МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
ОДЕСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ імені І. І. МЕЧНИКОВА
ФАКУЛЬТЕТ МАТЕМАТИКИ, ФІЗИКИ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ
КАФЕДРА МЕХАНІКИ, АВТОМАТИЗАЦІЇ ТА ІНФОРМАЦІЙНИХ ТЕХНОЛОГІЙ

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

ЕЛЕКТРОННИЙ МЕТОДИЧНИЙ ПОСІБНИК

до лекційних та лабораторних занять
для студентів факультету математики,
фізики та інформаційних технологій
першого (бакалаврського) рівня освіти
спеціальності 122 Комп'ютерні науки

ОДЕСА
ОНУ
2024

УДК 004.42:004.45(075.8)(072)
О-294

Укладачі:

А. Л. Рачинська, кандидат фізико-математичних наук, доцент кафедри механіки, автоматизації та інформаційних технологій;

О. А. Недєва, викладач кафедри механіки, автоматизації та інформаційних технологій;

К. С. Палій, викладач кафедри механіки, автоматизації та інформаційних технологій.

Рецензенти:

Є. В. Малахов, доктор технічних наук, професор, завідувач кафедри математичного забезпечення комп'ютерних систем факультету математики, фізики та інформаційних технологій ОНУ імені І. І. Мечникова;

Н. П. Волкова, кандидат технічних наук, доцент, завідувач кафедри прикладної математики та інформаційних технологій Національного університету «Одеська політехніка».

Рекомендовано вченою радою факультету математики, фізики та інформаційних технологій ОНУ імені І. І. Мечникова. Протокол № 2 від 30 жовтня 2023 р.

Об'єктно-орієнтоване програмування [Електронний ресурс] : елек-
О-294 трон. метод. посіб. до лек. та лаб. занять для студентів ф-ту математики, фізики та інформ. технол. першого (бакалавр.) рівня освіти спец. 122 Комп'ютерні науки / уклад.: А. Л. Рачинська, О. А. Недєва, К. С. Палій. – Одеса : Одес. нац. ун-т ім. І. І. Мечникова, 2024. – 338 с. – 11,4 МБ.

Пропонований методичний посібник стане у нагоді при вивченні обов'язкової дисципліни «Об'єктно-орієнтоване програмування», яка викладається студентам першого (бакалаврського) рівня вищої освіти спеціальності 122 «Комп'ютерні науки» факультету математики, фізики та інформаційних технологій Одеського національного університету імені І. І. Мечникова. Він включає в себе лабораторні роботи та лекційний матеріал, що охоплює ключові аспекти об'єктно-орієнтованого програмування.

Посібник також може бути використаний у навчанні здобувачами першого бакалаврського рівня спеціальності 126 «Інформаційні системи та технології».

УДК 004.42:004.45(075.8)(072)

ЗМІСТ

ВСТУП

Цей посібник присвячений вивченню фундаментальних принципів об'єктно-орієнтованого програмування (ООП). Основні аспекти ООП, такі як абстракція, інкапсуляція, успадкування та поліморфізм, становлять важливий компонент сучасної розробки програмного забезпечення.

Студенти ознайомляться з основними поняттями, такими як класи, об'єкти, методи та властивості, і зрозуміють, як ці концепції допомагають в структуруванні та організації програмного коду.

Однією з ключових тем є абстракція, яка дає можливість створювати абстрактні класи та інтерфейси, що розділяють концепції від конкретної реалізації. Ця концепція сприяє створенню універсальних та гнучких програмних рішень.

Інкапсуляція є іншим важливим аспектом, який детально розглядається на лекціях. Вона включає в себе обмеження доступу до даних та методів класу, що сприяє збереженню конфіденційності та цілісності коду.

Успадкування дозволяє створювати ієрархії класів та підкласів, розширюючи функціональність і властивості батьківських класів. Це важливий аспект створення ефективних та легко розширюваних програмних рішень.

Частина лекцій присвячена поліморфізму, який дозволяє використовувати однакові інтерфейси для різних типів об'єктів. Ця концепція сприяє створенню загальних та універсальних алгоритмів, що можуть застосовуватися до різних об'єктів. Окрема увага приділяється використанню об'єктно-орієнтованого програмування для розробки Windows-застосувань.

Методичний посібник дозволяє студентам не лише засвоїти основні принципи об'єктно-орієнтованого програмування, але й навчитися застосовувати їх в практиці для створення сучасних програмних рішень, що оптимально працюють під платформою Windows. Робота з лабораторними завданнями допомагає студентам активно використовувати здобуті знання та навички, розвивати аналітичне та критичне мислення, а також навчитися розв'язувати складні завдання у сфері програмного забезпечення.

В результаті цього комплексного підходу студенти отримають не лише глибоке розуміння об'єктно-орієнтованого програмування, але й практичні навички, які можуть успішно застосовувати в майбутніх програмних проектах.

ЗМІСТОВИЙ МОДУЛЬ 1. Принципи ООП

1. НАЛАГОДЖЕННЯ ПРОГРАМИ І ОБРОБКА ПОМИЛОК

Помилки в програмуванні – це факт, від якого неможливо повністю позбутися. Навіть досвідчені програмісти не є винятком і також зазнають помилок. У реальності розуміння цієї невід'ємної частини "досвідченого програміста" полягає в готовності вирішувати ці помилки і навіть використовувати їх для власного зростання та покращення навичок. Звичайно, існують деякі незначні помилки, які не впливають на хід виконання програми, такі, наприклад, як помилки в правописі виведеного тексту і т. п. Крім того, існують очевидні помилки, які призводять до абсолютної неможливості використання програми (їх зазвичай називають ***фатальними***). До розряду фатальних помилок відносяться як прості помилки, які перешкоджають компіляції програми (***синтаксичні*** помилки), так і більш складні помилки, які виявляються тільки в процесі виконання коду. Однак можуть зустрічатися ще більш тонкі помилки. Наприклад, написаний застосунок не зможе включити в базу даних запис, у якому відсутній один з обов'язкових полів, або в разі виникнення будь-яких виняткових обставин включить в запис невірні дані. Помилки, подібні цим, коли деякий недолік міститься в логіці програми, відомі під назвою ***семантичних*** помилок (або помилок ***в логіці***). Дуже часто ви будете дізнаватися про наявність тонких помилок у ваших додатках тоді, коли користувач поскаржиться, що щось не працює належним чином. В результаті вам доведеться виконати трасування програми, щоб визначити, в чому саме проблема і яким чином слід змінити код, щоб він виконував ті дії, для яких був призначений. У подібних ситуаціях ви виявите, що можливості для налагодження програм, що надаються VS, можуть надати вам ні з чим незрівнянну допомогу.

Крім цього, ми вивчимо способи ***обробки помилок***, наявні в C #. Це дозволить нам приймати відповідні запобіжні заходи в тих випадках, коли поява помилок найбільш ймовірна, і створювати код, який буде здатний справлятися з помилками, що можуть в іншому випадку стати фатальними. Такі способи є складової частини мови C #, а не налагоджувальні можливості **VS**, проте і в цьому випадку **VS** пропонує деякі вельми корисні інструменти.

1.1 Налагодження в Visual Studio

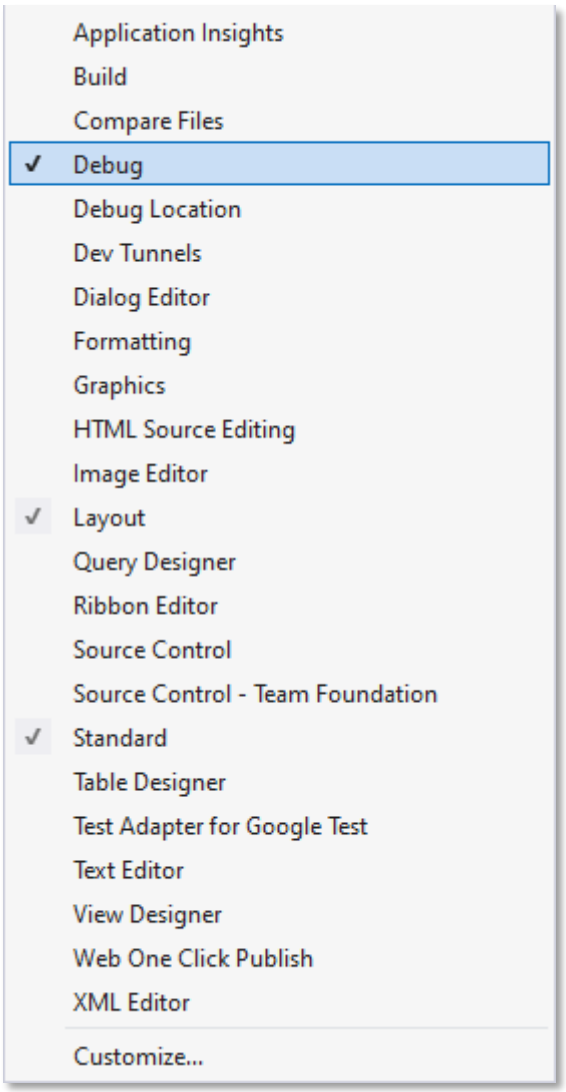
Коли програми запускаються в режимі налагодження, відбувається щось більше, ніж просто виконання написаного коду. При налагодженні створюється деяка ***символьна інформація*** про програму, яка дозволяє **VS** бути в курсі того, що саме відбувається при виконанні кожного рядка коду. Під символьною інформацією розуміється, наприклад, інформація, яка дозволяє встановлювати відповідність між іменами, використовуваними у вихідному коді, і значеннями, які існують у скомпільованому в машинні коди застосунку, що не містить інформації в зручній для сприйняття формі. Символьна інформація міститься в файлі з розширенням **.pdb**, які ви, ймовірно, зустрічали в директоріях **Debug** на вашому комп'ютері. Ця інформація дозволяє виконувати безліч корисних дій, наприклад:

- передачу налагоджувальної інформації системі **VS;**
- перегляд (і редагування) значень робочих змінних в процесі виконання за стосунку;
- припинення і продовження виконання програми;
- автоматичне припинення виконання програми в заздалегідь заданих точках;
- покрокове виконання програми (по одному рядку коду за крок);
- відстеження змін вмісту змінних в процесі виконання за стосунку;
- зміна вмісту змінних в момент виконання;
- здійснення тестових викликів функцій.

На цій лекції ви познайомитеся з цими методами, а також з їх використанням для визначення ділянок коду, які працюють не так, як очікується, і внесення в них виправлень, тобто для процесу, більш відомого під назвою "налагодження". Всі ці методи розділені на дві групи по тому, яким чином вони можуть бути застосовані. Взагалі кажучи, процедура налагодження полягає в тому, щоб або припиняти виконання програми, або зберігати інформацію для подальшого аналізу. У термінах **VS** застосунок може перебувати або в режимі виконання, або в ***режимі зупинки*** (**break mode**), що означає, що нормальне виконання програми перервано.

1.2 Налагодження в режимі зупинки

Перейти в ***режим зупинки*** можна декількома способами, і всі вони тим чи іншим шляхом призводять до тимчасової зупинки виконання програми. Найпростішим способом переходу в режим зупинки є натискання кнопки паузи в **VS** в процесі виконання програми. Ця кнопка розташовується на панелі інструментів **Debug**, яку необхідно додати до тих панелей інструментів, які з'являються в **VS** за замовчуванням. Для цього слід натиснути правою кнопкою миші в області панелі інструментів і обрати панель **Debug**.



Панель інструментів має вигляд:

Перші чотири кнопки на панелі **Debug** призначені для ручного керування зупинкою програми. Три з них пофарбовані в сірий колір, так як вони не можуть працювати з програмою, яка не виконується в даний момент. Доступна

кнопка – «**Start**» – ідентична кнопці, розташованої на стандартній панелі інструментів. Після запуску програми панель інструментів набирає вигляду:

Тепер ті три кнопки, які колись були сірого кольору, перейшли в робочий стан і дозволяють:

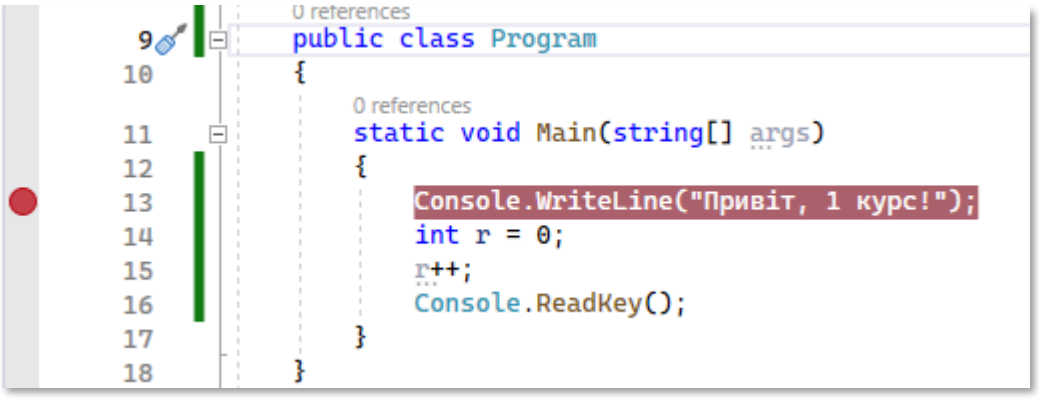
- тимчасово припинити виконання програми (зробити паузу) і перейти в режим зупинки;
- повністю припинити виконання програми (це призводить не до переходу в режим зупинки, а до повного виходу з програми);
- заново запустити застосунок.

Пауза, мабуть, є найбільш простим способом переходу в режим зупинки, але вона не дає можливості точно визначити, в якому саме місці застосунок буде зупинений. Найбільш поширене використання природних зупинок в роботі застосунку, наприклад, коли застосунок очікує введення інформації від користувача. Існує також можливість тимчасової зупинки застосунку при виконанні будь-якої тривалої операції або довгого циклу, але в цьому випадку конкретна точка, в якій ми зупинимося, виявиться абсолютно випадковою. У загальному випадку набагато корисніше використовувати точки зупинки (**breakpoints**). Точка зупинки – це деякий маркер у вихідному коді, який викликає автоматичне перемикання в режим зупинки. Можливі наступні варіанти ввімкнення:

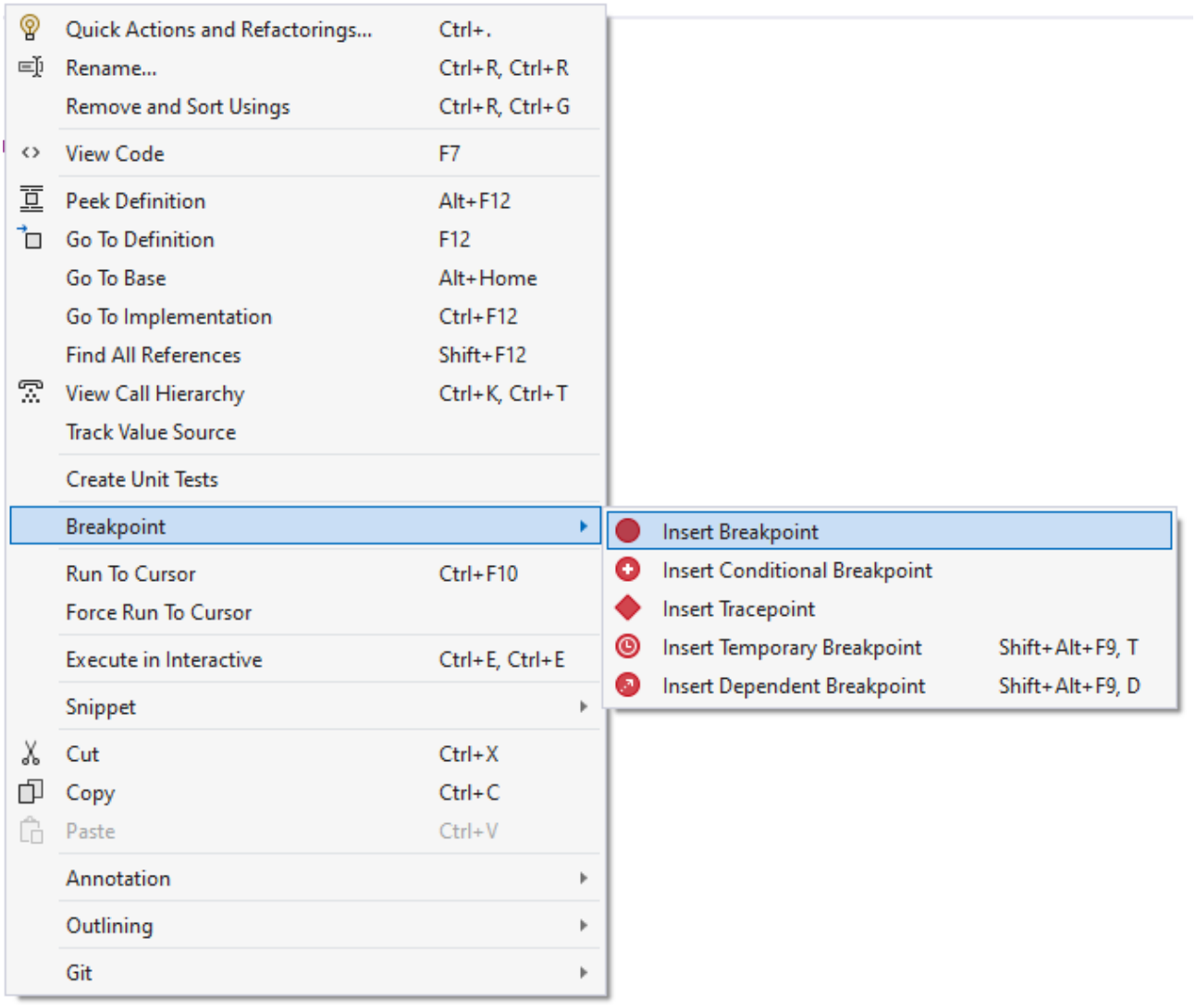
- переходити в режим зупинки негайно після досягнення точки зупину;
- переходити в режим зупинки після досягнення точки зупину в тому випадку, якщо логічний вираз має значення **true**;
- переходити в режим зупинки після досягнення точки зупину задане число раз;
- переходити в режим зупинки після досягнення точки зупину в тому випадку, якщо з моменту попереднього досягнення цієї точки значення змінної було змінено.

Зверніть увагу, що все перераховане вище виявляється можливим тільки в тому випадку, якщо застосунок будувався в режимі налагодження. Якщо ж відкомпілювати застосунок з побудовою остаточного варіанту, то всі крапки зупинки будуть проігноровані.

Існує три способи включати в програму точки зупину. Для завдання простих точок, які призводять до зупинки програми в момент їх досягнення, досить просто клацнути лівою кнопкою миші по сірій області, розташованої зліва від рядка коду,

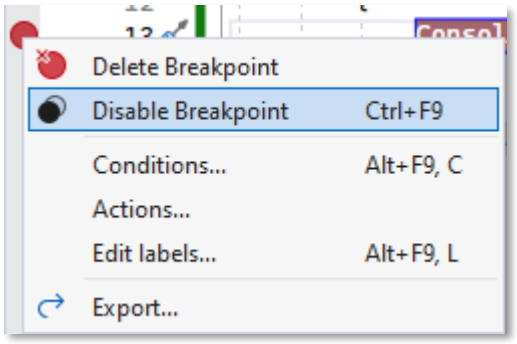


або натиснути правою кнопкою миші на самому рядку і вибрати пункт меню **Insert Breakpoint** (вставка точки зупину):

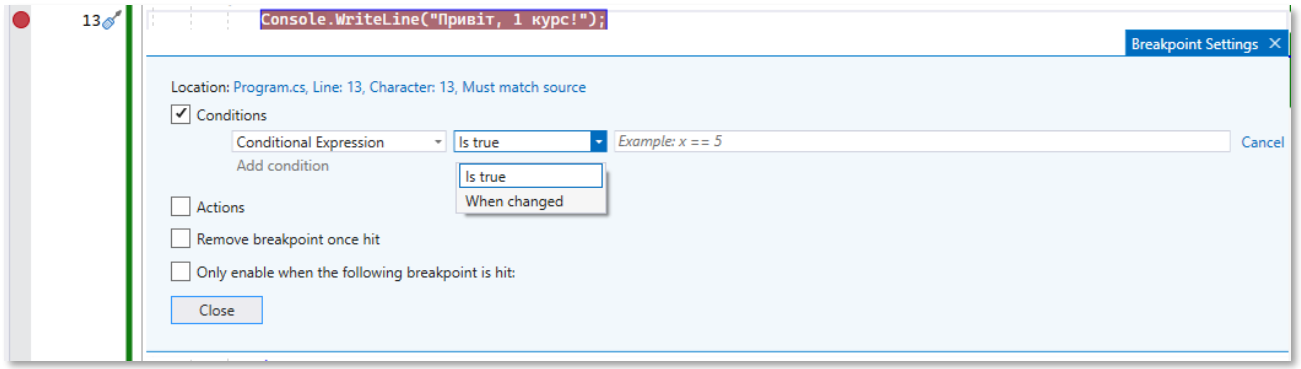


Включення точки зупину призведе до появи червоного кола поруч з відповідним рядком коду і до виділення самого рядка. Кількість точок зупину необмежена.

У точок зупину можна встановлювати властивості за допомогою контекстного меню, яке з'являється при натисканні правою кнопкою миші на точці зупинки.

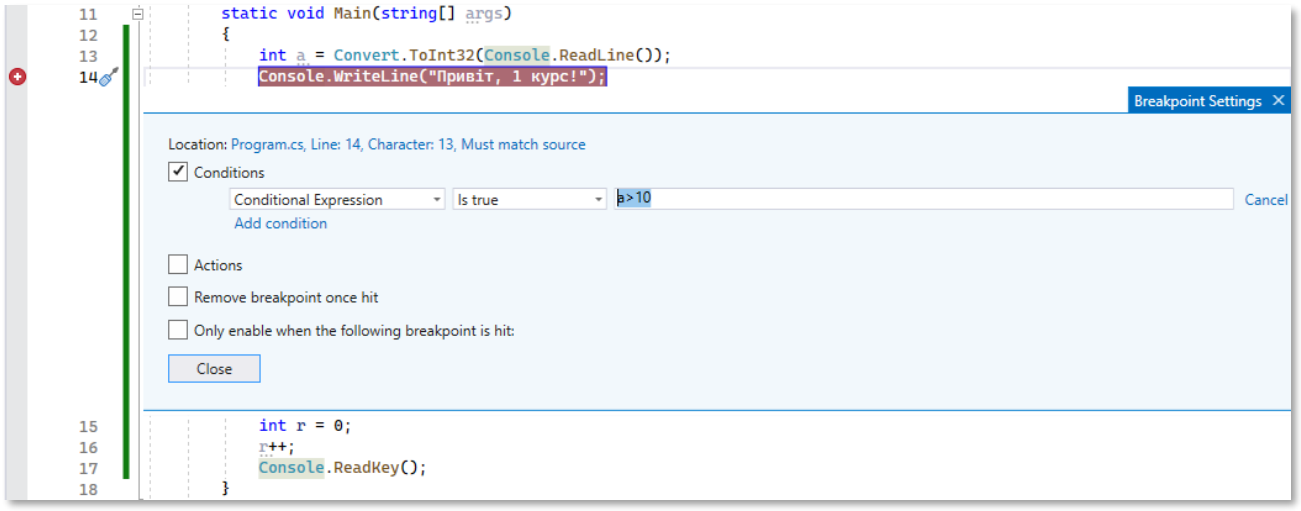


Якщо вибрати пункт меню **Conditions** *(умова)*, то з'являється вікно виду:



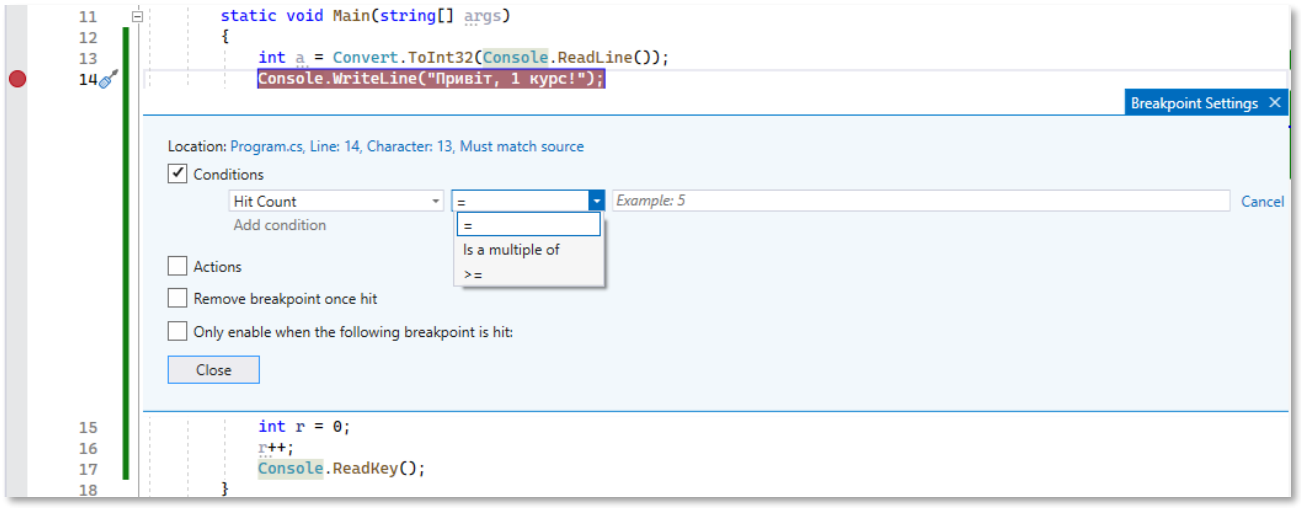
У вікні **Conditions** можна вказати, за якої умови відлагодження має бути призупинене. Наприклад, ви можете вказати, щоб відлагодження зупинялося, коли змінна має певне значення, або коли виконується певний умовний оператор (наприклад, **if-else**). В цьому вікні можна задати довільний логічний вираз, що складається з будь-яких змінних, область дії яких охоплює цю точку зупину.

Наприклад, створимо точку зупину у програмі:



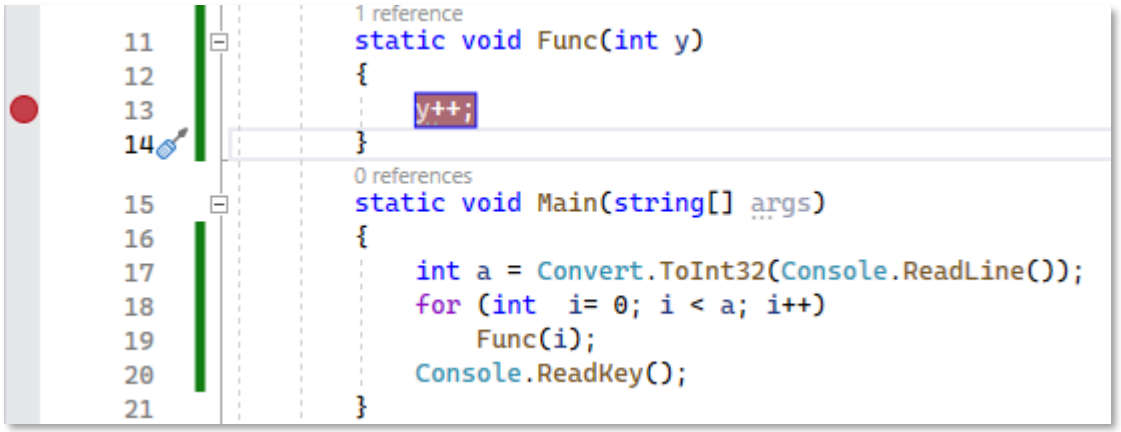
І додамо умову виду: **a >10**, залишивши вибір ***Is true***. Результат: якщо користувач введе число більше 10, то зупинка буде, в іншому випадку – ні. Цей підхід зручний, якщо ви хочете зробити зупинку, коли змінна, наприклад, в тілі циклу, перевищує деяке значення. Якщо обрати ***When changed***, то в поле необхідно ставити деяку змінну, зміни якої ви хочете відстежити.

Вибір команди **Hit Count** *(кількість попадань)* призводить до відкриття вікна:

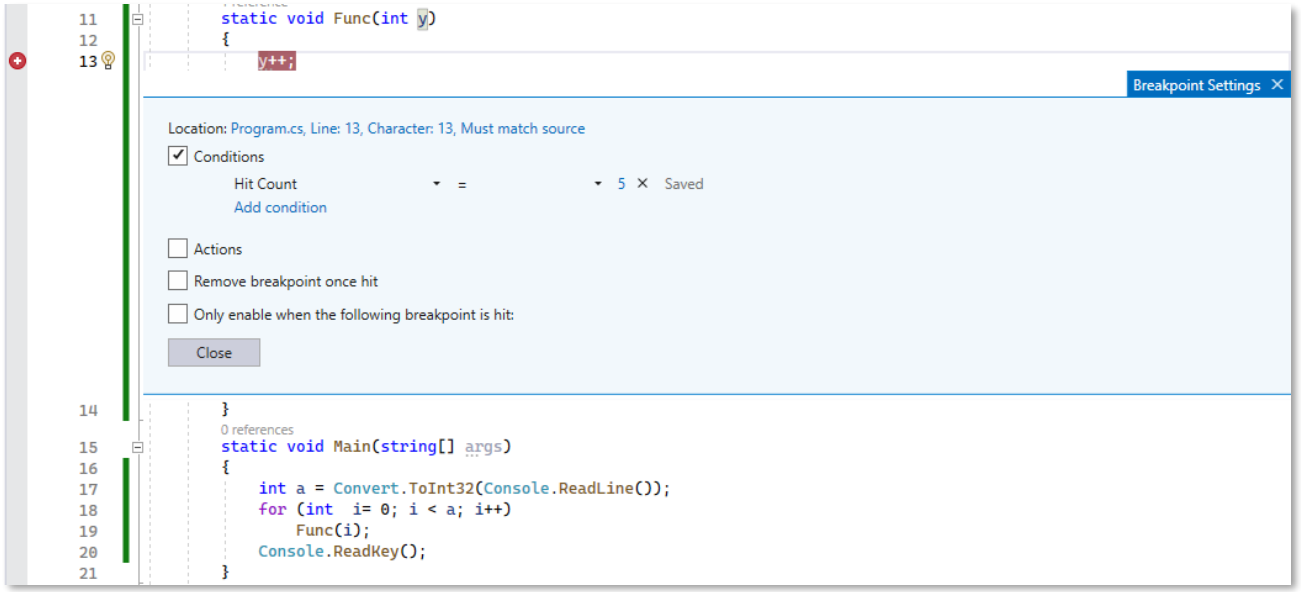


Це допомагає відлагоджувати складні або ітеративні операції, де важливо вивчити поведінку програми після певної кількості викликів коду. Ви можете встановити **Hit Count** з урахуванням потреб вашого відлагоджуваного коду для ефективного визначення проблем та їх вирішення.

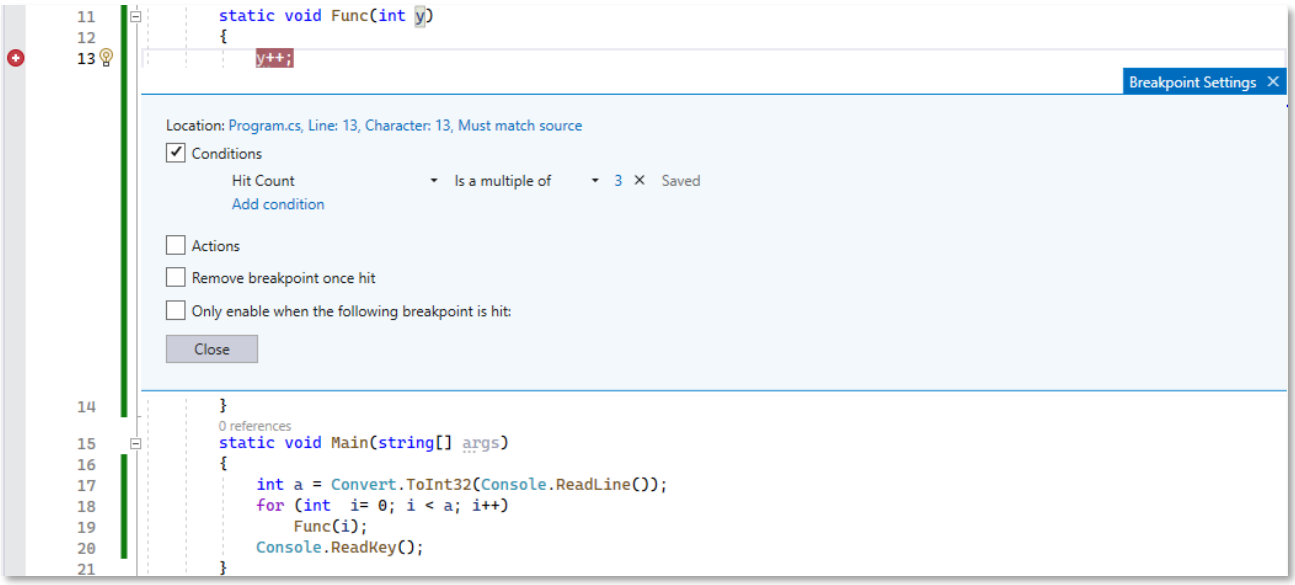
Розглянемо режим зупинки безумовно для програми:



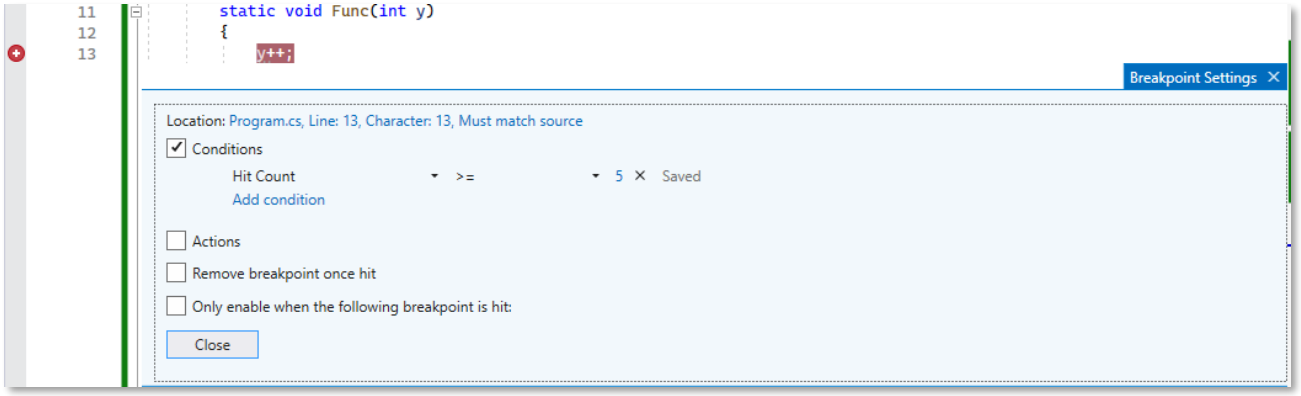
Зупинка відбудеться при першому ж виклику функції Func(), тобто при i = 0, якщо потім знову натиснути F5, то зупинка відбудеться при i = 1 і т. д.
Якщо змінити режим і встановити **Hit Count (=)**, а у текстовому полі поряд – 5, то зупинка відбудеться при y = 4 для a = 7. А продовження виконання завершить програму без зупинки.



Якщо вибрати режим **Hit Count (*Is a multiple of*)** і в поле поряд, наприклад 3, то зупинка відбудеться при y = 2, y = 5 для a = 7.

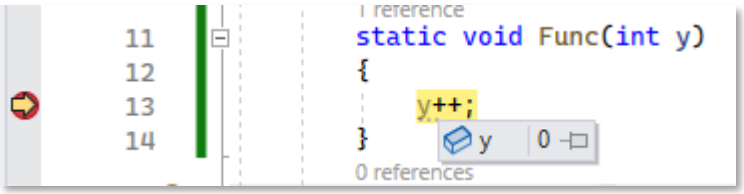


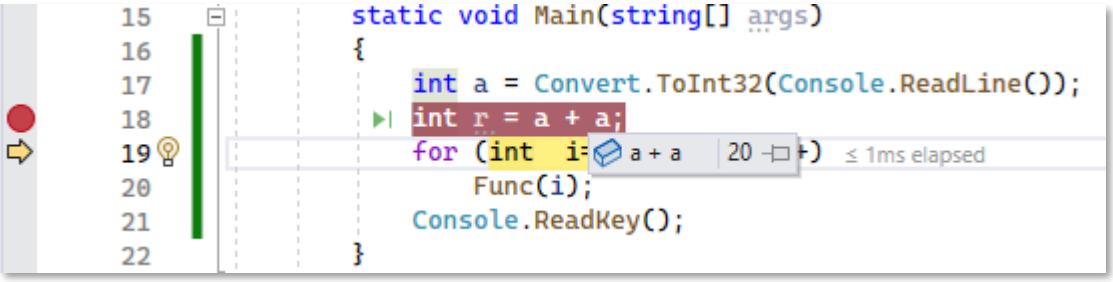
Якщо обраний режим – **Hit Count (>=)**, а значення поряд, наприклад 5, то зупинка відбудеться при y = 4 для a = 7. А продовження виконання призведе до зупинки для всіх наступних значень y.



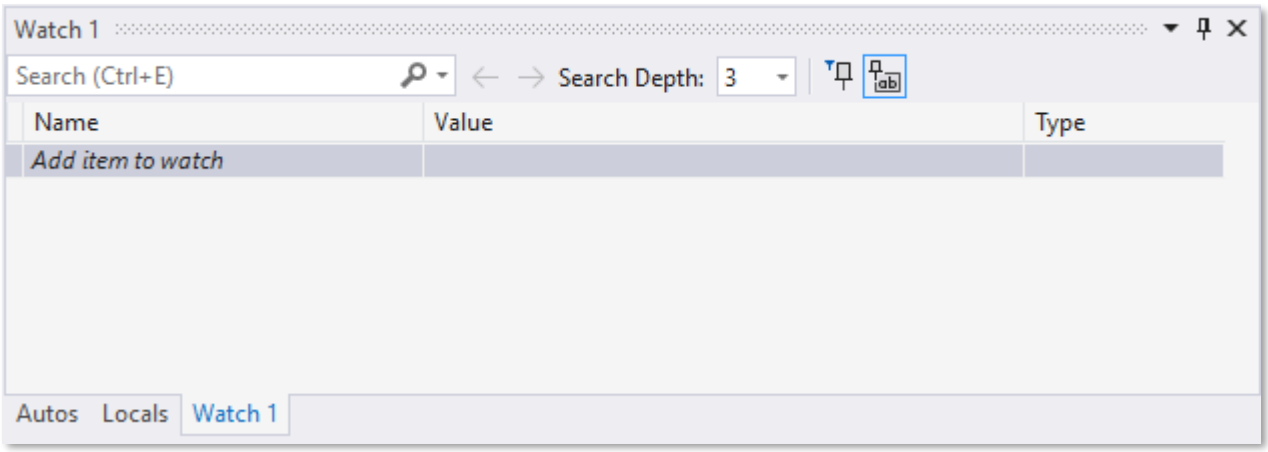
1.3 Моніторинг вмісту змінних

Моніторинг вмісту змінних являє собою приклад тієї допомоги, яку надає нам **VS**. Найпростіший спосіб дізнатися значення, яке зберігається в змінній, це підвести курсор миші до імені змінної у вихідному коді, перебуваючи в режимі зупинки. У цьому випадку на екрані з'явиться допоміжне поле жовтого кольору, в якому міститься інформація про цю змінну, включаючи її поточне значення. Ми можемо виділяти таким же чином цілі вирази, щоб отримати інформацію про результат їх виконання.

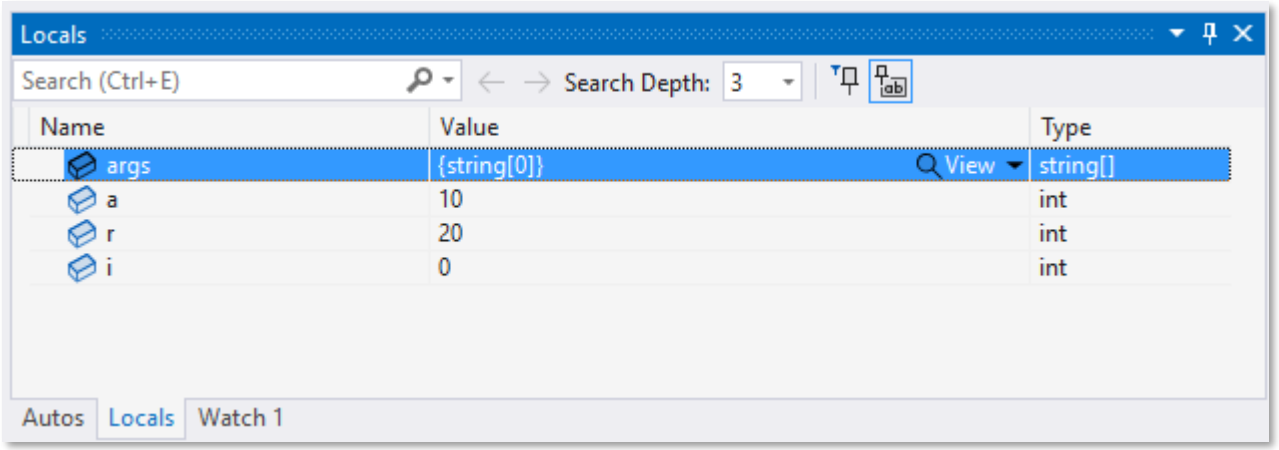




Однак, є більш зручний спосіб. Ви, ймовірно, вже встигли помітити, що коли ми запускаємо програму за допомогою **VS**, то зовнішнє уявлення різних вікон змінюється. У нижній частині вікна (як правило) з'являється вікно виду:

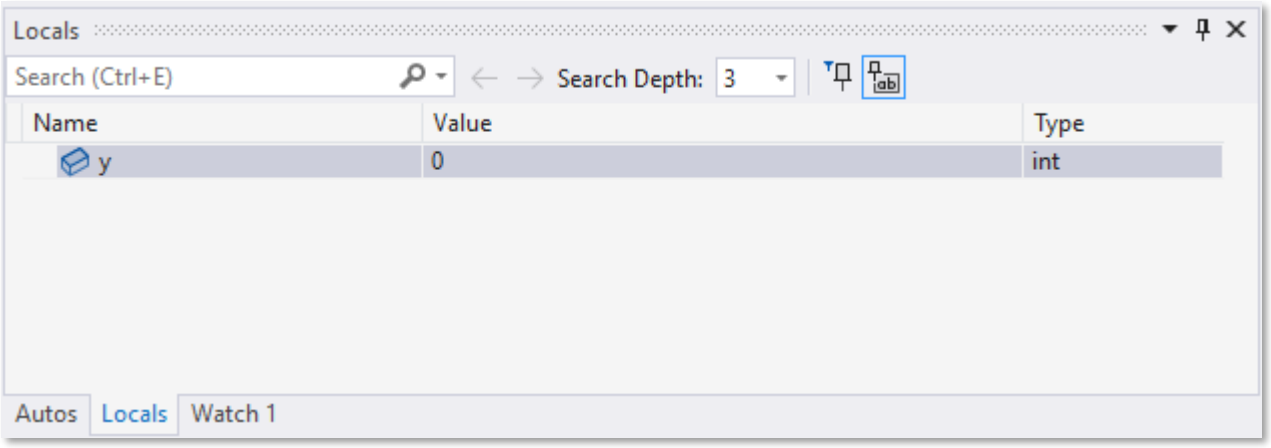


яке має три вкладки. Обираємо вкладку **Locals**, в якій відображаються значення локальних змінних.



У нашому прикладі перша безумовна точка зупинки знаходиться в функції `Main(string[] args)`, тому відображаються всі локальні змінні цієї функції: i, a, r. Після натискання клавіші **F5**, спрацьовує наступна точка зупинки в функції

`Func(int y)`, в якій локальна змінна y, тому вікно локальних змінних має вигляд:



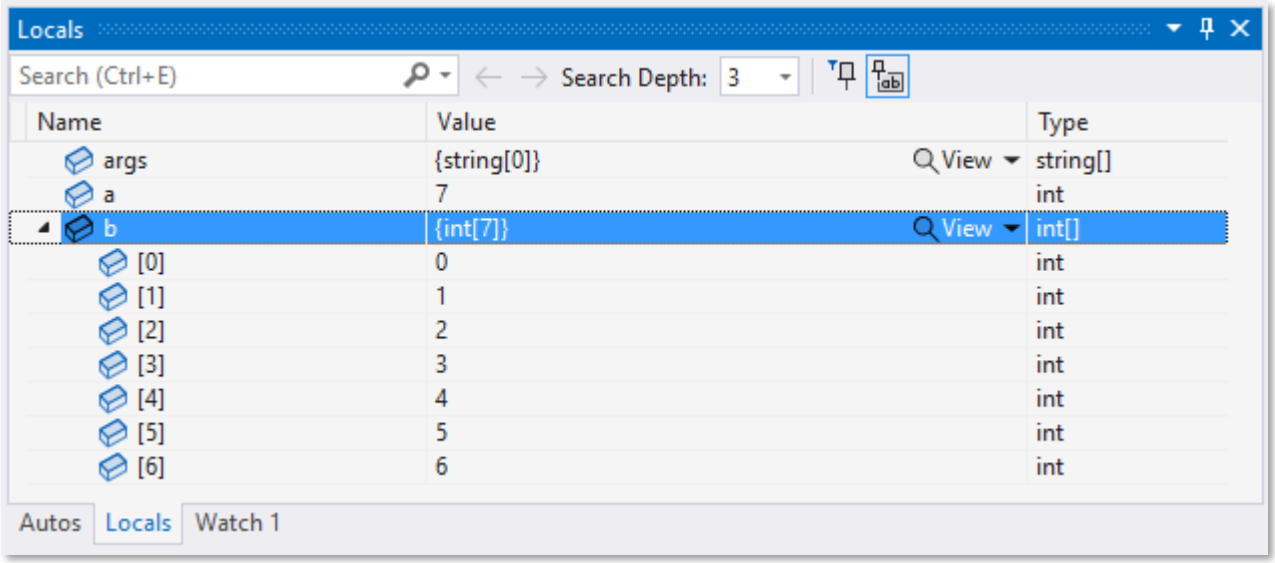
Зручним є вікно **Watch 1** – вікно спостережень, в якому можна задавати імена змінних для перегляду значень.

Більш складні змінні, такі як масиви, можуть переглядатися за допомогою додаткового запиту, для чого використовуються символи розгортання або згортання (+ і -) з лівого боку від кожного імені, що дозволяє представляти вміст змінних у вигляді дерева.

Приклад

```
15      static void Main(string[] args)
16      {
17          int a = Convert.ToInt32(Console.ReadLine());
18          int[] b = new int[a];
19          for (int i = 0; i < a; i++)
20          {
21              b[i] = i;
22          }
23          Func(5);
24      }
```

Якщо поставити точку зупинки на виконанні функції, то вікно має вигляд:



1.4 Покрокове виконання коду

Коли ми переходимо в режим зупинки, то зліва від тексту програми з'являється стрілка (якщо для переходу в режим зупинки використовувався пункт зупинки, то ця стрілка може спочатку з'являтися всередині червоного гуртка, що позначає точку зупину), яка вказує на той рядок коду, який повинен виконуватися: стрілка показує нам, якої точки програми ми досягли на момент переходу в режим зупинки. Звідси ми можемо продовжити виконання програми в покроковому режимі:

- **F10** – виконання наступного оператора без входу у функцію;
- **F11** – виконання наступного оператора з входом в функцію;
- **F5** – виконання програми до наступної точки зупину;
- **Shift + F5** – вихід з режиму **Debug** з перериванням роботи програми.

2. ПАРАДИГМА ПРОГРАМУВАННЯ ООП

У світі існує безліч різних підходів до програмування, в залежності від цих підходів формуються різні мови програмування. Перший підхід – це низькорівневий підхід, який використовується в асемблерах. Другий підхід – це процедурні мови програмування, які базуються на найпростіших базових конструкціях, які Ви вивчали в 1-му семестрі: умовні конструкції, цикли, структури і т. п. Третій підхід – об'єктно-орієнтоване програмування (ООП), яке містить в собі нові синтаксичні конструкції: клас і об'єкти. Клас є фундаментом, на якому будується вся мова C#. У класі визначено дані і код, який працює з

цими даними. Тільки зрозумівши, що представляють собою класи, об'єкти і методи, ви зможете писати більш складні і досконалі програми.

2.1 Визначення класів в C#

Для оголошення класу використовується ключове слово **class**, за яким слідує ім'я класу, потім відкривається фігурна дужка, а за нею – список даних-членів і методів класу. Оголошення завершується фігурної дужкою, що закривається:

```
8   class body
9   {
10      int speed = 0;
        0 references
11      void Move() { Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed); }
12  }
```

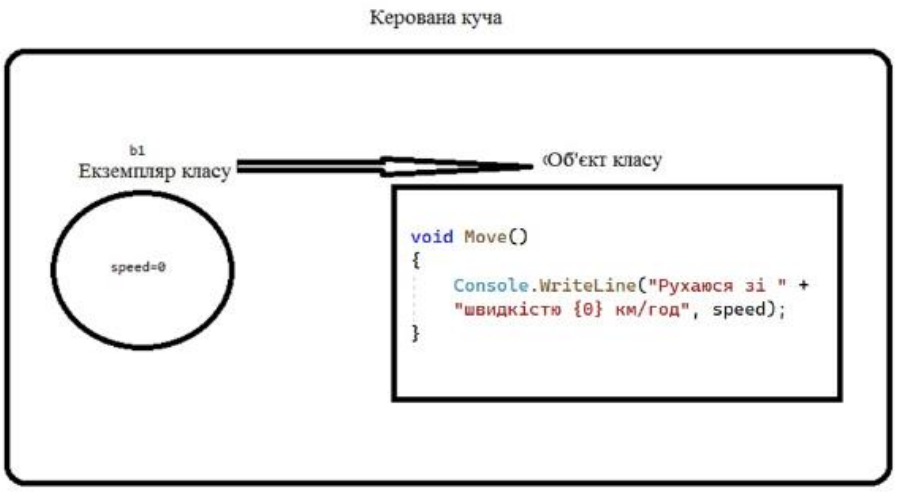
Таким чином, клас – це синтаксична конструкція, яка складається з трьох частин: службове слово **class**, ім'я класу **body** і тіло класу. Тіло класу містить в собі і інші конструкції: оголошення цілочисельної змінної **speed** і визначення функції **Move()**.

Змінні, оголошені в тілі класу, називаються полями класу (дані-члени класу). Функції, оголошені в тілі класу, називаються методами класу (функції-члени класу).

Змінна нового типу визначається в такий спосіб:

```
body b1 = new body();
```

Перед Вами оператор присвоювання, для якого спочатку виконується **r-операнд**: за допомогою службового слова **new** виділяється блок пам'яті в керованій кучі. Керована куча – це спеціальна область пам'яті, де можуть зберігатися об'єкти. Об'єкт – це копія класу в кучі.



В **l-операнде** оператора присвоювання оголошується змінна **b1** типу **body**, це змінна посилального типу, яка посилається на молодший байт блоку пам'яті, виділеної під екземпляр. У свою чергу екземпляр класу посилається на блок пам'яті, що виділений під об'єкт класу. У пам'яті екземпляра класу зберігаються значення полів, а в пам'яті блоку об'єкта класу містяться всі методи класу.

Всі поля класу характеризують стан екземпляра класу, а методи характеризують поведінку екземпляра класу.

Після створення екземпляра виникає необхідність в отриманні доступу до його членам. Доступ можна отримати за допомогою оператора прямого доступу **(.)**, якщо члени класу будуть оголошені відкритими. Для оголошення доступу до членів класу використовуються ключові слова: **private** (закриті) і **public** (відкриті). У нашому прикладі ми не вказали явно: чи є дані і метод відкритими або закритими. За замовчуванням – всі члени класу є закритими за замовчуванням. До закритих членів можна отримати доступ тільки за допомогою методів самого класу, а відкриті члени доступні для всіх інших функцій програми. При зазначеному оголошенні класу отримати доступ до членів класу неможливо, тому оголошення класу набуває вигляду:

Приклад № 1

```
8   class body
9   {
10      public int speed = 0;
        0 references
11      public void Move()
12      {
13          Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed);
14      }
15  }
```

Тепер ми можемо скористатися оператором прямого доступу **(.)**. Отже, щоб привласнити число 60 змінної-члену **speed** об'єкта **b1**, можна записати

```
b1.speed = 60;
```

Аналогічно, для виклику методу **Move()** досить використовувати такий запис:

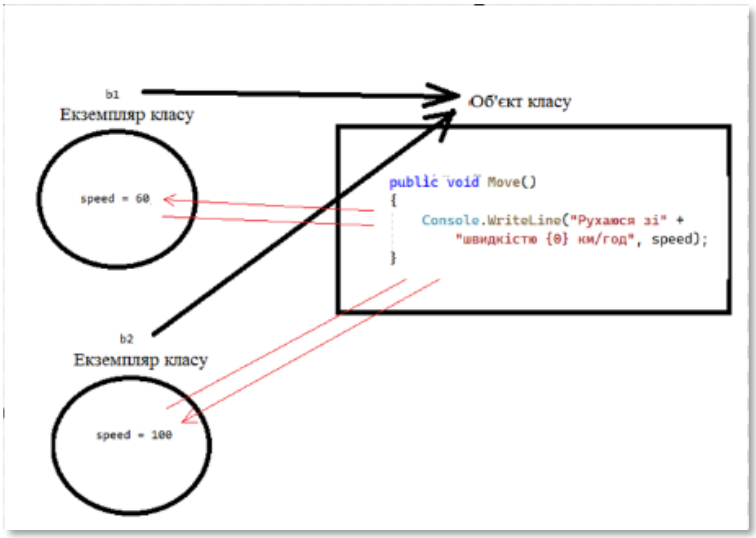
```
b1.Move();
```

В даному прикладі ми використовували сильне посилання, тобто маємо ім'я посилання, за допомогою якого ми можемо отримати доступ до примірника класу. Можна використовувати слабкі посилання:

```
new body().speed = 100;
```

Можна оголошувати кілька примірників одного класу:

```
51      static void Main(string[] args)
52      {
53          body b1 = new body();
54          b1.speed = 60;
55          b1.Move();
56
57          body b2 = new body();
58          b2.speed = 100;
59          b2.Move();
60      }
```



Відповідно до загальної стратегії об'єктно-орієнтованого програмування змінні-члени класу слід залишати закритими. ***Це перший принцип ООП: принцип інкапсуляції***, тобто приховування даних від користувача класу.

Доступ слід відкривати тільки до функцій-членів класу, що забезпечує доступ до його закритих даних (ці функції називають методами доступу). Ці методи можна викликати з будь-якого місця в програмі для повернення або установки значень закритих змінних-членів.

Приклад 2

```
8       class body
9       {
10          private int speed = 0;
            0 references
11          public void SetSpeed(int value)
12          {
13              if (value >= 0)
14                  speed = value;
15          }
            0 references
16          public int GetSpeed()
17          {
18              return speed;
19          }
            0 references
20          public void Move()
21          {
22              Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed);
23          }
24      }
```

Тоді для модифікації значення **speed** об'єкта **b1** необхідно викликати функцію **SetSpeed()**:

```
b1.SetSpeed(60);
```

Метод запису **SetSpeed ()** називається також методом-мутатором (***mutator***) або **setter**, а метод читання **GetSpeed ()** – методом-аксессором (***accessor***) або ***getter***.

2.2 Властивості

Властивість – це спеціальний тип членів класу. Властивість включає в себе поле і методи доступу до цього поля. Властивість пропонує більш зручний спосіб вирішення проблеми доступу до закритого поля порівняно з методами доступу, але це особливість мови C#.

Властивість складається з імені і пари мутатора ***set*** і аксессора ***get***. Основна перевага властивості полягає в тому, що його ім'я можна використовувати у виразах і інструкціях присвоювання подібно звичайній змінній, хоча в дійсності тут будуть автоматично викликатися ***get-аксессор*** і та ***set-мутатор***.

Формат запису властивості такий:

```
тип ім’я{
get
{
// код аксессору читання поля
}
set
{
// код мутатору запису поля
}
```

Тут тип – це тип властивості (наприклад, **int**), а ім'я – його ім'я. Після визначення властивості будь-яке використання її імені означає виклик відповідного мутатора-аксессора. Аксессор **set** автоматично приймає параметр з іменем **value**, який містить значення, що присвоюється властивості.

Важливо розуміти, що властивості не визначають область пам'яті. Отже, властивість управляє доступом до поля, але самого поля не забезпечує. Це поле повинно бути задано незалежно від властивості.

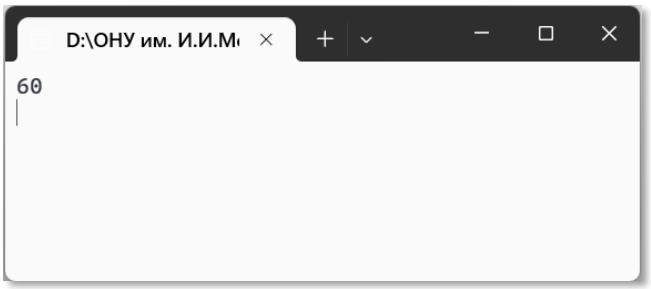
Приклад № 3

Розглянемо приклад, де рішення **Solution Example_1** включає проект

Example_1_1

```
6   namespace Example_1_1
7   {
        2 references
8       class body
9       {
10          private int speed = 0;
            2 references
11          public int Speed
12          {
13              set
14              {
15                  if (value >= 0)
16                      speed = value;
17              }
18              get
19              {
20                  return speed;
21              }
22          }
            0 references
23          public void Move()
24          {
25              Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed);
26          }
27      }
        0 references
28      class Program
29      {
            0 references
30          static void Main(string[] args)
31          {
32              body b1 = new body();
33              b1.Speed = 60;
34              Console.WriteLine(b1.Speed);
35              Console.ReadKey();
36          }
37      }
38  }
```

Результат:



Властивість **Speed** призначена для читання і запису, оскільки дозволяє як прочитати вміст свого базового поля, так і записати в нього нове значення. Але можна створювати властивості, призначені тільки для читання (визначивши лише ***get-аксессор***) або тільки для запису (визначивши лише ***set-мутатор***).

На використання властивостей накладаються досить серйозні обмеження. По-

перше, оскільки у властивості не визначається область пам'яті, його не можна передавати методу в якості **ref**- або **out**-параметра. По-друге, властивість не можна перевантажувати. (При необхідності ви можете мати дві різні властивості, які використовують одну і ту ж базову змінну, але до такої організації властивостей вдаються нечасто.) Нарешті, властивість не повинна змінювати стан базової змінної при виклику ***get-аксессора***, хоча недотримання цього правила компілятор виявити не в змозі.

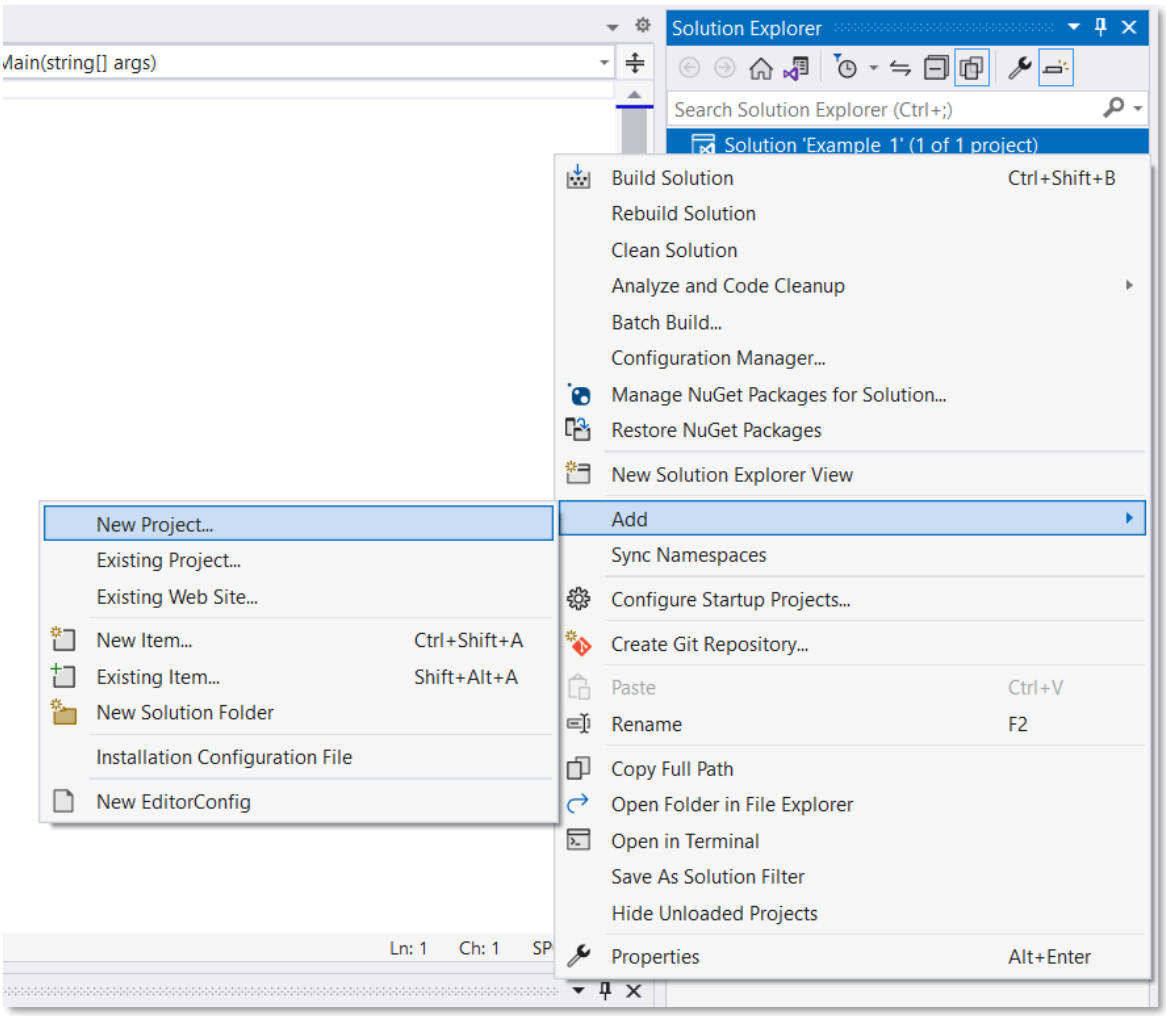
2.3 Конструктор

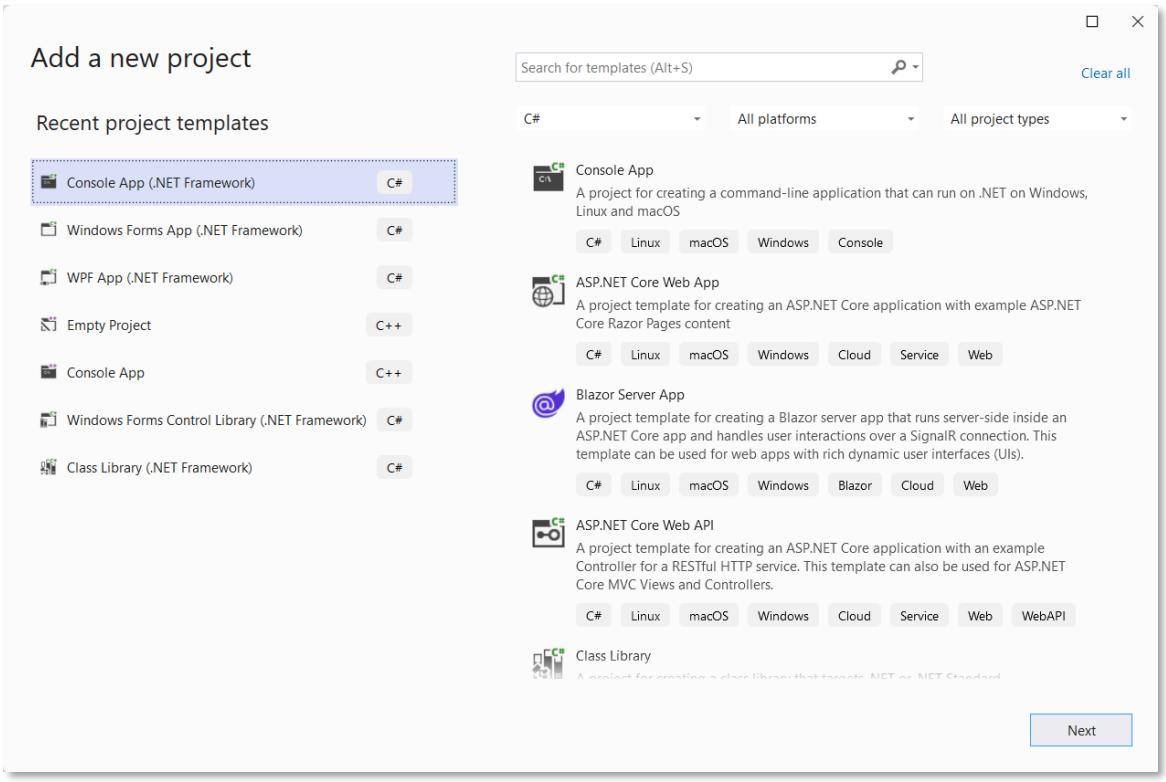
У класі є спеціальний метод, званий конструктором, який викликається під час побудови класу. Конструктори бувають двох видів: конструктори за замовчуванням (без параметрів) і призначені для користувача конструктори. Призначений конструктор в першу чергу – для ініціалізації полів класу. При необхідності конструктор може приймати параметри, але не може повертати значення навіть типу **void**. ***Конструктор – це метод класу, ім'я якого збігається з ім'ям самого класу.*** Конструктор може бути перевантаженим. Крім того, зазвичай в якості ідентифікатора доступу використовується **public**, оскільки конструктори, як правило, викликаються поза ними їхнім класом.

Попередження. *Якщо в тілі класу не оголосити жодного конструктора, то буде автоматично додано конструктор за замовчуванням. Якщо в класі оголосити хоча б один користувацький конструктор, то конструктор за замовчуванням додаватися не буде.*

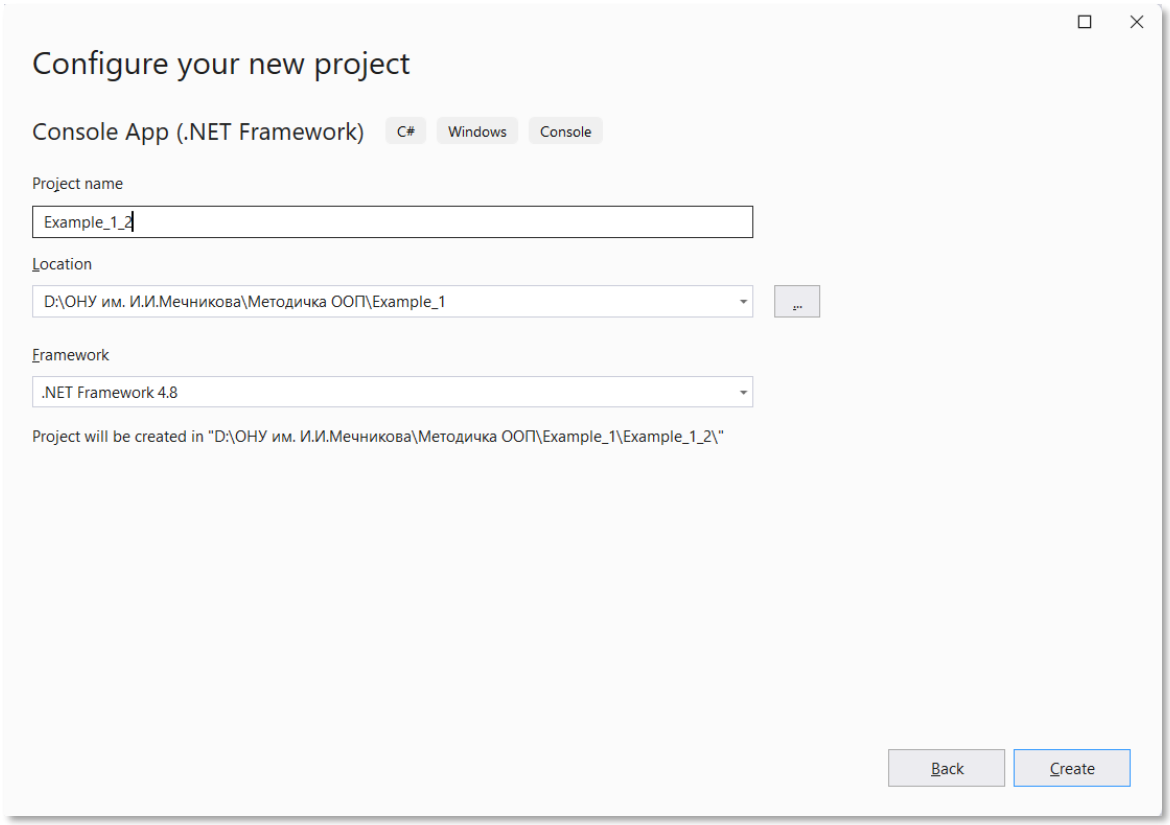
Перш ніж розглянути новий приклад, внесемо зміни в структуру нашого рішення.

1. Додамо новий проект **Example_1_2** за допомогою оглядача рішень **Solution Explorer**. Для цього вибираємо ім'я рішення і викликаємо контекстне меню; натисканням правої кнопки миші вибираємо команду **Add =>New Project…**

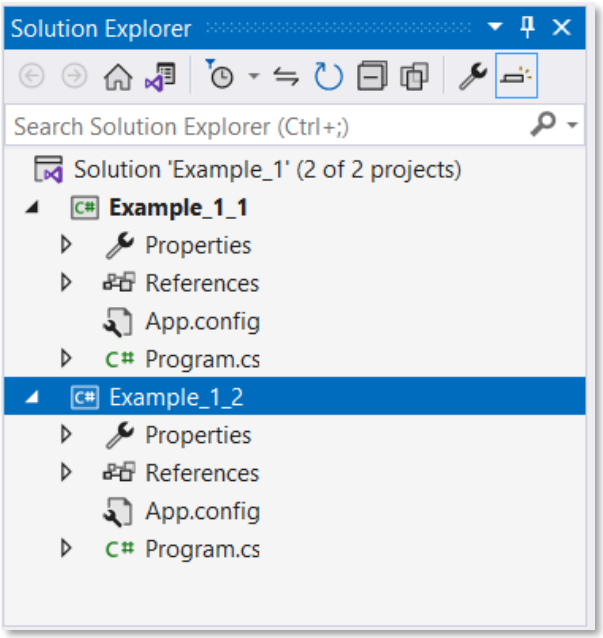

Solution Explorer
Main(string[] args)
Search Solution Explorer (Ctrl+;)
Solution 'Example_1' (1 of 1 project)
Build Solution Ctrl+Shift+B
Rebuild Solution
Clean Solution
Analyze and Code Cleanup
Batch Build...
Configuration Manager...
Manage NuGet Packages for Solution...
Restore NuGet Packages
New Solution Explorer View
Add
Sync Namespaces
Configure Startup Projects...
Create Git Repository...
Paste Ctrl+V
Rename F2
Copy Full Path
Open Folder in File Explorer
Open in Terminal
Save As Solution Filter
Hide Unloaded Projects
Properties Alt+Enter
New Project...
Existing Project...
Existing Web Site...
New Item... Ctrl+Shift+A
Existing Item... Shift+Alt+A
New Solution Folder
Installation Configuration File
New EditorConfig
Ln: 1
Ch: 1


Add a new project
Search for templates (Alt+S)
Clear all
Recent project templates
C#
All platforms
All project types
Console App (.NET Framework) C#
Windows Forms App (.NET Framework) C#
WPF App (.NET Framework) C#
Empty Project C++
Console App C++
Windows Forms Control Library (.NET Framework) C#
Class Library (.NET Framework) C#
Console App
A project for creating a command-line application that can run on .NET on Windows, Linux and macOS
C# Linux macOS Windows Console
ASP.NET Core Web App
A project template for creating an ASP.NET Core application with example ASP.NET Core Razor Pages content
C# Linux macOS Windows Cloud Service Web
Blazor Server App
A project template for creating a Blazor server app that runs server-side inside an ASP.NET Core app and handles user interactions over a SignalR connection. This template can be used for web apps with rich dynamic user interfaces (UIs).
C# Linux macOS Windows Blazor Cloud Web
ASP.NET Core Web API
A project template for creating an ASP.NET Core application with an example Controller for a RESTful HTTP service. This template can also be used for ASP.NET Core MVC Views and Controllers.
C# Linux macOS Windows Cloud Service Web WebAPI
Class Library
Next

2. Відкривається вікно створення нового проекту, який ми можемо назвати: **Example_1_2**



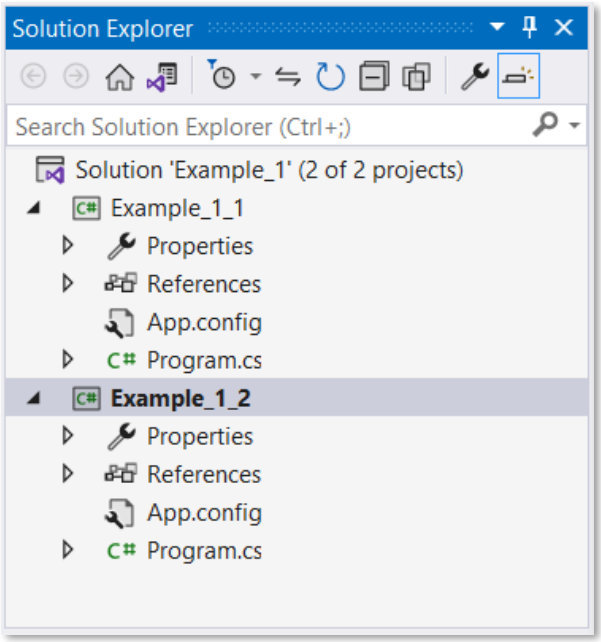
3. Вікно оглядача рішень **Solution Explorer** показує, що тепер рішення включає два проекти:



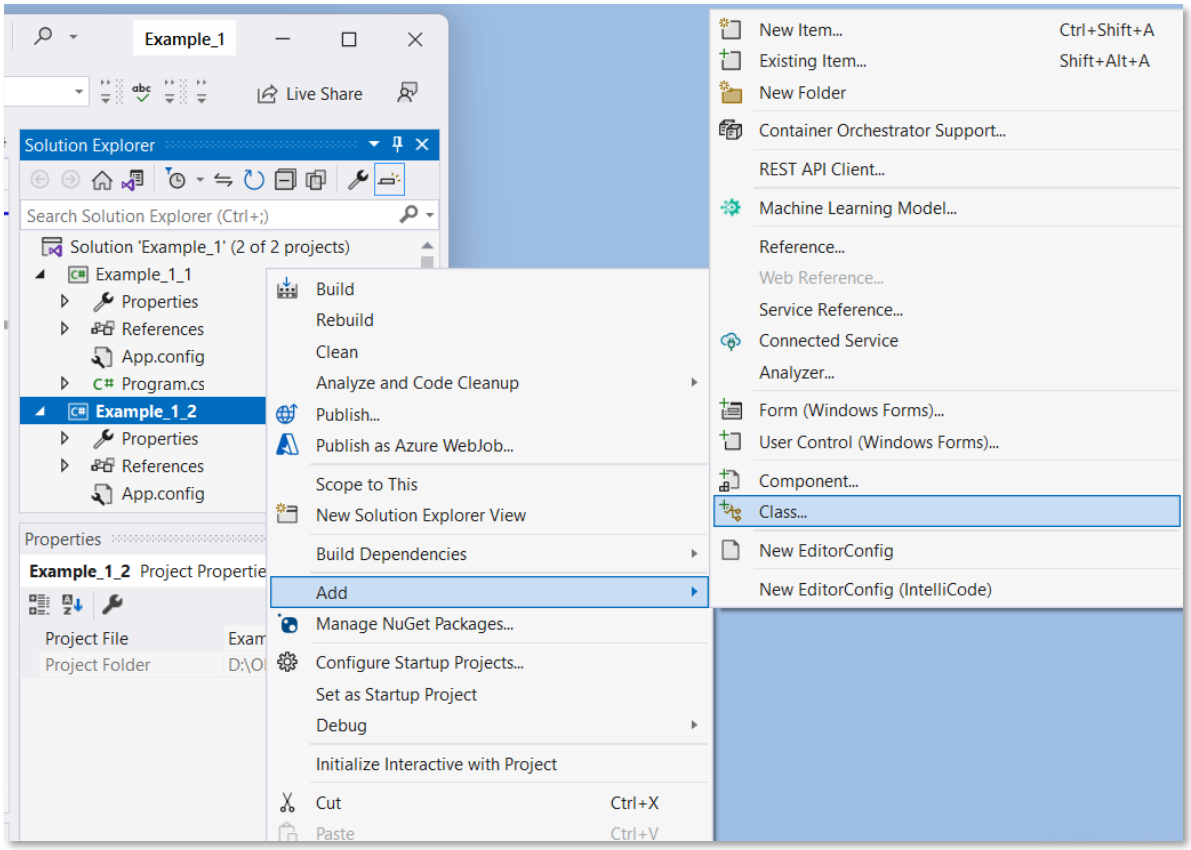
4. Чорним кольором підсвічується перший проект **Example_1_1**, це означає, що цей проект буде запускатися під час виконання рішення.
5. Змінимо проект рішення, що запускається: вибираємо **Example_1_2** і

викликаємо його контекстне меню. Обираємо команду: **Set as Startup Project**.

