Sofia University
Department of Mathematics and Informatics

Course : OO Programming C#.NET
Date:
Student Name:

Lab No. 8

Submit the all C# .NET files developed to solve the problems listed below. Use comments and Modified-Hungarian notation.

Problem No. 1

Use **Explicit Interface Member Name Qualification** to implement interfaces in the following problems:

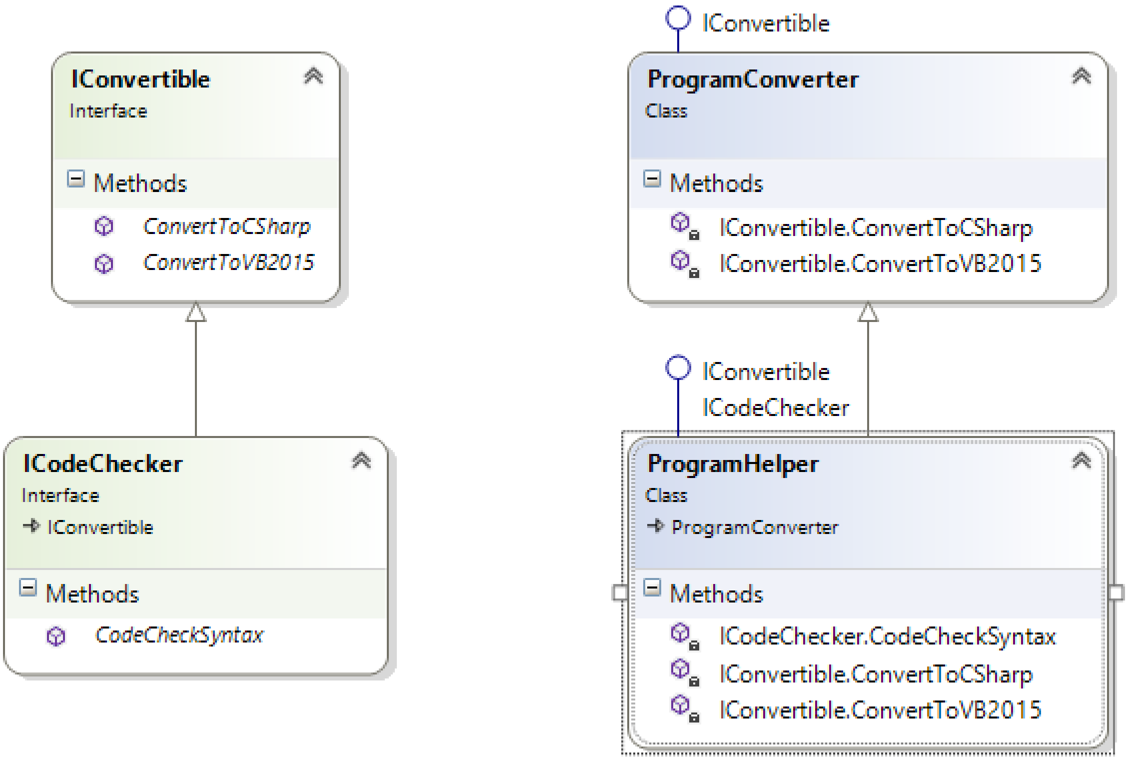
A) Define an **`interface IConvertible`** that indicates that the **`class`** can convert a **`string`** to **`C#`** or **`VB2015`**. The interface should have two methods: **`ConvertToCSharp`** and **`ConvertToVB2015`**. Each method should take a **`string`**, and return a **`string`**.

B) Implement that **`interface`** and test it by creating a **`class ProgramHelper`** that implements **`IConvertible`**. You can use simple **`string`** messages to simulate the conversion.

C) Extend the **`interface IConvertible`** by creating a new interface, **ICodeChecker**. The new interface should implement one new method, **`CodeCheckSyntax`**, which takes two strings: the **`string`** to check, and the language to use. The method should return a **`bool`**. Revise the **`ProgramHelper`** class from **`Problem B`** to use the new **`interface`**.

D) Demonstrate the use of **`is`** and **`as.`** Create a **`new class`**, **`ProgramConverter`**, that **`implements IConvertible`**. **`ProgramConverter`** should **`implement`** the **`ConvertToC-Sharp`**( ) and **`ConvertToVB`**( ) methods.

