Тема 5: Розробка і використання власних бібліотек в C#

В процесі розробки різних додатків власні універсальні функції доцільно не копіювати щоразу в нові проекти, а розмістити у власних бібліотеках і підключати та викликати при потребі, як це реалізовується, наприклад, з стандартними функціями вводу/виводу.

Бібліотеки мають розширення `.dll`. Вони самостійно не завантажуються, оскільки не містять точки входу – процедури `Main()`. В C# бібліотеки називаються *бібліотеками класів*. Для створення бібліотеки необхідно послідовно вибрати `Файл – Создать – Проект Visual C# - Библиотека классов`, перейти в папку проекту бібліотеки та ввести його ім'я. Після вдалої компіляції бібліотеки її DLL-файл буде знаходитися у вкладеній папці `…Bin\Debug`.

Кожна функція в C# має належати до якогось класу. Для того, щоб методи класу могли викликатися без створення його екземпляра-об’єкта, необхідно забезпечити статичність цих методів. З цією метою перед описом кожного такого методу вказують модифікатор `static`. Додатково, для уникнення створення об'єктів класів лише з статичними методами на початку їх опису теж можуть вказувати цей самий модифікатор.

По замовчуванню в проекті C# створюється простір імен з назвою, що співпадає з ім'ям самого проекту, хоча її й можна змінити. В цьому просторі імен доцільно функції групувати по класах з узагальнюючими назвами:

```
namespace LR7Unit
 {public static class analizText
   {public static bool rozdilZnak(char c)
     {if (c == ' ' | c == '.' | c == ',')
       return true;
      else
       return false;
     } // завершення опису функції
   } // завершення опису класу

  public static class corectInput
   {public static bool inputDouble(ref double x, string povidom)
     ...
   }
 } // завершення опису простору імен
```

Після введення тексту коду бібліотеки її необхідно запустити на виконання чи просто відкомпілювати та виправити помилки. Якщо помилки відсутні, то, незважаючи на повідомлення про неможливість запуску, буде створено її DLL-файл.

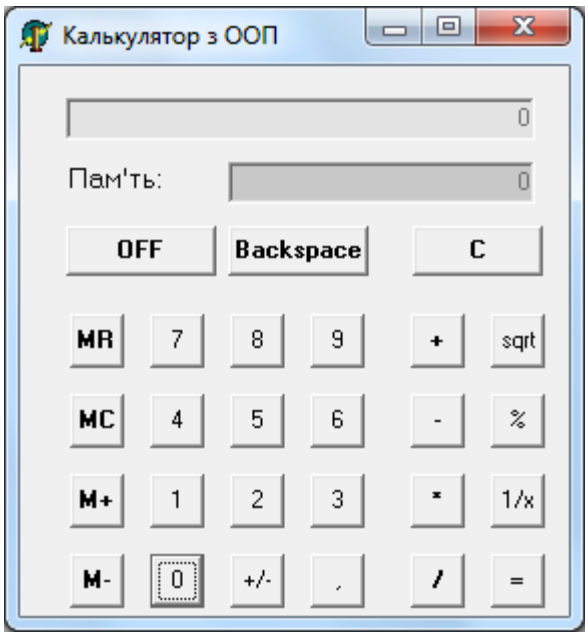
Для використання функції власної розробленої бібліотеки в **іншому** проекті необхідно:

1. Підключити в ньому файл розробленої бібліотеки. Для цього слід в панелі `Обозреватель решений` відмітити пункт `References`, обрати в його контекстному меню пункт `Добавить`, після чого натиснути кнопку `Обзор`, перейти в папку DLL-файлу бібліотеки, відмітити його і натиснути `OK`.
2. В модулі коду цього проекту відкрити простір імен розробленої бібліотеки (наприклад, при підключенні бібліотеки класів з наведеним вище кодом в модулі коду проекту вказують `using LR7Unit`);
3. В потрібних місцях коду проекту викликають розроблені методи власної бібліотеки згідно синтаксису `<назва класу>.<назва методу>` (наприклад, `analizText.rozdilZnak(S[i])`.

Тема 6: Організація взаємодії елементів керування у формах

Створіть форму-калькулятор, подібну до зразка, наведеного нижче, з власним оригінальним дизайном та забезпечте її функціональність. Введення чисел має відбуватися за допомогою кнопок “+”; “-”; “*”; зміни знаку √; % з числа; ділення на знаменник; очистку та зчитування з пам'яті.

Обов'язково відобразіть у формі вміст комірки пам’яті. У нижній частині форми виведіть інформацію про розробника.