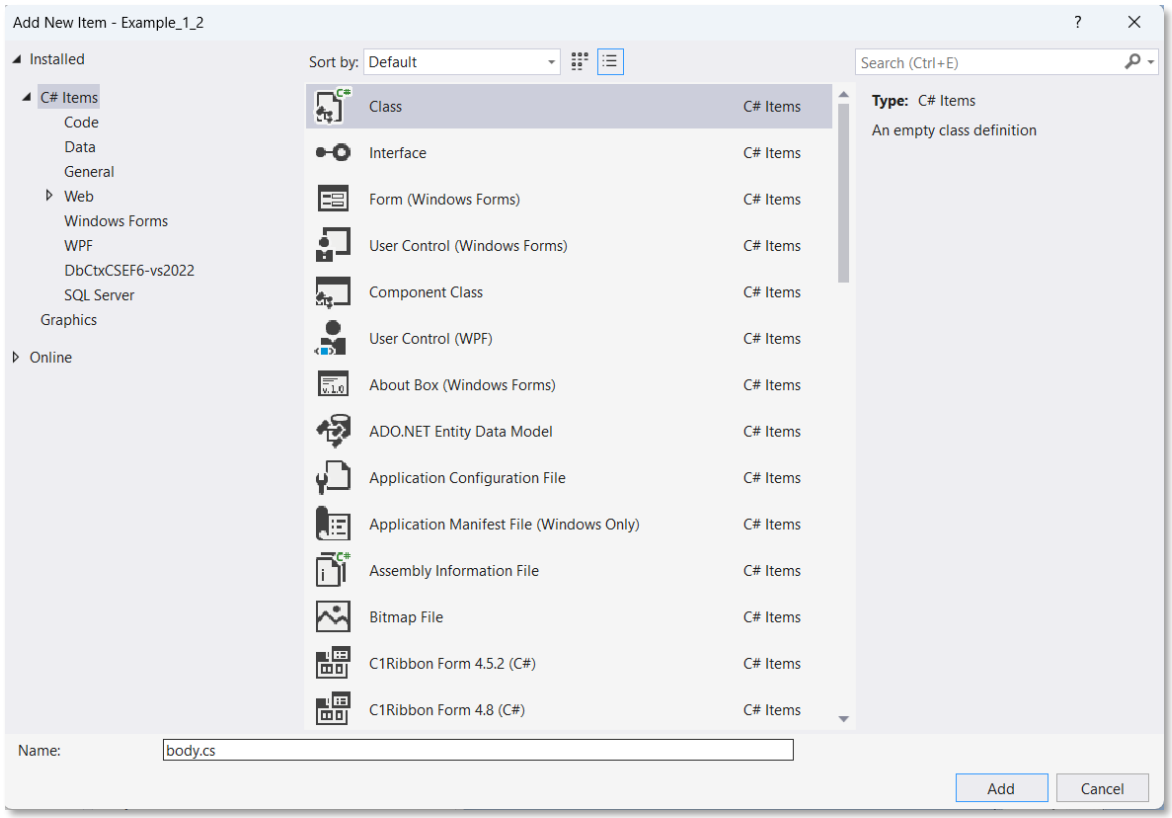
6. Після виконання даної команди у вікні оглядача рішень буде підсвічуватися чорним шрифтом ім'я другого проекту:



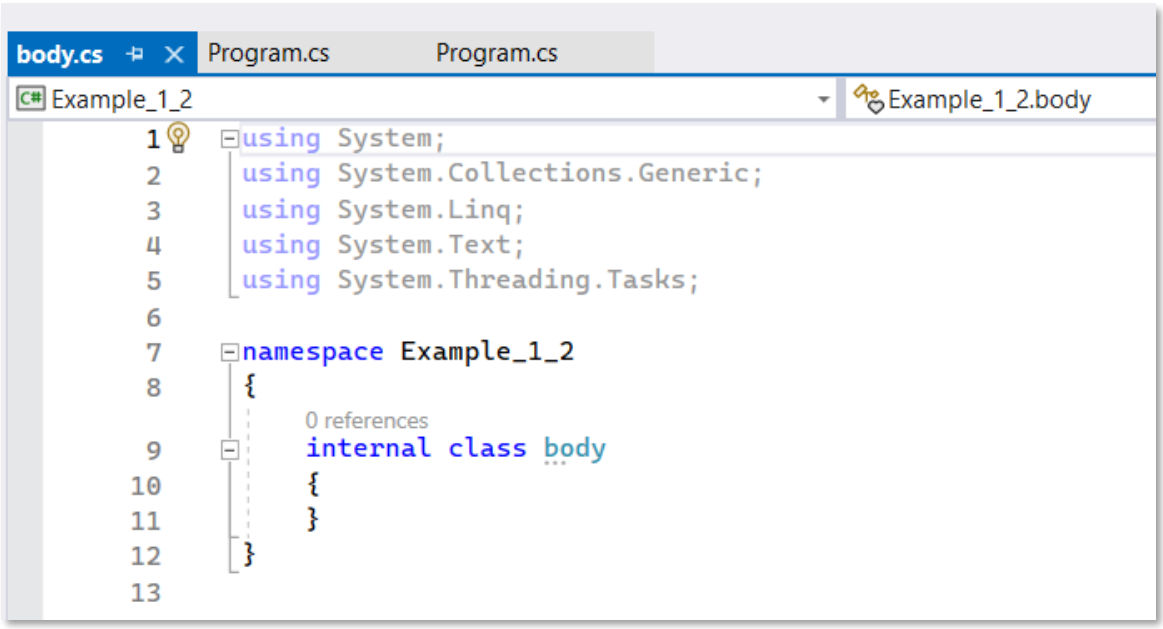
7. Необхідно розмістити клас **body** в окремому файлі, що є хорошим стилем програмування. Щоб додати в проект **Example_1_2** новий клас викликаємо контекстне меню проекту **Example_1_2** і обираємо команду **Add => Class**



8. Відкривається вікно створення нового класу – **body.cs**



9. Після цього відкриється вікно редактора коду з доданим кодом:



Додамо в тілі класу два нових поля: масу, назву і відповідні властивості, а також два конструктори – за замовчуванням і призначений для користувача, і метод виведення інформації про стан екземпляра.

Приклад 4

body.cs | Program.cs | Program.cs

Example_1_2 | Example_1_2.body

```
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Example_1_2
7   {
        6 references
8       class body
9       {
10          private int speed;
11          private string name;
12          private double mass;
            0 references
13          public int Speed
14          {
15              set
16              { if (value >= 0)   speed = value; }
17              get
18              { return speed; }
19          }
            0 references
20          public string Name
21          {
22              set
23              { name = value; }
24              get
25              { return name; }
26          }
            0 references
27          public double Mass
28          {
29              get
30              { return mass; }
31          }
            1 reference
32          public body()
33          {
34              speed = 0;
35              name = "Просто тіло";
36              mass = 2.5;
37          }
```

```
38
            3 references
39          public body(string myName, int mySpeed, double myMass)
40          {
41              speed = mySpeed;
42              name = myName;
43              mass = myMass;
44          }
            0 references
45          public void Move()
46          {
47              Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed);
48          }
            6 references
49          public void Inform()
50          {
51              Console.WriteLine("Я тіло {0} маю масу {1} і швидкість {2}", name, mass, speed);
52          }
```

```
Program.cs    body.cs    Program.cs
Example_1_2                          Example_1_2.Program
1   using System;
2   using System.Collections.Generic;
3   using System.Linq;
4   using System.Text;
5
6   namespace Example_1_2
7   {
        0 references
8       class Program
9       {
            0 references
10          static void Main(string[] args)
11          {
12              body b1 = new body();
13              b1.Inform();
14
15              body b2 = new body("Колесо", 100, 10.5);
16              b2.Inform();
17          }
18      }
19  }
```

Результат:

```
D:\ОНУ им. И.И.Ме
Я тіло Просто тіло маю масу 2,5 і швидкість 0
Я тіло Колесо маю масу 10,5 і швидкість 100
```

Зауваження 1. *Конструктор за замовчуванням викликається при виконанні команди виду:*

```
body b1 = new body();
```

В даному випадку викликається конструктор без вхідних параметрів, тому об'єкт b1 матиме speed = 0 і mass = 2.5, як значення за замовчуванням.

Зауваження 2. *Конструктор з вхідними параметрами викликається при виконанні команди виду:*

```
body b2 = new body("Колесо", 100, 10.5);
```

Зауваження 3. *Якщо ви не оголосите конструктор, то компілятор зробить це за вас. Стандартний конструктор не приймає аргументів, він ініціалізує всі змінні-члени, які мають тип значень, нулями, а змінні-члени посилального типу – **null**-значеннями. Але якщо ви визначите власний конструктор, то конструктор за замовчуванням більше не використовується.*

2.4 Ключове слово this

При виклику будь-якого з методів класу, у нього неявним чином передається аргумент, який являє собою посилання на екземпляр, що його викликає (тобто екземпляр, для якого викликається метод). Це посилання і називається ключовим словом **this**.

Розглянемо приклад використання **this**. У прикладі 2 для конструктора з вхідними параметрами використовувалася сигнатура вигляду:

```
public body(string myName, int mySpeed, double myMass);
```

Багато програмістів вважають за краще конструктор з вхідними параметрами виду:

Приклад 5

```
public body(string name, int speed, double mass)
        {
            this.speed = speed;
            this.name = name;
            this.mass = mass;
        }
```

Ключове слово **this** має ще одне функціональне призначення. Розглянемо побудову конструкторів класу **body** за наступною схемою:

Приклад 6

```
public body(string name, int speed, double mass)
        {
            this.speed = speed;
            this.name = name;
            this.mass = mass;
        }
        public body(string name, int speed)
            : this(name,speed,2.5)
        {
        }
        public body(string name, double mass)
            : this(name, 0, mass)
        {
        }
        public body(string name)
            : this(name, 0, 2.5)
        {
        }
        public body()
            : this("Просто тіло", 0, 2.5)
        {
        }
```

Зауваження 1. *Використання службового слова* ***this*** *в такому контексті призводить до виклику конструктора цього ж класу з відповідною сигнатурою.*

3. ОБ'ЄКТИ І КЛАСИ

3.1 Змінні типу посилання

Для створення екземпляра класу **body** використовувалася команда виду:

```
body b1 = new body();
```

Це оголошення виконує дві дії. По-перше, воно оголошує змінну з ім'ям **b1** типу **body**. По-друге, за допомогою оператора **new** виділяється область з керованої кучі під змінну типу **body**. Оператор **new** повертає посилання на область пам'яті, яка буде присвоюватися змінній **b1**. Це посилання (яке збережено в конкретній змінній) служить адресою об'єкта в пам'яті, виділеної для нього оператором **new**. Тому змінні класового типу називають змінними ***типу посилання***.

Змінні ***типу посилання*** можуть мати значення **null**. Наприклад, якщо вищевказану команду розбити на дві команди:

```
body b1;
b1 = new body();
```

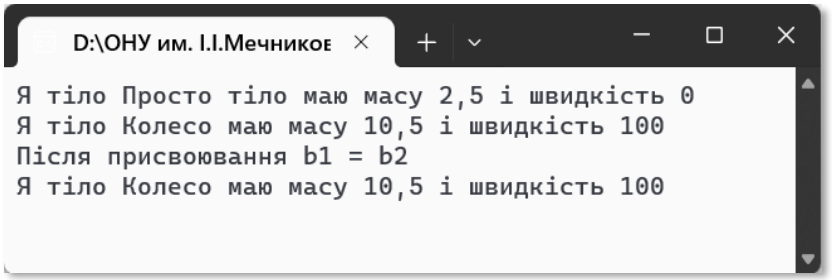
Після виконання першої команди значення змінної **b1** буде дорівнювати **null**, що означає – це посилання ні на що не посилається.

В операції присвоєння змінні типу посилання діють не так, як змінні типу значень (наприклад, типу **int**). Присвоюючи одній змінній (типу значення) значення іншої, ми маємо досить просту ситуацію.

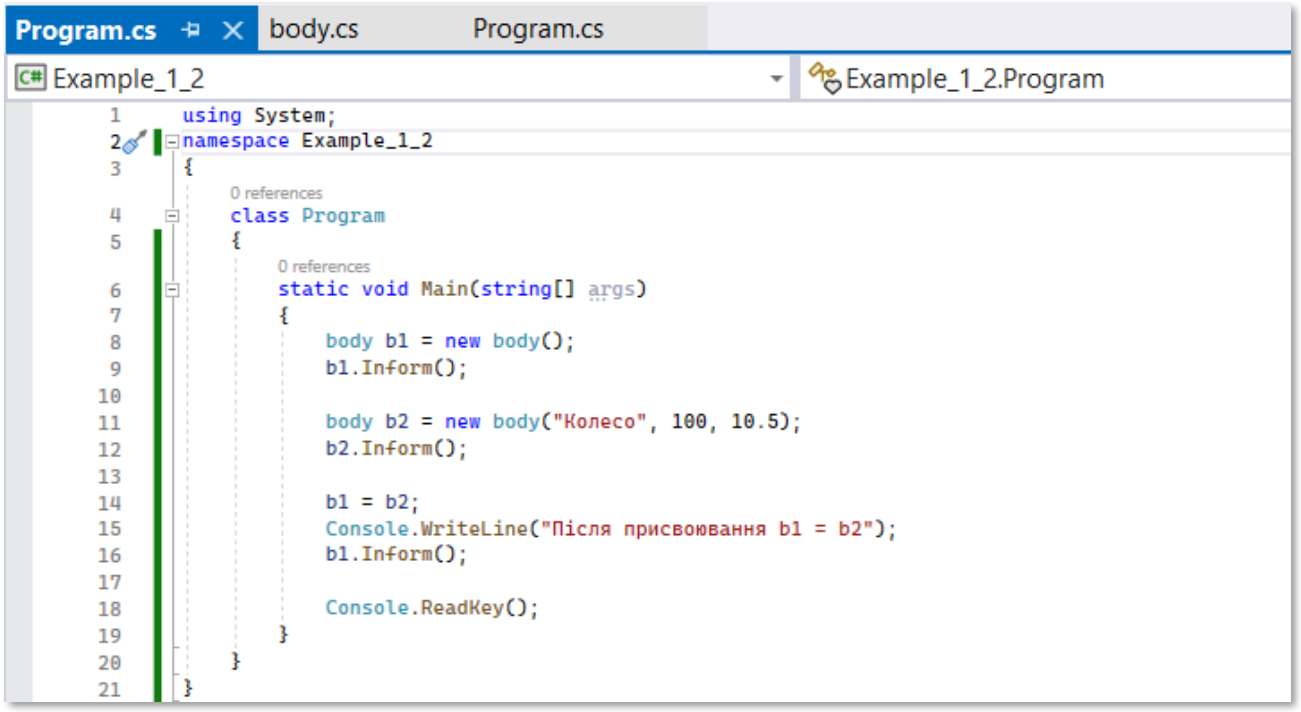
```
int x = 5;
int y = x;
```

L-операнд отримує копію значення **r**-операнду. При виконанні аналогічної (здавалося б) операції присвоювання між двома змінними посилального типу ситуація ускладнюється.

Приклад 1. Розглянемо приклад, де змінній **b1** присвоюється значення змінної **b2**, але оскільки змінна **b2** є посиланням на область пам'яті, тобто фактично містить адресу області пам'яті, то змінній **b1** присвоюється просто копія адреси, тобто вона тепер посилається на ту ж область пам'яті, що і змінна **b2**. Після запуску отримаємо такий результат:

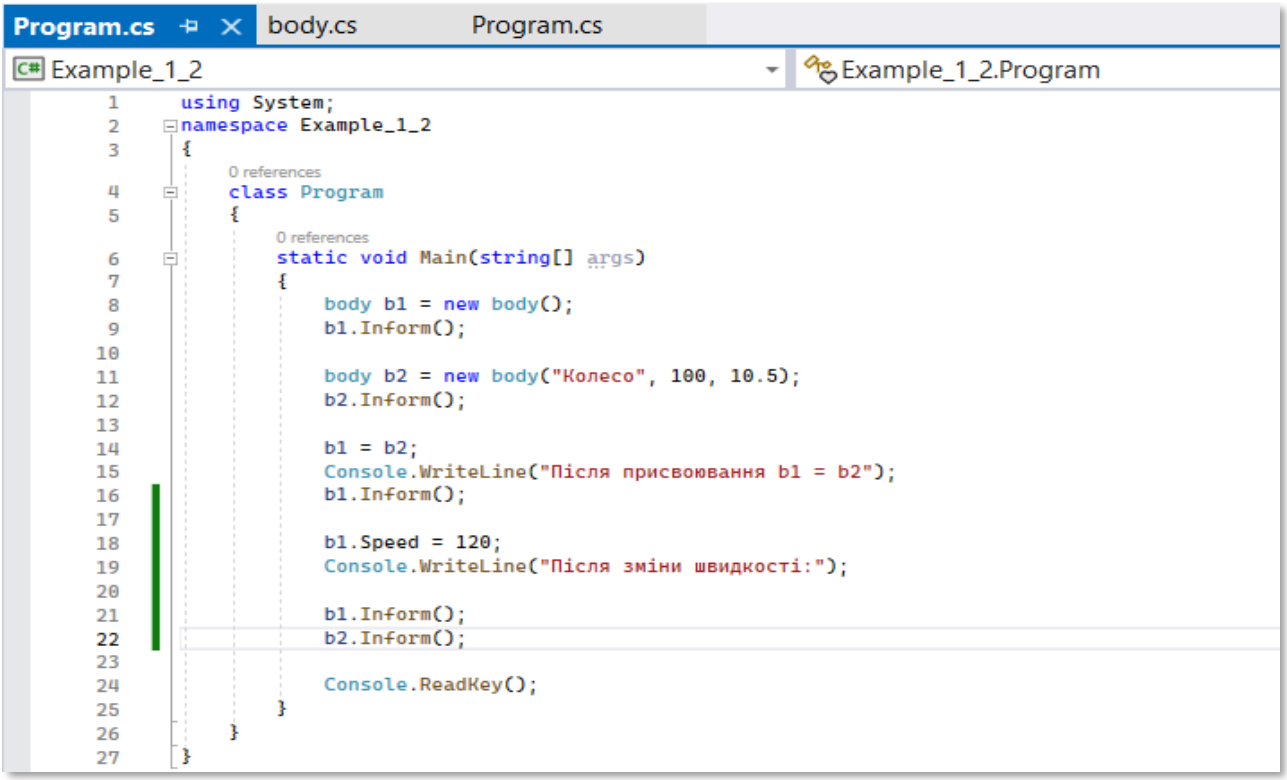


Код має такий вигляд:



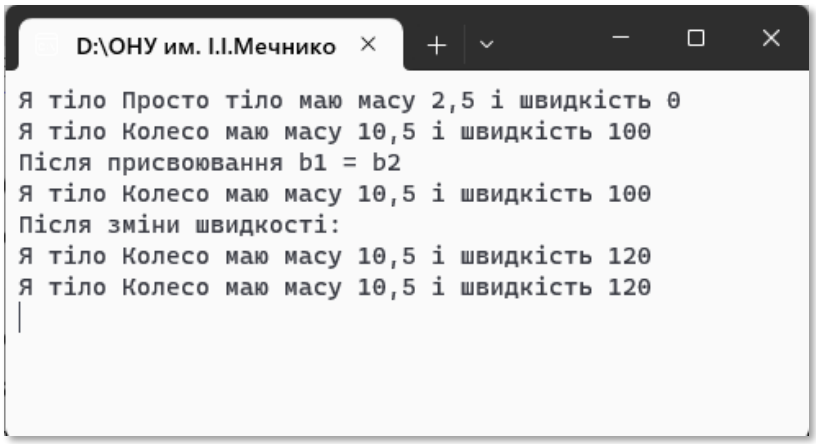

```
using System;
namespace Example_1_2
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            body b1 = new body();
            b1.Inform();

            body b2 = new body("Колесо", 100, 10.5);
            b2.Inform();

            b1 = b2;
            Console.WriteLine("Після присвоювання b1 = b2");
            b1.Inform();

            Console.ReadKey();
        }
    }
}
```

Трохи змінимо код в попередньому прикладі і додамо наступні рядки:




```
using System;
namespace Example_1_2
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            body b1 = new body();
            b1.Inform();

            body b2 = new body("Колесо", 100, 10.5);
            b2.Inform();

            b1 = b2;
            Console.WriteLine("Після присвоювання b1 = b2");
            b1.Inform();

            b1.Speed = 120;
            Console.WriteLine("Після зміни швидкості:");

            b1.Inform();
            b2.Inform();

            Console.ReadKey();
        }
    }
}
```

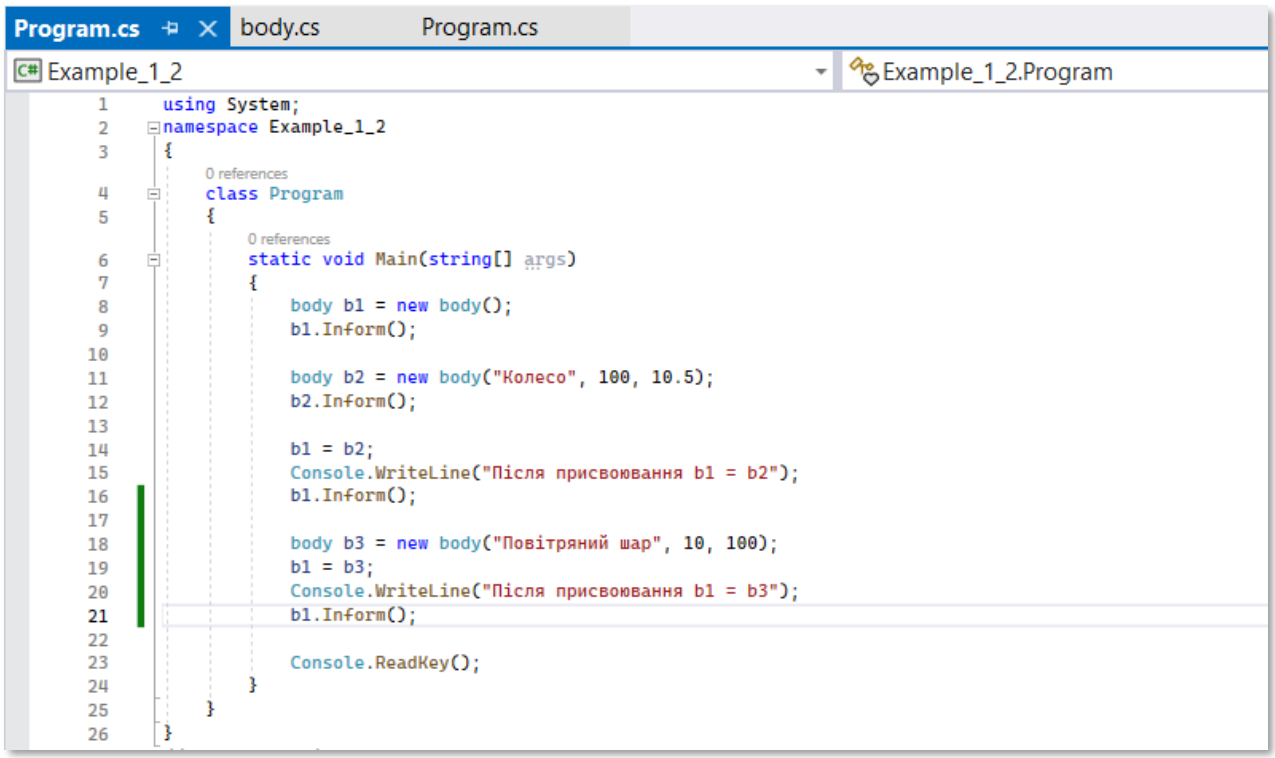
Результат:



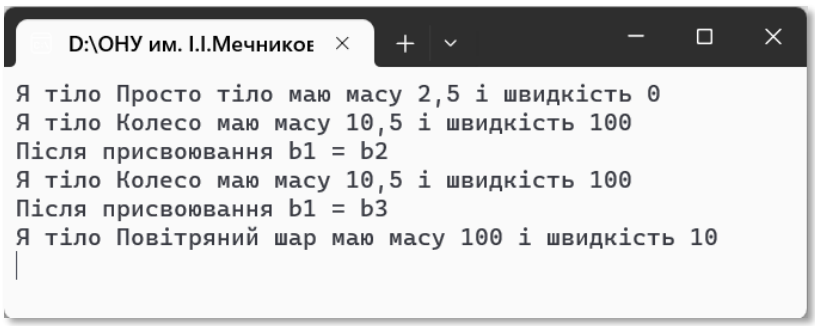
Виходить, що дві змінні представляють один і той самий об'єкт класу **body**, який розташовується в кучі за вказаною адресою.

Крім того, можна ввести в розгляд третю змінну.

Приклад 2



Результат:



Бачимо, що посилання **b1** легко зв'язується з пам'яттю, що належить екземпляру класу **b3**, та при цьому в програмі немає змінної, яка б вказувала на первісну область пам'яті, виділену під екземпляр **b1**.

3.2 Використання екземплярів класу в якості вхідних параметрів і значень, що повертаються

Екземпляри класів можуть передаватися у функції в якості вхідних параметрів. Розглянемо приклад програми, в якому використовується статична функція класу **Program**, яка в якості вхідного параметра використовує об'єкт класу **body**.

Приклад 3

Program.cs | body.cs | Program.cs

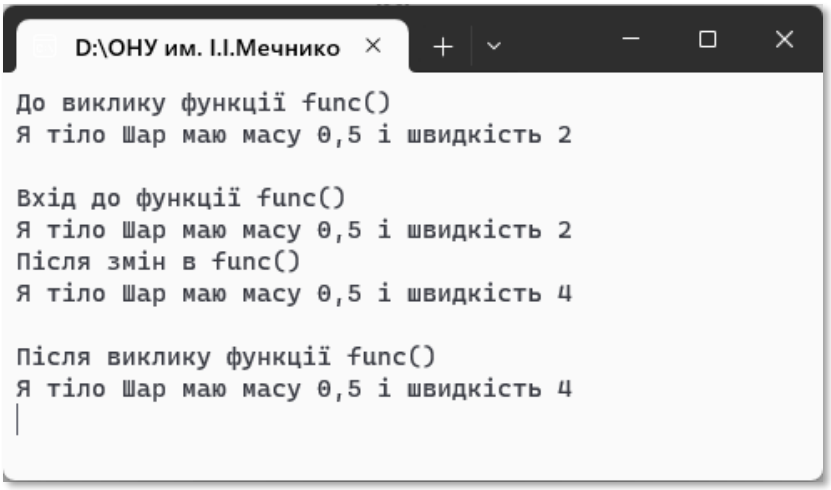
Example_1_2 | Example_1_2.Program

```
1   using System;
2   namespace Example_1_2
3   {
        0 references
4       class Program
5       {
            1 reference
6           static void func(body myt)
7           {
8               Console.WriteLine("\nВхід до функції func()");
9               myt.Inform();
10              myt.Speed *= 2;
11              Console.WriteLine("Після змін в func()");
12              myt.Inform();
13          }
            0 references
14          static void Main(string[] args)
15          {
16
17              body b1 = new body("Шар", 2, 0.5);
18              Console.WriteLine("До виклику функції func()");
19              b1.Inform();
20              func(b1);
21              Console.WriteLine("\nПісля виклику функції func()");
22              b1.Inform();
23
24              Console.ReadKey();
25          }
26
27      }
28  }
```

Зауваження 1. *З даного прикладу видно, що об'єкти передаються по посиланню.*

Об'єкти можна передавати не тільки у функціях-членах інших класів, а й у власних. Наприклад, часто використовується конструктор із вхідним параметром типу власного класу.

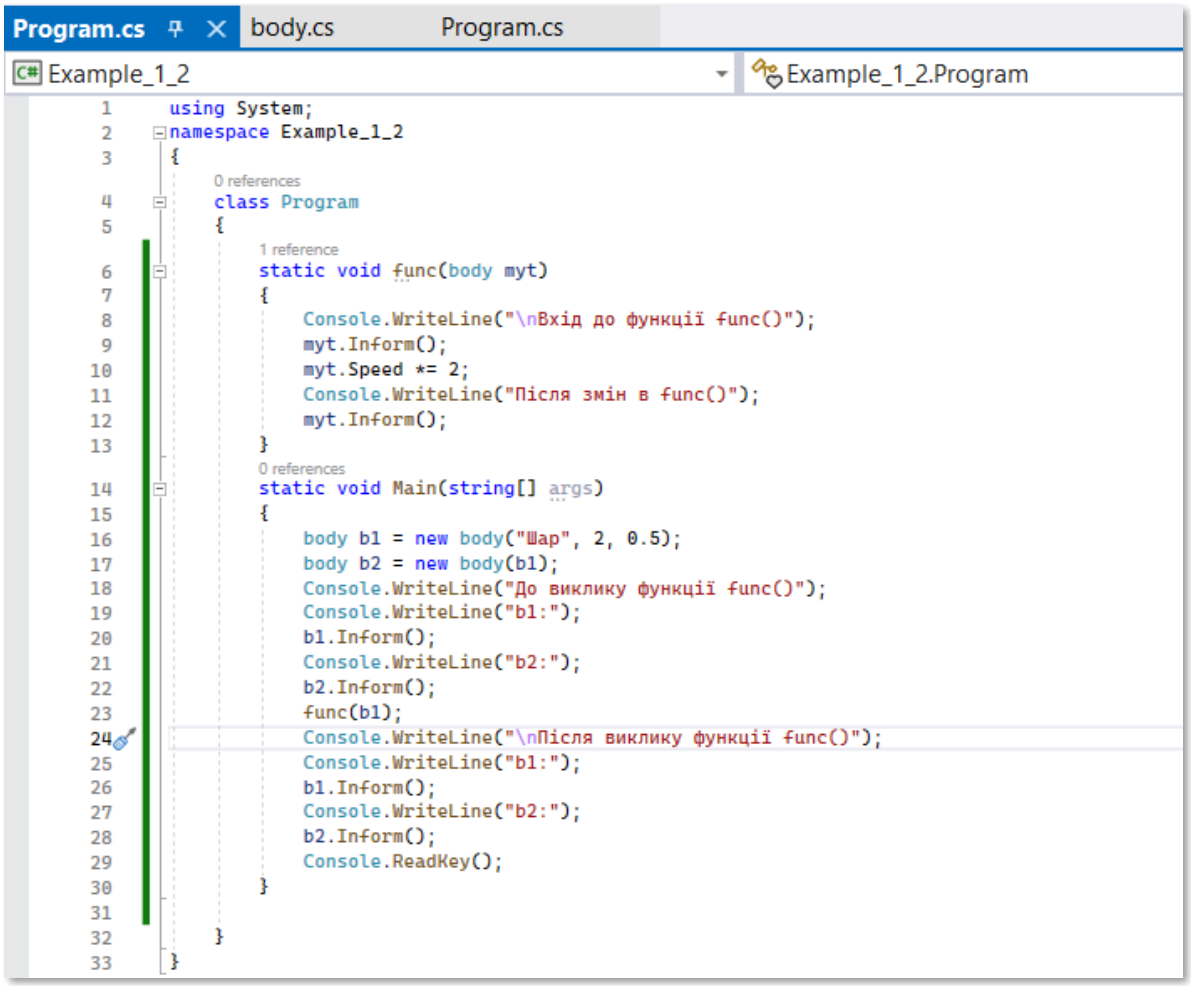
Результат:



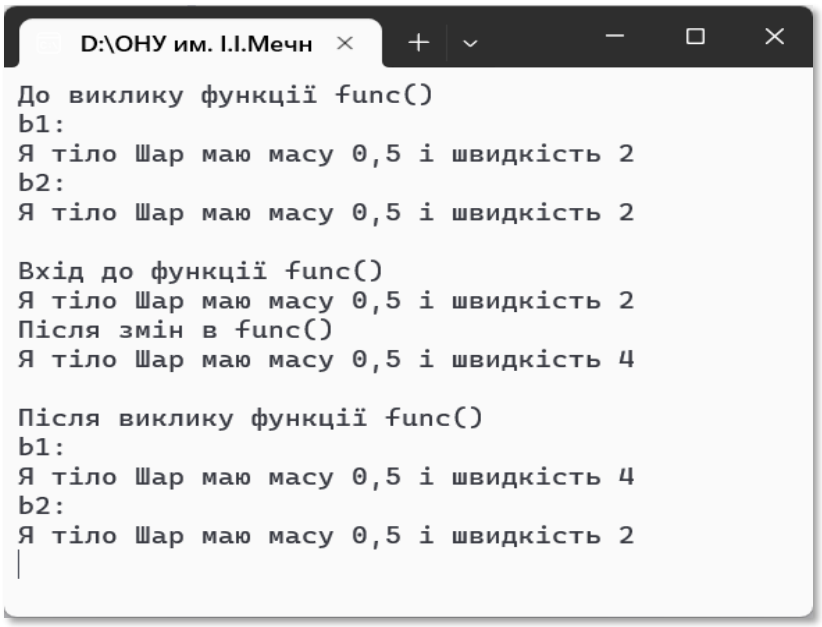
Додамо в код класу **body** конструктор виду:

```
public body(body my)
        {
            this.speed = my.speed;
            this.mass = my.mass;
            this.name = my.name;
        }
```

Приклад 4



Результат:



Зауваження 1. *Видно, що в тілі функції* ***Main*** *є два примірника класу* ***body*** *з однаковим станом.*

Зауваження 2*. У функцію* ***func*** *передається примірник* ***b1*** *по посиланню, тому його швидкість збільшується в два рази.*

Зауваження 3*. Після виконання функції* ***func****, екземпляри* ***b1*** *і* ***b2*** *мають різну швидкість, що показує нам: ці екземпляри розташовуються в різних областях кучі, тобто мають різну адресу.*

Розглянемо приклад функції, яка в якості значення, що повертається, використовує тип **body**. Під час створення об'єкта **b2** в функції **DoubleBody()**, резервується пам'ять, і адреса цієї виділеної пам'яті повертається як значення, яке присвоюється змінній **b2**. Іншими словами, змінна **b2** тепер містить вказівник на область пам'яті, яка виділена для об'єкта **bb** у функції **DoubleBody()**. Це дозволяє легко створювати та взаємодіяти з об'єктами, що знаходяться в динамічній пам'яті, зберігаючи їх адреси у змінних посилань.

Приклад 5

```
Program.cs | body.cs | Program.cs
Example_1_2 | Example_1_2.Program
```

```
1   using System;
2   namespace Example_1_2
3   {
        0 references
4       class Program
5       {
            0 references
6           static void func(body myt)...
            1 reference
14          static body DoubleBody(body my)
15          {
16              body bb = new body(my.Name + my.Name, my.Speed * 2, my.Mass * 2);
17              return bb;
18          }
            0 references
19          static void Main(string[] args)
20          {
21              body b1 = new body("Шар", 2, 0.5);
22              body b2 = DoubleBody(b1);
23              Console.WriteLine("b1:");
24              b1.Inform();
25              Console.WriteLine("b2:");
26              b2.Inform();
27              b1.Speed = 129;
28              Console.WriteLine("b1:");
29              b1.Inform();
30              Console.WriteLine("b2:");
31              b2.Inform();
32              Console.ReadKey();
33          }
34      }
35  }
```

Результат:

```
D:\ОНУ им. І.І.Мечни
Я тіло Шар маю масу 0,5 і швидкість 2
b2:
Я тіло ШарШар маю масу 1 і швидкість 4
b1:
Я тіло Шар маю масу 0,5 і швидкість 129
b2:
Я тіло ШарШар маю масу 1 і швидкість 4
```

Додамо в код попереднього прикладу в клас **Program** функцію **change().** Загальна ідея функції **change()** полягає в тому, що вона приймає існуючий об'єкт класу **body**, а потім створює новий об'єкт цього ж класу, який вказує на іншу область пам'яті в кучі. Тобто виділяється нова область пам'яті в кучі та її адреса записується в змінну **myt**

Лістинг

Program.cs | body.cs | Program.cs

Example_1_2 | Example_1_2.Program

```
using System;
namespace Example_1_2
{
    0 references
    class Program
    {
        0 references
        static void func(body myt)...
        0 references
        static body DoubleBody(body my)...
        1 reference
        static void change(body myt)
        {
            Console.WriteLine("\nВхід в функцію change()");
            myt.Inform();
            myt = new body("Мене побудували в  функції change", 100, 10);
            Console.WriteLine("Після змін в change())");
            myt.Inform();
        }

        0 references
        static void Main(string[] args)
        {
            body b1 = new body("Шар", 2, 0.5);
            Console.WriteLine("До виклику функції change()");
            b1.Inform();
            change(b1);
            Console.WriteLine("\nПісля виклику функції change()");
            b1.Inform();
            Console.ReadKey();

        }
    }
}
```

Результат:

D:\ОНУ им. І.І.Мечникова

```
До виклику функції change()
Я тіло Шар маю масу 0,5 і швидкість 2

Вхід в функцію change()
Я тіло Шар маю масу 0,5 і швидкість 2
Після змін в change())
Я тіло Мене побудували в  функції change маю масу 10 і швидкість 100

Після виклику функції change()
Я тіло Шар маю масу 0,5 і швидкість 2
```

Результат вказує на те, що у функції **Main()** створюється об'єкт **b1**, і для нього виділяється область пам'яті в кучі. В цю область пам'яті записується стан об'єкта, включаючи **"Шар"**, масу **0.5** і швидкість **2**. Об'єкт **b1** передається в функцію **change()**, де виділяється ще одна область пам'яті, і ця область пам'яті стає доступною під іменем **myt**. Вивід на екран показує, що **myt** має той самий стан, що і **b1**. Потім для змінної типу посилання виділяється інша область пам'яті з іншим станом: **"Мене побудували в функції change()",** масою **10** і швидкістю **100**. Після виходу з тіла функції **change()**, виводячи стан об'єкта **b1**, ми бачимо, що він залишився незмінним: **"Шар", маса 0.5 і швидкість 2**. Це

вказує на те, що зміни, внесені до об'єкта внутрішньо функцією **change()**, не вплинули на об'єкт **b1**, оскільки вони стосувалися іншої області пам'яті.

Питання: Поясніть результат роботи прикладу?

3.3 Методи класів, передача параметрів, повернення значень

Розглянемо клас плоских трикутників, який задається за допомогою координатного способу, тобто кожна вершина трикутника – це точка з відомими координатами. Спочатку побудуємо клас точки на площині:

Приклад 7

Point.cs* | Program.cs | body.cs | Program.cs

Example_1_3 | Example_1_3.Point

```
1
2   namespace Example_1_3
3   {
        15 references
4       class Point
5       {
            7 references
6           public double x { get; set; }
            7 references
7           public double y { get; set; }
            5 references
8           public Point(double x, double y)
9           {
10              this.x = x;
11              this.y = y;
12          }
            2 references
13          public Point()
14              : this(0, 0)
15          { }
16      }
17  }
```

Клас **Point** містить дві властивості: координати x і y. Тут використовуються автоматично реалізовані властивості. Це більш лаконічна форма властивостей, їх є сенс використовувати, коли в методах доступу (**get** і **set**) не потрібна додаткова логіка. При створенні автоматично реалізованих властивостей, компілятор створить закрите, анонімне резервне поле, яке буде доступно за допомогою методів **get** і **set**. Крім того, в класі є два конструктори: без вхідних параметрів і з двома вхідними параметрами. За замовчуванням координати точки вважаються нульовими.

Клас трикутника **Triangle** повинен містити три точки, які реалізовані за допомогою відкритих автоматично реалізуються властивостей.

```
class Triangle
{
        public Point A { get; set;}
        public Point B { get; set;}
        public Point C { get; set;}
…
}
```

Для зручності внутрішньої реалізації класу є три закритих поля для сторін трикутника і відповідні їм властивості. Тут не можна використовувати автоматично реалізовані властивості, так як користувач може задавати тільки координати вершин, а сторони обчислюються і можуть бути доступні користувачеві тільки для читання. Якщо використовувати автоматично реалізовану властивість тільки лише з аксессором ***get***, то ми не маємо можливість в коді класу обчислювати довжини сторін і змінювати значення полів, оскільки

доступ до анонімного полю є тільки через мутатор **set**.

```
class Triangle
{   …
        double fAB, fBC, fAC;

        public double AB
        {
            get { return fAB; }
        }
        public double BC
        {
            get { return fBC; }
        }
        public double AC
        {
            get { return fAC; }
        }
    …
}
```

З курсу планіметрії Ви знаєте, що не будь-які три точки будуть задавати Вам фігуру трикутник, можна провести перевірку за допомогою нерівності трикутника і заповнити результат перевірки в поле **reality** булєвого типу, яке відіграє роль прапорця. Цей прапорець необхідний для реалізації класу, щоб уникнути помилок при обчисленні площі трикутника, периметра, радіусів кіл, вписаних в трикутник або описаних навколо трикутника.

```
class Triangle
{ …
     bool reality;
    public bool Reality
       {  get { return reality; } }
  …
}
```

Клас буде містити перевантажені конструктори без вхідних параметрів і з параметрами, які задають координати вершин трикутника на площині.

```
public Triangle(Point A, Point B, Point C)
{
    this.A = A;
    this.B = B;
    this.C = C;

 fAB = Math.Sqrt(Math.Pow(A.x - B.x, 2) + Math.Pow(A.y - B.y, 2));
 fBC = Math.Sqrt(Math.Pow(C.x - B.x, 2) + Math.Pow(C.y - B.y, 2));
 fAC = Math.Sqrt(Math.Pow(C.x - A.x, 2) + Math.Pow(C.y - A.y, 2));

 reality=(AB >= AC + BC || BC >= AB + AC || AC >= AB +   BC)?false:true;
}
public Triangle(): this(new Point(),new Point(0,3),new Point(4, 0))
{  }
```

За замовчуванням будується єгипетський трикутник в першій чверті, де вершина прямого кута збігається з початком відліку системи координат.

Додамо до класу різні методи, тобто наділимо наші об'єкти поведінкою. Методи класу можуть повертати значення різного типу, наприклад, функції визначення кутів трикутника.

```
private double Alfa()
{ return Math.Acos(0.5 * (AB * AB + AC * AC - BC * BC) / AB / AC) * 180 /
Math.PI; }

private double Beta()
{ return Math.Acos(0.5 * (AB * AB + BC * BC - AC * AC) / AB / BC) * 180 /
Math.PI; }

private double Gamma()
{ return Math.Acos(0.5 * (AC * AC + BC * BC - AB * AB) /AC / BC) * 180 /
Math.PI; }
```

Функція знаходження периметра трикутника:

```
public double P()
        {
            double p = 0;
            if (reality)
                p = AB + BC + AC;
            return p;
        }
```

Функції-члени класу можуть бути перевантажені:

```
public double S()
        {
            double s=0;
            if (reality)
            {
                double p = P() / 2;
                s=Math.Sqrt(p * (p - AB) * (p - BC) * (p - AC));
            }
```

```
            return s;  }

public double S(bool b)
        {
            double s = 0;
            if (reality)
                s = 0.5 * AB * AC * Math.Sin(Alfa() * Math.PI / 180);
            return s;
        }
```

Функції-члени можуть приймати вхідні параметри, як за значенням, так і за посиланням.

```
public void Angles(out double al, out double bet, out double gam)
        {
            al = 0; bet = 0; gam = 0;
            if (reality)
            {
                al = Alfa();
                bet = Beta();
                gam = Gamma();
            }
        }

public void Radius(ref double r, ref double R)
        {
            r = 0; R = 0;
            if (!reality) return;
            double p = P() / 2;
            double s = S(true);
            r = Math.Sqrt(s / p);
            R = 0.25 * AB * BC * AC / s;
        }
```

Функція-член описує трикутник:

```
public List<string> Inform()
        {
            List<string> report = new List<string>();
            report.Add("Сторона AB=" +Convert.ToString(AB));
            report.Add("Сторона AC=" + Convert.ToString(AC));
            report.Add("Сторона BC=" + Convert.ToString(BC));
            report.Add("Кут A=" + String.Format("{0:F0}",Alfa()));
            report.Add("Кут B=" + String.Format("{0:F0}",Beta()));
            report.Add("Кут C=" + String.Format("{0:F0}", Gamma()));
            report.Add("P=" + Convert.ToString(P()));
            report.Add("S=" + Convert.ToString(S(true)));
            double r=0;
            double R=0;
            Radius(ref r,ref R);
            report.Add("r=" + Convert.ToString(r));
            report.Add("R=" + Convert.ToString(R));
            return report;
        }
```

Остаточний вигляд класу **Triangle**:

Triangle.cs* | Point.cs* | Program.cs | body.cs | Program.cs

Example_1_3 | Example_1_3.Triangle

```
4   namespace Example_1_3
5   {
        7 references
6       class Triangle
7       {
            1 reference
8           public Point A {get;set;}
            1 reference
9           public Point B { get; set;}
            1 reference
10          public Point C { get; set;}
11
12
13          double fAB, fBC, fAC;
14          bool reality;
15
            16 references
16          public double AB
17          {
18              get { return fAB; }
19          }
            15 references
20          public double BC
21          {
22              get { return fBC; }
23          }
            16 references
24          public double AC
25          {
26              get { return fAC; }
27          }
            0 references
28          public bool Reality
29          {
30              get { return reality; }
31          }
            2 references
32          public Triangle(Point A, Point B, Point C)
33          {
34              this.A = A;
35              this.B = B;
36              this.C = C;
37              fAB = Math.Sqrt(Math.Pow(A.x - B.x, 2) + Math.Pow(A.y - B.y, 2));
38              fBC = Math.Sqrt(Math.Pow(C.x - B.x, 2) + Math.Pow(C.y - B.y, 2));
39              fAC = Math.Sqrt(Math.Pow(C.x - A.x, 2) + Math.Pow(C.y - A.y, 2));
40              reality=(AB >= AC + BC || BC >= AB + AC || AC >= AB + BC)?false:true;
41          }
            1 reference
42          public Triangle(): this(new Point(),new Point(0,3),new Point(4, 0))
43          {
44          }
```

```
46          private double Alfa()
47          { return Math.Acos(0.5 * (AB * AB + AC * AC - BC * BC) / AB / AC) * 180 / Math.PI; }
48
            2 references
49          private double Beta()
50          { return Math.Acos(0.5 * (AB * AB + BC * BC - AC * AC) / AB / BC) * 180 / Math.PI; }
51
            2 references
52          private double Gamma()
53          { return Math.Acos(0.5 * (AC * AC + BC * BC - AB * AB) /AC / BC) * 180 / Math.PI; }
54
55
            3 references
56          public double P()
57          {
58              double p = 0;
59              if (reality)
60                  p = AB + BC + AC;
61              return p;
62          }
```

```
63          public double S()
64          {
65              double s=0;
66              if (reality)
67              {
68                  double p = P() / 2;
69                  s=Math.Sqrt(p * (p - AB) * (p - BC) * (p - AC));
70              }
71              return s;
72          }
73
            2 references
74          public double S(bool b)
75          {
76              double s = 0;
77              if (reality)
78                  s = 0.5 * AB * AC * Math.Sin(Alfa() * Math.PI / 180);
79              return s;
80          }
81
            0 references
82          public void Angles(out double al, out double bet, out double gam)
83          {
84              al = 0; bet = 0; gam = 0;
85              if (reality)
86              {
87                  al = Alfa();
88                  bet = Beta();
89                  gam = Gamma();
90              }
91          }
92
            1 reference
93          public void Radius(ref double r, ref double R)
94          {
95              r = 0; R = 0;
96              if (!reality) return;
97              double p = P() / 2;
98              double s = S(true);
99              r = Math.Sqrt(s / p);
100             R = 0.25 * AB * BC * AC / s;
101         }
```

```
103         public List<string> Inform()
104         {
105             List<string> report = new List<string>();
106             report.Add("Сторона AB=" +Convert.ToString(AB));
107             report.Add("Сторона AC=" + Convert.ToString(AC));
108             report.Add("Сторона BC=" + Convert.ToString(BC));
109             report.Add("Угол A=" + String.Format("{0:F0}",Alfa()));
110             report.Add("Угол B=" + String.Format("{0:F0}",Beta()));
111             report.Add("Угол C=" + String.Format("{0:F0}", Gamma()));
112             report.Add("P=" + Convert.ToString(P()));
113             report.Add("S=" + Convert.ToString(S(true)));
114             double r=0;
115             double R=0;
116             Radius(ref r,ref R);
117             report.Add("r=" + Convert.ToString(r));
118             report.Add("R=" + Convert.ToString(R));
119             return report;
120         }
121
122     }
123 }
```

Давайте розглянемо приклад використання цього класу. Для цього в класі **Program** в функції **Main** додамо наступний код:

Program.cs | Triangle.cs* | Point.cs* | Program.cs | body.cs | Program.cs

Example_1_3 | Example_1_3.Program

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace Example_1_3
{
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Triangle t1 = new Triangle();
            List<string> Result = t1.Inform();
            foreach (string s in Result)
                Console.WriteLine(s);

            Console.WriteLine();

            Triangle t2 = new Triangle(new Point(),new Point(3,3*Math.Sqrt(3.0)),new Point(6,0));
            Result = t2.Inform();
            foreach (string s in Result)
                Console.WriteLine(s);

            Console.ReadKey();
        }
    }
}
```

Результат:

D:\ОНУ им. І.І.Мечн

```
Сторона AB=3
Сторона AC=4
Сторона BC=5
Угол A=90
Угол B=53
Угол C=37
P=12
S=6
r=1
R=2,5

Сторона AB=6
Сторона AC=6
Сторона BC=6
Угол A=60
Угол B=60
Угол C=60
P=18
S=15,5884572681199
r=1,31607401295249
R=3,46410161513775
```

4. ЧАСТКОВІ КЛАСИ

4.1 Ключове слово operator

У C # реалізована можливість розділити створення класу або методи між двома або більше вихідними файлами або модулями. Кожен файл містить визначення типу або методу, і всі частини об'єднуються при компіляції програми.

Для поділу класу на кілька частин, використовується ключове слово **partial**. Поділ класу на частини є доцільним коли:

1. клас розробляється командою програмістів. Кожен програміст може бути відповідальним за реалізацію конкретної частини класу. Цей підхід полегшує спільну роботу над великими кодовими базами та дозволяє впроваджувати розділення праці;
2. клас розбивається на кілька частин через свою функціональності. Розбиття класу на частини, які відповідають за конкретні аспекти функціональності, сприяє полегшенню розуміння коду, його обслуговуванню та вдосконаленню. Такий підхід використовується при створенні Windows-додатків, які ми будемо з Вами вивчати пізніше в цьому семестрі.

Приклад 1

PartialClass_1.cs* | Program.cs

Example_1_4 | Example_1_4.PartialClass

```
1  using System;
2  namespace Example_1_4
3  {
4      //Перша частина класу
       3 references
5      partial class PartialClass
6      {
           0 references
7          public void MethodFromPart1()
8          {  Console.WriteLine("I Part"); }
9      }
10 }
```

PartialClass_2.cs* | PartialClass_1.cs* | Program.cs

Example_1_4 | Example_1_4.PartialClass

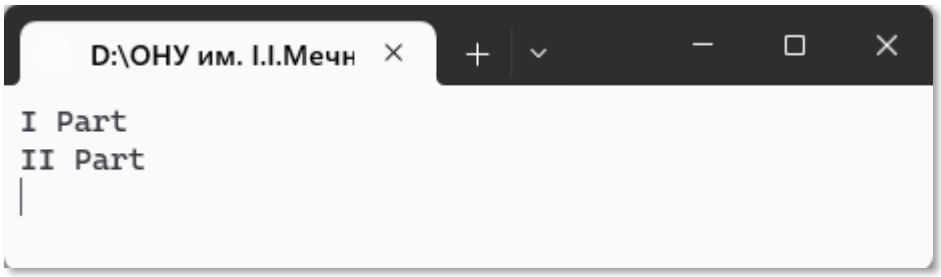
```
1  using System;
2  namespace Example_1_4
3  {
4      //Друга частина класу
       3 references
5      partial class PartialClass
6      {
           0 references
7          public void MethodFromPart2()
8          {  Console.WriteLine("II Part"); }
9      }
10 }
```

Program.cs* | PartialClass_2.cs* | PartialClass_1.cs*

Example_1_4 | Example_1_4.Program

```
1   using System;
2   namespace Example_1_4
3   {
        0 references
4       class Program
5       {
            0 references
6           static void Main(string[] args)
7           {
8               PartialClass my=new PartialClass();
9               my.MethodFromPart1();
10              my.MethodFromPart2();
11              Console.ReadKey();
12          }
13      }
14  }
```

Результат:



4.2 Часткові методи

Часткові методи – це методи, де «прототип» або сигнатура методу визначена при створенні часткового класу, а реалізація виконується в будь-який інший (тільки одній) частині цього класу.

Правила використання часткових методів:

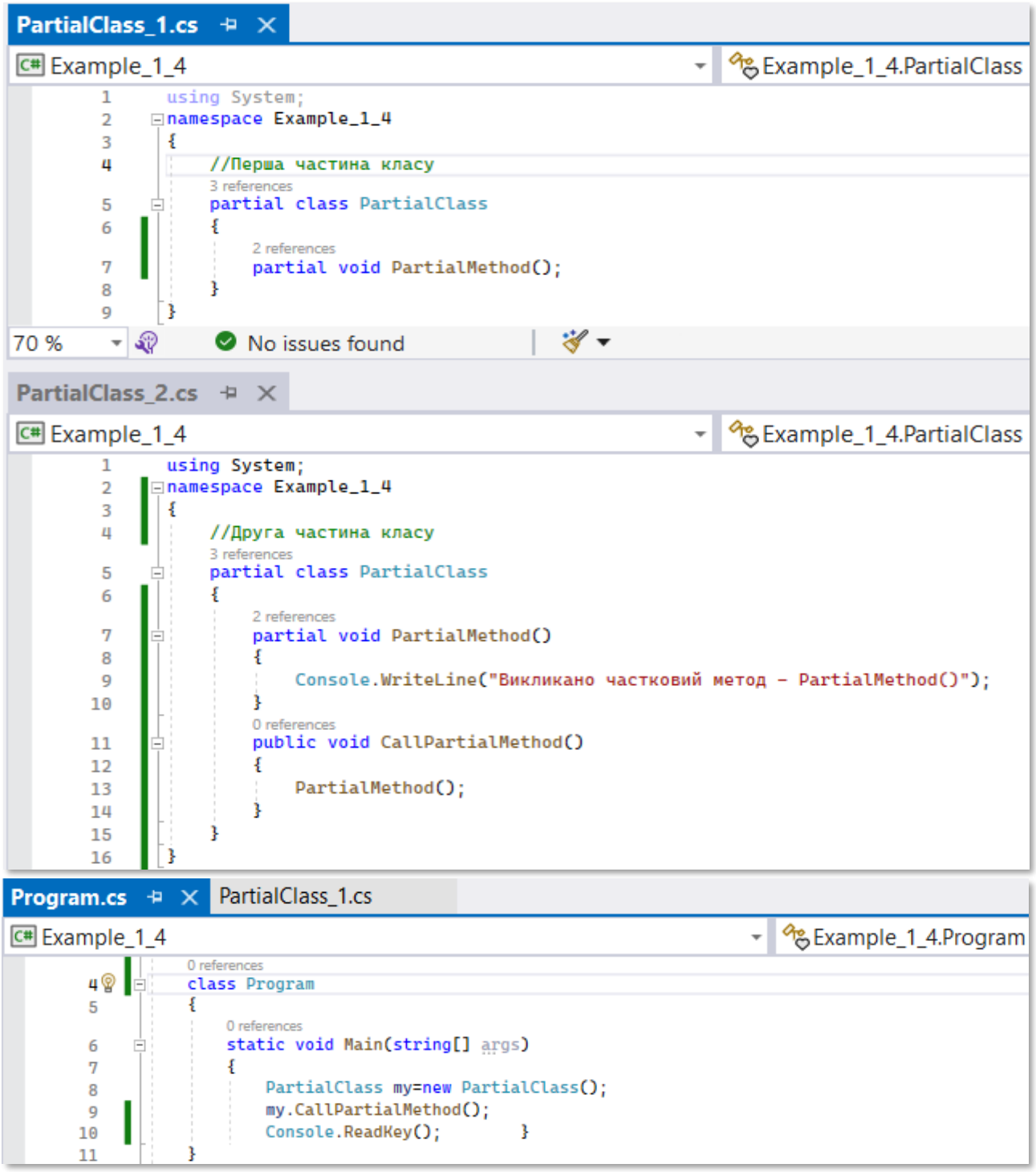
1. Часткові методи повинні бути визначені тільки в часткових класах.
2. Часткові методи повинні бути позначені ключовим словом **partial**.
3. Часткові методи завжди є **private** (за замовчуванням), спроба явного використання з ними модифікатора доступу призведе до помилки.
4. Часткові методи повинні повертати **void**.
5. Часткові методи можуть бути нереалізованими.

Приклад 2

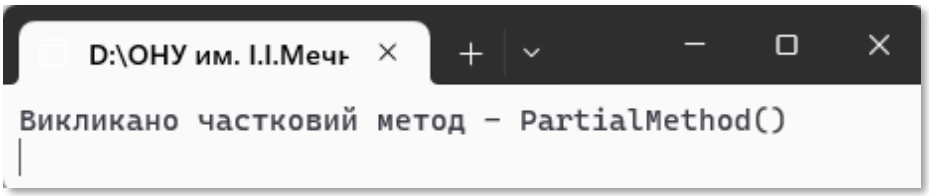
```
//Перша частина класу
partial class PartialClass
    {
        …
        partial void PartialMethod();
        …
    }
```

```
//Друга частина класу
partial class PartialClass
     {…
          partial void PartialMethod()
          {     Console.WriteLine("PartialMethod()");     }
     …}
```

Але якщо необхідно користувачеві Вашого класу надати можливість виклику часткового методу (так як він закритий, то користувач не може його безпосередньо викликати), необхідно в класі оголосити відкритий метод, в тілі якого здійснюється виклик часткового методу.



Результат:



Приклад №3

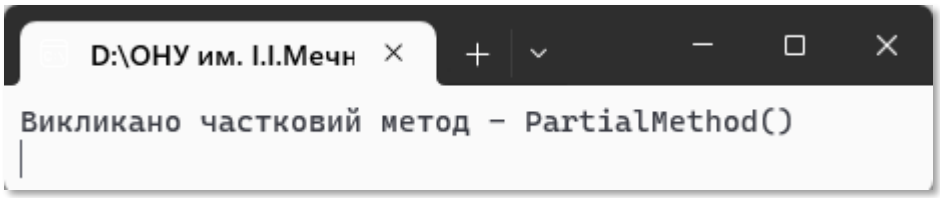
Додамо в першу частину класу ще один частковий метод, що не будемо ніде реалізовувати:

```
//Перша частина класу
partial class PartialClass
    {…
        partial void MyMethod();
     … }
```

Додамо його виклик у другій частині класу:

```
//Друга частина класу
partial class PartialClass
    {
        public void CallPartialMethod()
        {
            PartialMethod();
            MyMethod();
        }
    }
```

Виконання програми відбувається без помилок і результат буде такий же, як і в попередньому прикладі



4.3 Використання модифікатора типу static

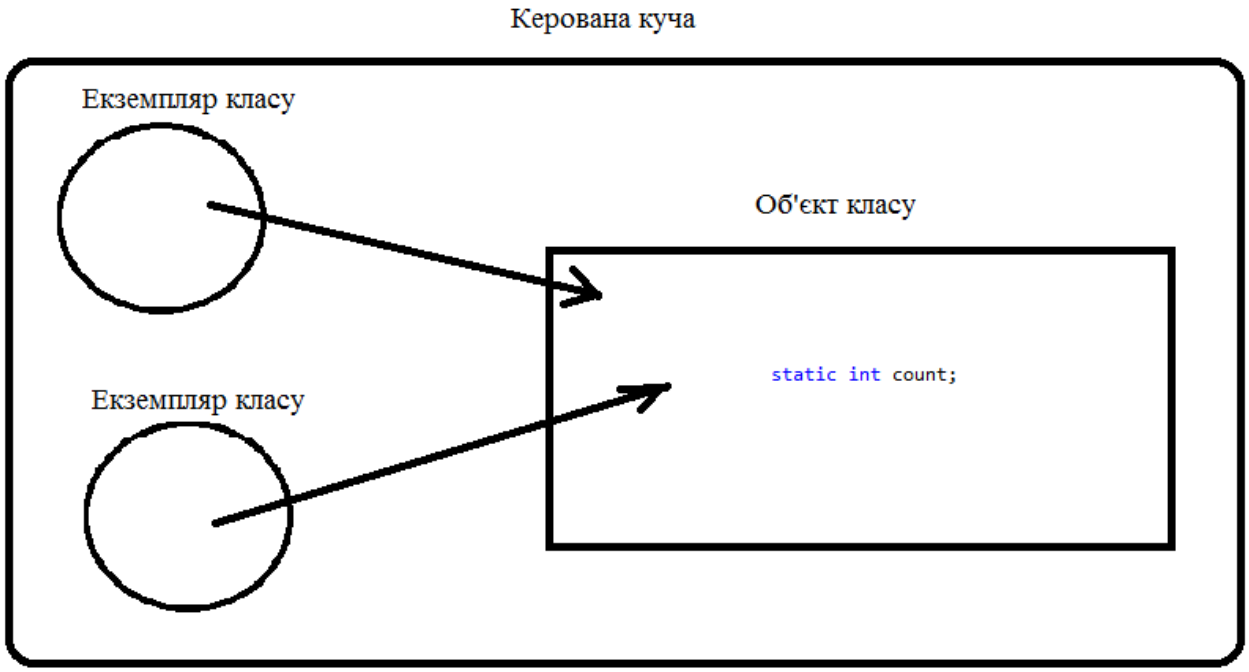
Іноді потрібно визначити член класу, який повинен використовуватися незалежно від екземпляра цього класу. Зазвичай до члена класу доступ надається через екземпляр цього класу. Однак можна створити член, який свідомо дозволено використовувати сам по собі, тобто без посилання на конкретний екземпляр. Щоб створити такий член, передуватимете його оголошення ключовим словом **static**.

При використанні **static**-члена поза класом необхідно вказати ім'я класу і наступний за ним оператор прямого доступу (**.**). Примірник класу при цьому не

потрібно створювати. Наприклад, щоб привласнити число 10 **static**-змінної з ім'ям **count**, яка є членом класу **Timer**, використовуйте наступний рядок коду:

```
Timer.count = 10; // вимовляється: на класі-об'єкті звертаємося до статичного полю
```

Змінні, оголошені як **static**-члени, є по суті глобальними змінними. При оголошенні примірників класу копії **static**-змінної не створюються, причому всі екземпляри класу спільно використовують одну і ту ж **static**-змінну. Ініціалізація **static**-змінної відбувається при завантаженні класу.



Якщо ініціалізатор явно не вказано, **static**-змінна, призначена для зберігання числових значень, ініціалізирується нулем; об'єктні посилання – **null**-значеннями, а змінні типу **bool** – значенням **false**. Таким чином, **static** – змінного завжди має значення.

З використанням ключового слова **static** можна оголошувати як змінні, так і методи. Як перший приклад static-члена наведемо метод **Main()**, який повинен бути викликаний операційною системою на початку роботи програми. Методи, які ми використовували, **Console.WriteLine()** і **Convert.ToString()** також є статичними. Ні в який момент часу від нас не вимагається створювати екземпляри класів **Console** або **Convert** (більш того, навіть якби ми і спробували це зробити, то все одно нічого б не вийшло, оскільки до конструкторів цих класів не існує загального доступу).

Різниця між **static-** і звичайним методом полягає в тому, що **static**-метод можна викликати за допомогою імені класу, без необхідності створення екземпляра цього класу. Ми зверталися до **static**-методу **Sqrt()**, що належить

класу **System.Math**.

На **static**-методи накладається ряд обмежень:

1. **static**-метод не має посилання **this**.
2. **static**-метод може безпосередньо викликати тільки інші **static**-методи. Він не може безпосередньо викликати метод примірника свого класу. Справа в тому, що методи примірників працюють з конкретними екземплярами класу, чого не скажеш про **static**-методи.
3. **static**-метод повинен отримувати прямий доступ тільки до **static**-даних. Він не може безпосередньо використовувати змінні екземплярів, оскільки не працює з екземплярами класу.

Приклад 3. Розглянемо приклад використання статичної змінної, яка використовується для того, щоб підрахувати скільки об'єктів даного класу створено (зазвичай використовується при створенні веб-сторінок – скільки користувачів зараз на даній сторінці.)

my.cs

Example_1_5 | Example_1_5.my | Print(my s)

```
{
    8 references
    class my
    {
        static int i = 0;
        int a;
        1 reference
        public my(int k) { a = k; i++; }
        2 references
        static public int GetI() { return i; }
        0 references
        public int GetA() { return a; }
        1 reference
        static public void Print() { Console.WriteLine("i={0}", i); }
        1 reference
        static public void Print(my s) { Console.WriteLine("i={0}\ta={1}", i, s.a); }
    }
```

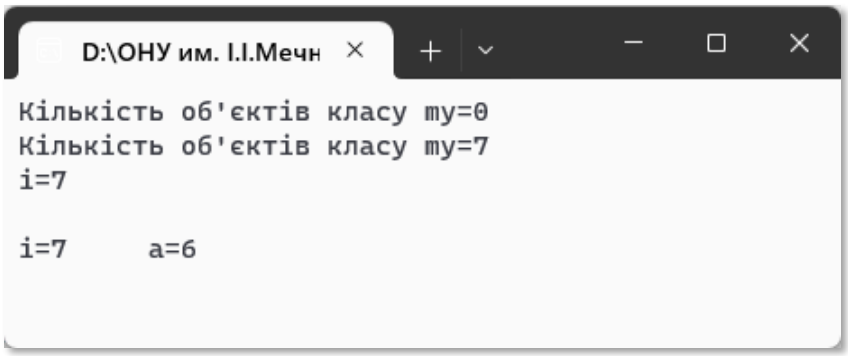
70 % | No issues found

Program.cs

Example_1_5 | Example_1_5.Program | Main(string[

```
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            Console.WriteLine("Кількість об'єктів класу my={0}", my.GetI());
            Random b = new Random();
            int n = b.Next(25);
            my f = null;
            for (int i = 0; i < n; i++)
                f = new my(b.Next(100));
            Console.WriteLine("Кількість об'єктів класу my={0}", my.GetI());
            my.Print();
            Console.WriteLine();
            my.Print(f);
            Console.ReadKey();
        }
    }
```

Результат:



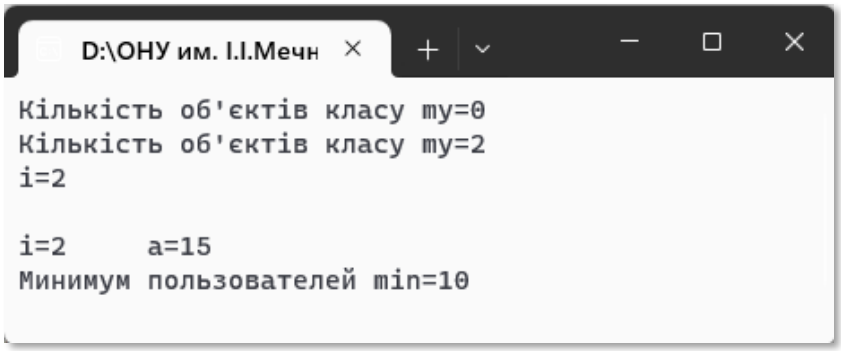
Як члени класу можна використовувати константи з використанням ключового слова **const**. Такі члени є початково статичними і використання ключового слова **static** з такими членами заборонено.

Додамо в наш клас член: `public const int min = 10;`

Виводимо інформацію на екран за допомогою команди виду:

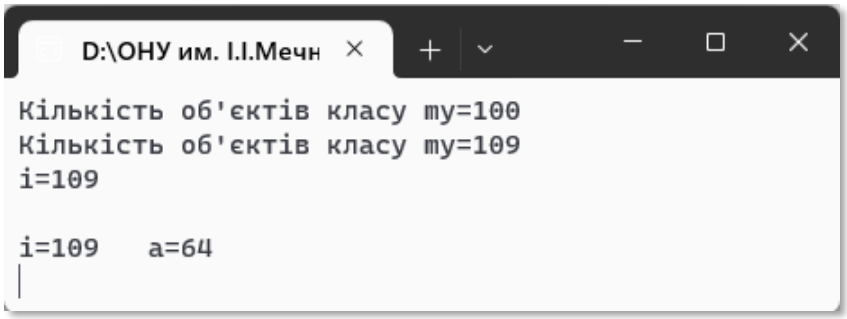
```
Console.WriteLine("Мінімум користувачів min={0}", my.min);
```

Отримаємо такий результат:



Конструктор класу також можна оголосити статичним. Статичний конструктор не має модифікаторів доступу. Статичний конструктор викликається автоматично і використовується для ініціалізації статичних даних, які застосовуються до класу в цілому, а не до конкретного його екземпляру. Таким чином, статичний конструктор служить для ініціалізації аспектів класу до створення екземплярів цього класу.

```
static my() { i = 100; }
```



4.4 Визначення статичних класів

Ключове слово **static** допускається також застосовувати прямо на рівні класу. Коли клас визначений як статичний, його екземпляри можна створювати з використанням ключового слова **new**, і він може включати в себе тільки члени або поля даних, помічені ключовим словом **static**. Якщо це правило порушити, виникне помилка компіляції.

На перший погляд це може здатися досить дивним засобом, враховуючи неможливість створення екземплярів класу. Однак слід врахувати, що клас, який не містить нічого крім статичних членів і / або константних даних, і не потребує виділення пам'яті. Ви вже користувалися таким класами: **Console**, **Convert** і **Math**.

Статичний клас не можна використовувати в якості поля, параметра методу або локальної змінної, оскільки це має на увазі існування змінної, що посилається на екземпляр, що заборонено. Виявивши подібне звернення зі статичним класом, компілятор поверне повідомлення про помилку.

Зауваження. Клас, який відкриває тільки статичну функціональність, часто називають обслуговуючим.

Приклад 5

Використання статичного класу

SimpleClass.cs

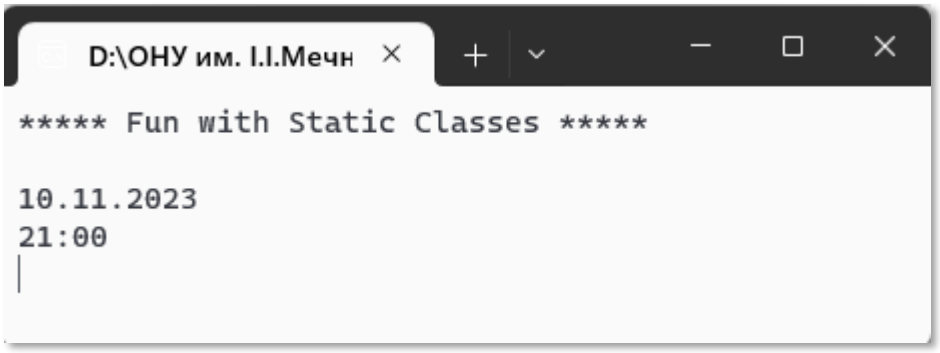
```
StaticClass | StaticClass.SimpleClass | PrintTime(
2 references
static class SimpleClass
{
    1 reference
    public static void PrintTime()
    { Console.WriteLine(DateTime.Now.ToShortTimeString()); }
    1 reference
    public static void PrintDate()
    { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

70 % No issues found

Program.cs

```
StaticClass | StaticClass.Program | Main(strin
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Static Classes *****\n");
        SimpleClass.PrintDate();
        SimpleClass.PrintTime();
        Console.ReadLine();
    }
}
```

Результат:



Питання: *Поясніть чому Ви в першому семестрі всі функції оголошували з модифікатором* ***static****?*

5. ПЕРЕВАНТАЖЕННЯ ОПЕРАТОРІВ

5.1 Ключове слово operator

Подібно будь-якій мові програмування, в C# є готовий набір лексем, що використовуються для виконання базових операцій над вбудованими типами. Наприклад, відомо, що операція + може застосовуватися до двох цілих, щоб дати їх суму:

```
int a = 100;
int b = 240;
int c = a + b;
```

Та ж операція + може застосовуватися до більшості вбудованих типів даних C#. Наприклад, розглянемо такий код:

```
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2;
```

По суті, функціональність операції + унікальним чином базується на представлених типах даних (в цьому випадку рядках або цілих числах). Коли операція + застосовується до числових типів, ми отримуємо арифметичну суму операндів. Однак коли та ж операція застосовується до строкових типів, виходить конкатенація рядків.

У C# призначені для користувача типи можуть перевантажувати оператори шляхом визначення функцій статичних членів за допомогою ключового слова **operator**. Це називається перевантаженням, оскільки ми використовуємо нашу власну реалізацію роботи цих операторів в тих випадках, коли вони застосовуються для виконання операцій над параметрами певних типів; багато в чому це аналогічно тому, як перевантажуються методи, коли

передаються різні параметри методам з одним і тим же ім'ям.

Для перевантаження оператора в призначеному для користувача класі потрібно створити метод в класі з правильною сигнатурою. Метод потрібно назвати **operator X**, де **X** – ім'я або символ оператора, що перевантажується. Унарні оператори мають один параметр, а бінарні – два. У кожному разі один параметр повинен бути такого ж типу, як клас, який оголосив оператор.

Загальний формат перевантаження для унарного оператора:

```
public static тип_значення operator операт.(тип_параметру ім’я_операнду)
    {
      //операції
    }
```

Загальний формат перевантаження для бінарного оператора:

```
public static тип_значення operator оператор(тип_параметру
ім’я_операнда1, тип_параметру ім’я_операнда2)
    {
      //операції
    }
```

Для унарних операторів тип операнда повинен збігатися з класом, для якого визначений оператор. Що стосується бінарних операторів, то тип хоча б одного операнда повинен збігатися з відповідним класом.

Якщо операнди не задовольняють правило зіставлення типів, може виникнути помилка чи неочікувана поведінка програми.

На перевантаження операторів накладається ряд обмежень:

- не можна перевантажувати оператори для класів, не призначених вами. Наприклад, ви не можете перевантажити оператор "+" для типів **int** або **string**.
- не можна змінювати пріоритет оператора.
- не можна змінювати кількість операндів, прийнятих оператором.
- параметри операторів не повинні використовувати модифікатор **ref** або **out**.
- можна перевантажувати оператори +, -, *, /,%, ++, -,!, ==,! =, <,>, <=,> =.

Приклад 1

Побудуємо клас **vector** з перевантаженими операторами +, -, ++,==, !=.

vector.cs

OverloadedOps — OverloadedOps.vector

```
{
    40 references
    public class vector
    {
        public int Length;
        private int[] a;
        5 references
        public vector(int raz)
        {
            Length = raz; a = new int[Length];
        }
        2 references
        public vector(int[] mas)
        {
            Length = mas.Length; a = new int[Length];
            for (int i = 0; i < Length; i++)
                a[i] = mas[i];
        }
        2 references
        public vector(vector t1)
        {
            Length = t1.Length; a = new int[Length];
            for (int i = 0; i < Length; i++)
                a[i] = t1.a[i];
        }
        0 references
        public int[] GetVector() { return a; }
        0 references
        public void SetVector(int[] mas)
        {
            if (mas.Length != Length) return;
            for (int i = 0; i < Length; i++)
                a[i] = mas[i];
        }
        1 reference
        public static vector operator +(vector t1, vector t2)
        {
            if (t1.Length != t2.Length) return null;
            vector rez = new vector(t1.Length);
            for (int i = 0; i < t1.Length; i++)
                rez.a[i] = t1.a[i] + t2.a[i];
            return rez;
        }
        public static vector operator +(vector t1, int t2)
        {
            vector rez = new vector(t1.Length);
            for (int i = 0; i < t1.Length; i++)
                rez.a[i] = t1.a[i] + t2;
            return rez;
        }
        1 reference
        public static vector operator +(int t2, vector t1)
        {
            vector rez = new vector(t1.Length);
            for (int i = 0; i < t1.Length; i++)
                rez.a[i] = t1.a[i] + t2;
            return rez;
        }
        1 reference
        public static vector operator -(vector t1, vector t2)
        {
            if (t1.Length != t2.Length) return null;
            vector rez = new vector(t1.Length);
            for (int i = 0; i < t1.Length; i++)
                rez.a[i] = t1.a[i] - t2.a[i];
            return rez;
        }
```

```
0 references
65      public static vector operator -(vector t1, int t2)
66      {
67          vector rez = new vector(t1.Length);
68          for (int i = 0; i < t1.Length; i++)
69              rez.a[i] = t1.a[i] - t2;
70          return rez;
71      }
        3 references
72      public static bool operator ==(vector t1, vector t2)
73      {
74          if (t1.Length != t2.Length) return false;
75          bool fl = true;
76          for (int i = 0; i < t1.Length; i++)
77              if (t1.a[i] != t2.a[i])
78              {  fl = false; break; }
79          return fl;
80      }
        2 references
81      public static bool operator !=(vector t1, vector t2)
82      {
83          if (t1.Length != t2.Length) return true;
84          bool fl = t1 == t2;
85          return !fl;
86      }
        11 references
87      public void Print()
88      {
89          for (int i = 0; i < Length; i++)
90              Console.Write("{0}\t", a[i]);
91          Console.WriteLine();
92      }
        3 references
93      public static vector operator ++(vector t1)
94      {
95          vector rez = new vector(t1);
96          for (int i = 0; i < t1.Length; i++)
97              rez.a[i]++;
98          return rez;
99      }
100 }
```

Розглянемо програму, яка використовує цей клас:

Program.cs

OverloadedOps — OverloadedOps.Program

```
        0 references
8       class Program
9       {
            0 references
10          static void Main(string[] args)
11          {
12              int n = 10; int[] a = new int[n];
13              Random sd = new Random();
14              for (int i = 0; i < n; i++)
15                  a[i] = sd.Next(-100, 100);
16              vector A = new vector(a);
17              Console.WriteLine("A:");   A.Print();
18              for (int i = 0; i < n; i++)
19                  a[i] = sd.Next(-100, 100);
20              vector B = new vector(a);
21              Console.WriteLine("B:"); B.Print();
22
23              vector C = new vector(A + B);
24              Console.WriteLine("C=A+B:");  C.Print();
25
26              Console.WriteLine("C=A-B:");  C = A - B; C.Print();
27
28              C = A + 5; Console.WriteLine("C=A+5:"); C.Print();
29
30              C = 10 + A; Console.WriteLine("C=10+A:"); C.Print();
```

```
31
32              A++; Console.WriteLine("A:"); A.Print();
33
34              C = A++; Console.WriteLine("C=A++:"); C.Print();
35
36              Console.WriteLine("A:"); A.Print();
37
38              C = ++A; Console.WriteLine("C=++A:"); C.Print();
39
40              Console.WriteLine("A:"); A.Print();
41
42              if (A == B)  Console.WriteLine("Вектори рівні");
43              if (A != B)  Console.WriteLine("Вектори не рівні");
44              B = A;
45              if (A == B)  Console.WriteLine("Вектори рівні");
46              if (A != B)  Console.WriteLine("Вектори не рівні");
47
48              Console.ReadKey();
49          }
50      }
```

Результат:

```
D:\ОНУ им. І.І.Мечникова
A:
-85     -80     3       3       15      -72     2       99      -92     8
B:
42      -96     47      13      76      31      -43     52      50      -33
C=A+B:
-43     -176    50      16      91      -41     -41     151     -42     -25
C=A-B:
-127    16      -44     -10     -61     -103    45      47      -142    41
C=A+5:
-80     -75     8       8       20      -67     7       104     -87     13
C=10+A:
-75     -70     13      13      25      -62     12      109     -82     18
A:
-84     -79     4       4       16      -71     3       100     -91     9
C=A++:
-84     -79     4       4       16      -71     3       100     -91     9
A:
-83     -78     5       5       17      -70     4       101     -90     10
C=++A:
-82     -77     6       6       18      -69     5       102     -89     11
A:
-82     -77     6       6       18      -69     5       102     -89     11
Вектори не рівні
Вектори рівні
```

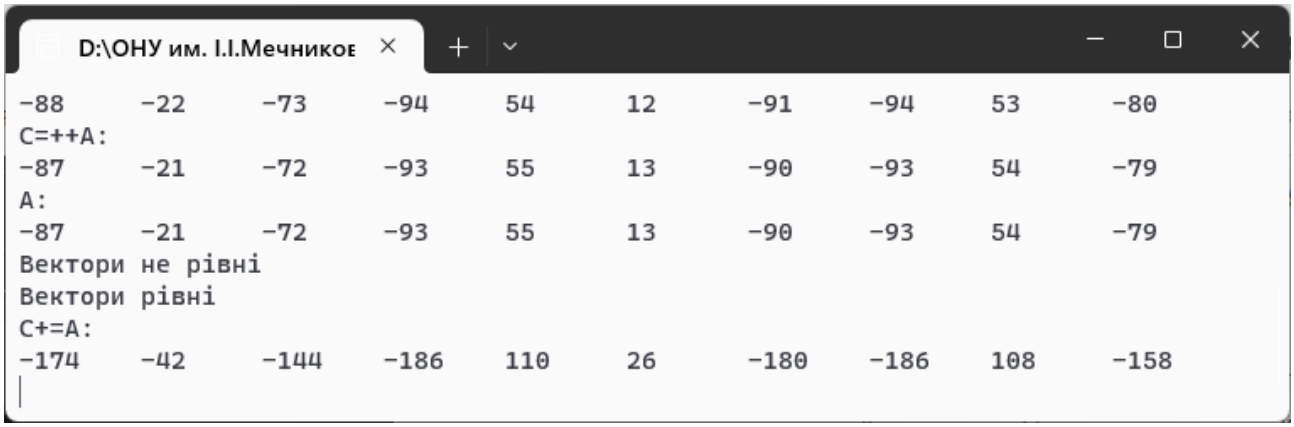
Зауваження 1. У **C#** неможливо перевантажувати операції префіксного і постфіксного інкремента / декремента окремо; тим не менш, значення, що повертається інкременту/декременту автоматично обробляється правильно.

Зауваження 2. У **C#** операції скороченого присвоювання автоматично емулюються при перевантаженні відповідних бінарних операцій. Таким чином,

якщо в класі **vector** вже перевантажені операції **+** і **-**, можна написати команду:

```
C += A;
```

Результат:



Зауваження 3. У разі перевантаження операцій еквівалентності для певного класу пам'ятайте, що **C#** вимагає, щоб у разі перевантаження операції == обов'язково перевантажувалася також і операція != (компілятор нагадає, якщо ви цього це зробите).

