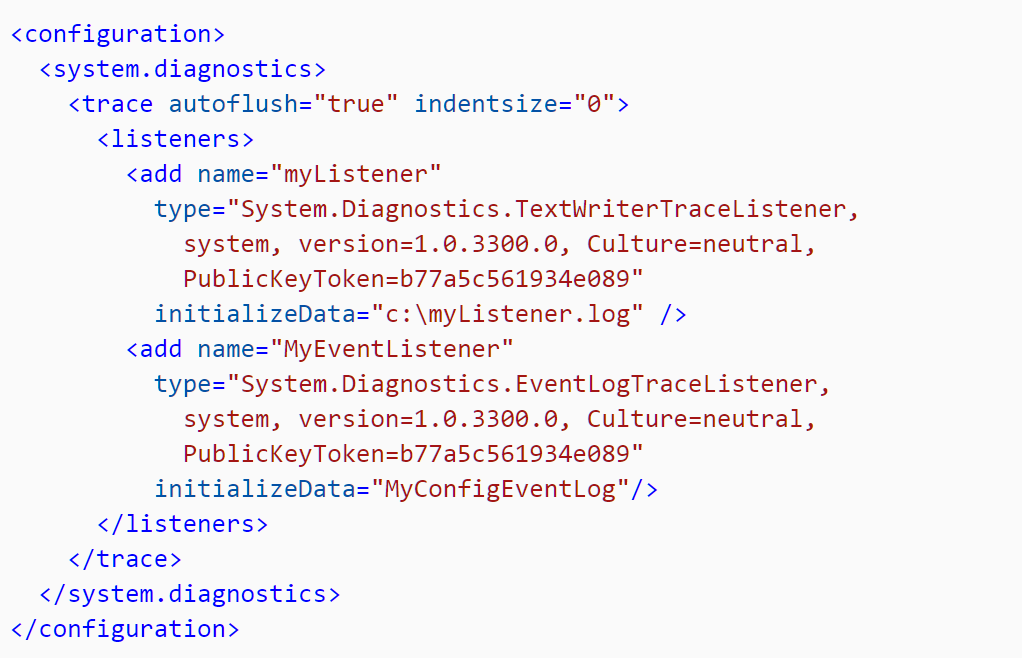


Change Debug/Release in VS and run the app in both modes:



Next:

Add 2 more trace listeners using configuration:



Check if the result goes to the File and also in Event Log.

# Module 3. Enums, Structs. Collections.

## Task 1. Create enumeration Day with 7 values.

enum Day { .. }

use it in IF/Else and also in switch/case statements.

## Task 2. Create struct Coffee.

Create struct Coffee with 3 properties/fields. Later create an instance. Pass this instance to method and change it in the method. Check the values after method is finished/completed/executed.

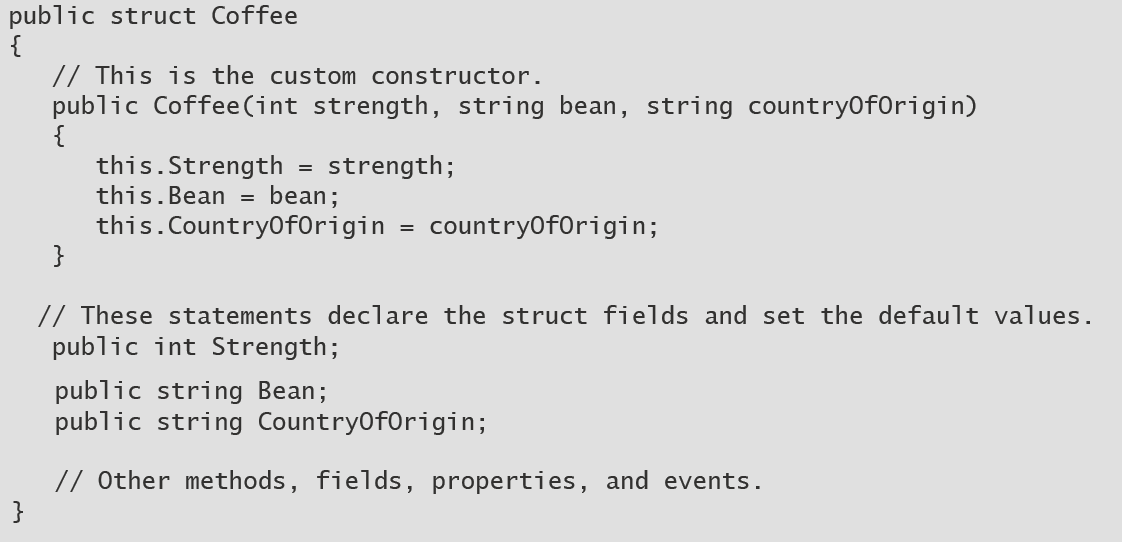
Example:

Coffee coffee1 = new Coffee();

Change(coffe1);

Print(coffee1);

Sample struct definition:



## Task 3. Getter/Setter

If in previous task you used fields, change them to properties. In every **get** print the value in console. In every **set** print the previous and next values. (You should have in total 3\*2=6 getters and setters).

In setters, if strings are null or empty, throw exceptions. If numbers are negative throw exception.

Try to invoke/call all getters and setters.

## Task 4. Indexer.

Create struct/class Menu and add 2 indexers to search Drinks by Id (int) and by Name (string). Drink is struct/class with 2 properties. Drink has indexer to return ingredients. Later should be able to invoke the menu in this manner:

Menu menu = new Menu();

string secretIngredient = menu[“coca cola”][3];

## Task 5. Use collections. Understand collections. (non generic)

Play with next 10 collections.

**For each** collection:

1. Add 100 elements/items
2. Print all elements – 100 elements – and check their order.
3. Remove first 3 and/or last 3 elements.
4. Print all remaining elements.

Collections to use:

1. ArrayList
2. Hashtable
3. Queue
4. Stack
5. SortedList
6. ListDictionary
7. HybridDictionary
8. OrderedDictionary
9. NameValueCollection
10. StringCollection

## Task 6. Delegates/Events

Define delegate with this signature:

void OutOfBeansHandler(Coffee, EventArgs)

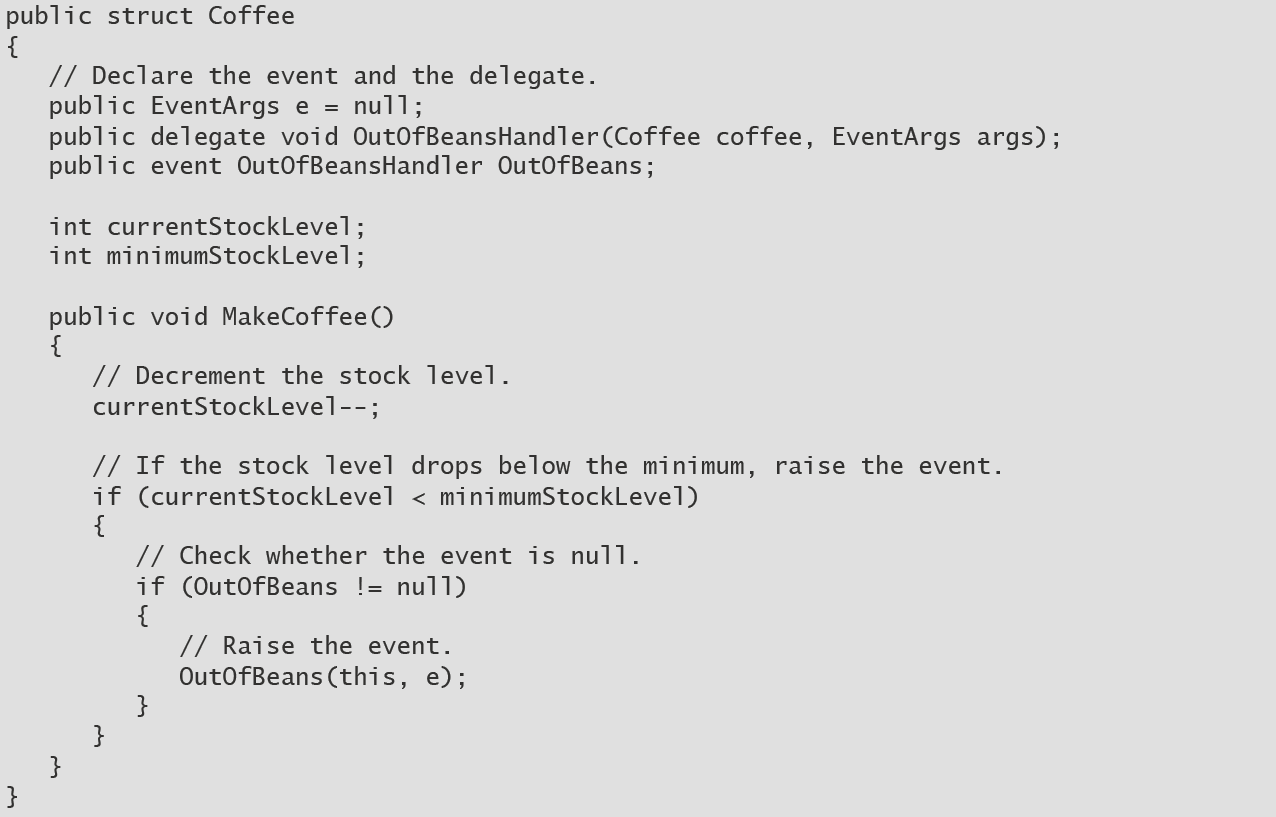
Create variable of type the delegate above. Add 5 methods in the variable. Invoke the variable and check if 5 methods are invoked.

Create class/struct with 1 event of type OutOfBeansHandler.

Add 2 subscribers/handlers. Raise the event when method MakeCoffee is invoked.

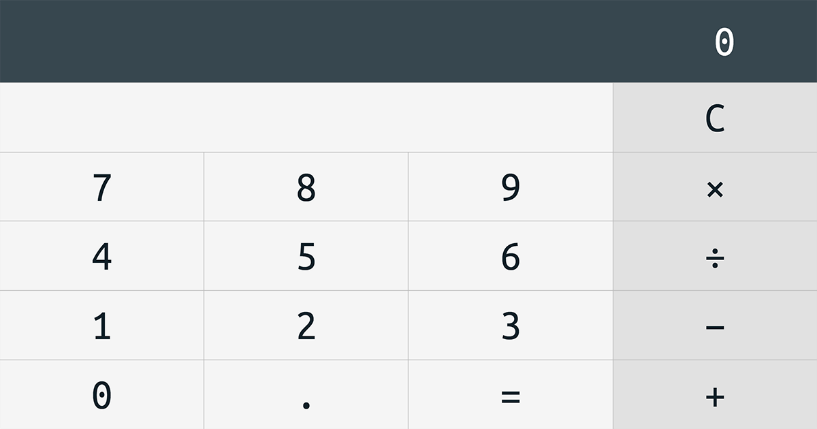
coffee1.OutOfBeans += HandleOutOfBeans;

Example:



## Task 7. Create calculator

Create calculator using WPF or Windows Forms (your choice) (if you are familiar with Desktop application development); The UI is just 1 textbox and many buttons. Example:



Draw the UI using VS designer using Buttons. Subscribe for Click events using C# (for example