Назвемо поле де відображаються результати - **textTablo**. А поле де відображається пам'ять – **textMemory**. Заборонимо до них доступ за допомогою властивості – **Anable**, і задамо значення по замовчуванню – (text) рівне «0».На далі забезпечимо відображення, як на табло, так і в пам'яті, хоча б однієї цифри.

1)При натисненні кнопок з зображенням цифр необхідно спочатку витерти не значущий «0», якщо він один на табло, а потім записати введену цифру. Щоб не повторювати ці дії 10 разів створимо підпрограму в класі калькулятора.

```
public partial class FormCalc : Form
   void plusTablo(char Symbol)
        { if (textTablo.Text == "0")
              textTablo.Text = "";
            textTablo.Text += Symbol.ToString();
        }
```

2) Після цього в процедурі обробки події натиснення кнопки 1 :

```
plusTablo('1');
```

Впроцедурі обробки події натиснення кнопки 2:

```
plusTablo('2');
```

І так для всіх кнопок.

3)При введенні коли незначущий «0» не відкидається, і вводити її можна лише тоді коли до цього не було, тому процедура обробки події натиснення кнопки з комою буде мати вигляд:

```
bool available = false;
        int i, len;
        len = textTablo.Text.Length;
        for (i = 0; i < len; i++)
            if (textTablo.Text[i].ToString() == ",")
            {
```

```
                        available = true;
                        break;
                    }
                if (!available)
                    textTablo.Text += ",";
```

4)Реалізуємо процедуру натиснення кнопки – **Backspace:**

```
        int len = textTablo.Text.Length;
                textTablo.Text  =  textTablo.Text.Substring     (0,
len-1);
                if (textTablo.Text == "")
                    textTablo.Text = "0";
```

5)При виконанні обчислень калькуляторів необхідно написати код попередньої операції. Цей код будемо зберігати у змінній класу – operation.

```
{ byte operation;
```

При завантаженні форми, ще не введена жодна операція, тому для форми віднайдемо подію Load і в процедурі обробки запишемо : operation = 0 .

Створимо загальну процедуру класу, яка буде виконувати обрану операцію.

```
private void runOperation()
        {
            double Visual, Memory;
            if (Operation == 0)
                return;
            try
            {
                Visual = Convert.ToDouble(textTablo.Text);
                Memory = Convert.ToDouble(textMemory.Text);
            }
            catch (System.FormatException)
            {
                MessageBox.Show("Операцію виконати
неможливо", "Увага", MessageBoxButtons.OK, MessageBoxIcon.Error);
                return;
            }
            switch (Operation)
            {
                case 1:
                    Visual += Memory;
                    Memory = 0;
                    break;
                case 2:
                    Visual = Memory - Visual;
                    Memory = 0;
                    break;
                case 3:
                    Visual *= Memory;
                    Memory = 0;
                    break;
                case 4:
                    if (Visual == 0)
                    {
                    MessageBox.Show("Операція неможлива:
ділення на нуль", "Увага", MessageBoxButtons.OK,
MessageBoxIcon.Error);
```

```
                            return;
                        }
                        Visual = Memory / Visual;
                        Memory = 0;
                        break;
                    case 5:
                        if (Visual < 0)
                        {
                        MessageBox.Show("Операція неможлива:
корінь з від'ємного числа", "Увага", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                            return;
                        }
                        Visual = Math.Sqrt(Visual);
                        Memory = 0;
                        break;
                    case 6:
                        Visual *= 0.01;
                        break;
                    case 7:
                        if (Visual == 0)
                        {
                            MessageBox.Show("Операція неможлива:
ділення на нуль", "Увага", MessageBoxButtons.OK,
MessageBoxIcon.Error);
                            return;
                        }
                        Visual = 1 / Visual;
                        break;
                    case 8:
                        Memory += Visual;
                        break;
                    case 9:
                        Visual = 0;
                        break;
                    case 10:
                        Memory = 0;
                        break;
                    case 11:
                        Visual = Memory;
                        break;
                    case 12:
                        Memory = Visual;
                        break;
                    case 13:
                        Visual *= -1;
                        break;
                    case 14:
                        Memory -= Visual;
                        break;
                }
                Operation = 0;
                textTablo.Text = Visual.ToString();
                textMemory.Text = Memory.ToString();
```

Забезпечимо виклик цієї процедури. Наприклад для кнопки «=» запишемо - **runOperation()** , для кнопки «+» виклики будуть наступними:

```
runOperation();
Operation = 12;
runOperation();
Operation = 9;
runOperation();
Operation = 1;
```

Аналогічно виконуються реалізації операцій: «+», «*», «/».