5.2 Індексатори

Як ви знаєте, індексація масивів реалізується з використанням оператора **[]**. У своїх класах ви можете перевантажити його, але не вдаючись до "послуг" методу **operator()**, а за допомогою створення індексатора (**indexer**).

Індексатор представлений як дещо видозмінене визначення властивості **C#**. Індексатор дозволяє забезпечити індексований доступ до об'єкта. Головне призначення індексаторів – підтримати створення спеціалізованих масивів, на які накладається одне або декілька обмежень. При цьому індексатори можна використовувати в синтаксисі, подібному до реалізованому в масивах. Індексатори можуть характеризуватися однією або кількома розмірностями, але ми почнемо з одновимірних індексаторів.

5.3 Створення одновимірних індексаторів

Одновимірний індексатор має такий вигляд:

тип_елементу this[*тип_індексу індекс*]

{ // Аксессор читання даних,

get { // Повернення значення, що задається елементом і*ндекс.* }

// Мутатор установки даних,

set { // Установка значення, заданого елементом і*ндекс.* } }

Тут *тип_елемента* – базовий тип індексатора. Таким чином, *тип_елемента* – це тип кожного елемента, до якого надається доступ за допомогою індексатора. Він відповідає базовому типу масиву. Параметр *індекс* отримує індекс опитуваного (або встановлюваного) елемента. Строго кажучи, цей параметр не обов'язково повинен мати тип **int**, але оскільки індексатори зазвичай використовуються для забезпечення індексації масивів, цілочисельний тип – найбільш підходящий.

Приклад 2.

Додамо в наш клас **vector** одновимірний індексатор із цілочисельним індексом, закритий метод для перевірки індексу на допустиме значення **ok()** і відкрите поле **errflag** булевого типу, яке відіграватиме роль індикатора роботи індексатора.

vector.cs*

Indexer — Indexer.vector

```
7   {
        36 references
8       public class vector
9       {
10          public int Length;
11          private int[] a;
12          public bool errflag;
13
```

```
117         4 references
            private bool ok(int index)
118         {
119             if (index >= 0 & index < Length) return true;
120             return false;
121         }
            3 references
122         public int this[int index]
123         {
124             get
125             {
126                 if (ok(index))
127                 {
128                     errflag = false;  return a[index];
129                 }
130                 else
131                 {
132                     errflag = true;   return 0;
133                 }
134             }
135             set
136             {
137                 if (ok(index))
138                 {
139                     a[index] = value; errflag = false;
140                 }
141                 else errflag = true;
142             }
143         }
```

Розглянемо програму, яка використовує цей клас:

Program.cs | vector.cs

Indexer | Indexer.Program

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        vector fs = new vector(5);
        int x;

        Console.WriteLine();
        for (int i = 0; i < (fs.Length * 2); i++)
            fs[i] = i * 10;
        for (int i = 0; i < (fs.Length * 2); i++)
        {
            x = fs[i];
            Console.Write(x + " ");
        }
        Console.WriteLine('\n');

        Console.WriteLine("Робота з повідомленням про помилки.");
        for (int i = 0; i < (fs.Length * 2); i++)
        {
            fs[i] = i * 10;
            if (fs.errflag)
                Console.WriteLine("fs[" + i + "] поза межами");
        }
        for (int i = 0; i < (fs.Length * 2); i++)
        {
            x = fs[i];
            if (!fs.errflag) Console.Write(x + " ");
            else
                Console.Write("\nfs[" + i + "] поза межами");
        }
        Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечн

0 10 20 30 40 0 0 0 0 0

Робота з повідомленням про помилки.
fs[5] поза межами
fs[6] поза межами
fs[7] поза межами
fs[8] поза межами
fs[9] поза межами
0 10 20 30 40
fs[5] поза межами
fs[6] поза межами
fs[7] поза межами
fs[8] поза межами
fs[9] поза межами
```

Зауваження 1. Створений тут індексатор запобігає порушенню меж масиву.

5.4 Перевантаження індексаторів

Індексатори можна перевантажувати. Тут наведено приклад визначення класу **vector**, у якому перевантажується індексатор для індексів типу **double**. Насправді **double** індексатор округлює індекс до найближчого цілого числа. Таким чином, із двох визначених у класі індексаторів буде виконуватися той, для якого виявиться найкращою відповідність типів параметра індексатора і його аргументу, використовуваного як індекс.

Приклад 3

```
144         public int this[double idx]
145         {
146             get
147             {
148                 int index;
149                 // Округлення до найближчого цілого int-значення.
150                 if ((idx - (int)idx) < 0.5) index = (int)idx;
151                 else index = (int)idx + 1;
152                 if (ok(index))
153                 {
154                     errflag = false;  return a[index];
155                 }
156                 else
157                 {
158                     errflag = true;   return 0;
159                 }
160             }
161             set
162             {
163                 int index;
164                 // Округлення до найближчого цілого int-значення.
165                 if ((idx - (int)idx) < 0.5) index = (int)idx;
166                 else index = (int)idx + 1;
167                 if (ok(index))
168                 {
169                     a[index] = value;  errflag = false;
170                 }
171                 else errflag = true;
172             }
173         }
```

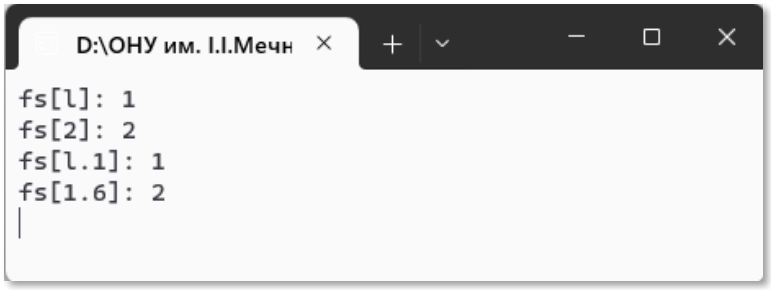
Розглянемо програму, яка використовує цей клас:

Program.cs | vector.cs

Indexer | Indexer.Program

```
0 references
8   class Program
9   {
        0 references
10      static void Main(string[] args)
11      {
12          vector fs = new vector(5);
13          for (int i = 0; i < fs.Length; i++)
14              fs[i] = i;
15          //Тепер використовуємо як індекс int- і double-значення.
16
17          Console.WriteLine("fs[1]: " + fs[1]);
18          Console.WriteLine("fs[2]: " + fs[2]);
19          Console.WriteLine("fs[1.1]: " + fs[1.1]);
20          Console.WriteLine("fs[1.6]: " + fs[1.6]);
21
22          Console.ReadKey();
23      }
24  }
```

Результат:



Як підтверджують результати виконання цієї програми, **double**-індекси округлюються до найближчих цілих значень. Зокрема, число 1.1 округляється до 1, а число 1.6 – до 2.

Незважаючи на те, що перевантаження індексатора, показане в цій програмі, цілком допустиме, цей приклад – нетиповий. Найчастіше індексатор перевантажується, щоб мати можливість використовувати екземпляр класу як індекс, значення якого обчислюється спеціальним чином.

5.5 Індексаторам не потрібен базовий масив

Індексатор може не використовувати базовий масив. Цілком достатньо, щоб індексатор забезпечував функціонування, яке для користувача виглядало б, як то, що забезпечують масиви. Наприклад, наступна програма включає індексатор, який діє подібно масиву, призначеному тільки для читання.

Цей "масив" містить ступені числа 2 для чисел від 0 до 15. Однак насправді ніякого масиву не існує. Замість цього індексатор обчислює відповідне значення для заданого індексу.

Приклад 4

PwrOfTwo.cs*

NoneMassiv — NoneMassiv.PwrOfTwo

```
2 references
class PwrOfTwo
{
    3 references
    public int this[int index]
    {
        // Обчислюємо та повертаємо ступінь числа 2.
        get
        {
            if ((index >= 0) && (index < 16)) return pwr(index);
            else return -1;
        }
    }
    1 reference
    int pwr(int p)
    {
        int result = 1;
        for (int i = 0; i < p; i++)
            result *= 2;
        return result;
    }
}
```

```
Program.cs
NoneMassiv                                   NoneMassiv.Program
    {
        0 references
        static void Main(string[] args)
        {
            PwrOfTwo pwr = new PwrOfTwo();
            Console.Write("Перші 8 ступенів числа 2: ");
            for (int i = 0; i < 8; i++)
                Console.Write(pwr[i] + " ");
            Console.WriteLine();
            Console.Write("А ось кілька помилок: ");
            Console.Write(pwr[-1] + " " + pwr[17]);

            Console.ReadKey();
        }
    }
```

Результат:

```
D:\ОНУ им. І.І.Мечнико
Перші 8 ступенів числа 2: 1 2 4 8 16 32 64 128
А ось кілька помилок: -1 -1
```

Зауваження 1. Зверніть увагу на те, що індексатор класу **PwrOfTwo** включає тільки **get**-аксессор, що означає: індексатор призначений тільки для читання. Таким чином, об'єкт класу **PwrOfTwo** можна використовувати в правій частині інструкції присвоєння, але ні в якому разі не в лівій. Наприклад, спроба додати цю інструкцію в попередню програму, приречена на невдачу:

```
pwr [11] = 0; // НЕ скомпілюється
```

Зауваження 2. На використання індексаторів накладається два обмеження. По-перше, оскільки в індексаторі не визначається область пам'яті, отримане індексатором значення не можна передавати методу в якості **ref**- або **out**-параметр (як і властивості). По-друге, індексатор повинен бути членом примірника свого класу, тому його не можна оголошувати з використанням ключового слова **static**.

5.6 Багатовимірні індексатори

Можна створювати індексатори і для багатовимірних масивів. Наприклад, ось як працює двовимірний відмовостійкий масив. Зверніть особливу увагу на спосіб оголошення індексатора в цьому класі.

Приклад 5

SoftArray2D.cs | PwrOfTwo.cs | Program.cs

Massiv2D | Massiv2D.SoftArray2D

```
3 references
class SoftArray2D
{
    int[,] a; // Посилання на базовий двовимірний масив.
    int rows, cols; // розмірності
    public int Length; // Length - відкритий член.
    public bool errflag; // Індикатор результату останньої операції.
    // Створюємо масив заданого розміру,
    1 reference
    public SoftArray2D(int r, int c)
    {
        rows = r;
        cols = c;
        a = new int[rows, cols];
        Length = rows * cols;
    }
    // Це індексатор для класу SoftArray2D.
    4 references
    public int this[int indexl, int index2]
    {
        get
        {
            if (ok(indexl, index2))
            {
                errflag = false;  return a[indexl, index2];
            }
            else
            {
                errflag = true;   return 0;
            }
        }
        set
        {
            if (ok(indexl, index2))
            {
                a[indexl, index2] = value; errflag = false;
            }
            else errflag = true;
        }
    }// Метод повертає значення true, якщо індекси
     // перебувають усередині кордонів,
    2 references
    private bool ok(int indexl, int index2)
    {
        if (indexl >= 0 & indexl < rows &
        index2 >= 0 & index2 < cols)
            return true;
        return false;
    }
```

Клас **SoftArray2D**, який представляє "м'який" двовимірний масив з можливістю обробки помилок при виході за межі масиву. Метод **ok** перевіряє, чи знаходяться індекси в межах масиву.

Розглянемо застосунок, що використовує цей клас

Program.cs*

Massiv2D — Massiv2D.Program

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        SoftArray2D fs = new SoftArray2D(3, 5);
        int x;
        // Демонструємо "м'яку посадку" при помилках.
        Console.WriteLine("М'яке приземлення.");
        for (int i = 0; i < 6; i++)
            fs[i, i] = i * 10;
        for (int i = 0; i < 6; i++)
        {
            x = fs[i, i];
            Console.Write(x + " ");
        }
        Console.WriteLine();
        //А тепер генеруємо помилки.
        Console.Write("Робота з повідомленням про помилки.");
        for (int i = 0; i < 6; i++)
        {
            fs[i, i] = i * 10;
            if (fs.errflag)
                Console.Write("\nfs[" + i + ", " + i + "] поза межами");
        }
        Console.WriteLine();
        for (int i = 0; i < 6; i++)
        {
            x = fs[i, i];
            if (!fs.errflag) Console.Write(x + " ");
            else
                Console.Write(
                "\nfs[" + i + ", " + i + "] поза межами");
        }
        Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечн
М'яке приземлення.
0 10 20 0 0 0
Робота з повідомленням про помилки.
fs[3, 3] поза межами
fs[4, 4] поза межами
fs[5, 5] поза межами
0 10 20
fs[3, 3] поза межами
fs[4, 4] поза межами
fs[5, 5] поза межами
```

6. СПАДКУВАННЯ

Спадкування – це механізм створення нового класу з існуючого класу. Існуючий клас може бути доповнений або змінений для створення похідного класу. Спадкування – це потужний механізм повторного використання кодів. За допомогою успадкування може бути створена ієрархія родинних типів, які спільно використовують код і інтерфейси.

Спадкування є однією з найбільш важливих особливостей ООП. Будь-який клас може успадковуватися від іншого класу, що означає, що він буде володіти станом і поведінкою того класу, від якого він успадковується. У термінології ООП клас, від якого відбувається спадкування, називається **класом-батьком** або базовим класом. Клас, який успадковує базовий клас, називається **дочірнім** класом або **похідним.**

6.1 Похідний клас

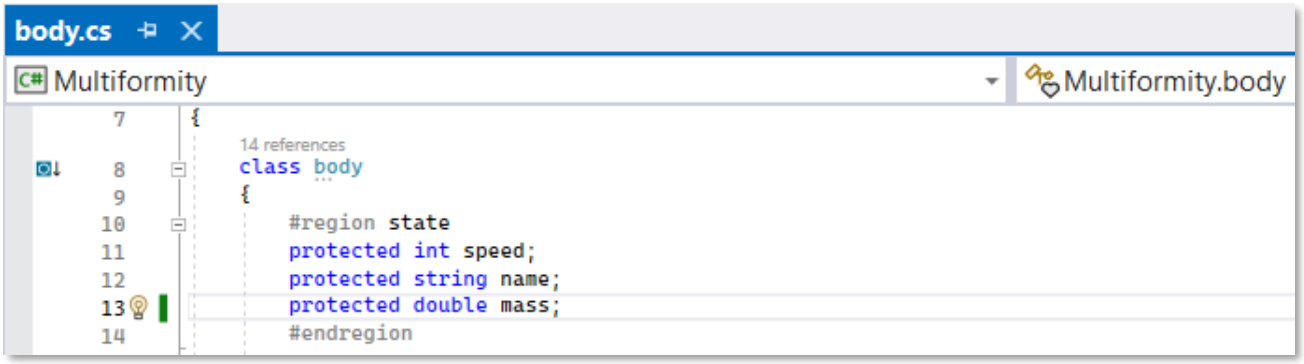
Загальна форма оголошення класу, який успадковує базовий клас, має такий вигляд:

```
class ім'я_похідного_класу : ім'я_базового_класу
{  //Тіло класу   }
```

Для створюваного похідного класу можна вказати тільки один базовий клас. Можна створити ієрархію спадкування, в якій один похідний клас стає базовим для іншого похідного класу. І, звичайно ж, жоден клас не може бути базовим (ні прямо, ні побічно) для самого себе.

Ви пам'ятаєте, що члени класу можуть бути типів: **private** (закриті) і **public** (відкриті). Однак члени класу, оголошені як **private**, недоступні для наслідування. Існує третій тип членів класу – **protected** (захищені). Захищені члени класу доступні для всіх вироблених класів і недоступні для всіх зовнішніх класів.

Приклад 1. Для побудови ієрархії класів скористаємося розробленим класом **body**, змінимо в його даних-членах модифікатор доступу на **protected:**



Створимо похідний клас **Square** з базового класу **body**.

Square.cs | body.cs

Multiformity | Multiformity.Square

```
1 reference
class Square : body
{
    0 references
    public void picture(char c)
    {
        for (int i = 0; i < speed; i++)
        {
            for (int j = 0; j < speed; j++)
                Console.Write(c);
            Console.WriteLine();
        }
    }
}
```

Розглянемо додаток використовує побудови ієрархії класів:

Program.cs | Square.cs | body.cs

Multiformity | Multiformity.Program

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        body t1 = new body("Krug", 6, 10);
        Console.WriteLine("Body t1:");
        t1.Inform();
        Square t2 = new Square();
        Console.WriteLine("\nSquare t2:");
        t2.Inform();
        Console.WriteLine("\nПісля введених змін");
        t2.Name = "Kvad";
        t2.Speed = 5;
        //t2.Mass = 90;
        t2.Inform();
        t2.picture('*');
        t2.Move();
        Console.ReadLine();
    }
}
```

Результат:

C:\WINDOWS\system32\

```
Body t1:
Я тело Krug имею массу 10 и скорость 6

Square t2:
Я тело Просто тело имею массу 2,5 и скорость 0

Після введених змін
Я тело Kvad имею массу 2,5 и скорость 5
*****
*****
*****
*****
*****
Двигаюсь со скоростью 5 км/час
```

Зауваження 1. Екземпляри класу **Square** успадковують стан і поведінку класу **body**, які можуть бути як відкритими, так і захищеними. Коли в класі **Square** створюється об'єкт **t2**, то для цього з класу **body** викликається базовий конструктор. Потім викликається конструктор класу **Square**, який завершує створення екземпляра.

Зауваження 2. Оскільки екземпляр класу **Square** – спадкоємець класу **body**, то можна викликати методи класу **body**.

6.2 Конструктори і успадкування

В ієрархії класів як базові, так і похідні класи можуть мати власні конструктори. Конструктор базового класу створює частину екземпляра, що відповідає базовому класу, а конструктор похідного класу – частину екземпляра, що відповідає похідному класу. У ***прикладі 1*** в похідному класі не було власних даних-членів і власного конструктора, а значить, використовувався конструктор за замовчуванням похідного класу, який поставлявся компілятором. При цьому відбувається обов'язковий виклик конструктора базового класу, з прикладу видно: викликаний конструктор базового класу без вхідних параметрів. Виникає питання: *а якщо такого конструктора немає в базовому класі? Що буде?* Відповідь проста: помилка компіляції. Це пояснюється тим, що якщо в класі є хоч один конструктор, компілятор не постачає стандартний конструктор, який необхідний однозначно в даному прикладі.

Похідний клас може викликати конструктор, визначений у його базовому класі, використовуючи розширену форму оголошення конструктора похідного класу і ключове слово **base**. Формат розширеного оголошення такий:

конструктор_похідного_класу (*список_параметрів)* : base *(список_аргументів)*
{
// тіло конструктора
}

Тут за допомогою елемента *спісок_аргументів* задаються аргументи, необхідні конструктору в базовому класі. За допомогою ключового слова **base** можна викликати конструктор будь-якої форми, визначений у базовому класі. Реально ж виконається той конструктор, параметри якого відповідатимуть переданим при виклику аргументам. За відсутності ключового слова **base** автоматично викликається конструктор базового класу, що діє за замовчуванням.

Пример 2. Припустимо, що клас **Square** має власний закритий член **size**, який має методи доступу. Для класу **Square** ми задамо перевантажені конструктори. Перший конструктор ініціалізуватиме тільки член класу **Square – size**, а другий конструктор має ініціалізувати всі члени, як власні, так і батьківські.

Square.cs | Program.cs | body.cs

Multiformity | Multiformity.Square

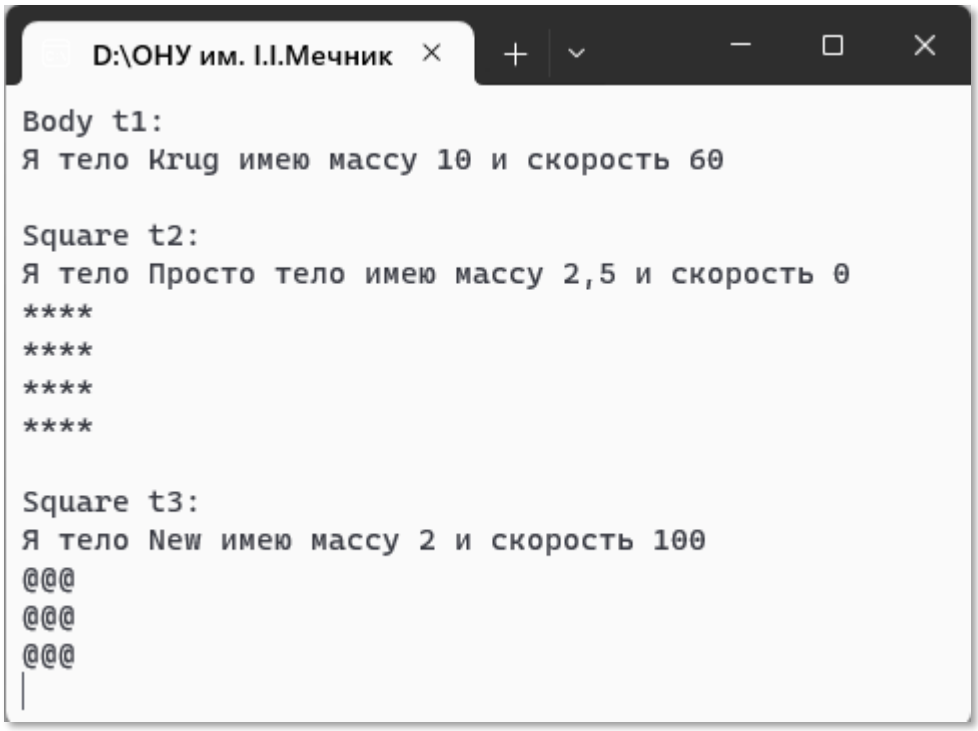
```
{
    6 references
    class Square : body
    {
        int size;
        0 references
        public int Size
        {
            get { return size; }
            set
            {
                if (value > 0)
                    size = value;
            }
        }
        1 reference
        public Square(int size)
        {
            this.size = size;
        }
        1 reference
        public Square(string name, int speed, double mass, int size) : base(name, speed, mass)
        {
            this.size = size;
        }
        2 references
        public void picture(char c)
        {
            for (int i = 0; i < size; i++)
            {
                for (int j = 0; j < size; j++)
                    Console.Write(c);
                Console.WriteLine();
            }
        }
}
```

Square.cs | Program.cs | body.cs

Multiformity | Multiformity.Program

```
    0 references
    class Program
    {
        0 references
        static void Main(string[] args)
        {
            body t1 = new body("Krug", 60, 10);
            Console.WriteLine("Body t1:");
            t1.Inform();
            Square t2 = new Square(4);
            Console.WriteLine("\nSquare t2:");
            t2.Inform();
            t2.picture('*');
            Square t3 = new Square("New", 100, 2, 3);
            Console.WriteLine("\nSquare t3:");
            t3.Inform();
            t3.picture('@');
            Console.ReadLine();
        }
    }
```

Результат:



6.3 Приховування членів базового класу.

Ми знаємо, що об'єкт класу **Square** має доступ до всіх функцій-членів класу **body**, але крім цього, базові функції можуть бути приховані у похідному класі. Якщо у похідному класі створюється функція з таким самим поверненням і сигнатурою як і в базовому класі, але виконувана особливим чином, то має місце приховування методу базового класу. При цьому компілятор видає попередження про приховування методу. Наприклад, додамо в клас **Square** функцію **Move()**, а також функцію виведення **Inform().**

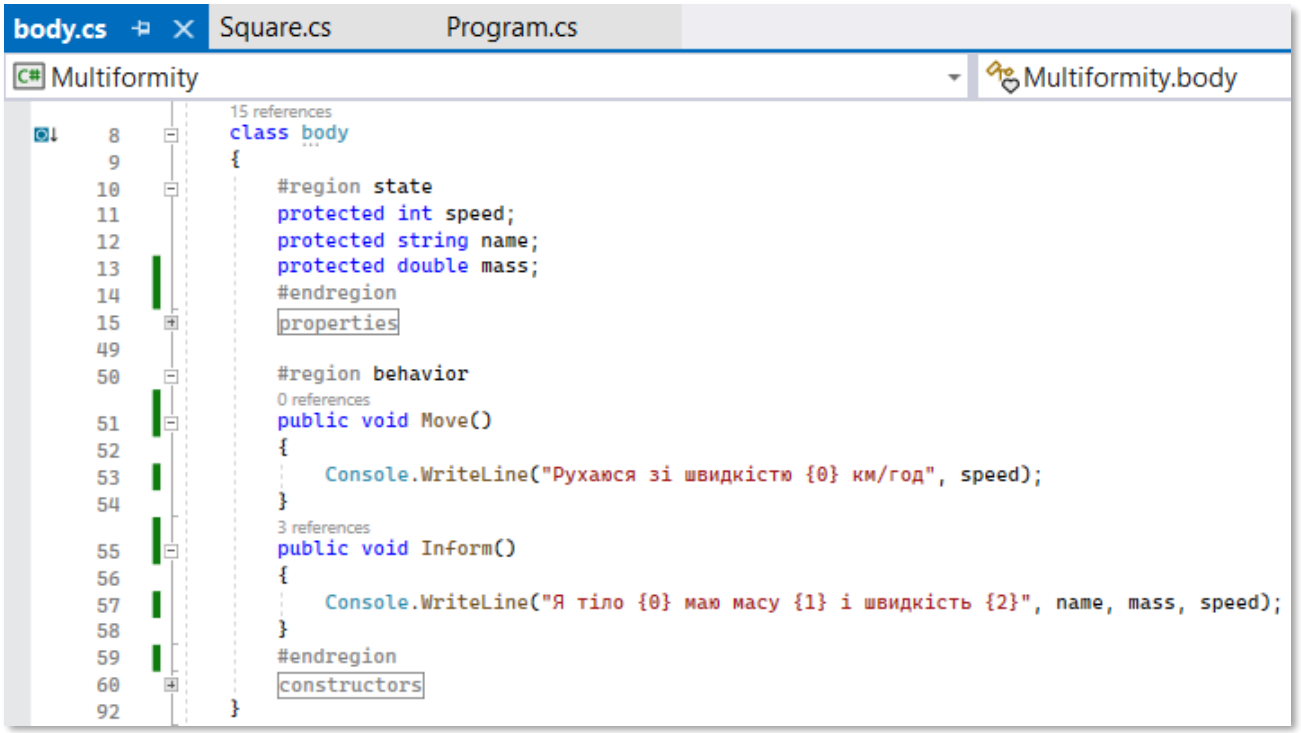
Крім того, у класі-спадкоємці можуть бути присутніми однойменні дані-члени, тобто імена яких збігаються з іменами даних-членів базового класу. Тоді теж відбувається приховування даних. Наприклад, нехай у класі **Square** є власний даний член

```
string name;
```

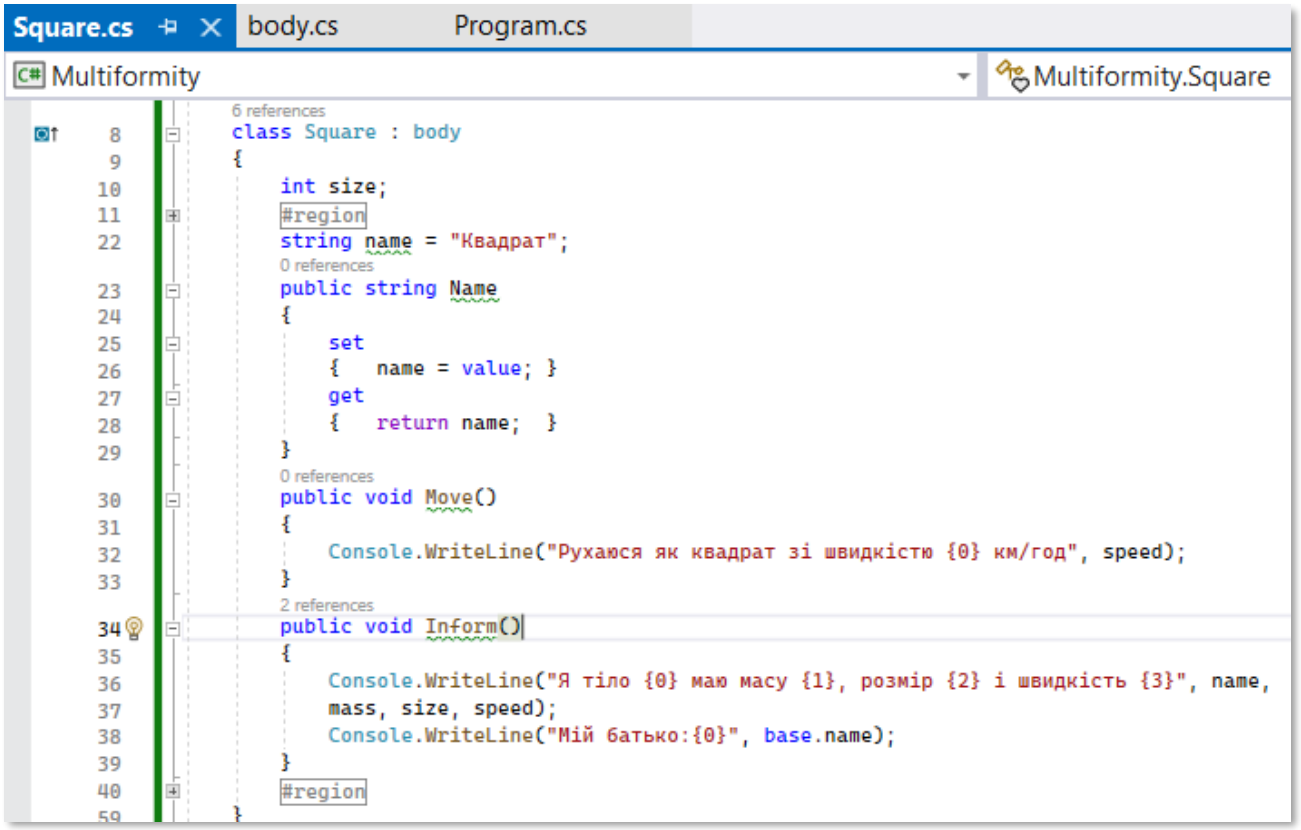
У разі потреби можна отримати доступ до прихованих даних і функцій базового класу, використовуючи ключове слово **base**, яке означає посилання на базовий клас (аналог посилання **this** для класу, що розробляється).

Приклад 3

(Базовий клас)



```
body.cs | Square.cs | Program.cs
Multiformity                                  Multiformity.body
15 references
class body
{
    #region state
    protected int speed;
    protected string name;
    protected double mass;
    #endregion
    properties

    #region behavior
    0 references
    public void Move()
    {
        Console.WriteLine("Рухаюся зі швидкістю {0} км/год", speed);
    }
    3 references
    public void Inform()
    {
        Console.WriteLine("Я тіло {0} маю масу {1} і швидкість {2}", name, mass, speed);
    }
    #endregion
    constructors
}
```



```
Square.cs | body.cs | Program.cs
Multiformity                                  Multiformity.Square
6 references
class Square : body
{
    int size;
    #region
    string name = "Квадрат";
    0 references
    public string Name
    {
        set
        {   name = value; }
        get
        {   return name;  }
    }
    0 references
    public void Move()
    {
        Console.WriteLine("Рухаюся як квадрат зі швидкістю {0} км/год", speed);
    }
    2 references
    public void Inform()
    {
        Console.WriteLine("Я тіло {0} маю масу {1}, розмір {2} і швидкість {3}", name,
        mass, size, speed);
        Console.WriteLine("Мій батько:{0}", base.name);
    }
    #region
}
```

Розглянемо застосунок, що використовує побудовану ієрархію класів:

```
Program.cs | Square.cs | body.cs
Multiformity | Multiformity.Program
```

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        Square t = new Square("Для квадрата", 100, 2, 3);
        t.picture('#');
        t.Move();
        t.Inform();
        Console.ReadLine();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечникова
###
###
###
Рухаюся як квадрат зі швидкістю 100 км/год
Я тіло Квадрат маю масу 2, розмір 3 і швидкість 100
Мій батько:Для квадрата
```

Зауваження 1. Викликається конструктор без параметрів, так як використовується конструктор похідного класу тільки з вхідними параметрами для ініціалізації власних даних-членів.

Зауваження 2. Викликаються методи **Move()** і **Inform()** для похідного класу, так як однойменні методи базового класу приховані.

Зауваження 3. Функція **Inform()** класу **Square** виводить на екран **name** базового класу, використовуючи службове слово **base**.

Зауваження 4. Для запобігання повідомлення компілятора про приховування членів класу використовується службове слово **new**.

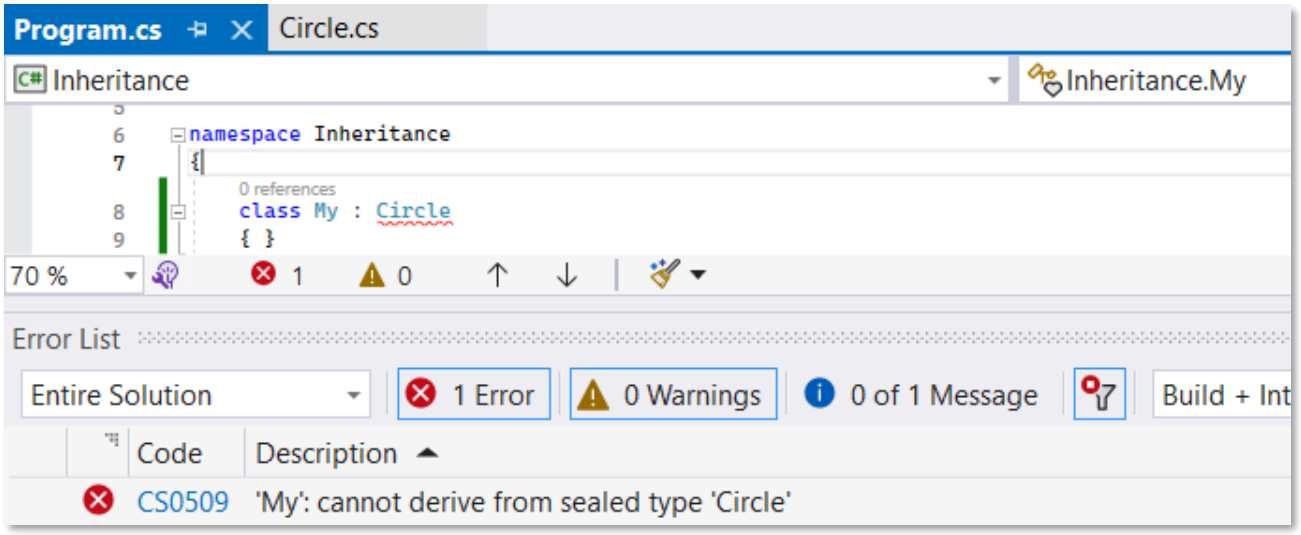
```
class Square : body
{
    int size;
    #region
    new string name = "Квадрат";
    0 references
    new public string Name
    {
        #region
    }
    1 reference
    new public void Move()
    {
        #region
    }
    1 reference
    new public void Inform()
    {
        #region
    }
```

6.4 Використання ключового слова sealed для запобігання успадкування

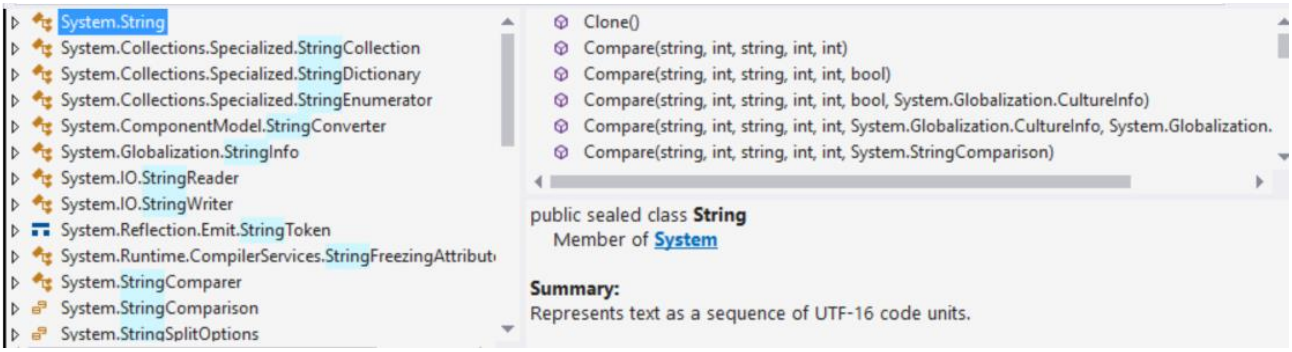
Яким би потужним і корисним не був механізм успадкування, все ж іноді необхідно його відключати. Щоб заборонити спадкування класу, передуватимете його оголошення ключовим словом **sealed** (запечатаний).

```
sealed class Circle: body
    {  // ...  }
```

Якщо ви (або колега по команді) спробуєте успадкувати від цього класу, то отримаєте помилку на етапі компіляції:



Найчастіше запечатування має сенс при проектуванні обслуговуючого класу. Наприклад, в просторі імен **System** визначено безліч запечатаних класів. У цьому легко переконатися, відкривши браузер об'єктів в **Visual Studio** (через меню **View**) і вибравши клас **String**, визначений у просторі імен **System** всередині збірки mscorlib.dll. Зверніть увагу на використання ключового слова **sealed**, виділеного у вікні **Summary**:



Зауваження. Структури **C#** завжди неявно запечатані. Тому ні успадкувати одну структуру від іншої, ні клас від структури, ні структуру від класу не вийде. Структури можуть використовуватися тільки для моделювання окремих атомарних, визначених користувачем типів.

6.5 Другий принцип ООП: подробиці про спадкування

Спадкування – це аспект ООП, що полегшує повторне використання коду. Строго кажучи, повторне використання коду існує в двох видах: успадкування (***відношення "є")*** і модель включення / делегації (***відношення "має"***). Ми з Вами розглянули класичну модель успадкування – відношення "є". При установці між класами ***відношення "є"*** будується залежність між двома або більше типами класів. Базова ідея, що лежить в основі класичного наслідування, полягає в тому, що нові класи можуть створюватися з застосуванням існуючих класів в якості відправної точки, тобто є спадкоємцями.

Приклад 4. Розглянемо ***відношення "має"*** (ще відоме під назвою моделі включення / делегації або агрегації). Припустимо, що у нас є система тіл.

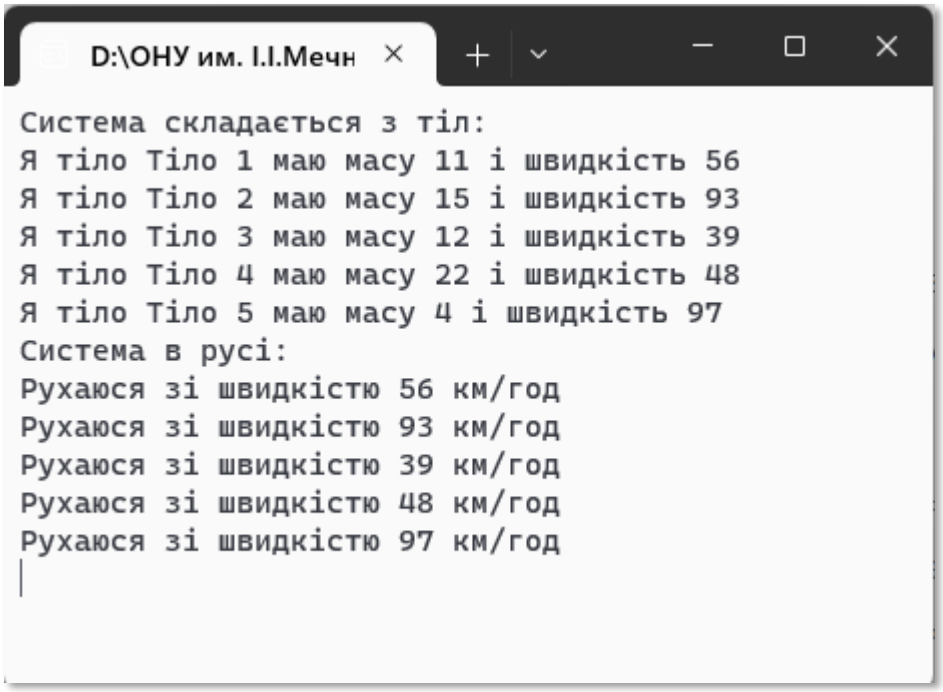
System.cs | Program.cs | Circle.cs

Inheritance | Inheritance.System.

```
2 references
class System
{
    protected List <body> a=new List<body>();
    Ups
    1 reference
    public void Move()
    {
        Console.WriteLine("Система в русі:");
        for (int i=0;i<a.Count;i++)
            a[i].Move();
    }
    1 reference
    public void Inform()
    {
        Console.WriteLine("Система складається з тіл:");
        for (int i = 0; i < a.Count; i++)
            a[i].Inform();
    }
    1 reference
    public void Add(body A) { a.Add(A); }
}
```

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        #region Pr4
        System my = new System();
        int n = 5;
        body b;
        Random r = new Random();
        for (int i = 0; i < n; i++)
        {
            b = new body("Тело " + Convert.ToString(i + 1), r.Next(30, 100), r.Next(1, 25));
            my.Add(b);
        }
        my.Inform();
        my.Move();
        #endregion

        Console.ReadLine();

    }
}
```

Результат:



Таким чином, один об'єкт успішно містить в собі інший об'єкт. Однак щоб представити функціональність включеного об'єкта зовнішнього світу, потрібна **делегація**. **Делегація** – це просто процес додавання відкритих членів до включаючого класу, які використовують функціональність включеного об'єкта

6.6 Визначення вкладених типів

У **C#** допускається визначати тип (перерахування, клас, інтерфейс, структуру або делегат) безпосередньо всередині контексту класу або структури (це різновид тільки, що розглянутого відносини "має"). При цьому вкладений (або "внутрішній") тип вважається членом охоплювального (або "зовнішнього") класу, і в очах виконуючої системи їм можна маніпулювати як будь-яким іншим членом (полем, властивістю, методом або подією). Синтаксис, який використовується для вкладення типу, досить простий:

```
class OuterClass
    {
        // Відкритий вкладений тип який може використовуватися повсюди
        public class PublicInnerClass { }
        // Закритий вкладений тип може використовуватися тільки членами включає класу
        private class PrivateInnerClass { }
    }
```

Хоча синтаксис досить очевидний, зрозуміти, для чого це може знадобитися, не так-то просто. Щоб розібратися з цією технікою, розглянемо

характерні особливості вкладених типів.

- Вкладені типи дозволяють отримати повний контроль над рівнем доступу внутрішнього типу, оскільки вони можуть бути оголошені як закриті (не вкладені класи не можуть бути оголошені з використанням ключового слова **private**).
- Оскільки вкладений тип є членом включаючого класу, він може мати доступ до закритих членів класу, що включає.
- Часто вкладені типи зручні в якості допоміжних для зовнішнього класу і не призначені для використання зовнішнім світом.

Коли тип включає в себе інший тип класу, він може створювати змінні-члени цього типу, як будь-який інший елемент даних. Однак якщо вкладений тип потрібно застосовувати поза включаючого типу, його знадобиться кваліфікувати ім'ям включаючого типу. Погляньте на наступний код:

```
class Program
    {
        static void Main(string[] args)
        {
            // Створити і використовувати відкритий вкладений клас.
Правильно!
            OuterClass.PublicInnerClass inner;
            inner = new OuterClass.PublicInnerClass();
            // Помилка компіляції! Доступ до закритого класу неможливий!
            //OuterClass.PrivateInnerClass inner2;
            //inner2 = new OuterClass.PrivateInnerClass();
             Console.ReadLine();
        }
    }
```

7. ТРЕТІЙ ПРИНЦИП ООП: ПІДТРИМКА ПОЛІМОРФІЗМУ В C#

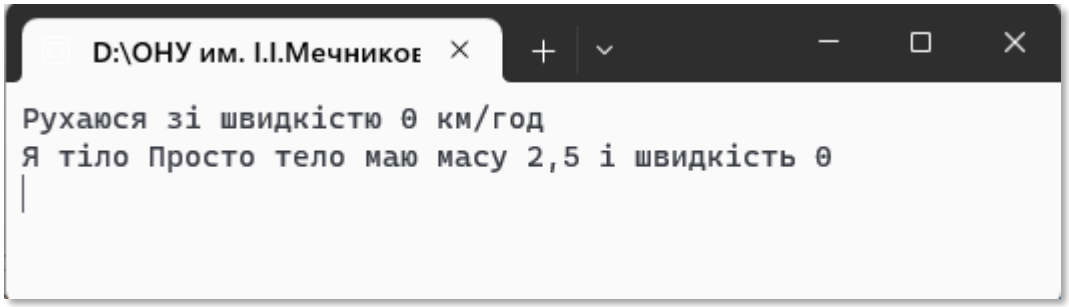
7.1 Віртуальні методи

Поліморфізм надає підкласу спосіб визначення власної версії методу, визначеного в його базовому класі, з використанням процесу, який називається перевизначенням методу. Для цього потрібно використовувати ключові слова **virtual** і **override**. Якщо базовий клас бажає визначити метод, який може бути (але не обов'язково) перевизначений в підкласі, він повинен позначити його ключовим словом **virtual** (такий метод називається віртуальним). Коли клас бажає змінити реалізацію деталей віртуального методу, він робить це за допомогою ключового слова **override**. Таким чином, кожен похідний клас може мати власну версію віртуального методу.

Мова **C#** допускає привласнення посиланню на базовий клас посилань на похідні класи.

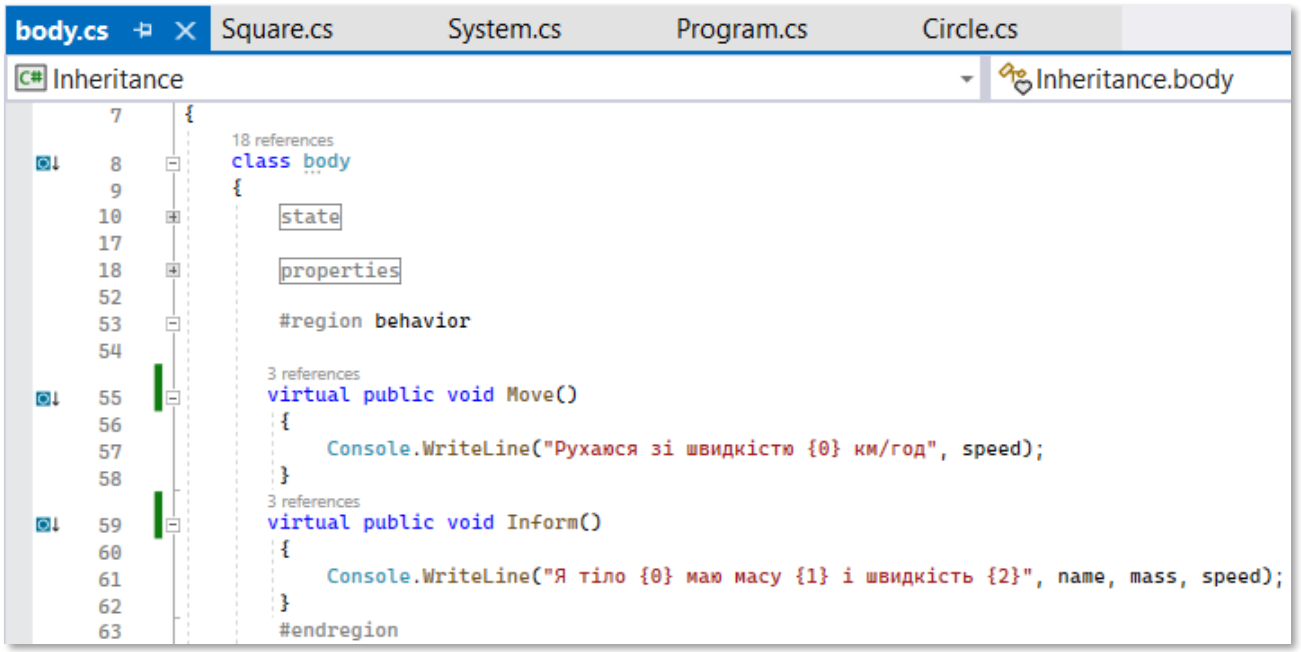
```
body t1 = new Square(5);
     t1.Move();
     t1.Inform();
```

Цей вираз створює в області динамічної пам'яті новий екземпляр класу Square і при цьому посилається на нього за допомогою посилання класу **body**. Потім за допомогою цього посилання можна викликати будь-який метод класу **body**.



***Зауваження*.** Видно, що викликаються методи **Move ()** і **Inform ()** класу **body.**

Якщо ми хочемо викликати перевантажені методи **Move()** і **Inform()** для класу **Square** при зверненні до методу через посилання на базовий клас, то ми повинні оголосити базові методи **Move()** і **Inform()** для класу **body** зі специфікатором **virtual**, а методи **Move()** і **Inform()** похідного класу зі специфікатором **override**. У цьому полягає суть використання віртуальних функцій.



```
Square.cs | body.cs | System.cs | Circle.cs
Inheritance                                   Inheritance.Square
        3 references
 8      class Square: body
 9      {
10          Pr3
68
            3 references
69          override public void Move()
70          {
71              Console.WriteLine("Рухаюся як квадрат зі швидкістю {0} км / год ", speed);
72          }
            3 references
73          override public void Inform()
74          {
75              Console.WriteLine("Я тіло {0} маю масу {1}, розмір {2} і швидкість {3}",
76                                name, mass, size, speed);
77              Console.WriteLine("Мій батько:{0}", base.name);
78          }
70 %    No issues found
```

```
Program.cs
Inheritance                                   Inheritance.Program
            0 references
10          class Program
11          {
                0 references
12              static void Main(string[] args)
13              {
14                  body t1 = new Square(5);
15                  t1.Move();
16                  t1.Inform();
17                  Console.ReadLine();
18
19              }
20          }
```

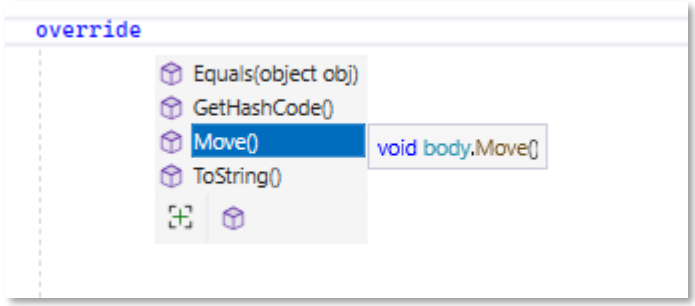
Результат:

```
D:\ОНУ им. І.І.Мечникое
Рухаюся як квадрат зі швидкістю 0 км / год
Я тіло Квадрат маю масу 2,5, розмір 5 ? швидкість 0
Мій батько:Просто тіло
```

За допомогою віртуальних методів реалізується третій принцип ООП – **поліморфізм**. Суть поліморфізму полягає в тому, що можна оголосити безліч екземплярів різних класів, які є спадкоємцями базового класу, у яких є один і той же метод, наприклад, руху, суть якого в кожному класі своя. Тоді створивши посилання на базовий клас і привласнюючи їй об'єкти похідних класів, можна звертатися до методу руху незалежно від того, з яким об'єктом в даний момент пов'язане посилання. Причому буде викликатися варіант методу, специфічний для класу обраного об'єкта.

Перевизначення віртуального методу формує базу для однієї з найпотужніших концепцій **C#**: ***динамічної диспетчеризації методів***. Динамічна диспетчеризація методів – це механізм виклику перевизначеного методу під час виконання програми, а не в період компіляції. Саме завдяки механізму диспетчеризації методів в **C#** реалізується динамічний поліморфізм.

Перевизначення віртуальних членів в **IDE**-середовищі **Visual Studio**. Як ви вже, можливо, помітили, при перевизначенні члена класу необхідно пам'ятати типи всіх параметрів, а також угоди про передачу параметрів (**ref**, **out** і **params**). У **Visual Studio** доступна дуже корисна можливість, якою можна користуватися при перевизначенні віртуального члена. Якщо набрати слово **override** всередині контексту типу класу (і натиснути клавішу пробілу), то **IntelliSense** автоматично відобразить список всіх переобумовлених членів батьківського класу, як показано на рисунку.



Після вибору члена і натискання клавіші **<Enter>** середовище **IDE** реагує автоматичним заповненням шаблону методу замість вас. Зверніть увагу, що також додається оператор коду, який викликає батьківську версію віртуального члена (цей рядок можна видалити, якщо він не потрібен). Наприклад, в разі застосування цієї техніки під час перевизначення методу **Move()** додасться наступний автоматично згенерований код:

```
public override void Move()
{
    base.Move();
}
```

Приклад 1

Розглянемо приклад з використання віртуальних методів. Цей код представляє собою просту консольну програму, яка визначає ієрархію геометричних фігур (**Rectangle**, **Triangle**, **Line**, **Trapezium**), що походять від базового класу **figure**. Клас **figure** містить загальні властивості та методи, а

похідні класи надають свою власну реалізацію методу **Draw()**, щоб відображати відповідні форми на консолі. Клас **Program** містить простий цикл, який дозволяє користувачеві переміщати фігури на консолі, натискуючи **1** на клавіатурі, або завершити роботу програми.

figure.cs

Boat — Boat.figure

```
13 references
class figure
{
    protected char ch;
    protected string name;
    protected int x;
    protected int y;
    protected int sk;

    0 references
    public figure(): this(0,10,1,"",'*')
    {
    }
    5 references
    public figure(int x, int y, int sk, string name, char ch)
    {
        this.ch = ch;
        this.x = x;
        this.y = y;
        this.sk = sk;
        this.name = name;
    }
    0 references
    public void SetCh(char c) { ch = c; }
    0 references
    public char GetCh() { return ch; }
    0 references
    public void SetX(int k) { x = k; }
    0 references
    public int GetX() { return x; }
    0 references
    public void SetY(int s) { y = s; }
    0 references
    public int GetY() { return y; }
    0 references
    public void SetName(string n) { name = n; }
    0 references
    public string GetName() { return name; }
    0 references
    public void SetSk(int k) { sk = k; }
    0 references
    public int GetSk() { return sk; }
    6 references
    virtual public void Draw() { }
    1 reference
    public void Move() { x = x + sk; }
}
```

Rectangle.cs* | Line.cs | Triangle.cs | figure.cs

Boat | Boat.Rectangle

```
5 references
class Rectangle : figure
{
    private int a;
    private int b;
    0 references
    public Rectangle() : this(0, 0) { }
    1 reference
    public Rectangle(int d1, int d2) { a = d1; b = d2; }
    1 reference
    public Rectangle(int k1, int k2, int d1, int d2, int s, string n, char c)
        : base(k1, k2, s, n, c) { a = d1; b = d2; }
    0 references
    public void SetA(int d) { a = d; }
    0 references
    public int GetA() { return a; }
    0 references
    public void SetB(int d) { b = d; }
    0 references
    public int GetB() { return b; }
    3 references
    override public void Draw()
    {
        Console.CursorLeft = x;
        Console.CursorTop = y;
        for (int i = 0; i < a; i++)
        {
            Console.CursorLeft = x;
            for (int j = 0; j < b; j++)
                Console.Write(ch);
            Console.WriteLine();
        }
    }
}
```

Line.cs | Triangle.cs | Trapezium.cs | Rectangle.cs | figure.cs

Boat | Boat.Line

```
6 references
class Line:figure
{
    private int d;
    0 references
    public Line():this(0) { }
    1 reference
    public Line(int a1) { d = a1; }
    2 references
    public Line(int k1, int k2, int d1, int s, string n, char c)
        : base(k1, k2, s, n, c) { d = d1; }
    0 references
    public void SetD(int dl) { d = dl; }
    0 references
    public int GetD() { return d; }
    3 references
    override public void Draw()
    {
        Console.CursorLeft = x; Console.CursorTop = y;
        for (int i = 0; i < d; i++)
        {
            Console.CursorLeft = x;
            Console.Write(ch);
            Console.WriteLine();
        }
    }
}
```

Trapezium.cs* | Rectangle.cs* | Line.cs | Triangle.cs | figure.cs

Boat | Boat.Trapezium

```
5 references
class Trapezium : figure
{
    private int a;
    private int h;
    0 references
    public Trapezium() : this(0, 0) { }
    1 reference
    public Trapezium(int d1, int d2) { a = d1; h = d2; }
    1 reference
    public Trapezium(int k1, int k2, int d1, int d2, int s, string n, char c)
        : base(k1, k2, s, n, c) { a = d1; h = d2; }
    0 references
    public void SetA(int d) { a = d; }
    0 references
    public int GetA() { return a; }
    0 references
    public void SetH(int d) { h = d; }
    0 references
    public int GetH() { return h; }
    3 references
    override public void Draw()
    {
        Console.CursorLeft = x;
        Console.CursorTop = y;
        for (int i = 0; i < h; i++)
        {
            Console.CursorLeft =x + i;
            for (int j = i; j < a - i; j++)
                Console.Write(ch);
            Console.WriteLine();
        }
    }
}
```

Triangle.cs | Trapezium.cs* | Rectangle.cs* | Line.cs | figure.cs

Boat | Boat.Triangle

```
5 references
class Triangle : figure
{
    private int h;
    0 references
    public Triangle():this(0) { }
    1 reference
    public Triangle(int a1) { h = a1; }
    1 reference
    public Triangle(int k1, int k2, int d1, int s, string n, char c) :
                    base(k1, k2, s, n, c) { h = d1; }
    0 references
    public void SetH(int d) { h = d; }
    0 references
    public int GetH() { return h; }
    3 references
    override public void Draw()
    {
        Console.CursorLeft = x;
        Console.CursorTop = y;
        for (int i = 0; i < h; i++)
        {
            Console.CursorLeft = x;
            for (int j = 0; j < i + 1; j++)
                Console.Write(ch);
            Console.WriteLine();
        }
    }
}
```

Program.cs | Triangle.cs | Trapezium.cs | Rectangle.cs | Line.cs | figure.cs

Boat | Boat.Program

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        char ch; int n = 5;
        figure[] Ris = new figure[n];

        Ris[0] = new Rectangle(19, 0, 2, 3, 2, "Flag", '*');
        Ris[1] = new Line(19, 2, 3, 2, "Shtok", '*');
        Ris[2] = new Triangle(19, 5, 10, 2, "Parus", '*');
        Ris[3] = new Line(19, 15, 2, 2, "Dergak", '*');
        Ris[4] = new Trapezium(0, 17, 40, 4, 2, "Korablik", '*');

        for (int i = 0; i < n; i++)
            Ris[i].Draw();
        interf();
        while (true)
        {
            ch = Convert.ToChar(Console.ReadLine());
            switch (ch)
            {
                case '1':
                    Console.Clear();
                    for (int i = 0; i < n; i++)
                    {
                        Ris[i].Move();
                        Ris[i].Draw();
                    }
                    interf();
                    break;
                default: return;
            }
        }
    }
    2 references
    static void interf()
    {  Console.WriteLine("\n\n1-dvishenie     2-vixod");   }
}
```

Результат:

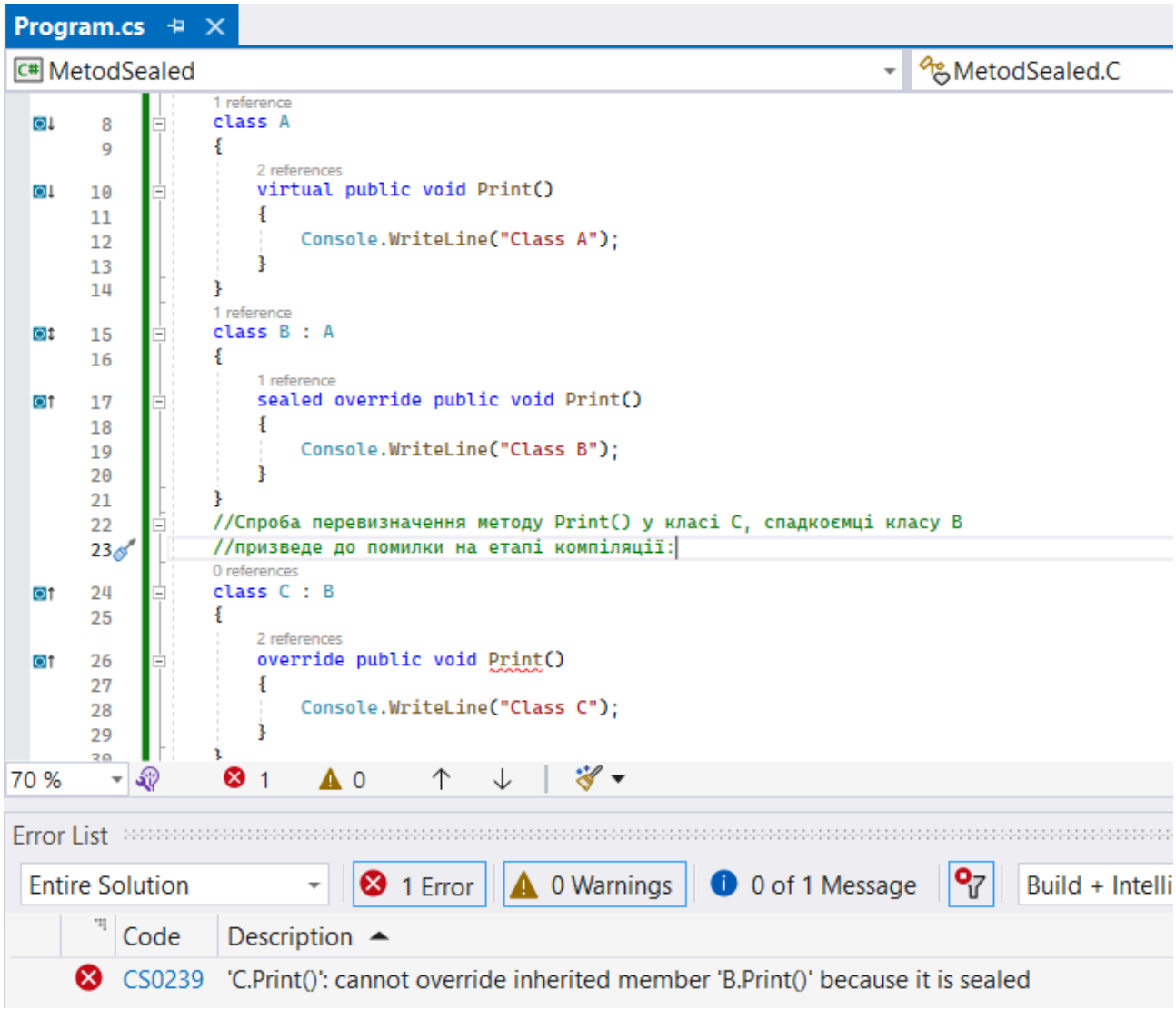
Зауваження. Віртуальний метод перевизначати необов'язково. Якщо похідний клас не надає власну версію віртуального методу, використовується версія, певна в базовому класі.

7.2 Запечатування віртуальних членів

Згадайте, що ключове слово **sealed** застосовується до типу класу для запобігання розширенню іншими типами його поведінки через наслідування. Іноді потрібно не запечатувати клас цілком, а просто запобігти перевизначення деяких віртуальних методів в похідних типах.

Якщо йдеться про запечатування тільки деяких віртуальних методів, а не всього класу, використовується ключове слово **sealed** у поєднанні з ключовим словом **override.**

Синтаксис виглядає наступним чином:



7.3 Абстрактні класи

З огляду на те, що багато базових класів схильні бути досить невизначеними сутностями, більш вдале проектне рішення для даного прикладу не повинно дозволяти безпосереднє створення в коді нового екземпляра класу **figure**.

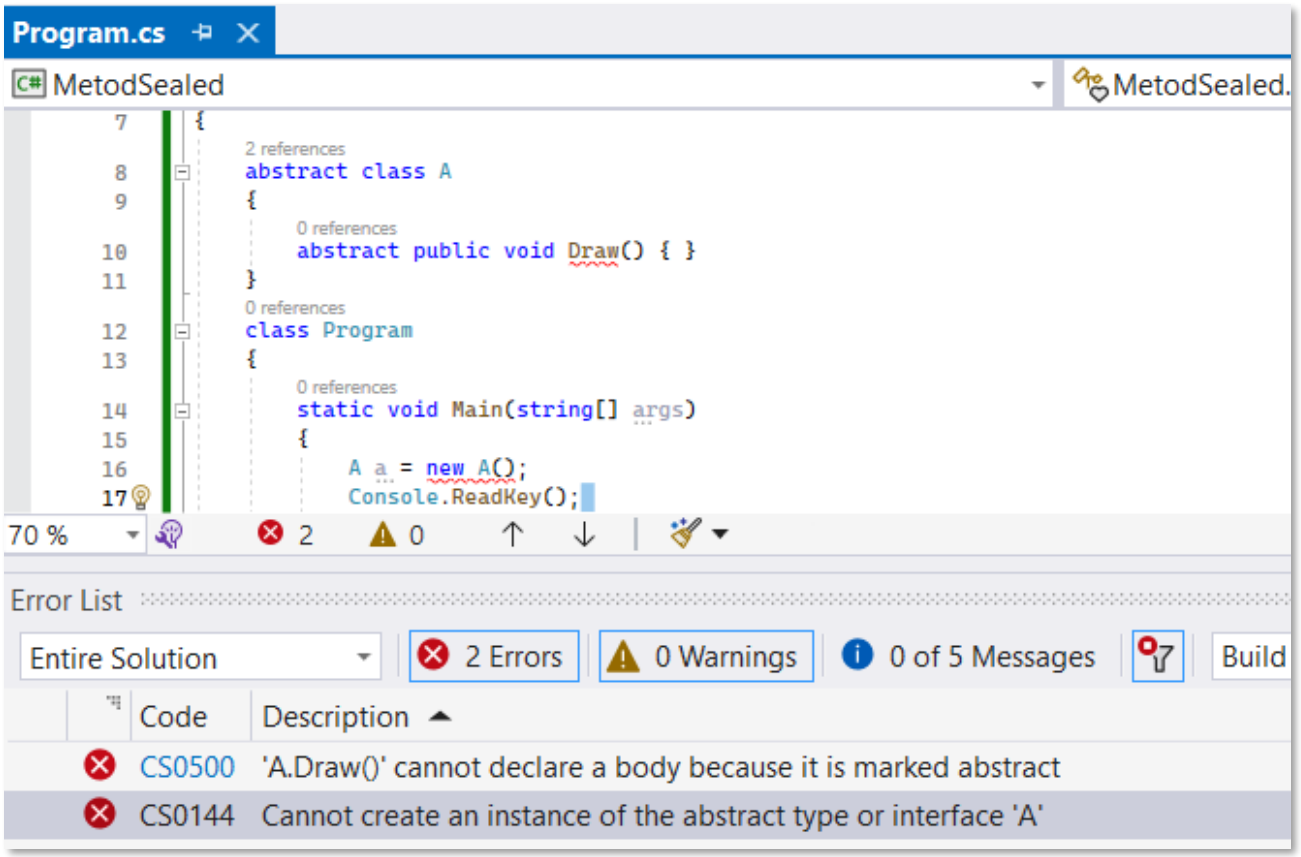
У **C#** можна домогтися цього з використанням ключового слова **abstract** у визначенні класу, таким чином створюючи абстрактний базовий клас. Клас, що містить абстрактні методи, називається абстрактним. Абстрактний метод створюється за допомогою модифікатора типу **abstract**. Абстрактний метод не містить тіла і, отже, не реалізується базовим класом. Тому похідний клас повинен його перевизначити, оскільки він не може використовувати версію, запропоновану в базовому класі. Неважко здогадатися, що абстрактний метод автоматично є віртуальним, тому і немає необхідності у використанні модифікатора **virtual**. Більш того, спільне використання модифікаторів **virtual** і

abstract вважається помилкою.

Для оголошення абстрактного методу використовуйте наступний формат запису:

abstract *ТИП ИМЯ*(*список_параметрів)*;

Модифікатор **abstract** можна використовувати тільки щодо звичайних, а не до **static**-методів. Клас, що містить один або кілька абстрактних методів, також повинен бути оголошений як абстрактний за допомогою специфікатора **abstract,** який ставиться перед оголошенням **class**. Оскільки абстрактний клас не реалізуємо в повному обсязі, то неможливо створити його екземпляри. Таким чином, спроба створити екземпляр абстрактного класу за допомогою оператора **new** призведе до виникнення помилки часу компіляції:



Якщо похідний клас виводиться з абстрактного, він може реалізувати всі абстрактні методи базового класу. В іншому випадку такий похідний клас також повинен бути визначений як абстрактний. Таким чином, атрибут **abstract** успадковується до тих пір, поки реалізація класу не буде повністю досягнута.

Ми розглядали з Вами приклад побудови ієрархії класів для зображення фігурки – кораблик. У базовому класі був оголошений віртуальний метод **Draw**(), який не мав операторів, так як за своєю суттю не міг бути виконаний.

Тому розумніше було зробити цей метод абстрактним і сам клас **figure** також.

```
abstract  class figure
{…
        abstract public void Draw();
}
class Trapezium: figure
{…
      override public void  Draw()
        {    …     }
}…
```

Результат: Програма буде працювати також, як з віртуальними методами

7.4 Клас Object

У C# визначений спеціальний клас з ім'ям **object**, який є неявним базовим класом всіх інших класів і типів (включаючи типи значень). Іншими словами, всі інші типи виводяться з класу **object**. Це означає, що посилальна змінна типу object може вказувати на об'єкт будь-якого типу. Крім того, оскільки C#-масиви реалізовані як класи, змінна типу **object** також може посилатися на будь-який масив. Строго кажучи, C# -ім'я **object** – ще одне ім'я для класу **System.Object**, який є частиною бібліотеки класів **.NET Framework**.

З урахуванням того, що **object** – базовий клас для всіх інших C#-типів і що приведення значення до об'єктного типу і його відновлення з об'єкта відбуваються автоматично, клас **object** можна використовувати в якості узагальненого типу даних. Наприклад, розглянемо наступну програму, яка створює масив об'єктів класу **object**, а потім привласнює його елементам дані різних типів.

Приклад 2

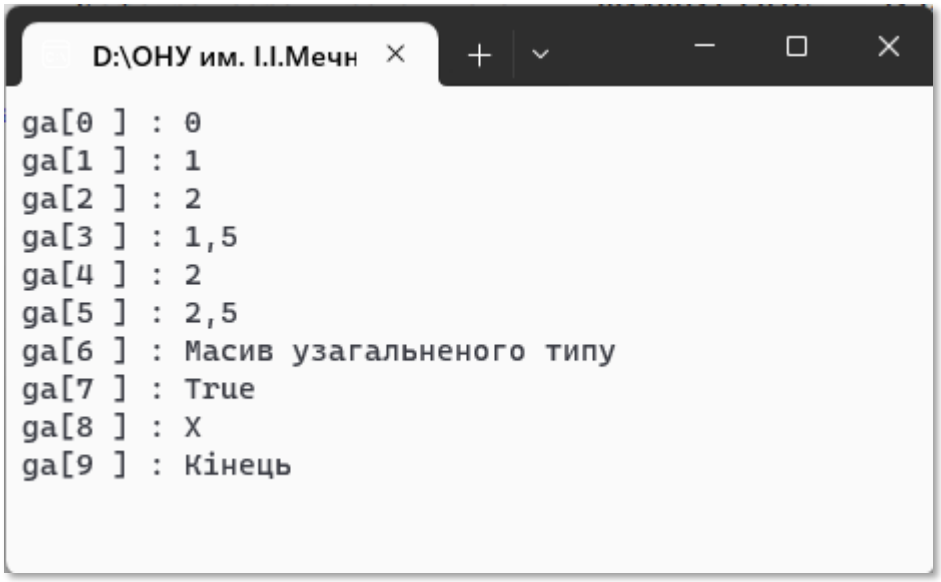
Program.cs

MetodSealed

MetodSealed.

```
0 references
class Program
{
    0 references
    static void Main(string[] args)
    {
        #region MyRegion
        object[] ga = new object[10];
        // Зберігаємо int-значення.
        for (int i = 0; i < 3; i++)
            ga[i] = i;
        // Зберігаємо double-значення.
        for (int i = 3; i < 6; i++)
            ga[i] = (double)i / 2;
        // Зберігаємо два рядки, bool- і char-значення,
        ga[6] = "Масив узагальненого типу";
        ga[7] = true;
        ga[8] = 'X';
        ga[9] = "Кінець";
        for (int i = 0; i < ga.Length; i++)
            Console.WriteLine("ga[" + i + " ] : " + ga[i] + " ");
        Console.ReadKey();
        #endregion
    }
}
```

Результат:



7.5 Правила приведення до базового і похідного класу

Тепер, коли Ви навчилися будувати сімейства взаємопов'язаних типів класів, слід ознайомитися з правилами, яким підкоряються операції приведення класів.

Перше правило приведення між типами класів свідчить, що коли два класи пов'язані ставленням "є", завжди можна безпечно зберегти похідний тип на засланні базового класу. Формально це називається неявним приведенням, оскільки воно "просто працює" відповідно до законів успадкування.

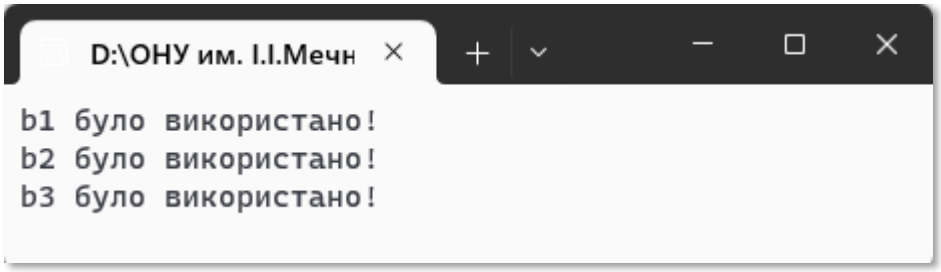
Припустимо, що в поточному класі **Program** визначено новий метод:

```
static void Function(body e)
        {
            Console.WriteLine("{0} було використано!", e.Name);
        }
```

Оскільки цей метод приймає єдиний параметр типу **body**, можна ефективно передавати цим методом будь-якого спадкоємця від класу **body**, з огляду на ставлення "є":

```
static void Main(string[] args)
        {
            body b1 = new body("b1");
            Function(b1);
            body b2 = new Square("b2", 100, 10, 10);
            Function(b2);
            Square b3 = new Square("b3", 0, 0, 0);
            Function(b3);
            Console.ReadKey();
        }
```

Результат:



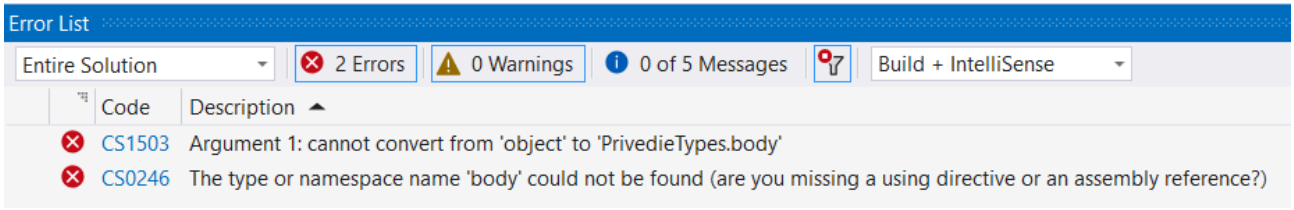
Код компілюється завдяки неявному приведенню від типу базового класу (**body**) до похідного класу. Але що, якщо також потрібно викликати метод **Function()** для екземпляра виду:

```
object b4 = new body("b4");
```

Примірник зберігається в даний момент в узагальненій посиланням **System.Object**, що дозволено з точки зору поліморфізму, так як даний клас є базовим для всіх типів. Якщо Ви передасте екземпляр **b4** безпосередньо в **Function ()**, як показано нижче:

```
Function(b4);
```

то отримаєте помилку на етапі компіляції:



Проблема в тому, що робиться спроба передати змінну, яка є не **body**, а більш загальним об'єктом **Object**. Оскільки в ланцюжку спадкування він знаходиться вище, ніж **body**, компілятор не допустить неявного приведення, намагаючись забезпечити максимально можливу безпеку типів. Незважаючи на те що Ви можете визначити, що об'єктна посилання вказує на сумісний з **body** клас в пам'яті, компілятор цього зробити не може, оскільки це не буде відомо аж до часу виконання. Щоб задовольнити компілятор, знадобиться виконати явне приведення.

Друге правило приведення говорить: необхідно явно виконувати приведення "вниз", використовуючи операцію приведення **C#**.

Базовий шаблон, якому потрібно слідувати при виконанні явного приведення, виглядає приблизно так:

(Клас_до_якого_треба_привести) існуюче_посилання

Таким чином, щоб передати змінну **object** методу **Function()**, буде

потрібно написати наступний код:

```
Function((body)b4);
```

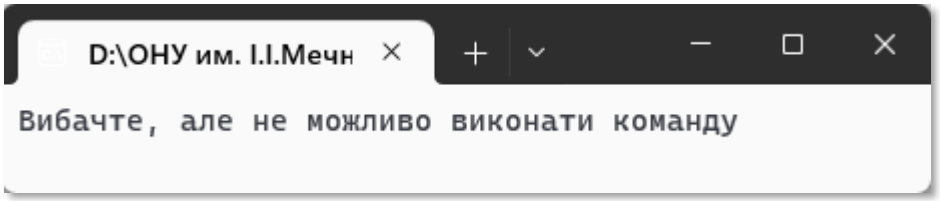
Пам'ятайте, що явне приведення відбувається під час виконання, а не на етапі компіляції, тому Ви несете відповідальність за можливе приведення типів і не виникнення помилок виконання коду.

```
body b1 = new body("b1");
Square bb = (Square)b1;
```

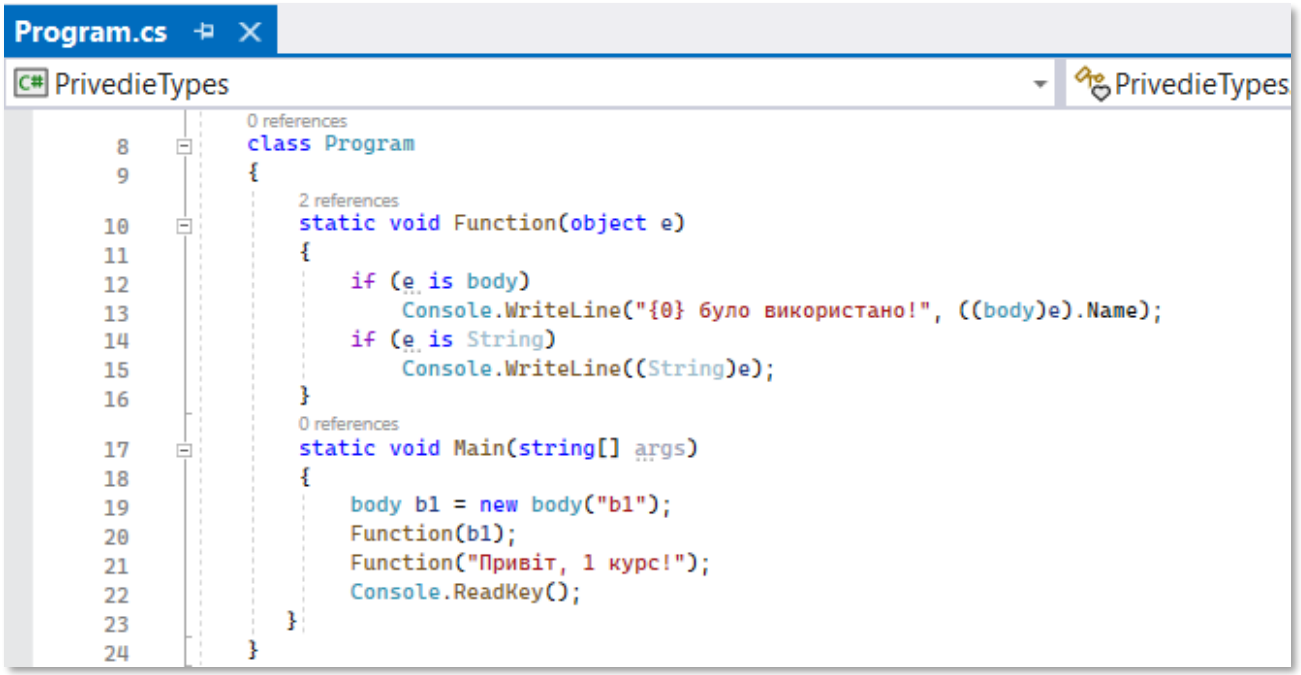
Зауваження. Надалі ми будемо розглядати детально обробку винятків.