E) Revise **`ProgramHelper`** so that it derives from **`ProgramConverter`**, and implements **`ICodeChecker`**.



Problem No. 2

Create a **`struct Point`** which has coordinates ***`(double x, double y, double z)`***.

Create a **`struct Vector`** which has a starting **`Point`** and an end **`Point`**.

Create a **`struct Triangle`** which has sides **`Vector`** ***`a`*** and **`Vector`** ***`b.`***

Add default implementations of methods **`Equals()`** and **`GetHashCode()`** in these structs

Define **an `interface`** ***`Comparable`*** and implement it with explicit name qualification in **`structs`** **`Point, Vector`** and **`Triangle`**.

Include in **`interface`** ***`Comparable`*** the following:

- -method ***`double SizeOf`***();

  // the *`SizeOf()`* a *`Point`* is the absolute value of the total of its coordinates

  // the *`SizeOf()`* a *`Vector`* is the *length* of the *`Vector`*

  // the *`SizeOf()`* a *`Triangle`* is the absolute value of its area

- **an `indexer` get and set `property using a string to access the datamemebers of` *`Point, Vector`* and *`Triangle`***

Provide **general purpose constructor** for the above **`structs`** and override the inherited ***`ToString()`*** method that displays the data members and the **`SizeOf`**() the respective object (*properly formatted with 2 digits after the decimal point*). Override method ***`Equals()`*** inherited from class ***`object.`***

Define a public **`delegate`**

bool ***GreaterThan(Comparable obj1, Comparable obj2)*** // obj1 is greater than obj2

to compare **Comparable** objects in terms of ***SizeOf();***

For each of the **structs** ***Point, Vector*** and ***Triangle*** define a ***private*** static method **GetSizeOf*(Comparable obj1, Comparable obj2)*** to implement the delegate ***GreaterThan*** for the respective ***struct***. Return ***true*** when ***obj1.SizeOf()*** is greater than ***obj2.SizeOf()*** and ***false*** otherwise.

Define a static *get* **property** returning the instance of ***GreaterThan*** for ***GetSizeOf().***

For structs ***Vector*** and ***Triangle*** **overload the operators** :

a) operator +

For struct ***Vector***- add the coordinates of the two vectors in addition; For struct ***Triangle***- add the areas of the ***Triangles*** in addition

b) operator *

For **struct *Vector***- the **vector** product of two vectors in multiplication; as **well as**, **the product of a *Vector* by an Integer** number. For ***Triangle***- a product of a ***Triangle*** and an **integer number** (***zoom factor***)- each of the **Vector** sides of the ***Triangle*** are multiplied by the *zoom factor*

Define a ***BubbleSort(Comparable[],GreaterThan g)*** **method** to sort an array of ***Comparable*** objects, where the **delegate *GreaterThan*** determines the ordering sequence (Assume the elements of ***Comparable[]*** are all Points, Vectors or Triangles only)

Write a Windows application that defines ***Points***, ***Vectors***, and ***Triangles*** and ***sorts*** them by clicking respective buttons, ***adds Vector*** objects, ***adds Triangle*** objects and ***zooms Triangle*** objects by a user defined **factor**.

Problem No. 3

Create a ***class InvoiceDetails*** (it has a ***double lineTotal*** member with a ***get property, constructors***).

Create a ***class Invoice.*** Every ***Invoice*** has a (**unique**) sequential long number (***invoiceNumber*** member with a ***get*** property, constructors) and an ***ArrrayList*** (named ***detailLines***) of ***InvoiceDetails*** objects. It also has a method ***PrintInvoice*** () (prints out on the Console the ***invoiceNumber*** and the ***LineTotals*** of the ***InvoiceDetails*** objects in ***detailLines***). Overload the ***operator+*** for ***class Invoice***, allowing you to add the ***LineTotals*** of the ***InvoiceDetails*** objects comprising two Invoice objects given as arguments for the operator into the ***detailLines*** of a new ***Invoice*** object that has to be returned.