Але є операції, які мають виконуватися безпосередньо над вмістом табло, а попередня операція має бути збережена в пам'яті. Тому запам'ятаємо код операції яка міститься в пам'яті, виконаємо потрібну інформацію, і відновимо код попередньої операції. Наприклад для операції «√» процедура обробки натиснення кнопки буде мати вигляд:

```
int sqrt = Operation;
Operation = 5;
 runOperation();
Operation = sqrt;
```

Тема 7: Розробка додатків з багатьма формами

Створимо програму , головну кнопкову форму, яка буде викликати форми аналогічні розроблені нами раніше. Як відомо при створенні візуального додатку розробляється рішення, яке може містити декілька проектів, в свою чергу в кожному проекті міститься файл: program.CS, де вказується, що з проекту необхідно відразу завантажувати.

При завантаженні рішення може відразу завантажуватися декілька проектів. В кожному проекті може міститися декілька форм, але по замовчуванню створюється одна форма: FORM1.CS.

Тому для того щоб розробляти головну кнопкову форму, необхідно відкрити рішення попередніх лабораторних і змістовно перейменувати їх.

Наприклад: для калькулятора можна дати назву – FormCalc.cs.

Після цього створюємо проект головної кнопкової форми, виділяємо в ньому ім'я проекта і в його контекстному меню обираємо: додати існуючий елемент, і додаємо FormCalc.cs.

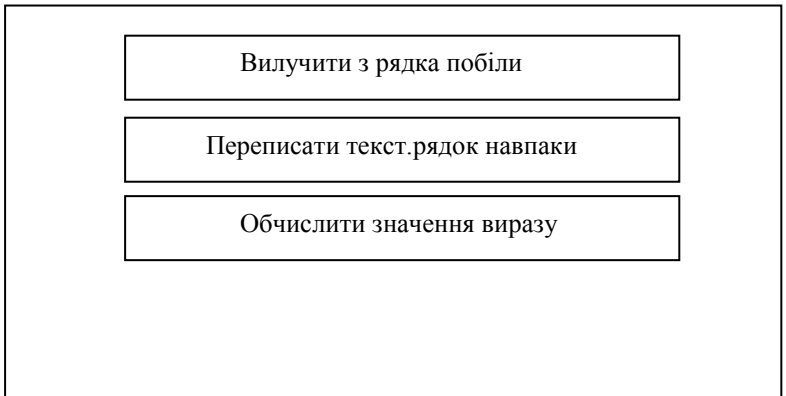
Для того щоб в головній кнопковій формі використовувати інші форми необхідно відкрити їх простір імен: namespace, як правило назва простору імен співпадає з назвою попереднього проекту Using.

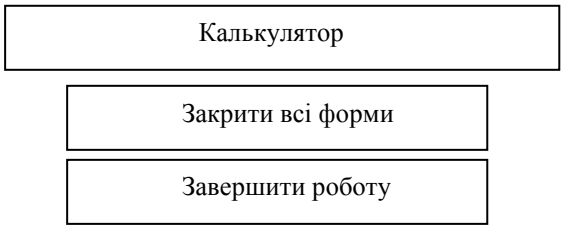
Наприклад: Using LR6 Calc;

Клас належить до посилкового типу. Оголошення змінної класу створює лише вказівку, яка нікуди не вказує. На початку завантаження головної кнопкової форми. Інші форми ще відсутні, тому оголосимо в модулі головної кнопкової форми, три вказівки на інші форми і занесемо в них значення Null.

```
public partial class Form1 : Form
    {
        Form Calc F1 = Null;
        Form Sgr F2 = Null;
        Form Math F3 = Null;
```

За допомогою цих вказівок ми будемо керувати формами, які викликаються з головної кнопкової форми. Типовий вигляд головної кнопкової форми:





Реалізуємо процедуру обробки події «закриття всіх відкритих форм». Форму можна закривати, коли вона була створена (вказівник вказує не на Null,і не було одкрита користувачем)

```
If (F1! = Null)
If (!F1. isDispouseD)
{ F1.Close();
F1 = Null
}
```

Аналогічно F2; F3

Тема 8: Робота з багатьма формами за допомогою масивів

Створюємо головну кнопкову форму з якої можна буде викликати декілька калькуляторів, форм для обробки рядків, обчислення значення виразів та керувати ними.

Для[illegible]ив вказівок на форми. Ці вказівки можуть вказувати на FormCalc, FormString або FormVuraz, а тип елемента масиву має бути однаковий, тому для цього типу оберемо спільний батківський клас Forn і заносити в цей клас масив можна довільний породжений об'єкт.