C# надає ключове слово **as** для швидкого визначення сумісності одного типу з іншим під час виконання. За допомогою ключового слова **as** можна визначити сумісність, перевіривши повернене значення на рівність **null**. Погляньте на наступний код:

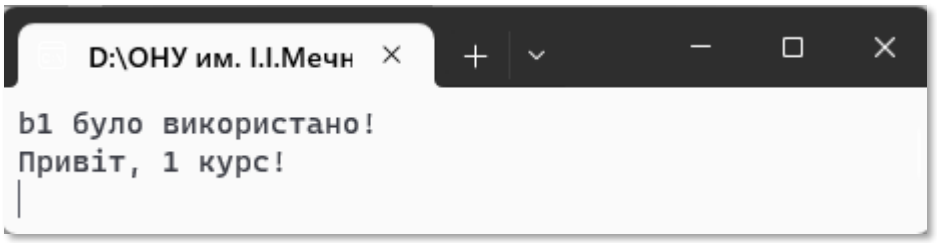
```
body b1 = new body("b1");
Square bb = b1 as Square;
if (bb == null)
Console.WriteLine("Вибачте, але не можливо виконати команду");
```



На додаток до ключового слова **as**, в **C#** пропонується ключове слово **is**, яке дозволяє визначити сумісність двох типів. На відміну від ключового слова **as**, якщо типи не сумісні, ключове слово **is** повертає **false**, а не посилання **null**.



Результат:



Зауваження. Тут під час виконання проводиться перевірка, на що саме в пам'яті вказує посилання вхідного параметра. Визначивши, що прийнятий екземпляр **body** або **String**, можна застосувати явне приведення і отримати доступ до спеціалізованих членів класу.

8. ІНТЕРФЕЙСИ

В об'єктно-орієнтованому програмуванні іноді потрібно визначити, що клас повинен робити, а не як він буде це робити. Ви вже бачили такий підхід на прикладі абстрактного методу. Абстрактний метод визначає сигнатуру для методу, але не забезпечує його реалізації. У похідному класі кожен абстрактний метод, визначений базовим класом, реалізується по-своєму. Таким чином, абстрактний метод задає інтерфейс для методу, але не спосіб його реалізації. Незважаючи на всю корисність абстрактних класів і методів, цю ідею можна розвинути. У **C#** передбачена можливість повністю відокремити інтерфейс класу від його реалізації за допомогою ключового слова **interface**.

Інтерфейси синтаксично подібні абстрактним класам. Однак в інтерфейсі жоден метод не може включати тіло, тобто інтерфейс в принципі не передбачає будь-якої його реалізації. Він визначає, що повинно бути зроблено, але не уточнює, як. Якщо вже інтерфейс визначено, його може реалізувати будь-яка кількість класів. При цьому один клас може реалізувати будь-яке число інтерфейсів.

Для реалізації інтерфейсу клас повинен забезпечити тіло (способи реалізації) методів, описаних в інтерфейсі. Кожен клас може визначити власну реалізацію. Таким чином, два класи можуть реалізувати один і той же інтерфейс різними способами, але все класи підтримують однаковий набір методів. Отже, код, «інформований» про наявність інтерфейсу, може використовувати об'єкти будь-якого класу, оскільки інтерфейс для всіх об'єктів

однаковий. Надаючи програмістам можливість застосування такого засобу програмування, як інтерфейс, **C#** дозволяє в повній мірі використовувати аспект поліморфізму, що виражається як "один інтерфейс – багато методів".

Інтерфейси оголошуються за допомогою ключового слова **interface**. Ось як виглядає спрощена форма оголошення інтерфейсу:

interface ім'я {
тип повернення ім'я метода1 (список параметрів);
тип повернення ім'я метода2 (список параметрів);
//. . .
тип повернення ім'я методаN (список параметрів);
}

Ім'я інтерфейсу задається елементом *ім'я*. Методи оголошуються з використанням лише типу значення, яке вони повертають, і сигнатури. Всі ці методи, по суті, абстрактні. Як згадувалося вище, для методів в інтерфейсі не передбачені способи реалізації. Отже, кожен клас, який включає інтерфейс, повинен реалізувати всі його методи. В інтерфейсі методи неявно є відкритими (**public**-методами), при цьому не дозволяється явно вказувати специфікатор доступу.

Розглянемо приклад інтерфейсу для класу, який генерує ряд чисел.

```
1 reference
9   public interface ISer
10  {
        4 references
11      int getNext(); // Повертає наступне число ряду,
        2 references
12      void reset(); // Виконує перезапуск,
        2 references
13      void setStart(int x); // Встановлює початкове значення.
14  }
```

Цей інтерфейс має ім'я **ISer**. Хоча префікс "**I**" необов'язковий, багато програмістів його використовують, щоб відрізняти інтерфейси від класів. Інтерфейс **ISer** оголошено відкритим, тому він може бути реалізований будь-яким класом в будь-якій програмі.

Крім сигнатур методів інтерфейси можуть оголошувати сигнатури властивостей, індексаторів і подій (про події буде розказано пізніше). Інтерфейси не можуть мати членів даних. Вони не можуть визначати конструктори, деструктори або операторні методи. Крім того, жоден член інтерфейсу не може бути оголошений статичним.

8.1 Реалізація інтерфейсів

Якщо інтерфейс визначено, один або кілька класів можуть його реалізувати. Щоб реалізувати інтерфейс, потрібно вказати його ім'я після імені класу подібно до того, як при створенні похідного вказується базовий клас. Формат запису класу, який реалізує інтерфейс, такий:

class ім'я класу: ім'я інтерфейсу

{ // тіло класу }

Якщо клас реалізує інтерфейс, він повинен це зробити в повному обсязі, тобто реалізація інтерфейсу не може бути виконана частково.

Класи можуть реалізувати кілька інтерфейсів. В цьому випадку імена інтерфейсів відокремлюються комами. Клас може успадковувати базовий клас і реалізувати один або кілька інтерфейсів. В цьому випадку список інтерфейсів має очолювати ім'я базового класу.

При реалізації інтерфейсу існує можливість множинного спадкоємства

```
public class A { }
public interface I1 { }
public interface I2 { }
public class B : A, I1, I2 { }
```

Методи, які реалізують інтерфейс, повинні бути оголошені відкритими. Справа в тому, що методи усередині інтерфейсу неявно оголошуються відкритими, тому їх реалізації також повинні бути відкритими. Крім того, сигнатура типу в реалізації методу повинна в точності збігатися з сигнатурою типу, заданою у визначенні інтерфейсу.

Приклад 1

ByT.cs | ISer.cs | Program.cs

ConsoleApp3 | ConsoleApp3

```
3 references
class ByT: ISer
{
    int start;
    int val;
    1 reference
    public ByT()
    { start = 0; val = 0; }
    4 references
    public int getNext()
    { val += 2; return val; }
    2 references
    public void reset()
    { val = start; }
    2 references
    public void setStart(int x)
    { start = x; val = start;}
}
```

Це приклад реалізації інтерфейсу **ISer,** оголошення якого наведено вище. Створюється клас з ім'ям **ByT**, що генерує ряд чисел, в якому кожне наступне число більше попереднього на два. Як бачите, клас **ByT** реалізує всі три методи, визначені інтерфейсом **ISer**. Інакше і бути не може, оскільки класу забороняється створювати часткову реалізацію інтерфейсу.

Розглянемо приклад, який демонструє використання класу **ByT**:

Program.cs | ByT.cs | ISer.cs

ConsoleApp3 | ConsoleApp3.

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        ByT ob = new ByT();
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.getNext());
        Console.WriteLine("Перехід у вихідний стан.");
        ob.reset();
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.getNext());
        Console.WriteLine("Починаємо з числа 100.");
        ob.setStart(100);
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.getNext());
        Console.ReadLine();
    }
}
```

Результат:

```
D:\ОНУ им. И.И.Ме
Наступне значення дорівнює 2
Наступне значення дорівнює 4
Наступне значення дорівнює 6
Наступне значення дорівнює 8
Наступне значення дорівнює 10
Перехід у вихідний стан.
Наступне значення дорівнює 2
Наступне значення дорівнює 4
Наступне значення дорівнює 6
Наступне значення дорівнює 8
Наступне значення дорівнює 10
Починаємо з числа 100.
Наступне значення дорівнює 102
Наступне значення дорівнює 104
Наступне значення дорівнює 106
Наступне значення дорівнює 108
Наступне значення дорівнює 110
```

У класах, які реалізують інтерфейси, можна визначати додаткові члени. Наприклад, в представлену нижче версію класу **ByT** доданий метод **getPr()**, який повертає попереднє значення ряду.

Реалізація інтерфейсу **ISer** з додатково визначеним методом **getPr().**

Приклад 2

ByT.cs | Program.cs | ISer.cs

ConsoleApp3 | ConsoleApp3.

```
3 references
class ByT: ISer
{
    int start;
    int val;
    int prev;
    1 reference
    public ByT()
    {
        start = 0;
        val = 0;
        prev = -2;
    }
    4 references
    public int getNext()
    {
        prev = val;
        val += 2;
        return val;
    }
    2 references
    public void reset()
    {
        val = start;
        prev = start - 2;
    }
    2 references
    public void setStart(int x)
    {
        start = x;
        val = start;
        prev = val - 2;
    }
    // Метод, не оголошений в інтерфейсі ISer
    0 references
    public int getPr()
    {
        return prev;
    }
}
```

Зверніть увагу на те, що додавання методу **getPr()** потребувало внесення змін до реалізації методів, визначених інтерфейсом **ISer**. Але оскільки інтерфейс для цих методів залишається колишнім, при зміні не руйнується код, написаний раніше. В цьому і полягає одна з переваг використання інтерфейсів. Як згадувалося вище, інтерфейс може реалізувати будь-яку кількість класів.

Розглянемо, наприклад, клас **Pr**, який генерує ряд простих чисел. Зверніть увагу на те, що його спосіб реалізації інтерфейсу **ISer** в корені відрізняється від використовуваного класом **ByT**.

Приклад 3. Використання інтерфейсу **ISer** для реалізації ряду простих чисел.

Pr.cs | ByT.cs | Program.cs | ISer.cs

ConsoleApp3 | ConsoleApp3.

```
1 reference
public class Pr: ISer
{
    int start;
    int val;
    0 references
    public Pr()
    {
        start = 2;
        val = 2;
    }
    0 references
    public int getNext()
    {
        int i, j;
        bool ispr;
        val++;
        for (i = val; i < 1000000; i++)
        {
            ispr = true;
            for (j = 2; j < (i / 2 + 1); j++)
            {
                if ((i % j) == 0)
                {  ispr = false; break; }
            }
            if (ispr)
            {
                val = i; break;
            }
        }
        return val;
    }
    0 references
    public void reset()  { val = start; }
    0 references
    public void setStart(int x) { start = x; val = start; }
    }
}
```

Тут важливо розуміти, що, хоча класи **Pr** і **ByT** генерують різні ряди чисел, обидва вони реалізують один і той самий інтерфейс **ISer**. І в цьому немає нічого дивного, оскільки кожен клас вільний вирішити цю задачу так, як "вважає" за потрібне. Використання класу реалізації має вигляд:

Program.cs | Pr.cs | ByT.cs | ISer.cs

ConsoleApp3 | ConsoleApp3.P

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        Pr prOb = new Pr();
        for (int i = 0; i < 15; i++)
            Console.WriteLine("Наступне просте число дорівнює " + prOb.getNext());
    }
}
```

8.2 Використання інтерфейсних посилань

Можна створити змінну-посилання на інтерфейс. Така змінна може посилатися на будь-який об'єкт, який реалізує її інтерфейс. При виклику методу для об'єкта за допомогою інтерфейсного посилання буде виконана та версія зазначеного методу, яка реалізована цим об'єктом. Цей процес аналогічний використанню посилання на базовий клас для доступу до об'єкта похідного класу.

ISer.cs | Program.cs | Pr.cs | ByT.cs

ConsoleApp3 | ConsoleApp3

```
1 reference
public interface ISer
{
    1 reference
    int getNext(); // Повертає наступне число ряду,
    1 reference
    void reset(); // Виконує перезапуск,
    1 reference
    void setStart(int x); // Встановлює початкове значення.
}
```

Приклад 4 Використання інтерфейсного посилання

ByT.cs | ISer.cs | Program.cs | Pr.cs

ConsoleApp3 | ConsoleApp3.

```
1 reference
class ByT: ISer
{
    int start;
    int val;
    0 references
    public ByT()
    {
        start = 0; val = 0;
    }
    1 reference
    public int getNext()
    {
        val += 2; return val;
    }
    1 reference
    public void reset()
    {
        val = start;
    }
    1 reference
    public void setStart(int x)
    {
        start = x; val = start;

    }
}
```

Pr.cs | ByT.cs | ISer.cs | Program.cs

ConsoleApp3

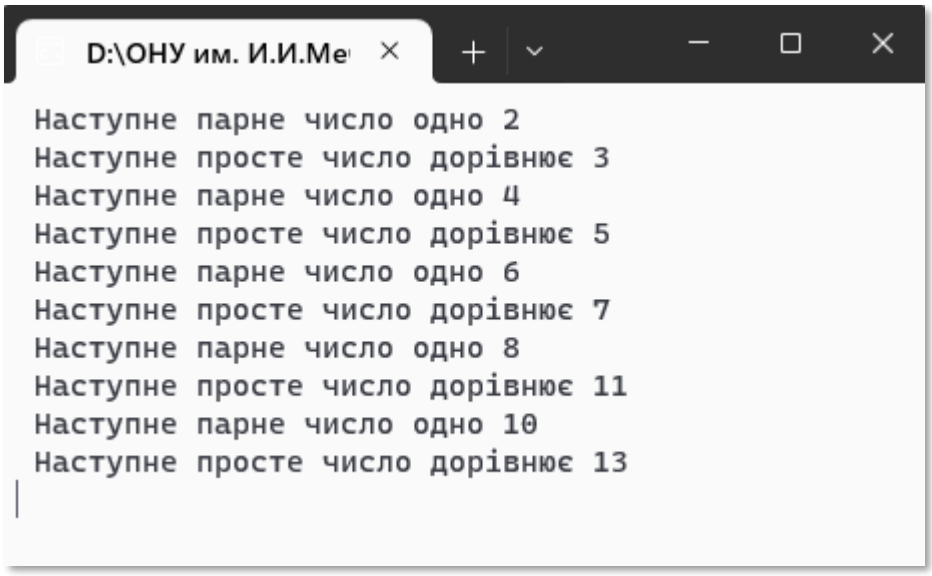
```
3 references
public class Pr: ISer
{
    int start;
    int val;
    1 reference
    public Pr()
    {
        start = 2;
        val = 2;
    }
    3 references
    public int getNext()
    {
        int i, j; bool ispr;
        val++;
        for (i = val; i < 1000000; i++)
        { ispr = true;
            for (j = 2; j < (i / j + 1); j++)
            {
                if ((i % j) == 0)
                { ispr = false; break; }
            }
            if (ispr)
            {   val = i; break; }
        }
        return val;
    }
    1 reference
    public void reset() { val = start; }
    1 reference
    public void setStart(int x) { start = x; val = start; }
}
```

Program.cs | Pr.cs | ByT.cs | ISer.cs

ConsoleApp3

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        ByT tOb = new ByT();
        Pr prOb = new Pr();
        ISer ob;
        for (int i = 0; i < 5; i++)
        {
            ob = tOb;
            Console.WriteLine(" Наступне парне число одно " + ob.getNext());
            ob = prOb;
            Console.WriteLine(" Наступне просте число дорівнює " +
            ob.getNext());
        }
        Console.ReadKey();
    }
}
```

Результат:



У методі **Main()** оголошується змінна **ob** як посилання на інтерфейс **ISer**. Це означає, що її можна використовувати для зберігання посилань на будь-який об'єкт, який реалізує інтерфейс **ISer**. У цьому випадку вона слугує для посилання на об'єкти **tOb** і **prOb**, які є екземплярами класів **ByT** і **Pr**, відповідно, причому обидва класи реалізують один і той самий інтерфейс, **ISer**.

8.3 Інтерфейсні властивості

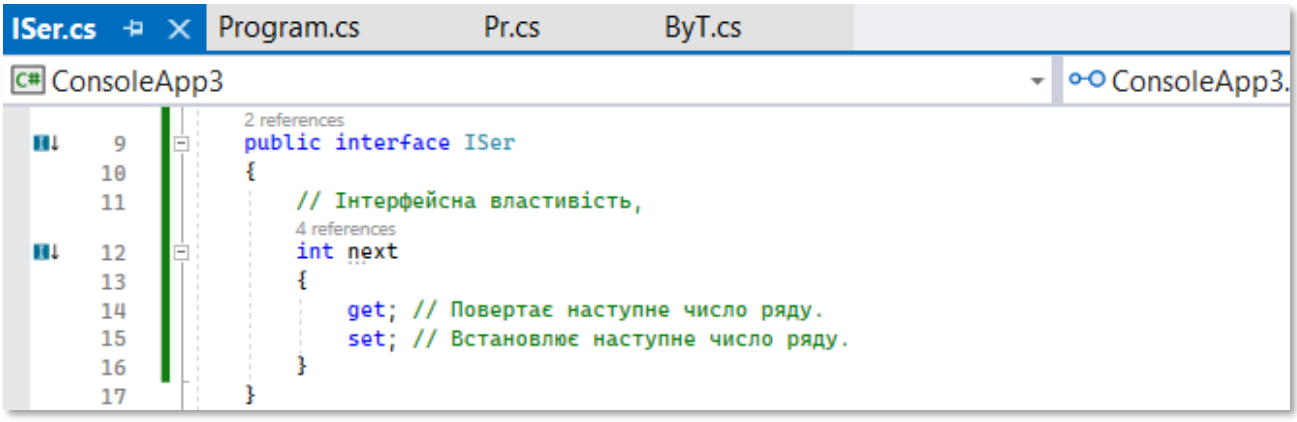
Як і методи, властивості визначаються в інтерфейсі без тіла. Нижче наведено формат специфікації властивості.

// інтерфейсна властивість
тип ім'я { get;
set; }

Властивості, призначені тільки для читання або тільки для запису, містять лише **get**- або **set**-елемент, відповідно.

Розглянемо ще одну версію інтерфейсу **ISer** і класу **ByT**, в якому для отримання наступного елемента ряду і його установки використовується властивість.

Приклад 5



ByT.cs | ISer.cs | Program.cs | Pr.cs

ConsoleApp3

```
3 references
class ByT: ISer
{
    int val;
    1 reference
    public ByT()
    {
        val = 0;
    }
    // Отримуємо або встановлюємо значення ряду
    4 references
    public int next
    {
        get { val += 2; return val; }
        set { val = value; }
    }
}
```

Program.cs | ByT.cs | ISer.cs | Pr.cs

ConsoleApp3

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        ByT ob = new ByT();
        // Отримуємо доступ до ряду через властивість.
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.next);
        Console.WriteLine("Починаємо з числа 21");
        ob.next = 21;
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.next);
        Console.ReadKey();
    }
}
```

Результат:

D:\ОНУ им. И.И.Ме

```
Наступне значення дорівнює 2
Наступне значення дорівнює 4
Наступне значення дорівнює 6
Наступне значення дорівнює 8
Наступне значення дорівнює 10
Починаємо з числа 21
Наступне значення дорівнює 23
Наступне значення дорівнює 25
Наступне значення дорівнює 27
Наступне значення дорівнює 29
Наступне значення дорівнює 31
```

8.4 Інтерфейсні індексатори

В інтерфейсі можна визначити і індексатор. Оголошення індексатора в інтерфейсі має наступний формат запису:

// інтерфейсний індексатор
тип элементу this[int індекс]
{ get;
set; }

Індексатори, призначені тільки для читання або тільки для запису, містять лише **get**- або **set**-метод, відповідно.

Приклад 6. В цьому прикладі доданий індексатор, призначений тільки для читання елемента ряду.

ISer.cs | Program.cs | ByT.cs | Pr.cs

ConsoleApp3 | ConsoleApp3.

```
1 reference
public interface ISer
{
    // Інтерфейсна властивість,
    5 references
    int next
    {
        get; // Повертає наступне число ряду.
        set; // Встановлює наступне число ряду.
    }
    // Інтерфейсний індексатор
    2 references
    int this[int index]
    {
        get; // Повертає заданий член ряду.
    }
}
```

ByT.cs | ISer.cs | Program.cs | Pr.cs

ConsoleApp3 | ConsoleApp3.

```
3 references
class ByT: ISer
{
    int val;
    1 reference
    public ByT()
    {
        val = 0;
    }
    // Отримуємо або встановлюємо значення за допомогою властивості
    5 references
    public int next
    {
        get { val += 2; return val; }
        set { val = value; }
    }
    // Отримуємо значення за допомогою індексатора
    2 references
    public int this[int index]
    {
        get
        {
            val = 0;
            for (int i = 0; i < index; i++)
                val += 2;
            return val;
        }
    }
}
```

```
Program.cs  ByT.cs  ISer.cs  Pr.cs
ConsoleApp3                                   ConsoleApp3.
```

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        ByT ob = new ByT();
        // Отримуємо доступ до ряду за допомогою властивості
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.next);
        Console.WriteLine("Починаємо з числа 21");
        ob.next = 21;
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob.next);
        Console.WriteLine("Перехід у вихідний стан.");
        ob.next = 0;
        // Отримуємо доступ до ряду за допомогою індексатора.
        for (int i = 0; i < 5; i++)
            Console.WriteLine("Наступне значення дорівнює " + ob[i]);
    Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. И.И.Ме
Наступне значення дорівнює 2
Наступне значення дорівнює 4
Наступне значення дорівнює 6
Наступне значення дорівнює 8
Наступне значення дорівнює 10
Починаємо з числа 21
Наступне значення дорівнює 23
Наступне значення дорівнює 25
Наступне значення дорівнює 27
Наступне значення дорівнює 29
Наступне значення дорівнює 31
Перехід у вихідний стан.
Наступне значення дорівнює 0
Наступне значення дорівнює 2
Наступне значення дорівнює 4
Наступне значення дорівнює 6
Наступне значення дорівнює 8
```

8.5 Спадкування інтерфейсів

Один інтерфейс може успадкувати "багатство" іншого. Синтаксис цього механізму аналогічний синтаксису, що використовується для наслідування класів. Якщо клас реалізує інтерфейс, який успадковує інший інтерфейс, цей клас повинен забезпечити способи реалізації для всіх членів, визначених всередині ланцюжка спадкування інтерфейсів. Розглянемо такий приклад:

Приклад 7

A.cs | B.cs | MyClass.cs | Program.cs

ConsoleApp1

```
1 reference
public interface A
{
    2 references
    void meth1();
    2 references
    void meth2();
}
```

B.cs | A.cs | MyClass.cs | Program.cs

ConsoleApp1

```
// Інтерфейс B тепер включає методи meth1() і meth2(),
// а також додає метод meth3().
1 reference
public interface B: A
{
    2 references
    void meth3();
}
```

MyClass.cs | B.cs | A.cs | Program.cs

ConsoleApp1

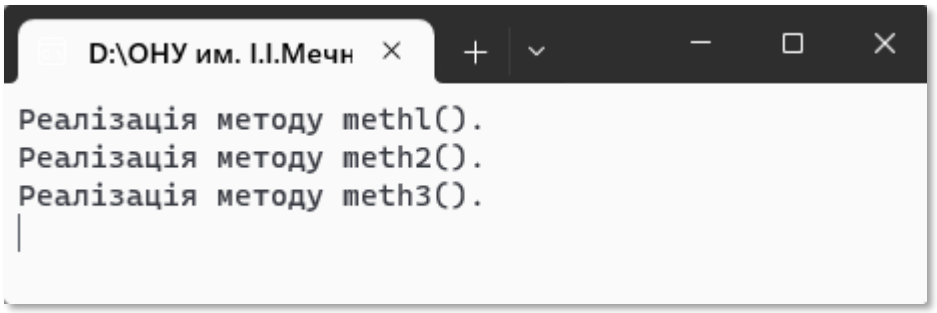
```
2 references
public class MyClass: B
{
    2 references
    public void meth1()
    {
        Console.WriteLine("Реалізація методу meth1().");
    }
    2 references
    public void meth2()
    {
        Console.WriteLine("Реалізація методу meth2().");
    }
    2 references
    public void meth3()
    {
        Console.WriteLine("Реалізація методу meth3().");
    }
}
```

Program.cs | MyClass.cs | B.cs | A.cs

ConsoleApp1

```
0 references
static void Main(string[] args)
{
    MyClass ob = new MyClass();
    ob.meth1();
    ob.meth2();
    ob.meth3();
    Console.ReadKey();
}
```

Результат:



Якби в якості експерименту ви спробували видалити метод **methl()**, реалізований у класі **MyClass**, то відразу ж отримали б від компілятора повідомлення про помилку. Як згадувалося вище, будь-який клас, який реалізує інтерфейс, повинен реалізувати всі методи, визначені цим інтерфейсом, включно з методами, які успадковані від інших інтерфейсів.

8.6 Приховування імен за допомогою успадкування інтерфейсів

У похідному інтерфейсі можна оголосити член, який приховує член, певний в базовому інтерфейсі. Це відбувається при збігу їх сигнатур. Такий збіг викличе попередження, якщо член похідного інтерфейсу не модифікувати за допомогою ключового слова **new**.

8.7 Явна реалізація членів інтерфейсу

При реалізації члена інтерфейсу можна кваліфікувати його ім'я з використанням імені інтерфейсу. У цьому випадку говорять, що член інтерфейсу реалізується явно, або має місце його явна реалізація.

Наприклад, при визначенні інтерфейсу

```
interface IMyIF
{   int myMeth(int x ) ; }
```

цілком допустимо реалізувати інтерфейс **IMyIF** наступним чином:

```
class MyClass : IMyIF
{
    int IMyIF.myMeth(int x)
  {   return x / 3;}
}
```

Як бачите, при реалізації методу **myMeth()** члена інтерфейсу **IMyIF** вказується його повне ім'я, яке включає ім'я інтерфейсу.

Явна реалізація членів інтерфейсу може знадобитися з двох причин. По-перше, реалізуючи метод з використанням повністю кваліфікованого імені, ви тим самим позначаєте частини закритої реалізації, які не «видно» коду, визначеного поза класом. По-друге, клас може реалізувати два інтерфейси, які оголошують методи з однаковими іменами і типами. Повна кваліфікація імен дозволяє уникнути невизначеності ситуації.

8.8 Закрита реалізація

Наступна програма містить інтерфейс з ім'ям **IEven**, який визначає два методи **isEven()** і **isOdd()**, що встановлюють факт парності і непарності числа, відповідно. Клас **MyClass** реалізує інтерфейс **IEven**, причому його член **isOdd()** реалізується явно.

Приклад 8

IEven.cs* | Program.cs | MyClass.cs | B.cs | A.cs

ConsoleApp1

```
3 references
public interface IEven
{
    2 references
    bool isOdd(int x);
    2 references
    bool isEven(int x);
}
```

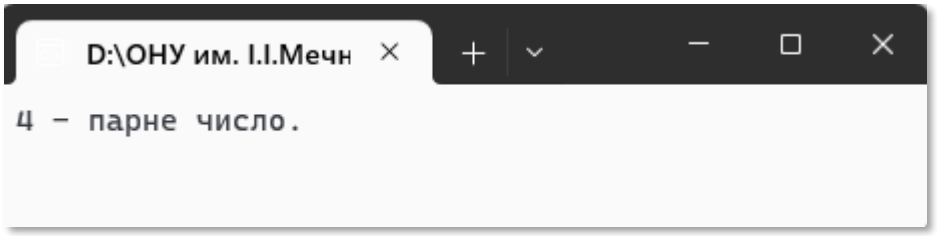
MyClass.cs | IEven.cs* | Program.cs | B.cs | A.cs

ConsoleApp1

```
2 references
public class MyClass: IEven
{
    // Явна реалізація
    2 references
    bool IEven.isOdd(int x)
    {
        if ((x % 2) != 0) return true;
        else return false;
    }
    // Звичайна реалізація
    2 references
    public bool isEven(int x)
    {
        IEven o = this; // Посилання на об'єкт, що викликає
        return !o.isOdd(x);
    }
}
```

```
Program.cs | MyClass.cs | IEven.cs* | B.cs | A.cs
ConsoleApp1                                   ConsoleApp1.
    0 references
9   public class Program
10  {
        0 references
11      static void Main(string[] args)
12      {
13          MyClass ob = new MyClass();
14          bool result;
15          result = ob.isEven(4);
16          if (result) Console.WriteLine("4 - парне число.");
17          else Console.WriteLine("3 - непарне число.");
18          // result = ob.isOdd(); // Помилка, член не видно.
19      }
20  }
```

Результат:



Оскільки метод **isOdd()** реалізовано в явному вигляді, він недоступний поза класом **MyClass**. Такий спосіб реалізації робить його надійно закритим. Усередині класу **MyClass** до методу **isOdd()** можна отримати доступ тільки через посилання на інтерфейс. Ось чому він прекрасно викликається для об'єкта **ob** в реалізації методу **isEven().**

8.9 Як уникнути невизначеності за допомогою явної реалізації

Розглянемо приклад, в якому реалізовано два інтерфейси, причому обидва оголошують метод з ім'ям **meth()**. У цій ситуації явна реалізація використовується для того, щоб уникнути невизначеності.

Приклад 9

MyClass.cs | Program.cs* | IEven.cs | B.cs | A.cs

ConsoleApp1 | ConsoleApp1.M

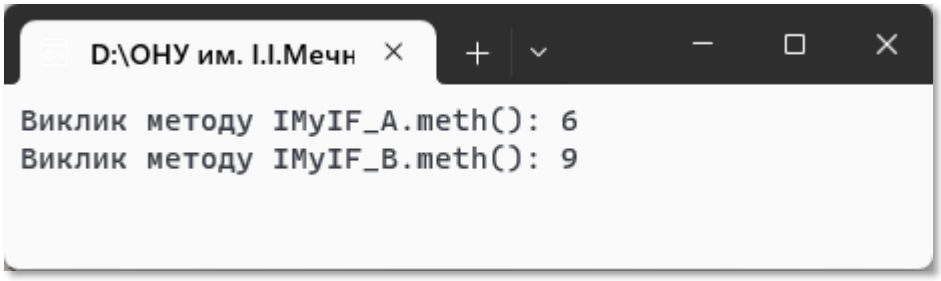
```
3 references
interface IMyIF_A
{
    2 references
    int meth(int x);
}
3 references
interface IMyIF_B
{
    2 references
    int meth(int x);
}
// У класі MyClass реалізовано обидва інтерфейси
2 references
public class MyClass : IMyIF_A, IMyIF_B
{
    // Явним чином реалізуємо два методи meth().
    2 references
    int IMyIF_A.meth(int x)
    {
        return x + x;
    }
    2 references
    int IMyIF_B.meth(int x)
    {
        return x * x;
    }
    // Викликаємо метод meth() за допомогою посилання на інтерфейс,
    1 reference
    public int methA(int x)
    {
        IMyIF_A a_ob;
        a_ob = this;
        return a_ob.meth(x); // Имеется в виду интерфейс IMyIF_A.
    }
    1 reference
    public int methB(int x)
    {
        IMyIF_B b_ob;
        b_ob = this;
        return b_ob.meth(x); // Имеется в виду интерфейс IMyIF_B
    }
}
```

Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1 | ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        MyClass ob = new MyClass();
        Console.Write("Виклик методу IMyIF_A.meth(): ");
        Console.WriteLine(ob.methA(3));
        Console.Write("Виклик методу IMyIF_B.meth(): ");
        Console.WriteLine(ob.methB(3));
    }
```

Результат:



Зверніть увагу на те, що метод **meth()** має однакову сигнатуру в інтерфейсах **lMylF_A** і **lMylF_B**. Отже, якщо клас **MyClass** реалізує обидва ці інтерфейси, він має реалізувати кожен метод окремо, повністю вказавши його ім'я (з використанням імені відповідного інтерфейсу). Оскільки єдиний спосіб виклику явно заданого методу полягає у використанні інтерфейсного посилання, метод **meth(),** оголошений в інтерфейсі **IMyIF_A**, створює посилання на інтерфейс **IMylF_A**, а метод **meth(),** оголошений в інтерфейсі **IMylF_B**, створює посилання на інтерфейс **lMyiF_B**. Створені посилання потім використовуються під час виклику цих методів, завдяки чому можна уникнути неоднозначності.

8.10 Вибір між інтерфейсом і абстрактним класом

У програмуванні на **C#** за необхідності описати функції, а не спосіб їхньої реалізації, важливо знати, коли слід використовувати інтерфейс, а коли – абстрактний клас. Загальне правило таке. Якщо ви повністю описуєте дії класу і не потрібно уточнювати, як він це робить, слід використовувати інтерфейс. Якщо ж потрібно включити в опис деталі реалізації, має сенс представити концепцію програми (або її частини) у вигляді абстрактного класу.

9. ОБРОБКА ВИНЯТКОВИХ СИТУАЦІЙ

Виняткова ситуація (або виключення) – це помилка, яка виникає під час виконання програми. Використовуючи **C#**-підсистему обробки виняткових ситуацій, з такими помилками можна справлятися.

Перевага підсистеми обробки виключень полягає в автоматизації створення більшої частини коду, який раніше необхідно було вводити в програми ***"вручну"***. Наприклад, в будь-якій комп'ютерній мові при відсутності такої підсистеми практично кожен метод повертав коди помилок, і ці значення

перевірялися вручну при кожному виклику методу. Такий підхід досить обтяжливий, крім того, при цьому можливе виникнення помилок. Обробка винятків спрощує ***"роботу над помилками"***, дозволяючи в програмах визначати блок коду, іменований оброблювачем виключення, який буде автоматично виконуватися при виникненні певної помилки. В цьому випадку не обов'язково перевіряти результат виконання кожної конкретної операції або методу вручну. Якщо помилка виникне, її належним чином обробить обробник виключень. Ще однією перевагою обробки виняткових ситуацій в **C#** є визначення стандартних винятків для таких поширених програмних помилок, як поділ на нуль або потрапляння поза діапазоном дії визначення індексу. Щоб відреагувати на виникнення таких помилок, програма повинна відстежувати і обробляти ці винятки.

Без знання можливостей **C#**-підсистеми обробки винятків успішне програмування на **C#** попросту неможливо.

9.1 Клас System.Exception

У **C#** виключення представляються класами. Всі класи виключень повинні бути виведені з вбудованого класу винятків **Exception**, який є частиною простору імен **System**. З класу **Exception** виведені класи **SystemException** і **ApplicationException**. Вони підтримують дві загальні категорії винятків, певні в **C#**: ті, які генеруються **C#**-системою динамічного управління, або загальномовним засобом управління (**Common Language Runtime - CLR**), і ті, які генеруються прикладними програмами. Але ні клас **SystemException**, ні клас **ApplicationException** не привносить нічого нового на додаток до членів класу **Exception**. Вони просто визначають вершини двох різних ієрархій класів винятків.

C# визначає вбудовані виключення, які виводяться з класу **SystemException**. Наприклад, при спробі виконати поділ на нуль генерується виключення класу **DivideByZeroException**.

Управління C#-механізмом обробки винятків ґрунтується на чотирьох ключових словах: **try, catch, throw** і **finally**. Вони утворюють взаємопов'язану підсистему, в якій використання одного з них передбачає використання іншого. Їх робота полягає в наступному: програмні інструкції, які потрібно проконтролювати на предмет винятків, поміщаються в **try**-блок. Якщо

виключення виникає в цьому блоці, то з'являється певного роду інформація. Цей виняток може бути перехоплено програмним шляхом за допомогою **catch**-блоку і оброблено відповідним чином. Системні виключення автоматично генеруються **C#**-системою динамічного управління. Щоб згенерувати виключення вручну, використовується ключове слово **throw**. Будь-який код, який повинен бути обов'язково виконаний при виході з **try**-блоку, поміщається в блок **finally**.

Ядром обробки винятків є блоки **try** і **catch**. Ці ключові слова працюють разом; не можна використовувати слово **try** без **catch** або **catch** без **try**. Синтаксис запису **try / catch**-блоків обробки винятків має вигляд:

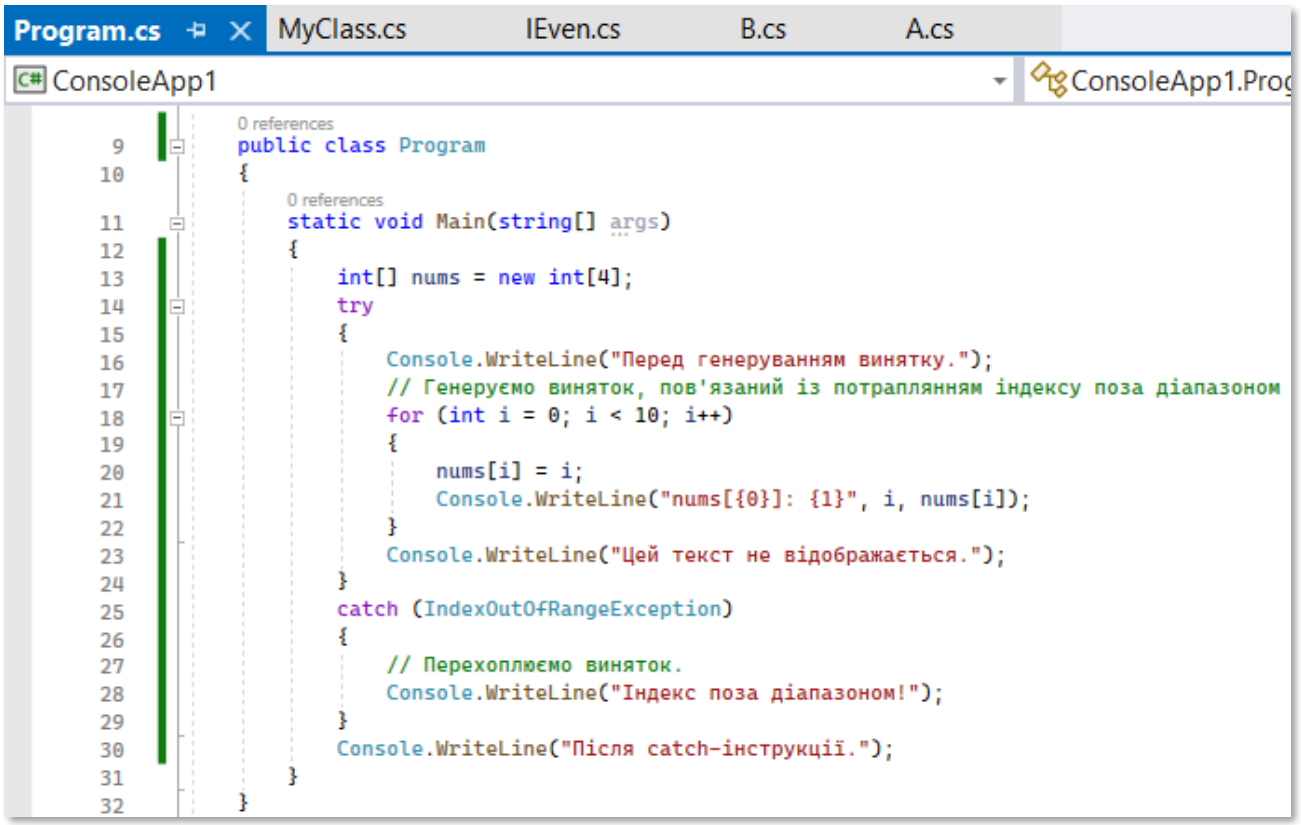
```
try {
// Блок коду, який підлягає перевірці на наявність помилок.
}
catch (ExcepType1  exOb) {
// Оброблювач для виключення типу ExcepType1.
}
catch (ExcepType2 exOb) {
// Оброблювач для виключення типу ExcepType2.
}...
```

Тут ***ExcepType*** – це тип згенерованого виключення. Після генерування помилки виключення перехоплюється відповідною інструкцією **catch**, яка його обробляє. Як видно з формату запису **try / catch**-блоків, з **try**-блоком може бути пов'язана не одна, а кілька **catch**-інструкцій. Яка саме з них буде виконана, визначить тип винятку. Іншими словами, буде виконана та **catch**-інструкція, тип виключення якої збігається з типом згенерованого виключення (а всі інші будуть проігноровані). Після перехоплення виключення параметр **exOb** прийме його значення. Задавати параметр **exOb** необов'язково. Якщо оброблювачу виключення не потрібен доступ до об'єкта виключення (як це часто буває), в завданні параметра **exOb** немає необхідності.

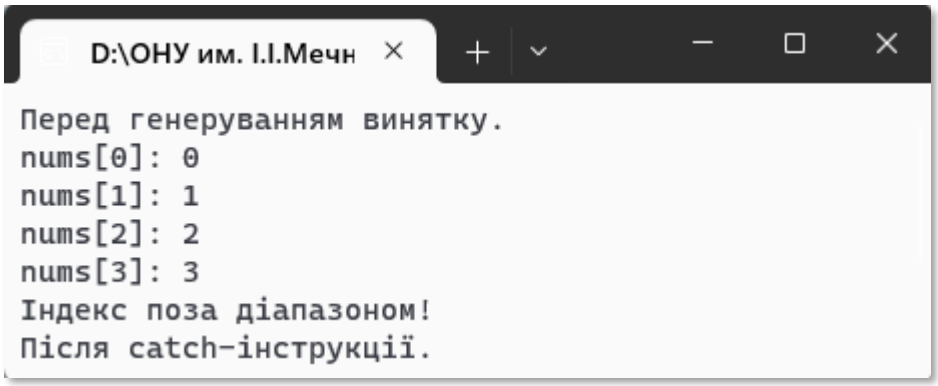
Важливо розуміти наступне. Якщо виключення не генерується, **try**-блок завершується нормально, і все його **catch**-інструкції ігноруються. Виконання програми продовжується з першої інструкції, яка стоїть після останньої інструкції **catch**. Таким чином, **catch**-інструкція (із запропонованих після **try**-блоку) виконується тільки в разі, якщо згенеровано відповідне виключення.

Розглянемо простий приклад, який демонструє, як відстежити і перехопити виняток. Відомо, що спроба індексувати масив за межами його кордонів викликає помилку порушення діапазону. В цьому випадку **C#**-система динамічного управління генерує виняток типу **IndexOutOfRangeException**, яке представляє собою стандартний виняток, визначений мовою **C#**. У наступній програмі такий виняток навмисно генерується, а потім перехоплюється.

Пример 1



Результат:



Зверніть увагу на те, що **nums** – це **int**-масив для зберігання чотирьох елементів. Однак в циклі **for** робиться спроба індексувати цей масив від 0 до 9, і як тільки значення індексу встановлюється рівним чотирьом, генерується

виключення типу **IndexOutOfRangeException**. Незважаючи на невеликий розмір, попередня програма ілюструє ряд ключових аспектів обробки винятків. По-перше, код, що перевіряється, міститься всередині **try**-блоку. По-друге, при виникненні виключення (в даному випадку через спроби всередині **for**-циклу індексувати масив **nums** за межами його діапазону) виконання **try**-блоку припиняється, а саме виключення перехоплюється **catch**-інструкцією. Іншими словами, управління програмою передається **catch**-інструкції, незалежно від того, чи всі інструкції **try**-блоку виконалися. При цьому важливо те, що інструкція **catch** не викликається, а їй передається керування програмою. Тому інструкція

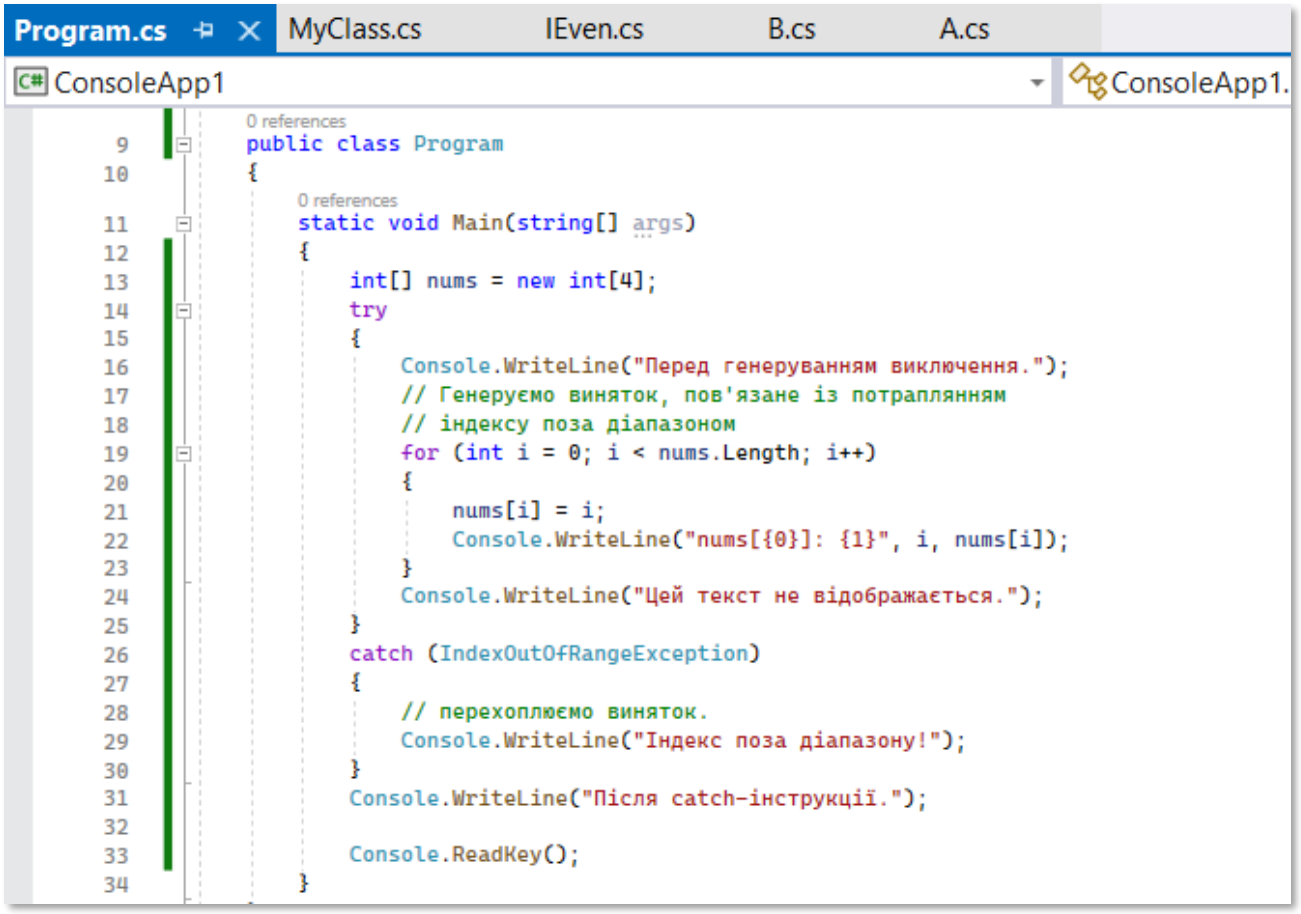
```
Console.WriteLine("Цей текст не відображається.");
```

ніколи не виконається. Після виконання **catch**-інструкції програма продовжиться з наступної інструкції. Отже, щоб ваша програма могла нормально продовжити своє виконання, обробник повинен усунути проблему, яка стала причиною виникнення виняткової ситуації.

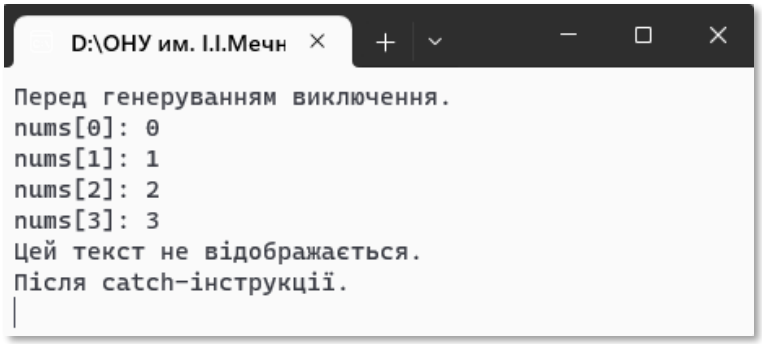
Зверніть увагу на те, що в інструкції **catch** параметр відсутній. Як згадувалося вище, параметр необхідний тільки в тих випадках, коли потрібен доступ до об'єкта виключення. У деяких випадках значення об'єкта виключення використовується оброблювачем для отримання додаткової інформації про помилку, але частіше за все досить просто знати про те, що виключення мало місце. Тобто у випадках, коли вам просто потрібно виявити те, що виняток стався, і вам не потрібно додаткової інформації з самого об'єкта виключення, використання блоку **catch** без параметра є допустимим та часто використовується. Це полегшує код та робить його більш читабельним, особливо в тих випадках, коли обробник виключень використовується просто для логування помилок чи повідомлень. Отже, у відсутності **catch**-параметра в обробнику виключення немає нічого незвичайного, як у випадку, проілюстрованому у *Прикладі 1*.

Як уже згадувалося, якщо **try**-блоком виняток не згенеровано, жодна з **catch**-інструкцій не виконується, і управління програмою буде передано інструкції, наступної за **catch**-інструкцією. Змінимо цикл **for** в *Прикладі 1*.

Приклад 2



Результат:



Тепер цикл **for** не порушує кордону індексування масиву **nums.** Тому виключення не генерується, і **catch**-блок не виконується.

Якщо передбачена обробка одного виключення, а виникає інше, то воно не перехоплюється.

Приклад 3

```
Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs
ConsoleApp1                                  ConsoleApp1
```

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        int[] nums = new int[4];
        try
        {
            Console.WriteLine("Перед генеруванням виключення.");
            // Генеруємо виняток, пов'язане із потраплянням
            // індексу поза діапазоном
            for (int i = 0; i < 10; i++)
            {
                nums[i] = i;
                Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
            }
            Console.WriteLine("Цей текст не відображається.");
        }
        /* Якщо в catch-інструкції вказано тип виключення
           DivideByZeroException, то з її допомогою неможливо
           перехопити помилку порушення меж масиву. */
        catch (DivideByZeroException)
        {
            // перехоплюємо виняток.
            Console.WriteLine("Індекс поза діапазону!");
        }
        Console.WriteLine("Після catch-інструкції.");
    }
}
```

Результат:

```
C:\WINDOWS\system32\
Перед генеруванням виключення.
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3

Необработанное исключение: System.IndexOutOfRangeException: Индекс
 находился вне границ массива.
   в ConsoleApp1.Program.Main(String[] args) в D:\ОНУ им. ?.?.Мечн
икова\ConsoleApp1\Program.cs:строка 21
Для продолжения нажмите любую клавишу . . .
```

Як підтверджують результати виконання цієї програми, **catch**-інструкція, призначена для перехоплення винятку типу **DivideByZeroException**, не в змозі перехопити виняток типу **IndexOutOfRangeException**.

Одна з основних переваг обробки винятків полягає в тому, що вона дає

змогу програмі відреагувати на помилку і продовжити виконання. Розглянемо, наприклад, таку програму, яка ділить елементи одного масиву на елементи іншого. Якщо під час цього зустрічається ділення на нуль, генерується виняток типу **DivideByZeroException**. В програмі це виключення обробляється видачею повідомлення про помилку, після чого виконання програми триває. Отже, спроба поділити на нуль не викликає раптову динамічну помилку, внаслідок якої припиняється виконання програми.

Замість аварійної зупинки виняток дає змогу красиво вийти з помилкової ситуації та продовжити виконання програми.

Приклад 4

Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1 | ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        int[] numer = { 4, 8, 16, 32, 64, 128 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] +
                " равно " + numer[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {
                // Перехоплюємо виняток.
                Console.WriteLine("Ділити на нуль не можна!");
            }
        }
    }
}
```

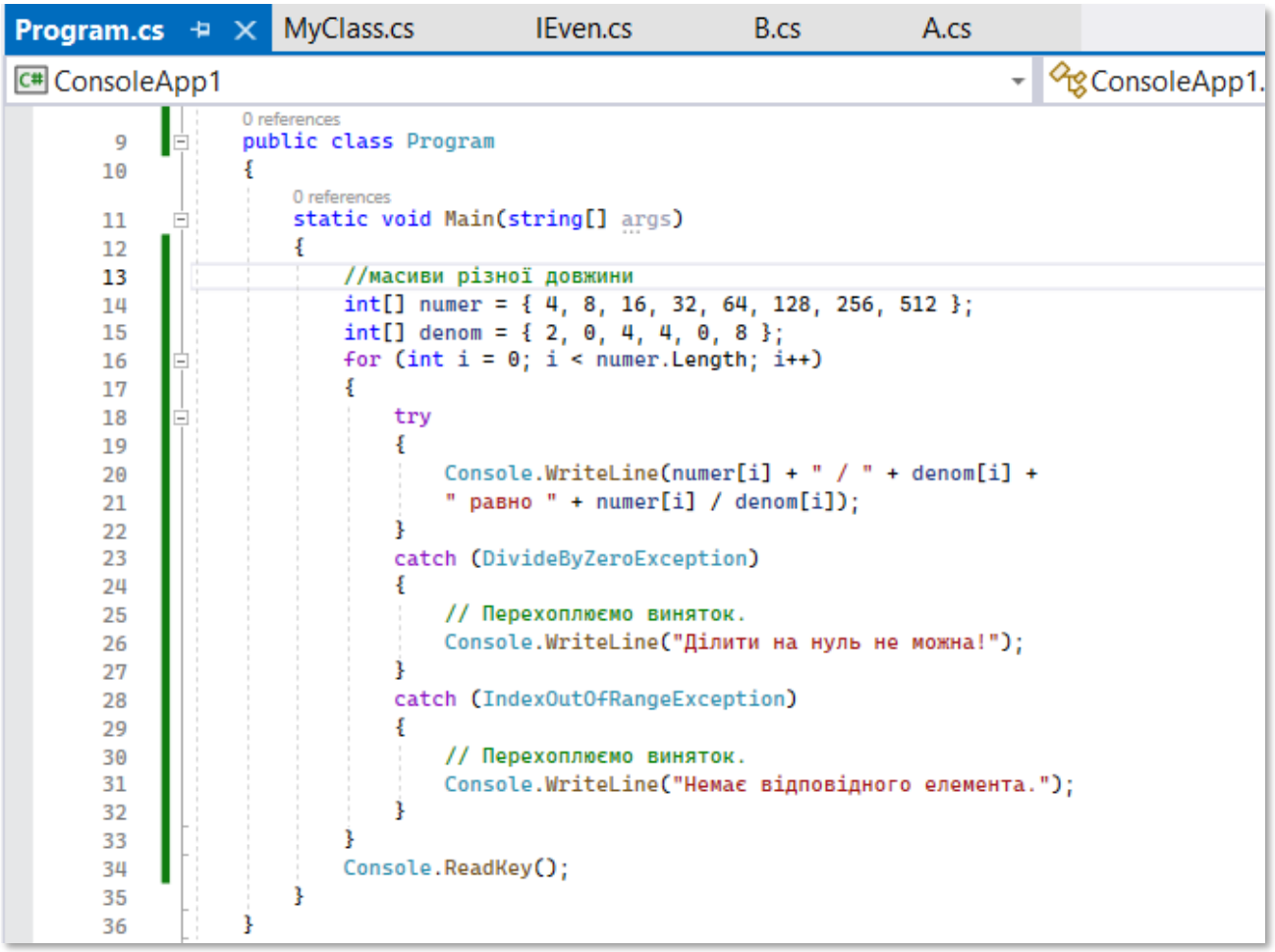
Результат:

```
D:\ОНУ им. І.І.Мечн
4 / 2 равно 2
Ділити на нуль не можна!
16 / 4 равно 4
32 / 4 равно 8
Ділити на нуль не можна!
128 / 8 равно 16
```

З **try**-блоком можна зв'язати не одне, а кілька **catch**-інструкцій. І це – досить поширена практика програмування. Однак все **catch**-інструкції повинні перехоплювати виключення різного типу. Наприклад, наступна програма

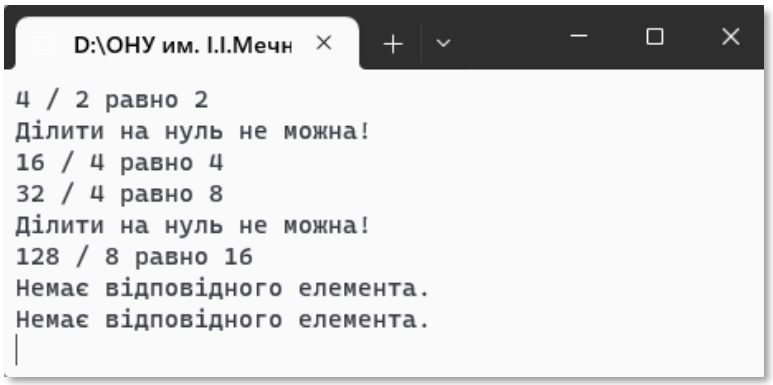
перехоплює як помилку порушення меж масиву, так і помилку ділення на нуль.

Приклад 5



```
Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs
ConsoleApp1
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        //масиви різної довжини
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] +
                " равно " + numer[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {
                // Перехоплюємо виняток.
                Console.WriteLine("Ділити на нуль не можна!");
            }
            catch (IndexOutOfRangeException)
            {
                // Перехоплюємо виняток.
                Console.WriteLine("Немає відповідного елемента.");
            }
        }
        Console.ReadKey();
    }
}
```

Результат:



```
D:\ОНУ им. І.І.Мечн
4 / 2 равно 2
Ділити на нуль не можна!
16 / 4 равно 4
32 / 4 равно 8
Ділити на нуль не можна!
128 / 8 равно 16
Немає відповідного елемента.
Немає відповідного елемента.
```

Як підтверджують результати виконання цієї програми, кожна **catch**-інструкція реагує тільки на власний тип винятку.

У загальному випадку **catch**-вирази перевіряються в тому порядку, в якому вони зустрічаються в програмі. Виконується тільки інструкція, тип виключення якої збігається зі згенерованих винятком. Всі інші **catch**-блоки ігноруються.

Іноді потрібно перехоплювати всі виключення, незалежно від їх типу. Для цього використовуйте **catch**-інструкцію без параметрів. У цьому випадку створюється обробник "глобального перехоплення", який використовується, щоб програма гарантовано опрацювала всі винятки. У наступній програмі наведено приклад використання такого обробника, який успішно перехоплює генеровані тут виняток типу **IndexOutOfRangeException** і виняток типу **DivideByZeroException**.

Приклад 6

Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1 | ConsoleApp1.

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        // масиви різної довжини
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        for (int i = 0; i < numer.Length; i++)
        {
            try
            {
                Console.WriteLine(numer[i] + " / " + denom[i] +
                    " дорівнює " + numer[i] / denom[i]);
            }
            catch
            {
                Console.WriteLine(" Відбулося певне виключення.");
            }
        }
        Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечн
4 / 2 дорівнює 2
 Відбулося певне виключення.
16 / 4 дорівнює 4
32 / 4 дорівнює 8
 Відбулося певне виключення.
128 / 8 дорівнює 16
 Відбулося певне виключення.
 Відбулося певне виключення.
```

Відносно **catch**-інструкції, призначеної для ***"глобального перехоплення"***, необхідно запам'ятати наступне: вона повинна бути останньою в послідовності **catch**-інструкцій.

Один **try**-блок можна вкласти в інший. Виняток, сгенерований у внутрішньому **try**-блоці і не перехоплений **catch**-інструкцією, яка пов'язана з цим **try**-блоком, передається в зовнішній **try**-блок. Наприклад, в наступній програмі виняток типу **IndexOutOfRangeException** перехоплюється не внутрішнім **try**-блоком, а зовнішнім.

Приклад 7

```
Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs
ConsoleApp1                                  ConsoleApp1.
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        // масиви різної довжини
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        try
        { // Зовнішній try-блок.
            for (int i = 0; i < numer.Length; i++)
            {
                try
                { // Вкладений try-блок.
                    Console.WriteLine(numer[i] + " / " +
                    denom[i] + " дорівнює " +
                    numer[i] / denom[i]);
                }
                catch (DivideByZeroException)
                {
                    // Перехоплюємо виключення.
                    Console.WriteLine("На нуль ділити не можна!");
                }
            }
        }
        catch (IndexOutOfRangeException)
        {
            // Перехоплюємо виключення.
            Console.WriteLine("Немає відповідного елемента.");
            Console.WriteLine("Невиправна помилка - програму завершено.");
        }
            Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечн
4 / 2 дорівнює 2
На нуль ділити не можна!
16 / 4 дорівнює 4
32 / 4 дорівнює 8
На нуль ділити не можна!
128 / 8 дорівнює 16
Немає відповідного елемента.
Невиправна помилка - програму завершено.
```

Виняток, який може бути оброблений внутрішнім **try**-блоком (в даному випадку це поділ на нуль), дозволяє програмі продовжувати роботу. Однак порушення меж масиву перехоплюється зовнішнім **try**-блоком і змушує програму завершитися.

У попередній програмі хочеться звернути увагу на те, що найчастіше використання вкладених **try**-блоків обумовлено бажанням обробляти різні категорії помилок різними способами. Одні типи помилок носять катастрофічний характер і не підлягають виправленню. Інші – безпечні для подальшого функціонування програми, і з ними можна справитися прямо на місці їх виникнення. Багато програмістів використовують зовнішній **try**-блок для перехоплення найсерйозніших помилок, дозволяючи внутрішнім **try**-блокам обробляти менш небезпечні. Зовнішні **try**-блоки можна також використовувати в якості механізму ***"глобального перехоплення"*** для обробки тих помилок, які перехоплюються внутрішнім блоком.

9.2 Генерування виключень вручну

У попередніх прикладах демонструвалось перехоплення винятків, згенерованих автоматично засобами **C#**. Однак можна згенерувати виняток вручну, використовуючи інструкцію **throw**. Синтаксис її виглядає так:

throw exceptOb;

Елемент *exceptOb* – це об'єкт класу винятків, похідного від класу **Exception.** Розглянемо приклад, який демонструє використання інструкції **throw** для генерування винятку типу **DivideByZeroException** вручну.

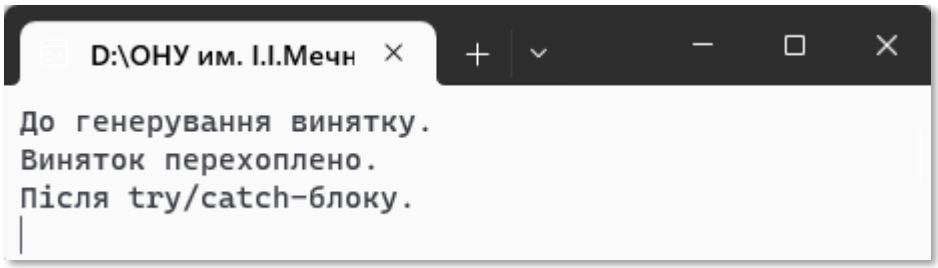
Приклад 8

Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        try
        {
            Console.WriteLine("До генерування винятку.");
            throw new DivideByZeroException();
        }
        catch (DivideByZeroException)
        {
            // Перехоплюємо виняток.
            Console.WriteLine("Виняток перехоплено.");
        }
        Console.WriteLine("Після try/catch-блоку.");
        Console.ReadKey();
    }
}
```

Результат:



У прикладі був створений об'єкт виключення типу **DivideByZeroException** за допомогою оператора new в інструкції **throw**. Інструкція **throw** генерує об'єкт, так як не можна просто згенерувати "тип винятку". В даному випадку при створенні об'єкта класу **DivideByZeroException** використовувався конструктор за замовчуванням, але для генерування виключень передбачені й інші конструктори.

Найчастіше винятки, що генеруються, є екземплярами класів винятків, створюваних в програмі. Створення власних класів винятків дозволяє обробляти помилки в коді, і ця процедура може стати частиною загальної стратегії обробки винятків програми.

Виняток, перехоплений однієї **catch**-інструкцією, можна перегенерувати, щоб забезпечити можливість його перехоплення іншою (зовнішньою) **catch**-інструкцією. Найпоширеніша причина для повторного генерування винятку – дозволити кільком обробникам отримати доступ до виключення. Наприклад, можлива така ситуація, що один обробник виключень управляє одним аспектом виключення, а другий – іншим. Щоб повторно згенерувати виключення, досить використовувати ключове слово **throw**, що не вказуючи виключення. Необхідно використовувати наступну форму інструкції **throw**:

throw ;

При повторному генеруванні винятку воно не буде повторно перехоплюватися тієї ж **catch**-інструкцією, а передається наступній **catch**-інструкції.

Пример 9

Rethrow.cs | Program.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1

```
1 reference
public class Rethrow
{
    1 reference
    public static void genException()
    {   // Тут масив numer довше масиву denom.
        int[] numer = { 4, 8, 16, 32, 64, 128, 256, 512 };
        int[] denom = { 2, 0, 4, 4, 0, 8 };
        for (int i = 0; i < numer.Length; i++)
            try
            {
                Console.WriteLine(numer[i] + " / " +
                denom[i] + " дорівнює " +
                numer[i] / denom[i]);
            }
            catch (DivideByZeroException)
            {   // перехоплюємо виняток.
                Console.WriteLine("Ділити на нуль не можна!");
            }
            catch (IndexOutOfRangeException)
            {   // перехоплюємо виняток.
                Console.WriteLine("Немає відповідного елемента.");
                throw; // Генеруємо виключення повторно.
            }

    }
}
```

Program.cs | Rethrow.cs | MyClass.cs | IEven.cs | B.cs | A.cs

ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        try
        {
            Rethrow.genException();
        }
        catch (IndexOutOfRangeException)
        {    // перехоплюємо повторно
            // сгенерированное виняток.
            Console.WriteLine("Фатальна помилка -" + " програма завершена.");
        }
        Console.ReadKey();
    }
}
```

Результат:

```
D:\ОНУ им. І.І.Мечн
4 / 2 дорівнює 2
Ділити на нуль не можна!
16 / 4 дорівнює 4
32 / 4 дорівнює 8
Ділити на нуль не можна!
128 / 8 дорівнює 16
Немає відповідного елемента.
Фатальна помилка - програма завершена.
```

У цій програмі помилки ділення на нуль обробляються локально (за місцем), тобто в самому методі **genException()**, але помилка порушення меж масиву генерується повторно. В даному випадку виключення типу **IndexOutOfRangeException** обробляється функцією **Main()**.

9.3 Використання блоку finally

Іноді виникає потреба визначити програмний блок, який повинен виконуватися по виходу з **try/catch**-блоку. Наприклад, виключення може викликати помилку, яка завершує поточний метод і, отже, є причиною передчасного повернення. Однак такий метод може залишити відкритим файл або з'єднання з мережею, які необхідно закрити. Подібні обставини – звичайне явище в програмуванні, і **C#** надає зручний шлях виходу з них: блок **finally**.

Загальна форма запису послідовності **try/catch**-блоків, що містить блок **finally**, виглядає наступним чином.

```
try {
// Блок коду, призначений для обробки помилок.
}
catch (ExcepTypel exOb) {
// Оброблювач для виключення типу ExcepType1
}
catch (ExcepType2 exOi) {
// Оброблювач для виключення типу ExcepType2
finally {
// Код завершення обробки винятків
}
```

Блок **finally** буде виконаний після виходу з **try/catch**-блоку, незалежно від умов його виконання. Іншими словами, при нормальному завершенні **try**-блоку або в умовах виникнення винятку вміст **finally**-блоку буде безумовно відпрацьовано. Блок **finally** виконається і в тому випадку, якщо будь-який код всередині **try**-блоку або будь-яка з його **catch**-інструкцій визначені всередині методу.

Блок **finally** часто використовується для звільнення ресурсів, таких як закриття файлів або роз'єднання з'єднань з базою даних, оскільки він викликається навіть після виникнення винятку.

Приклад 10

UseFinally.cs | Program.cs | Rethrow.cs | MyClass.cs | IEven.cs

ConsoleApp1

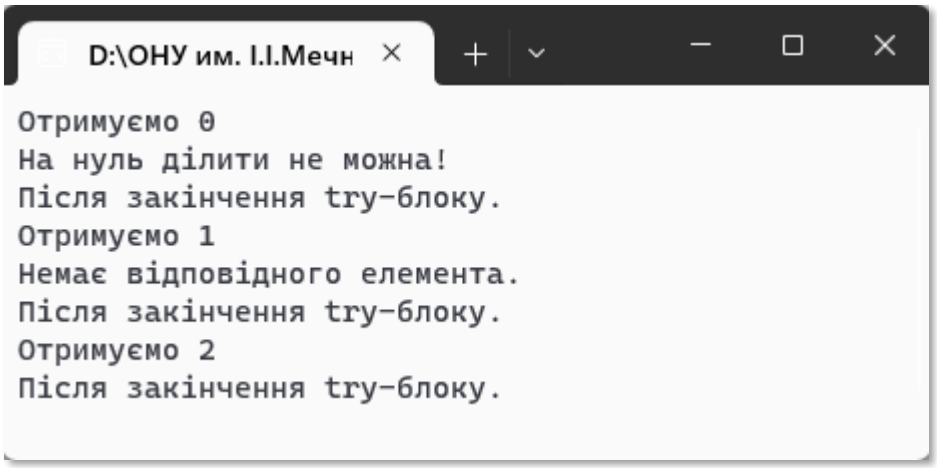
```
0 references
public class UseFinally
{
    0 references
    public static void genException(int what)
    {
        int t;
        int[] nums = new int[2];
        Console.WriteLine("Отримуємо " + what);
        try
        {
            switch (what)
            {
                case 0:
                    t = 10 / what; // Генеруємо помилку ділення на нуль,
                    break;
                case 1:
                    nums[4] = 4; // Генеруємо помилку індексування масиву.
                    break;
                case 2:
                    return;
                    // Повернення з try-блоку.
            }
        }
        catch (DivideByZeroException)
        {
            // Перехоплюємо виняток.
            Console.WriteLine("На нуль ділити не можна!");
            return; // Возврат из catch-блока.
        }
        catch (IndexOutOfRangeException)
        {
            // Перехоплюємо виняток.
            Console.WriteLine("Немає відповідного елемента.");
        }
        finally
        {
            Console.WriteLine("Після закінчення try-блоку.");
        }
    }
}
```

Program.cs | UseFinally.cs

ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        for (int i = 0; i < 3; i++)
            UseFinally.genException(i);
        Console.WriteLine();

        Console.ReadKey();
    }
}
```

Результат:



Як підтверджують результати виконання цієї програми, незалежно від результату завершення **try**-блоку, блок **finally** виконується обов'язково.

Ми розглядали приклади, в яких перехоплювали винятки, але нічого не робили з самим об'єктом виключення. Як розповідалося на попередній лекції, **catch**-інструкція дає змогу задати тип виключення та параметр. Параметр призначений для прийняття об'єкта виключення. Оскільки всі класи винятків є похідними від класу **Exception**, всі винятки підтримують члени, визначені в цьому класі. Ми розглянемо найбільш корисні члени та конструктори класу **Exception**.

Основні властивості класу **Exception**: **Message**, **StackTrace** і **TargetSite**. Усі вони призначені тільки для читання. Властивість **Message** містить рядок, який описує причину помилки, а властивість **StackTrace** – рядок зі стеком викликів, що призвели до виникнення виключень. Властивість **TargetSite** приймає об'єкт, який задає метод, який згенерував виняток.

У класі **Exception** також визначено низку методів. Найчастіше використовується метод **ToString()**, який повертає рядок з описом винятку. Метод **ToString()** автоматично викликається, якщо деяке виключення відображається, наприклад, за допомогою методу **WriteLine().** Коли використовується **Console.WriteLine()** або інші методи для виведення виключення на консоль, вони автоматично викликають метод **ToString()** для об'єкта винятку. Таким чином, можна отримати зручний рядок, який містить текстовий опис винятку та його деталі.

Використання властивостей і методів класу **Exception** демонструється в такій програмі.

Приклад 11

ExcTest.cs | Program.cs | UseFinally.cs

ConsoleApp1 | ConsoleApp1.

```
1 reference
public class ExcTest
{
    1 reference
    public static void genException()
    {
        int[] nums = new int[4];
        Console.WriteLine("Перед генеруванням винятку.");
        // Генеруємо виняток, пов'язаний із потраплянням
        // індексу поза діапазоном,
        for (int i = 0; i < 10; i++)
        {
            nums[i] = i;
            Console.WriteLine("nums[{0}]: {1}", i, nums[i]);
        }
        Console.WriteLine("Цей текст не відображається.");
    }
}
```

Program.cs | ExcTest.cs | UseFinally.cs

ConsoleApp1 | ConsoleApp1

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        try
        { ExcTest.genException(); }
        catch (IndexOutOfRangeException exc)
        {
            // Перехоплюємо виняток.
            Console.WriteLine("Стандартне повідомлення таке: ");
            Console.WriteLine(exc); // Виклик методу ToString().
            Console.WriteLine("Властивість StackTrace: " + exc.StackTrace);
            Console.WriteLine("Властивість Message: " + exc.Message);
            Console.WriteLine("Властивість TargetSite: " + exc.TargetSite);
        }
        Console.WriteLine("Після catch-інструкції.");
    }
}
```

Результат:

C:\WINDOWS\system32\

```
nums[0]: 0
nums[1]: 1
nums[2]: 2
nums[3]: 3
Стандартне повідомлення таке:
System.IndexOutOfRangeException: Индекс находился вне границ массива.
   в ConsoleApp1.ExcTest.genException() в D:\ОНУ им. ?.?.Мечникова\ConsoleApp1\ExcTest.cs:строка 19
   в ConsoleApp1.Program.Main(String[] args) в D:\ОНУ им. ?.?.Мечникова\ConsoleApp1\Program.cs:строка 14
Властивість StackTrace:    в ConsoleApp1.ExcTest.genException() в D:\ОНУ им. ?.?.Мечникова\ConsoleApp1\ExcTest.cs:
строка 19
   в ConsoleApp1.Program.Main(String[] args) в D:\ОНУ им. ?.?.Мечникова\ConsoleApp1\Program.cs:строка 14
Властивість Message: Индекс находился вне границ массива.
Властивість TargetSite: Void genException()
Після catch-?нструкції.
```

У класі **Exception** визначено чотири конструктори. Найчастіше використовуються такі: **Exception()** і **Exception(string str).**

Перший – це конструктор за замовчуванням. Другий приймає значення властивості **Message**, пов'язане з виключенням. Під час створення власних класів винятків необхідно реалізувати обидва ці конструктори.

У просторі імен **System** визначено кілька стандартних вбудованих винятків.

Усі вони виведені з класу **SystemException**, оскільки генеруються системою динамічного керування (**Common Language Runtime**) при виникненні динамічних помилок. Деякі із найуживаніших стандартних винятків, визначених у **C#**, наведено в таблиці:

виняток	***значення***
ArrayTypeMismatchException	Тип зберігаємого значення несумісний з типом масиву
DivideByZeroException	Спроба ділення на нуль
IndexOutOfRangeException	Індекс масиву опинився поза діапазоном
InvalidCastException	Невірно виконано динамічне приведення типів
OutOfMemoryException	Звернення до оператора **new** виявилося невдалим через недостатній обсяг вільної пам'яті
OverflowException	Має місце арифметичне переповнення
NullReferenceException	Була зроблена спроба використовувати нульове посилання, тобто посилання, яке не вказує ні на який об'єкт
StackOverflowException	Переповнення стека

Більшість винятків, перелічених у таблиці, не потребує додаткових роз'яснень, за винятком класу **NullReferenceException**. Це виняток генерується при спробі використати нульове посилання, наприклад, при спробі викликати метод, передавши йому замість посилання на об'єкт нульове посилання. ***Нульове посилання*** не вказує ні на який об'єкт. Один зі способів створити нульове посилання – явно присвоїти посилальній змінній **null**-значення, використовуючи ключове слово **null.** Також важливо бути обережним при отриманні результатів виклику методів чи властивостей, які можуть повертати **null**. Нульові посилання можна отримати й іншими шляхами. Розглянемо програму, в якій демонструється виникнення винятку типу **NullReferenceException**.

Приклад 12

X.cs | Program.cs

ConsoleApp1 | ConsoleApp1.

```
2 references
public class X
{
    int x;
    0 references
    public X(int a)
    { x = a; }
    0 references
    public int add(X o)
    { return x + o.x; }
}
```

Program.cs | X.cs

ConsoleApp1 | ConsoleApp1.

```
0 references
public class Program
{
    0 references
    static void Main(string[] args)
    {
        X p = new X(10);
        X q = null; // Змінній q явно присвоюється значення null
        int val;
        try
        { val = p.add(q); } //Такий виклик методу призведе до виключення.
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException!");
            Console.WriteLine("Виправляємо помилку...\n");
            // Виправляємо помилку
            q = new X(9);
            val = p.add(q);
        }
        Console.WriteLine("Значення val дорівнює {0}", val);
    }
}
```

Результат:

```
C:\WINDOWS\syste
NullReferenceException!
Виправляємо помилку...