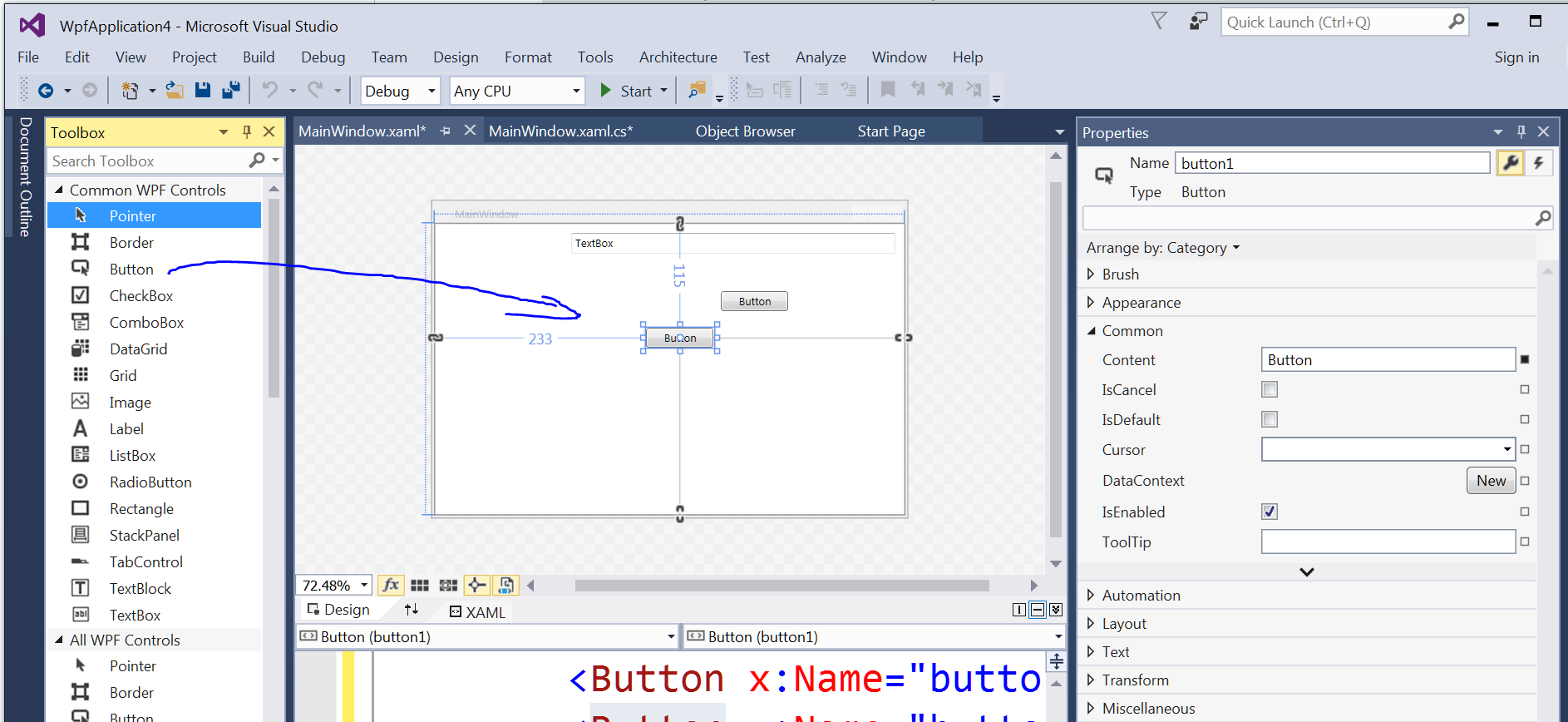
button1.Click+=Add;

button2.Click+=o.Add;

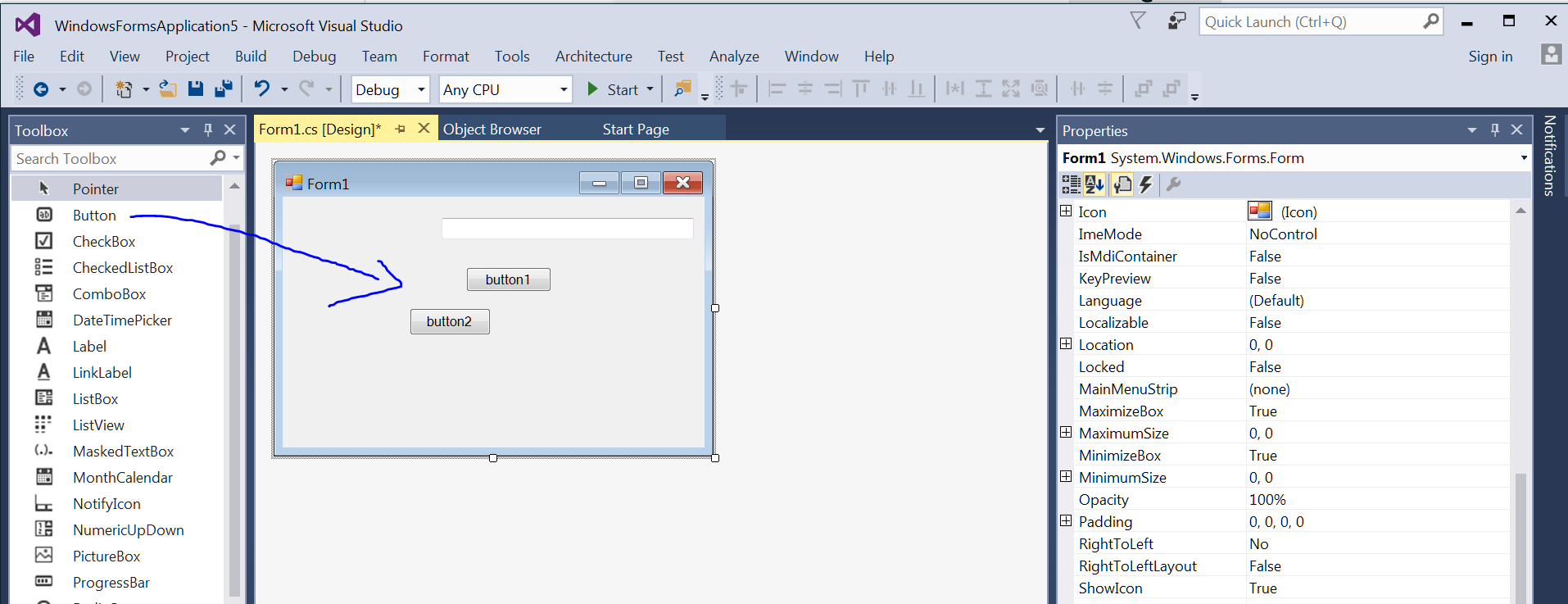
fiveButton.Click+=five;

numberButton.Click+=SomeNumberClicked;).

Use WPF designer + Toolbox:



Or use Windows forms designer + Toolbox:



(To read the value of the textbox, use this.textBox.Text, where textbox is the name of the TextBox UI control; You need to parse the text; Use Parse, or TryParse methods).

Use try/catch for errors handling. The app should not stop/crash on error. It should display “Error” in the textbox!

The calculator support numbers with max digits 28-29 digits (use **decimal** type). Textbox shows 30 symbols.

“C” button clears the result and display “0”.

Sample calculations:

1+1 = 2

1/2 = 0.5

5/ 12341234123 = 0.0000000004051458671124020779

12341234123123 \* 2222222 = 27424961975554639306

All calculations/operations should be done in class Calculator’s methods (one method for each operation).

You can implement keyboard shortcuts:

* Esc – clear the textbox and put 0 as value.
* 0-9 key – enter number
* . – decimal separator – only 1 is allowed
* +, - ,\* ,/ keys – arithmetic operation
* = key – evaluate the calculation

# Module 4. Interfaces. Generics. Collections.

## Task 1. Static class.

Create custom static class called Mathematics (without using built-in class Math) with these members(method and properties):

double E = 2.7182818284590451;

double PI = 3.1415926535897931;

//Returns the absolute value of a System.Decimal number.

decimal Abs(decimal value);

//Produces the full product of two 32-bit numbers.

static long BigMul(int a, int b);

//Returns the smallest integral value that is greater than or equal to the specified

// decimal number.

double Ceiling(double d);

//Returns the largest integer less than or equal to the specified decimal number.

double Floor(double d);

//Returns e raised to the specified power.

double Exp(double d);

Returns the larger/smaller of two double-precision floating-point numbers.

double Max(double val1, double val2);

double Min(double val1, double val2);

// Rounds a decimal value to the nearest integral value.

double Round(double a);

// Returns the sine of the specified angle.

double Sin(double a);

// Returns the cosine of the specified angle.

double Cos(double d);

// Returns the tangent of the specified angle.

double Tan(double a);

Also add overloads for double, decimal, int where necessary!

Write unit tests for all methods (if we already discussed unit-testing).

## Task 2. Write custom generic collection.

Write custom generic collection (which uses an array as private collection) which implements IEnumerable<T>, ICollection<T>, IList<T> with following memebers:

int Count { get; }

void Add(T item);

void Clear();

bool Contains(T item);

void CopyTo(T[] array, int arrayIndex);

bool Remove(T item);

IEnumerator<T> GetEnumerator();

T this[int index] { get; set; }

int IndexOf(T item);

void Insert(int index, T item);

void RemoveAt(int index);

void Sort();

void Sort(Comparison<T> comparison);

void Sort(IComparer<T> comparer);

int Capacity { get; set; }

int Count { get; }

void AddRange(IEnumerable<T> collection);

void ForEach(Action<T> action);

void Reverse();

Note:

delegate int Comparison<in T>(T x, T y);

public interface IComparer<in T>

{

int Compare(T x, T y);

}

Write unit tests for each operation (if we already discussed unit-testing)!

## Task 3. Use generic collections.

Play with next 7 collections.

For each collection:

1. Add 10 elements/items
2. Print all elements – 10 elements – and check their order.
3. Remove first 3 and/or last 3 elements.
4. Print all remaining elements.

Collections to use:

1. List<T>
2. LinkedList<T>
3. Queue<T>
4. Stack<T>
5. **Dictionary<TKey, TValue>**
6. **SortedList<TKey, TValue>**
7. **SortedDictionary<TKey, TValue**>

## **Task 4\*. Implement custom collection "Red-Black Tree".**

Implement custom generic collection "Red-Black Tree" using C#.