Код модуля головної кнопкової кнопкової форми:

```
Using LR6Calc;
Using LR3Pole;
Using LR5Vuraz;
Namespace LR8Menu
{ Public partial class Form1: Form
    { const maxForm=67;
       Form[ ] masForm= New Form[maxForm];
  Public Form1()
    {InitializeComponent();
     For (int i=0; i< maxForm; i++)
     masForm[i]=Null;
     }
}
```

При натисненні першої кнопки має з'явитися калькулятор, 2-ї –форма для обчислення значення виразу, 3-ї –форма для обробки рядків. При цьому нам потрібно знайти в масиві вільне місце, створити і вивести форму і записати в масив вказівку на цю форму, тому створимо підпрограму для відкриття нової форми, параметром якої буде тип потрібної форми.

```
Private Void CreateForm(int index)
{ int i=0;
   While (i<maxForm)
              If(msForm[i]==Null)
   Break;
              Else
                    i++;
   If (i==masForm)
{MessageBox.Show(«Більше форм створити неможливо»…);
   Return;
}
Switch(index)
  {Case1: masForm[1]=New FormCalc();
   Break;
   Case2: masForm[1]=New FormPole();
   Break;
```

```
        Case3: masForm[1]=New FormVuraz();
        Break;
        }
    masForm[i].Show();
    }
```

Після цього в процедурі обробки події натиснення 1-ої кнопки:

CreareForm(1);

Для 2-ої кнопки:

CreareForm(2);

Для 3-ої кнопки:

CreareForm(3);

Тема 9: Використання списків в ООП

Список – це сукупність однотипних об'єктів. Звертання до елементу списку можливо за його індексом, але найчастіше для обробки списку використовують перелічений цикл : Foreach.

Для класів значень у списках містяться конкретні значення. Для посилкових типів – вказівки на об'єкти.

Знищення елементів списку для посилкових класів не призводить до знищення відповідних об'єктів. Адже знищується лише вказівки на об'єкти.

Для створення найскоріших списків в C# використовується клас – List<T>, де T- вказує на тип елемента списку. Наприклад: для створення списку який початково містить перелік чисел необхідно вказати:

```
List <int> perelic = New List <int> ()
{1,2,7,14};
```

Початкову ємність списку можна задати в кутових дужках ставимо 7.

```
List <7> perelic = New List <int> ()
{1,2,7,14};
```

Вказуваеея початкової ємності присвоює заповнення списку, хоча може призвести до зайвих витрат пам'яті. Крім цього задання початкової ємності списку можливо за допомогою властивості: Capacity.

Основні методи списків

1. **Void ADD (T);** - додає новий елемент в список.
2. **Perelic.ADD(18);**
3. **Void ADD range (collection);** - додає елементи з іншої колекції чи масиву.
4. **Int Binary Search(T);** - виконує бінарний пошук елемента «T» у списку, при результативному пошуку повртає його індекс. Перед виконанням пошуку список має бути відсортований, для чого використовується метод: Insert.
5. **Void insert (int index, T);** - встановлює у список позицію індекс, елемент «T». Int indexOf (T);
6. **Void Sort();** - повертає індекс першоговходження елемента «T»
7. **Bool Remove (T);** - видаляє елемент «T» у списку. Якщо видалення пройшло вдало, то повертає: True, інакше повертає: False.
8. **Void Remowe At (int index);** - видаляє елемент за вказеним індексом. Для визначення кількості елементів списку використовується властивість: Count.

Виведемо, наприклад , числа зі списку «перелік».

```
Int I, count;
Count = Perelic.Count;
For (i = 0; 1 < count; i ++)
```

```
Console.WriteLine (Perelic[i]);
```
Альтернативний варіант, який перебирає самі елементи:
```
Foreach (int i in perelic)
Console.WriteLine(i);
```
Перед виконанням створюється цілочисельна змінна «i»; і за допомогою цикла Foreach в неї послідовно заносяться елементи списку – перелік. Для кожного з цих елементів виконується тіло циклу.
```
Perelic. ADD(5);
```
При використанні Foreachтип даних параметра зліва від int має співпадати з типом даних елементів списку.