Значення val дорівнює 19
```

Ця програма створює клас **X**, у якому визначається член x і метод **add(),** призначений для додавання значення x, що належить об'єкту, який викликає, з членом **x**, який визначено в об'єкті, переданому як параметр. У методі **Main()** створюються два об'єкти класу **X**. Перший **p** ініціалізується, а другий **q** – ні

(йому явним чином присвоюється значення **null**). Потім викликається метод **p.add()**, якому значення **q** передається як аргумент. Оскільки змінна **q** не посилається на жоден об'єкт, при спробі отримати значення члена **q. x** генерується виняток типу **NullReferenceException**.

Заслуговує на увагу виняток типу **StackOverflowException**, який генерується при переповненні системного стека. Воно може виникнути при некоректному визначенні рекурсивного методу. Програмісту, який захоплюється рекурсією, можливо, варто уважно відстежити появу винятку цього типу, вживши відповідних заходів у разі його виявлення. Однак тут слід проявити обережність. Якщо вже цей виняток згенеровано, значить, системний стек вичерпав свої можливості, тому найкраще просто почати аналіз із рекурсивного виклику.

10.ЛАБОРАТОРНІ РОБОТИ

10.1 Лабораторна робота 1: Масиви масивів

Варіант № 1. Дано двовимірний масив **A** цілих чисел розміру **p*n**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться додатні числа з рядків масиву **A**. Визначити скільки елементів у кожному рядку масиву B більше числа **N**, заданого користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 2. Дано двовимірний масив **A** цілих чисел розміру **p*n**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться від'ємні числа з рядків масиву **A**. Визначити, скільки елементів у кожному рядку масиву **B** менше числа **N**, заданого користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 3. Дано двовимірний масив **A** цілих чисел розміру **m*n**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться парні числа з рядків масиву **A**. Визначити скільки елементів у масиві **B** більше числа **N**, заданого користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію

розв'язання поставленої задачі.

Варіант № 4. Дано двовимірний масив **A** цілих чисел розміру **n*m**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться непарні числа з рядків масиву **A**. Визначити, скільки елементів у масиві **B** менше числа **N**, заданого користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 5. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться числа з рядків масиву **A**, кратні **5**. Визначити скільки елементів у кожному рядку масиву B менше числа **N**, заданого користувачем. У додатку використовувати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 6. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться числа зі стовпців масиву **A**, кратні **3**. Визначити скільки елементів у кожному рядку масиву B більше числа **N**, заданого користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 7. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться дільники числа **D** (задає користувач) з рядків масиву **A**. Визначити скільки елементів у кожному рядку масиву **B** кратні числу **N**, заданому користувачем. У додатку використовувати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 8. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться дільники числа **D** (задає користувач) зі стовпчиків масиву **A.** Визначити, скільки елементів у кожному рядку масиву **B** кратні числу **N**, заданому користувачем. У додатку використовувати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої

задачі.

Варіант № 9. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться прості числа з рядків масиву **A**. Визначити середнє арифметичне в кожному рядку масиву **B**. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 10. Дано двовимірний масив **A** цілих чисел розміру **p*k**, який заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться непрості числа з рядків масиву **A**. Визначити середнє арифметичне в кожному рядку масиву **B**. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язування поставленої задачі.

Варіант № 11. Дано двовимірний масив **A** розміру **p*k**, елементи якого є векторами у двовимірному просторі. Масив заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться вектори, модуль яких дорівнює числу **M**, заданому користувачеві з рядків масиву **A**. Визначити скільки векторів у кожному рядку масиву **B** перпендикулярні вектору **N(a1,a2)**, заданому користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

Варіант № 12. Дано двовимірний масив **A** розміру **p*k**, елементи якого є векторами в тривимірному просторі. Масив заповнено випадковим чином. Побудувати масив масивів **B**, у якому в рядку містяться вектори, модуль яких дорівнює числу **M**, заданому користувачеві, зі стовпчиків масиву **A**. Визначити, скільки векторів у кожному рядку масиву **B** є колінеарними вектору **N(a1,a2,a3)**, заданому користувачем. У додатку використати функцію побудови масиву масивів, функцію виведення масиву масивів, функцію розв'язання поставленої задачі.

10.2 Лабораторна робота 2: Списки

Варіант № 1

- Сформувати список із дійсних чисел. Знайти найбільший елемент списку.
- Побудувати копію даного лінійного списку (зі збереженням порядку проходження елементів).
- Ввести символи, формуючи з них стек. Поміняти місцями перший і останній елементи стека.

Варіант № 2

- Сформувати список із дійсних чисел. Знайти індекс першого входження від'ємного значення у списку чисел.
- Створіть додаток, який розділить вихідний список із цілих чисел на два списки: список додатних чисел і список від'ємних чисел.
- Ввести символи, формуючи з них стек. Видалити елемент, який знаходиться в середині стека, якщо непарне число елементів, а якщо парне, то два середніх.

Варіант № 3

- Сформувати список із дійсних чисел. Обчислити середнє арифметичне елементів списку.
- Створіть додаток, який розділить вихідний список із цілих чисел на два списки: список чисел кратних A і список чисел некратних A.
- Ввести числа, формуючи з них стек. Знайти мінімальний елемент і вставити після нього 0.

Варіант № 4

- Сформувати список із дійсних чисел. Замінити всі входження додатних значень на значення першого елемента у списку чисел.
- Створіть додаток, який розділить початковий список з дійсних чисел на два списки: список чисел з діапазону [a,b] і список чисел не із заданого діапазону.
- Ввести числа, формуючи з них стек. Знайти максимальний елемент і вставити після нього 0.

Варіант № 5

- Сформувати список із дійсних чисел. Обчислити суму елементів, що знаходяться на парних позиціях.

- Створіть додаток, який розділить вихідний список з дійсних чисел на два списки: список чисел, більших за середнє арифметичне заданого списку, і список чисел менше середнього арифметичного заданого списку.
- Ввести символи, формуючи з них стек. Вставити символ '*' у середину стека, якщо парне число елементів, а якщо непарне, то після середнього елемента.

Варіант № 6

- Сформувати список із цілих чисел. Збільшити елементи вихідного списку на одиницю.
- Створіть додаток, який розділить вихідний список із цілих чисел на два списки: список чисел більших за A і список чисел менших за A.
- Ввести символи, формуючи з них стек. Видалити кожен другий елемент стека.

Варіант № 7

- Сформувати список із дійсних чисел. Вставити новий елемент перед n-им елементом у списку.
- Створіть додаток, який розділить вихідний список рядків на два списки: список рядків, що містять тільки великі літери, і список рядків, що містять не тільки великі літери.
- Ввести символи, формуючи з них стек. Розгорнути стек, тобто "дно" стека зробити вершиною, а вершину – "дном".

Варіант № 8

- Сформувати список із дійсних чисел. Написати функцію, яка залишає у списку тільки перші входження елементів, що збігаються.
- Створіть додаток, який розділить вихідний список рядків на два списки: список рядків, що закінчуються голосною літерою, і список рядків, що закінчуються приголосною буквою.
- Написати додаток, який за двома стеками L1, L2 формує новий стек, включаючи в нього по одному разу елементи, які входять хоча б в один зі стеків L1, L2.

Варіант № 9

- Сформувати список із дійсних чисел. Видалити N елементів зі списку, починаючи з K-го.
- Створіть додаток, який розділить вихідний список рядків на два списки:

список рядків, що починаються з голосної літери, і список рядків, що починаються з приголосної літери.

- Написати додаток, який за двома стеками L1, L2 формує новий стек, що складається з елементів, які входять до L1, але не входять до L2.

Варіант № 10

- Сформувати список із дійсних чисел. Написати функцію, яка видаляє зі списку елементи, що входять до нього тільки по одному разу.
- Створіть додаток, який розділить вихідний список рядків на два списки: список рядків з кількістю голосних літер, більшою за А, і список рядків з кількістю голосних букв, меншою за А.
- Написати додаток, який формує стек L, включаючи в нього по одному разу елементи, які входять в один зі стеків L1, L2, але водночас не входять в обидва.

Варіант № 11

- Сформувати список із рядків. Написати функцію, яка перевіряє, чи впорядковані елементи списку за довжиною рядків.
- Створіть додаток, який розділить вихідний список одновимірних масивів на два списки: список одновимірних масивів із сумою, більшою за заданий А, і список одновимірних масивів із сумою, меншою за заданий А.
- Дано стек дітей. Побудувати гру "Лічилка" із заданим числом n. Вибуває дитина з номером n. Дітей рахувати по колу. Знайти переможця.

Варіант № 12

- Сформувати список із рядків. Написати функцію, яка перевіряє, чи впорядковані елементи списку за алфавітом.
- Створіть додаток, який розділить вихідний список двовимірних масивів на два списки: список одновимірних масивів з рядків двовимірних масивів, у яких сума елементів більша за заданий А і список одновимірних масивів з рядків двовимірних масивів, у яких сума елементів менша за заданий А.
- Дано 2 стеки, що містять прізвища спортсменів 2-х фехтувальних команд. Провести жеребкування. У першій команді вибирається кожен n-й гравець (по колу), а у другій – кожен m-й (по колу). Сформувати пари супротивників і вивести на екран

10.3 Лабороторна робота 3: Створення власного класу

Додаток містить меню з пунктами вибору для реалізації різних функціональних можливостей додатка. Після виконання функції обраного пункту на екрані щоразу з'являється ***Меню****. Меню містить пункт* ***Виходу****, вибір якого завершує роботу додатка.*

Варіант № 1.Створити клас прямокутників, який містить закриті дані-члени: довжина та ширина має перевантажені конструктори, включаючи конструктор за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку периметра, площі, довжин діагоналей. У головній функції програми створити кілька прямокутників за допомогою різних конструкторів і вивести на екран усі характеристики кожного об'єкта.

Варіант № 2. Створити клас паралелограмів, який містить закриті дані-члени довжин сторін і гострого кута, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку периметра, площі, довжин діагоналей, висот. У головній функції додатка створити кілька об'єктів за допомогою різних конструкторів і вивести на екран усі характеристики кожного об'єкта.

Варіант № 3. Створити клас ромбів, який містить закриті дані-члени довжини сторони та гострого кута, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку периметра, площі, довжин діагоналей, висот. У головній функції додатка створити кілька об'єктів за допомогою різних конструкторів і вивести на екран усі характеристики кожного об'єкта.

Варіант № 4. Створити клас трикутних правильних пірамід, що містить закриті дані-члени довжини сторони основи та висоти, має перевантажені конструктори, включно з конструктор за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку площі бічної поверхні, площі основи, площі повної поверхні, об'єму. У головній функції програми створити кілька об'єктів за допомогою різних конструкторів і вивести на екран усі характеристики кожного об'єкта.

Варіант № 5. Створити клас правильних чотирикутних пірамід, який містить закриті дані-члени довжини сторони основи і висоти, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартним). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку площі бічної поверхні, площі основи, площі повної поверхні, об'єму. У головній функції додатка створити кілька об'єктів за допомогою різних конструкторів і вивести на екран усі характеристики кожного об'єкта.

Варіант № 6. Створити клас арифметичної прогресії, який містить закриті дані-члени: перший член, різницю, номер поточного члена; має перевантажені конструктори, включаючи конструктор за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку члена прогресії, суми заданої кількості членів за допомогою двох формул. У головній функції програми створювати об'єкти у кількості, необхідної користувачеві, за допомогою різних конструкторів, і вивести на екран інформацію, необхідну користувачеві, обчислити скільки позитивних членів прогресії. Створені об'єкти зберегти у файлі після завершення роботи додатка.

Варіант № 7. Створити клас геометричної прогресії, який містить закриті дані-члени: перший член, знаменник, номер поточного члена, має перевантажені конструктори, включаючи конструктор за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи розрахунку члена прогресії, суми заданої кількості членів за допомогою двох формул. У головній функції програми створювати об'єкти, у кількості, необхідної користувачеві, за допомогою різних конструкторів, і вивести на екран інформацію, необхідну користувачеві. Створені об'єкти зберегти у файлі після завершення роботи програми.

Варіант № 8. Створити клас кутів на площині, який містить закриті дані-члени в градусах і хвилинах, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи переведення в радіани, приведення до діапазону 0-360, збільшення і зменшення кута на задану величину, отримання тригонометричних функції кута, порівняння кутів. У головній функції додатка створювати об'єкти у кількості, необхідній користувачеві, за

допомогою різних конструкторів, і вивести на екран інформацію, необхідну користувачеві. Створені об'єкти зберегти у файлі по завершенню роботи додатка.

Варіант № 9. Створити клас точок на площині, який містить закриті дані-члени координати x і y, має перевантажені конструктори, включаючи конструктор за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи переміщення точки по осі X, переміщення по осі Y, визначення відстані до початку координат, відстані між двома точками, перетворення в полярні координати, промальовування точки на площині консолі у вигляді символу. У головній функції додатка створювати об'єкти, у кількості, необхідної користувачеві, за допомогою різних конструкторів, і вивести на екран інформацію, необхідну користувачеві. Створені об'єкти зберегти у файлі по завершенню роботи програми.

Варіант № 10. Створити клас точок у просторі, який містить закриті дані-члени координати x, y і z, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартний). Містить відкриті властивості для доступу до закритих полів. Має методи переміщення точки по осях X, Y і Z, визначення відстані до початку координат, відстані між двома точками, перетворення у сферичні координати, знаходження координат точок, симетричних до заданої відносно площин XY, YZ і XZ, осей X, Y і Z. У головній функції додатка створювати об'єкти у кількості, необхідної користувачеві, за допомогою різних конструкторів, і вивести на екран інформацію, необхідну користувачеві. Створені об'єкти зберегти у файлі після завершення роботи додатка.

Варіант № 11. Створити клас Банкомат, який містить закриті дані-члени ідентифікаційного номера банкомату, інформацію про поточну суму грошей, що залишилася у банкоматі, мінімальну та максимальну суми, які дозволяється клієнту зняти, має перевантажені конструктори, включно зі стандартним. Містить відкриті властивості для доступу до закритих полів. Має методи завантаження купюр у банкомат, зняття певної суми грошей, виведення інформації про суму грошей, що залишилася, на екран. Реалізувати роботу з цим класом. Кількість банків, з яким працює клієнт, не обмежена. Створені об'єкти зберегти у файлі після завершення роботи програми.

Варіант № 12. Створити клас Зарплата, який містить закриті дані-члени прізвище, ім'я-по-батькові, оклад, рік надходження на роботу, відсоток надбавки, прибутковий податок, кількість відпрацьованих днів у місяці, нараховану суму та утриману суму, має перевантажені конструктори, включно з конструктором за замовчуванням (стандартним). Містить відкриті властивості для доступу до закритих полів. Має методи обчислення нарахованої суми, обчислення утриманої суми, обчислення суми, що видається на руки, обчислення стажу. Стаж обчислюється як повна кількість років, що минули від року надходження на роботу до поточного року. Нарахування являють собою суму, нараховану на відпрацьовані дні, і надбавки, тобто частки від першої суми. Утримання являють собою відрахування до пенсійного фонду (1 % від нарахованої суми) і прибутковий податок (13 % від нарахованої суми без відрахувань до пенсійного фонду). Реалізувати роботу з цим класом. Кількість зарплат, з якою працює бухгалтер не обмежена. Створені об'єкти зберегти у файлі після завершення роботи додатка.

10.4 Лабораторна робота 4: Перевантаження операторів

Варіант № 1. Визначити клас **Pair** із двома полями, **first** і **second**, типу **float**. Реалізувати в класі конструктори ініціалізації з присвоєнням параметрам значень за замовчуванням. Перевантажити операцію **operator+** як метод.

Варіант № 2. Створити клас **Time** для роботи з часом у форматі "**година:хвилина:секунда**". Метод обчислення різниці двох моментів часу реалізувати як перевантажену операцію **operator-**, а метод, що обчислює час через задану кількість хвилин, – як перевантажену операцію додавання **operator+**.

Варіант № 3. Розробити клас **Date** для роботи з датами. У класі має бути три поля: **рік**, **місяць** і **день**. Для представлення місяців визначити в класі перечислюваний тип **month**. Перевантажити операцію **operator+** для обчислення дати через задану кількість днів. Перевантажити операції інкременту **operator++** і декременту **operator--** для обчислення наступної та попередньої дат.

Варіант № 4. Створити клас **Fraction** для роботи з дробовими числами. Число має бути представлено двома полями: **ціла** частина – ціле зі знаком, **дробова**

частина – беззнакове ціле. Реалізувати арифметичні операції додавання, віднімання, множення та порівняння на рівність.

Варіант № 5. Створити клас **Goods** (товар). У класі мають бути представлені поля **найменування** товару, **ціни** товару, **кількості** одиниць товару і номера накладної, за якою товар надійшов на склад. Реалізувати операції зміни ціни товару, зміни кількості товару (збільшення і зменшення), обчислення вартості товару.

Варіант № 6. Скласти опис класу для представлення комплексних чисел. Забезпечити виконання операцій додавання, віднімання та множення комплексних чисел. Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант № 7. Створіть клас **Polar**, який призначений для зберігання полярних координат (**радіуса** та **кута**). Перевантажте операцію **+** для виконання додавання для об'єктів **Polar**. Додавання двох об'єктів виконується шляхом додавання координат x і y об'єктів. Результат буде координатами нової точки. Таким чином, вам потрібно буде перетворити полярні координати до прямокутних, скласти їх, а потім назад перетворити прямокутні координати результату до полярних.

Варіант № 8. Створити клас **Polinom** для роботи з многочленами до 100-го ступеня. Коефіцієнти мають бути представлені масивом зі 100 елементів-коефіцієнтів. Молодший ступінь має менший індекс (нульовий ступінь – нульовий індекс). Розмір масиву задається як аргумент конструктора ініціалізації. Реалізувати арифметичні операції та операції порівняння, обчислення значення полінома для заданого значення, диференціювання, інтегрування.

Варіант № 9. Описати клас, що реалізує тип даних "**дійсна матриця**" та роботу з ними. Клас має реалізовувати такі операції над матрицями:
- додавання, віднімання (як на іншу матрицю, так і на число);
- операції порівняння на рівність і нерівність;
- операції обчислення оберненої та транспонованої матриці;

- доступ до елемента за індексами.

Написати програму, що демонструє роботу з цим класом. Програма повинна містити меню, що дозволяє здійснити перевірку всіх методів класу.

Варіант № 10. Створити клас **Int**. Перевантажте п'ять цілочисельних арифметичних операцій **(+, -, *, /, %)** так, щоб можна було використовувати для операцій з об'єктами класу **Int**. Якщо результат якоїсь операції виходить за межі int, то операція повинна надіслати повідомлення про помилку і завершити програму. Такі типи корисні там, де помилки можуть бути спричинені арифметичним переповненням, яке є неприпустимим.

Підказка: для полегшення перевірки переповнення виконуйте обчислення з використанням типу **long double**.

Варіант № 11. Створити клас "**набір доміно**", що включає закритий масив елементів класу "**доміно**". У доміно зберігаються **два числа**. Забезпечити можливість виведення доміно за номером, виведення всього набору доміно, перемішування доміно і видача доміно по 7 штук у випадковому порядку. Набори доміно можна складати і порівнювати.

Варіант № 12. Створити клас "**колода карт**", що включає закритий масив елементів класу "**карта**". У карті зберігаються **масть** і **номер**. Забезпечити можливість виведення карти за номером, виведення всіх карт, перемішування колоди і видача всіх карт із колоди поодинці та по 6 штук у випадковому порядку. Колоди карт можна складати і порівнювати.

10.5 Лабораторна робота 5: Масиви і класи

Варіант № 1. Створити клас, що містить як дані члени: одновимірний масив цілих чисел і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній функції додатка проілюструвати роботу класу.

Варіант № 2. Створити клас, що містить як дані члени: одновимірний масив дійсних чисел і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній

функції додатка проілюструвати роботу класу.

Варіант № 3. Створити клас, що містить як дані члени: одновимірний масив символів і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром і параметром символом. У головній функції додатка проілюструвати роботу класу.

Варіант № 4. Створити клас, що містить як дані члени: двовимірний масив цілих чисел і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельними параметрами. У головній функції додатка проілюструвати роботу класу.

Варіант № 5. Створити клас, що містить як дані члени: двовимірний масив дійсних чисел і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельними параметрами. У головній функції додатка проілюструвати роботу класу.

Варіант № 6. Створити клас, що містить як дані члени: двовимірний масив символів і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельними параметрами (номер рядка і стовпця), параметром символом (номер рядка) і цілим числом (номер стовпця). У головній функції додатка проілюструвати роботу класу.

Варіант № 7. Створити клас, що містить як дані члени: одновимірний масив структур **Машина** і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній функції додатка проілюструвати роботу класу.

Варіант № 8. Створити клас, що містить у якості даних членів: двовимірний масив структур **Дисципліна** та його розміри (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельними параметрами. У

головній функції додатка проілюструвати роботу класу.

Варіант № 9. Створити клас, що містить як дані члени: одновимірний масив об'єктів класу **Точка** і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній функції додатка проілюструвати роботу класу.

Варіант № 10. Створити клас, що містить як дані члени: двовимірний масив об'єктів класу **Багатокутник** і його розміри (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельними параметрами. У головній функції додатка проілюструвати роботу класу.

Варіант № 11. Створити клас, що містить у якості даних членів: одновимірний масив списків об'єктів класу **Студент** і його розмір (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній функції додатка проілюструвати роботу класу.

Варіант № 12. Створити клас, що містить як дані члени: двовимірний масив списків об'єктів класу **Книга** і його розміри (дотримуватися принципу інкапсуляції). Клас містить відкриті властивості для доступу до закритих полів. Забезпечити індексований доступ до об'єкта з цілочисельним параметром. У головній функції додатка проілюструвати роботу класу.

10.6 Лабораторна робота 6: Спадкування та поліморфізм

Побудувати ієрархію класів для побудови зображення, заданого умовою варіанта. При побудові власної ієрархії дотримуватися принципів інкапсуляції, успадкування та поліморфізму. Для закритих даних-членів використовувати відкриті властивості. У користувацькому інтерфейсі головної функції додатка передбачити таку функціональну можливість: зображення з'являється на консолі в місці, координати якого задає користувач.

Варіант № 1. Будиночок

Варіант № 2. Машина

Варіант № 3. Багатоповерхівка

Варіант № 4. Літак

Варіант № 5. Паровоз

Варіант № 6. Собака

Варіант № 7. Робот

Варіант № 8. Мобільний телефон

Варіант № 9. Поїзд

Варіант № 10. Кіт

Варіант № 11. Комп'ютер

Варіант № 12. Палац

ЗМІСТОВИЙ МОДУЛЬ 2

1. ТЕХНОЛОГІЯ WINDOWS FORMS APPLICATION

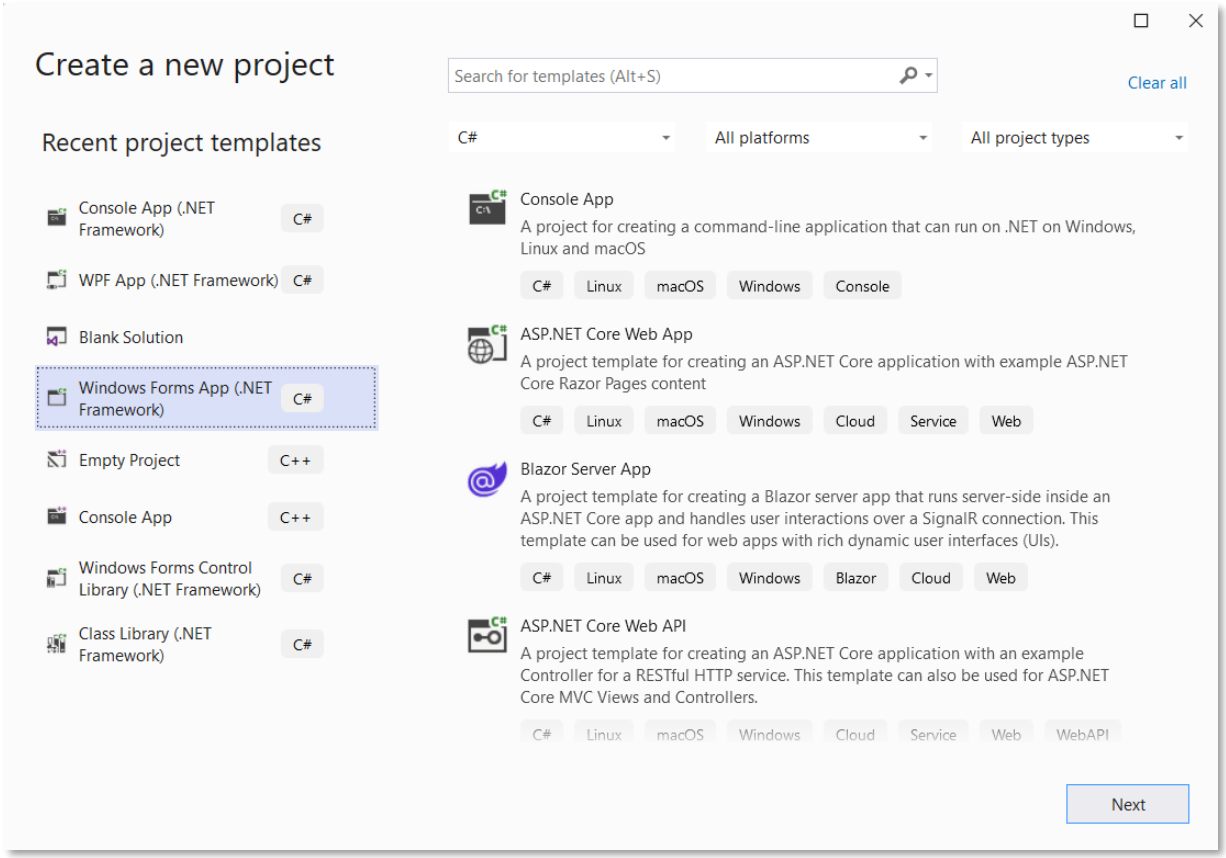
Об'єктно-орієнтований підхід вже давно став природним для будь-яких прикладних програм **Windows**. Коли ви починаєте виконувати будь-яку програму **Windows**, ви бачите зазвичай діалогове вікно з меню, кнопками, індикаторами, що випадають списками і т.п. Все це **об'єкти**. Самі по собі вони не проводять ніяких дій, поки не отримають якесь повідомлення від користувача. Після отримання повідомлення відбуваються якісь дії. Після цього будь-яка активність знову згасає, поки той чи інший об'єкт не отримає нового повідомлення від користувача або іншого об'єкта. Таким чином, об'єктно-орієнтована програма не має жорсткого алгоритму роботи. Вона являє собою систему об'єктів, кожен з яких може виконувати якісь функції у відповідь на отримане повідомлення, зокрема, може сам генерувати повідомлення, на які будуть реагувати інші об'єкти.

1.1 Основи візуального програмування інтерфейсу

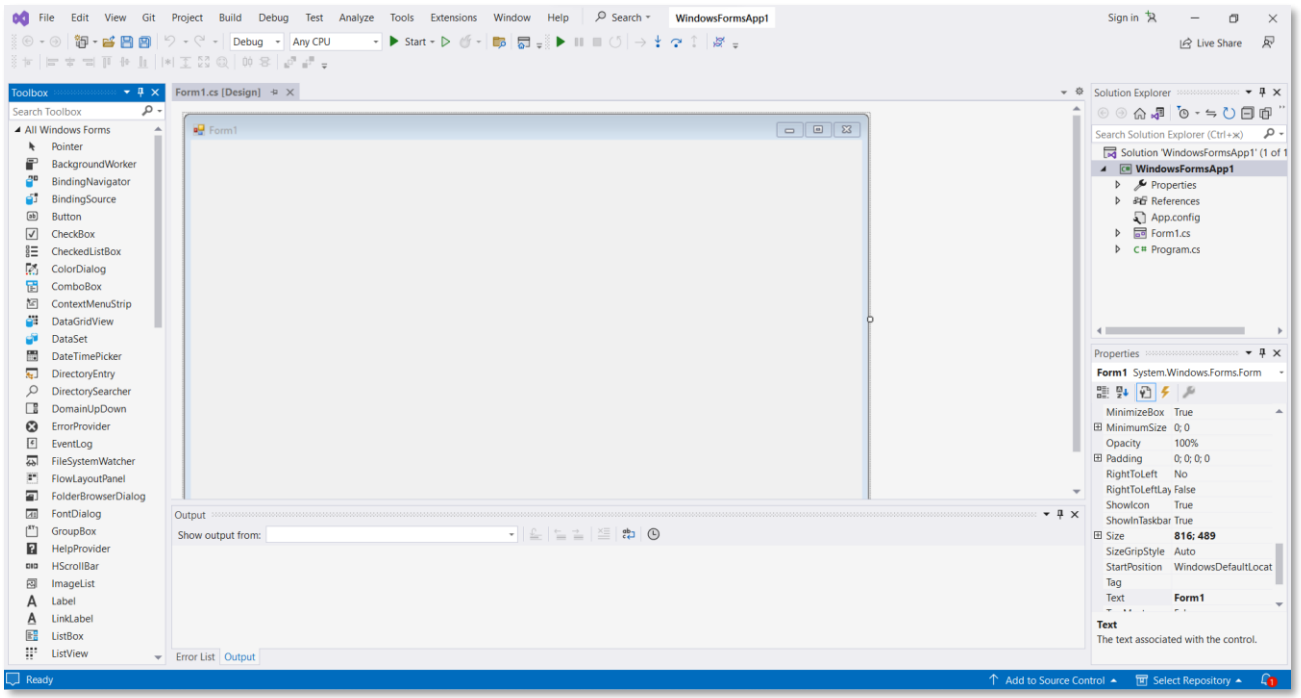
Для створення програми **Windows Forms Application** необхідно:

1. Запустити **Visual Studio** на вашому комп'ютері. Після запуску відкриється стартове вікно, де можна обрати потрібну дію, а саме - **Create a new project**.

   Якщо стартового вікна немає, ви можете відкрити його, вибравши **File | New | Project** з верхнього меню.
2. Далі у діалоговому вікні, що відкрилося, вибираємо шаблон Windows Forms Application - **Windows Forms App(.NET Framework)** і натискаємо кнопку **Next**



3. У поле **Project name** задаємо ім'я проекту, в поле **Location** вказуємо шлях проекту (можна здійснити вибір за допомогою кнопки **Browse**). Далі натискаємо кнопку **Create.**
4. Відкриється проект:



Інтегроване середовище розробки **(ICP) VS** надає вам форми, на яких розміщуються компоненти. На форму за допомогою миші переносяться і розміщуються піктограми компонентів, наявних в бібліотеках **VS**. За допомогою простих маніпуляцій Ви можете змінювати розміри і розташування цих компонентів. Результати проектування Ви побачите, навіть не компілюючи програму, негайно після виконання якоїсь операції за допомогою миші. Та найголовніша перевага візуального програмування не зводиться до цього. Найголовніше полягає в тому, що під час проектування форми і розміщення на ній компонентів **VS** автоматично формує код програми, включаючи в нього відповідні фрагменти, що описують даний компонент. А потім у відповідних діалогових вікнах користувач може змінити задані за замовчуванням значення якихось властивостей цих компонентів і, при необхідності, написати обробники якихось подій. Тобто проектування зводиться фактично до розміщення компонентів на формі, завданням деяких їх властивостей і написання, при необхідності, обробників подій.

Компоненти можуть бути візуальні (в термінах **Microsoft VS** керуючі елементи), видимі при роботі додатка, і невізуальні (в термінах **Microsoft VS** компоненти), які виконують ті чи інші службові функції. Візуальні компоненти відразу видно на екрані в процесі проектування в такому вигляді, в якому їх побачить користувач під час виконання програми. Невізуальні компоненти видно в процесі проектування у вигляді піктограми, але користувачеві під час

виконання їх не видно, хоча виконують за кадром свою роботу. Візуальні компоненти є елементами призначеного для користувача інтерфейсу. Невізуальні компоненти не мають призначеного для користувача інтерфейсу і не можуть розташовуватися на формі. **VS** розміщує їх внизу вікна дизайнера форми.

Компоненти розташовуються на панелі елементів (**Toolbox**), яка знаходиться зліва від форми в згорнутому вигляді за замовчуванням. На панелі елементів містяться всі керуючі елементи, які доступні розробнику, котрі використовують .**NET**. Зокрема, надається можливість вибору елементів, найбільш важливих для розробки **Windows Application.**

Щоб помістити компонент на форму, треба вказати курсором миші необхідний компонент (натиснути ліву кнопку миші). Перевести курсор миші на форму, а потім натиснути ще раз ліву кнопку миші. Обраний вами компонент буде поміщений на формі в зазначеному місці. Можна це ж зробити іншим способом, а саме, зробити подвійне клацання на обраній піктограмі, тоді він буде поміщений в центрі форми. Непотрібний компонент можна прибрати з форми, якщо клацнути на нього один раз і натиснути клавішу **Delete** на клавіатурі.

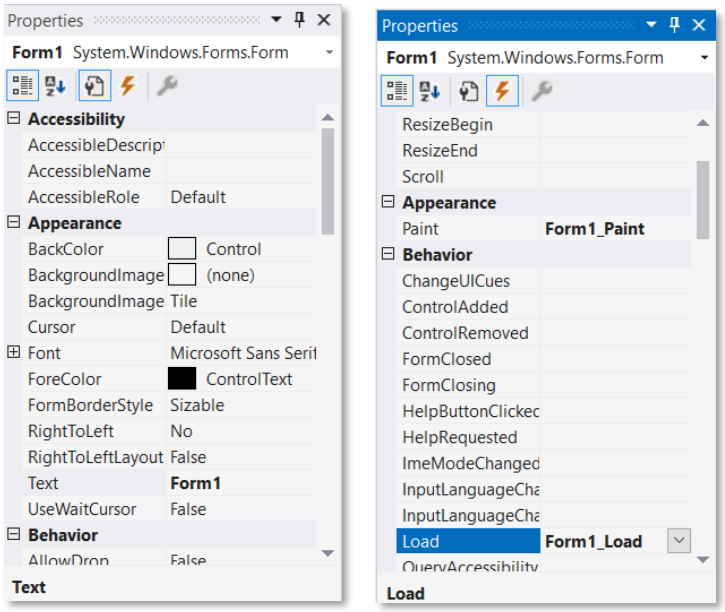
Форма – це основний елемент призначеного для користувача інтерфейсу, основа для побудови програми. Зазвичай форми відображають логічно пов'язані дані і містять інструментальні засоби, що дозволяють користувачеві отримувати інформацію, необхідну для продовження роботи. Будь-яка форма одночасно і клас, тому вдається створювати екземпляри форм і породжувати одні форми від інших.

Компоненти, які ви маєте в своєму розпорядженні на формі: меню, кнопки, індикатори, що випадають списки і т.п. – це об'єкти відповідних класів. Всі компоненти мають ряд властивостей, які представляють собою сукупність даних і методів їх читання і запису. Властивості бувають двох видів: властивості доступні на етапі візуалізації і якості виконання, які доступні тільки під час виконання програми. Властивості першого виду можна змінювати, як програмним шляхом, так і на етапі візуалізації, використовуючи панель Властивості (**Properties**). Безпосередньо в цьому вікні можна змінювати властивості компонента, і він тут же буде міняти свій зовнішній вигляд. Наприклад, якщо змінити координати компонента, то він тут переміститися на

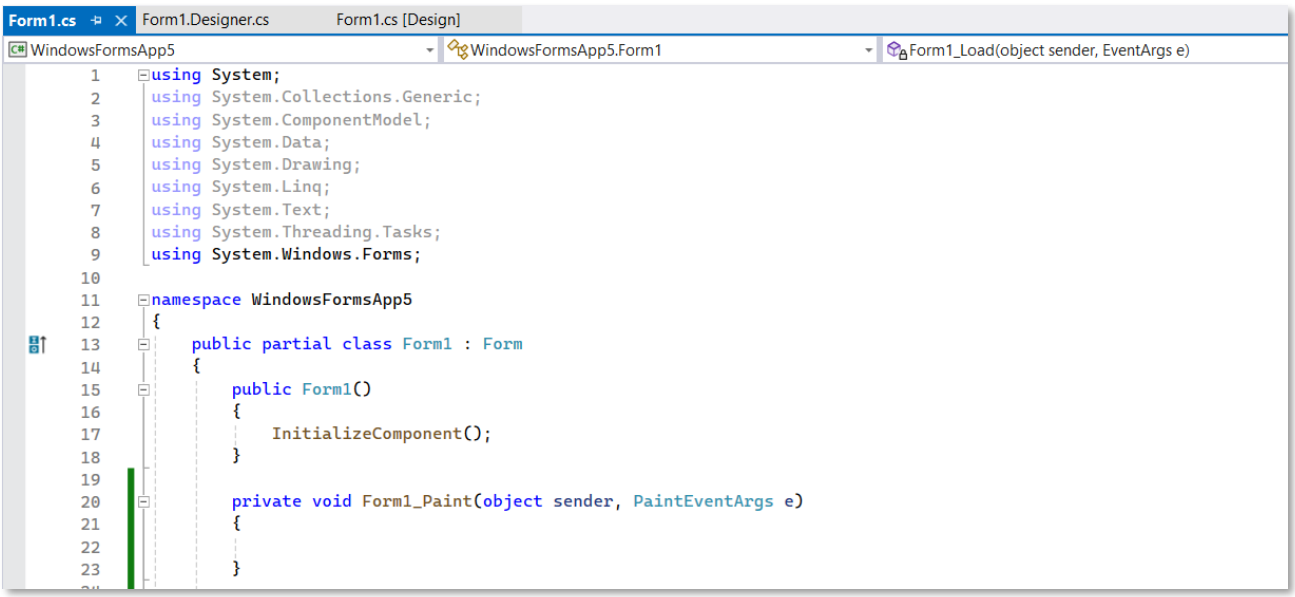
нове місце на формі, якщо змінити колір, то компонент забарвлюється.

Всі об'єкти взаємодіють один з одним. Середовищем взаємодії об'єктів (як би силовим полем, в якому існують об'єкти) є повідомлення, що генеруються в результаті різних подій. Події наступають, перш за все, внаслідок дій користувача – переміщення курсору миші, натискання кнопок мишки або клавіш клавіатури. Але події можуть наступати і в результаті роботи самих об'єктів. У кожному об'єкті визначено безліч подій, на які він може реагувати. У конкретних примірниках об'єкта можуть бути визначені обробники якихось із цих подій, які і визначають реакцію даного екземпляра об'єкта. До написання цих обробників і зводиться основне програмування при розробці графічного інтерфейсу користувача.

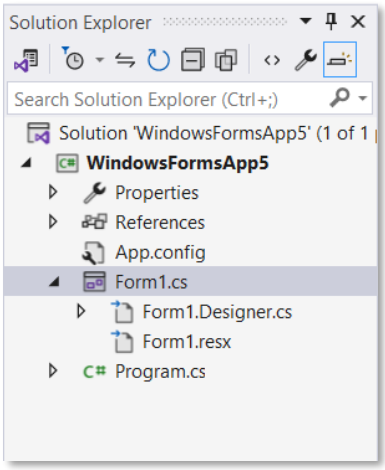
Події обраного компонента також відображає панель Властивості, яка має другий режим роботи Події (**Events**). Перехід з режиму в режим здійснюється за допомогою піктограм, які розташовані на цій панелі і мають відповідні назви.

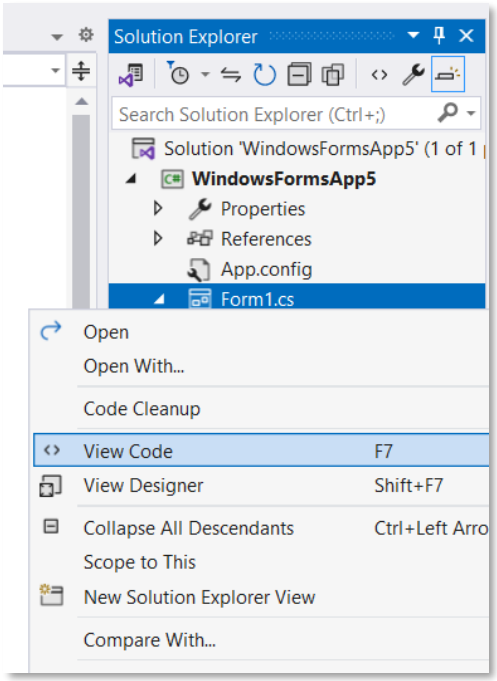


На сторінці Події вказані всі події, на які може реагувати вибраний об'єкт. Щоб створити функцію, що відповідає обраному події, необхідно зробити подвійне клацання на білому полі навпроти обраного події. **VS** згенерує відповідну функцію і відкриє редактор коду.



При цьому стають недоступними панелі елементів і властивостей. Ці панелі доступні тільки, якщо відкрито вікно з проектованої формою - режим конструктора форми. Перехід з вікна у вікно здійснюється за допомогою панелі Оглядач рішень (**Solution Explorer**). При відкритті форми їй за замовчуванням дається ім'я Form1, тому вікно проекту має вигляд





Подвійне клацання на вузлі **Form1.cs** відкриває вікно форми для проектування **Form1.cs [Конструктор] (Form1.cs [Design])**. Для відкриття вікна редактора коду подієвих функцій компонент форми, необхідно викликати контекстне меню вузла і вибрати команду **Перейти до коду (View Code)**. Якщо вікна були вже відкриті, то по ним можна переміщатися за допомогою закладок.

1.2 Приклад простого застосунку.

Для початку створимо застосунок, в якому при натисканні на кнопку буде з’являтися напис на формі. Для цього необхідно зробити наступні кроки:

1. Перенесіть на порожню форму, яка відкрилася перед вами, кнопку типу **Button** зі сторінки **Common Controls** палітри елементів. На формі з'явиться кнопка, якої **VS** присвоїть ім'я за замовчуванням - **button1**.
2. Перенесіть так само з тієї ж сторінки палітри компонентів мітку **Label**. У цій мітці в процесі виконання програми буде з'являтися напис при натисканні кнопки. **VS** дасть їй ім'я **label1**.
3. Виділіть на формі компонент **button1** - кнопку. Перейдіть в панель Властивості і змініть її властивість **Text**, яке за замовчуванням дорівнює **button1** на «Пуск».
4. Клацніть на порожньому місці на формі, передавши їй фокус, у вікні панелі властивостей задайте властивість **Text** рядок «Простий застосунок».
5. Вкажіть мітці **label1**, що написи на ній треба робити жирним шрифтом. Для цього виділіть мітку, у вікні панелі властивостей розкрийте властивість **Font** (шрифт) і встановіть в **true** властивість **Bold** (жирний).
6. Зітріть текст у властивості **Text** мітки **label1**, щоб він не висвічується до тих пір, поки користувач не натисне кнопку застосунку.
7. Виділіть кнопку **button1** на формі, перейдіть в панель властивостей, клацніть на піктограмі подій **Events**, знайдіть подію кнопки **Click** і зробіть подвійне клацання у вікні праворуч від імені цієї події. Ви опинитеся в вікні ***Редактора Коду*** та побачите там текст:

   ```
   private void button1_Click(object sender, EventArgs e)
    {
    }
   ```

   Напишіть в обробнику оператор завдання написи мітки label1:

   ```
   label1.Text = "Привіт, студент!";
   ```
8. Запустіть застосунок.

1.3 Приклад більш складного застосунку.

Створимо застосунок, який при натисканні кнопки перемножує два числа, введені користувачем, і демострує результат обчислення.

1. Перенесіть на форму зі сторінки стандартних елементів управління (**Common Controls**) три вікна редагування **TextBox**, три мітки **Label** і одну кнопку **Button**.
2. Змініть написи в мітках (властивість **Text**) на «Ширина», «Висота» і «Площа». Корисно поставити для міток жирний шрифт, як це було в попередньому прикладі.
3. Замініть властивість **Text** кнопки на «**Розрахунок**».
4. Очистіть властивість **Text** вікна редагування **textBox3**.
5. У властивості **Text** вікон редагування **textBox1** і **textBox2** задайте «**1**» - початкове значення тексту.
6. Напишемо обробник клацання кнопки.

```
private void button1_Click(object sender, EventArgs e)
{
  textBox3.Text = textBox1.Text + "*" + textBox2.Text + "=" +
  Convert.ToString(Convert.ToDouble(textBox1.Text) *
     Convert.ToDouble(textBox2.Text));
}
```

1.4 Керуючі елементи

До компонентів можна звертатися не тільки при розробці програми, але і під час виконання програми. Головне, що слід знати при роботі з компонентами, - те, що до них можна звертатися як до змінних, привласнюючи значення певним властивостям або зчитуючи їх.

Властивості визначають зовнішній вигляд і функціонування компонента управління. Наприклад, якщо потрібно встановити новий напис, то слід змінити властивість **Text**. Як це робиться на стадії проектування, вже було розказано. Програмним шляхом це має вигляд:

```
button1.Text=”Привіт!”;
```

В даному випадку властивості **Text** об'єкта **button1** присвоюється значення ***"Привіт!"***. Зчитування значення властивостей компонент відбувається аналогічним чином. Тому, щоб дізнатися текст на командній

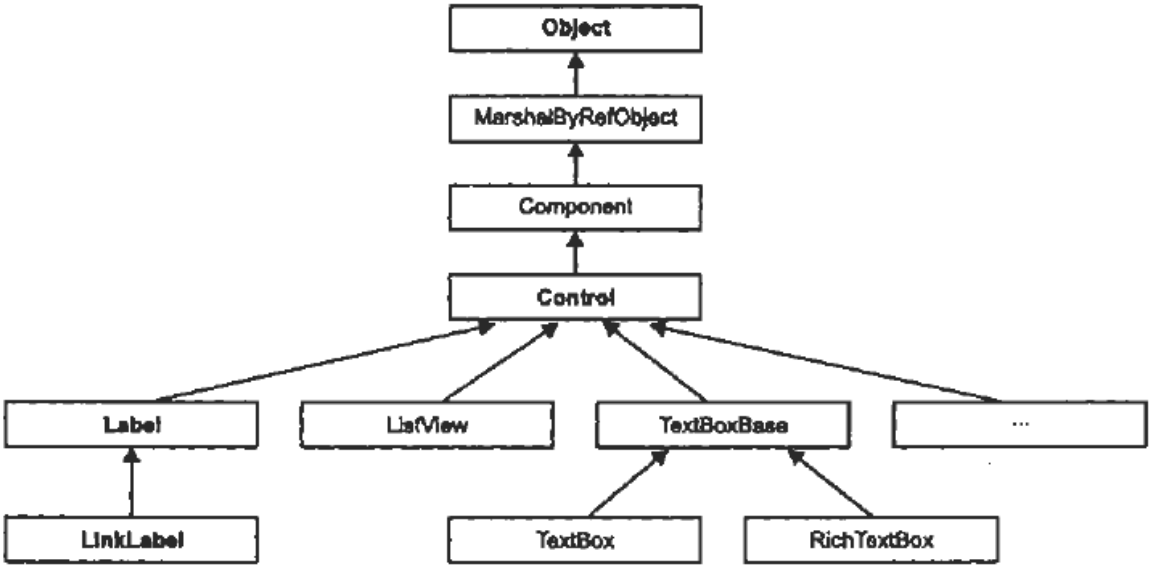
кнопці, досить записати:

textBox1.Text=button1.Text;

У цьому прикладі в текстовому вікні **textBox1** буде відображатися напис командної кнопки **button1**.

Більшість керуючих елементів в .**NET** є похідними від класу **System.Windows.Form.Control**. У ньому визначаються базові функціональні можливості керуючих елементів, і з цієї причини багато властивостей і події є ідентичними. Багато з цих класів самі по собі є базовими для інших керуючих елементів, як у випадку з класами **Label** і **TextBoxBase**, зображеними на діаграмі.

Всі керуючі елементи мають набір властивостей, які використовуються для управління поведінкою цих елементів. Клас **Control**, що є базовим для більшості керуючих елементів, має властивості, які інші елементи або безпосередньо успадковують, або скасовують для досягнення необхідного способу дій.



Розглянемо деякі з найбільш поширених властивостей класу **Control**. Ці властивості будуть присутні в більшості керуючих елементів і нам не доведеться детально зупинятися на їх описі по другому разу, за винятком тих випадків, коли для даного керуючого елемента поведінку будь-якого властивості є відмінним. Списку не буде вичерпним, якщо ви хочете познайомитися з повним переліком властивостей даного класу, то вам слід

звернутися в бібліотеку MSDN.

1. Позиція.

Позицію компонента управління визначають дві властивості: **Location** і **Size**. Кожне з них є підмножиною властивостей. **Size** має дві підвластивості: **Height** і **Width**. Вони задають висоту і ширину компонента. **Location** має два подствойства: **X** і **Y**. Вони задають координати верхнього лівого кута компонента управління. Відлік в системі координат ведеться зверху вниз (**Y**) і зліва направо (**X**). Всі чотири властивості можна змінювати на етапі візуального програмування за допомогою інспектора об'єктів, задаючи їм цілі значення. У режимі виконання програми ці властивості доступні тільки для читання.

```
textBox1.Text=Convert.ToString(textBox2.Size.Height);
```

Програмним шляхом змінювати ці властивості можна тільки за допомогою конструкторів для відповідних типів. Властивість **Location** має тип **Point**, тому командний рядок повинна мати вигляд:

```
textBox1.Location = new Point(200,0);
```

Властивість **Size** має тип **Size**:

```
textBox2.Size = new Size(30, 50);
```

Крім того, є чотири властивості виконання **Left, Top, Right, Bottom**, які доступні тільки під час роботи програми (В інспекторові об'єктів таких властивостей немає).

Left= X; Right= X+ Width;

Top=Y; Bottom=Y+Height;

Властивості **Left, Top** доступні в режимі читання-запису, тобто можна змінювати програмним шляхом, задаючи цілі числа в якості значень.

```
textBox1.Left = 200;
```

Властивості **Right** і **Bottom** доступні тільки для читання.

Приклад.

Подієва функція кнопки «***Позиція***»:

```
private void button1_Click(object sender, EventArgs e)
      {   label1.Text = "X=" + Convert.ToString(textBox1.Location.X);
          label2.Text = "Y=" + Convert.ToString(textBox1.Location.Y);
```

```
        label4.Text = "X=" + Convert.ToString(label3.Location.X);
        label5.Text = "Y=" + Convert.ToString(label3.Location.Y);
        textBox1.Text =
Convert.ToString(textBox1.Bottom)+"="+
Convert.ToString(textBox1.Location.Y)+
"+"+Convert.ToString(textBox1.Height);
label3.Text = Convert.ToString(label3.Right) + "=" +
Convert.ToString(label3.Location.X) +
 "+" + Convert.ToString(label3.Width);        }
```

Подієва функція кнопки «Переміщення»:

```
    private void button2_Click(object sender, EventArgs e)
    {   textBox1.Top = 100;
        label3.Left = 150; }
```

2. Колір.

Властивість **BackColor** визначає колір фону керуючого елемента. Властивість **ForeColor** визначає колір керуючого елемента. За замовчуванням вони мають стандартний колір Windows. Наприклад, текстове вікно має фон – білий (Window), а текст – чорний (WindowText). При бажанні кольори можна міняти, як на етапі візуального програмування, так і програмним шляхом.

Тип даних властивостей **Color**, тому командний рядок матиме вигляд:

```
textBox1.ForeColor = Color.Blue;
```

3. Доступність і видимість компонента управління.

Часто при роботі програми потрібен зробити недоступними для користувача деякі компоненти управління. Для цього використовуються властивості – **Enabled і Visible.**

Властивість **Enabled** визначає, чи буде компонент управління реагувати на подію чи ні. Якщо значення властивості рівне **false**, компонент управління буде не доступний і користувач не зможе його використовувати. Зазвичай при цьому компонент підсвічується сірим кольором, так само, як елементи меню, які можна вибрати.

Властивість **Visible** дозволяє зробити елемент управління невидимим. Якщо його значення дорівнює **false**, то його не видно і звернутися до нього не можна.

Вибір властивості **Visible** або властивості **Enabled** залишається за вами. Якщо

ви вибираєте властивість **Enabled**. Це означає, що елемент управління є, але звернутися до нього не можна поки. А властивість **Visible** дозволяє «приховати» елемент від користувача.

4. Властивість Name.

Властивість **Name** відіграє особливу роль. Помилки при його завданні призводять до серйозних наслідків. Ім'я є ідентифікатором компонента управління. З цього імені до керуючого, елементу можна звертатися програмним шляхом. При написанні застосунку бажано змінювати імена компонентів на розумні, не залишаючи імена за замовчуванням, які не пояснюють призначення компонентів**.**

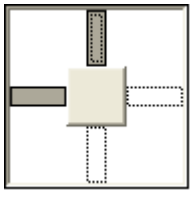
5. Властивість Tag.

Це властивість зазвичай не використовується самим керуючим елементом; в ньому може зберігатися інформація по самому керуючому елементі. Якщо значення цієї властивості присвоюється за допомогою програми, то допустимим є тільки текстовий рядок.

5. Властивості Anchor і Dock.

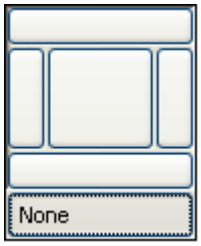
Ці дві властивості виявляються особливо корисними при розробці форми. Завдання гарантувати, що вікно не прийме безладний вид в тому випадку, якщо користувач змінить його розміри, виявляється далеко не тривіальним, і для її вирішення було написано незліченну кількість рядків коду. Безліч програм вирішують цю проблему, накладаючи заборону на зміну розміру вікна, що, безумовно, є найбільш простим, але не найкращим способом впоратися з цим завданням. Властивості **Anchor і Dock**, які з'явилися разом з .NET, дозволяють вирішити цю проблему, не написавши жодного рядка коду.

Властивість **Anchor** дозволяє визначити, яким чином повинен поводитися керуючий елемент в той момент, коли користувач змінює розмір вікна. За замовчуванням, незмінними є **Top і Left.**



Властивість **Dock** може бути використано для того, щоб керуючий елемент прилягав до краю свого контейнера. У тому випадку, якщо користувач змінить розміри вікна, елемент все одно залишиться прилеглим до його краю. Якщо, наприклад, задано, що керуючий елемент повинен прилягати до нижньої межі

вікна, то він буде завжди змінювати своє місце розташування так, щоб займати нижню частину екрану незалежно від розмірів вікна. Розмір же керуючого елемента змінюватися при цьому не буде – він завжди буде прилягати до відповідного краю вікна.



1.5 Основні події.

Коли користувач натискає на кнопку або клацає по ній мишею, то вам як програмісту-розробнику цього застосунку, хотілося б отримати інформацію про те, що це сталося. З цією метою керуючі елементи використовують події. Клас **Control** визначає кілька подій, які є загальними для всіх елементів, розглянемо основні.

1. **Події клацання миші.**
   Є дві події, що викликаються клацанням миші: **Click і DoubleClick.**
   Подія **Click** викликається, як тільки користувач виконає клацання на компоненті управління.
   Подія **DoubleClick** викликається подвійним клацанням кнопкою миші на компоненті управління. Часовий інтервал між клацаннями подвійного клацання встановлюється в панелі управління Windows. Використання події **Click** для деяких керуючих елементів, таких як керуючий елемент **Button**, означає, що подія **DoubleClick** ніколи не буде викликатися.
2. **Події, пов'язані з мишею.**
   Подія **MouseDown** викликається при натисканні кнопки миші. Подія **MouseUp** викликається при відпуску кнопки миші. Подія **MouseMove** викликається, коли користувач пересуває курсор миші.
3. **Події клавіатури.**
   Існує три події **KeyDown, KeyUp, KeyPress**. Вони викликаються при натисканні або при відпуску клавіші. Після натискання клавіші події наступають в такій послідовності: **KeyDown, KeyPress, KeyUp**.

2. ФОРМА ТА СТАНДАРТНІ КОМПОНЕНТИ

2.1 Форма

Форма є основним компонентом програми. Спочатку розглянемо деякі властивості форми, події і методи.

FormBorderStyle.

Повертає або задає стиль кордону форми. Стилем кордону форми визначається спосіб відображення її зовнішнього краю. Крім зміни відображення кордону форми, певні стилі кордонів запобігають зміна розміру форми. Дана властивість у вікні властивостей має список, що випадає значень, які можна використовувати на етапі візуального програмування. За замовчуванням, властивість **FormBorderStyle** встановлено в **Sizable** - вікно із змінними розмірами. Якщо встановити властивість **FormBorderStyle** в **FixedSingle**, то це буде вікно з незмінними кордонами. Значення має тип **FormBorderStyle,** яке можна використовувати в програмному коді:

```
this.FormBorderStyle = FormBorderStyle.Sizable;
```

(Використовувати покажчик **this** не обов'язково).

Якщо форма залишається змінною, то добре було б ставити її максимальний і мінімальниі розміри за допомогою властивостей **MaximumSize** і **MinimumSize**. Ці властивості за замовчуванням мають значення нуль, що означає - немає ніяких обмежень.

```
MaximumSize = new Size(500,700);
MinimumSize = new Size(125, 135);
```

StartPosition.

Повертає або задає початкове положення форми в режимі виконання. Розташування форми встановлюється вручну або за замовчуванням відповідно до настанов Windows. Є можливість розташувати форму в центрі екрану або в центрі її батьківської форми, якщо форма є дочірньою MDI-формою. Ця властивість має бути встановлена в конструкторі форми.

Віконне меню в заголовку форми.

Властивість **ControlBox** повертає або задає значення, яке вказує, відображається чи ні кнопка віконного меню в рядку заголовка форми.

Властивості **MaximizeBox** і **MinimizeBox** повертають або задають значення, що вказують, чи відображається кнопка розгортання/згортання в рядку заголовка форми. Всі властивості булевого типу.

Розмір форми при запуску програми.

Властивість **WindowState** повертає або задає стан вікна форми. За замовчуванням встановлено параметр **Normal**. Програмним шляхом ця властивість змінюється командою виду:

WindowState = FormWindowState.Maximized;

Подієва функція відображення форми.

До початкового відображення форми виконується подія **Load**. Якщо ви хочете виконати якісь дії в цей момент, то ви повинні це написати в подієвій функції **Form1_Load ().**

Метод закриття форми.

Щоб закрити форму необхідно використовувати метод **Close ().**

Напис.

Напис в заголовку форми визначається властивістю **Text**.

2.2 Керуючий елемент Button

У Visual Studio.NET є три типи кнопок. Перемикачі та вікна з вибором також є кнопки. З цієї причини клас **Button** не є похідним безпосередньо від класу **Control**, він є похідним від класу **ButtonBase**, який, в свою чергу, є похідним від класу **Control**. Зараз ми познайомимося з компонентом **Button**, а до перемикачів і вікон для вибору перейдемо пізніше.

Керуючий елемент у вигляді кнопки присутній практично у всіх діалогах **Windows**. Кнопки використовуються насамперед для виконання завдань наступних трьох видів:

• Для завершення діалогу зі збереженням стану (наприклад, кнопки **ОК** і **Скасування**).

• Для виконання будь-яких дій над даними, введеними в процесі діалогу (наприклад, при натисканні кнопки **Пошук** після введення деякого критерію пошуку).

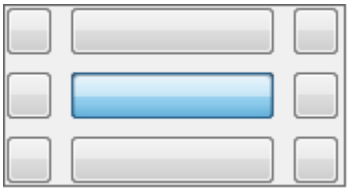
• Для відкриття іншого діалогу або додатки (наприклад, кнопка **Допомога**).

Вся робота полягає в додаванні елемента **Button** до форми і двічі клацнути мишею на ньому для введення коду, який займається обробкою події **Click**, -

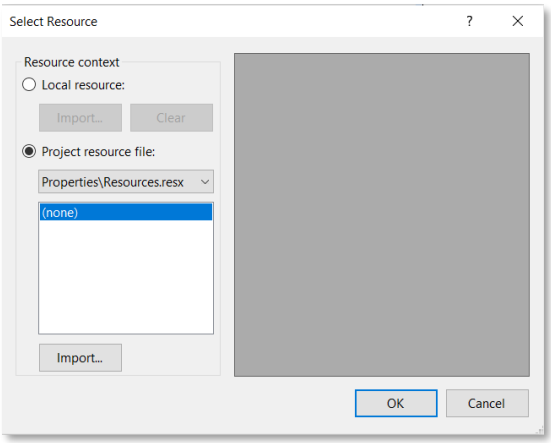
цього виявляється цілком достатньо для більшості додатків, з якими ви будете працювати.

Властивості:

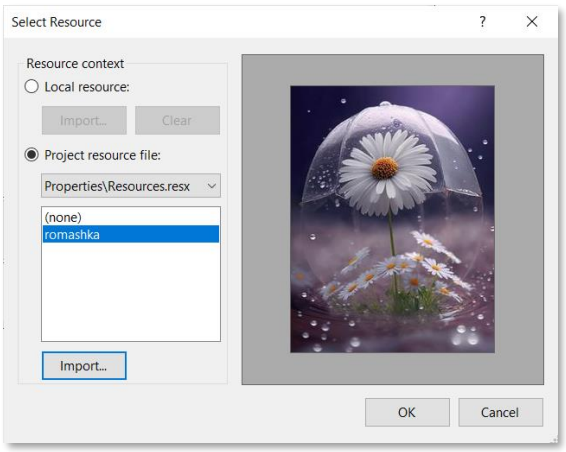
1. **Text** – Напис на кнопці.
2. **TextAlign** – Визначає, яким чином текст розташовується в керуючому елементі. За замовчуванням має значення **MiddleCenter**, що означає вирівнювання по центру



3. **Image** – Дозволяє задати малюнок (в двійковому вигляді, у вигляді іконки і т.д.), який буде зображуватися на кнопці. При виборі в вікні властивостей поле значення цієї властивості з'являється кнопка з трьома крапками. Такі кнопки означають, що для даної властивості є відповідний вбудований редактор, який відкривається при натисканні на три крапки поряд



4. У лівій частині вікна є список ресурсних файлів, якщо він порожній, то можна в нього імпортувати файл за допомогою кнопки **Import.** Після вибору файлу необхідно натиснути кнопку **ОК.**



5. **ImageAlign** − Дозволяє визначати місце розташування малюнка, що розміщується на кнопці. Вибір здійснюється аналогічно властивості **TextAlign**.
6. **AutoSize** − булевого типу. Якщо воно **true**, то розміри компонента відповідають розмірам картинки, яка розміщена на кнопці (якщо вона завантажена на компонент).
7. **FlatStyle −** Дозволяє змінювати стиль кнопки. Якщо задати стиль **Popup**, то кнопка буде виглядати плоскою до тих пір, поки користувач не наведе на неї курсор миші. У цей момент кнопка прийме свій звичайний об'ємний вигляд.

Найбільш часто використовуваною подією для кнопок є подія **Click**. Ця подія настає, коли користувач клацає мишею на кнопці, що означає, що він натискає ліву кнопку миші і відпускає її, поки курсор миші знаходиться на даному керуючому елементі. Звідси, зокрема, випливає, що в тому випадку, якщо спочатку буде натиснута ліва кнопка миші, а потім курсор миші буде поведений з керуючого елемента до того, як вона буде відпущена, то це не призведе до настання події **Click**. Крім того, подія **Click** виникає, якщо дана кнопка знаходиться у фокусі, а користувач натискає клавішу **Enter**. Якщо у вашій формі використовується будь-яка кнопка, то ви повинні неодмінно обробляти цю подію.

Якщо на формі розташовано кілька кнопок, то можна одну з них зв'язати з натисканням клавіші **Enter**, а іншу **Esc**. Для цього у форми є властивості **AcceptButtton** і **CancelButton** відповідно. Обидві ці властивості мають список, що випадає, з іменами всіх кнопок розташованих на формі. Необхідно зв'язати ці властивості з наявними кнопками. Допускається використання одного і того ж компонента в двох властивостях.

2.3 Керуючий елемент Label

Керуючий елемент **Label** є найбільш вживаним. Він є практично у всіх додатках Windows. **Label** - простий елемент, призначений для єдиної мети: він являє собою напис або коротку рекомендацію, яка пояснює що-небудь користувачеві даної форми.

Керуючий елемент **Label** володіє великою кількістю властивостей, які можна задавати. Більшість з них є похідними від класу **Control.**

1. **BorderStyle** − Дозволяє визначати стиль обрамлення навколо **Label**. За замовчуванням обрамлення відсутня.
2. **FlatStyle, Image, ImageAlign, Text** та **TextAlign** як у **Button**.
3. Властивість **AutoSize** встановлено в **true** забезпечує автоматичну зміну розмірів компонента згідно з текстом, розташованому на мітці.

2.4 Керуючий елемент TextBox

Керуючі елементи типу **TextBox** (текстове вікно) слід використовувати в тих випадках, коли користувач вводить текст, про який нічого не відомо на момент розробки (наприклад, ім'я користувача). Основне завдання текстових вікон - дозволити користувачу вводити текст, але, оскільки допускається введення довільних символів, можна змусити користувача вводити, наприклад, тільки цифри.

Керуючий елемент **TextBox** є похідним від базового класу **TextBoxBase**, який, своєю чергою, є похідним від класу **Control**. Клас **TextBoxBase** забезпечує базові функціональні можливості по роботі з текстом в текстовому вікні, такі як вибір тексту, вирізання та вставка тексту, а також широкий спектр подій.

1. **Text -** Отримує або задає текст, зіставлений з цим елементом управління.
2. **MaxLength** − Значення, яке визначає максимальну довжину тексту **TextBox**, виражену в символах. Якщо довжина обмежена тільки об'ємом наявної пам'яті, даній властивості слід привласнити значення, рівне нулю.
3. **Multiline** − Вказує, чи допускає даний керуючий елемент наявність декількох рядків. Складний елемент дозволяє одночасне виведення декількох рядків тексту. За замовчуванням ця властивість має значення **false.**

   Якщо дозволити багатостроковий текст, то у вікні властивостей **Text** можна задавати у вигляді списку рядків, який при введенні рядок від

рядка відокремлює символом Enter.

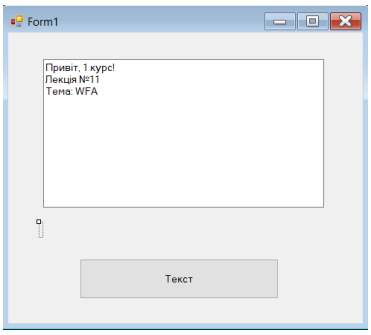
Програмним шляхом багатостроковий текст можна задавати за допомогою команди виду:

```
textBox1.Text="Перший рядок" + Environment.NewLine + "Другий рядок";
```

Тут використовується статичний клас **Environment** і його властивість **NewLine**, яка повертає рядок, що позначає початок нового рядка.

4. **Lines** – Отримує або задає рядки тексту в елементі управління ***"Текстове поле"***. Являє собою масив рядків. На етапі візуального програмування може задаватися за допомогою вбудованого редактора рядків, який з'являється при натисканні на три крапки у вікні властивостей. За замовчуванням колекція рядків є доступною тільки для читання копією рядків в елементі управління **TextBox**. Для отримання колекції рядків з можливістю запису використовуйте програмний код, подібний з наступним кодом:

```
textBox1.Lines = new string[] { "abcd" };
```



Приклад 1.

Розглянемо приклад програми, яка використовує властивість **Lines**. На етапі візуального програмування форма повинна мати вигляд:

Оголосимо в класі форми дві змінні цілого типу:

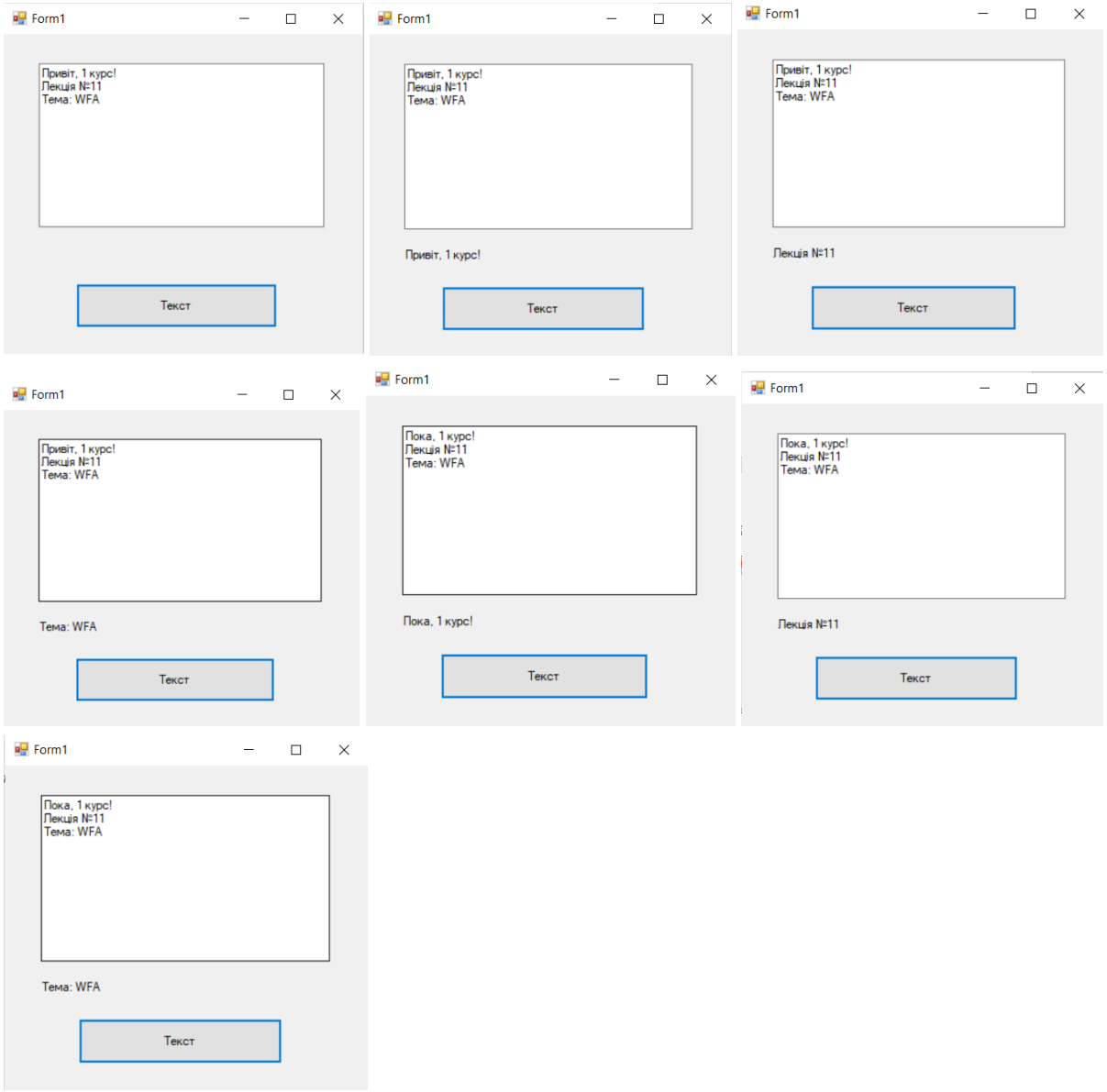
```
int i = 0;
int j = 0;
```

Подієва функція для кнопки «Текст» має вигляд:

```
private void button1_Click(object sender, EventArgs e)
{           if (i == textBox1.Lines.Length)
          {
             if (j == 1) return;
             string[] myLines = textBox1.Lines;
             string stroka = "Пока" + textBox1.Lines[0].Substring(6);
             myLines[0] = stroka;
```

```
            textBox1.Lines = myLines;
            i = 0;
            j++;
        }
        label1.Text = textBox1.Lines[i++];
}
```

При послідовних клацання на кнопці «Текст» форма має вигляд:



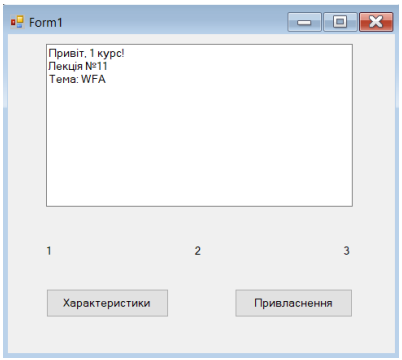
Подальші клацання на цій кнопці ні до чого не приводять.

5. **ScrollBars -** Дозволяє вказати, що вікно, яке складається з декількох рядків, має виводитися з лінійками прокрутки. За замовчуванням прокручувань немає, але можна встановити вертикальну, горизонтальну

або обидві прокрутки. Даній властивості відповідає однойменне перерахування, тому програмним шляхом команда виглядає:
textBox1.ScrollBars = ScrollBars.None;

6. **WordWrap** − Визначає, чи потрібно в багаторядковому вікні автоматично переносити слова на новий рядок в тому випадку, якщо довжина рядка перевищує ширину керуючого елемента.
7. **SelectedText** − Визначає обраний у вікні текст.
8. **SelectionLength** − Визначає число символів в обраному тексті. Якщо присвоєне цій властивості значення перевищує кількість символів в тексті, воно буде змінено на різницю між кількістю символів в тексті і значенням властивості SelectionStart.
9. **SelectionStart** − Визначає початок виділеного тексту в текстовому вікні.

Приклад 2. Розглянемо приклад програми використовує три останніх властивості. На етапі візуального програмування форма повинна мати вигляд:



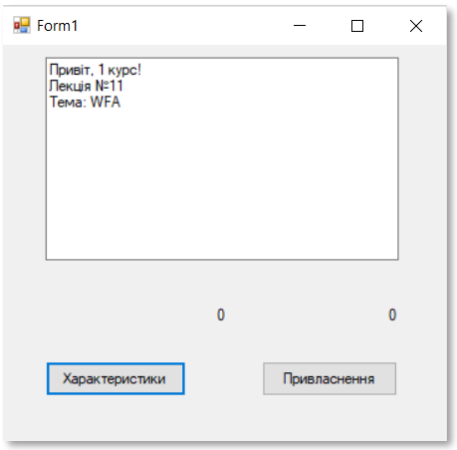
Подієва функція для кнопки «***Характеристики***» має вигляд:

```
private void button1_Click(object sender, EventArgs e)
{       label1.Text=textBox1.SelectedText;
        label2.Text = Convert.ToString(textBox1.SelectionLength);
        label3.Text = Convert.ToString(textBox1.SelectionStart);       }
```

Подієва функція для кнопки «***Привласнення***» має вигляд:

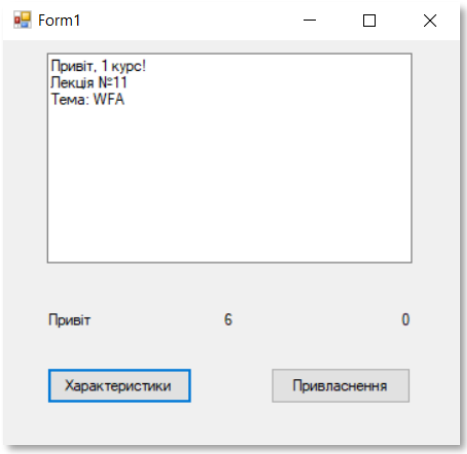
```
private void button2_Click(object sender, EventArgs e)
{       textBox1.SelectionLength = 7;
        textBox1.SelectionStart = 7;
        textBox1.SelectedText = " лектор";                              }
```

При запуску програми і натиснути кнопку «***Характеристики***» маємо:



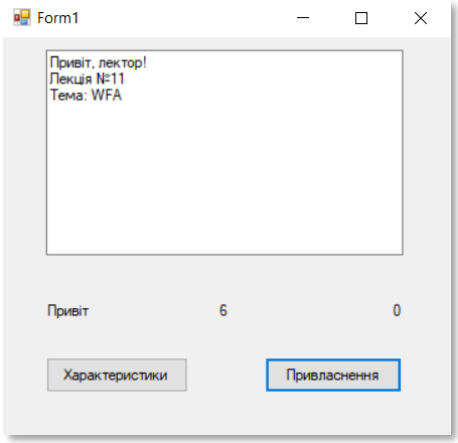
Результат: немає виділеного тексту, положення курсору за замовчуванням - на початку тексту, тобто нуль, кількість виділених символів - 0.

Виділимо текст *Привіт* і натиснемо на кнопку «***Характеристики***», маємо:



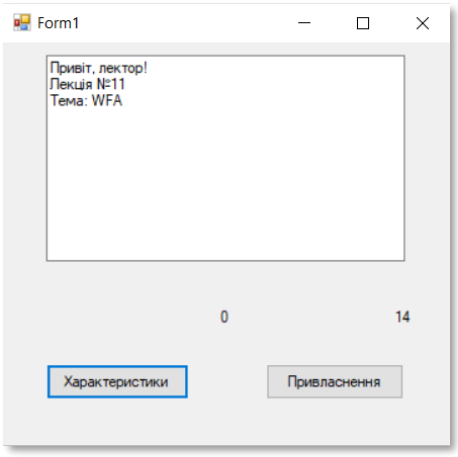
Результат: виділений текст *Привіт*, кількість символів - 6, номер позиції - 0.

Якщо тепер натиснути кнопку «Привласнення», то матимемо:



Результат: Курсор перемістився на 7 позицію, починаючи з нуля, і вставився текст, який знаходився у властивості **SelectedText**.

Запустимо програму і натиснемо кнопку «Привласнення», а потім кнопку «Характеристики»:

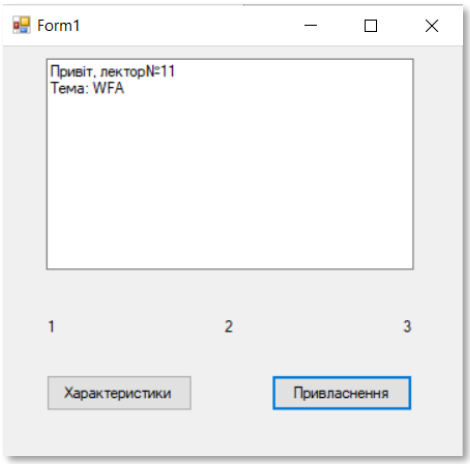


Питання: Поясніть результат?

Внесемо зміни в код подієвої функції кнопки «Присвоєння»:

```
private void button2_Click(object sender, EventArgs e)
  {       textBox1.SelectionLength = 17;
          textBox1.SelectionStart = 7;
          textBox1.SelectedText = " лектор";    }
```

Запустимо програму і натиснемо кнопку «Привласнення», маємо:



Питання: Поясніть результат?

10. **ReadOnly** – Логічне значення, яке вказує на те, що текст може використовуватися в режимі "тільки читання", тобто як компонент відображення тексту.
11. **PasswordChar** – Дозволяє приховати текст, що вводиться, заміщаючи його зазначеним символом.

Подія **TextChange** настає при будь-яких змінах тексту в текстовому вікні незалежно від їх характеру.

Компонент має ряд методів, які можна використовувати в програмі:

1. **Clear()** - Видаляє весь текст з елемента управління.
   textBox1.Clear();
2. **AppendText() -** Додає текст в кінець поточного тексту в текстовому полі.
   textBox1.AppendText("Привіт");
3. **SelectAll() -** Обирає весь текст в текстовому полі.
   textBox1.SelectAll();
4. **Select()** − Вибирає текст з вказаної позиції, на даний кількість символів.
   textBox1.Select(15, 7);

При виборі тексту підсвічування синім кольором не відбувається, але обраний текст автоматично заноситься в властивість **SelectedText.**

2.5 Керуючі елементи HScrollBar та VScrollBar

Дані керуючі елементи розташовані на палітрі в закладці **All Windows Forms**. Обидва компоненти є спадкоємцями класу **ScrollBar**, тому мають однакові властивості, методи і події. Являють собою вертикальну і горизонтальну прокрутки, які можуть використовувати для завдання значень будь-яких параметрів із заданого діапазону. У багатьох візуальних компонент є свої власні лінійки прокрутки, які не вимагають написання спеціального коду. Ми розглядали можливість завдання прокручувань для керуючого компонента **TextBox**. У форми є булева властивість **AutoScroll**, яке за замовчуванням стоїть в **false**. Якщо встановити **true**, то при зміні розмірів форми, якщо якісь компоненти виходять з області видимості, то автоматично з'являються горизонтальна або вертикальна прокрутки.

Властивості прокручувань:

1. **Maximum, Minimum** − Повертає або задає верхню (нижню) межу значень діапазону прокрутки.
2. **Value** − Повертає або задає числове значення, що представляє поточний стан повзунка на смузі прокрутки.
3. **SmallChange** − Повертає або задає значення, яке додається або віднімається з властивості **Value**, коли користувач клацає мишею по кнопках зі стрілками (на кінцях лінійки) або натисканням клавіші позиціонування.

4. **LargeChange** – Повертає або встановлює значення, яке додається або віднімається з властивості **Value**, коли користувач клацає мишею на самій лінійці (по обидва боки від бігунка).

Подія **Scroll** відбувається під час переміщення повзунка за допомогою миші або клавіатури. Подія **ValueChange** відбувається під час зміни властивості **Value** або внаслідок події **Scroll**, або програмно.

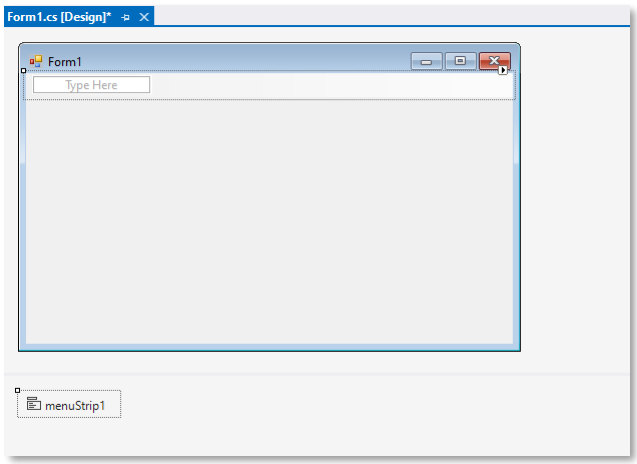
2.6 Меню

Цікаво, скільки застосунків **Windows** ви зможете згадати, в яких би не було якогось меню? Найімовірніше, що таке число буде дуже близьким до нуля. Отже, меню можна уявити собі як дуже важливу складову частину будь-якої програми, яка створюється для використання під операційною системою **Windows**. Для полегшення роботи зі створення меню для додатків в **Visual Studio.NET** передбачений керуючий елемент, який дозволяє дуже швидко створювати прості меню, витрачаючи не набагато більше зусиль на створення складних.

Керуючий елемент меню, який ми будемо використовувати, називається **MenuStrip**. Він є похідним від базового класу з ім'ям **Menu**. Другим різновидом меню, яке буде вивчатися, є **ContextMenuStrip,** виведене в той момент, коли користувач клацає правою кнопкою миші на будь-якому об'єкті.

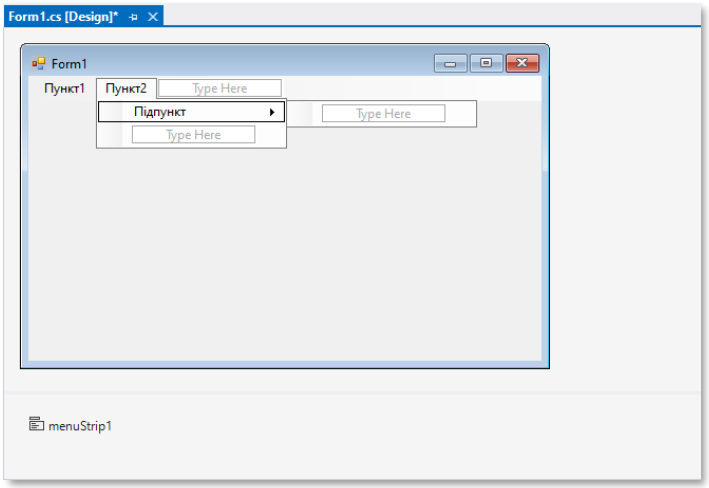
Якщо спробувати перенести керуючий елемент **MenuStrip** з панелі інструментів (**Menus** & **Toolbars**) на робочу поверхню, то можна помітити, що одночасно відбувається розміщення, як на самій формі, так і під нею. **MenuStrip** виступає свого роду контейнером для окремих пунктів меню, які представлені об'єктом **ToolStripMenuItem**.

Для додавання є три види елементів: **MenuItem** (об'єкт **ToolStripMenuItem**), **ComboBox** і **TextBox**. Таким чином, в меню ми можемо використовувати випадаючі списки і текстові поля, однак, як правило, ці



елементи застосовуються в основному на панелі інструментів. Меню ж зазвичай містить набір об'єктів **ToolStripMenuItem**.

Для того щоб створити нові пункти меню, потрібно просто перемістити курсор у вікно з написом **Type Here**. Введіть назву меню в це вікно.

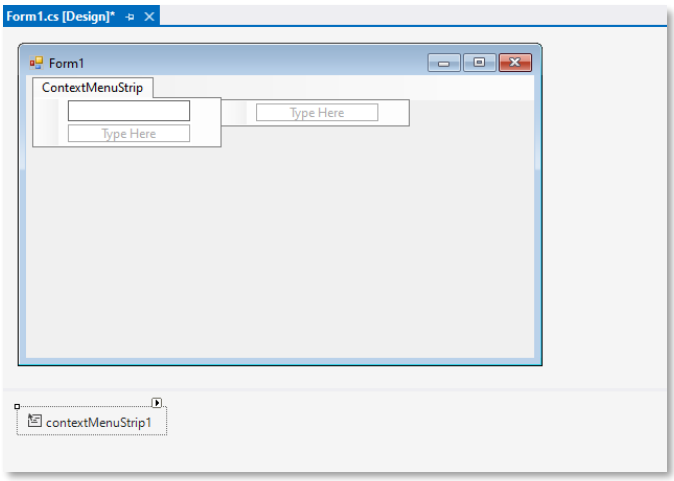


Як роздільник при наборі тексту в меню використовується дефіс (-). Це неприпустимо для пунктів меню самого верхнього рівня, а може застосовуватися тільки для пунктів меню нижчих рівнів. Введення одного дефіса генерує розділову лінію.

Якщо клацнути на піктограмі компонента на панелі невізуальних компонентів, то вікно властивостей відображає властивості меню. Якщо клацати мишею на виділеному пункті меню, то можна помітити, що властивості, що виводяться на панелі властивостей, змінилися. Це відбувається тому, що більшість властивостей, які можуть використовуватися при роботі з меню, мають відношення до окремих пунктів меню. Кожен пункт меню може містити в собі інші пункти меню, що дозволяє створювати підменю для кожного пункту основного меню. Кожен пункт меню має тип **ToolStripMenuItem.** його властивості:

1. **Checked** − вказує, чи є даний пункт меню обраним, якщо обраний, то ставиться галочка.
2. **ShortcutKeys** − це властивість визначає комбінацію клавіш, яка дозволяє здійснювати безпосередній доступ до пунктів меню (наприклад, **Ctrl+S** для пункту меню **Save**).
3. **ShowShortcutKeys** − присвоєння цій властивості значення **true** означатиме, що комбінація клавіш для безпосереднього доступу буде виводитися в **ToolStripMenuItem** в текстовому вигляді.

Властивість **Click** настає, коли користувач клацає мишею на пункті меню. Створити контекстне меню можна за допомогою перенесення такого меню на форму і додавання до нього необхідних пунктів – точно так само, як це може бути зроблено з керуючим елементом **MenuStrip**.



Єдина відмінність полягає в тому, що в контекстному меню не допускається наявність пунктів самого верхнього рівня. Для того щоб прив'язати контекстне меню до якого-небудь керуючому елементу в формі, слід вибрати відповідний керуючий елемент і привласнити його властивості **ContextMenuStrip** – ім'я створеного вами контекстного меню з наявного випадаючого списку.

2.7 Керуючі елементи RadioButton та CheckBox

Керуючі елементи **RadioButton** (перемикач) і **CheckBox** (вікно для вибору) мають спільне з керуючим елементом **Button** походження, хоча їх зовнішній вигляд і використання істотно від цього елемента відрізняються.

Перемикачі традиційно представлені у вигляді напису з крапкою, розташованою від них з лівого боку, які можуть бути або обрані, або не обрані. Перемикачі рекомендується використовувати, коли необхідно надати користувачеві можливість вибору з декількох взаємовиключних варіантів. Наприклад, обов'язкова відповідь – «**Так**» або «**Ні**».

Керуючий елемент **CheckBox** традиційно являє собою напис з маленьким віконцем із прапорцем, яке розташовується зліва від неї. Цей керуючий елемент слід використовувати, коли необхідно надати користувачеві можливість вибору однієї або декількох можливостей.

Властивості компонента **RadioButton**:

1. **CheckAling** – дозволяє змінювати місце розташування перемикача за допомогою графічного інтерфейсу:

   Програмним шляхом змінюється за допомогою команди виду:

   ```
   radioButton2.CheckAlign = ContentAlignment.BottomCenter;
   ```

2. **Checked** – визначає стан керуючого елемента. Його значення дорівнює **true**, якщо у даного керуючого елемента встановлено прапорець вибору, і **false** в іншому випадку.
3. **Appearance** – керуючий елемент **RadioButton** може виводитися у вигляді напису з круглим прапорцем вибору, розташованим зліва, в середині або праворуч від неї, або у вигляді стандартної кнопки. В останньому випадку кнопка виглядає втопленою, якщо її обрано, і опуклою – якщо ні. За замовчуванням має значення **Normal**, для зовнішнього вигляду у вигляді кнопки використовується значення – **Button**.

   Програмним шляхом змінюється за допомогою команди виду:

   ```
   radioButton2.Appearance = Appearance.Button;
   ```

Події компонента **RadioButton:**

1. **Click –** повідомлення про настання цієї події вирушає всякий раз, коли відбувається клацання мишею на **RadioButton**. Це зовсім не те ж саме, що подія, пов'язана зі зміною стану керуючого елемента, оскільки клацання мишею на будь-якому керуючому елементі два або більше разів поспіль приводить тільки до однократної зміни властивості "обрано" – і то тільки в тому випадку, якщо вона не була вибрана до цього.
2. **Checkchanged** – повідомлення про настання цієї події відправляється в той момент, коли прапорець вибору **RadioButton** змінює свій стан. Якщо в рамках форми або групового вікна існує більше одного керуючого елемента **RadioButton**, то повідомлення про цю подію буде відправлятися двічі: спочатку керуючому елементу, який до цього був обраний і тепер таким не є, а потім керуючому елементу, який був обраний в даний момент.

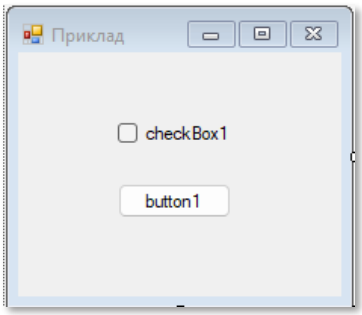
Події компонента **CheckBox:**

1. **CheckState** – на відміну від **RadioButton, CheckBox** може перебувати в одному з трьох станів:
   - **Checked** (обрано)
   - **Indeterminate** (не визначено),
   - **Unchecked** (не обрано)

   Якщо стан вікна вибору описується як **Indeterminate**, то вікно вибору, відповіднє до цього напису, зазвичай зображується сірим кольором, що означає, що поточне значення у вікні вибору або неприпустимо, або позбавлене сенсу при даних обставинах.
2. **ThreeState** – коли ця властивість має значення **false**, то немає можливості змінювати стан **CheckBox** на **Indeterminate**. Хоча при цьому як і раніше можна встановлювати дане значення програмним шляхом. Це значення прийнято за замовчуванням.
3. **Checked** – визначає стан керуючого елемента. Його значення дорівнює **true**, якщо у даного керуючого елементу встановлено прапорець вибору, і **false** в іншому випадку.

Приклад 1. Розглянемо додаток з використанням компонента CheckBox:

На етапі візуалізації форма має вигляд:

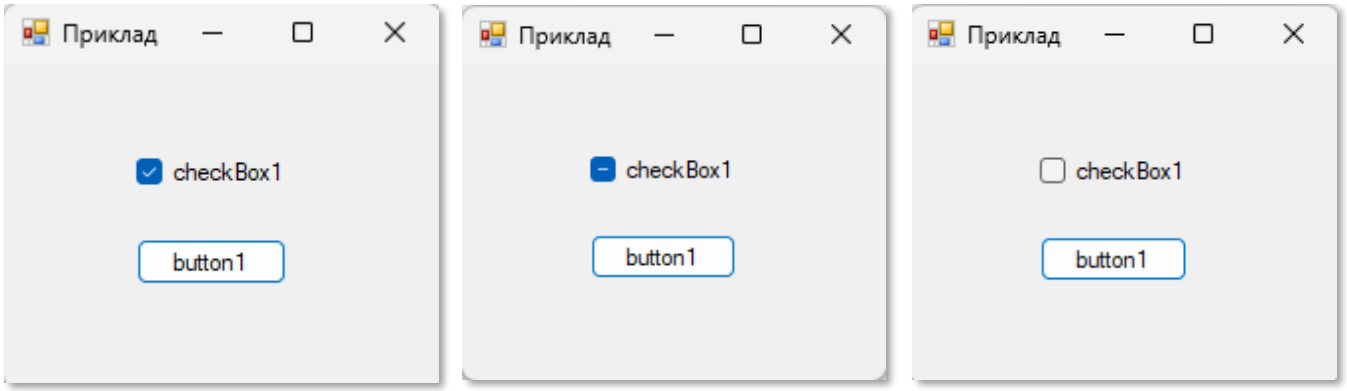


Програмний код має вигляд:

```
13      public partial class Form1 : Form
14      {
15          int k = 0;
            1 reference
16          public Form1()
17          { InitializeComponent(); }
18
            1 reference
19          private void button1_Click(object sender, EventArgs e)
20          {
21              switch (k)
22              {
23                  case 0: checkBox1.CheckState = CheckState.Checked; break;
24                  case 1: checkBox1.CheckState = CheckState.Indeterminate; break;
25                  case 2: checkBox1.CheckState = CheckState.Unchecked; break;
26              }
27              k++;
28          }
29      }
```

Результат:

Перше клацання Друге клацання Третє клацання



Подальші клацання на кнопці не призводять до зміни стану вікна вибору.

Питання: Як змінити код, щоб зміна стану вікна вибору відбувалося постійно?

Події компонента **CheckBox:**

1. **Checkchanged –** настає при кожній зміні значення властивості **Checked** вікна з вибором. Слід звернути увагу на те, що для керуючого елемента **CheckBox**, у якого властивість **ThreeState** має значення **true,** існує можливість клацати мишею на вікні вибору без зміни значення властивості **Checked**. Це відбувається, коли вікно вибору змінює свій стан з "вибрано" на "не визначено".
2. **CheckStateChanged** – настає при кожній зміні значення властивості **CheckedState**. Оскільки і значення **Checked**, і значення **Unchecked** є допустимими значеннями властивості **CheckedState**, то це повідомлення про цю подію буде відправлятися кожен раз, коли значення властивості **Checked** змінюється. Крім того, воно також буде відправлятися і в тому випадку, якщо стан змінюється зі значення **Checked** на значення **Indeterminate**.

2.8 Пошук «зниклих» компонентів

Іноді буває, що ви не можете знайти на формі компонент, який, як ви знаєте, на ній присутній. Наприклад, якщо ви використовуєте мітку **Label,** встановивши в ній властивість **AutoSize** в **true** і стерши значення напису **Text,** то горизонтальний розмір мітки зменшиться до нуля і її не буде видно на формі, поки під час виконання програми не зміниться властивість **Text**. Можливий такий вибір кольору, що компонент зливається з фоном.

Знайти «зниклий» компонент можна, вибравши його ім'я в списку, що

випадає, розташованому у верхній частині вікна **Properties**. Цей список містить всі компоненти, розміщені на формі. Якщо ви виберете в ньому потрібний компонент, то на формі навколо нього з'явиться рамка з маркерами. При цьому у вікні **Properties** стануть видні сторінки властивостей і подій знайденого компонента.

3. КОНТЕЙНЕРИ. ФОРМИ. TIMER

Зазвичай радиокнопки розміщуються всередині попередньо встановленого на формі групового контейнера. Якщо обрана одна з кнопок, вибір всіх інших кнопок в тій же групі автоматично знімається. Кілька радіокнопок можуть бути обрані одночасно тільки в тому випадку, коли вони розміщені в різних контейнерах. Контейнери можуть виконувати як чисто декоративні функції, візуально об'єднуючи компоненти, пов'язані один з одним за призначенням, так і функції управління, організовуючи спільну роботу своїх дочірніх компонентів. Ми сьогодні розглянемо контейнери **GroupBox** і **Panel**.

Використання елемента, керуючого групою вікон, не представляє ніякої складності – досить помістити його на форму, а потім перенести в нього ті керуючі елементи, які повинні в ньому міститися. Результатом цього є те, що керуючі елементи належать груповому вікну.

Компонент **GroupBox** створює контейнер у вигляді прямокутної рамки, візуально об'єднує логічно пов'язану групу деяких інтерфейсних елементів. Компонент має вбудовану рамку з написом, яка зазвичай використовується для виділення на формі групи функціонально об'єднаних компонентів. За напис відповідає властивість **Text**.

Об'єкт **GroupBox** не може відобразити смугу прокрутки. Якщо необхідний елемент управління, схожий з **GroupBox**, який може містити смугу прокрутки, то використовується елемент управління **Panel**.

Елемент управління **Panel** за замовчуванням відображається без кордонів. Властивість **BorderStyle** дозволяє відобразити стандартну або тривимірну рамку, щоб виділити область панелі серед інших областей на формі. Оскільки елемент управління **Panel** виводиться з класу **ScrollableControl** за допомогою властивості **AutoScroll**, можна включити смуги прокрутки в елементі управління **Panel**. Якщо властивість **AutoScroll** має значення true, всі елементи управління, розташовані в **Panel** (але поза його видимою межою), можна прокручувати за допомогою присутніх смуг прокрутки.

Елемент управління **Panel** не відображує заголовок. Якщо необхідний елемент управління, схожий з **Panel**, який може відображати заголовок, то використовується елемент управління GroupBox.

3.1 Батьки компонентів – Parent

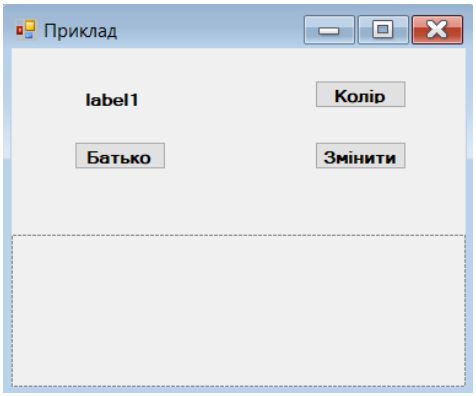
Віконний компонент – форма, панель і т. д., що включає в себе як контейнер інші компоненти, виступає по відношенню до них як батьківський компонент. У кожного компонента на формі є батько. У процесі виконання програми ви можете дізнатися з батьків того чи іншого компонента по властивості **Parent**. Це властивість можна читати і змінювати тільки під час виконання, у вікні властивостей ви його не знайдете.

Що дає поняття батьківського компонента? Значення властивостей **Left** і **Top** вимірюються в системі координат батьківського компонента. Таким чином, при переміщенні батьківського компонента будуть синхронно переміщатися і всі його дочірні компоненти. Якщо в процесі виконання застосунку зробити в батьківському компоненті властивість **Visible** рівним **false**, то стане невидимим не тільки батьківський, а й всі його дочірні компоненти. За аналогією з властивістю **Enabled**.

Якщо ви перенесли компонент з бібліотеки не на форму, а на панель, то ця панель стає для неї батьківського. Ви ніякими пересуваннями не зможете перемістити його за межі батьківської панелі. Якщо ж ви передумали розміщувати компонент на даній панелі і хочете перемістити його на іншу панель або безпосередньо на форму, то це можна зробити через буфер обміну.

Зауваження. У форми немає батька, дана властивість для форми є нульовим посиланням.

Приклад. Розглянемо додаток з використанням властивості **Parent**. На етапі візуального програмування форма має вигляд:

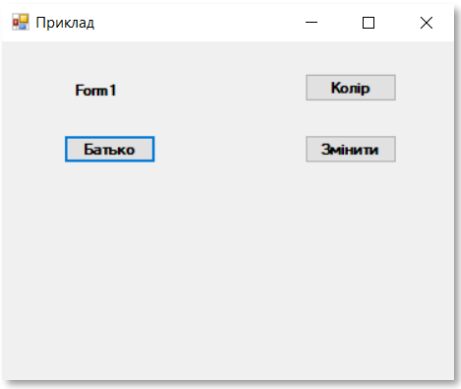


Код має вигляд:

```
Form1.cs | Form1.cs [Design]
WindowsFormsApplication1 | WindowsFormsApplication1.Form1 | k
10  namespace WindowsFormsApplication1
11  {
12      public partial class Form1 : Form
13      {
14          int k = 0;
15          public Form1()
16          {
17              InitializeComponent();
18          }
19          //кнопка "Батько"
20          private void button1_Click(object sender, EventArgs e)
21          {
22              label1.Text = label1.Parent.Name;
23          }
24          //кнопка "Змінити"
25          private void button2_Click(object sender, EventArgs e)
26          {
27              label1.Parent = panel1;
28          }
29          //кнопка "Колір"
30          private void button3_Click(object sender, EventArgs e)
31          {
32              if (k == 0)
33                  label1.Parent.BackColor = Color.Green;
34              else
35                  label1.Parent.BackColor = Color.Blue;
36              k++;
37          }
38      }
39  }
```

Результат:

Після клацання на кнопці «***Батько***»:

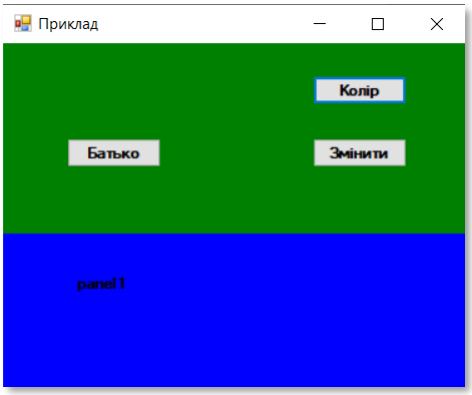


Після клацання на кнопці «***Колір***»:

Після клацання на кнопці «***Змінити***»:

Після клацання на кнопці «***Батько***»:

Після клацання на кнопці «***Колір***»:



3.2 Застосунок з декількома формами

Основним елементом будь-якої програми є форма-контейнер, в якому розміщуються інші візуальні і невізуальні компоненти. Для кожної форми **VS** додає в файл програми **Program.cs**, відповідний оператор її створення вигляду: Application.Run(**new** Form1());

До початкового відображення форми відбувається подія **Load**, потім подія **VisibleChanged**. Подія **Shown** відбувається перед самим відкриттям форми. Тут можна провести всі необхідні налаштування, поки користувач не

побачив форму на моніторі.

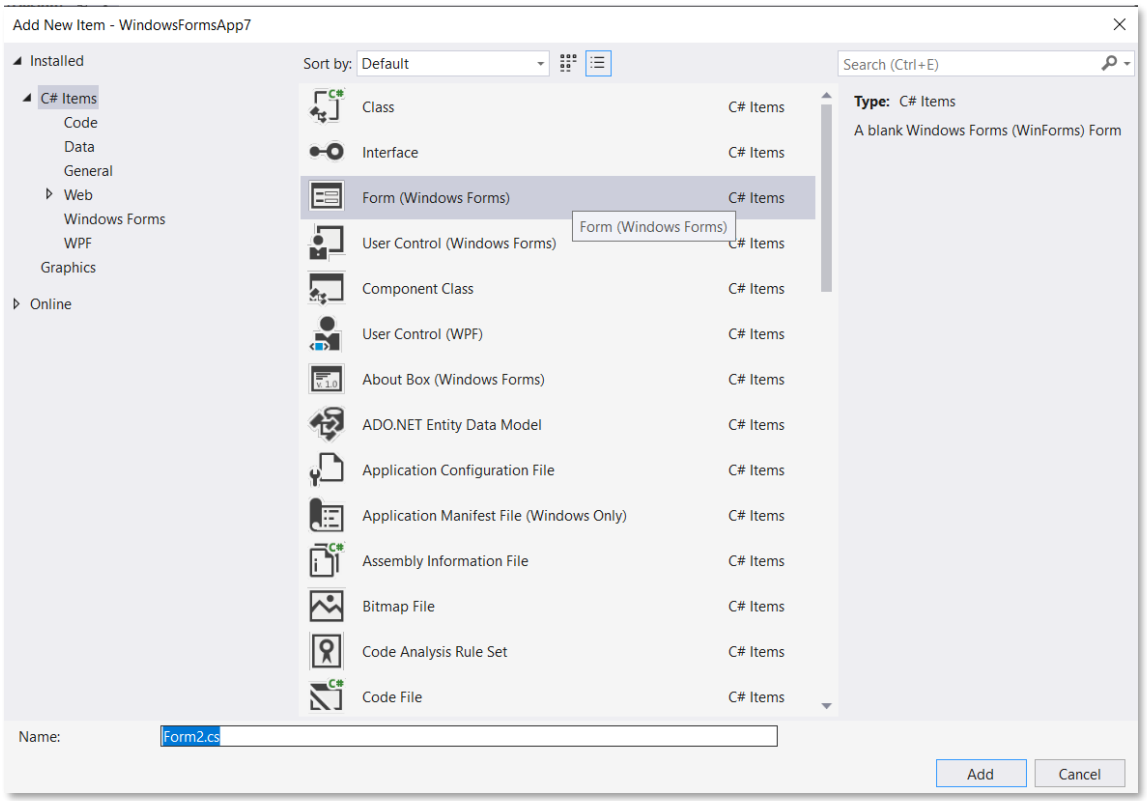
Кожній формі відповідає свій клас в додатку. Написання класу відбувається автоматично, при чому формуються одночасно два файли: **Form1.cs** і **Form1.Designer.cs**. Файл **Form1.cs** містить оголошення класу і коструктор

```
public Form1()
    {   InitializeComponent();  }
```

Сам метод **InitializeComponent()** оголошений в файлі F**orm1.Designer.cs** і код пише **VS**.

До сих пір ми мали справу тільки з застосунками, які мають по одній формі, але в багатьох випадках ваш проект буде містити не одну, а кілька форм. Додати форму в додаток можна за допомогою команди
Project | Add Windows Form ...

або в вікні *Оглядача рішень* викликати контекстне меню проекту і знайти команду **Add | Form (Windows Forms)**. Відкриється діалогове вікно виду з обраним шаблоном **Windows Forms** і з ім'ям форми за замовчуванням:



Необхідно натиснути кнопку **Add** (Додати).

При додаванні нової форми в додаток автоматично не додається команда

створення даної форми. В застосунок додається новий клас **Form2** і відповідні йому файли **Form2.cs** і **Form2.Designer.cs**. Для створення нової форми необхідно використовувати команду виду

```
Form2 f = new Form2();
```

за допомогою якої створюється об'єкт **f** класу **Form2**. У потрібний момент форму можна зробити видимою методами **Show()** або **ShowDialog()**. Останній метод відкриває форму як модальну. При виконанні методів **Show()** і **ShowDialog() –** виникає подія форми **Shown**.

Відкриття форм як модальних використовується в більшості діалогових вікон. Модальна форма призупиняє виконання процедури, що її викликала, доти, доки користувач не закриє цю форму. Модальна форма не дозволяє також користувачеві переключити фокус курсором миші на інші форми цього застосунку, поки форма не буде закрита. Модальною може бути зроблена будь-яка форма, якщо вона робиться видимою методом **ShowDialog()**. Якщо та ж сама форма робиться видимою методом **Show(),** то вона не буде модальною.

У будь-якому **Windows**-застосунку існує поняття головної форми, тобто форми, закриття якої призводить до завершення роботи програми. Видно, що при розказаній схемі побудови програми головною буде форма, яка перша додана в застосунок. У разі необхідності можна це змінити, виправивши командний рядок в файлі **Program.cs:**

```
Application.Run(new Form2());
```

Написання такої команди означає, що головною тепер буде форма **Form2**.

Відкриті додаткові форми необхідно закривати в ході виконання програми (особливо модальні). Це можна зробити двома способами – методами **Close()** і **Hide()**. Метод **Hide()** робить форму невидимою, а метод **Close()** її закриває. Коли метод **Close()** викликається для допоміжного вікна, то неможливо потім викликати метод **Show()** для відображення цієї форми, так як ресурси форми вже відключені. Для приховування та відображення форми використовується метод **Hide().** Якщо ж форма повинна з'явитися тільки один раз в процесі виконання програми, наприклад, форма запиту пароля, то її можна закрити методом **Close().**

Метод **ShowDialog()** має прототип: `public DialogResult ShowDialog();`

За значенням цього методу, що повертається, можна визначити, яким чином було закрито вікно. Тип **DialogResult** – перерахування, що містить наступні константи: **None, Ok, Cancel, Abort, Retry, Ignore, Yes, No**. Однойменна властивість є у кнопок типу **Button** у *Вікні властивостей*. Значення цієї властивості вибираються зі списку, що містить такі ж константи. Для кнопок, розташованих на формі, можна задати відповідне значення властивості **DialogResult**. Натискання на такий кнопці призводить до того, що форма закривається, а повертається значення методу **ShowDialog(),** відповідає значенню DialogResult кнопки. Константа **None** відповідає моменту відкриття форми в модальному режимі. Значення **Cancel** також повертається в разі закриття форми за допомогою системної кнопки або за допомогою методів **Close(), Hide().**

При закритті форми настає подія **FormClosing**. Якщо форма закривається, вона віддаляється, що звільняє всі пов'язані з нею ресурси. При скасуванні цієї події форма залишається відкритою. Для скасування згортання форми для властивості Cancel вхідного параметра **e** типу **FormClosingEventArgs**, переданого обробнику подій, встановлюється значення **true**.

```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
    {  e.Cancel = true; }
```

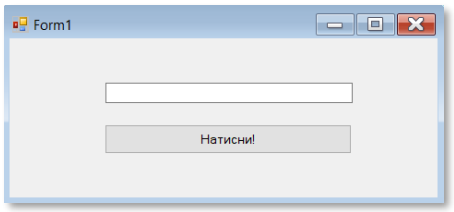
Є можливість перевизначити значення, призначене властивості **DialogResult** форми, якщо користувач клацає кнопку закриття, встановлюючи властивість **DialogResult** в обробнику подій для події форми **FormClosing**.

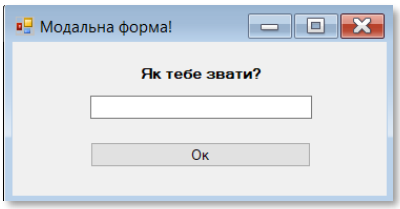
```
private void Form2_FormClosing(object sender, FormClosingEventArgs e)
    {   this.DialogResult = DialogResult.Cancel;   }
```

3.3 Приклад програми з модальної формою

Розташуємо на формі **Form1** одне вікно редагування **textBox1** і одну командну кнопку **button1**. Задамо для кнопки **Text = "*Натисни мене*".** Для **textBox1** встановимо **Text = ""** і **Visible = false**. Нехай при натисканні на button1 буде відкривати модальна форма. Для цього додамо в проект нову форму Form2, на якій розташуємо **label1**, **textBox1** і **button1**. Встановимо для **textBox1** властивість, для **label1** властивість **Text = "*Як тебе звуть?*"**, А для **button1** властивість **Text = "*ОК*"**. Для кнопки **button1** встановити властивість

DialogResult в **OK**. Для форми **Form2** встановити властивість **AcceptButton** в **button1**. Користувач повинен ввести своє ім'я в **textBox1** на формі **Form2** і натиснути клавішу **OK**, після чого модальна форма закриється і на формі **Form1** у вікні редагування **textBox1** з'явиться напис "***Привіт, (ім'я введене користувачем)***".





Код програми для **Form1** може мати вигляд:

```
13      public partial class Form1 : Form
14      {
15          static public string myName { get; set; }
16          public Form1()
17          {
18              InitializeComponent();
19          }
20
21          private void button1_Click(object sender, EventArgs e)
22          {
23              Form2 myf2 = new Form2();
24              if (myf2.ShowDialog() == DialogResult.OK)
25              {
26                  textBox1.Text = "Привіт, " + myName;
27                  textBox1.Visible = true;
28              }
29          }
30
31      }
```

Для форми **Form2**:

```
13      public partial class Form2 : Form
14      {
15          public Form2()
16          {
17              InitializeComponent();
18          }
19          private void button1_Click(object sender, EventArgs e)
20          {
21              Form1.myName = textBox1.Text;
22          }
23      }
```

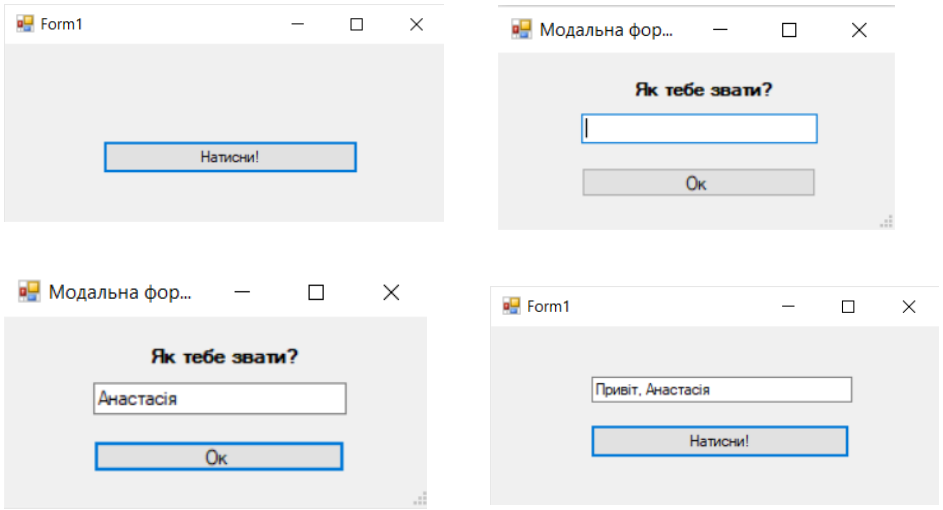
Зауваження:

1. *У класі форми **Form1** оголошена статична змінна **myName**, яка необхідна для обміну інформацією між додатками, тому вона оголошена*

зі специфікатором ***public.***

2. *Оскільки змінна* ***myName*** *статична, то в класі форми Form2 можна до неї звертатися безпосередньо через ім'я класу.*

На етапі виконання додаток має вигляд:



3.4 Володіння вікнами

В **.NET** вікно може "володіти" іншими вікнами. Вікна, що мають вікно-власника, зручно застосовувати для плаваючих вікон панелей інструментів і вікон команд. Одним із прикладів такого вікна є вікно **Find and Replace** (Знайти і замінити) в **Microsoft Word**. Коли вікно-власник згортається, вікно, яким воно володіє, теж автоматично згортається. Коли вікно перекриває вікно, яке ним володіє, воно завжди відображається зверху.

Для підтримки володіння вікном клас **Window** пропонує дві властивості: **Owner** і **OwnedWindows**. Властивість **Owner** є посилання, яке вказує на вікно, що володіє поточним вікном (якщо таке є), а властивість **OwnedWindows** – колекцію всіх вікон, якими володіє поточне вікно (знову-таки, якщо вони є). Налаштування володіння вікна має на увазі просто установку властивості **Owner**, як показано нижче.

Внесемо зміни в текст програми. Для форми **Form1**:

```
13      public partial class Form1 : Form
14      {
15          public string myName { get; set; }
16          public Form1()
17          {
18              InitializeComponent();
19          }
20
21          private void button1_Click(object sender, EventArgs e)
22          {
23              Form2 myf2 = new Form2();
24              myf2.Owner = this;
25              if (myf2.ShowDialog() == DialogResult.OK)
26              {
27                  textBox1.Text = "Привіт, " + myName;
28                  textBox1.Visible = true;
29              }
30
31          }
32      }
```

Для форми **Form2**:

```
13      public partial class Form2 : Form
14      {
15          public Form2()
16          {
17              InitializeComponent();
18          }
19          private void button1_Click(object sender, EventArgs e)
20          {
21              (this.Owner as Form1).myName = textBox1.Text;
22          }
23      }
```

Результат буде таким самим.

3.5 Загальні відомості про фокус

Фокус клавіатури відноситься до елемента, який отримує введення даних з клавіатури. На всьому робочому столі одночасно може існувати тільки один елемент, який має фокус введення. У додатку елемент, який має фокус введення, буде мати для властивості **Focused** значення **true.** Фокус клавіатури може бути отриманий переходом до елементу або клацанням миші на таких елементах, як, наприклад, **textBox**. Фокус клавіатури також може бути отриманий програмним способом за допомогою використання методу **Focus()**. При цьому властивість **Focused** встановлюється в **true** автоматично при передачі фокусу. Даний метод необхідно використовувати, якщо при відкритті форми необхідно, щоб фокус завжди знаходився на одному і тому ж компоненті. Такий підхід часто використовується в вікнах-діалогах.

У розглянутому прикладі у допоміжному вікні фокус при відкритті форми

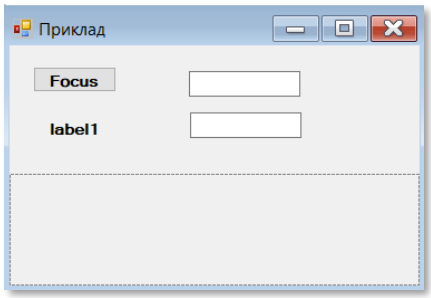
знаходиться на кнопці, що є концептуально неправильним: очікується введення інформації користувачем в текстове вікно, отже, фокус необхідно перевести. Це можна здійснити в подієвій функції **Shown** форми **Form2**:

```
private void Form2_Shown(object sender, EventArgs e)
    {   textBox1.Focus();  }
```

Результат: тепер фокус знаходиться в текстовому вікні введення імені**.**

Слід враховувати, що якщо **Visible** компонента дорівнює **false**, то не виникає помилки.

Приклад. На етапі візуального програмування форма має вигляд, представлений на рисунку. Друге вікно має властивість **Visible = false.**

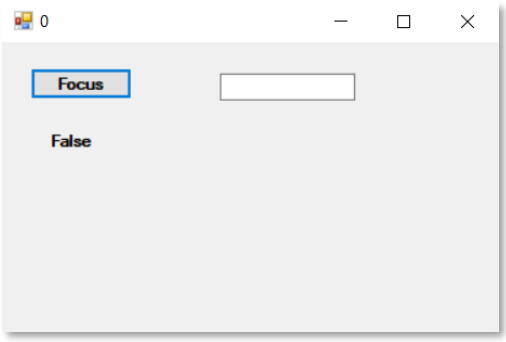


Програмний код має вигляд:

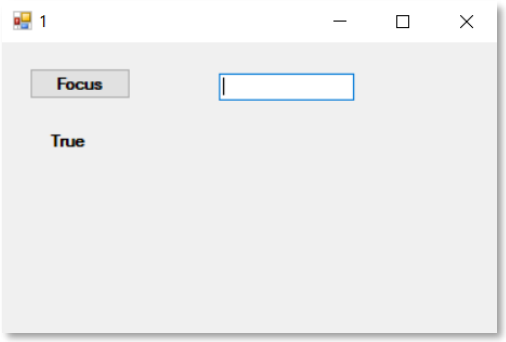
```
public partial class Form1 : Form
{
    int k = 0;
    public Form1()
    {
        InitializeComponent();
    }
    private void button1_Click(object sender, EventArgs e)
    {
        switch (k % 4)
        {
            case 0:
                Text = "0";
                label1.Text = Convert.ToString(textBox1.Focused);
                break;
            case 1:
                Text = "1";
                textBox1.Focus();
                label1.Text = Convert.ToString(textBox1.Focused);
                break;
            case 2:
                Text = "2";
                panel1.Focus();
                label1.Text = Convert.ToString(panel1.Focused);
                break;
            case 3:
                Text = "3";
                textBox2.Focus();
                label1.Text = Convert.ToString(textBox2.Focused);
                break;
        }
        k++;
    }
```

Результат:

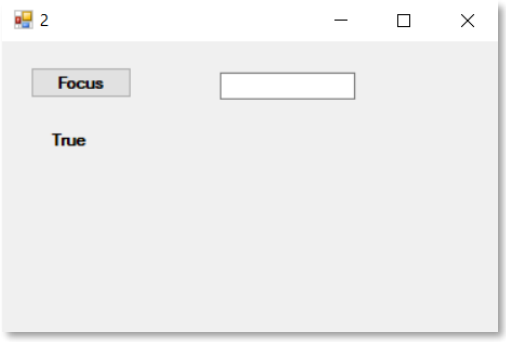
Після першого клацання на кнопці:



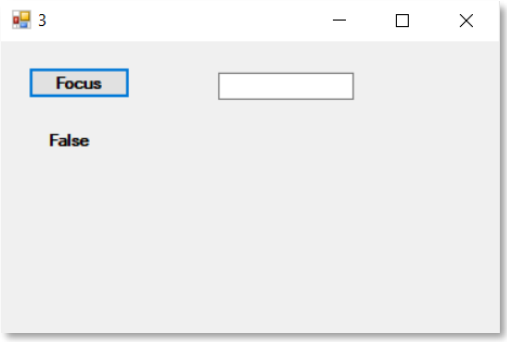
Після другого клацання



Після третього клацання



Після четвертого клацання



На етапі візуального програмування в міру розміщення компонентів на формі будується так звана послідовність табуляцій. Кожному компоненту присвоюється порядковий номер (нумерація починається з 0), який автоматично заноситься у властивість компонента **TabIndex**. Ця послідовність використовується при передачі фокусу від компонента до компонента за допомогою клавіші **Tab**. При первинному відкритті форми фокус передається першому компоненту в послідовності табуляцій.

Все це відноситься до компонентів, які мають властивість **TabStop**, тобто на які в принципі може передаватися фокус за допомогою клавіші **Tab**. Наприклад, компонент **Label** не має такої властивості взагалі, а компонент **Panel** має, але зі значенням за замовчуванням **false**, у компонентів **TextBox** і **Button** ця властивість дорівнює **true**.

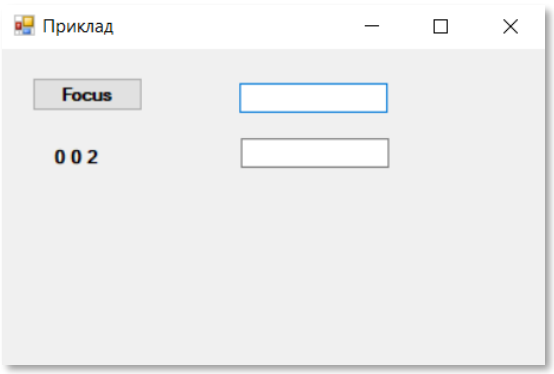
Наприклад, в попередньому прикладі першій в послідовності табуляцій є кнопка, тому під час запуску програми фокус спочатку передається на неї. Змінимо у другого текстового вікна властивість **Visible** на **true**. При натисканні клавіші **Tab**, фокус послідовно переміщається по колу: **textBox1**, **textBox2, button1**.

Змінимо властивість першого текстового вікна **TabStop** на **false**. Тоді при натисканні клавіші **Tab** фокус буде передаватися тільки між компонентами **textBox2** і **button1**.

Недолік в середовищі розробки **VS**: можна двом компонентам задати однакове значення для властивості **TabIndex**. Стає незрозумілим принцип переміщення по послідовності. Наприклад, в попередньому прикладі за допомогою вікна властивостей задаємо для компонента **textBox1** значення властивості **TabIndex = 0**. При цьому у компонента **button1** так само залишається **TabIndex = 0**. Для перевірки – додамо в застосунок подієву функцію **Shown**

```
private void Form1_Shown(object sender, EventArgs e)
    {
     label1.Text = Convert.ToString(button1.TabIndex) + " " +
                   Convert.ToString(textBox1.TabIndex) + " " +
                   Convert.ToString(textBox2.TabIndex);
    }
```

Після запуску програми маємо:



***Зауваження**:*

1. *Фокус спочатку знаходиться на компоненті **textBox1**.*
2. *Згідно з міткою дійсно два компоненти мають однакове значення властивості **TabIndex**.*

3.6 Компонент Timer

Клас **Timer** реалізує таймер, який викликає подія через певні інтервали часу, задані користувачем. Компонент даного класу є невізуальним компонентом, тому розташовується на панелі нижче вікна форми, так само як і компонент **MenuStrip**. Таймер знаходить широке застосування в графіці. Він має дві властивості, що дозволяють їм управляти: **Interval** – інтервал часу в мілісекундах і **Enabled** – доступність.

Властивість **Interval** задає період спрацьовування таймера. Через заданий інтервал часу після попереднього спрацювання або після запуску програми, якщо значення **Interval** встановлено під час проектування, таймер спрацьовує, викликаючи подію **Tick**. У обробнику цієї події записуються необхідні оператори.

Якщо необхідно в деякій процедурі запустити таймер, який відрахував б заданий інтервал часу, наприклад, 5 секунд, після чого треба виконати деякі операції і відключити таймер, це можна зробити наступним чином. Під час проектування задається значення **Interval** = 5000 і значення **Enabled** = false. У момент, коли треба запустити таймер виконується оператор

```
timer1.Enabled = true;
```

В обробник події Tick, яке настає через 5 секунд після запуску таймера, можна вставити оператор `timer1.Enabled = false;` який відключить таймер.

3.7 Функції-члени класу форми

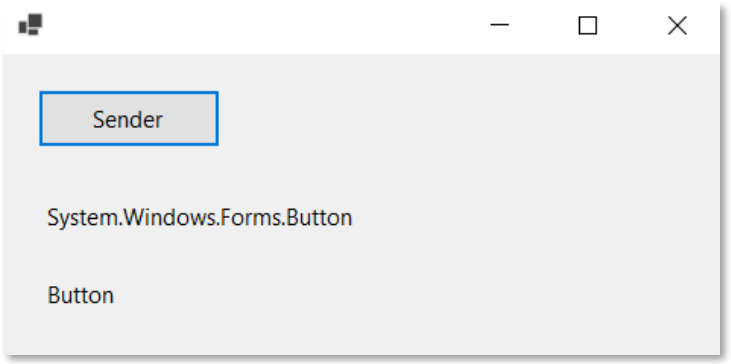
При створенні подієвих функцій **VS** автоматично генерує визначення функції. У всі обробники подій передається параметр **sender** типу **Object**. Цей параметр містить компонент, в якому відбулася подія. Він не потрібен, якщо пишеться обробник події для одного конкретного компонента. Однак часто один обробник застосовується для декількох компонентів. При цьому якісь операції можуть бути загальними для будь-яких джерел події, а якісь вимагати специфічних дій. Тоді **sender** можна використовувати для розпізнавання джерела події.

Для того щоб один і той же обробник події був підключений до подій різних компонент необхідно:

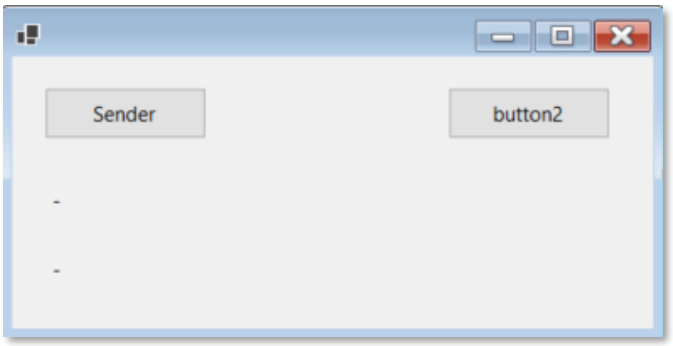
– Створити подієву функцію для якогось компонента за допомогою ***вікна властивостей,*** як було розказано раніше.
– Вибрати компонент, для обробника події якого необхідно використовувати створену подієву функцію.
– Перейти на вкладку «**Events**» у ***вікні властивостей***.
– Вибрати необхідну Вам подію і зі списку вибрати вже створену подієву функцію.

В результаті одна подієва функція відноситься до подій різних компонентів. Якщо потрібно розпізнати тільки тип об'єкта, то можна використовувати функцію **GetType()**, наприклад:

```
private void button1_Click(object sender, EventArgs e)
  {  label1.Text = Convert.ToString(sender.GetType());
     label2.Text = Convert.ToString(sender.GetType()).Substring(21);  }
```



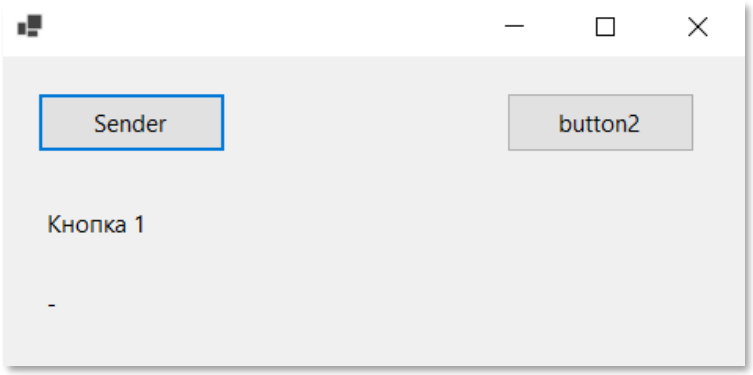
Якщо потрібно розпізнати джерело події на ім'я, то використовується метод **Equals()**. Наприклад, розглянемо додаток виду:



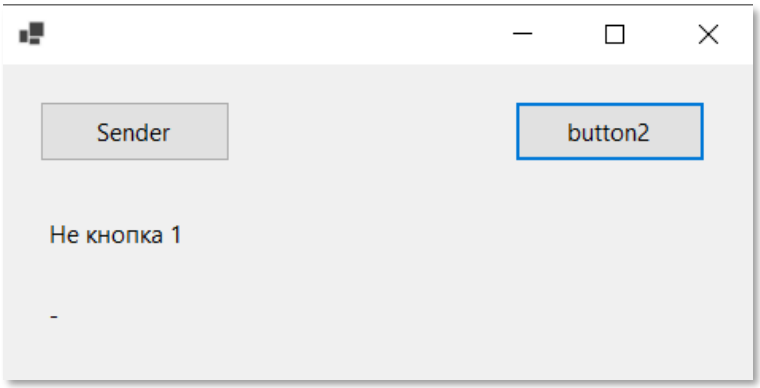
Обидві кнопки мають одну подієву функцію для Click**:**

```
private void button1_Click(object sender, EventArgs e)
    {
      if (sender.Equals(button1))
         label1.Text = "Кнопка 1";
      else
         label1.Text = "Не кнопка 1";
    }
```

Після клацання на першій кнопці:



Після клацання на другий кнопці:



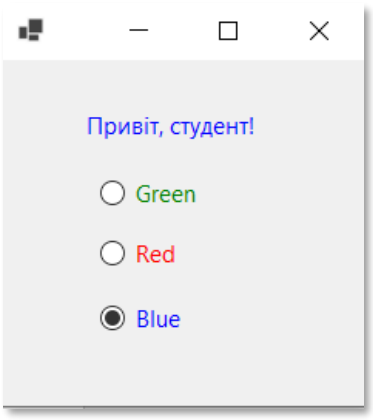
Правда, якщо треба перевірити тільки, чи є **sender** певним компонентом, то можна зробити простіше:

if (sender == button1) …

Оскільки тип **Object** не має ніяких корисних для користувача властивостей і методів, то об'єкт **sender** слід розглядати як об'єкт одного з похідних від **Object** типів. Якщо треба застосувати до джерела події деякий метод, оголошений починаючи з деякого класу, треба використовувати операцію явного приведення типу на об'єкт цього класу.

Розглянемо наступний приклад. Розташуємо на формі текстове вікно **label1** і три радиокнопки **radioButton1, radioButton2, radioButton3**. Написи на радіокнопках **Text**: Green, Red, Blue зроблені відповідним кольором. Текст **Text** в текстовому полі **textBox1**: **«Привіт, студент!»** повинен міняти колір шрифту при натисканні на радіокнопку відповідним чином. З цією метою вводимо програмний код в подієву функцію клацання на **radioButton1**:

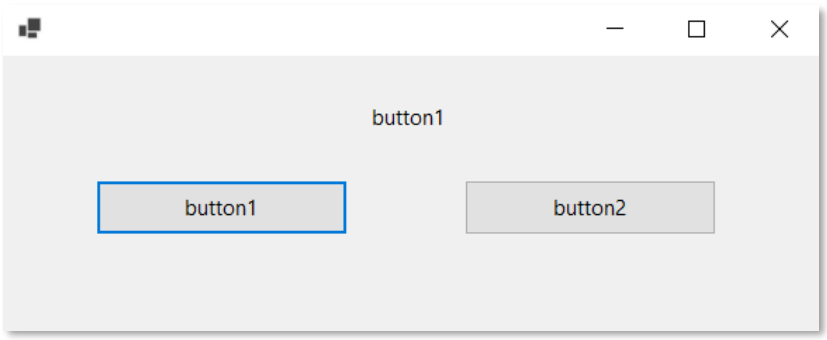
```
private void radioButton1_Click(object sender, EventArgs e)
    {
      RadioButton my = sender as RadioButton;
      label1.ForeColor = my.ForeColor;
    }
```

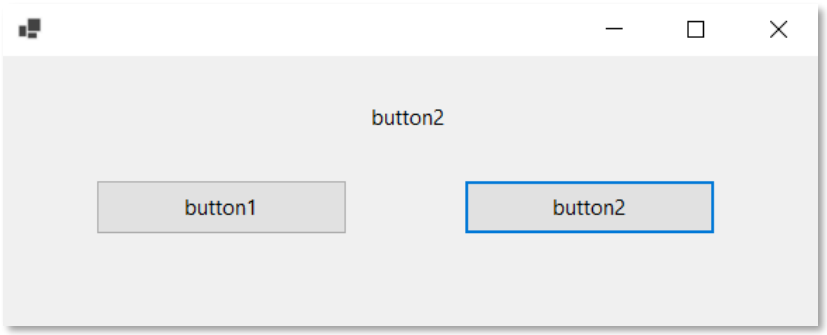


Оскільки при натисканні на двох інших радіокнопках має відбуватися те ж саме, ми підключимо їх через ***вікно властивостей***.

Подієву функцію, в разі необхідності, можна викликати за допомогою команди виду:

```
private void button1_Click(object sender, EventArgs e)
    {
      label1.Text = (sender as Button).Name;
    }

    private void button2_Click(object sender, EventArgs e)
    {
      button1_Click(button2, e);
    }
```





Слід не забувати, що в класі форми можуть міститися не тільки подієві функції, а й функції, які потрібно використовувати в подієвих для виконання якіхось дій, які протягом роботи програми повторюються при різних подіях. Модифікатор оголошення функції залежить від області доступності цієї функції. Якщо необхідно викликати функцію тільки всередині класу даної форми, то використовується модифікатор private.

Клас **Form1**: `private void myFunc(ref int r) { r++; }`
Виклик функції в функціях цього ж класу: `myFunc(ref ty);`

Якщо функція може викликатися з класу іншої форми, то використовується – public.

Клас **Form2**: `public void myFunc(ref int r) { r++; }`
Виклик функції в класі форми **Form1:**

```
Form2 f = new Form2();
f.myFunc(ref k);
```

У разі використання функції без об'єкта класу форми, тобто безпосередньо через ім'я класу, то використовується додатково static.

Клас **Form2:** `static public void myFunc(ref int r) { r++; }`
Виклик функції в класі форми **Form1:** `Form2.myFunc(ref k);`

4. ПОДІЇ КЛАВІАТУРИ

4.1 Події клавіатури KeyDown, KeyPress, KeyUp

Windows Forms містить дві події, які виникають при натисканні користувачем клавіші клавіатури, і одну подію, яка виникає при відпуску користувачем клавіші.

Подія **KeyDown** виникає один раз. Подія **KeyPress** може зустрічатися кілька разів, якщо користувач утримує натиснуту клавішу. Подія **KeyUp** виникає один раз при відпуску клавіші.

Коли користувач натискає кнопку, **Windows Forms** визначає, яку подію викликати в залежності від того, на що вказує повідомлення клавіатури – на клавішу з символом або на фізичну клавішу.

Подія **KeyDown** виникає при натисканні фізичної клавіші користувачем. Обробник події **KeyDown** отримує параметр **e** типу **KeyEventArgs**, який має властивості:

1. KeyCode – вказує на фізичну клавішу клавіатури і має тип **Keys**. **Keys** – це перерахування задає коди і модифікатори клавіш.

```
if (e.KeyCode == Keys.F1)
        label1.Text =" Допомога!";
```

Якщо натиснута клавіша F1, то видається повідомлення.

2. Modifiers – вказує, яка комбінація клавіш CTRL, SHIFT і ALT була натиснута. Має тип **Keys.**

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
    {   if (e.KeyCode == Keys.A && e.Modifiers == Keys.Shift)
        label1.Text ="Допомога!";     }
```

*Якщо натиснута кнопка **A**, утримуючи клавішу **Shift**, то видається повідомлення.*

Є властивості Shift, Alt, Control всі булевого типу, які показують, чи була натиснута відповідна клавіша. Попередню подієву функцію можна записати у вигляді:

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
    {   if (e.KeyCode == Keys.A && e.Shift)
        label1.Text ="Помощь!";       }
```

3. KeyData - об'єднує код клавіші і модифікатора CTRL, SHIFT або ALT.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
    {   label1.Text =
        label1.Text + "\nNew" + "\nKeyData=" + Convert.ToString(e.KeyData) +
                      "\nKeyCode=" + Convert.ToString(e.KeyCode) +
                      "\nModifiers=" + Convert.ToString(e.Modifiers);      }
```

4. Властивість **SuppressKeyPress** – булевого типу, яка може використовуватися для подавлення подій KeyPress і KeyUp для даного натискання клавіші.

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
    {     e.SuppressKeyPress = true;      }
```

Ця подієва функція не викликається.

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {      this.Text = "KeyPress";          }
```

Обробник події **KeyUp** має такі ж самі вхідні параметри.

Подія KeyPress виникає при натисканні клавіші або клавіш, які призводять до введення знака. Обробник події приймає параметр **e** типу **KeyPressEventArgs**. Параметр має властивості:

KeyChar – символ відповідний кнопці в момент виклику події.

Handled – булева властивість, якщо встановити в **true**, то обробка символу скасовується.

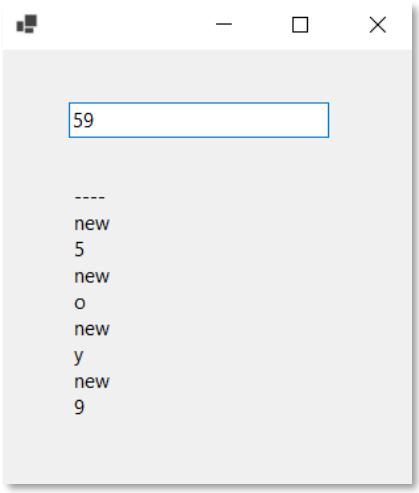
Приклад. Розглянемо приклад програми, що містить текстове вікно тільки для введення цілих чисел.

На етапі візуального програмування розташуємо на формі текстове вікно **textBox1** і мітку **label1**. Текст в обох компонентах очистимо.

Для текстового вікна створимо подієву функцію **KeyPress**:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
        label1.Text = label1.Text + "\nnew\n" + Convert.ToString(e.KeyChar);
        if (e.KeyChar < '0' || e.KeyChar > '9')
        e.Handled = true;
    }
```

Якщо натискати не на клавіші з цифрами, то вони не будуть відображатися в текстовому вікні. Всі цифри відображаються. Мітка відображає інформацію про всіх натиснутих символах.



Зауваження. Недолік – неможливість виправити цифру, що вводиться, у разі помилки, оскільки натискання клавіші **Back** призводить до необроблення символу, що вводиться. Виправимо умову в подієвої функції:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
        label1.Text = label1.Text + "\nnew\n" + Convert.ToString(e.KeyChar);
        if ((e.KeyChar < '0' || e.KeyChar > '9') && e.KeyChar!='\b')
        e.Handled = true;
    }
```

4.2 Обробка введення з клавіатури на рівні форми

Щоб повідомлення клавіатури приймалися формою, перш ніж вони досягнуть елементів управління у формі, потрібно обробити події **KeyPress** або **KeyDown** форми і привласнити властивості форми **KeyPreview** значення **true**.

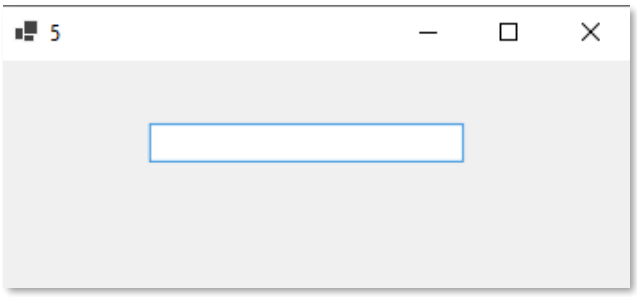
Приклад. На формі розташовано текстове вікно, для якого підключений обробник події:

```
private void textBox1_KeyPress(object sender, KeyPressEventArgs e)
    {  Text = " Це текстове вікно:)";  }
```

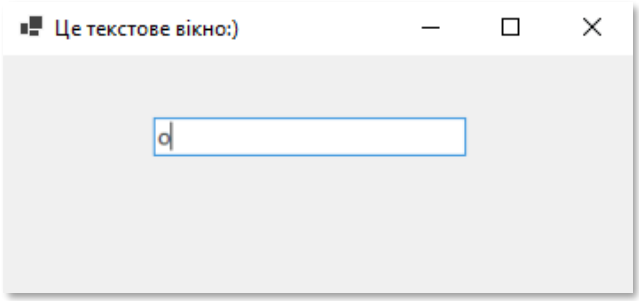
Форма має властивість **KeyPreview = true** і обробник події:

```
private void Form1_KeyPress(object sender, KeyPressEventArgs e)
{
   if (e.KeyChar == '1' || e.KeyChar == '5')
   {
      e.Handled = true;
      Text = Convert.ToString(e.KeyChar);
   }
}
```

Якщо фокус в текстовому вікні і натиснути клавіші цифр 1 або 5, то символи не відображаються в текстовому вікні і змінюється текст заголовка форми.



Якщо натиснути всі інші символи:



Зауваження. *Якщо Ви хочете в подієвих функціях клавіатури перехоплювати клавішу стрілки, TAB, RETURN або ESC (зазвичай вони ігноруються деякими елементами управління, оскільки вони не вважаються натисканнями клавіш введення), то необхідно для компонента, який перебуває у фокусі в момент натискання потрібної клавіші, обробити подію* ***PreviewKeyDown*** *і встановити в ній властивість* ***IsInputKey*** *вхідного параметра* ***e*** *в* ***true****.*

4.3 Обробка подій миші

Windows Forms включає різноманітні події миші і забезпечує додаткову підтримку настроюваних курсорів миші, захоплення миші і поведінки при перетягуванні.

Ми розглянемо п'ять подій, пов'язаних з мишею: **MouseUp, MouseDown, Click, MouseClick і MouseMove**.

Якщо потрібно обробляти події клацання миші в певному порядку, необхідно знати порядок, в якому викликаються події клацання в елементах управління **Windows Forms**. Коли кнопка миші (будь-яка) натискається і відпускається, майже всі елементи управління **Windows Forms** викликають події клацання в одному і тому ж порядку. Нижче наведено порядок подій, що викликаються одинарним клацанням миші.

Подія MouseDown.
Подія Click.
Подія MouseClick.
Подія MouseUp.

Подія **MouseDown (MouseUp)** відбувається при натисканні (відпуску) кнопки миші, якщо курсор миші знаходиться на елементі управління. Другим вхідним параметром в функцію є об'єкт **e** типу **MouseEventArgs**. Цей об'єкт вказує, яка кнопка миші натиснута, скільки разів була натиснута і відпущена кнопка миші, координати миші і відстань, на яку перемістилося коліщатко миші. Має такі властивості:

1. Button – отримує значення, що показує, яка кнопка миші була натиснута. Це властивість типу MouseButtons.

```
if (e.Button == MouseButtons.Left)
   label1.Text = "ліва";
if (e.Button == MouseButtons.Right)
   label1.Text = "права";
if (e.Button == MouseButtons.Middle)
   label1.Text = "середня";
```

2. Location – отримує розташування курсору миші на момент створення події.
X – отримує координату **x** покажчика миші на момент створення події.

Y – отримує координату **y** покажчика миші на момент створення події.

```
label1.Text = "Location.X=" + Convert.ToString(e.Location.X);
label1.Text= X="+ Convert.ToString(e.X);
```

В результаті з'являється одне і теж число, причому координати відраховуються в системі координат компонента. Це так звані відносні або локальні координати.

Подія **Click** відбувається при натисканні елемента управління. Подія **Click** передає об'єкт **e** типу **EventArgs** його обробнику подій, вказуючи що клацання щойно було виконано. Якщо необхідні більш точні відомості про мишу, слід використовувати подію MouseClick. Однак подія MouseClick не відбувається, якщо клацання елементу управління було виконано не за допомогою миші (наприклад, натиснута клавіша ENTER). У подієву функцію **MouseClick** передається вхідний параметр типу **MouseEventArgs**.

Подія **MouseMove** відбувається при переміщенні курсору миші по елементу управління і також має вхідний параметр типу **MouseEventArgs**.

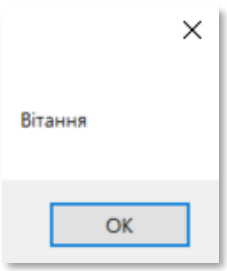
5. ВІКНО ПОВІДОМЛЕНЬ

5.1 Метод MessageBox.Show() класу MessageBox

Показує вікно повідомлення, що містить текст, кнопки і символи, за допомогою яких інформується і інструктується користувач. Неможливо створити новий екземпляр класу **MessageBox**. Щоб вивести на екран вікно повідомлення, слід викликати static різновид методу **MessageBox.Show**. Тема, повідомлення, кнопки і значки, які відображаються у вікні повідомлення, визначаються параметрами, переданими в цей метод.

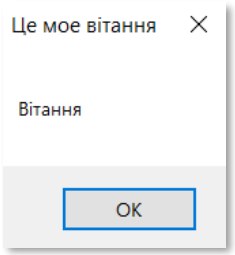
public static DialogResult Show(String text) – показує вікно повідомлення з заданим текстом.

```
MessageBox.Show("Вітання ");
```



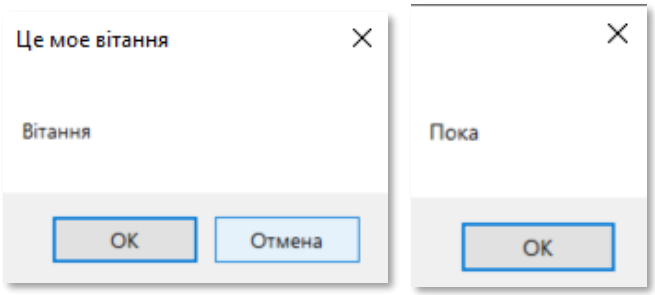
public static DialogResult Show(String text, String caption) – показує вікно повідомлення з заданим текстом і заголовком.

```
MessageBox.Show("Вітання ", "Це мое вітання ");
```



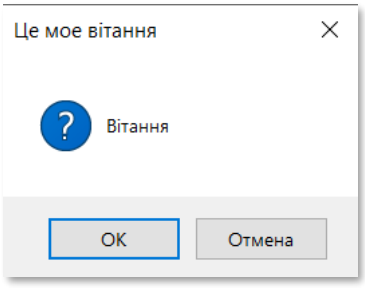
public static DialogResult Show(String text, String caption, MessageBoxButtons buttons) – показує вікно повідомлення з заданим текстом, заголовком і кнопками. Наявність двох кнопок і більш на увазі аналіз вибору кнопки користувача.

```
if (MessageBox.Show("Вітання ", "Це мое вітання ",
    MessageBoxButtons.OKCancel) == DialogResult.Cancel)
        MessageBox.Show("Пока");
```



public static DialogResult Show(String text, String caption, MessageBoxButtons buttons, MessageBoxIcon icon) – показує вікно повідомлення з заданим текстом, заголовком, кнопками і значком.

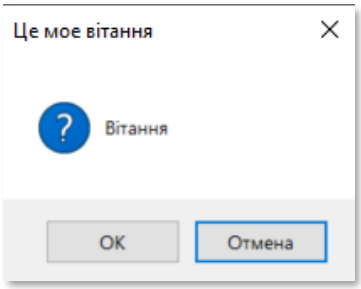
```
MessageBox.Show("Вітання ", "Це мое вітання ",
MessageBoxButtons.OKCancel,MessageBoxIcon.Question);
```



public static DialogResult Show(String text, String caption, MessageBoxButtons

buttons, MessageBoxIcon icon, MessageBoxDefaultButton defaultButton) – показує вікно повідомлення з заданим текстом, заголовком, кнопками, значком і кнопкою за замовчуванням. Останній параметр задає кнопку обрану за замовчуванням.

```
MessageBox.Show("Вітання ", "Це мое вітання ",
MessageBoxButtons.OKCancel, MessageBoxIcon.Question,
MessageBoxDefaultButton.Button2);
```



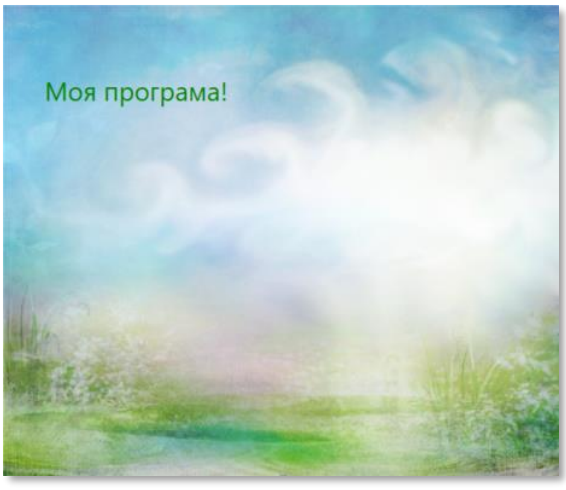
Фокус буде перебувати на кнопці **Cancel**. У діалоговому вікні може бути до трьох кнопок.

5.2 Приклад програми з модальними формами заставки і запиту пароля.

Припустимо, що ми хочемо створити застосунок, в якому, по-перше, спочатку з'являється заставка, що носить інформаційний характер, і, по-друге, форма запиту пароля.

1. Дамо назву головній формі застосунку **FMain**.

2. Додамо в застосунок нову форму, яку назвемо **FLog** – це буде форма-заставка. Встановимо **FormBorderStyle = None, StartPosition = CenterScreen**. Розміри зробимо невеликі, а на форму помістимо малюнок в властивість **BackgroundImage**, а також напис ***«Моя програма»*** на мітці. Для мітки встановимо властивість **BackColor = Transparent**, при цьому мітка стане прозорою. Можна змінити властивість **ForeColor**, наприклад, в цьому прикладі встановлений зелений колір.



3. Напишемо обробники подій, які при будь-якій дії користувача закривали б форму. По-перше, напишемо подієву функцію **KeyDown** для форми **FLog**:

```
private void FLog_KeyDown(object sender, KeyEventArgs e)
    {   this.Close();   }
```

Аналогічний обробник напишемо для події **OnMouseDown.**

```
private void FLog_MouseDown(object sender, MouseEventArgs e)
    {   this.Close();   }
```

Цю подієву функцію підключимо за допомогою ***вікна властивостей*** до події **MouseDown** мітки.

Зауваження. *Не можна використовувати одну і ту ж подієву функцію для всього третього пункту, так як це функції з різною сигнатурою.*

Зробимо так, щоб і при відсутності якихось дій з боку користувача форма закривалася сама, наприклад, через 5 секунд. Для цього додамо на форму компонент **Timer**. Задамо **Interval = 5000**, **Enabled = true**. У подієву функцію **Tick** запишемо:

```
private void timer1_Tick(object sender, EventArgs e)
    {   this.Close();   }
```

Тепер при будь-якій дії і навіть бездіяльності форма-заставка буде закриватися.

У файлі форми **FMain** напишемо обробник події форми **Load:**

```
private void FMain_Load(object sender, EventArgs e)
    {
      FLog myF2 = new FLog();
      myF2.ShowDialog();
    }
```

Оператор відкриває форму **FLog** як модальну, передає їй управління і подальше виконання програми зупиняється до тих пір, поки модальна форма не буде закрита. Після закриття модальної форми виконання програми продовжиться і головна форма стане видимою.

Тепер додамо в додаток форму запиту пароля **FPar**. Встановимо **FormBorderStyle = FixedDialog, StartPosition = CenterScreen, Text = *“Введіть пароль і натисніть Enter”*.**

Помістимо в центрі форми мітку з написом ***“Ведіть пароль доступу”*** і вікно редагування **TextBox**, в яке користувач буде вводити пароль. Встановимо **PasswordChar = "*"**. Напишемо обробник події **KeyDown**:

```
private void textBox1_KeyDown(object sender, KeyEventArgs e)
    {
      if (e.KeyCode == Keys.Enter)
        this.Close();
    }
```

Якщо користувач натиснув Enter, то вважається, що користувач завершив введення пароля і форма закривається.

Напишемо обробник події **FormClosed** форми **FPar**:

```
private void FPar_FormClosed(object sender, FormClosedEventArgs e)
    {
      if (textBox1.Text == "1")
        this.DialogResult = DialogResult.OK;
    }
```

Якщо пароль введений правильно, то властивості **DialogResult** присвоюється деяке умовне значення - **ОК** (можна було вибрати інше значення, окрім **None** і **Cancel**).

Якщо пароль неправильний, то залишається значення **DialogResult = Cancel**, яке автоматично присвоюється при будь-якій спробі закрити форму.

У файлі форми **FMain** в обробнику події **Load** після раніше введеного оператора **myF2.ShowDialog ()** додамо:

```
private void FMain_Load(object sender, EventArgs e)
    {
      FLog myF2 = new FLog();
      myF2.ShowDialog();
      FPar myF3 = new FPar();
      if (myF3.ShowDialog() != DialogResult.OK)
      {  MessageBox.Show("Ваш пароль невірний ");
        this.Close();  }
      else  MessageBox.Show("Ваш пароль вірний ");
     }
```

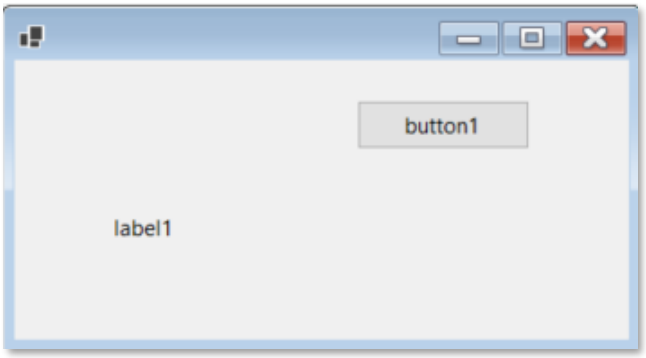
Цей оператор аналізує значення властивості **DialogResult** форми запиту пароля.

6. СТВОРЕННЯ І ВИКОРИСТАННЯ ДИНАМІЧНИХ КОМПОНЕНТ

6.1 Властивість Controls. Методи Add() та AddRange().

У об'єктно-орієнтованому програмуванні компоненти, які безпосередньо розташовуються на формі, на етапі візуального програмування, називаються статичними компонентами. Часто при розробці застосунків є такі компоненти, типи яких відомі лише під час виконання програми і залежать від дій користувача. У таких випадках має сенс використовувати динамічні компоненти. Динамічні компоненти – це компоненти, які створюються програмним шляхом.

При створенні динамічного компонента слід пам'ятати, що керуючий елемент є об'єктом певного класу. Тип об'єкта визначається умовами завдання, яку Ви вирішуєте. При створенні динамічного компонента конкретного типу використовується змінна цього ж типу. Наприклад, нехай на формі розташовуються два статичних елемента управління **button1** і **label1**

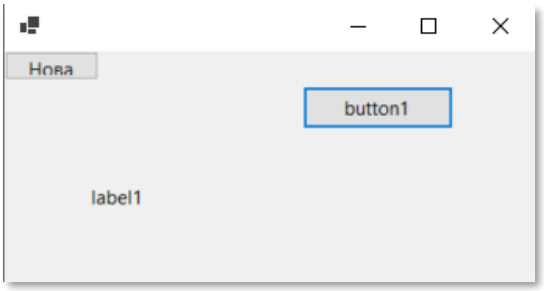


А за допомогою команди виду:

```
private void Form1_Load(object sender, EventArgs e)
    {
      Button myBut = new Button();
      this.Controls.Add(myBut);
      myBut.Text = "Нова";
    }
```

буде створена кнопка з ім'ям **myBut** типу **Button** і з текстом напису **“Нова”**. У функції є три оператора: перший створює об'єкт **myBut**, другий - додає створений об'єкт в колекцію компонент даної форми, а третій оператор, який є не обов'язковим, генерує напис на цій кнопці. Зверніть увагу, що всі інші властивості компонента не задаються, вони будуть мати значення прийняті за

замовчанням в конструкторі класу **Button**. Наприклад, координати розташування кнопки за замовчуванням **(0,0)**.



Необхідно детально зупиниться на властивості **Controls**, яке було використано нами при створенні динамічного компонента. Це властивість класу **Control**. Отримує колекцію компонентів, що містяться в елементі управління. Має тип **ControlCollection**. При додаванні елементів управління до об'єкта **Form**, кожен з елементів управління є елементом об'єкта типу **ControlCollection**, призначеного властивості Controls форми, так як форма є похідною від класу **Control**. Можна керувати елементами управління в об'єкті властивості **Controls**, шляхом використання методів, доступних в класі **ControlCollection**.

При створенні динамічного компонента ми використовували метод **Add().** Такий же метод використовує сама **VS** при створенні статичних компонент на етапі візуального програмування. Ми вже говорили, що **VS** самостійно генерує код в файл форми **Form1.Designer.cs**. Наприклад, якщо на формі на етапі візуального програмування були розташовані кнопка і мітка, то у файлі **Form1.Designer.cs** будуть знаходитися рядки:

this.Controls.Add(this.label1);

this.Controls.Add(this.button1);

Метод має прототип: public virtual void Add(Control value)

і додає об'єкти **Control** в кінець колекції елементів управління. Об'єкт **Control** може бути включений тільки до однієї колекції **ControlCollection** одночасно. Якщо він вже є дочірнім елементом для іншого елемента управління, його треба видалити з цієї колекції перед додаванням до іншого елемента управління.

Якщо на формі необхідно розташувати кілька динамічних компонент, то можна скористатися методом AddRange().

Метод має прототип: public virtual void AddRange(Control[] controls)

Наприклад

```
private void Form1_Load(object sender, EventArgs e)
    {
       Button myBut = new Button();
       myBut.Text = "Нова";
       myBut.Location = new Point(0, 10);
       Label myLab = new Label();
       myLab.Text = "Нова";
       myLab.Location = new Point(0, 50);
       this.Controls.AddRange(new Control[] { myBut, myLab });
    }
```

У розглянутих прикладах оголошення об'єктів розташовано всередині подієвих функцій. Це означає, що імена цих об'єктів мають локальний характер і область видимості цих об'єктів – тільки сама функція. При виході з функції сам динамічний керуючий елемент на формі залишається, але доступ до нього за допомогою оголошеного імені не буде. Розумніше використовувати імена об'єктів, як члени-класу, тоді можна мати можливість доступу до даного елементу управління через його ім'я з усіх функцій-членів класу.

```
public partial class Form1 : Form
   {
      Button myBut;
      Label myLab;
      public Form1()
      {  InitializeComponent();  }

      private void Form1_Load(object sender, EventArgs e)
     {     myBut = new Button();
           myBut.Text = "Нова";
           myBut.Location = new Point(0, 10);
           myLab = new Label();
           myLab.Text = "Нова";
           myLab.Location = new Point(0, 50);
           this.Controls.AddRange(new Control[] { myBut, myLab });
      }
    }
```

Існують методи, за допомогою яких можна видаляти компоненти з колекції керуючого елемента.

Remove() – видаляє зазначений елемент з колекції. Має прототип: public virtual void Remove(Control value)

```
private void button1_Click(object sender, EventArgs e)
{    this.Controls.Remove(myBut);  }
```

Після виконання цієї команди кнопка **myBut** зникає з форми. Коли **Control** видаляється з колекції елементів управління, всі наступні елементи управління пересуваються вгору на одну позицію в колекції.

RemoveAt() – видаляє елемент управління з колекції за його індексом. Має прототип: public void RemoveAt(int index)

```
private void button1_Click(object sender, EventArgs e)
{    this.Controls.RemoveAt(3);    }
```

Після виконання цієї команди мітка MyLab зникає з форми, так як вона мала індекс 3.

Clear() – видаляє всі елементи управління з колекції. Має прототип: public virtual void Clear()

```
private void button1_Click(object sender, EventArgs e)
{    this.Controls.Clear();    }
```

Після виконання цієї команди всі чотири компонента зникають з форми.