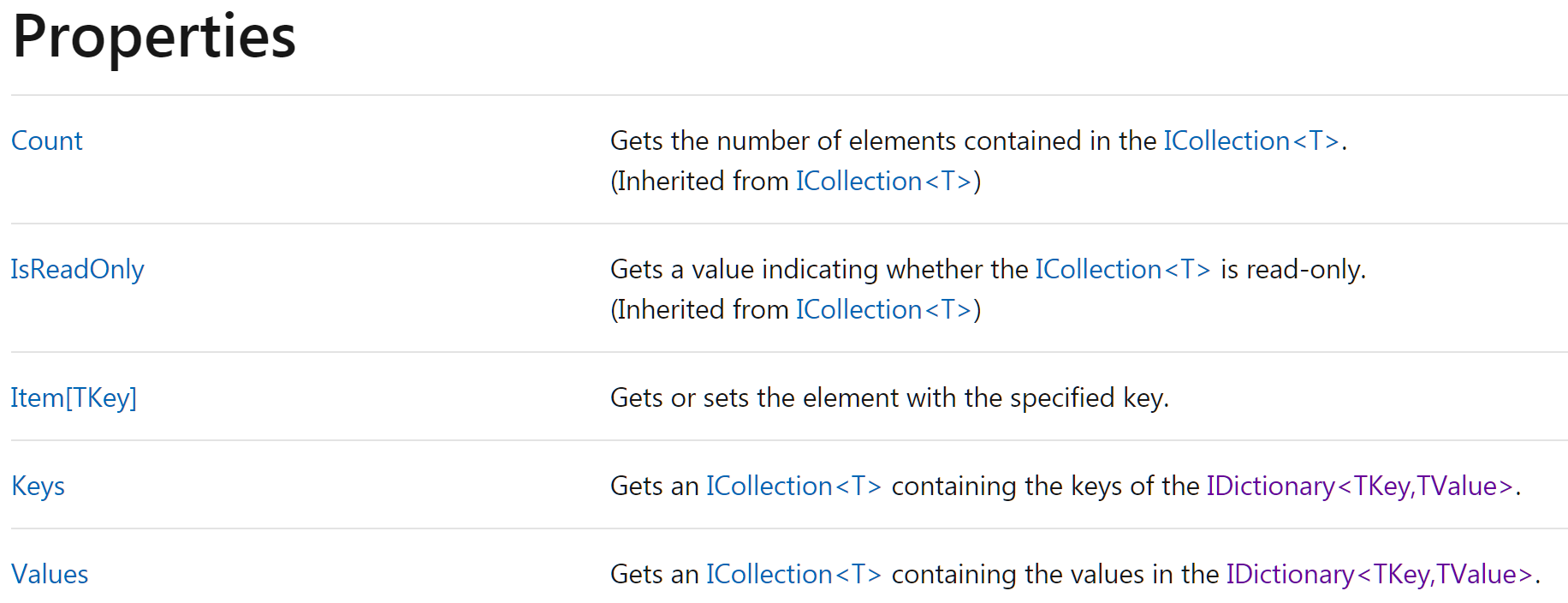
class RedBlackTree<Key,Value>

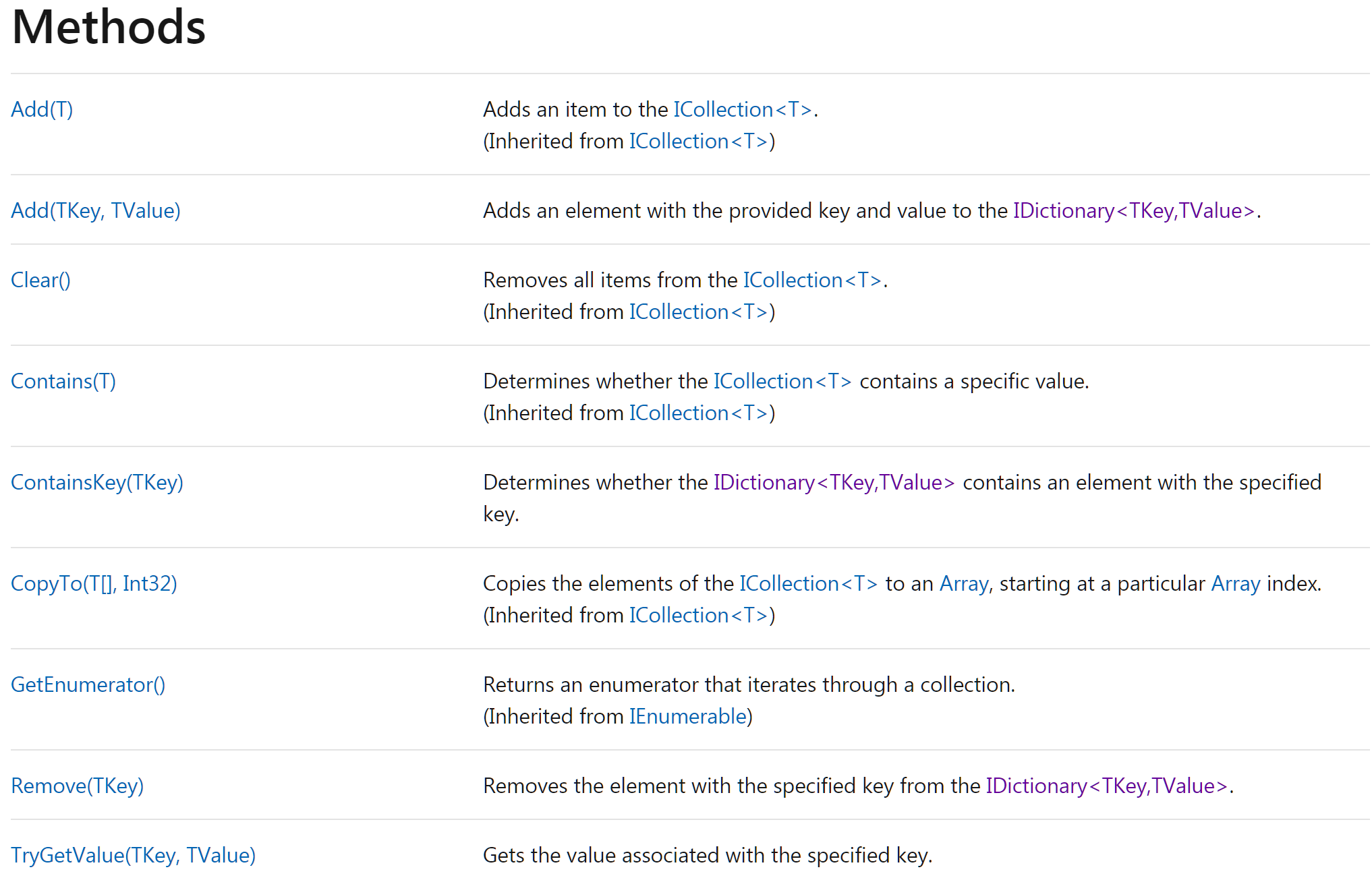
It should implement .net IDictionary<Key,Value> (properties and methods are listed below)

[IDictionary<TKey,TValue> Interface (System.Collections.Generic) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2?view=net-5.0)

It should guarantees searching in O(log n) time, where n is the number of nodes of the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log n) time.

The required members (at least these operations, you can add your own):



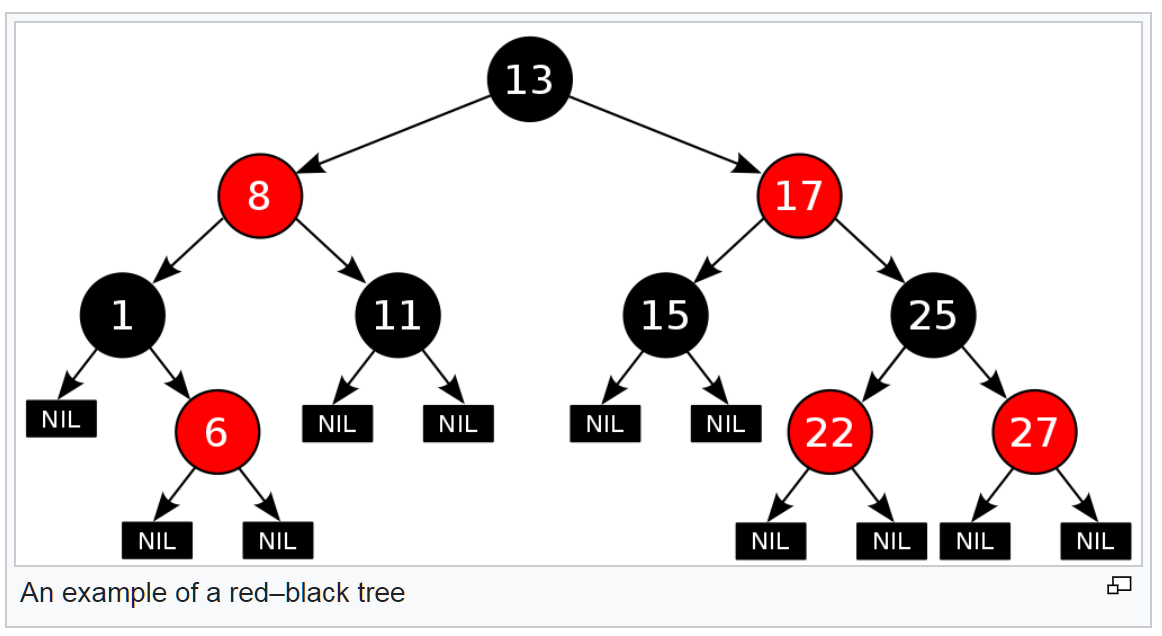


*(In computer science, a red–black tree is a kind of self-balancing binary search tree. Each node stores an extra bit representing "color" ("red" or "black"), used to ensure that the tree remains approximately balanced during insertions and deletions.*

*When the tree is modified, the new tree is rearranged and "repainted" to restore the coloring properties that constrain how unbalanced the tree can become in the worst case. The properties are designed such that this rearranging and recoloring can be performed efficiently.*

*The re-balancing is not perfect, but guarantees searching in O(log n) time, where n is the number of nodes of the tree. The insertion and deletion operations, along with the tree rearrangement and recoloring, are also performed in O(log n) time.*

*Tracking the color of each node requires only 1 bit of information per node because there are only two colors. The tree does not contain any other data specific to its being a red–black tree so its memory footprint is almost identical to a classic (uncolored) binary search tree. In many cases, the additional bit of information can be stored at no additional memory cost.)*



Additional info:

[Red–black tree - Wikipedia](https://en.wikipedia.org/wiki/Red%E2%80%93black_tree)

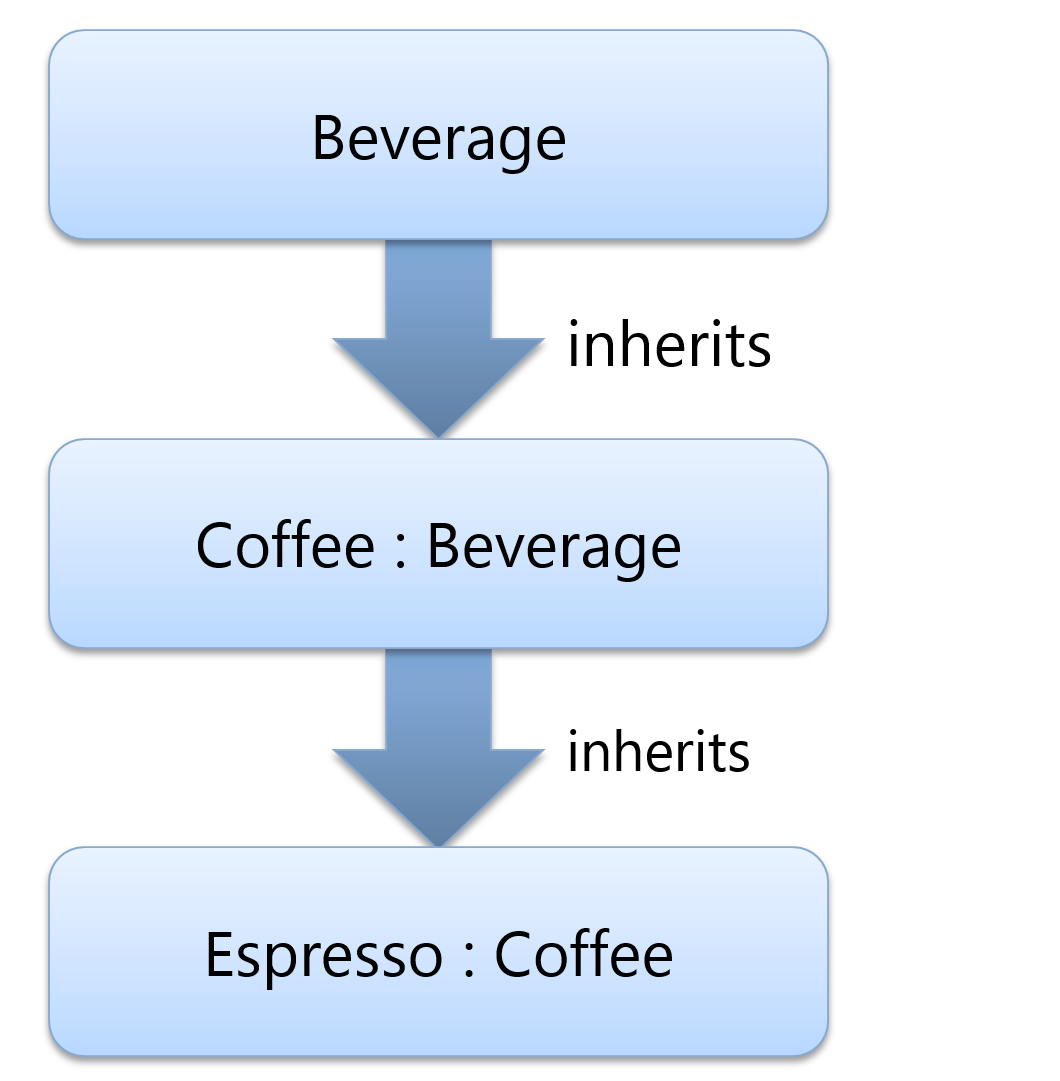
[Working With Red-Black Trees In C# (c-sharpcorner.com)](https://www.c-sharpcorner.com/article/working-with-red-black-trees-in-c-sharp/)

<https://docs.microsoft.com/en-us/dotnet/api/system.collections.generic.idictionary-2?view=net-5.0>

# Module 5. Class hierarchies.

## Task 1. Class hierarchy.

Write a class hierarchy where a class named Espresso inherits from a class named Coffee, which in turn inherits from a class named Beverage:



Notes:

Beverage is abstract.

Add class Tea : Beverage. Tea is sealed.

Add virtual int GetServingTemperature() in Beverage and later override it in all classes. Invoke base class implementation! (base.GetServingTemperature();)

Add constructors which invoke base class constructor, like:

public Coffee(string name, bool isFairTrade, int temp)

: base(name, isFairTrade, servingTemp)

Create a collection of Beverages, using List<T>. Add different objects, and in loop (for or foreach) invoke GetServingTemperature method. Check the results.

## Task 2. Implement custom Exception

Implement custom Exception called SuperDetailedException.

1. Inherit from the **System.Exception** class
2. Implement three standard constructors:

base()

base(string message)

base(string message, Exception inner)

1. Add additional 3 members -> Details (string), Date Time (when happened), the name of the method which throw the exception.

Create 2 sample methods which throw and catch this exception. Method 1 throw it, Method 2 catch the exception and print all properties.

## Task 3. Inherit generic collection and specify type

Inherit next collection and specify the type of T, TKey, TValue

T, TKey = int

TValue = string.