Overload the ***operator>*** **and** ***operator<*** for class Invoice, allowing you to compare two Invoice objects provided as arguments (by comparing the ***total*** amount of the ***lineTotals*** of their ***detailLines***)

Overload the ***operator**** so that it takes as a second argument a ***double*** number (***discount***). As a result return a ***new Invoice*** object having the ***lineTotal*** of all the ***InvoiceDetails*** objects of the first argument of the ***operator**** multiplied by ***discount*** *(*a discounted ***Invoice*** object*)*

Write a Console application to test the above classes.- create two Invoices with different sets of InvoiceDetails and apply the overloaded operators to them, run the ***PrintInvoice*** () method.

__Hint__: **Create** an instance of ***ArrayList*** as follows:

```
private  ArrayList  detailLines;
..>>,,,,>>
detailLines= new ArrayList();
```

Add elements to an `ArrayList` as follows:

```
detailLines.Add(new InvoiceDetailLine(intInvoiceDetailTotal);
```

__**Problem No. 4**__

A ***RationalNumber*** is any number that could be represented as the division of two integer numbers- a numerator and a denominator. Thus, any ***RationalNumber*** has a numerator and a denominator. For instance, the numbers -5, $\frac{3}{4}$, $-\frac{1}{2}$ etc are rational numbers (*the numerator and denominator of* ***-5*** *are respectively, the integer numbers* ***-5*** *and* ***1***). Write a ***RationalNumber*** class in ***C#.NET*** with the following capabilities:

a) Create a general purpose constructor that **prevents a 0 (zero) denominator**, **reduces or simplifies** fractions that are not in reduced form (for instance, $2/4$ and $1/2$ represent the same ***RationalNumber***) and **avoids negative denominators**. (for instance, $2/(-4)$ and $-1/2$ represent the same ***RationalNumber***)

b) Create a **default constructor** (the default rational number is ***1/1***) and **a copy constructor**

c) Define ***set/get*** properties for the ***nominator*** and ***denominator*** **- prevents a 0 (zero) denominator**, **reduces or simplifies** fractions that are not in reduced form (for instance, $2/4$ and $1/2$ represent the same ***RationalNumber***) and **avoids negative denominators**. (for instance, $2/(-4)$ and $-1/2$ represent the same ***RationalNumber***)

d) Create an ***int*** to ***RationalNumber*** constructor (the result should be a rational number with the given ***int*** as ***nominator*** and ***denominator*** equal to ***1***)

e) Overload the **addition (+)**, **subtraction(-)**, **multiplication(*)** and **division(/)** operators for this class, as well as, (the corresponding ***+=, -=, /=, *=*** operators will be evaluated on the basis of **addition (+)**, **subtraction(-)**, **multiplication(*)** and **division(/)**).

f) Catch ***DivideByZeroException*** with the operator **/**

g) Overload the **relational** (***<,>***) and **equality** ***(==, !=)*** operators. (override the virtual ***Equals(), GetHashCode()*** methods, as well)

h) Overload the virtual ***ToString()*** method (display the ***numerator*** and the ***denominator*** separated by a slash)

i) Overload the __**explicit**__ type conversion operator ***(int)*** from ***Rationalnumber*** objects to ***int***. (the result should be an ***int*** number equal to the integer division of the ***numerator*** over the ***denominator***), as well as, **implicit** type conversion from ***int*** to ***RationalNumber*** ***(***thus, it must be possible to add a ***RationalNumber*** to an ***int***, divide ***RationalNumber*** by an ***int*** etc, by means of the operators defined in (e)***)***

j) Write a **C#.NET** Windows application, which tests **completely** each one of the capabilities *a) – i)* (use textboxes and labels to manage the input and output, use buttons to manage the overloaded operators.

Problem No. 5a

Modify the **payroll system** of Employees (*see the sample code Fig12.rar*) to include private instance variable ***birthDate*** in ***class Employee***. Use ***class Date*** (*see the sample code Fig12.rar)* to represent an employee's birthday. **Assume that payroll is processed once per month. Create an array** of ***Employee*** variables to store references to the various employee objects. In a loop, calculate the payroll for each ***Employee*** (polymorphically), and add a **$100.00** bonus to the person's payroll amount if the current month is the month in which the ***Employee's*** birthday occurs.