Зауваження. *При виклику методів Remove(), RemoveAt() і Clear() керуючі об'єкти не видаляються з пам'яті. У разі необхідності їх можна знову додати на форму:*

```
private void button1_Click(object sender, EventArgs e)
    {
      this.Controls.RemoveAt(3);
      myLab.Text = "Це знову я:)";
      this.Controls.Add(myLab);
    }
```

Якщо необхідно видалити компонент і звільнити ресурси, які вмкористовуються об'єктом, треба викликати метод **Dispose().**

```
private void button1_Click(object sender, EventArgs e)
    {
      this.Controls[3].Dispose();
      myLab.Text = " Це знову я:)";
      this.Controls.Add(myLab);
    }
```

Можна організувати очищення ресурсів за допомогою методу **Dispose()** для всіх компонентів форми:

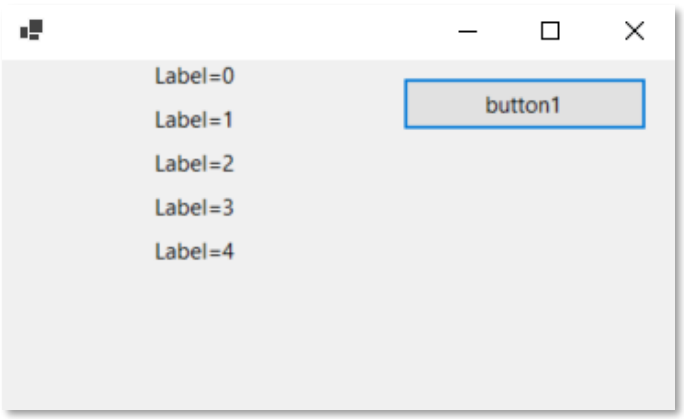
```
private void button1_Click(object sender, EventArgs e)
    {
      for (int i = this.Controls.Count - 1; i >= 0; i--)
      {    this.Controls[i].Dispose();    }
    }
```

Зауваження. *Якщо метод Dispose() застосувати до форми, то так само звільняються всі ресурси пам'яті виділені під форму.*

6.2 Масиви динамічних компонент

Динамічно можна створювати не тільки окремі компоненти, але і масив керуючих елементів. Наприклад, оголошуємо дані-члени класу форми:

```
Label[] myMasLabel;
int n = 5;
private void button1_Click(object sender, EventArgs e)
    {
      myMasLabel = new Label[n];
      for (int i = 0; i < n; i++)
      {
        myMasLabel[i] = new Label();
        this.Controls.Add(myMasLabel[i]);
        myMasLabel[i].Location = new Point(100, 30 * i);
        myMasLabel[i].Text = "Label=" + Convert.ToString(i);
      }
    }
```



Можна використовувати інший метод додавання в колекцію

```
private void button1_Click(object sender, EventArgs e)
    {
       myMasLabel = new Label[n];
       for (int i = 0; i < n; i++)
       {
          myMasLabel[i] = new Label();
          myMasLabel[i].Location = new Point(100, 30 * i);
          myMasLabel[i].Text = "Label=" + Convert.ToString(i);
       }
       this.Controls.AddRange(myMasLabel);
    }
```

Зауваження. *При додаванні декількох елементів управління до батьківського елементу рекомендується викликати метод* ***SuspendLayout()*** *перед початком ініціалізації доданих елементів управління. Після додавання елементів управління до батьківського елементу необхідно викликати метод* ***ResumeLayout()****. Це допомагає підвищити продуктивність застосунків з великою кількістю елементів управління. Метод* ***SuspendLayout()*** *тимчасово призупиняє події розміщення, які відбуваються під час початкової ініціалізації елемента управління. Після цього, в кінці методу, слід викликати* ***ResumeLayout()****, щоб повернути все в норму.*

```
private void button1_Click(object sender, EventArgs e)
    {
      myMasLabel = new Label[n];
      SuspendLayout();
      for (int i = 0; i < n; i++)
      {
        myMasLabel[i] = new Label();
        myMasLabel[i].Location = new Point(100, 30 * i);
        myMasLabel[i].Text = "Label=" + Convert.ToString(i);
      }
      this.Controls.AddRange(myMasLabel);
      ResumeLayout();
    }
```

6.3 Підключення подієвих функцій до динамічних компонентів.

Динамічні компоненти, як і статичні повинні реагувати на різні події, а значить, для них необхідно написати обробники подій і підключити їх програмним шляхом. Для цього необхідно:

1. Створити функцію з власним ім'ям для обробки події, яка має відповідну сигнатуру.

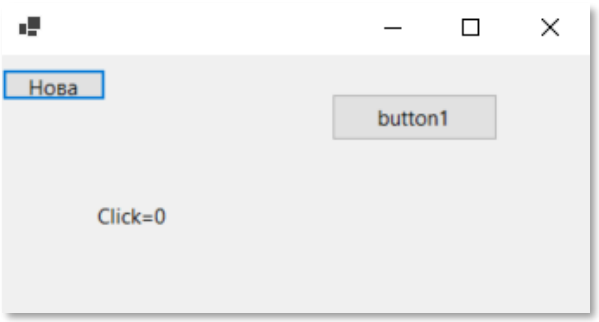
```
private void myBut_Click(object sender, EventArgs e)
        {
          label1.Text = "Click=" + Convert.ToString(i) + "\n";
          i++;
        }
```

2. Підключити створену функцію за допомогою команди виду:

```
private void Form1_Load(object sender, EventArgs e)
        {
          myBut = new Button();
          myBut.Text = "Нова";
          myBut.Location = new Point(0, 10);
          myBut.Click += new EventHandler(myBut_Click);
          this.Controls.Add(myBut);
        }
```

3. У подієвої функції використовується змінна, яка оголошена в класі форми: int i;

Результат:



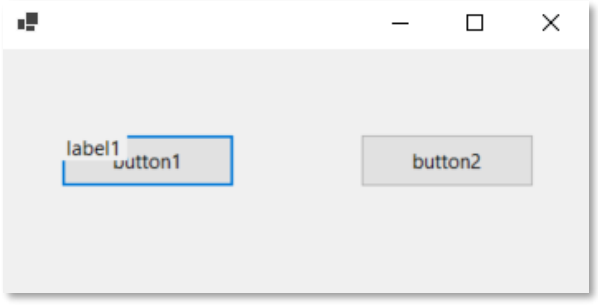
***Зауваження**. Властивість **Controls** належить не тільки контейнерам, але й усім спадкоємцям класу **Control**.*

Наприклад, нехай на формі розташована мітка label1 і кнопки button1 і button2.

Нехай при натисканні на кнопці button1 виконується подійна функція Click виду:

```
private void button1_Click(object sender, EventArgs e)
    {
      this.Controls.Remove(label1);
      button1.Controls.Add(label1);
      label1.Location = new Point(0, 0);
    }
```

В результаті маємо: (**label1** з'явиться як дочірній елемент кнопки **button1**)



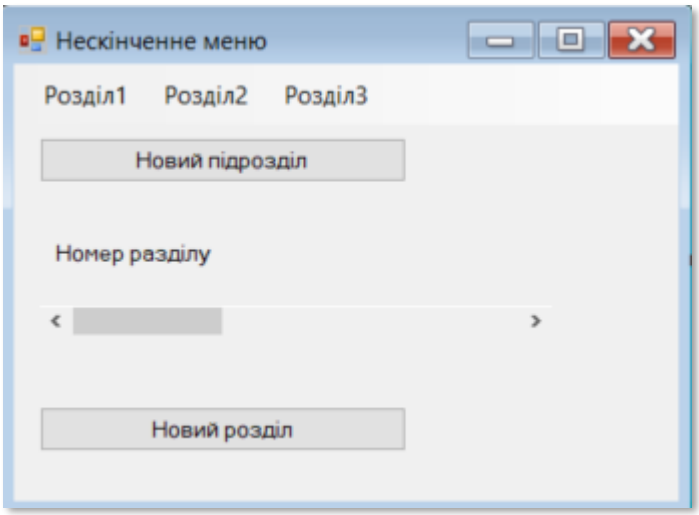
Зауваження. *Обов'язково задавати координати, інакше компонента не видно, тому що його координати будуть розраховуватись у новій клієнтській області.*

6.4 Створення нескінченного меню.

Робота з об'єктами класів дозволяє створювати у застосунку нескінченне меню. Розглянемо приклад програми. На формі **Form1** розташуємо компонент меню **MenuStrip**, дві кнопки **Button**, бігунок **hScrollBar** та мітку **Label**.
Меню MainStrip має мати вигляд:

Розділ1 *Розділ2* *Розділ3*
Розділ11 *Розділ21*
Розділ12 *Розділ22*

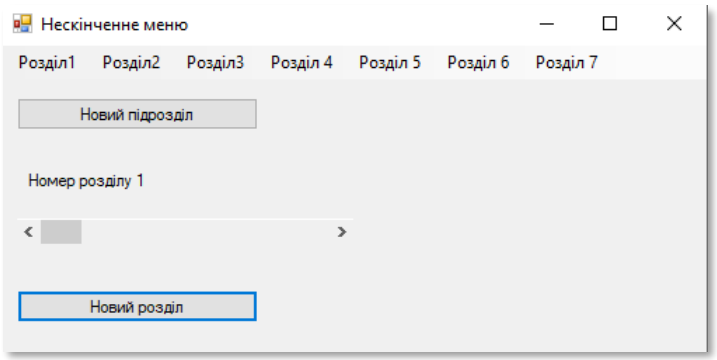
Перша кнопка **button1** має напис ***«Новий підрозділ»*** і при натисканні на ній, додається новий підрозділ у розділ з номером, вказаним за допомогою бігунка. Бігунок має мінімальне значення **Min=1** та **Max=3**, що відповідає заданій кількості розділів на етапі візуального програмування. Мітка **label1** розташована над прокручуванням і відображає номер вибраного розділу. Друга кнопка **button2** має напис ***«Новий розділ»*** і при натисканні на ній додає новий розділ у головне меню.



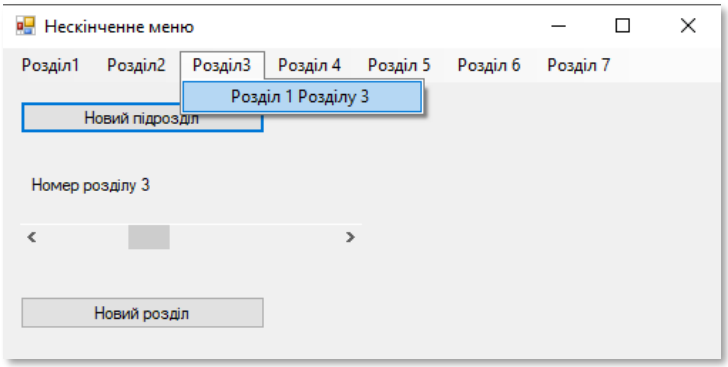
Код має вид:

```
12      public partial class Form1 : Form
13      {
14          int i = 1;
15          public Form1()
16          {
17              InitializeComponent();
18          }
19          private void Form1_Load(object sender, EventArgs e)
20          {
21              label1.Text = "Номер розділу " + Convert.ToString(hScrollBar1.Value);
22          }
23
24          //Новий підрозділ
25          private void button1_Click(object sender, EventArgs e)
26          {
27              ToolStripMenuItem my = new ToolStripMenuItem();
28              my.Name = "Підрозділ " + Convert.ToString(i);
29              my.Text = "Розділ " + Convert.ToString(i++) + " Розділу " + Convert.ToString(hScrollBar1.Value);
30              (menuStrip1.Items[hScrollBar1.Value - 1] as ToolStripMenuItem).DropDownItems.Add(my);
31          }
32
33          //Новий розділ
34          private void button2_Click(object sender, EventArgs e)
35          {
36              ToolStripMenuItem my = new ToolStripMenuItem();
37              my.Name = "Розділ " + Convert.ToString(hScrollBar1.Maximum + 1);
38              my.Text = "Розділ " + Convert.ToString(hScrollBar1.Maximum + 1);
39              menuStrip1.Items.Add(my);
40              hScrollBar1.Maximum++;
41          }
42          private void hScrollBar1_Scroll(object sender, ScrollEventArgs e)
43          {
44              label1.Text = "Номер розділу " + Convert.ToString(hScrollBar1.Value);
45          }
46      }
```

Результат. При неодноразовому натисканні кнопки **button2** головне меню набуває вигляду:



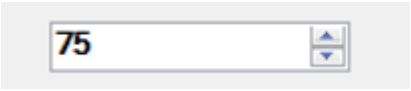
При виборі розділу меню за допомогою прокручування та натискання кнопки **button1**:



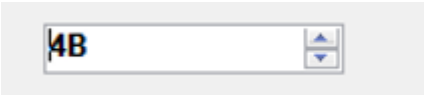
7. КОМПОНЕНТИ NumericUpDown і ToolTip.

7.1 Компонент NumericUpDown.

Елемент управління **NumericUpDown** є поєднанням текстового поля і пари кнопок зі стрілками для вибору значення користувачем. Він виводить та задає окреме числове значення у списку варіантів фіксованих числових значень. Користувач може збільшувати та зменшувати число, натискаючи кнопки зі стрілками вгору та вниз або клавіші зі стрілками ВВЕРХ та ВНИЗ, а також вводячи число в область текстового поля елемента управління. При натисканні кнопки зі стрілкою ВВЕРХ число збільшується до максимуму; при натисканні клавіші зі стрілкою ВНИЗ число зменшується до мінімуму.

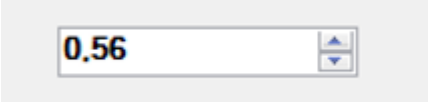


Компонент **NumericUpDown** дозволяє представляти значення у текстовому полі елемента управління в різних форматах, включаючи шістнадцятковий. Якщо властивість **Hexadecimal** має значення true, то цей елемент управління може відображати значення у шістнадцятковому форматі замість десяткового. За замовчуванням, ця властивість має значення false. Наприклад, число **75** може бути представлене у шістнадцятковому форматі як **0x4B**:



Властивість **ThousandsSeparator** визначає, чи буде вставлено роздільник між кожними трьома десятковими знаками; за замовчанням використовується значення **false**.

Властивість **DecimalPlaces** визначає кількість знаків після десяткової коми, за замовчуванням використовується нульове значення.



Зауваження. *Якщо вікно властивостей не дозволяє задавати дійсні значення, їх можна встановити програмним шляхом.*

Ключові властивості елемента управління: **Value, Maximum** (значення за замовчуванням **– 100**), **Minimum** (значення за умовчанням **– 0**) та **Increment** (значення за умовчанням **– 1**).

Властивість **Value** встановлює поточне значення, вибране елементом керування.

Властивість **Increment** визначає крок зміни значення при натисканні кнопки зі стрілкою вгору або вниз. При знятті фокусу з елемента управління будь-які введені дані перевіряються на відповідність до мінімального та максимального числових значень. Основними методами цього елемента управління є методи **UpButton** та **DownButton**. Числове значення визначається з допомогою його властивості **Value**. Навіть у тому випадку, якщо значення відображається у шістнадцятковому форматі, у будь-яких перевірках для властивості **Value** перевірятиметься його десяткове значення. Можна створити перевірку умов для значення елемента управління так само, як для будь-якої іншої властивості. Після завдання властивості Value його можна налаштувати безпосередньо за допомогою коду для виконання над ним операцій або викликати методи **UpButton** та **DownButton**.
numericUpDown1.UpButton(); - збільшення інкременту.

При виклику методу **UpButton** або **DownButton**, як у коді, так і при натисканні кнопки ВВЕРХ або кнопки ВНИЗ, нове значення перевіряється, і елемент управління оновлюється, приймаючи нове значення у відповідному форматі.

7.2 Компонент ToolTip

Клас **ToolTip** дозволяє згенерувати підказки користувачеві, коли він наводить курсор миші на елемент управління. Це невізуальний компонент. Клас **ToolTip** зазвичай використовується для попередження користувачів про призначення елемента управління. Наприклад, для елемента керування **TextBox**, призначеного для введення імені, можна вказати текст підказки, яка описує формат імені, що вводиться в елемент управління. Крім надання підказок, клас **ToolTip** можна використовувати для надання відомостей про стан під час виконання.

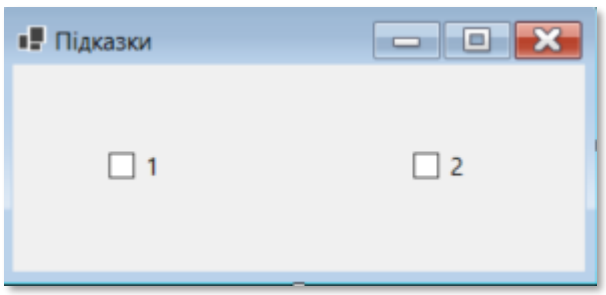
Для створення спливаючих підказок для кількох елементів керування

однієї форми зазвичай використовується один компонент **ToolTip**. Після створення підказки **ToolTip** можна використовувати окремий виклик методу **SetToolTip(),** щоб зв'язати відображуваний текст спливаючої підказки для окремого елемента керування. Потім, коли користувач поміщає курсор над елементом управління, з'являється підказка з відповідним текстом. Для того самого елемента керування метод **SetToolTip()** можна викликати кілька разів, щоб змінити текст, пов'язаний з цим елементом. Якщо потрібно отримати текст, пов'язаний із елементом керування, слід скористатися методом **GetToolTip().** Для видалення всіх встановлених зв'язків між текстами підказок і екземпляром класу **ToolTip** використовується метод **RemoveAll()**. Якщо потрібно вимкнути весь текст підказок, щоб вони не могли бути відображені в програмі, можна скористатися властивістю **Active**.

Властивості компонента ToolTip:

1. **AutomaticDelay** - отримує або задає автоматичну затримку підказки. Приймає цілі значення у мілісекундах.
2. **AutoPopDelay** - отримує або задає інтервал часу, протягом якого підказка відображається на екрані, коли курсор миші зупиняється в межах елемента управління з текстом цієї підказки.
3. **InitialDelay** - отримує або задає інтервал часу перед появою підказки.
4. **ReshowDelay** - отримує або задає інтервал часу, який повинен пройти перед появою вікна чергової підказки при переміщенні курсору миші з одного елемента управління на інший.
5. **ToolTipIcon** - отримує або задає значення, що визначає тип значка, який відображається разом із текстом підказки.
6. **ToolTipTitle** - отримує або задає заголовок вікна підказки.

Розглянемо приклад програми, яка на етапі візуального програмування має вигляд:

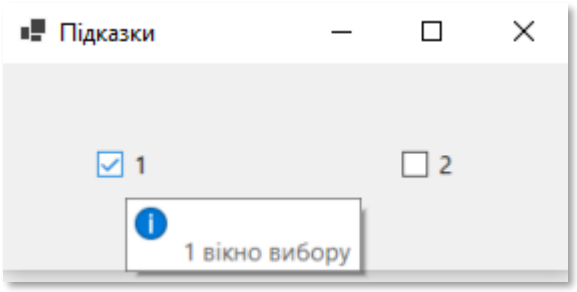


Крім того, є компонент **toolTip1**.

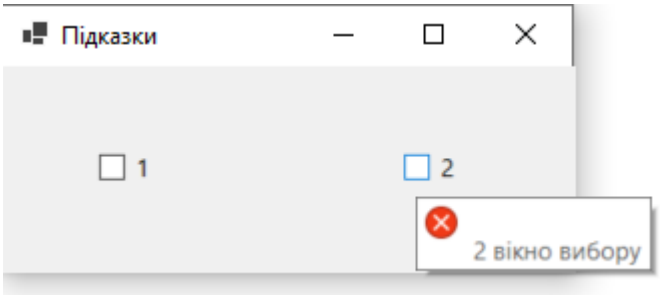
Код програми має вигляд:

```
5       public Form1()
6       {
7           InitializeComponent();
8       }
9       private void checkBox1_CheckedChanged(object sender, EventArgs e)
10      {
11
12          if (checkBox1.Checked)
13          {
14              toolTip1.RemoveAll();
15              toolTip1.SetToolTip(checkBox1, "1 вікно вибору");
16              toolTip1.ToolTipIcon = ToolTipIcon.Info;
17          }
18          else
19          {
20              toolTip1.RemoveAll();
21              toolTip1.SetToolTip(checkBox2, "2 вікно вибору");
22              toolTip1.ToolTipIcon = ToolTipIcon.Error;
23          }
24
25      }
```

Під час запуску програми жодне з вікон вибору не має підказки. Після натискання на компоненті **checkBox1** біля вікна вибору **checkBox1** з'являється підказка виду:



Друге вікно вибору підказки не має. Після повторного натискання на компоненті **checkBox1:**



При цьому перше вікно вибору підказки немає. Такого ефекту досягнуто за рахунок використання методу **RemoveAll()**.

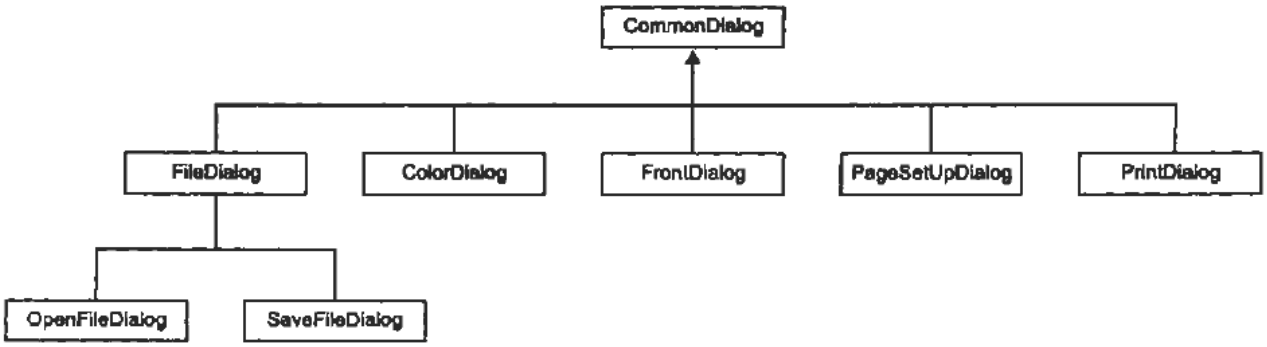
8. ВИКОРИСТАННЯ ДІАЛОГОВИХ ВІКОН

8.1 Клас CommonDialog. Класи OpenFileDialog та SaveFileDialog

Ми знаємо, як створювати прості діалогові вікна, щоб отримати інформацію від користувача. Ми також створювали багатовіконні програми, щоб вимагати у користувача більш специфічну інформацію. Однак для виконання стандартних операцій, таких як відкриття та збереження файлів, виведення на друк, рекомендується використовувати заздалегідь створений клас діалогових вікон. У цьому випадку не тільки зменшується обсяг коду, але й використовуються стандартні, добре всім знайомі діалогові вікна **Windows**.

У **.NET Framework** є класи, які дозволяють використовувати діалогові вікна **Windows**. Їх використання дозволяє користувачеві не вивчати складну методологію, що було б необхідним для кодування таких функціональних можливостей з "нуля".

Наявні в .NET Framework класи діалогових вікон представлені на малюнку.



Це є невізуальні компоненти. У класі **CommonDialog** описані методи та події, доступні для будь-якого класу загального діалогового вікна. Всі ці класи побудовані на основі спільного діалогового вікна **Windows** для того, щоб зробити його доступним для застосунків **.NET**.

Клас **CommonDialog**:

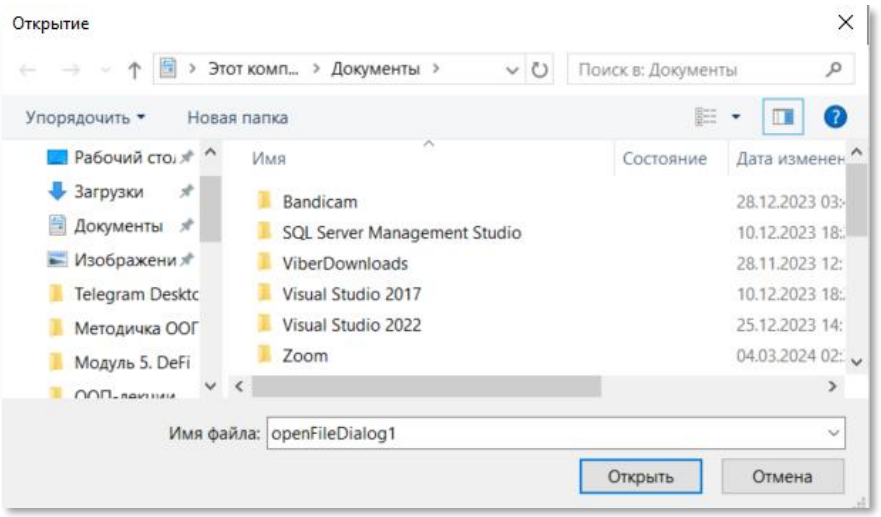
1. **ShowDialog() –** цей метод реалізується у похідних класах для виведення загального діалогового вікна.
2. **Reset()** – кожен похідний клас діалогового вікна реалізує метод **Reset()** для встановлення значень за замовчанням для всіх властивостей даного класу.

Класи **OpenFileDialog** та **SaveFileDialog** є похідними від базового абстрактного

класу **FileDialog**, який додає загальні файлові характеристики для діалогових вікон під час відкриття та збереження файлів.

Клас **FileDialog** відображає діалогове вікно, за допомогою якого користувач може вибрати файл. **FileDialog** – це абстрактний клас, який містить поширену поведінку для класів **OpenFileDialog** та **SaveFileDialog.** Цей клас не призначений для прямого використання, проте містить поширену поведінку для згаданих двох класів. Неможливо створити екземпляр класу **FileDialog**. Для створення діалогового вікна, призначеного для вибору або збереження файлу, використовується **OpenFileDialog** або **SaveFileDialog. FileDialog** є модальним діалоговим вікном. Тому, якщо це вікно відображається, частина програми, що залишилася, блокується до тих пір, поки користувач не обере файл.

Щоб відобразити діалогове вікно, виконується команда виду:
openFileDialog1.ShowDialog();



Перед викликом методу **ShowDialog()** є можливість задати значення деяким властивостям, які впливатимуть на поведінку та зовнішній вигляд діалогового вікна або накладатимуть обмеження на файли, які можуть бути відкриті.

Як заголовок вікна **OpenFileDialog** за замовчанням використовується слово ***«Відкрити»***. Заголовок діалогу може бути змінений, для чого необхідно надати відповідне значення властивості **Title**. Слово ***«Відкрити»*** не завжди є найкращим варіантом назви, наприклад, для програми, в якій необхідно проаналізувати файли реєстрації з метою вилучення з них будь-якої інформації або визначення розміру файлу і яке закриває файл безпосередньо відразу після виконання необхідних дій. У таких випадках файли не залишаються відкритими

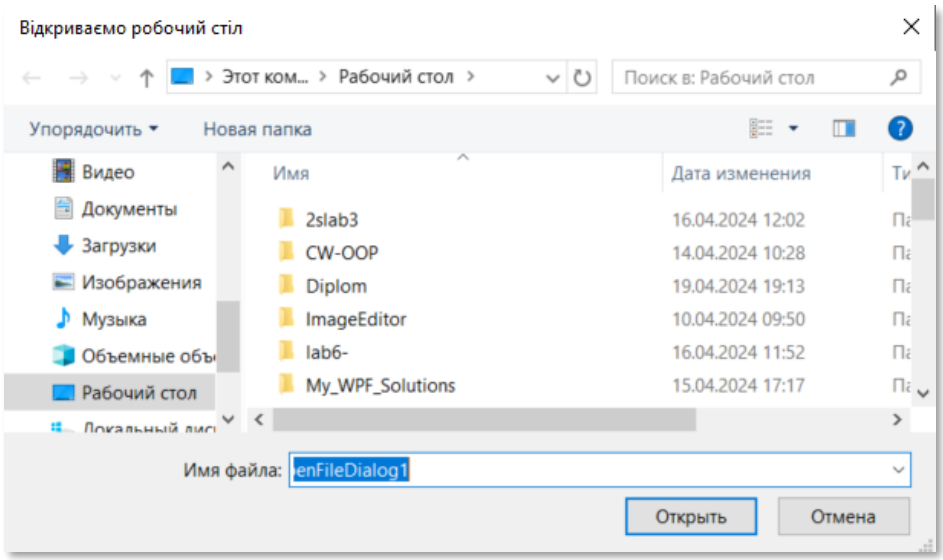
для користувача, тому назва типу ***«Аналіз файлів»*** може виявитися більш підходящою.

```
openFileDialog1.Title = " Аналіз файлів ";
openFileDialog1.ShowDialog();
```

Діалог відкриває директорію за замовчанням. При другому зверненні до діалогового вікна буде виводитися та ж директорія, в якій знаходиться останній файл. Загальне діалогове вікно **Windows**, яке використовується вікном **OpenFileDialog**, застосовує реєстр для зберігання імені останнього файлу. Ця поведінка може бути змінена за допомогою завдання властивості **InitialDirectory**. При використанні класів, похідних від **FileDialog**, уникайте використання рядкових літералів, які містять абсолютні шляхи. Натомість слід динамічно отримати шлях. Залежно від типу програми, зберігання даних, пов'язаних з програмою, і причиною доступу до файлової системи існує безліч можливих шляхів створення шляху до каталогу.

Стандартні шляхи **Windows.** Для отримання доступу до спеціальних системних папок можна скористатися статичним методом **GetFolderPath()** класу **System.Environment**. Метод **GetFolderPath()** приймає параметр типу **Environment.SpecialFolder**, що перераховується, в якому можна задати системну директорію, шлях до якої необхідно отримати.

```
openFileDialog1.Title = " Відкриваємо робочий стіл ";
string dir = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
    openFileDialog1.InitialDirectory = dir;
    openFileDialog1.ShowDialog();
```



Зауваження. *Клас* ***Environment*** *використовувався нами для генерування символу нового рядка під час роботи з багаторядковим текстовим вікном.*

Клас **Application** містить статичні елементи для отримання певних шляхів, такі як

StartupPath (шлях для виконуваного файлу, що запустив програму, не включаючи ім'я, що виконується);

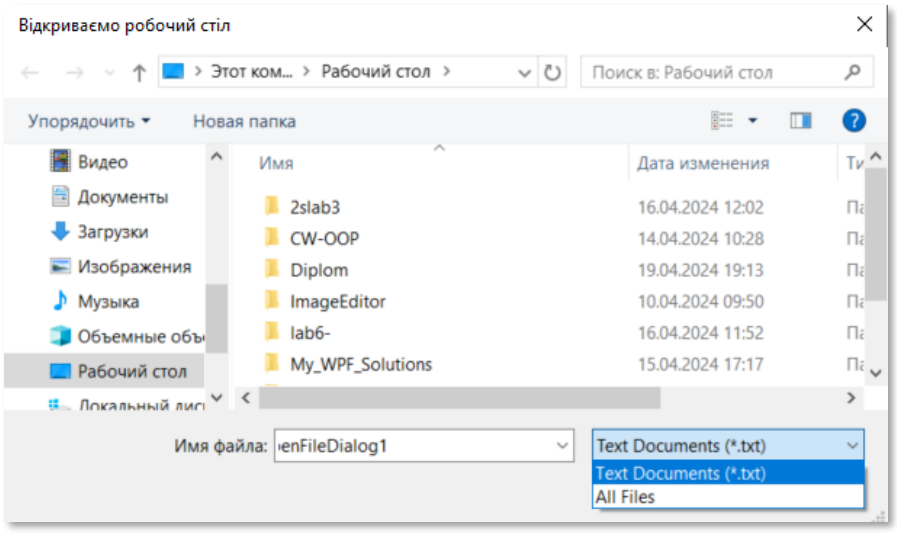
ExecutablePath (ім'я для виконуваного файлу, що запустив програму, включаючи ім'я, що виконується).

```
openFileDialog1.Title = " Відкриваємо файл ";
string dir = Application.StartupPath;
openFileDialog1.InitialDirectory = dir;
openFileDialog1.ShowDialog();
```

У діалогових вікнах роботи з файлами можна встановити фільтр для файлів. Фільтр визначає типи файлів, які користувач може вибирати для відкриття. Фільтр складається з кількох відділів, розділених вертикальною межею (|). Він може складатися лише з парних рядків, тому кількість відділів завжди має бути парною. Перший рядок визначає текст, який виводитиметься у вікні зі списком; другий рядок використовується для завдання розширень файлів, які виводитимуться в процесі діалогу. Рядок фільтра присвоюється властивості **Filter**. Цю властивість можна ставити як на етапі візуального програмування, наприклад *Text Documents (*.txt) |*.txt|All Files |*.**

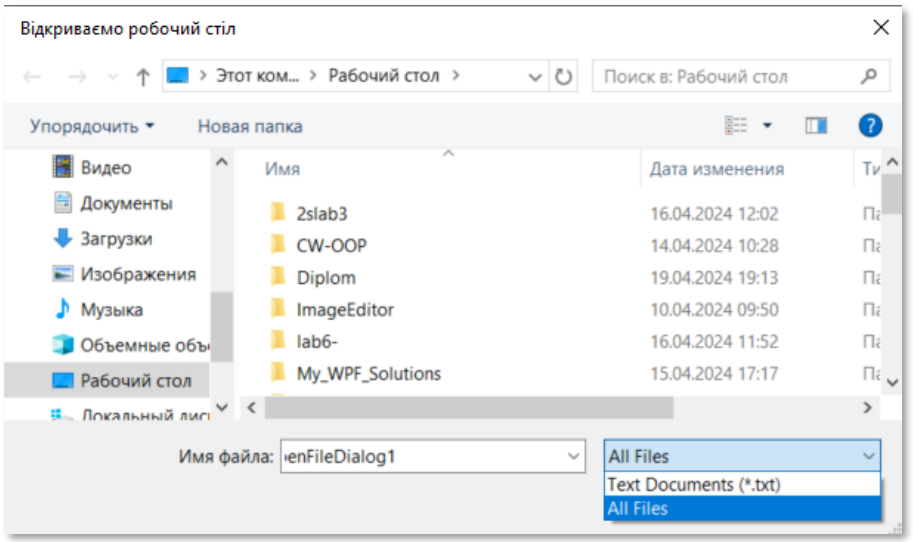
або програмним шляхом:

```
openFileDialog1.Filter = "Text Documents (*.txt) |*.txt|All Files |*.*";
```



Властивість **FilterIndex** визначає стандартний вибір у вікні зі списком. Якщо є можливість вибору з кількох типів файлів, можна присвоїти властивості **FilterIndex** тип файлу, який використовується за замовчуванням. Зверніть увагу, що **FilterIndex** рахується з 1!

openFileDialog1.FilterIndex = 2;



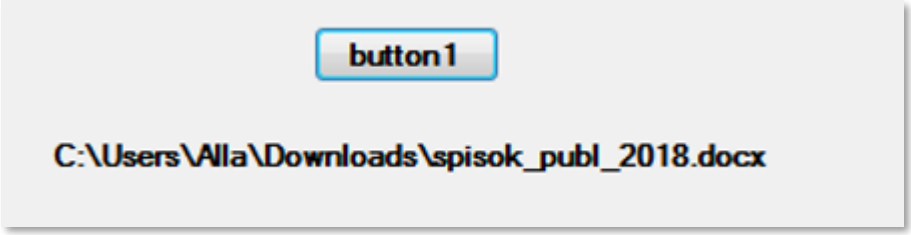
В результаті в полі Тип файлу завантажиться рядок **All Files**.

Видно, що у всіх прикладах у полі **Ім'я файлу** завантажується за промовчанням рядок ***"openDialog1"***. Можна задавати ім'я файлу, яке відображатиметься при відкритті діалогового вікна за допомогою властивості **FileName**.

openFileDialog1.FileName = "Privit.txt";

Ця властивість визначає ім'я вибраного файла. В ім'я файлу включаються шлях до файлу та розширення. За відсутності вибраних файлів, цей метод повертає порожній рядок (""). Ця властивість може бути виключно ім'ям одного вибраного файлу.

label1.Text = openFileDialog1.FileName;



Є також властивості:

1. **AddExtension** – повертає або задає значення, що визначає, чи додає автоматично діалогове вікно розширення до імені файлу, якщо

користувач опускає це розширення. Має значення true, якщо діалогове вікно додає автоматично розширення до імені файлу, коли користувач опускає це розширення; в іншому випадку - **false**. Значенням за замовчуванням є **true**.

2. **DefaultExt** - повертає або задає розширення імені файлу за замовчанням. Має тип **String**. Рядок не містить крапку.

Метод **ShowDialog()** повертає результат у вигляді перерахованого типу **DialogResult**. З цим типом ми знайомилися, коли вивчали модальні форми. Значення **None** — це значення, яке використовується за замовчанням, доки користувач не закрив діалог. Результат, що повертається, залежатиме від того, яку кнопку натиснув користувач. У діалоговому вікні **OpenFileDialog** можна повернути лише значення **DiaiogResult.OK** і **DiaiogResult.Cancel**. Якщо користувач натиснув кнопку **ОК**, доступ до обраного користувачем імені файлу можна отримати за допомогою властивості **FileName**. Якщо користувач скасує діалог, то властивість **FileName** міститиме порожній рядок

```
if (openFileDialog1.ShowDialog() == DialogResult.OK)
        label1.Text = openFileDialog1.FileName;
    else
        label1.Text = " Файл не вибрано ";
```

Діалогове вікно **OpenFileDialog** може бути використане для множинного вибору. Для цього необхідно встановити **true** властивість **Multiselect**. Значення за замовчанням – **false.** У цьому випадку імена файлів, вибрані користувачем, знаходяться у властивості **FileNames**. Це масив рядків. Кожне ім'я файлу включає шлях до файлу і розширення. За відсутності вибраних файлів цей метод повертає порожній масив.

```
openFileDialog1.Title = " Відкриваємо файл ";
openFileDialog1.Multiselect = true;
openFileDialog1.Filter = "Text Documents (*.txt) |*.txt|All Files |*.*";
if (openFileDialog1.ShowDialog() == DialogResult.OK)
{    foreach (string s in openFileDialog1.FileNames)
     label1.Text += s + "\n";    }
 else
     label1.Text = " Файл не обрано ";
```

8.2 Діалогове вікно SaveFileDialog.

Класи **SaveFileDialog** і **OpenFileDialog** дуже схожі і мають цілий набір однакових властивостей. У цій лекції ми не розглядатимемо властивості, які працюють так само, як і для діалогового вікна відкриття файлів. Натомість ми зосередимо увагу на властивостях, властивих лише діалоговому вікну збереження файлів.

У вікні **SaveFileDialog** потрібне виконання деяких додаткових перевірок та завдання значень деяких додаткових властивостей. По-перше, якщо властивості **CreatePrompt** присвоєно значення **true**, буде виведено запит, чи має створюватись новий файл. Якщо властивості **OverwritePrompt** присвоєно значення **true**, це означає, що буде виводитися запит, чи дійсно користувач хоче здійснити запис на місце вже існуючого файлу. За замовчуванням **OverwritePrompt** присвоюється значення **true,** а властивості **CreatePrompt** — **false**. При таких установках, якщо користувач спробує зберегти існуючий файл, буде виведено повідомлення про існування такого файлу.

8.3 Діалогове вікно FontDialog.

Діалогове вікно **FontDialog** дозволяє користувачеві програми здійснювати вибір шрифту. Користувачеві надається можливість змінювати шрифт, стиль, розмір та колір шрифту. Дане діалогове вікно може використовуватися так само, як і всі попередні.

Код, призначений для використання **FontDialog**, може виглядати так:

```
if (fontDialog1.ShowDialog() == DialogResult.OK)
        label1.Font= fontDialog1.Font;
```

Властивості класу **FontDialog:**

- **AllowVectorFonts** – логічне значення, яке визначає, чи можуть зі списку шрифтів вибиратися векторні шрифти.
- **AllowVerticalFonts** – логічне значення, яке визначає, чи можуть зі списку шрифтів вибиратися вертикальні шрифти. Вертикальний запис текстів використовується у деяких далекосхідних країнах. Цілком імовірно, що на вашому комп'ютері немає жодного інстальованого вертикального шрифту.
- **FixedPitchOnly** – встановлення цієї властивості призведе до того, що у

списку шрифтів виводитимуться лише шрифти з фіксованими розмірами. У шрифтів з фіксованими розмірами всі літери мають однакові розміри. Значення за промовчанням — **false**.

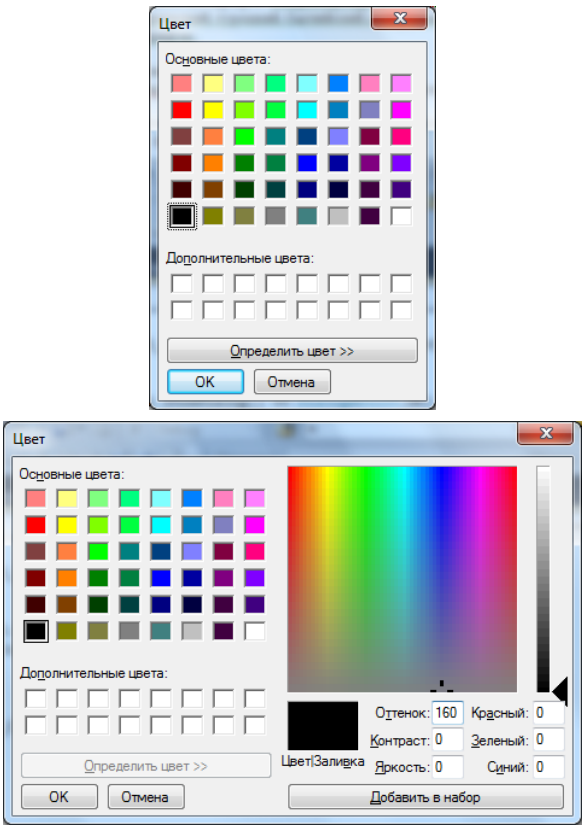
- **MaxSize** – дозволяє визначити максимальний розмір шрифту, який може вибрати користувач.
- **MinSize** – дозволяє визначити мінімальний розмір шрифту, який може вибирати користувач.
- **ShowApply** – якщо потрібно вивести кнопку **Apply**, то цій властивості слід присвоїти значення **true**. Натиснувши кнопку **Apply**, користувач отримує можливість переглянути, як виглядає новий шрифт у застосунку, не залишаючи діалогу вибору шрифтів.
- **ShowColor** – за замовчуванням можливість вибору кольору в цьому діалозі відсутня. Якщо ви хочете надати користувачеві можливість вибирати колір у діалозі вибору шрифту, то властивості **ShowColor** мають бути присвоєні значення true.
- **ShowEffects** – За замовчуванням можна встановлювати прапорці у вікнах вибору елементів **Strikeout** (закреслення) та **Underlin**e (підкреслення) для маніпуляції над шрифтами. Якщо ви не бажаєте, щоб ці можливості виводилися на екран, то властивості **ShowEffects** слід присвоїти значення **false**.
- **AllowScriptChange** – Присвоювання властивості **AllowScriptChange** значення **false** накладає заборону зміну користувачем стилю вибраного шрифту. Стилі, доступні користувачеві, залежать від вибраного ним шрифту. Наприклад, шрифт **Arial** підтримує західний, єврейський, арабський, грецький, турецький, балтійський, центрально-європейський, кириличний та в'єтнамський шрифти.

8.4 Діалогове вікно ColorDialog.

У діалоговому вікні **ColorDialog** не так багато можливостей, що настроюються, як у **FontDialog**.

Код, призначений для використання ColorDialog, може виглядати так:

```
if (colorDialog1.ShowDialog() == DialogResult.OK)
        label1.ForeColor = colorDialog1.Color;
```



- **AllowFullopen** – надавши цій властивості значення **false**, можна заборонити користувачеві самостійно визначати кольори, відключивши кнопку Визначити колір. Значення цієї властивості за замовчанням — **true.**
- **FullOpen** – надавши цій властивості значення **true,** перед тим як діалог буде виведений, можна домогтися автоматичного відкриття діалогового вікна з можливістю створення власних кольорів.
- **AnyColor** – надавши цій властивості значення true, можна отримати всі допустимі кольори в списку основних кольорів.
- **SolidColorOnly** – якщо цій властивості присвоєно значення **true**, то користувач має можливість вибирати лише однорідні кольори.

8.5 Клас FolderBrowserDialog.

Цей клас надає спосіб видачі запрошення користувача для перегляду, створення та, зрештою, для вибору папки. Цей клас використовується, коли потрібно дозволити користувачеві вибирати лише папки, але не файли. Огляд папок здійснюється за допомогою дерева. Можуть вибиратися лише папки із файлової системи; вибір віртуальних папок неможливий. Об'єкт

FolderBrowserDialog є модальним діалоговим вікном.

```
folderBrowserDialog1.ShowDialog();
```

Зазвичай після створення нового об'єкта **FolderBrowserDialog** встановлюється розташування **RootFolder**, з якого починається перегляд. Властивість **RootFolder** отримує або задає кореневу папку, з якої починається перегляд. Властивість має тип **Environment.SpecialFolder**. Стандартне значення — **Desktop** (Логічний робочий стіл, а не фізичне розташування файлів системи).

```
folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
    folderBrowserDialog1.ShowDialog();
```

Додатково можна встановити для **SelectedPath** абсолютний шлях вкладеної папки RootFolder, який буде вибиратися спочатку. Властивість **SelectedPath** має тип **String**. Містить шлях папки, вперше обраної в діалоговому вікні, або останньої папки, обраної користувачем. Значення за замовчанням - порожній рядок ("").

Якщо властивість **SelectedPath** встановлена перед відображенням діалогового вікна, буде вибрано папку з цим шляхом.

```
folderBrowserDialog1.SelectedPath = "D:\\";
folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
folderBrowserDialog1.ShowDialog();
```

Також є можливість додатково встановити властивість Description, щоб надати користувачеві додаткові інструкції.

```
folderBrowserDialog1.SelectedPath = "D:\\";
folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
folderBrowserDialog1.Description = " виберіть папку для встановлення ";
folderBrowserDialog1.ShowDialog();
```

Якщо метод **ShowDialog()** повертає **OK**, що вказує на те, що користувач натиснув кнопку **ОК**, властивість **SelectedPath** поверне рядок, що містить шлях до вибраної папки. Якщо метод **ShowDialog()** повертає значення Cancel, що вказує на те, що користувач виконав скасування діалогового вікна, ця властивість матиме те саме значення, яке воно мало до відображення діалогового вікна. Якщо користувач вибирає папку, яка не має фізичного шляху

(наприклад, **"Мій комп'ютер"**), кнопку ОК діалогового вікна буде відключена. Коли діалогове вікно закривається і його результатом із **ShowDialog** є **DialogResult.OK,** об'єкт **SelectedPath** буде рядком, що містить шлях до обраної папки.

```
folderBrowserDialog1.SelectedPath = "D:\\";
folderBrowserDialog1.RootFolder = Environment.SpecialFolder.MyComputer;
folderBrowserDialog1.Description = " виберіть папку для встановлення ";
        if (folderBrowserDialog1.ShowDialog() == DialogResult.OK)
            label1.Text = folderBrowserDialog1.SelectedPath;
```

Можна використовувати властивість **ShowNewFolderButton**, щоб вказати, чи є у користувача можливість створення нових папок за допомогою кнопки **New Folder**. Значення true, якщо кнопка **New Folder** відображається у діалоговому вікні; в іншому випадку - значення **false.** Значення за промовчанням — **true**.

```
folderBrowserDialog1.ShowNewFolderButton = false;
```

9. КЕРУЮЧІ ЕЛЕМЕНТИ ListBox, ComboBox ТА PictureBox.

9.1 Клас ListBox

Класи **ListBox** і **ComboBox** є похідними від класу **ListControl**, що надає базові функціональні можливості для двох керуючих елементів з використанням списку.

Вікна зі списком **ListBox** використовуються для виведення списку рядків, з яких одночасно можна вибрати один або кілька рядків. Вікно зі списком так само, як вікна з вибором та перемикачі, надають можливість запитувати вибір користувача, що складається з одного або кількох найменувань. Вікна зі списками слід використовувати, коли в період розробки не відома фактична кількість елементів, з яких може вибирати користувач (як приклад можна взяти список співробітників). Навіть якщо на момент розробки відомі всі можливі значення, використання вікна зі списком переважно, якщо кількість таких значень дуже велика.

Властивості керуючого елемента ListBox:

1. **Items** − Містить усі елементи вікна зі списком. Властивості цього сімейства можна використовувати для додавання та видалення елементів. Наприклад, для форми на етапі візуального програмування було задано 5

рядків:

Перший

Другий

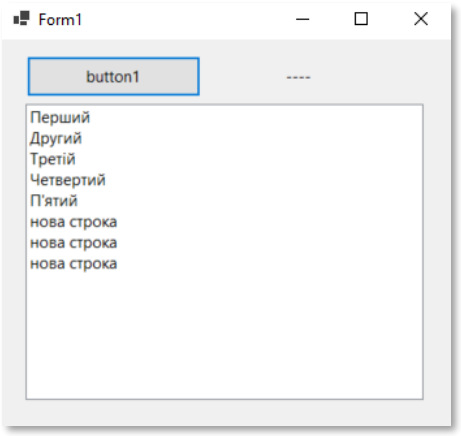
Третій

Четвертий

П'ятий

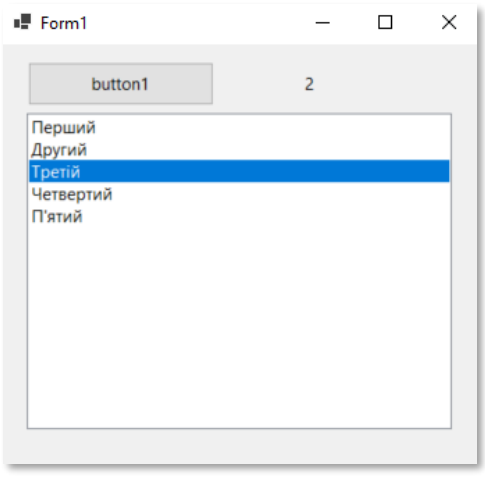
Ця властивість має тип списку, тому для нього можна використовувати функції зі списками, наприклад:

```
private void button1_Click(object sender, EventArgs e)
    {       listBox1.Items.Add("нова строка");       }
```



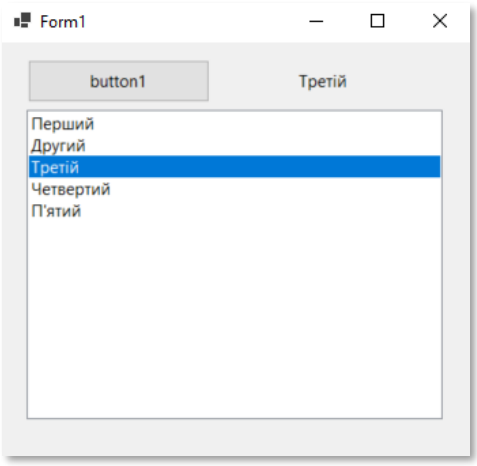
2. **SelectedIndex** – значення цієї властивості представляє індекс вибраного елемента у вікні зі списком, що рахується з нуля. Якщо вікно зі списком допускає одночасний вибір кількох елементів, ця властивість містить значення першого вибраного елемента списку.

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
    {       label1.Text = Convert.ToString(listBox1.SelectedIndex);      }
```



3. **SelectedItem** – якщо у вікні зі списком допускається вибір лише одного елемента, ця властивість міститиме обраний елемент, якщо такий існує. Якщо у вікні допускається вибір більш ніж одного елемента, воно міститиме перший із вибраних елементів. Має тип **Object.**

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
    {       label1.Text = Convert.ToString(listBox1.SelectedItem);      }
```



4. **SelectionMode** – існує можливість задати один із чотирьох різних режимів вибору у вікні зі списком:
   - **None**: Елементи не можуть бути обраними.
   - **One**: У кожний момент часу допускається вибір лише одного елемента.
   - **MultiSimple**: Допускається вибір кількох елементів.
   - **MultiExtended**: Дозволяється вибір кількох елементів, і користувач при здійсненні свого вибору може використовувати клавіші Ctrl, Shift та стрілки.
5. **SelectedIndices** – є список, у якому містяться індекси всіх вибраних елементів у вікні зі списком (відлік індексів ведеться з нуля). Має властивість **Count.**

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
    {
      label1.Text = "";
      for (int i = 0; i < listBox1.SelectedIndices.Count;i++)
         label1.Text += Convert.ToString(listBox1.SelectedIndices[i])+"\n";
    }
```

6. **SelectedItems** − являє собою список, в якому містяться всі елементи, вибрані зараз

```
private void listBox1_SelectedIndexChanged(object sender, EventArgs e)
        {
           label1.Text = "";
           for (int i = 0; i < listBox1.SelectedItems.Count;i++)
              label1.Text += Convert.ToString(listBox1.SelectedItems[i])+"\n";
        }
```

7. **MultiColumn** − вікно зі списком може мати більше одного стовпця. Це властивість булевого типу.

8. **ColumnWidth** − у вікні зі списком, яке допускає наявність кількох стовпців елементів, ця властивість визначає ширину стовпців. Наприклад, задаємо значення 50.

9. **Sorted** − привласнення цієї властивості значення true призведе до того, що всі елементи, що містяться в елементі керування **ListBox**, будуть відсортовані в алфавітному порядку.

```
listBox1.Sorted = true;
```

10. **Text** − властивість **Text** зустрічалася в багатьох керуючих елементах, проте в даному випадку вона працює зовсім інакше, порівняно з тим, що ми бачили досі. Якщо властивості **Text** керуючого елемента **ListBox** надається деяке значення, воно здійснює пошук елемента з відповідним текстом і позначає його як обраний. Якщо значення властивості **Text** зчитується, то як значення, що повертається використовується перший вибраний елемент зі списку. Ця властивість не може використовуватися, якщо значення властивості **SelectionMode** дорівнює **None**.

```
listBox1.Text = "Перший";
label1.Text = listBox1.Text;
```

Зазвичай для роботи з елементом управління **ListBox** необхідно знати про настання подій, пов'язаних з тим, які елементи вибирає користувач. Подія **SelectedlndexChanged** настає тоді, коли змінюється індекс обраного елемента.

9.2 Клас ComboBox

Комбіноване вікно ComboBox поєднує в собі кілька елементів, що управляють, а саме **Textbox** і **ListBox**. На відміну від **ListBox** не існує можливості за жодних обставин вибирати більше одного елемента зі списку елементів, що містяться в **ComboBox**, крім того існує додаткова можливість вводити нові входження в список в **Textbox**, що є складовою частиною **ComboBox.**

Зазвичай керуючий елемент **ComboBox** використовується для економії місця, що відводиться для діалогу, оскільки та частина комбінованого вікна, яку видно постійно, складається з текстового вікна та кнопки. Коли користувач натискає стрілку, розташовану праворуч від текстового вікна, відкривається вікно зі списком, в якому користувач може здійснити свій вибір. Після того як процедура вибору завершена, вікно зі списком згортається і картинка, що виводиться, знову набуває свого звичайного вигляду.

Оскільки керуючому елементу **ComboBox** притаманні риси таких елементів, як **Textbox** і **ListBox**, багато властивостей і подій **ComboBox** відносяться також і до цих двох керуючих елементів.

Властивості елемента управління ComboBox:

1. **DropDownStyle** − при виведенні комбінованого вікна можуть використовуватися три різні стилі:
    - **DropDown**: Користувач має можливість редагувати текстове вікно, а для того щоб вивести на екран ту частину елемента, що містить список, повинен клацнути мишею на стрілці.
    - **Simple**: Аналогічно DropDown за винятком того, що та частина керуючого елемента, яка містить список, є постійно видимою - майже так само, як і для звичайного керуючого елемента **ListBox**.
    - **DropDownList**: Користувач не має можливості редагувати текстове вікно, а щоб вивести на екран ту частину керуючого елемента, яка містить список, повинен клацнути мишею на стрілці.
2. **DroppedDown** − вказує на те, в якому стані – розгорнутому або згорнутому – знаходиться та частина керуючого елемента, яка містить список. Якщо надати цій властивості значення **true**, то список буде розгорнутий.
3. **SelectedText**, **SelectionStart**, **SelectionLength** і **MaxLength** як у **Textbox.**

4. **Items**, **Selectedlndex**, **Selectedltem** і **Sorted** як у **ListBox.**
5. **Text** − Якщо цій властивості привласнити значення, що дорівнює **null**, то будь-який вибір у тій частині керуючого елемента, що містить список, буде скинутий. Якщо привласнити йому якесь існуюче у списку значення, це значення стає обраним. Якщо такого значення у списку не існує, текст буде просто виведений у текстовому вікні керуючого елемента **ComboBox.**

Існує три основні дії, що відбуваються в комбінованому вікні **ComboBox**, про які вам може знадобитися повідомлення:

- **SelectedlndexChanged** виникає у момент, коли змінюється вибір у частині керуючого елемента, що містить список.
- **DropDown** виникає в момент, коли розгортається частина елемента, що містить список.
- **TextChanged** виникає у момент зміни властивості **Text**.

9.3 Клас PictureBox.

Компонент **PictureBox** забезпечує відображення ілюстрації. Стандартний елемент керування відображається без меж. Властивість **BorderStyle** дозволяє відобразити стандартну або тривимірну межу, щоб відрізнити графічне вікно від інших областей форми, навіть якщо у цьому вікні немає зображення. Об'єкт **PictureBox** не є виборним елементом керування, що означає неможливість отримання ним фокуса введення.

Основна властивість **Image**, яка отримує або задає зображення, яке відображається елементом керування **PictureBox.** Цю властивість можна ставити як на етапі візуального програмування, так і програмним шляхом. На етапі візуального програмування використовують діалогове вікно **Select Resource**. Програмним шляхом завантажити зображення можна за допомогою методу **Load()**, який відображає зображення, вказане у властивості **ImageLocation** об'єкта **PictureBox**. Властивість **ImageLocation** отримує або вказує шлях зображення для відображення.

Режим відображення ілюстрації (спосіб масштабування), якщо розмір ілюстрації відповідає розміру компонента визначається властивістю **SizeMode**.

Можливі значення:

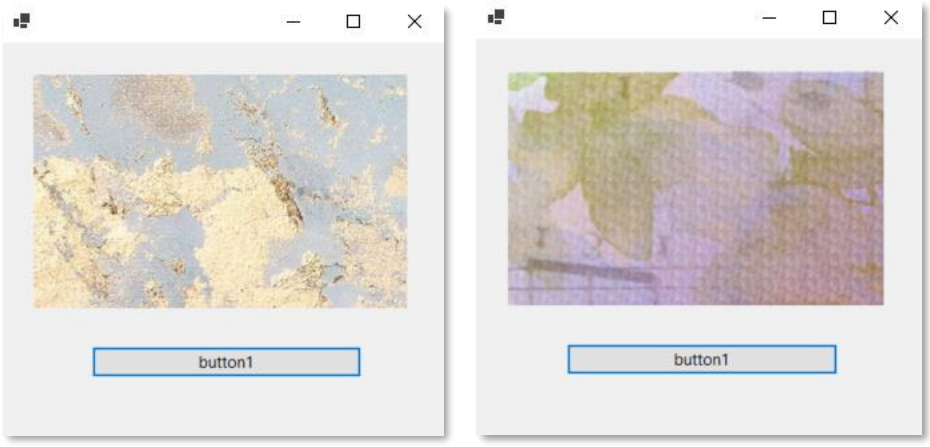
- **Normal** − масштабування не виконується (якщо розмір компонента менший за розмір ілюстрації, то відображається лише частина ілюстрації);
- **StretchImage** − виконується масштабування ілюстрації таким чином, що вона займає всю область відображення (якщо розмір компонента не пропорційний до розміру ілюстрації, вона буде спотворена);
- **AutoSize** − розмір компонента автоматично змінюється та відповідає розміру ілюстрації;
- **CenterImage** − центрування ілюстрації в полі компонента, якщо розмір ілюстрації менше області відображення.

Розглянемо приклад програми з використанням компонента **PictureBox**. Розташуємо на формі два компоненти: **PictureBox** і **Button**. Виберіть зображення за допомогою діалогового вікна **Select Resource** для властивості **Image**. Задамо властивість **SizeMode = StretchImage**.

Для кнопки запишемо наступну подію клацання:

```
private void button1_Click(object sender, EventArgs e)
{
   if (openFileDialog1.ShowDialog() == DialogResult.OK)
   {      pictureBox1.ImageLocation = openFileDialog1.FileName;
          pictureBox1.Load();      }
}
```

В результаті маємо:



10. СТВОРЕННЯ БІБЛІОТЕК

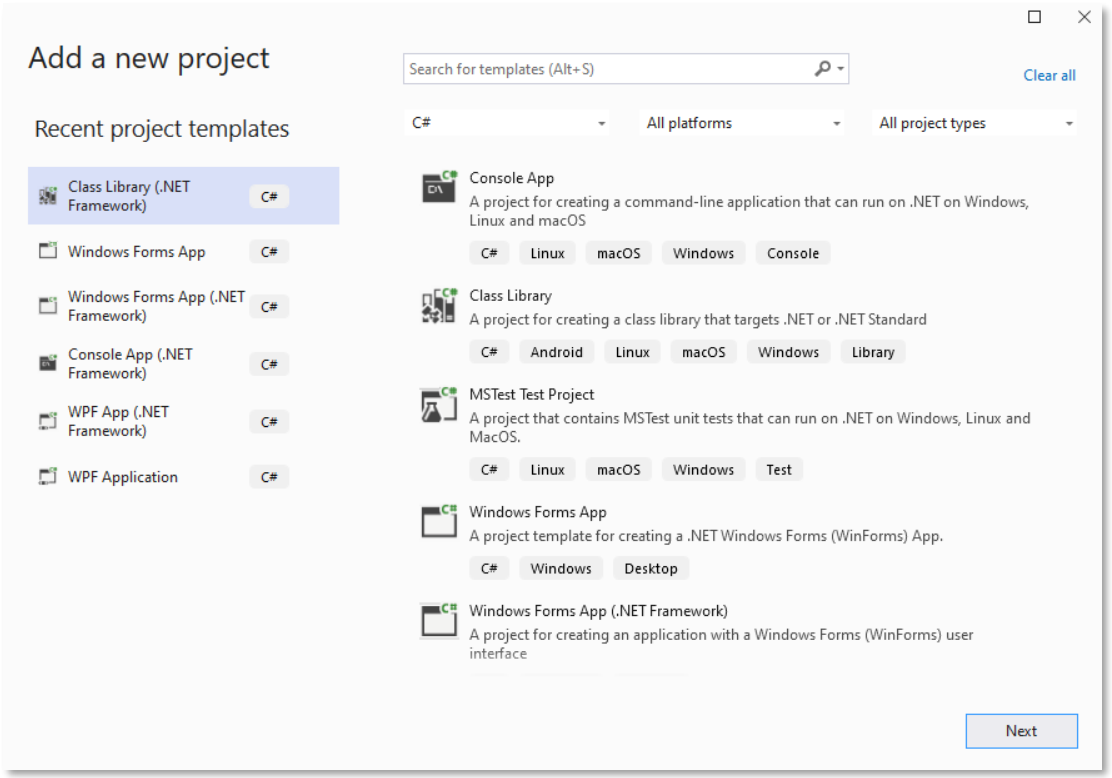
Рано чи пізно всі приходять до думки про об'єднання своїх власних напрацювань у єдиний пакет (ну або в кілька пакетів), щоб створені один раз класи можна було використовувати в різних програмах. У **C#** такі пакети називають збірками. Збірки можуть бути оформлені у вигляді **DLL**.

Класи можна розміщувати не тільки в окремих файлах у рамках одного проекту, а й у абсолютно незалежних проектах. Проект, в якому не міститься нічого, крім класів (разом з описами інших типів, що стосуються справи, але без точки входу), називається бібліотекою класів.

Проекти, що є бібліотеками класів, компілюються в модулі **.dll**, а доступ до їх вмісту можна отримати, включаючи інші проекти посилання на них. Такий підхід дозволяє збільшити ступінь інкапсуляції, що забезпечується об'єктами, оскільки процес зміни та вдосконалення бібліотек класів може відбуватися, абсолютно не торкаючись тих проектів, які використовують ці бібліотеки. Це дозволяє легко вдосконалювати послуги, що надаються різними класами (і може впливати відразу на кілька різних застосунків, що використовують ці класи).

Розглянемо приклад, у якому класи розмістимо у бібліотеці, а сам код у застосунку.

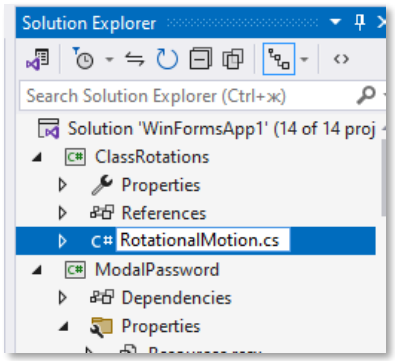
Вибираємо команду головного меню **File|New|Project.**



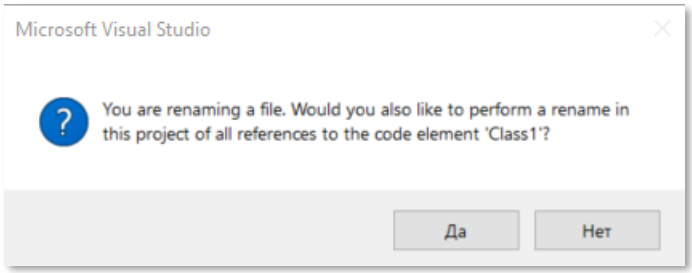
Введіть ім'я в першому полі **ClassRotations.** Згенерується код виду:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ClassRotations
{
   public class Class1
   {
   }
}
```

Перейменуйте файл Class1.cs на RotationalMotion.cs. Для цього необхідно клацнути правою кнопкою миші на ім'я файлу у вікні Solution Explorer і вибравши опцію Rename (Перейменувати).



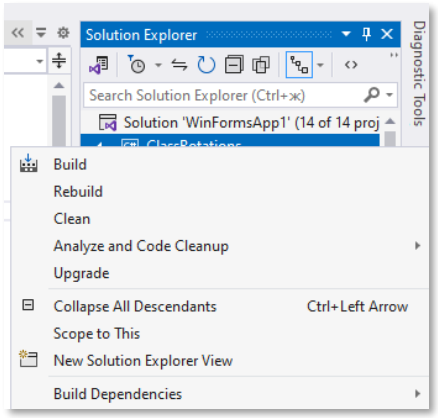
При цьому буде зроблено запит на перейменування класу.



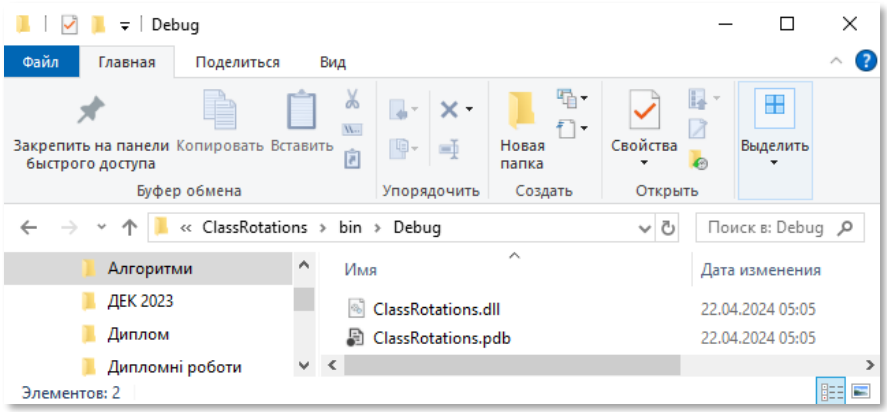
Додамо до класу код виду:

```
9     public class RotationalMotion
10    {
11        double omega0;
12        double epsilon;
13        public RotationalMotion(double om, double eps)
14        {
15            omega0 = om;
16            epsilon = eps;
17        }
18        public double Velocity(double r, double t)
19        {
20            return (omega0 + epsilon * t) * r;
21        }
22    }
```

Відкомпілюйте проект (зверніть увагу, що в цьому проекті немає точки входу, тому його неможливо запустити як звичайний проект - натомість ви можете просто побудувати його, скориставшись опцією меню **Build**).



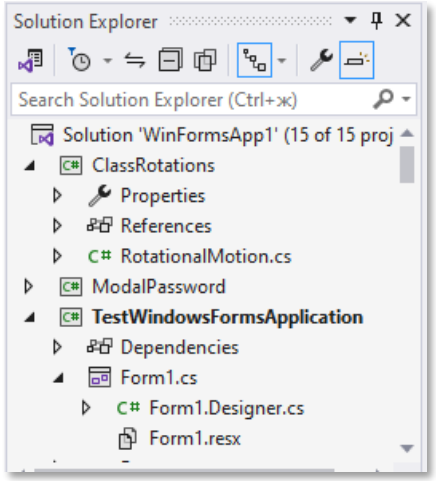
В результаті побудовано файл бібліотеки **ClassRotations.dll**.



Для тестування роботи створеної бібліотеки зробимо такі кроки.

Додамо до рішення новий проект **TestWindowsFormsApplication**:

В результаті в одному рішенні будуть обидва проекти:

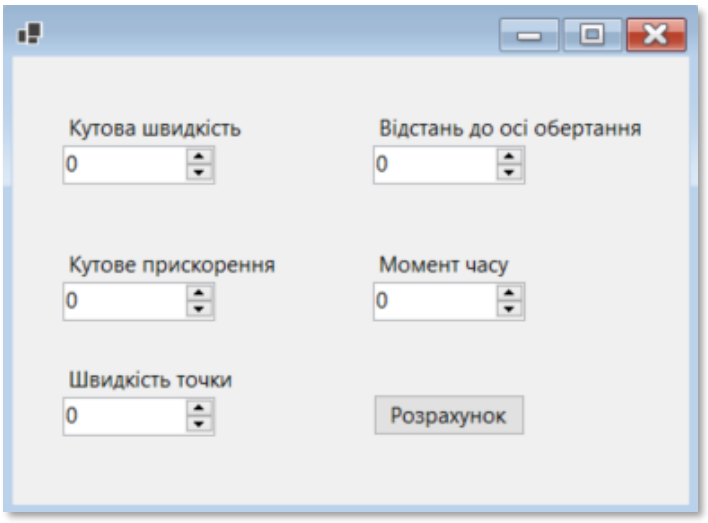


Зробимо стартовим проект TestWindowsFormsApplication.

На формі розмістимо 5 міток **Label** та 5 текстових полів типу **NumericUpDown**.

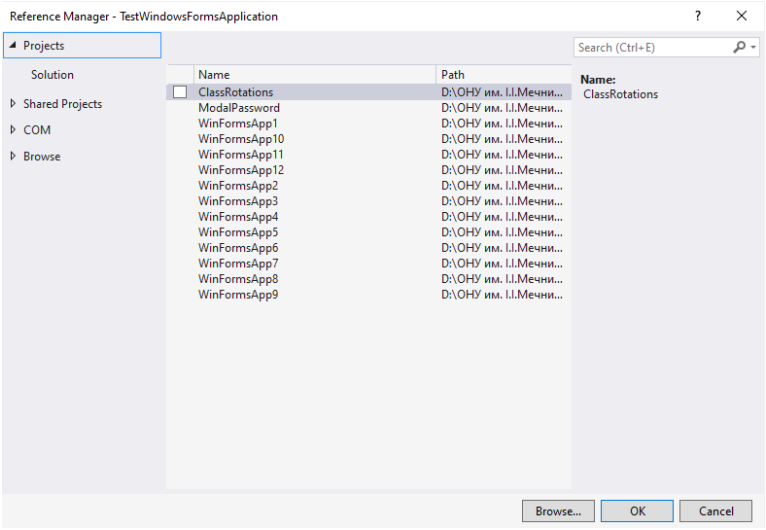
4 текстові поля призначені для введення значень початкової кутової швидкості

обертання тіла, кутового прискорення, моменту часу та радіус вектора точки, для якої обчислюється лінійна швидкість. П'яте текстове поле використовується для виведення числового значення.



Виберіть пункт меню **Project | Add Reference...** або виберіть ту саму опцію, двічі клацнувши правою кнопкою миші по пункту **References** у вікні **Solution Explorer**.

Відкриється вікно Add Reference.



Виберіть файл **ClassRotations.dll.** Переконайтеся, що до цього проекту додалося нове посилання (у вікні проекту Solution Explorer).

Щоб безпосередньо звертатися до класу **RotationalMotion** без імені посилання, необхідно написати команду виду:

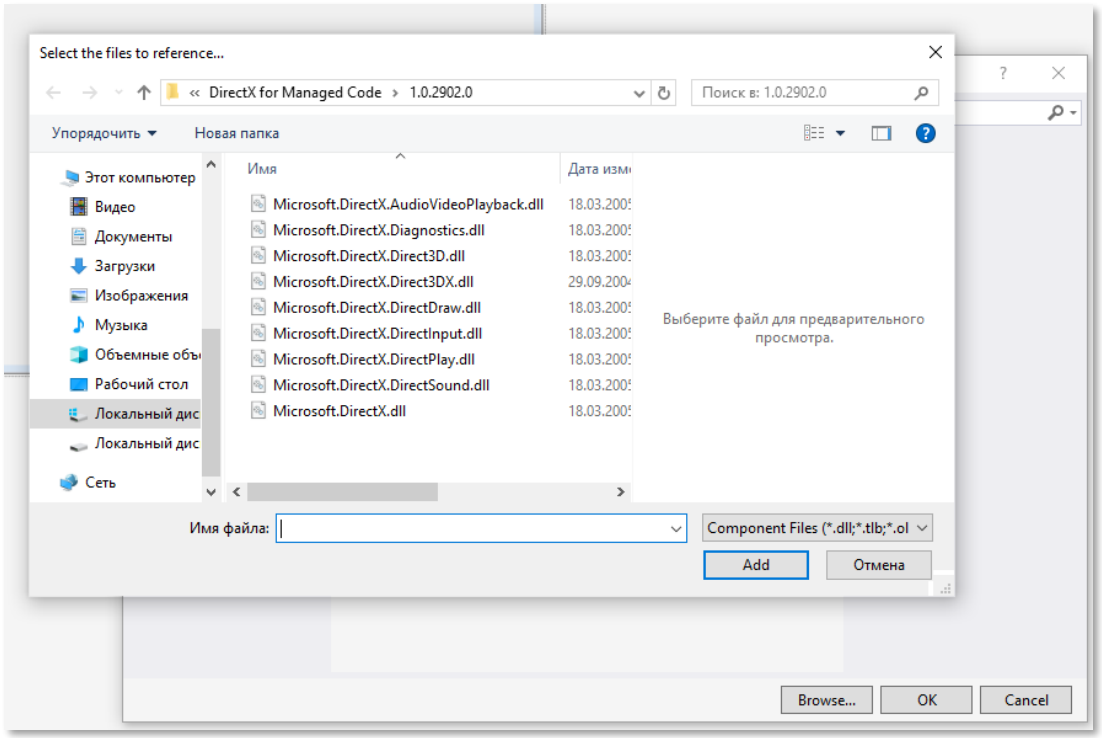
```
using ClassRotations;
```

На формі розмістимо кнопку, на яку встановимо **Text = «Розрахунок».** При натисканні на цій командній кнопці буде виконуватися функція:

```
private void Culc_Click(object sender, EventArgs e)
        {
            RotationalMotion r = new
RotationalMotion(Convert.ToDouble(nUpOmega0.Value),
Convert.ToDouble(nUpEpsilon.Value));
            tBV.Text=Convert.ToString(r.Velocity(Convert.ToDouble(nUpR.Value),
Convert.ToDouble(nUpt.Value)));
        }
```

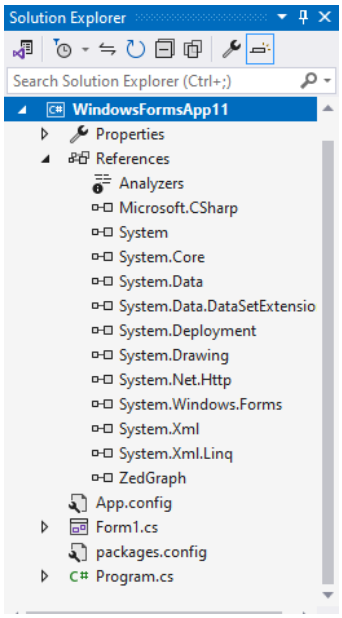
Примітка 1. *Ви можете переглянути, які об'єкти містяться в бібліотеці, вибравши подвійним клацанням у вікні* ***оглядача рішень****. Відкриється допоміжне вікно* ***оглядач об'єктів****.*

Примітка 2. *Якщо Ви хочете використовувати бібліотеку в проекті, який знаходиться в іншому рішенні, то при підключенні посилання необхідно шукати її за допомогою вкладки* ***Browse:***

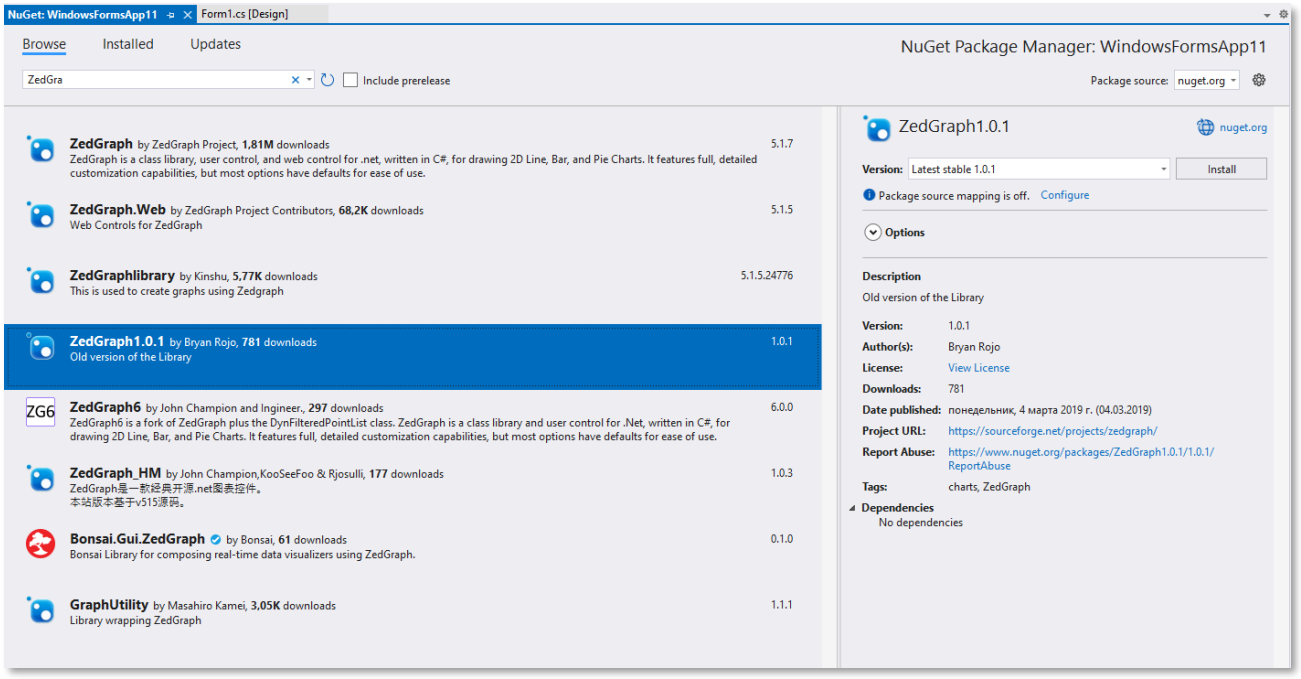


11. ПОБУДОВА ГРАФІКІВ

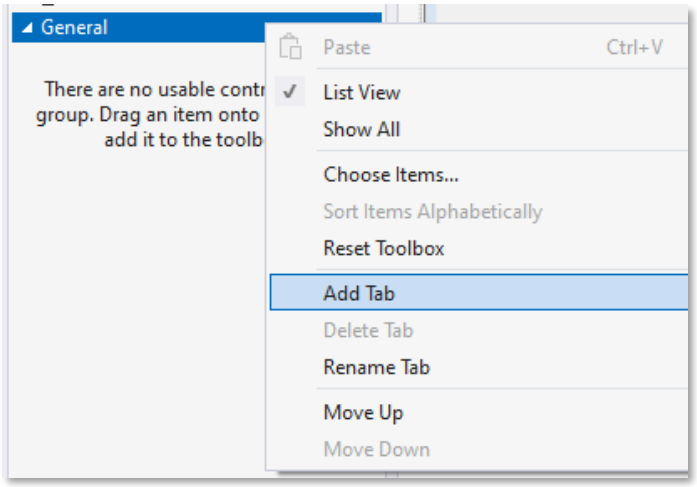
Для створення графіків можна використовувати бібліотеку **ZedGraph.dll**. Бібліотеку необхідно підключити до проекту за допомогою пункту меню **Project | Add Reference...** або виберіть ту ж опцію з контекстного меню, клацнувши правою кнопкою миші по пункту **References** у вікні **Solution Explorer**. Відкриється вікно **Add Reference**. Перейдіть на закладку **Browse.** Виберіть **ZedGraph.dll**. Переконайтеся, що цей проект додав нове посилання (вікно проекту Solution Explorer).