1. List<T>
2. LinkedList<T>
3. Queue<T>
4. Stack<T>
5. **Dictionary<TKey, TValue>**
6. **SortedList<TKey, TValue>**
7. **SortedDictionary<TKey, TValue**>

## **Task 4\*. Linq to Objects**

Suppose you have these classes in C#:

public enum **Countries** {

    USA,

    Italy,

}

public class **Customer** {

    public string Name;

    public string City;

    public Countries Country;

    public Order[] Orders;

    public override string ToString() {

        return String.Format("Name: {0} – City: {1} – Country: {2}",

        this.Name, this.City, this.Country );

    }

}

public class **Order** {

    public int IdOrder;

    public int Quantity;

    public bool Shipped;

    public string Month;

    public int IdProduct;

public class **Product** {

    public int IdProduct;

    public decimal Price;

    public override string ToString() {

       return String.Format("IdProduct: {0} – Price: {1}", this.IdProduct,

         this.Price );

    }

}

Also, suppose you have these collection:

var **customers** = new Customer[] {

  new Customer {Name = "Paolo", City = "Brescia",

             Country = Countries.Italy, Orders = new Order[] {

                 new Order { IdOrder = 1, Quantity = 3, IdProduct = 1 ,

                             Shipped = false, Month = "January"},

                 new Order { IdOrder = 2, Quantity = 5, IdProduct = 2 ,

                             Shipped = true, Month = "May"}}},

  new Customer {Name = "Marco", City = "Torino",

             Country = Countries.Italy, Orders = new Order[] {

                 new Order { IdOrder = 3, Quantity = 10, IdProduct = 1 ,

                             Shipped = false, Month = "July"},

                 new Order { IdOrder = 4, Quantity = 20, IdProduct = 3 ,

                             Shipped = true, Month = "December"}}},

  new Customer {Name = "James", City = "Dallas",

             Country = Countries.USA, Orders = new Order[] {

                 new Order { IdOrder = 5, Quantity = 20, IdProduct = 3 ,

                             Shipped = true, Month = "December"}}},

  new Customer {Name = "Frank", City = "Seattle",

             Country = Countries.USA, Orders = new Order[] {

                 new Order { IdOrder = 6, Quantity = 20, IdProduct = 5 ,

                             Shipped = false, Month = "July"}}}};

var **products** = new Product[] {

    new Product {IdProduct = 1, Price = 10 },

    new Product {IdProduct = 2, Price = 20 },

    new Product {IdProduct = 3, Price = 30 },

    new Product {IdProduct = 4, Price = 40 },

    new Product {IdProduct = 5, Price = 50 },

    new Product {IdProduct = 6, Price = 60 }};

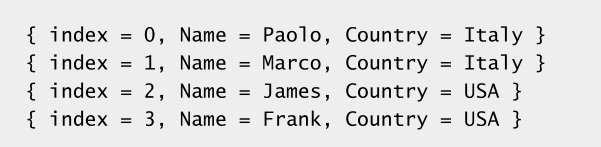
The task:

Write LINQ queries against the collections above (LINQ to Objects).

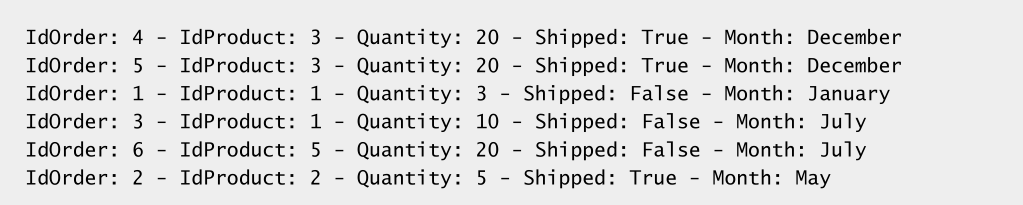
For some queries you have sample output.

Write LINQ queries that will select/filter from collection above and later print the result in the console:

1. Select all customers, and print their index in the collection. The output should be something like:



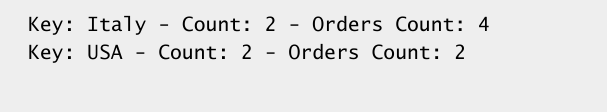
1. Select all customers from Italy
2. Select all customers from Italy without first and last customer (list order – where with index)
3. Select only name and country of all the customers
4. Use SelectMany() to select all customer’s orders.
5. Select all orders for customers from Italy. Sample output:



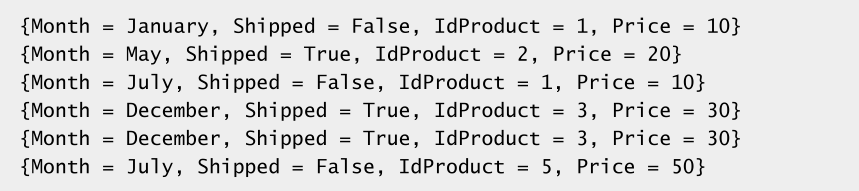
1. Select all customers and their orders for customers from Italy
2. Select all customers and their orders for customers from Italy ordered by customer name and customer city
3. Group customers by country. Sample output:



1. Group customers by country – print only country and ‘count’ of customers per country. The result should looks like:

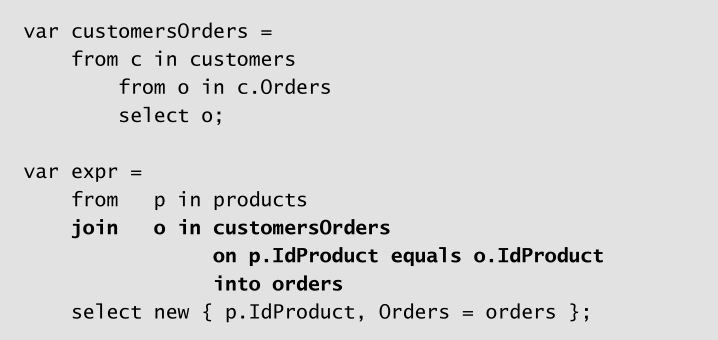


1. Join customer’s orders and products; select order’s Month, is order shipped, product id and product price. The result should looks like:

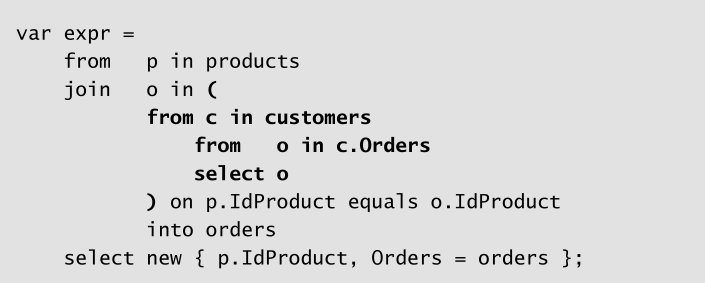


1. Compare the output of ‘expr’ in both samples:

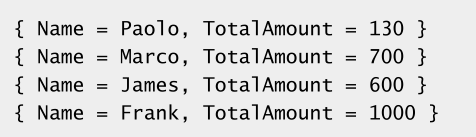
Sample 1:



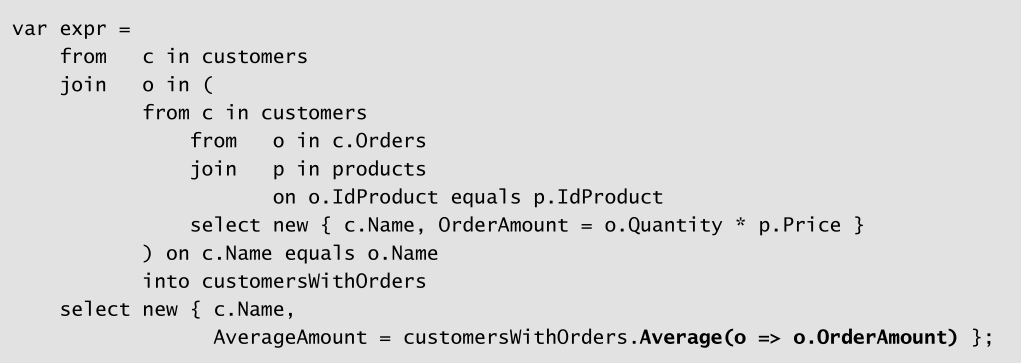
Sample 2:



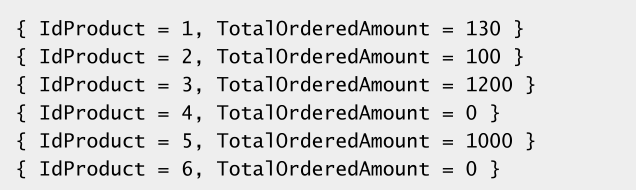
1. Select customers and the total sum of orders for each customer. Sample output:



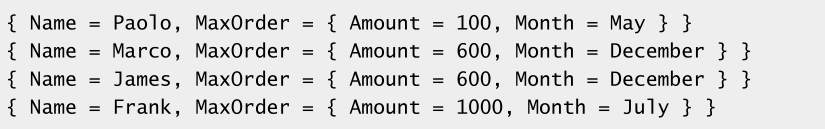
1. Select min, max, avg order quantity per customer. You can use these query as an example for avg (Average):



1. Select ‘Total Ordered Amount’ per product. The output should looks like this:



1. Select “Customers and their most expensive orders paired with the month of execution”. The output should looks like this:



# Module 6. Reading and Writing Local Data

## Task 1. Recursive Search.

For drive c:\ , find all \*.txt or \*.dll files; recursively. For all of them, print:

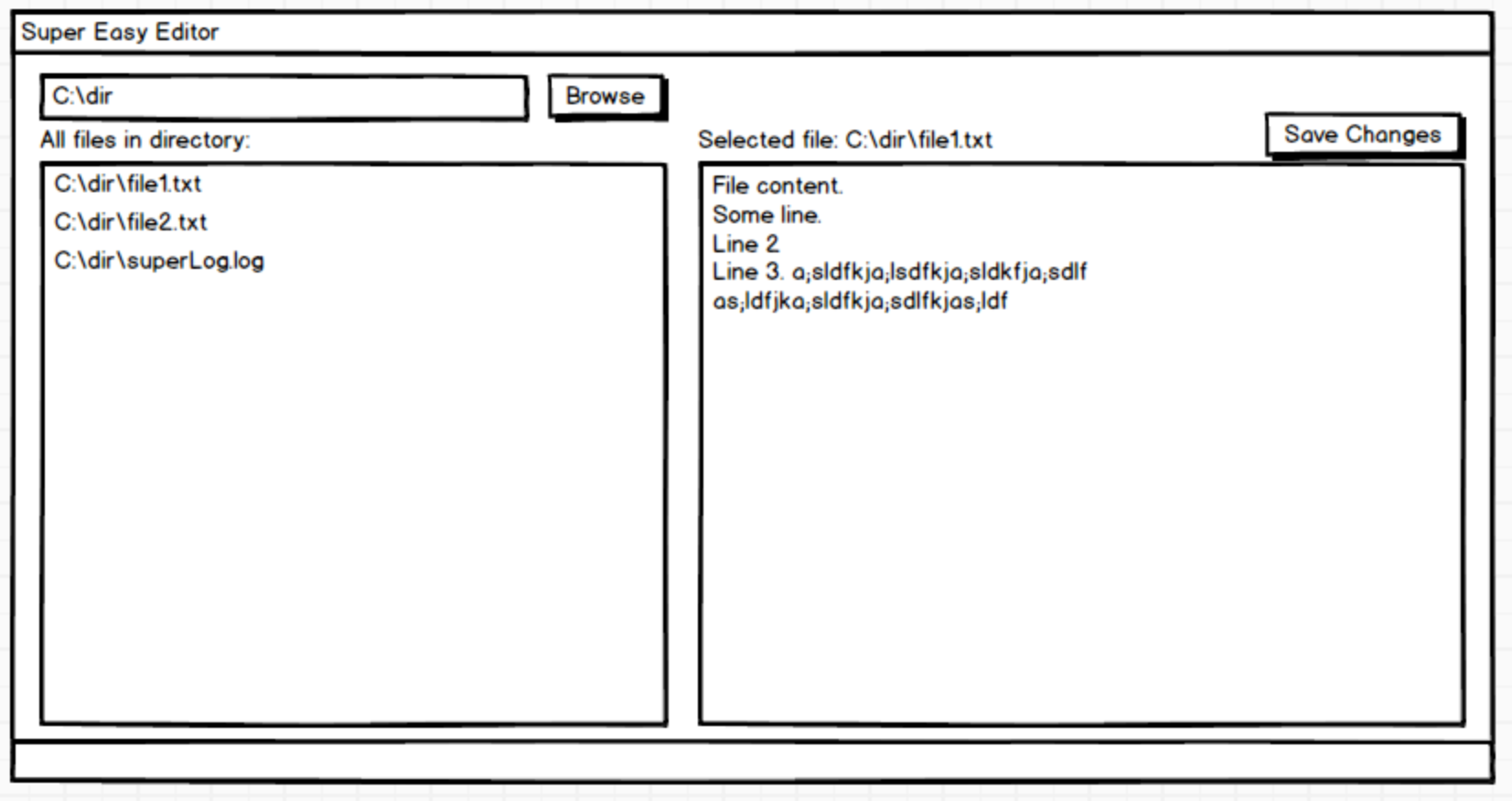
1. Name
2. Extension (.txt or .dll)
3. Size
4. Date of creation
5. Date of last modification

Notes:

For some folders you can receive exception. Handle this exception and show error message for this folder.