Problem No. 5b

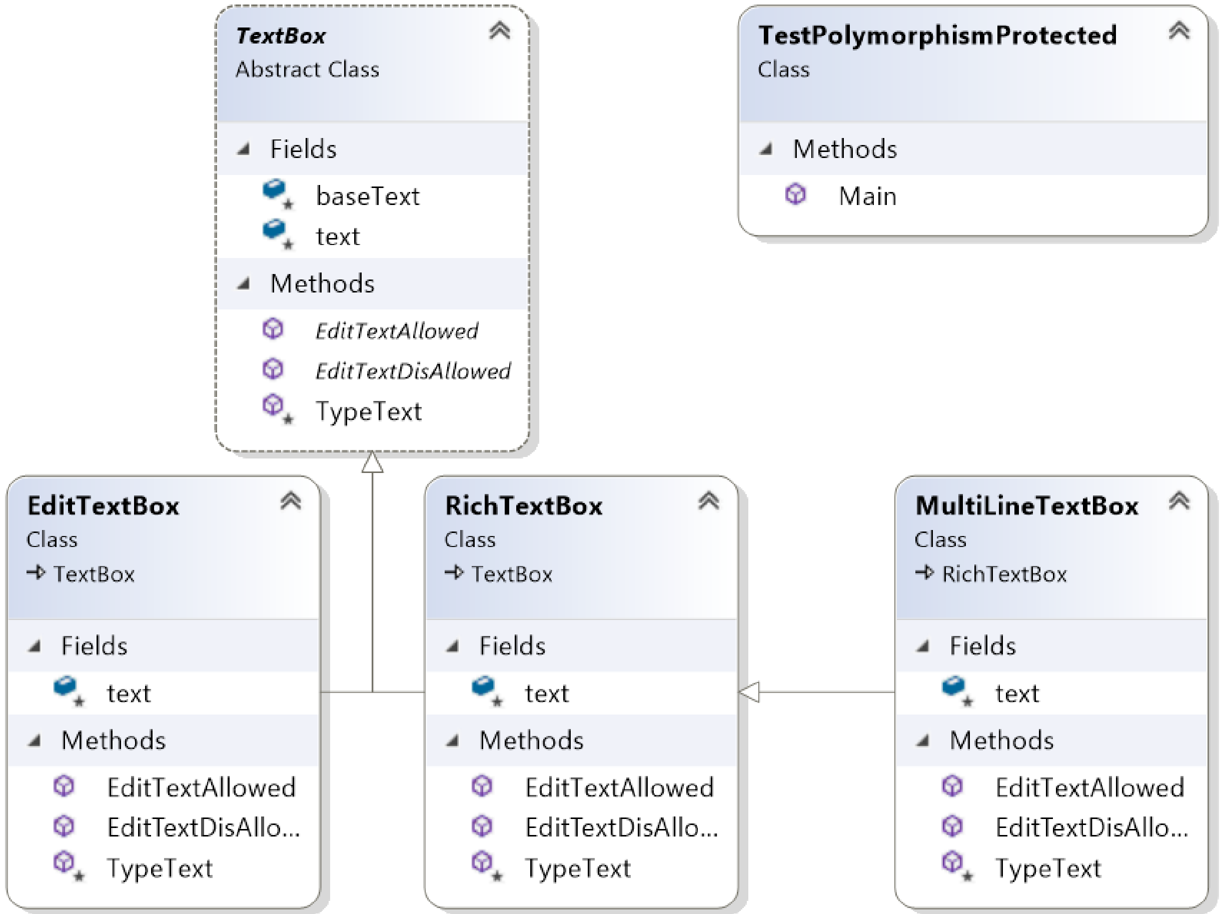
Modify the **payroll system** Employee- SalariedEmployee in **Figs. 12.4–12.11** (*see the sample code Fig12.rar*) to include private instance variable ***birthDate*** in ***class Employee***. Use ***class Date*** (*see the sample code Fig12.rar)* to represent an employee's birthday. **Assume that payroll is processed once per month. Create an array** of ***Employee*** variables to store references to the various employee objects. In a loop, calculate the payroll for each ***Employee*** (polymorphically), and add a **$100.00** bonus to the person's payroll amount if the current month is the month in which the ***Employee's*** birthday occurs.