Перетворимо головну кнопкову форму для забезпечення можливості опрацювання безлічі форм головної кнопкової форми.Для цього створимо список об'єктів батьківського класу: FORM
```
List <Form>
Public Partial class Form1: Form
{ List <Form> ListPashko = new List <Form>();
Provate void creute Form (intIndex)
    { Form f = NULL;
Swith (index)
     {case 1: f = New FormCalc();
Break;
case 2: f = New FormString();
Break;
     }
ListPashko. ADD(f);
F.Show();
     }
```
Перетворимо, наприклад, процедуру «пересування всіх форм в право»
```
Foreach ( Form S in  ListPashko)
If (!S.isDisposed)
f.SetDesktopLocation(f.Location. x+10)
f.SetDesktopLocation(f.Location. y);
```
Створити процедуру обробки натиснення кнопки розгортання всіх згорнутих форм з використанням циклу Foreach
```
foreach (Form f in ListKek)
                  if (!f.IsDisposed)
if (f. Windows State = Form Windows State. Normal;
```
Реалізуємо процедуру обробки зменшення розміру всіх форм з використанням циклу For int I, count = ListPashko.Count.
```
for ( i = 0; i < maxForm; i++)
                  if (ListPashko [i]. IsDisposed)
                      IsDisposed [i].size = new Size(ListPashko
[i]. Size.Wight – 10, ListPashko [i]. Size.Height – 10);
```

Тема 10: Реалізація наслідування та пізнього зв’язування C#

Загальний синтаксис опису класів **[<модифікатор доступу>]** class **<назва класу> [:<назва батьківського класу>].** Класи належать до посилкових типів, тобто для кожного об'єкту класу необхідно виділяти місце під його дані з допомогою **New**.

Перед класом можуть використовуватися наступеі модифікатори:

- **Partial** – розділюваний клас, клас може описуватися в декількох місцях, при чому в одному місці може зазначитися заголовок методу, а в іншому – його реалізація.

- **Abstract** - клас є базовим, використовується для породження інших класів, створити екземпляр такого класу не можливо.
- **Sealed** – заборона наслідування від даного класу.

C# дозволяє звертатися до поля чи методу класу без створення екземпляра. Для цього сам клас поля чи екземпляр має бути статичними і описуватися з модифікатором **static**.

- **Public** – клас має необмежений доступ і може використовуватися у всьому проекті.
- **Private** – доступ обмежений лише своїм простором імен.
- **Internal** – доступ обмежений активним файлом.
- **Protected** - доступ обмежений активним класом і породженими класами.

Створити клас «паралелограм» з доступними процедурами визначення площі, периметра та друку його даних.

```
Class parallelogram
{private Double a,b, Alfa;
Public Double Area()
{Return a,b,Math.sin (Alfa/180.Math.Pi)
}
Public Doudle Perimetr()
{ Return 2*(a+b);
}
Public Void Info()
{ Console Write.Line («дані паралелограма:» + «площа: {0}, периметр: {1},» + «два кути по {2} градусів і два кути « + « по {3} градусів» Area(), Perimeter(), Alfa, 180 – Alfa);
```

В класах для задання позначень значень використовують спеціальні методи – конструктор. Вони не повертають жодного значення і мають назву, яка співпадає з назвою класу. Клас може мати декілька конструкторів з різною кількістю чи типом параметрів.

Приклад: Створити конструктор класу паралелограма, в якій будуть передаватися дві сторони і кут між ними та буде виводитися інформація про паралелограм.

Як правило конструктори описують після полів класу.

```
Public Parallelogram ( Double start A, Doudle start B, double start Alfa)
{ a = Start A;
  b = Start B;
  Alfa = Start Alfa;
 Info();
}
}//завершення процедури реалізації класу.
```

Після цього, якщо в програмі записати - Parallelogram P1 = New Parallelogram (5,7,30); то буде створений об'єкт паралелограм.

Опишемо тепер клас квадрата, як паралелограма з однаковими сторонами і кутами:

```
Class Square: Parallelogram
{ Public Square ( Double Start A) : Base ( Start A, Start A, 90)
{}
Public Void Info()
{ Concole. WriteLine («квадрат зі стороною», a ) ;
    }
}
```

Приклад використання класу

```
Sqare D1 = new Square (8);
```

Якщо викликати безпосередньо **D1.Info()**, то виведеться інформація про квадрат, але при ініціалізації за допомогою конструктора базового класу **(Base) Parallelogram**, який для квадрата виведе інформацію, як про паралелограм.

Справа в тому, що для кожного класу по замовчуванню викликаємо свої методи. Для того ж, щоб в межах нащадка з батьківського класу викликалися методи нащадка необхідно в батьківському класі оголошувати їх віртуальними з словом Virtual, а в породженому класі оголошувати їх з словом override. При цьому реалізується пізнє зв'язування і будуть викликатися лише методи нащадки.