## Task 2. File viewer/editor.

Create simple desktop app (wpf or windows forms – you choose, up to you) which show the list of \*.txt, \*.log files in directory. It will allow users to view/edit the files when the user select the file.



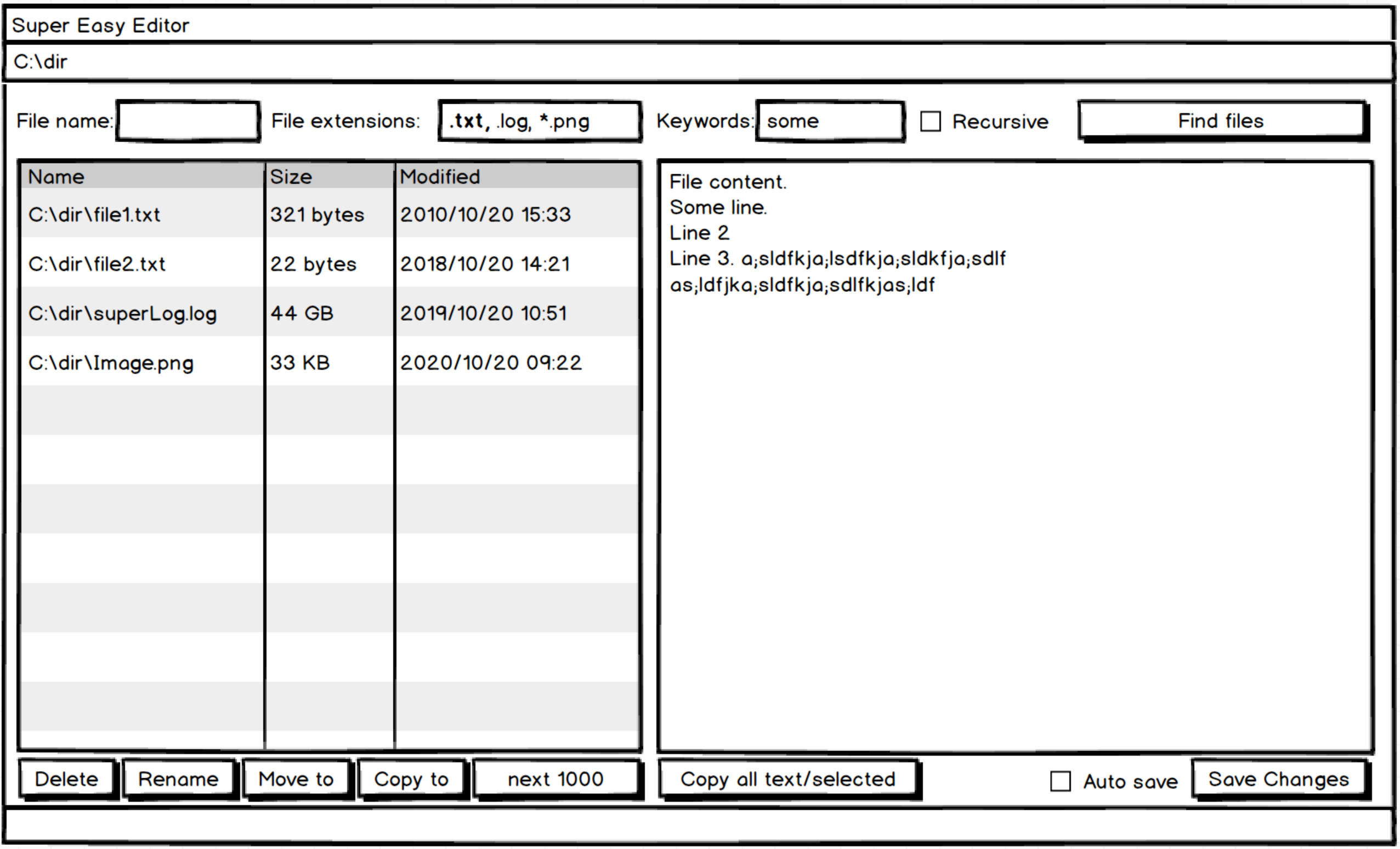
More functionality can be added, like “Save as”, “Refresh Directory”, “Search recursively”, …

## **Task 3\*. Advanced File viewer/editor/finder/explorer.**

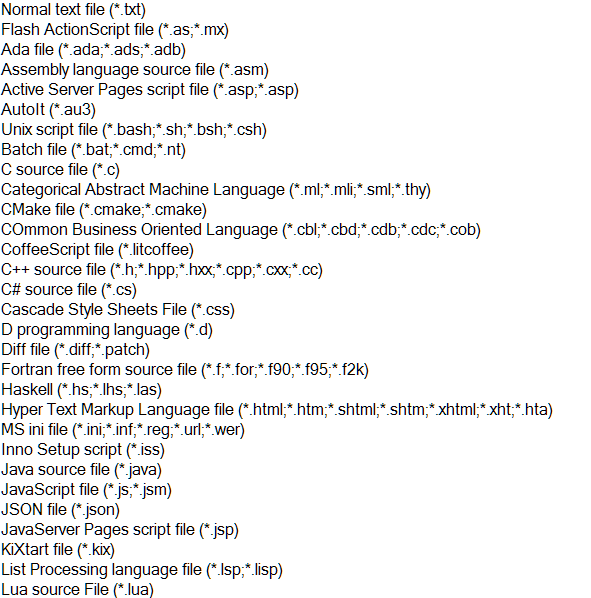
Implement previous ‘File viewer/editor’ task first. In addition to previous task, add these functionalities:

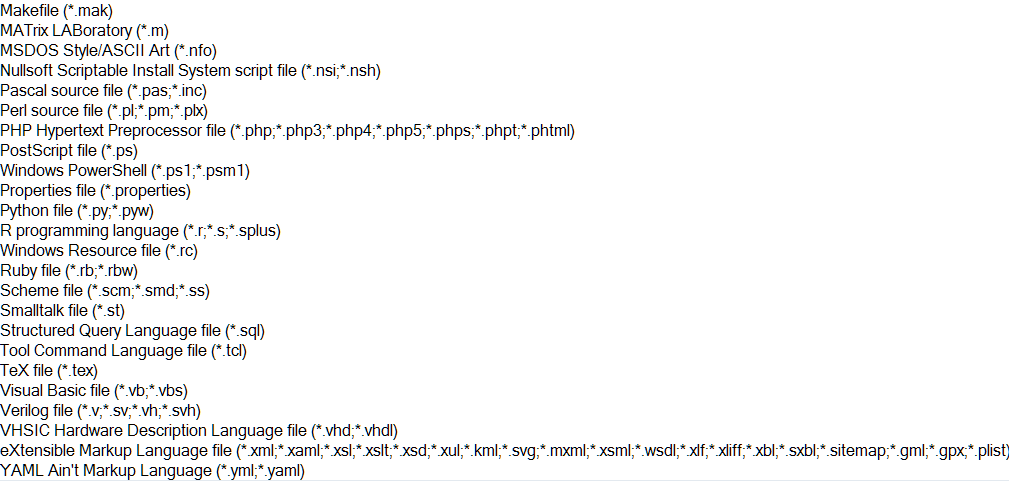
1. Checkbox with ‘recursive’ text, to allow users to search directories recursively.
2. Checkbox ‘auto save’ which will save all updated files automatically on closing the application (the app need to ‘remember’ all updated files) or when users click ‘Save Changes’. Users should see which files will be saved before app is closed. User can decide not to save any files on app close or select only some of them (you need to implement multi-select listbox UI control). The option ‘auto save’ is enabled by default. If uses do not close the app (and save the files) after 15 minutes, all edited files are saved automatically. ‘Save Changed’ button behaves differently depending of ‘auto save’ option; when ‘auto-save’ is disabled, the ‘Save Changes’ saves the current file; when ‘auto save’ option is enabled, ‘Save Changes’ saves all edited files.
3. Filter the files by filename, if there is a value in ‘File Name’ textbox.
4. Filter by file extension, if there is a value in ‘File Extensions’ textbox. (In previous task only \*.txt, \*.log extensions are searched; here the user can choose one or more file extensions, like: \*.txt, \*.log, \*.cs, \*.java, \*.css, \*.js, \*.xml.)
5. Filter files by ‘keywords’ found in text files. When file is displayed, the keywords are highlighted in yellow.
6. If there are more than 100 files users should see ‘Next 100 files’ button to load next 100 files according to chosen filter (by name or/and by extension).
7. If file is not text file, show ‘binary’ file. If file is image (\*.png, \*.jpg,…) the user see the image, and it is read-only.
8. Add buttons for more functionality for selected file(s), like “Delete”, “Rename”, “Move”.
9. In the list of files, users can use ‘Up’ and ‘Down’ keyboard keys, (arrows keys) to select the file. The file is loaded (text or image) when file is selected.
10. ‘File size’ and ‘Modified Date’ are shown in file list.
11. It can open/edit big txt files (GBs or TBs in file size).

Sample UI:



Please choose which sample text file extensions you are going to support:





## Task 4. Reverse file lines.

Create app which read all lines from file X and save them in another file X\_reversed, where lines are in reversed order.

Example:

File X:

Line1

Line 2

Line 3

File X\_Reversed:

Line 3

Line 2

Line 1

## Task 5. Super Encryptor.

Create app which encrypt files. Encrypt is in-place (overwrite the same file). Decrypt is in-place (overwrite the same file).

After Encrypt + Decript oprations, the file should be absolutely the same as it was before.

Algorithm:

Encrypt:

Read file in blocks/buffer of 512 bytes

1. Read 512 bytes
2. Reverse the bytes (reverse the array of bytes)
3. For each byte, add X to byte value, where X is its position. For example for byte X in position 5 use X=X+5.
4. Overwrite these 512 bytes
5. Proceed to next 512 bytes.

Decrypt:

Read file in blocks of 512 bytes

1. Read 512 bytes
2. Change each byte X using the formula X = X-BytePossition.
3. Reverse the bytes (reverse the array of bytes)
4. Overwrite these 512 bytes
5. Proceed to next 512 bytes.

Notes:

1. Byte values are between 0 and 255.
2. Last block can contains less than 512 bytes.

## **Task 6\*. Expressions**

We have 2 apps. Desktop and console.

The console app process files:

We have file with expressions on each line. You need to print the expression and the evaluation/result of this expression for each line in the console and also in output file. Supported operators and simbils: + - \* / and brackets ().

If the expression is not valid, throw new custom exception: InvalidExpressionException with property InvalidExpression of type string; Create another exception class which inherit InvalidExpressionInFileException with one more property “Line” which shows which line of the file is not valid expression.

Input file example/file content (7 lines):

1+1

2+3\*5

(2+3)\*(4+5)

10/3 + 3

3+10/3

1+2+3+4+5\*6+7\*8+9\*10

((1+2)\*(2+3))+33\*(1+1/10)

Console output + output file content (7 lines):

2

17

45

6.33333333

6.33333333

186

51.3

The expression evaluation should be done by class called ExpressionEvaluation, where it contains one method called Evaluate(string expression). ExpressionEvaluation is in another project/class library.

Test this class with unit tests. Write 10 unit tests with 10 different use cases.