Problem No. 6

According to [C# reference documentation] "*A protected member is accessible within its class and by derived class instances*".

Now let's investigate how this definition affects polymorphism involving protected methods of classes in inheritance relationships.

Create the following inheritance hierarchy:



Datamember **`text`** is of type **`string`** and it is **`protected`** in classes **`TextBox`**, **`EditTextBox`**, **`RichTextBox`** and **`MultlineTextBox`**. Use the keyword **`new`** to hide the inherited data member in the classes derived from the **`abstract`** class **`TextBox`** . Initialize **`text`** to the **`string`** $"**`{(GetType())}:Type text`**".

Data member **`baseText`** is of type **`string`** and it is **`protected`** in class **`TextBox`**. Initialize **`baseText`** to the **`string`** $"**`{(GetType())}:Type baseText`**".

Add **`protected void`** method **`TypeText`**() to **`class TextBox`** and **`override`** it in the derived classes. Each of the methods overriding **`TypeText`**() prints on standard output the current value of data member **`text`**.

Add **`public abstract void`** methods **`EditTextAllowed`**() and **`EditTextDisAllowed`** () to **`class TextBox`** and **`override`** them in the derived classes

Each of the overridden versions of method **`EditTextAllowed`**()

- Executes the version of in method **`TypeText`**() in the direct base class
- Prints on standard output the current value of datamember **`text`** in the direct base class. (**`baseText`** is protected and it is OK)
- Prints on standard output the current value of datamember **`baseText`** in the direct base class. (**`baseText`** is protected and it is OK)

Each of the overridden versions of method **`EditTextDisAllowed`** ()

- Upcasts an instance of the current class to the base **`class TextBox`**
- Attempt to execute method **`TypeText`**() via the upcasted instance results in compiler error. Polymorphism is impossible in this case although **`TypeText`**() is overridden in the derived class, **explain why**!
- Attempt to assign a new value to data member **`text`** via the upcasted instance results in compiler error. **`text`** is **`protected`** and it is still disallowed, **explain why**!
- Attempt to assign a new value to data member **`baseText`** via the upcasted instance results in compiler error**.** **`baseText`** is **`protected`** and it is still disallowed, **explain why**!

Add class **`TestPolymorphismProtected.`** Add an array of instances of classes **`EditTextBox`**, **`RichTextBox`** and **`MultlineTextBox`** in the **`public static void Main()`** method of this class. Write a loop **to execute polymorphically method `TypeText`**() of the array elements. Notice, you get compiler error, **explain why**.

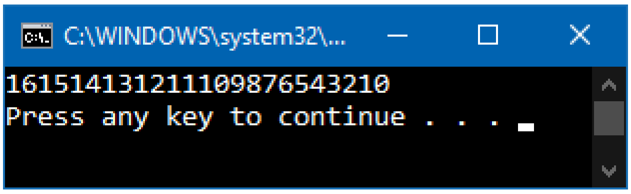
Problem No. 7a

Create **`interface IEnumerator`** with methods

```
bool MoveNext();
object Current { get; }
void Reset();
```

Write an implementation of **`IEnumerator`** in class **`CountDown`** allowing to use the **`interface`** methods in a while loop for the purpose of printing the sequence of numbers from 16 to 0. This class represents a default implementation of the interface methods(*without explicit qualification of the interface name in the method implementation*).

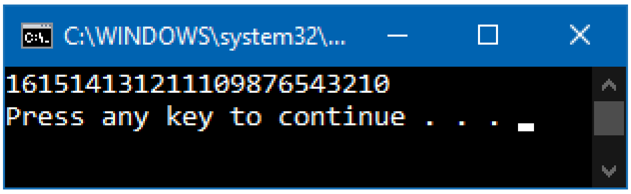
Test the implementation of the **`interface`** methods



Problem No. 7b (C# 8)

Provide a default implementation of **`IEnumerator`** methods by embedding the implementation of **`IEnumerator`** in class **`CountDown`** from Problem 7a inside **`interface IEnumerator.`**