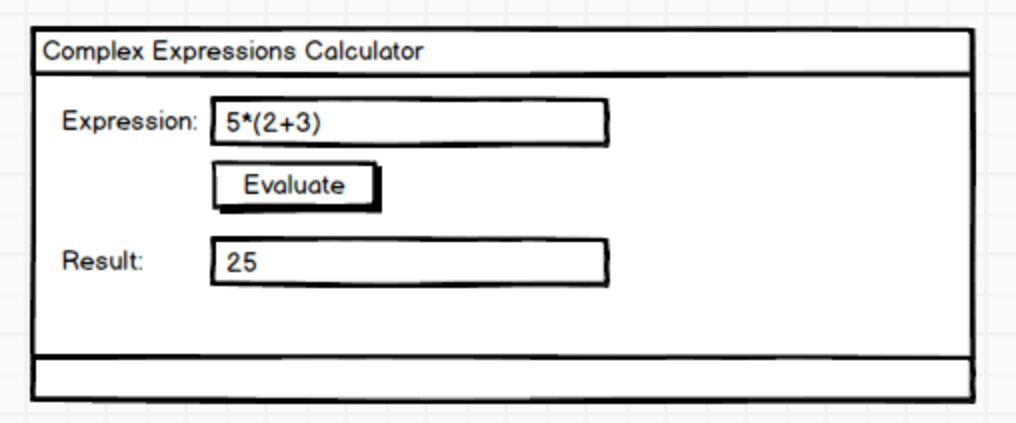
Desktop app:

Write simple desktop app which use the ExpressionEvaluation class. (ExpressionEvaluation should be in class library).

It has 2 textboxes – input and ouput/result textboxes and one button “Evaluate”.

It evaluate expressions if they are valid. If they are not valid, catch InvalidExpressionException and show the message in output/result textbox.

UI:



(Please use only C# and .net, without using external libraries, or external programs/process, like Powershell, Python and so on. If you are using such library or program, put ‘Using X’ in title, like “Complex Expressions Using Powershell”).

## Task 7. Simple Json, XML serilization

Create class Person with 3 properties: Age, Name, Address, where Address is a class with 3 properties: Country, City, Street.

Create 10 objects and set all properties. Serialize these 10 objects to 2 files in XML and JSON format. Later de-serialize the objects from the 2 files and print:

1. All 10 objects from XML file
2. All 10 objects from Json file.
3. Json file content.
4. XML file content.

## **Task 8\*. Json, XML, Binary serilization to network and file stream**

Write console and desktop app. Console app is sender “Person Sender”, desktop app “Person Receiver” is receiver of objects of type Person.

Create class Person with 3 properties: Age, Name, Address, where Address is a class with 3 properties: Country, City, Street. Move this class to Class Library, which is referenced by Console and Desktop app.

1. Console app:

Create Person objects and set all properties. Send these Person object to the desktop app using network streaming with Binary or Json format (you choose). When desktop app is not available/started, it write “Other party/side is not available”.

Write all these object to file stream for logging purposes (later to verify what was sent to other party/desktop app). For file use XML.

1. Desktop app: Listen for messages. Shows the objects in listbox or data grid, when they are received. Also when object is received, write it in file in JSON format.

Test with 1000 objects. If speed is not good, send them in batches (100 per batch).

Notes:

* Remember to call stream.Flush() where needed.
* For serializers/formatters, use one of these: XmlSerializer, JsonSerializer (Newtonsoft), DataContractJsonSerializer, BinaryFormatter
* For streams you can use: FileStream, NetworkStream, StreamWriter, StreamReader.

(Later we will change the communication using Web/WCF Services/Web Api).

## Task 10. Most common words.

Create small console/desktop application which will display most common words in text file or in list of text files.

The input of this application is 1 or more text files.

The output is in console or desktop application in this format:

Word1: 55 times

Word2: 45 times

Word3: 33 times.

The application shows first 1000 most common words in file(s).

The result is exported in XML or JSON format to a file called “MostCommonWorlds.txt”.

# Module 7. Accessing a Database

## Task 1. Code first. Console app.

Write EF code first app, which will generate DB for SQL Server for entities Student, Teacher, Course. Student and course are in many-to-many relationship. Teacher can teach many course, so Teacher to Course are in one-to-many relationship. Use properties of type: int, string, double, decimal, bool, DateTime.

Insert data in all tables.

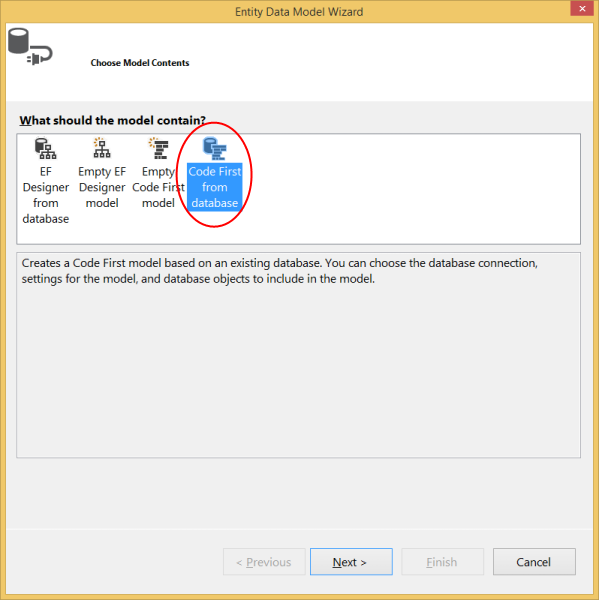
Select data from all tables using single LINQ query – Select all course with students and courses’ teachers. Print the result in console.

## **Task 2\*. LINQ to EF. Desktop app. Database first. Simple paging, sorting and filtering with LINQ to EF.**

1. Download DB AdventureWorks from the link below and attach/restore it in SQL Server:

[AdventureWorks sample databases - SQL Server | Microsoft Docs](https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms)

1. Create desktop app (win forms or WPF).
2. Create ‘Code First from DB’ Model, for all tables. All generated code should be in a folder in your project.



1. Use ‘Tabs’ control with 3 tabs and ‘Data Grid’/’Data Grid View’/Table in each tab to display the result of next LINQ queries:
   1. Select customer’s information: Customer Id, account number, person names, customer’s sales territory. Use tables Customer, Person, SalesTeritory in appropriate schemas. Display this information in tab “Customers Info”. Users should be able to filter customers by some of the persons’ name columns (first, middle or last name). Data is sorted by first name and then by last name. The data should be loaded using 1 single LINQ query.

Add ‘Save Data to JSON’ button in this tab and implement JSON serialization.

* 1. Select top 10 products by orders amount for each year (for last 20 years). For each product show ID, Name, Number, Size, ListPrice, Product Category, Product Sub Category, Product Model, Total order amount for year X, the year X. Shows data in “Top Products by Year” tab. The data is loaded with 1 single LINQ query.

Add ‘Save data to XML’ button in this tab and implement XML serialization.

* 1. For top X customers by total order amount, display customer info (names only), total order amount for each of their top Y products and these products information (Name, Number, Size, ListPrice, Product Category). Customers are shown in one ‘data grid’, their products are shown in another ‘data grid’ below the first one. Both of them shows max 10 rows. Users can choose/filter by **X** and **Y** values (using textboxes or numeric textboxes). Both grids are using paging. Use 2 LINQ queries – one for first ‘data grid’ and one for next ‘data grid’.

Add buttons ‘Load from DB’, ‘Save to XML’, ‘Save to JSON’, ‘Save to Binary’, ‘Load from JSON’, ‘Load from XML’, ‘Load from Binary’ to implement (de)serialization to/from different formats in local files.

In all pages, where result rows are more than 20, use paging for each ‘Data Grid’!

The data is loaded only when tab is first selected (next selections of tabs do not reload data).

# Module 8. Access Remote Data

## **Task 1\*. Web Api to return data to client apps/systems**

1. Download DB AdventureWorks from the link below and attach/restore it in SQL Server:

[AdventureWorks sample databases - SQL Server | Microsoft Docs](https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms)

1. Create Asp.Net Web Api project.
2. Create ‘Code First from DB’ Model, for all tables. All generated code should be in a folder in your project.

Create 1 controller with name SalesDataController with 5 operations (actions) in order for client application to be able to:

1. Select customer entities. Select next properties/columns from Customer entity/table:
   1. [CustomerID]
   2. [StoreID]
   3. [TerritoryID]
   4. [AccountNumber]
   5. [ModifiedDate]

Use 1 LINQ query to select the required data.

1. Select products from DB. Return all columns from DB which are not: primary key, foreign key, GUID/”unique identifier” type.
2. Select customer’s information: Customer Id, account number, person names, customer’s sales territory. Use tables Customer, Person, SalesTeritory in appropriate schemas. Display this information in tab “Customers Info”. Users should be able to filter customers by some of the persons’ name columns (first, middle or last name). Data is sorted by first name and then by last name. The data should be loaded using 1 single LINQ query.
3. Select top 10 products by orders amount for each year (for last 20 years). For each product show ID, Name, Number, Size, ListPrice, Product Category, Product Sub Category, Product Model, Total order amount for year X, the year X. Shows data in “Top Products by Year” tab. The data is loaded with 1 single LINQ query.
4. For top X customers by total order amount, display customer info (names only), total order amount for each of their top Y products and these products information (Name, Number, Size, ListPrice, Product Category). Clients can choose/filter by **X** and **Y** values (using parameters).

Important: For all operations use paging using 2 params: “PageNumber”, “PageSize” of type integer. PageSize cannot be more than 1000 (configurable in application configuration file; use app setting with name MaxPageSize).

Optional:

Create Desktop and Web applications (Asp.Net MVC) to consume the data from the Web API controller. (Do not reference the Web Api application directly and do not invoke the methods/actions of controller ‘directly’/in memory).

Use 5 tabs in Desktop app. Data is loaded when tab is selected (if data had not been loaded before).

Use 5 links in Web app. When link is clicked appropriate data is loaded from Web Api controller. Add checkbox ‘client-side loading’. When checkbox is checked, the Web API services are invoked from javascript. If checkbox is not selected, the data is loaded in server-side (asp.net) using C# and asp.net controller with 5 actions which return data to browser as html (using razor views).

# Module 9. Graphical UI apps.

Tasks from other modules:

[Module 6, task 3. Advanced File Viewer.](#_Task_3*._Advanced)

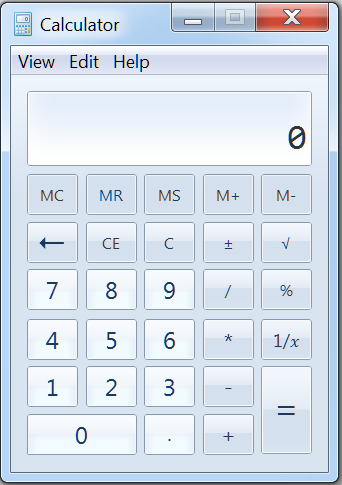
[Module 6, Task 7. Calculator.](#_Task_7._Create)

[Module 8, task 1 – create desktop app to consume the Web API.](#_Task_1*._Web)

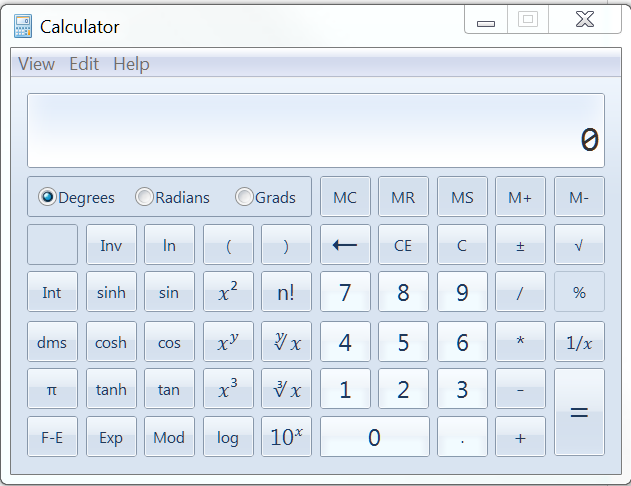
## **Task 1\*\*. Advanced Calculator.**

Implement similar to windows 7’s calculator using WPF. It has 2 modes:

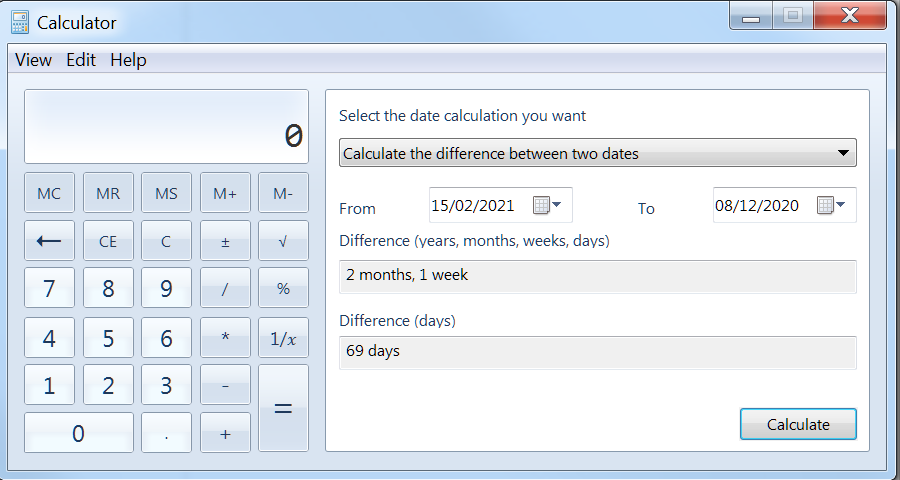
1. Standard, Simple:

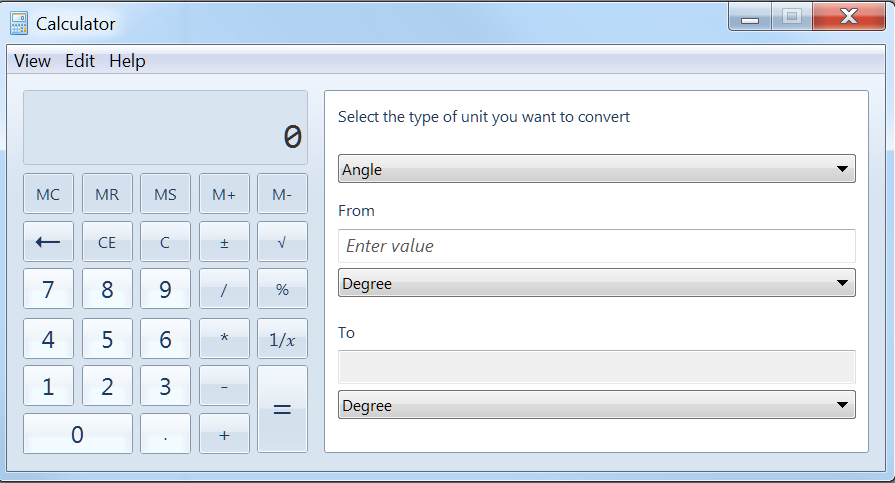


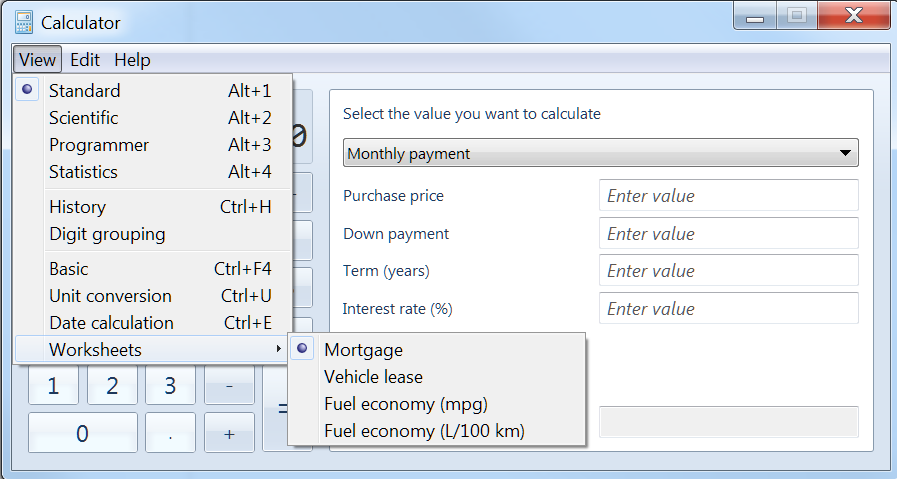
1. Scientific



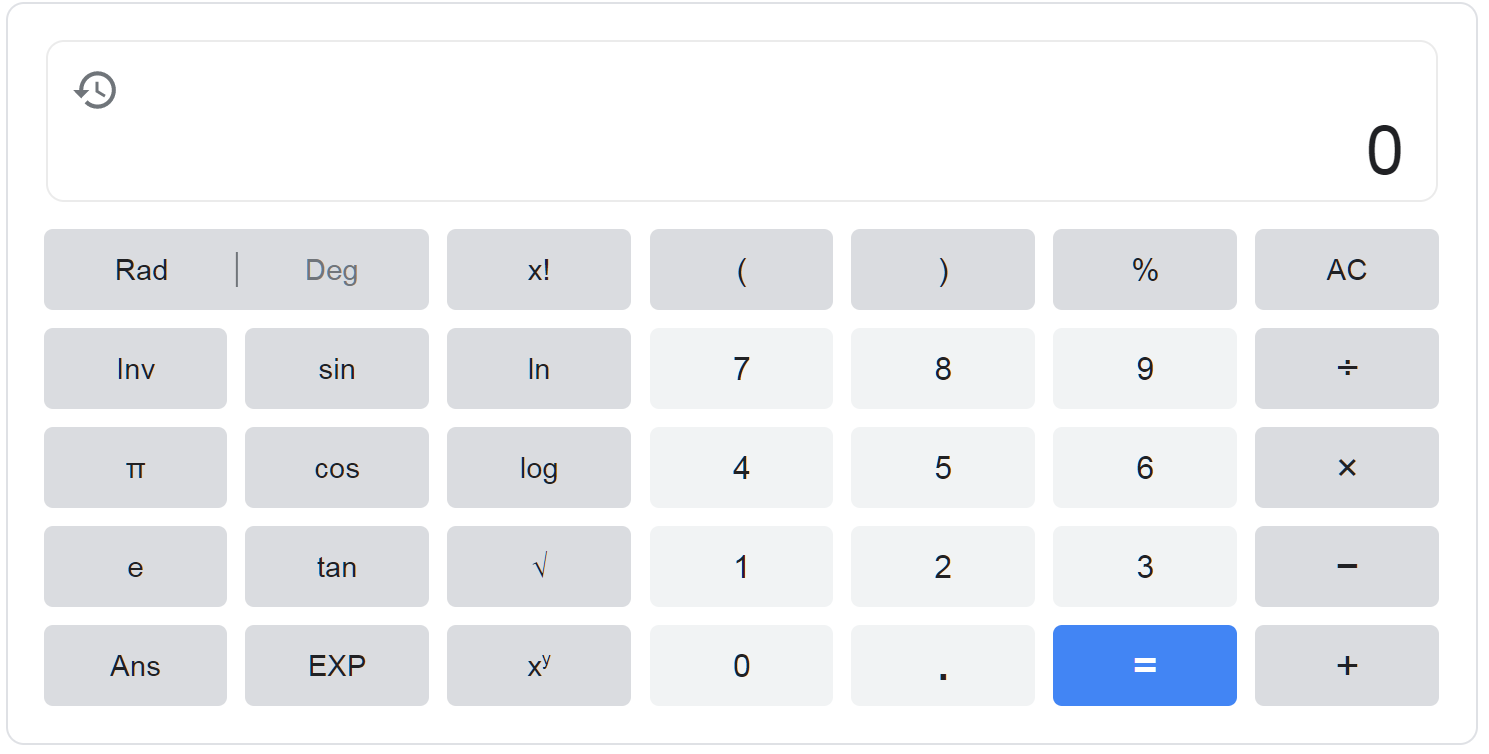
Also, it can calculate: Dates, Units and Mortgages:



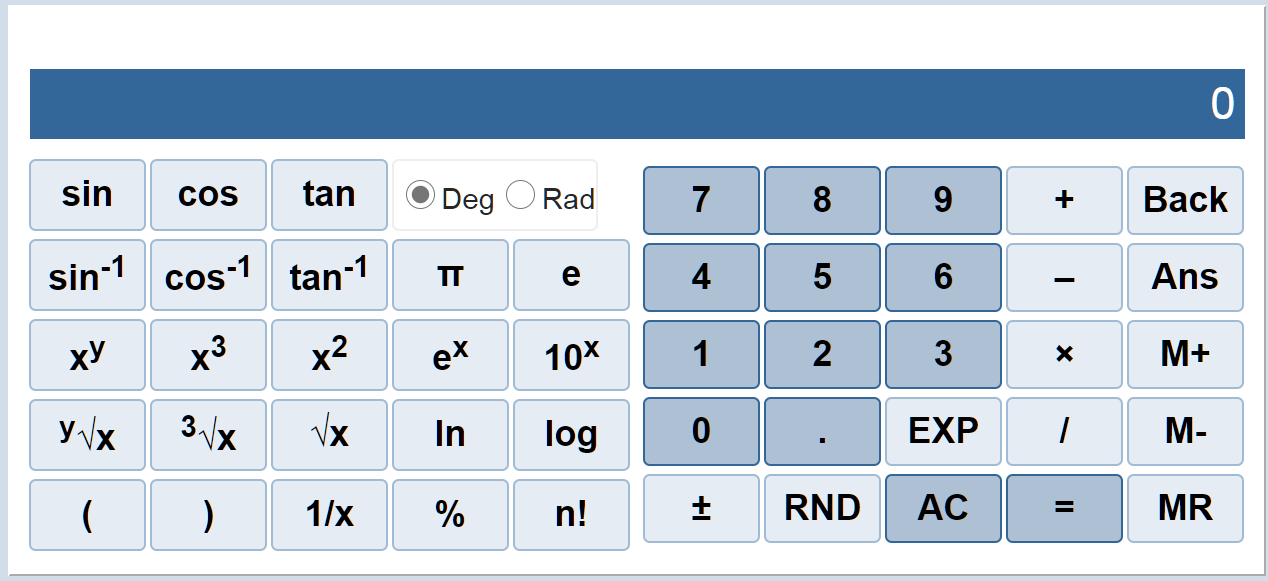




The UI is up to you. Can be something like:

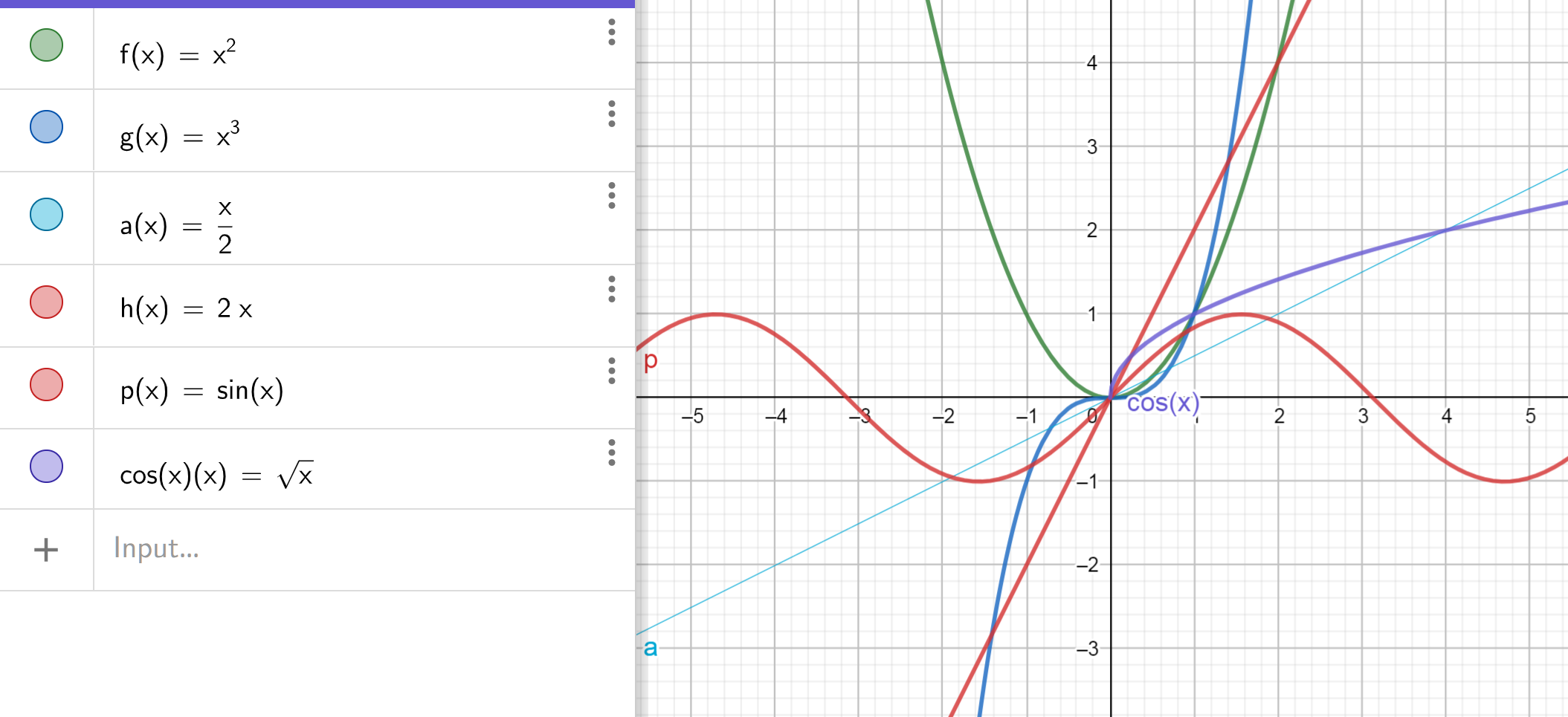


Or like:

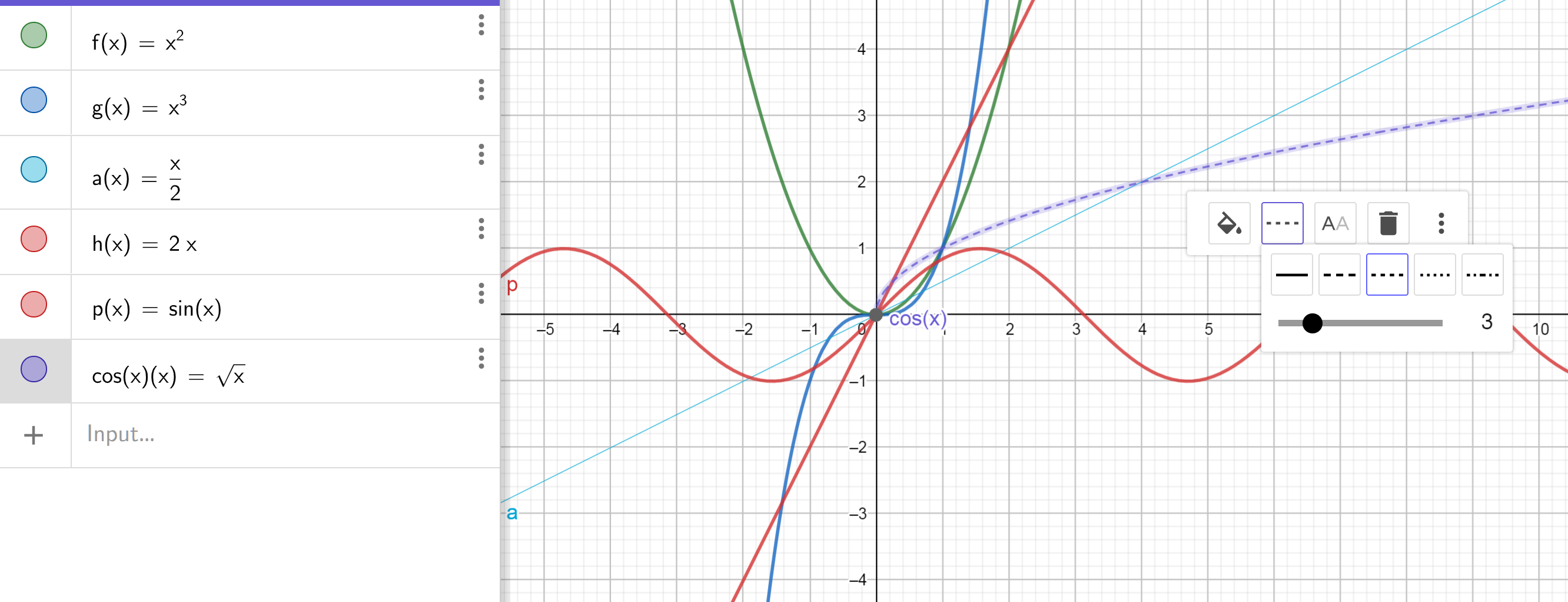


## **Task 2\*\*\*. Graphics Calculator**

Implement Graphics Calculator. Example:



It should support displaying/drawing multiple functions, for each function the user can choose the color and the type of line:



## **Task 3\*. Data-oriented Desktop App. CRUD operations. Paging, Sorting and Filtering.**

In this application you can choose to use files or database.

Create desktop app (WPF) to select/delete/update/insert entities in Adventure Works DB.

1. Download DB AdventureWorks from the link below and attach/restore it in SQL Server:

[AdventureWorks sample databases - SQL Server | Microsoft Docs](https://docs.microsoft.com/en-us/sql/samples/adventureworks-install-configure?view=sql-server-ver15&tabs=ssms)

1. Create WPF project.
2. Create ‘Code First from DB’ Model, for all tables. All generated code should be in a folder in your project.

Create 5 tabs for select/delete/update/insert for following entities (and their related information – up to 3 levels; for example Product and related Sub Category and Category, Product and related Model, Employee and related Person which is Related to Address (using many-to-many), Customer and related Person, Address, Phone tables. For each tab, show information from main table and also related tables in the same ‘Data Grid’. If relationship is many-to-many, show only the last (by modified date or ID) or related items; for example Person+last address+last phone in one ‘Data Grid’.

Tabs:

1. Tab ‘Products’ for CRUD (create/read/update/delete) operations. Note that Product has Category, Sub Category, and Model which are in different tables. For related entities/tables use drop-down controls.
2. ‘Customers’ tab for CRUD operations for Customers. Note that customer is related to Person table, Address, Phone.
3. ‘Orders’ tab. (SalesOrderDetail and SalesOrderHeader tables and related table Product) and later Customers table.
4. ‘Employees and Department’ tab for table Employee and related Department. Note that Employee is related to Person, Address, Phones table.
5. ‘Vendors’ tab – table ProductVendor and appropriate related tables.

Use ‘Data Grids’ to display entities. Use paging with size 20 per page. Filtering is supported only by ‘name’ columns (name column = category name, product name, customer names, employee name, vendor name)! Sorting is supported only for ‘name’ columns.

No ID (identity, primary key and foreign key) columns are shown in UI. Also, no GUID columns are displayed. Varbinary columns are not displayed also.

Optional (requires knowledge covered in Module 10):

All operations should use async/await pattern (or other asynchronous approach) to access the DB and display the data without ‘freezing’ the UI!

# Module 10. Multitasking and Multithreading.

## Task 1. Using Task<T>

Create a console application which is running X number of tasks, where X is the count of logical processor of the current machine. (Example: 4 cores processor + hyper threading = 8 logical processors). All of them are executing 1 000 0000 000 operations which increment single shared variable **counter** of type **long**. At the end print the result and check if it is equal to X\*1 000 0000 000.

Put the access to **counter** in **lock** statement. Execute again and check the results.

## Task 2. Controlling Task Execution

Create console application to use these Task methods:

* To start a task:
  + **Task.Start** method
  + **Task.Factory.StartNew** method
  + **Task.Run** method
* To wait for tasks to complete:
  + **Task.Wait** method
  + **Task.WaitAll** method
  + **Task.WaitAny** method

Questions:

1. Which are instance methods and which are static methods?
2. How many overloads they have?
3. Compare: **Task.Factory.StartNew** vs **Task.Run**

## Task 3. Use Parallel.Invoke

Write 3 methods to implement:

1. Use **Parallel.Invoke** to run multiple tasks simultaneously. Wait for all of them to complete. Task should execute the operations in Task 1 in this module.
2. Use **Parallel.For** to run **for** loop iterations in parallel to iterate 1000000 times to calculate x\*x. Compare the execution time if it is normal for loop statement.
3. Use **Parallel.ForEach** to run **foreach** loop iterations in parallel to iterate over a List<int> collection. Compare the execution time if it is normal foreach statement.

## Task 4. Creating Awaitable Methods to download an image.

Create one **async** method to download image file from specific URL. Invoke it with **await**. Invoke it in Desktop app and show the image.

## Task 5\*. Synchronizing Concurrent Access to Data

Create 6 console apps examples for “Using Synchronization Primitives with the Task Parallel Library”. Use these sync primitives:

1. Use lock
2. Use the **ManualResetEventSlim** class to limit resource access to one thread at a time
3. Use the **SemaphoreSlim** class to limit resource access to a fixed number of threads
4. Use the **CountdownEvent** class to block a thread until a fixed number of tasks signal completion
5. Use the **ReaderWriterLockSlim** class to allow multiple threads to read a resource or a single thread to write to a resource at any one time
6. Use the **Barrier** class to block multiple threads until they all satisfy a condition

# Module 11. IDisposable

## Task 1. Word count

Create a class which will count how many words you have in \*.txt files in a directory. Make the class IDisposable. Use it with **using** construct. Internally the class should use StreamReader with **using** construct.

[IDisposable Interface (System) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.idisposable?view=net-5.0)

# Module 12. Reusable Types and Assemblies - Reflection

## Task 1. Reflection investigation

Create a console application which display all classes and all members of these classes in list of specified assemblies.

* For each methods and constructors it shows all parameter types and also the return type.
* For each field and property it shows the name and the type
* For other members it shows the name, declared type and member type.

## **Task 2\*. Advanced method invoker/reflector**

Create desktop app which show all .net assemblies in the local machine in a list (listbox or ‘Data Grid’).

For selected assembly it shows all classes.

For selected class it shows all methods.

For selected method is shows list of parameters.

The user should be able to enter parameter values and click ‘Invoke’ button. The app will invoke the selected method with parameters’ values from the user. If the method has return type, the user can see the result. If the return type is primitive the result is shown, otherwise result public properties are shown (if they are primitive type) and so on, up to 5 levels.

# Module 13. Encrypting and Decrypting Data (this module will not be covered in some courses).

## Task 1. Symmetric Encryption. Hashing.

Create console/desktop app – File Encryptor. Use these classes for symmetric encryption:

* **AesManaged** class or
* **RijndaelManaged** class

To encrypt and decrypt data symmetrically, perform the following steps:

1. Create an **Rfc2898DeriveBytes** object
2. Create an **AesManaged** object
3. Generate a secret key and an IV
4. Create a stream to buffer the transformed data
5. Create a symmetric encryptor or decryptor object
6. Create a **CryptoStream** object
7. Write the transformed data to the buffer stream
8. Close the streams

Later sign the file. Save the hash in encrypted file (at the end). Use HMACSHA256 Class to implement the hashing.

The secret key for encryption and hashing is provided by the user.

[AesManaged Class (System.Security.Cryptography) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.aesmanaged?view=net-5.0)

[RijndaelManaged Class (System.Security.Cryptography) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.rijndaelmanaged?view=net-5.0)

[HMACSHA256 Class (System.Security.Cryptography) | Microsoft Docs](https://docs.microsoft.com/en-us/dotnet/api/system.security.cryptography.hmacsha256?view=net-5.0)

# Module 14. SQL

## Task1. Querying.

Write the following ‘select’ queries against Northwind or AdventureWorks sample databases:

1. Select all customers; products; orders;
2. Select all customers from specific country
3. Select all customers from specific country sorted by name
4. Select all country for customers
5. Select only customers placed/made orders
6. Select customers without orders
7. Select customers and their orders
8. Select top 10 customers by total orders amount
9. Select top 10 products ordered by customers
10. Select top 10 categories or products (by total orders amount)
11. Select products and all related tables and their related tables and so on (up to 6 levels). Sort the result by product, category name, sub category name. Use appropriate **inner** or **left outer joins**.
12. From Northwind select app products and all related tables in DB and their related tables and so on, until you select data from all tables. Do the same starting from customers. Use appropriate **inner** or **left outer joins**.
13. Select top 5 employee processed/done most orders for specific product.
14. Select top 5 employee processed/done most orders for each year for specific product.
15. Customers and employees from the same country/city.
16. Customers with or without orders.
17. Customers and their last 3 orders.
18. Select top 10 customers by total orders amount (total amount for all their orders) + their last 3 orders (only last 3 orders by order’s date)
19. Total sales amounts (orders amount) per year, country, city
20. Total sales amounts grouped by year and month
21. Total sales for last 12 months and % increase compared with the same month previous year.
22. Total sales for last 12 months and % increase compared with the same month previous year, for each country.