Test the default implementation of the **`interface`** methods

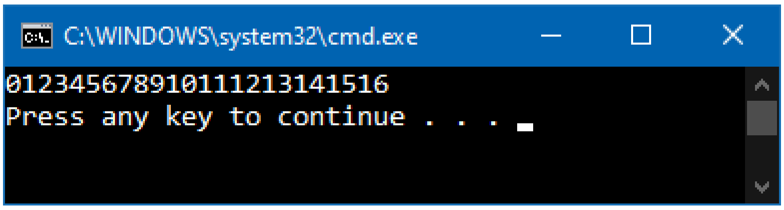


Problem No. 7c (C# 8)

Provide the same implementation for each of the **IEnumerator** methods in class **CountDown** as in 7b but make these methods **virtual**. Inherit **IEnumerator.CountDown** in class **CountDownWithOverride.**

Override the default methods implementation in **IEnumerator** done in class **CountDown** with versions allowing to printing the sequence of numbers from 0 to 16

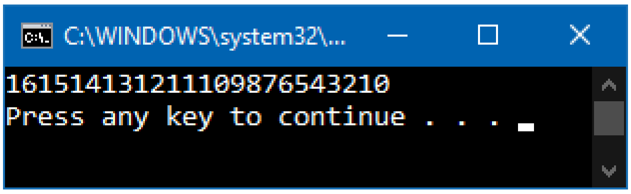
Test the default implementation of the interface methods in class **CountDownWithDefaults**



Problem No. 7d (C# 8)

Repeat tasks in Problem 7b where the implementation of **IEnumerator** methods in the embedded class **CountDown** in inside **interface IEnumerator** is done with explicit qualification of the interface name.

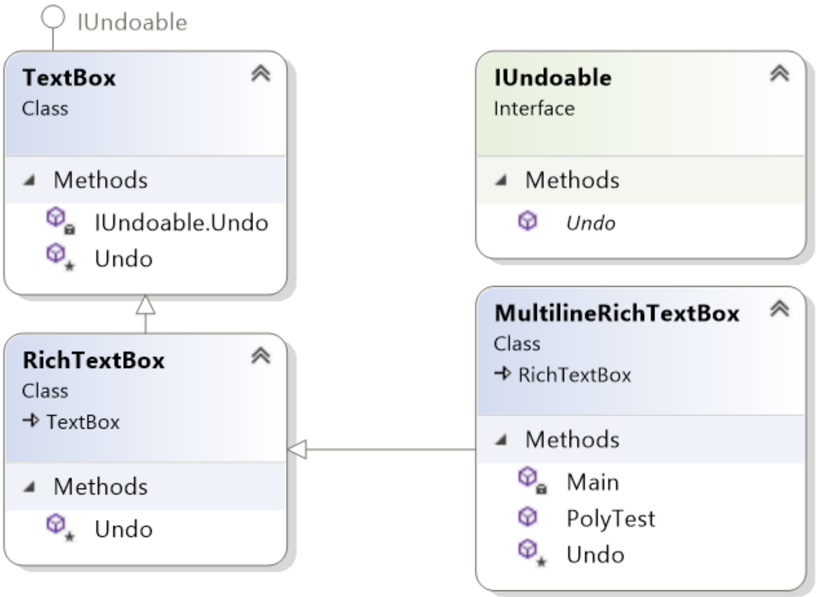
Test the implementation of the **interface** methods



Problem No. 8

Even with explicit member implementation, interface reimplementation is problematic for a couple of reasons:

- The subclass has no way to call the base class method.
- The base class author may not anticipate that a method will be reimplemented and may not allow for the potential consequences.



A better option, however, is to design a base class such that reimplementation will never be required.

Create the artefacts in the above UML class diagram, where:

- Method Undo() in IUndoable is void and takes no arguments

- method Undo() is implemented in class TextBox as protected and overriden in classes, RichTextBox and MultilineRichTextBox. Each one of the implementations of this method prints on the console text $"{(GetType())}.Undo"

- interface IUndoable is implemented with explicit name qualification in class TextBox by calling the protected method Undo().

Write method void PolyTest() in class MultilineRichTextBox to test the polymorphic behavior of the implementation of method Undo() with explicit interface name qualification:

- Create instances of classes RichTextBox and MultilineRichTextBox
- Upcast them to IUndoable and execute method Undo()

Test the execution of method PolyTest() and explain the result.