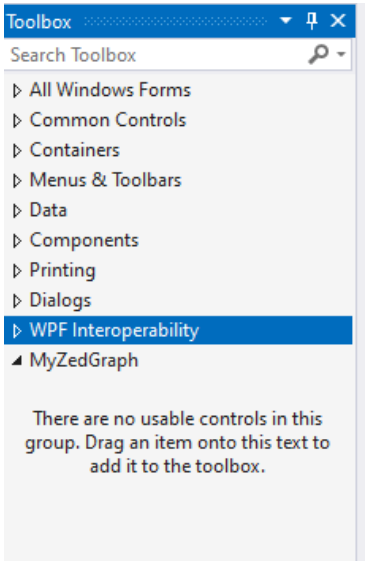
Або можна скористатися **NuGet Package Manager**



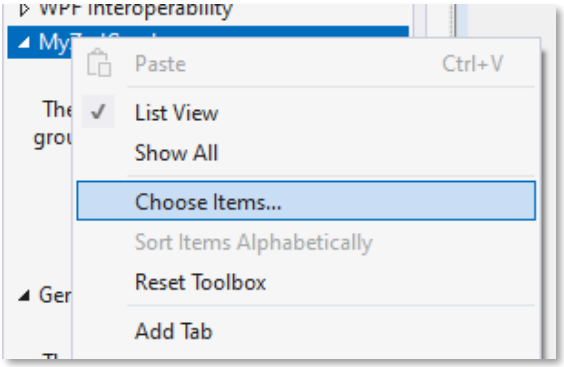
Бібліотека інкапсулює компонент **ZedGraphControl**, яку потрібно додати на панель **Toolbox**. Для зручності можна створити вкладку на панелі інструментів. Для цього викликається контекстне меню панелі інструментів та вибирається команда **Add Tab** (Додати вкладку).



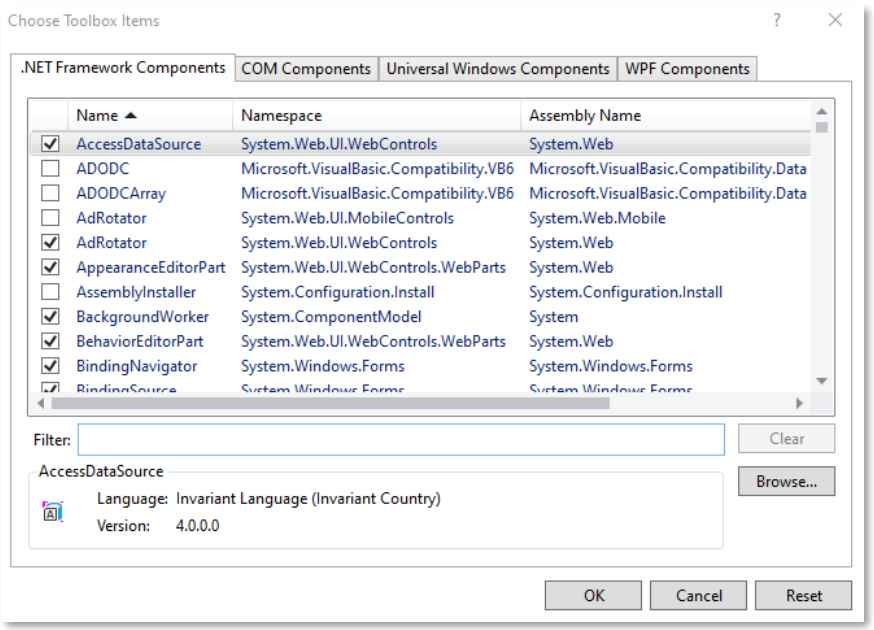
Створюється нова вкладка, на яку Ви задаєте ім'я.



Затем необходимо щелкнуть правой кнопкой на панели в новой вкладке и выбрать команду контекстного меню **Choose Items…**

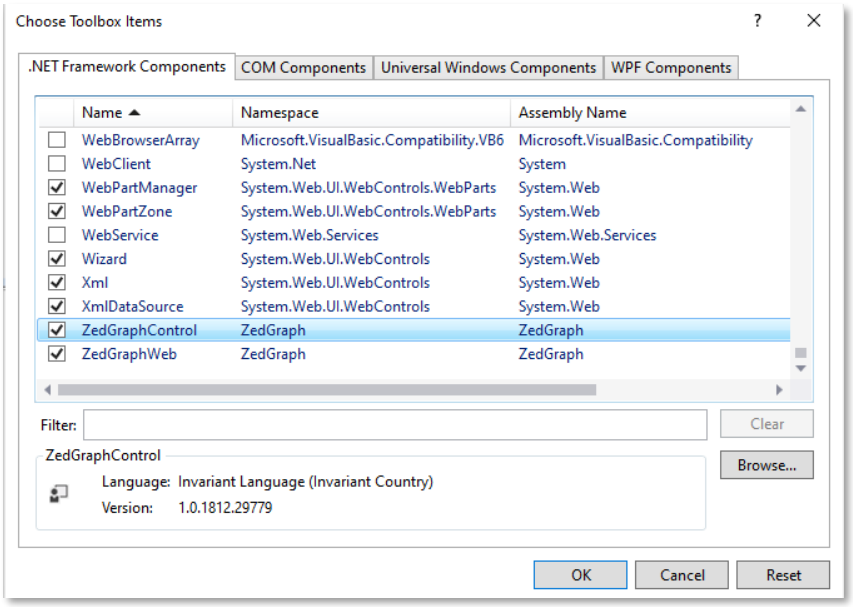


Відкриється вікно Вибір елементів панелі завдань:

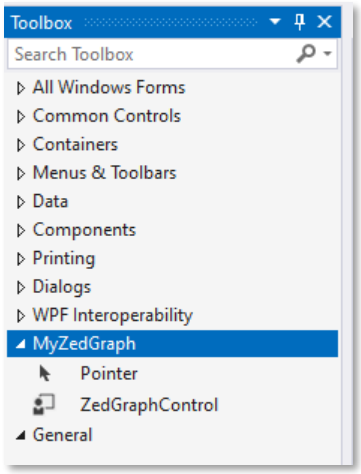


У списку бібліотек немає необхідної бібліотеки, її необхідно включити до цього списку. Для цього натисніть кнопку **Browse** (Огляд) вікна **Choose Toolbox Items** та вкажіть шлях до файлу **ZedGraph.dll.**

У вікні додасться відповідний рядок **ZedGraphControl**.



Після закриття вікна **Choose Toolbox Items** на панелі інструментів у створеній вкладці з'явиться піктограма компонента **ZedGraphControl**.

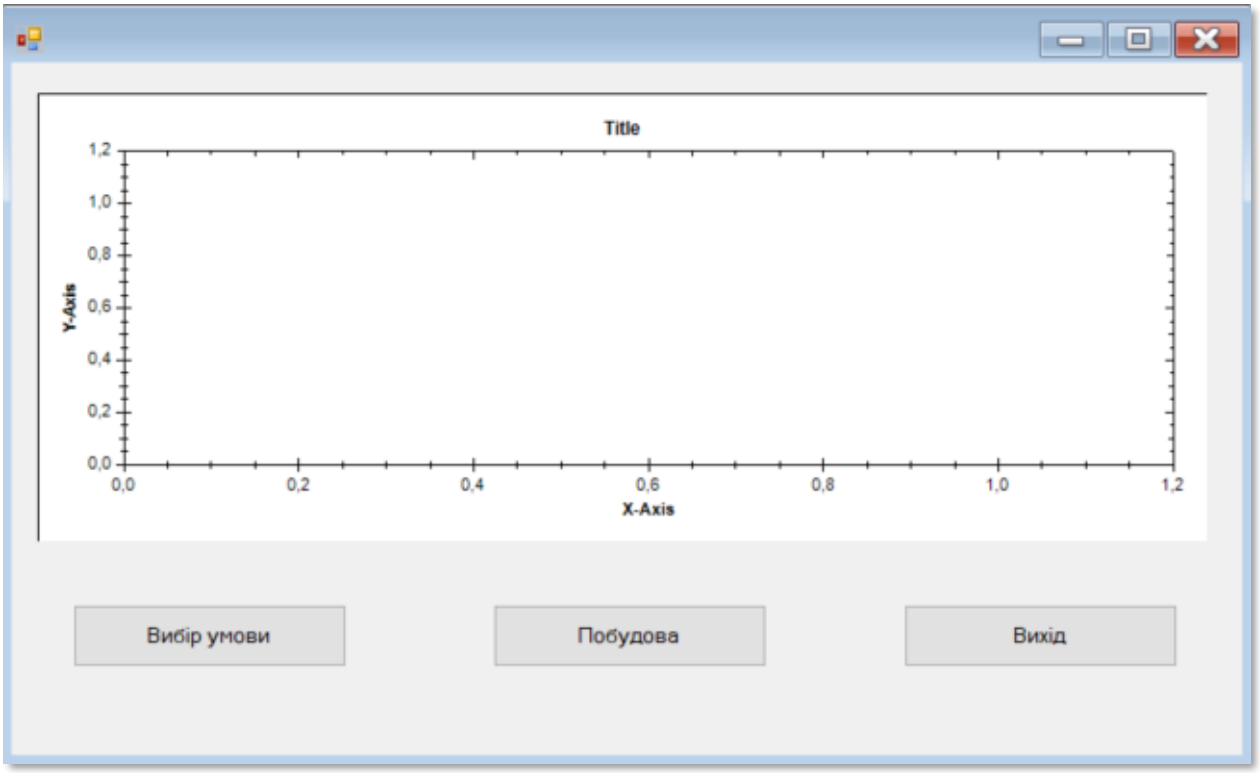


Для того щоб безпосередньо звертатися до бібліотеки **ZedGraph** без імені посилання, необхідно написати команду вигляду:

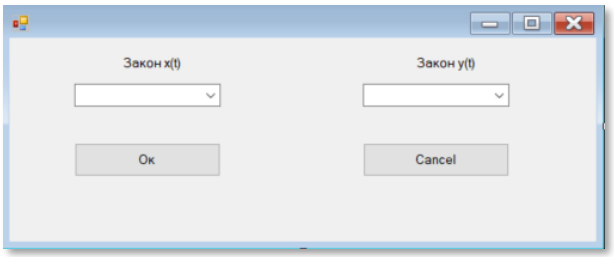
```
using ZedGraph;
```

Розглянемо на прикладі, як використовувати даний компонент для побудови графіка. Розробимо застосунок, за допомогою якого можна побудувати траєкторію руху матеріальної точки згідно із заданими законами зміни координат від часу.

Форма **Form1** на етапі візуального програмування має вигляд:



При натисканні на кнопці ***Вибір умови*** відкривається друга форма, яка дозволяє вибрати необхідний варіант.



На формі **Form2** є два **ComboBox**, які містять рядки виду: **0; 1; 2; 3; 4; 5; 6; 7; 8; 9**, що відповідає варіантам в методиці з механіки. У цих компонентах встановлено властивість **DropDownStyle=DropDownList** (без права введення, а лише вибір зі списку). Кнопки мають відповідні значення **DialogResult**. Для того, щоб запам'ятати вибіру користувача оголошено дві змінні в класі форми **Form1**:

```
static public int vx=0;
static public int vy=0;
```

При відкритті форми **Form2** у модальному режимі викликається подієва функція виду:

```
private void Form2_Shown(object sender, EventArgs e)
    {
       comboBox1.Text = Convert.ToString(Form1.vx);
       comboBox2.Text = Convert.ToString(Form1.vy);
    }
```

При виборі користувача умови для функцій **x(t)** та **y(t)** викликаються подієві функції:

```
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e)
    {
       Form1.vx = comboBox1.SelectedIndex;
    }
private void comboBox2_SelectedIndexChanged(object sender, EventArgs e)
    {
       Form1.vy = comboBox2.SelectedIndex;
    }
```

Для відображення графічних даних використовується клас **GraphPane**, який дозволяє додати до програми користувальницьку графіку. Для цього оголосимо відповідний об'єкт цього класу в класі форми **Form1.**

```
GraphPane myPane = null;
```

Виділення пам'яті під змінну даного типу буде проходити в функції подій форми **Form_Load():**

```
private void Form1_Load(object sender, EventArgs e)
    {          myPane = zedGraphControl1.GraphPane;       }
```

Сама побудова графіка знаходиться в подієвій функції клацання на кнопці ***Побудова***:

```
private void button2_Click(object sender, EventArgs e)
    {
       myPane = zedGraphControl1.GraphPane;
       myPane.CurveList.Clear();
       myPane.Title.Text = "Варіант " + Convert.ToString(vx) + " " +
                                  Convert.ToString(vy);
       myPane.XAxis.Title.Text = "Вісь X";
       myPane.YAxis.Title.Text = "Вісь Y";
       PointPairList list = new PointPairList();
//Заповнення списку
       LineItem myCurve=myPane.AddCurve("", list, Color.Black);
      myCurve.Symbol.IsVisible = false;
       zedGraphControl1.AxisChange();
       zedGraphControl1.Invalidate();
    }
```

Відкриття форми **Form2** для вибору умови відбувається при натисканні на кнопці Form1 - ***Вибір умови***.

```
private void button1_Click(object sender, EventArgs e)
    {   Form2 myf2 = new Form2();
       if (myf2.ShowDialog() == DialogResult.Cancel)
       {   vx = 0;  vy = 0;     }
       else
       {   myPane.CurveList.Clear();
           myPane.Title.Text = "Варіант " + Convert.ToString(vx) + " " +
Convert.ToString(vy);
          zedGraphControl1.Invalidate();
       }
    }
```

Розглянемо питання заповнення списку точок для кривої графіки. З курсу механіки Ви знаєте, що за умови може бути 10 варіантів для **x(t).** Залежно від вибору **x(t)** існує 30 варіантів для **y(t).** Усього 40 функцій.

Функції було оголошено у класі форми, згідно з формулами заданим у таблиці методички з механіки.

```
public double X0(double t)
{           return 2 - 3 * Math.Cos(Math.PI * t / 6);       }
public double X1(double t)
{           return 6 * Math.Cos(Math.PI * t / 6) - 3;       }
public double X2(double t)
{           return 4 * Math.Cos(Math.PI * t / 3);       }
public double X3(double t)
{           return  2-t;       }
public double X4(double t)
{           return 2 * t + 1;       }
public double X5(double t)
{           return  t  - 4;       }
public double X6(double t)
{           return 4-2*t;       }
public double X7(double t)
{           return 12 * Math.Sin(Math.PI * t / 6);       }
public double X8(double t)
{           return 4 - 6 * Math.Sin(Math.PI * t / 6);     }
public double X9(double t)
{           return 8 * Math.Sin(Math.PI * t / 6) - 2;     }
public double Y00(double t)
{           return 12 * Math.Sin(Math.PI * t / 6);       }
public double Y10(double t)
{           return -6 * Math.Cos(Math.PI * t / 3);       }
public double Y20(double t)
{           return -3 * Math.Pow(Math.Sin(Math.PI * t / 6), 2);       }
public double Y30(double t)
{           return 9 * Math.Sin(Math.PI * t / 3);       }
public double Y40(double t)
```

```
{           return 3 * Math.Cos(Math.PI * t / 6);       }
public double Y50(double t)
{           return 5 * Math.Sin(Math.PI * t / 6);       }
public double Y60(double t)
{           return 6 * Math.Pow(Math.Sin(Math.PI * t / 6), 2);       }
public double Y70(double t)
{           return -2 * Math.Sin(Math.PI * t / 3);       }
public double Y80(double t)
{           return 9* Math.Cos(Math.PI * t / 3);       }
public double Y90(double t)
{           return -8 * Math.Sin(Math.PI * t / 6);       }
public double Y01(double t)
{           return 2 * t * t + 2;       }
public double Y11(double t)
{           return 8 * Math.Sin(Math.PI * t / 4);       }
public double Y21(double t)
{           return (2 + t) * (2 + t);       }
public double Y31(double t)
{           return 2 * t * t * t + 4;       }
public double Y41(double t)
{           return 2 * Math.Cos(Math.PI * t / 4);       }
public double Y51(double t)
{           return 2 - 3 * t * t;       }
public double Y61(double t)
{           return 2 * Math.Sin(Math.PI * t / 4);       }
public double Y71(double t)
{           return  Math.Pow(t+1,3);       }
public double Y81(double t)
{           return 2 - t * t * t;       }
public double Y91(double t)
{           return 4 * Math.Cos(Math.PI * t / 4);       }
public double Y02(double t)
{           return 4 * Math.Cos(Math.PI * t / 6);       }
public double Y12(double t)
```

```
{           return 6 * Math.Pow(Math.Sin(Math.PI * t / 3), 2);        }
public double Y22(double t)
{           return 4 * Math.Cos(Math.PI * t / 3);        }
public duble Y32(double t)
{           return 10 * Math.Cos(Math.PI * t / 6);        }
public double Y42(double t)
{           return -4 * Math.Pow(Math.Cos(Math.PI * t / 6),2);        }
public double Y52(double t)
{           return -12 * Math.Cos(Math.PI * t / 3);        }
public double Y62(double t)
{           return -3 * Math.Cos(Math.PI * t / 6);        }
public double Y72(double t)
{           return -8 * Math.Cos(Math.PI * t / 3);        }
public double Y82(double t)
{           return 9 * Math.Cos(Math.PI * t / 6);        }
public double Y92(double t)
{           return -6 * Math.Cos(Math.PI * t / 3);        }
```

Імена функцій зберігатимемо у двох масивах рядків:

```
string[] masNameX;
string[,] masNameY;
```

Для завдання розмірів масивів використовуються змінні класи форми **Form1:**

```
int n = 3;
int m = 10;
```

Виділення пам'яті під дані масиви та їх ініціалізація знаходиться у конструкторі форми.

```
 masNameX = new string[m];
 masNameY = new string[m, n];
          for (int i = 0; i < m; i++)
          {    masNameX[i]="X"+Convert.ToString(i);
               for (int j=0;j<n;j++)
                 masNameY[i,j]="Y"+Convert.ToString(i)+Convert.ToString(j);     }
```

Для роботи з цими функціями будемо використовувати рефлексію. У світі **.NET** рефлексією (reflection) називається процес виявлення типів під час виконання. Багато класів, що підтримують рефлексію, входять до складу прикладного інтерфейсу .**NET Reflection API**, що відноситься до простору імен **System.Reflection**. Як і в будь-якому іншому просторі імен, **System.Reflection** містить набір взаємопов'язаних типів.

Для того, щоб безпосередньо звертатися до типів простору імен **Reflection** без імені посилання, необхідно написати команду виду:

```
using System.Reflection;
```

Для реалізації завдання необхідно оголосити об'єкт класу **MethodInfo**:

```
MethodInfo handler1;
```

MethodInfo – це абстрактний клас, який містить інформацію за певним методом. Щоб отримати інформацію, нам треба скористатися класом System.**Type**. Простір імен **System** підключено до проекту за замовчанням, тому ми можемо використовувати тип **Type** без вказівки імені посилання. Клас **System.Type** складає ядро підсистеми рефлексії, оскільки він інкапсулює тип даних. Він містить багато властивостей і методів, якими можна користуватися для отримання інформації про тип даних під час виконання. Клас **Type** є похідним від абстрактного класу **System.Reflection.MemberInfo**.

Щоб керувати типом та отримувати всю інформацію про нього, нам треба спершу отримати цей тип. Оголошуємо змінну типу **Type** і надаємо їй тип об'єкта, який створив метод. У нас всі функції знаходяться у класі форми:

```
Type tt = typeof(Form1);
```

Потім за допомогою методу класу **Type** із сигнатурою

public System.Reflection.MethodInfo GetMethod (string name);

Отримуємо об'єкт класу MethodInfo, що представляє відкритий метод із заданим ім'ям, якщо такий метод є, і **null,** якщо такого методу немає. Параметр name типу **string** – рядок, що містить ім'я відкритого методу.

У нашому випадку оператор має вигляд:

```
handler1 = tt.GetMethod(masNameX[vx]);
```

У класі MethodInfo є перевантажений метод Invoke() із трьома сигнатурами. Ми будемо використовувати прототип

```
public object Invoke (object obj, object[] parameters);
```

Вхідний параметр **obj** має тип **object** – це екземпляр, який створив цей метод. Вхідний параметр parameters типу **object**[] – це список аргументів для методу, що викликається. Це масив об'єктів, кількість, порядок і тип яких повинні відповідати списку параметрів методу або конструктора, що викликається. За відсутності параметрів значення параметрів повинно бути рівним null.

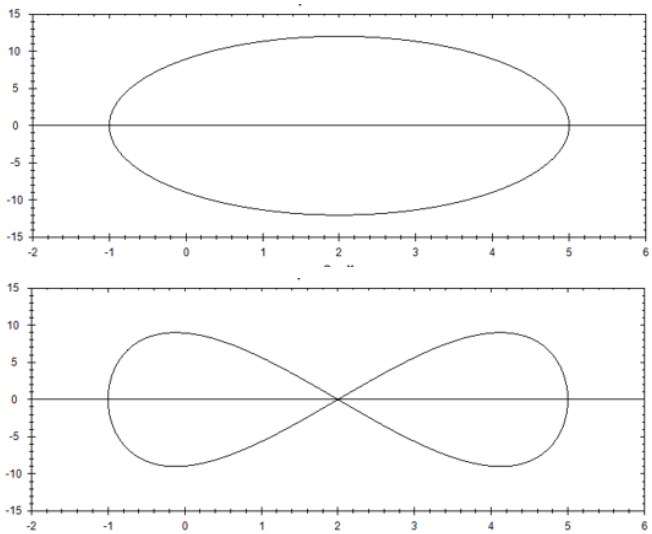
У нашому випадку обидві функції **x(t)** та **y(t)** приймають один вхідний параметр – час, тому оператор виклику функції має вигляд:

```
handler1.Invoke(this, new object[] { t })
```

Для нашої програми заповнення списку точок для кривої має вигляд:

```
int st=0;
        if (vx >= 0 && vx <= 2)      st = 0;
        if (vx >= 3 && vx <= 6)      st = 1;
        if (vx >= 7 && vx <= 9)      st = 2;
        handler1 = tt.GetMethod(masNameX[vx]);
        handler2 = tt.GetMethod(masNameY[vy, st]);
        for (double t = 0; t < 20; t = t + 0.1)
        list.Add(Convert.ToDouble(handler1.Invoke(this, new object[] { t})),
Convert.ToDouble(handler2.Invoke(this, new object[] { t })));
```

Результат



12. ТЕХНОЛОГІЯ DRAG&DROP

Технологія **Drag&Drop** має на увазі використання комп'ютерної миші і в перекладі з англійської означає буквально: тягни-і-кидай. Дія виконується шляхом оперування видимими на екрані комп'ютера об'єктами за допомогою маніпулятора миша. Суть дій полягає у переміщенні віртуальних об'єктів щодо один одного і таким чином у виконанні будь-яких дій у застосунку, або при взаємодії кількох додатків.

У **C#** реалізовано власний інтерфейс **Drag&Drop**, що дозволяє компонентам обмінюватися даними шляхом "перетягування" їх мишею. Всі властивості, методи та події, пов'язані з процесом перетягування, є спільними для всіх компонентів.

Клас **Control** має події **DragDrop**, **DragEnter**, **DragLeave** та **DragOver**. Подія **DragDrop** відбувається, коли користувач завершує операцію перетягування шляхом перетягування об'єкта до елемента керування та вивільнення об'єкта під час відпускання кнопки миші. Обробка події **DragEnter** відбувається, коли користувач переміщає вказівник миші до елемента керування під час перетягування об'єкта за допомогою миші. **DragLeave** відбувається при переміщенні курсору за межі елемента управління або при скасуванні поточної операції перетягування. Подія **DragOver** відбувається, коли користувач встановлює покажчик миші елемент управління при перетягуванні об'єкта з допомогою миші.

Обробники подій **DragDrop**, **DragEnter** і **DragOver** одержують аргумент типу **DragEventArgs**, що містить дані, що стосуються цієї події. Тип **DragEventArgs** надає такі члени:

1. **AllowedEffect** отримує значення, що показує, які операції перетягування дозволені вихідним елементом управління (джерелом) події перетягування (тип: DragDropEffects).
2. **Effect** отримує або задає ефект перетягування приймача операції перетягування (тип: DragDropEffects).
   Одне зі значень DragDropEffects:
   - **None -** Цільовий об'єкт не приймає дані.
   - **Copy -** Дані з джерела перетягування копіюються в цільовий об'єкт.
   - **Move -** Дані з джерела перетягування переміщуються в цільовий об'єкт.

3. **DragDropEffects** дозволяє відображати різні покажчики миші під час операції перетягування. Наприклад, можна встановити, щоб під час операції перетягування Copy відображався символ "плюс", під час операції перетягування Move відображалася "стрілка", під час операції перетягування None — червоне коло, перекреслене червоною лінією.
4. **KeyState** отримує поточний стан клавіш SHIFT, CTRL та ALT, а також стан кнопок миші (тип: Int32). Значення, що використовуються для заданої події:
   1 - Ліва кнопка миші.
   2 - Права кнопка миші.
   4 - Клавіша SHIFT.
   8 - Клавіша CTRL.
   16 - Середня кнопка миші.
   32 - Клавіша ALT.
5. **X** отримує координату по осі **X** для курсору миші (в екранних координатах).
6. **Y** отримує координату курсору миші по осі **Y** (в екранних координатах)**.**
7. **Data** отримує дані, пов'язані з цією подією. Має тип **IDataObject**. Щоб отримати ці дані, необхідно використовувати метод **GetData().** Має три перевантажені прототипи.

Ініціюється операція перетягування шляхом виклику методу **DoDragDrop()** класу **Control**, що визначає допустимий ефект цієї операції. Має прототип:

public DragDropEffects DoDragDrop(Object data, DragDropEffects allowedEffects);

Параметри

data типу **Object** задає дані, що перетягуються.

Застосовані ефекти типу **DragDropEffects** мають одне з значень перерахування. Значення, що повертається, має тип **DragDropEffects** і представляє кінцевий результат виконання операції перетягування.

Метод **DoDragDrop()** визначає елемент керування за поточного розташування курсору. Потім він перевіряє, чи є елемент управління допустимим для кінцевого розташування об'єкта перетягування. Якщо елемент керування є допустимим для кінцевого розташування об'єкта перетягування, викликається

подія **GiveFeedback** із вказаним ефектом перетягування. Для того, щоб компонент зміг прийняти інформацію, потрібна його властивість **AllowDrop** в **true**. Значення за промовчанням є **false**.
Подієва функція **GiveFeedback** має прототип:
private void компонент_GiveFeedback(object sender, GiveFeedbackEventArgs e);

Подія **GiveFeedback** дозволяє джерелу події перетягування змінювати зовнішній вигляд курсору миші, забезпечуючи користувачу візуальний зворотний зв'язок під час операції перетягування за рахунок вхідного параметра типу **GiveFeedbackEventArgs**. Це клас, який має властивості:

- **Effects** - повертає значення, що визначає наслідки операції перетягування. Член переказу **DragDropEffects.**
- **UseDefaultCursors** - отримує або задає логічне значення, що визначає необхідність використання поведінки відкликання курсору за замовчуванням для зазначеної операції перетягування. Значення **true** для використання поведінки відкликання курсору за замовчуванням; інакше - **false**. За замовчуванням, курсор миші змінюється для демонстрації дозволених дій під час операції перетягування.

Потім відстежуються зміни позиції курсору миші, стану клавіатури та кнопки миші.

- Якщо користувач переміщує курсор миші за межі вікна, відбувається подія **DragLeave**.
- Якщо курсор миші переміщується на інший елемент керування, для цього елемента викликається подія **DragEnter**.
- При переміщенні миші в межах одного елемента керування відбувається подія **DragOver**.

При зміні стану клавіатури або кнопки миші відбувається подія **QueryContinueDrag**, яка визначає, чи слід продовжити або завершити перетягування або скасувати операцію на підставі значення властивості **Action** аргументу **QueryContinueDragEventArgs** цієї події.
Має прототип
private void компонент_QueryContinueDrag(object sender,
QueryContinueDragEventArgs e);

Властивість **Action** має тип **DragAction** – перерахування. Має такі значення:

- **Continue -** операцію буде продовжено.
- **Drop -** операцію буде зупинено після перетягування.
- **Cancel -** операція скасовується з видачею повідомлення про неможливість перетягування.

За замовчанням подією **QueryContinueDrag** задається для властивості **Action** значення **DragAction.Cancel**, якщо була натиснута клавіша ESC, і задається для **Action** значення **DragAction.Drop**, якщо натискається ліва, середня або права кнопка миші.

Можливі дії:

- Якщо значення об'єкта **DragAction** дорівнює **Continue**, виникає подія **DragOver**, щоб продовжити операцію, і подія **GiveFeedback** з новим ефектом, що дозволяє задати відповідний візуальний зворотний зв'язок.
- Якщо значення **DragAction** дорівнює **Drop**, значення ефекту перетягування повертається джерелу, завдяки чому вихідна програма може виконати відповідну операцію з вихідними даними (наприклад, вирізати дані, якщо це була операція переміщення).
- Якщо значення **DragAction** дорівнює **Cancel**, відбувається подія **DragLeave**.

Функція **QueryContinueDrag** має прототип:

```
private void компонент_QueryContinueDrag(object sender,
QueryContinueDragEventArgs e);
```

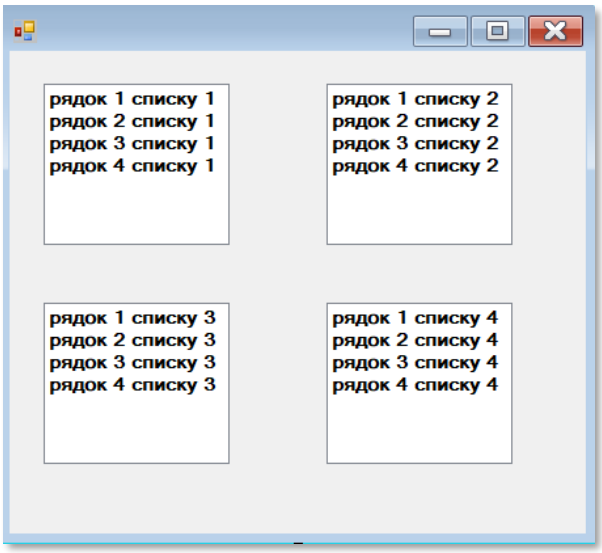
Тип **QueryContinueDragEventArgs** має такі властивості:

- **Action -** Отримує або визначає поточний статус пов'язаної операції перетягування. Член перерахування **DragAction** (можливі значення:
  Continue - Операція буде продовжена;
  Drop – операція буде зупинена після перетягування;
  Cancel – операція скасовується з видачею повідомлення про неможливість перетягування), що вказує на поточний статус пов'язаної операції перетягування.
- **EscapePressed -** Повертає логічне значення, яке показує, чи натиснуто клавішу ESC. true, якщо клавішу ESC було натиснуто; в іншому випадку **false**.

- **KeyStates** - Повертає перелік прапорів, що вказує на поточний стан клавіш SHIFT, CTRL і ALT, а також стан кнопок миші. Один або кілька членів переліку прапорів **DragDropKeyStates**. Можливі значення: None – Клавіші SHIFT, ALT, CTRL та кнопки миші не натиснуті; **LeftMouseButton** – натиснута ліва кнопка миші; **RightMouseButton** – натиснута права кнопка миші; **ShiftKey** – натиснута клавіша SHIFT; **ControlKey** – клавіша CTRL натиснута; **MiddleMouseButton** – натиснута середня кнопка миші; **AltKey** – клавіша ALT натиснута.

Коли процес перетягування розпочнеться, звичайний вид курсору змінюється. Поки він переміщується над формою або компонентами, які не можуть прийняти інформацію, він зазвичай має тип **crNoDrop**. Якщо ж він переміщується над компонентом, готовим прийняти інформацію з перетягуваного об'єкта, то набуває вигляду, який визначається властивістю перетягуваного об'єкта, що перетягується, **DragCorsor**. За замовчуванням ця властивість дорівнює **crDrag.**

Розглянемо приклад програми, яка використовує технологію **Drag&Drop**. На етапі візуального програмування форма має вигляд:



Можливе перетягування рядків з одного списку в інший, причому з трьох списків відбувається копіювання, а з одного – переміщення. У всіх списків встановлено властивість **AllowDrop** в true.

Процес перетягування почнемо, якщо натиснуто ліву кнопку миші і мишу

зсунуто на відстань більше 5 пікселів. Для цього використовуються три подієві функції списків **MouseDown, MouseUp** та **MouseMove.** Напишемо обробники подій для компонента listBox1 універсальним чином, а до інших списків дані подієві функції підключимо за допомогою ***вікна властивостей***. Коли натиснута ліва кнопка миші, необхідно запам'ятати координати миші, щоб відстежувати величини зміщення миші в змінних класу форми:

```
int x = 0;
int y = 0;
```

А також використовується прапор

```
13      public partial class Form1 : Form
14      {
15          int x = 0;
16          int y = 0;
17
18          bool left = false;
19          ListBox istok = null;
20          int nom = -1;
            1 reference
21          public Form1()
22          {
23              InitializeComponent();
24          }
            4 references
25          private void listBox1_MouseDown(object sender, MouseEventArgs e)
26          {
27              if (e.Button == MouseButtons.Left)
28              {
29                  left = true;
30                  x = e.X;
31                  y = e.Y;
32              }
33          }
            4 references
34          private void listBox1_MouseUp(object sender, MouseEventArgs e)
35          {
36              if (e.Button == MouseButtons.Left)
37              {  left = false;   }
38          }
            4 references
39          private void listBox1_MouseMove(object sender, MouseEventArgs e)
40          {
41              ListBox my = sender as ListBox;
42              istok = my;
43              nom = my.SelectedIndex;
44              if (Math.Sqrt((e.X - x) * (e.X - x) + (e.Y - y) * (e.Y - y)) > 5 && left)
45              {
46                  if (my.SelectedItem == null)
47                      return;
48                  if (sender == listBox3)
49                      my.DoDragDrop(my.SelectedItem, DragDropEffects.Move);
50                  else
51                      my.DoDragDrop(my.SelectedItem, DragDropEffects.Copy);
52                  left = false;
53              }
54
55          }
```

Примітка 1. *Змінна ту використовує для зручності написання програмного коду.*

Зауваження 2. *У функції подій використовуються змінні класу форми*

ListBox istok = null;

int nom = -1; *щоб запам'ятати, який список є джерелом інформації.*

При перетягуванні рядка зі списку до списку вибір рядка у джерелі скасовується, а у разі третього списку потрібно буде видаляти інформацію.

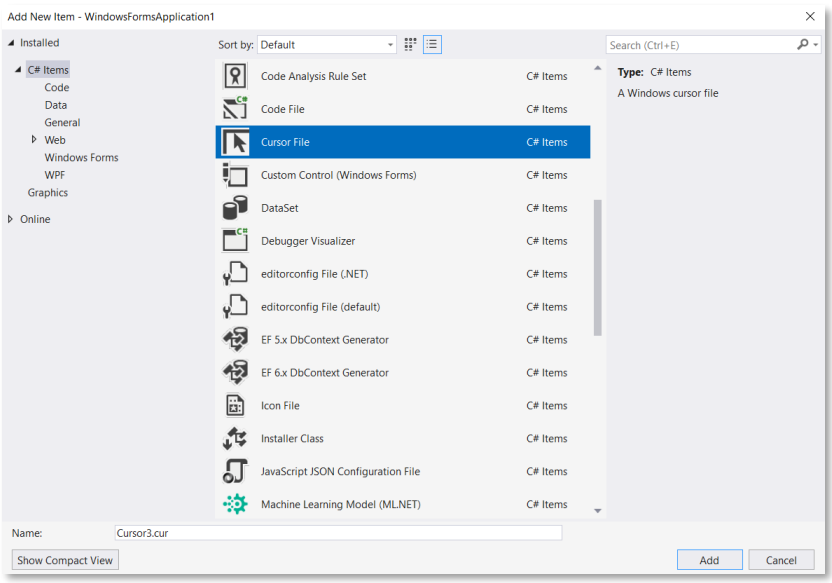
Процедура прийому інформації від об'єкта, що перетягується, записується в обробнику події **OnDragDrop** приймаючого компонента. Подія **OnDragDrop** настає в момент відпускання компонента, що перетягується над цим компонентом. В обробнику події треба описати, що в цей момент має статися.

```
57      private void listBox1_DragDrop(object sender, DragEventArgs e)
58      {
59          Object item = e.Data.GetData(DataFormats.Text);
60          if (istok == sender)
61              return;
62          ((ListBox)sender).Items.Add(item);
63          if (e.AllowedEffect == DragDropEffects.Move)
64              istok.Items.RemoveAt(nom);
65          istok.SelectedIndex = -1;
66
67      }
```

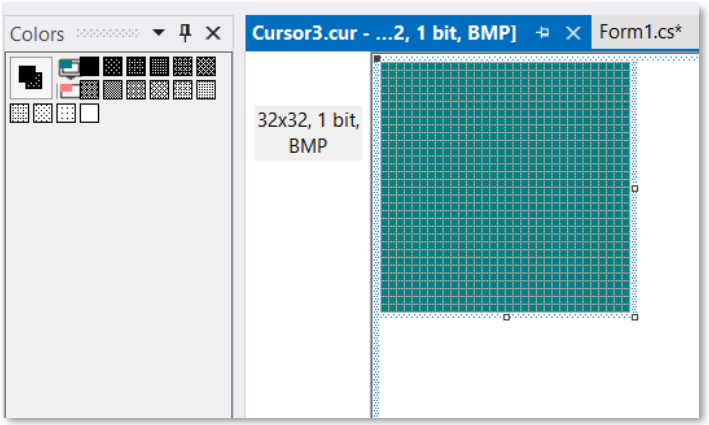
У процесі перетягування компоненти, над якими переміщається курсор, можуть інформувати про готовність прийняти інформацію від об'єкта, що перетягується. Для цього в компоненті повинен бути передбачений обробник події **OnDragOver,** що настає при переміщенні над цим компонентом курсору, що перетягує об'єкта.

```
69      private void listBox1_DragOver(object sender, DragEventArgs e)
70      {
71          if (e.AllowedEffect == DragDropEffects.Copy)
72              e.Effect = DragDropEffects.Copy;
73          else
74              e.Effect = DragDropEffects.Move;
75      }
```

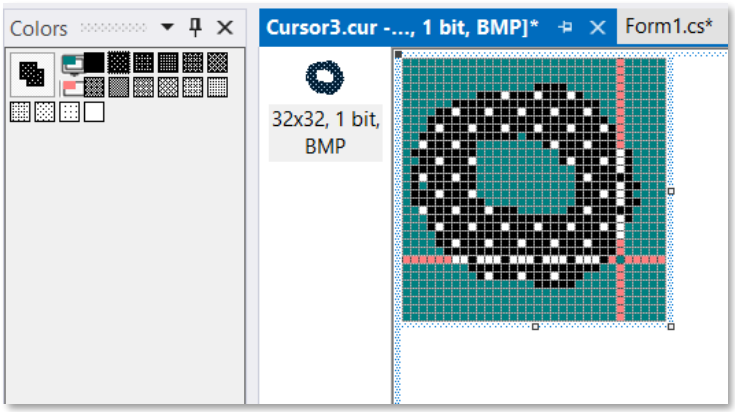
Було розказано, що є подія **GiveFeedback**, яка дозволяє змінювати вигляд курсору. Для цього потрібно створити відповідні файли з розширенням **cur**. Це можна зробити, використовуючи команду головного меню **Project| Add new Items** і у вікні вибрати



У редакторі коду відкриється відповідний файл виду:



На малюнку видно ще додаткову панель інструментів, яка завжди автоматично відкривається. Зробити її видимою над командою: **View | Toolbars | Image Editor**. Ви створюєте свій вид курсору, який можна задавати як особливий вид під час перетягування. У нашому прикладі курсор має різний вигляд: для копіювання та переміщення.



```
private void listBox1_GiveFeedback(object sender, GiveFeedbackEventArgs e)
    {
       e.UseDefaultCursors = false;
       if (e.Effect  == DragDropEffects.Copy)
          Cursor.Current = new Cursor("Cursor1.cur");
       else
          Cursor.Current = new Cursor("Cursor2.cur");      }
```

Зауваження. *Ця функція коректно працює за умови, що файли курсору знаходяться у папці з виконуваним файлом. Якщо файли курсору зберігаються в папці проекту, де знаходяться файли форми, то необхідно використовувати відносний шлях:*

```
77      private void listBox1_GiveFeedback(object sender, GiveFeedbackEventArgs e)
78      {
79          e.UseDefaultCursors = false;
80          if (e.Effect == DragDropEffects.Copy)
81              Cursor.Current = new Cursor(@"..\..\Cursor1.cur");
82          else
83              Cursor.Current = new Cursor(@"..\..\Cursor2.cur");
84      }
```

Процес **Drag&Drop** можна організувати не лише всередині програми. Додамо код, який дозволяє при перетягуванні файлу зображення на форму завантажувати це зображення на неї.

```
86      private void Form1_DragDrop(object sender, DragEventArgs e)
87      {
88          try
89          {
90              string[] files = (string[])e.Data.GetData(DataFormats.FileDrop);
91              BackgroundImage = new Bitmap(files[0]);
92          }
93          catch
94          { }
95      }
96
        1 reference
97      private void Form1_DragOver(object sender, DragEventArgs e)
98      {
99          e.Effect = DragDropEffects.Copy;
100     }
```

Примітка 1. *Для форми встановлено властивість* ***AllowDrop*** *у true.*

13. ДЕЛЕГАТИ ТА ПОДІЇ

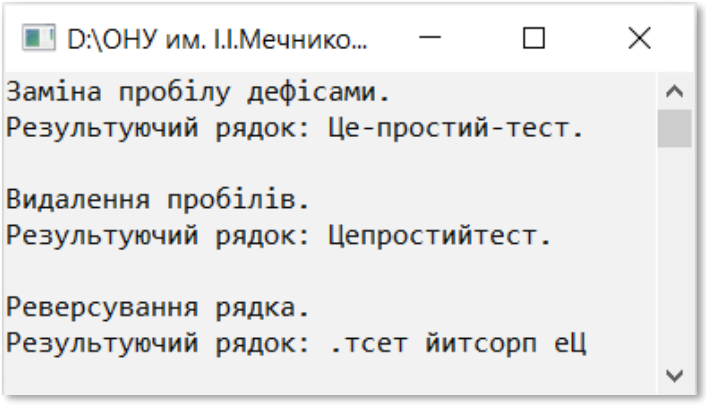
13.1 Інкапсулювання методу за допомогою делегата

Делегат надає можливість інкапсулювати метод, а подія - це свого роду повідомлення про те, що мала місце певна дія. Делегати та події пов'язані між собою, оскільки подія створюється на основі делегату. Ці засоби розширюють діапазон завдань програмування, до яких можна застосувати мову C#.

Приклад

```
9       delegate string strMod(string stx);
        0 references
10      class DelegateTest
11      {
            1 reference
12          static string replaceSpaces(string a) // Метод замінює пробіли дефісами.
13          {
14              Console.WriteLine("Заміна пробілу дефісами.");
15              return a.Replace(' ', '-');
16          }
            1 reference
17          static string removeSpaces(string a)  // Метод видаляє пробіли.
18          {
19              string temp = "";  int i;
20              Console.WriteLine("Видалення пробілів.");
21              for (i = 0; i < a.Length; i++)
22                  if (a[i] != ' ') temp += a[i];
23              return temp;
24          }
            1 reference
25          static string reverse(string a)    //Метод реверсує рядок
26          {
27              string temp = "";  int i;
28              Console.WriteLine("Реверсування рядка.");
29              for (i = a.Length - 1; i >= 0; i--)
30                  temp += a[i];
31              return temp;
32          }
            0 references
33          static void Main(string[] args)
34          {
35              // Створення делегату.
36              strMod strOp = new strMod(replaceSpaces);
37              string str;
38              // Викликаємо методи за допомогою делегата.
39              str = strOp("Це простий тест.");
40              Console.WriteLine("Результуючий рядок: " + str);  Console.WriteLine();
41              strOp = new strMod(removeSpaces);
42              str = strOp("Це простий тест.");
43              Console.WriteLine("Результуючий рядок: " + str);  Console.WriteLine();
44              strOp = new strMod(reverse);
45              str = strOp("Це простий тест.");
46              Console.WriteLine("Результуючий рядок: " + str);
47          }
48      }
```

Результат:



Примітка 1. У методі **Main()** створюється посилання типу **strMod** з ім'ям **strOp**, і йому присвоюється посилання на метод **replaceSpaces()**. Потім метод **replaceSpaces()** викликається за допомогою екземпляра делегата з ім’ям **strOp**. Потім екземпляру делегата **strOp** присвоюється посилання на метод **removeSpaces()**, після чого **strOp** викликається знову. Цього разу викликається метод **removeSpaces()**. Нарешті, екземпляру делегата **strOp** присвоюється посилання на метод **reverse()**, і **strOp** викликається ще раз. Виклик екземпляра делегата **strOp** трансформується у звернення до методу, на який посилається **strOp** під час виклику. Таким чином, рішення про метод, що викликається приймається під час виконання програми, а не в період компіляції.

Незважаючи на те, що у попередньому прикладі використовуються статичні методи, делегат може також посилатися на методи екземплярів класу. Однак він повинен використовувати при цьому об'єктне посилання.

Наприклад, як може виглядати попередня програма, переписана з метою інкапсуляції операцій над рядками всередині класу **StringOps**.

Приклад 2

```
9         delegate string strMod(string stx);
          2 references
10        class StringOps
11        {
              1 reference
12            public string replaceSpaces(string a)  // Метод замінює пробіли дефісами.
13            {
14                Console.WriteLine("Заміна пробілів дефісами.");
15                return a.Replace(' ', '-');
16            }
              1 reference
17            public string removeSpaces(string a) // Метод видаляє пробіли.
18            {
19                string temp = "";  int i;
20                Console.WriteLine("Видалення пробілів.");
21                for (i = 0; i < a.Length; i++)
22                    if (a[i] != ' ') temp += a[i];
23                return temp;
24            }
              1 reference
25            public string reverse(string a)    // Метод реверсує рядок,
26            {
27                string temp = "";  int i;
28                Console.WriteLine("Реверсування рядка.");
29                for (i = a.Length - 1; i >= 0; i--)
30                    temp += a[i];
31                return temp;
32            }
33        }
          0 references
34        internal class Program
35        {
              0 references
36            static void Main(string[] args)
37                {   StringOps so = new StringOps(); // Створюємо екземпляр класу StringOps.
38                    strMod strOp = new strMod(so.replaceSpaces);// Створення делегату.
39                    string str;
40                    // Викликаємо методи за допомогою делегата.
41                    str = strOp("Це простий тест.");
42                    Console.WriteLine("Результуючий рядок: " + str);   Console.WriteLine();
43                    strOp = new strMod(so.removeSpaces);
44                    str = strOp("Це простий тест.");
45                    Console.WriteLine("Результуючий рядок: " + str);   Console.WriteLine();
46                    strOp = new strMod(so.reverse);
47                    str = strOp("Це простий тест.");
48                    Console.WriteLine("Результуючий рядок: " + str);
49            }
50        }
```

Результат такий самий.

13.2 Багатоадресна передача

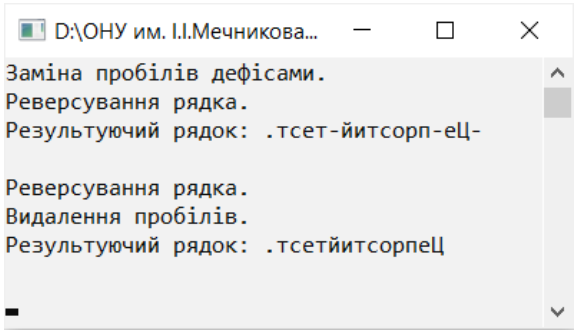
Однією з найцікавіших можливостей делегата є підтримка багатоадресатної передачі (**multicasting**). Багатоадресатна передача – це здатність створювати список викликів (або ланцюжок викликів) методів, які повинні автоматично викликатись при виклику делегата. Такий ланцюжок створити неважко. Достатньо створити екземпляр делегата, а потім для

додавання методів до цього ланцюжка використовувати оператор "+=". Для видалення методу з ланцюжка використовується оператор "-=". (Можна також для додавання та видалення методів використовувати окремо оператори "+", "-" та "=", але частіше застосовуються складові оператори "+=" та "-=".) Делегат з багатоадресатною передачею має одне обмеження: він повинен повертати тип **void**.

Приклад 3. Розглянемо наступний приклад багатоадресатної передачі. Це перероблений варіант попередніх прикладів, в якому тип **string** для значень, що повертаються методами обробки рядків, замінений типом **void**, а для повернення модифікованих рядків використовується **ref**-параметр.

```
9       delegate void strMod(ref string str);
        0 references
10      class StringOps
11      {
            1 reference
12          static void replaceSpaces(ref string a)  // Метод замінює пробіли дефісами.
13          {
14              Console.WriteLine("Заміна пробілів дефісами.");  a = a.Replace(' ', '-');
15          }
            1 reference
16          static void removeSpaces(ref string a) // Метод видаляє пробіли.
17          {
18              string temp = ""; int i;
19              Console.WriteLine("Видалення пробілів.");
20              for (i = 0; i < a.Length; i++)
21                  if (a[i] != ' ') temp += a[i];
22              a = temp;
23          }
            1 reference
24          static void reverse(ref string a)    // Метод реверсує рядок,
25          {
26              string temp = ""; int i;
27              Console.WriteLine("Реверсування рядка.");
28              for (i = a.Length - 1; i >= 0; i--)
29                  temp += a[i];
30              a = temp;
31          }
                0 references
32          static void Main(string[] args)
33          {   // Створюємо екземпляри делегатів.
34              strMod strOp;
35              strMod replaceSp = new strMod(replaceSpaces);
36              strMod removeSp = new strMod(removeSpaces);
37              strMod reverseStr = new strMod(reverse);
38              string str = " Це простий тест.";
39              // Організація багатоадресатної передачі.
40              strOp = replaceSp;
41              strOp += reverseStr;
42              // Виклик делегата з багатоадресною передачею.
43              strOp(ref str);
44              Console.WriteLine("Результуючий рядок: " + str); Console.WriteLine();
45              // Видаляємо метод заміни прогалин та додаємо метод їх видалення.
46              strOp -= replaceSp;
47              strOp += removeSp;
48              str = " Це простий тест."; // Відновлення вихідного рядка.
49              // Виклик делегата з багатоадресною передачею.
50              strOp(ref str);
51              Console.WriteLine("Результуючий рядок: " + str); Console.WriteLine();
52          }
```

Результат:



Зауваження. У методі **Main()** створюються чотири екземпляри делегата. Перший, **strOp**, має **null**-значення. Три інших посилаються на методи модифікації рядків. Потім організується делегат для багатоадресатної передачі, який викликає методи **removeSpaces()** і **reverse()**. Це досягається завдяки наступним рядкам програми:

```
strOp = replaceSp;
strOp += reverseStr;
```

Спочатку делегату **strOp** надається посилання **replaceSp**. Потім, за допомогою оператора "+=", до ланцюжка викликів додається посилання **reverseStr**. При виклику делегата **strOp** викликаються обидва методу, замінюючи пробіли дефісами і реверсуючи рядок. Потім під час виконання рядка програми:

```
strOp -= replaceSp;
```

з ланцюжка викликів видаляється посилання **replaceSp**, а за допомогою рядка:

```
strOp += removeSp;
```

до ланцюжка викликів додається посилання **removeSp**.

Потім делегат **strOp** викликається знову. На цей раз з вихідного рядка видаляються пробіли, після чого він реверсується.

Ланцюжки викликів, організовані за допомогою делегата, - потужний механізм, який дозволяє визначати набір методів, що виконуються "єдиним блоком". Ланцюжки делегатів мають особливе значення для подій.

13.3 Події

На основі делегатів побудовано важливий засіб C#: подія (**event**). Подія – це по суті автоматичне повідомлення про виконання певної дії. Події працюють в такий спосіб. Об'єкт, якому потрібна інформація про певну подію, реєструє обробник для цієї події. Коли подія відбувається, викликаються всі

зареєстровані обробники. Обробники подій є делегатами.

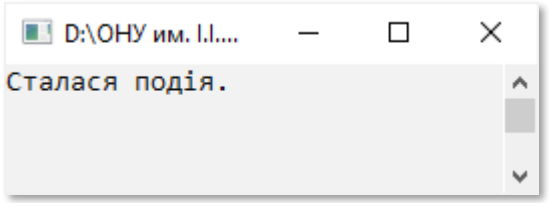
Події – це члени класу, які оголошуються за допомогою ключового слова **event.** Найбільш поширена форма оголошення події має такий вигляд:
event подієвий_делегат об'єкт;

Тут елемент ***подієвий_делегат*** означає ім'я делегата, що використовується для підтримки події, що оголошується, а елемент ***об'єкт*** — це ім'я створюваного подієвого об'єкта.

Приклад 4

```
0 references
class Program
{
    // Оголошуємо делегат для події
    delegate void MyEventHandler();
    // Оголошуємо клас події
    2 references
    class MyEvent
    {
        public event MyEventHandler SomeEvent;
        // Цей метод викликається для створення події
        1 reference
        public void OnSomeEvent()
        {
            if (SomeEvent != null)
                SomeEvent();
        }
    }
    0 references
    class EventDemo
    {
        // Обробник події
        1 reference
        static void handler()
        {
            Console.WriteLine("Сталася подія.");
        }
        0 references
        static void Main(string[] args)
        {
            MyEvent evt = new MyEvent();
            // Додаємо метод handler() до списку події
            evt.SomeEvent += new MyEventHandler(handler);
            // Генеруємо подію
            evt.OnSomeEvent();
        }
    }
}
```

Результат:



Програма починається з такого оголошення делегата для обробника події:

```
delegate void MyEventHandler();
```

Усі події активізуються за допомогою делегата. Отже, подієвий делегат визначає сигнатуру події. У цьому випадку параметри відсутні, однак подієві параметри дозволяються. Оскільки події зазвичай призначені для багатоадресатної передачі, вони мають повертати значення **void**.

Потім створюється клас події **MyEvent**. При виконанні наступного рядка коду, що належить цьому класу, оголошується подійний об'єкт **SomeEvent**:

```
public event MyEventHandler SomeEvent;
```

Зверніть увагу на синтаксис. Саме так оголошуються події всіх типів.

Крім того, всередині класу **MyEvent** оголошується метод **OnSomeEvent()**, який у цій програмі викликається сигналізувати про подію. (Іншими словами, цей метод викликається, коли відбувається подія.) Як показано в наступному фрагменті коду, він викликає обробник події за допомогою делегата **SomeEvent**.

```
if (SomeEvent != null)
    SomeEvent();
```

Зверніть увагу на те, що обробник події викликається тільки в тому випадку, якщо делегат **SomeEvent** не дорівнює **null**-значенню. Оскільки інші частини програми, щоб отримати повідомлення про подію, повинні зареєструватися, можна зробити так, щоб метод **OnSomeEvent()** був викликаний до реєстрації будь-якого обробника події. Щоб запобігти виклику **null**-об'єкта, подієвий делегат необхідно протестувати і переконатися в тому, що він не дорівнює **null** значенням.

Всередині класу **EventDemo** створюється обробник події handler(). У цьому прикладі обробник події просто відображає повідомлення, але зрозуміло, що інші обробники можуть виконувати більш корисні дії. Як показано в наступному фрагменті коду, у методі **Main()** створюється об'єкт класу

MyEvent, а метод **handler()** реєструється як обробник цієї події.

```
MyEvent evt = new MyEvent();
// Додаємо метод handler() до списку події
evt.SomeEvent += new MyEventHandler(handler);
```

Зверніть увагу, що обробник додається до списку з використанням складового оператора "+=". Слід зазначити, що події підтримують лише оператори "+=" та "-=". У нашому прикладі метод **handler()** є статичним, але в загальному випадку обробники подій можуть бути методами екземплярів класів. Зрештою, при виконанні

```
evt.OnSomeEvent();
```

"відбувається" подія, про яку ми так багато говорили. Під час виклику методу **OnSomeEvent()** викликаються всі зареєстровані обробники подій. У нашому прикладі такий обробник лише один.

13.4 Рекомендації щодо обробки подій у середовищі .NET Framework.

C# дозволяє програмісту створювати події будь-якого типу. Однак з метою компонентної сумісності із середовищем **.NET Framework** необхідно дотримуватися рекомендацій, підготовлених **Microsoft** спеціально для цього. Центральне місце у цих рекомендаціях посідає вимога того, щоб обробники подій мали два параметри. Перший має бути посиланням на об'єкт, який генеруватиме подію. Другий повинен мати тип **EventArgs** і містити іншу інформацію, необхідну обробнику. Таким чином, **.NET**-сумісні обробники подій повинні мати таку загальну форму запису:

```
void handler(object source, EventArgs arg)
{
//..
}
```

Зазвичай параметр ***source*** передається кодом, що викликає. Параметр типу ***EventArgs*** містить додаткову інформацію, яку можна проігнорувати у разі непотреби.

Клас **EventArgs** не містить полів, які використовуються при передачі додаткових даних обробнику; він використовується як базовий клас, з якого можна виводити клас, що містить необхідні поля. Але оскільки багато обробників обходяться без додаткових даних, до класу **EventArgs** включено статичне поле Empty, яке задає об'єкт, який містить ніяких даних.

Приклад 5.

```
namespace ConsoleApplication1
{
 // Створюємо клас, похідний від класу EventArgs.
class MyEventArgs : EventArgs {
public int eventnum;}
// Оголошуємо делегат для події.
delegate void MyEventHandler(object source,MyEventArgs arg);
// Оголошуємо клас події,
   class MyEvent
   {   static int count = 0;
      public event MyEventHandler SomeEvent;
      // Цей метод генерує SomeEvent-подію.
      public void OnSomeEvent()
      {
         MyEventArgs arg = new MyEventArgs();
         if (SomeEvent != null)
         {
            arg.eventnum = count++;
            SomeEvent(this, arg);
         }
      }
   }
   class X
   {
      public void handler(object source, MyEventArgs arg)
      {
         Console.WriteLine("Подія " + arg.eventnum + " отримано об'єктом X.");
         Console.WriteLine("Джерелом є клас " + source + ".");
         Console.WriteLine();
      }
   }
```

```
    class Y
    {
        public void handler(object source, MyEventArgs arg)
        {
            Console.WriteLine("Подія " + arg.eventnum +" отримано об'єктом Y.");
            Console.WriteLine("Джерелом є клас " + source + ".");
            Console.WriteLine();
        }
    }
    class EventDerno
    {
        public static void Main()
        {
            X ob1 = new X();
            Y ob2 = new Y();
            MyEvent evt = new MyEvent();
            // Додаємо обробник handler() до списку подій,
            evt.SomeEvent += new MyEventHandler(ob1.handler);
            evt.SomeEvent += new MyEventHandler(ob2.handler);
            // Генеруємо подію,
            evt.OnSomeEvent();
            evt.OnSomeEvent();
        }
    }
}
```

У цьому прикладі клас **MyEventArgs** виводиться із класу **EventArgs**. У класі **MyEventArgs** додано лише одне "власне" поле – **eventnum**. Відповідно до вимог **.NET Framework** делегат для обробника подій **MyEventHandler** тепер приймає два параметри. Як пояснювалося вище, перший з них є об'єктним посиланням на генератор подій, а другий - посилання на клас EventArgs або похідний від класу **EventArgs**. У цьому випадку тут використовується посилання на об'єкт типу **MyEventArgs**.

13.5 Використання вбудованого делегата EventHandler.

Для багатьох подій параметр **EventArgs** не використовується. Для спрощення процесу створення коду в таких ситуаціях середовище **.NET Framework** включає вбудований тип делегата, який називається **EventHandler**. Його можна використовувати для оголошення обробників подій, яким не потрібна додаткова інформація. Розглянемо приклад використання типу **EventHandler.**

Приклад 6

```
namespace ConsoleApplication1
{ // Оголошуємо клас події,
   class MyEvent
   {  // Оголошення використовує делегат
      public event EventHandler SomeEvent;
      //EventHandler.
      // Цей метод викликається для створення SomeEvent-подія.
      public void OnSomeEvent()
      {
         if (SomeEvent != null)       SomeEvent(this, EventArgs.Empty);
      }
   }
   class EventDemo
   {
      static void handler(object source, EventArgs arg)
      {
         Console.WriteLine("Подія сталася.");
         Console.WriteLine("Джерелом є клас " + source + " . ");
      }
      public static void Main()
      {
         MyEvent evt = new MyEvent();
         // Додаємо обробник handler() до списку подій,
         evt.SomeEvent += new EventHandler(handler);
         // Генеруємо подію,
         evt.OnSomeEvent();
      } }   }
```

14. СТВОРЕННЯ КЕРУЮЧИХ ЕЛЕМЕНТІВ

14.1 Бібліотека керуючих елементів Windows

Цілком ймовірна ситуація, при якій керуючі елементи, що поставляються разом з **Visual Studio.NET** у готовому вигляді, не задовольнятимуть ваших вимог. На те може існувати багато причин: керуючі елементи мають інший зовнішній вигляд, ці елементи накладають будь-які обмеження, вам потрібно включити до інтерфейсу керуючих елементів додаткові можливості, або, нарешті, необхідного керуючого елемента просто не існує.

Визнаючи це, компанія Microsoft надає засоби для створення елементів, які б задовольняли вимогам користувача. У **Visual Studio.NET** є тип проекту **Windows Control Library** (бібліотека керуючих елементів **Windows**), до якого слід вдатися за необхідності створити свій власний керуючий елемент.

Є можливість розробити два різних види власних елементів, які називаються керуючими елементами (або складними керуючими елементами) і спеціалізованими керуючими елементами (custom controls):

• ***Користувальницькі або складові керуючі елементи:***

Вони будуються з урахуванням функціональних можливостей вже існуючих керуючих елементів, у результаті створюється новий керуючий елемент. Такі елементи зазвичай створюються або для об'єднання функціональних можливостей з інтерфейсом керуючого елемента, або для розширення можливостей інтерфейсу керуючого елемента шляхом об'єднання декількох різних елементів в одну логічну одиницю.

• ***Спеціалізовані керуючі елементи:***

Цей різновид керуючих елементів створюється, коли жоден з існуючих елементів не задовольняє вашим потребам і керуючий елемент створюється з нуля. Він відтворює весь інтерфейс користувача, і при його створенні не використовуються готові керуючі елементи. Зазвичай необхідність у таких елементах виникає у тих випадках, коли інтерфейс створюваного управляючого елемента істотно відрізняється від наявних.

Ми зупинимося на створенні керуючих елементів користувача. У **.NET** керуючі елементи існують у вигляді звичайних **DLL**.

Користувальницькі елементи керування успадковуються від класу **UserControl**. Це базовий клас надає новоствореному елементу всі базові можливості, які повинен мати керуючий елемент в **.NET** – програмісту залишається тільки

створення самого елемента. Як керуючий елемент може створюватися все, що завгодно, починаючи від напису з яким-небудь оригінальним дизайном і до табличного керуючого елемента, що займає весь екран. (На відміну від елементів користувача, керуючі елементи, що спеціально створюються, зазвичай є похідними від класу Control, а не від класу UserControl.) При роботі з керуючими елементами слід приймати ряд положень як належне, і якщо створюваний вами елемент не відповідає їм, то висока ймовірність того , що такий елемент не використовуватиметься іншими користувачами.

• ***Поведінка керуючого елемента*** в період розробки має якнайточніше відповідати поведінці цього керуючого елемента в період виконання. Це означає, що якщо елемент складається з напису та текстового вікна і називається **LabelTextbox**, то і напис, і текстове вікно, і текст, що вводиться для напису, повинні бути видимими в процесі розробки. Це може створювати певні проблеми, що вимагатиме від вас знаходити прийнятний компроміс.

• Необхідно забезпечити ***логічний доступ*** до властивостей керуючого елемента з форми, що використовується для розробки.

Нижче розглядається процес створення елементів, що управляють, на конкретному прикладі. Ми створимо елемент **LabelTextbox** і продемонструємо основи створення проекту з розробки користувача керуючого елемента, його властивостей та подій та налагодження керуючого елемента.

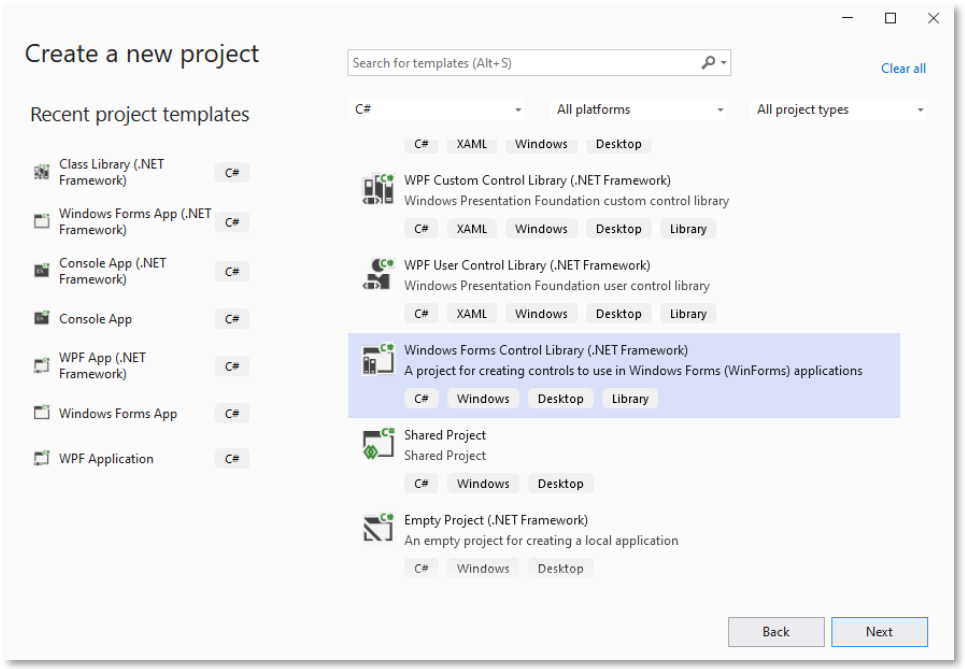
Для створення нового елемента в ньому використовуються два вже існуючі, що дозволяє за один крок максимально просто вирішити задачу, яка досить часто виникає при програмуванні в **Windows**: додати до форми напис, далі додати до тієї ж форми текстове вікно, а потім визначити місце розташування цього вікна щодо напису.

Можливості компонента:

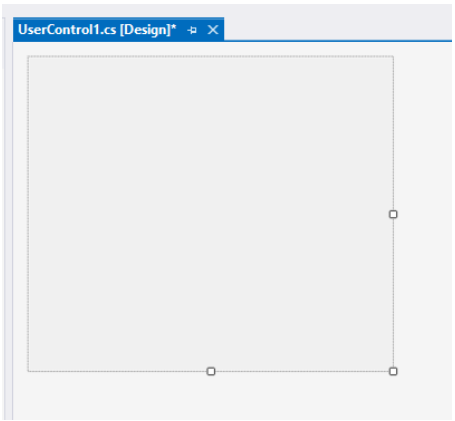
• Можливість розміщувати текстове вікно праворуч від напису, так і під написом. Якщо текстове вікно розташовується праворуч від напису, то має бути можливість задавати фіксовану відстань між лівим краєм керуючого елемента та текстовим вікном, щоб розташувати кілька текстових вікон одне під одним.

• Звичайні властивості та подій текстового вікна та написи, які доступні у звичайному режимі.

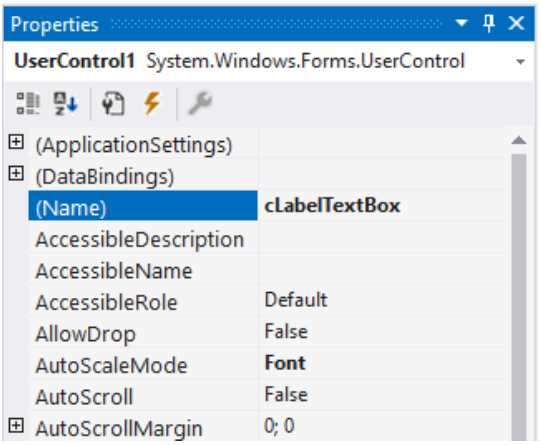
Створіть новий проект **C#** у **Visual Studio**. Виберіть проект під назвою **Windows Form Control Library (Windows)**. Назвіть новий проект **LabelTextbox.**



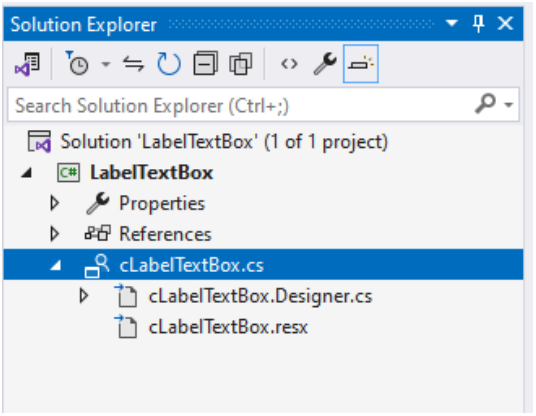
Програма розробки компонент надає робочу поверхню, яка дещо відрізняється від тієї, що використовується досі. По-перше, її площа виявляється суттєво меншою, ніж зазвичай, а, по-друге, вона взагалі не має вигляду діалогового вікна.



Клацніть мишею на робочій поверхні, і ви побачите властивості керуючого елемента. Змініть ім'я елемента на **cLabelTextBox.**



Якщо при цьому не перейменувався клас компонента, зробіть це за допомогою однойменної команди контекстного меню:



Перемістіть на форму мітку з панелі компонентів. Змініть її ім'я на **title**. Надайте властивості **Text** значення **title**. Перенесіть текстове вікно з палітри компонентів на форму. Змініть його ім'я на **text.**

На етапі розробки нам невідомо, як користувач захоче розташувати ці керуючі елементи. Тому потрібно написати код, який відповідатиме за розташування напису та текстового вікна. Цей же код визначатиме місце розташування керуючих елементів і тоді, коли елемент **LabelTextbox** буде поміщений у форму.

Структура керуючого елемента має поки що вельми не вражаючий вигляд: **Textbox** вільно переміщається по робочій поверхні, а сама поверхня занадто велика. Насправді насправді, це не має значення, оскільки, те, що ви бачите, це зовсім не те, що ви отримаєте в підсумку. Код, який ми зараз додамо до керуючого елемента, змінить його зовнішній вигляд, але тільки після додавання елемента до форми.Спочатку слід визначити взаємне розташування керуючих елементів щодо один одного. Оскільки користувач повинен самостійно приймати рішення про це розташування, ми додамо елементу не одну, а дві

властивості. Одне буде називатися **Position** (позиція) — воно надає користувачеві можливість вибору: **Right** (праворуч) і **Below** (внизу). Якщо користувач вибирає значення **Right**, то справа вступає друга властивість. Воно називається **TextBoxMargin** (поля текстового вікна) і є цілим числом, що відповідає числу пікселів між лівим краєм керуючого елемента і тим місцем, в якому має бути розміщене текстове вікно. Якщо користувач задає значення 0, то текстове вікно розміщується так, щоб його правий край розташовувався врівень з правим краєм керуючого елемента.

14.2 Додавання властивостей

Для того, щоб користувач міг вибирати між **Right** і **Below**, перш за все буде потрібно перелічуваний тип з цими значеннями. Поверніться до проекту створення керуючого елемента, перейдіть в редактор коду і введіть наступний код:

```
public partial class cLabelTextBox : UserControl
   {
      public enum PositionEnum { Right, Below }
      public cLabelTextBox()
      {       InitializeComponent();      }
   }
```

Нам потрібно, щоб місце розташування було властивістю, яка може задаватися користувачем як програмним шляхом, так і в процесі розробки. Цього можна досягти, включивши цю властивість до класу **cLabelTextBox**. Спочатку потрібно буде створити два поля, в яких будуть зберігатися значення, вибрані користувачем. Поле - даний член цього класу, в якому зберігатиметься інформація про вибір, зроблений користувачем.

```
public partial class cLabelTextBox : UserControl
   {
      public enum PositionEnum { Right, Below }
      private PositionEnum position = PositionEnum.Below;
      private int textboxMargin = 0;
…
      }
```

Потім додаємо властивість **Position**.

```
public PositionEnum Position
    {
      get
      {   return position;  }
      set
      {   position = value;  MoveControls();  }
    }
```

Ця властивість додана в клас так само, як і будь-яка інша. Якщо виникне потреба повернути її значення, ми повернемо поле **position**, а якщо потрібно змінити властивість **position** — надамо полю position відповідне значення і викличемо метод **MoveControls().** До методу **MoveControls()** ми повернемося трохи нижче, а поки що досить просто знати, що цей метод має в своєму розпорядженні два керуючі елементи на поверхні компонента відповідно до значень полів **position** і **textBoxMargin.**

Аналогічно створюється властивість **TextBoxMargin** за винятком того, що вона працює з цілими числами:

```
public int TextboxMargin
    {
      get
      {    return textboxMargin;     }
      set
      {    textboxMargin = value;  MoveControls();  }
    }
```

Перш ніж перейти до тестування цих двох властивостей, необхідно додати два обробники подій. Коли розроблений компонент поміщається у форму, викликається подія **Load.** Цю подію слід використовувати для ініціалізації компонента і всіх ресурсів, які можуть ним використовуватися. Ми обробляємо цю подію, щоб переміщати керуючі елементи і підганяти розмір всього компонента під розмір елементів, що містяться в ньому. Ми оброблятимемо ще одну подію — **SizeChanged.** Вона виникає завжди, коли змінюються розміри керуючого елементу і необхідно обробляти цю подію, щоб компонент відображався належним чином. Ми підписуємось на події у конструкторі керуючого елемента:

```
public cLabelTextBox()
  {
       InitializeComponent();
       this.SizeChanged += new System.EventHandler(this.OnSizeChanged);
       // Обробка події Load
       this.Load += new EventHandler(this.OnLoad);        }
```

Потім додаємо обробники подій:

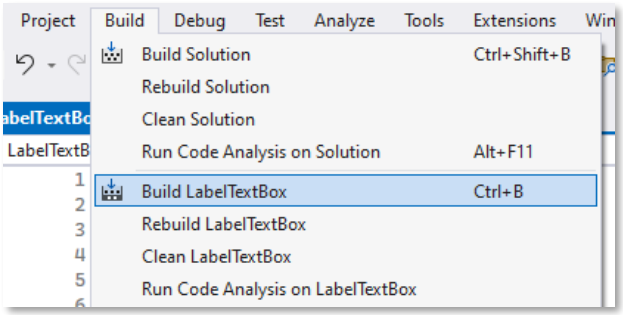
```
private void OnLoad(object sender, EventArgs e)
      {
         // Додає текст до напису
         title.Text = this.Name;
         // Завдання висоти керуючого елемента
         this.Height = title.Height + text.Height;
         MoveControls();
      }
      private void OnSizeChanged(object sender, System.EventArgs e)
      {    MoveControls();     }
```

Метод **MoveControls()** викликається для зміни розташування керуючих елементів. Слід розглянути цей метод, після чого протестувати роботу керуючого елемента:

```
private void MoveControls()
      {
         title.Location = new Point(0, 0);
         switch (position)
         {      case PositionEnum.Below:
                // Маємо в своєму розпорядженні Textbox таким чином,
                // щоб його верхній край знаходився безпосередньо під написом
                text.Top = title.Bottom;
                text.Left = title.Left;
                // Змінюємо ширину Textbox, роблячи її рівною ширині
                // керуючого елемента
                text.Width = this.Width;
                this.Height = text.Height + title.Height;
                break;
            case PositionEnum.Right:
                // Маємо в своєму розпорядженні Textbox таким чином, щоб його
                // верхній край збігався з верхнім краєм напису
                text.Top = title.Top;
                // Якщо поля мають нульове значення, то Textbox
                // розташовується безпосередньо поруч із написом
                if (textboxMargin == 0)
                {
                   text.Left = title.Right + 3;
                   text.Width = this.Width - title.Width - 3;
```

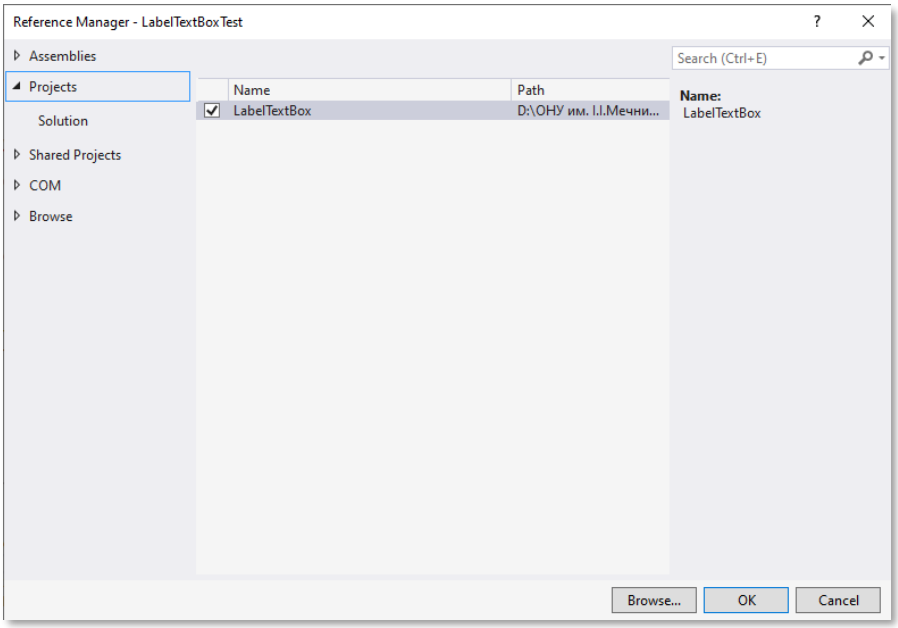
```
            if (text.Height > title.Height)
                this.Height = text.Height + 2;
            else
                this.Height = title.Height + 2;
        }
        else
        {
            // Якщо поля мають ненульове значення, то Textbox
            // розташовується в місці, визначеному користувачем
            text.Left = textboxMargin;
            text.Width = this.Width - textboxMargin;
        }
        break;
    }
}
```

Значення, що зберігається в **position**, використовується для визначення того, чи слід розташовувати текстове вікно знизу або праворуч від напису. Якщо користувач вибирає варіант **Below**, ми переміщуємо текстове вікно таким чином, щоб його верхня межа збіглася з нижньою межею напису. Після цього ми переміщуємо лівий край текстового вікна на рівень лівого краю всього компонента та встановлюємо ширину текстового вікна, що дорівнює ширині компонента. Якщо користувач вибирає варіант **Right**, то у цьому випадку є дві можливості. Якщо **textBoxMargin** дорівнює 0, то ми починаємо з того, що визначаємо ширину незайнятої частини елемента, що управляє, призначеної для розміщення текстового вікна. Після цього ми встановлюємо лівий край текстового вікна так, щоб він торкався правого краю тексту, і задаємо йому таку ширину, щоб він займав простір, що є. Якщо користувач задасть поля, ми знову помістимо лівий край текстового вікна в ту саму позицію і заново визначимо його ширину. Необхідно побудувати цей проект за допомогою команди виду **Build|Build LabelTextbox.**



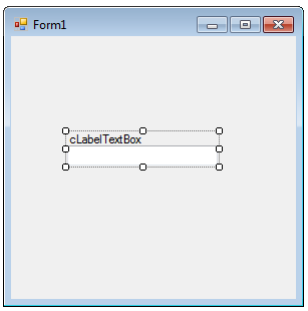
Тепер можна протестувати керуючий елемент. Налагодження власних керуючих елементів істотно відрізняється від налагодження програм **Windows**. Засобу керування потрібен деякий контейнер, у якому він буде виводитися, і нам доведеться доведеться такий контейнер йому надати. Ми зробимо це, додавши в рішення новий проект **Windows Forms** і назвемо його **LabelTextboxTest**. Задамо його стартовим застосунком.

Для того, щоб передати програмі Windows інформацію про створюваний нами керуючий елемент, необхідно включити посилання на цей керуючий елемент. Клацніть правою кнопкою миші на заголовку **References** у **Solution Explorer** під проектом **LabelTextboxTest** і виберіть пункт **Add References**.... Виберіть вкладку з іменем **Projects.** Розкриється список, в якому міститься запис про проект створення керуючого елемента.

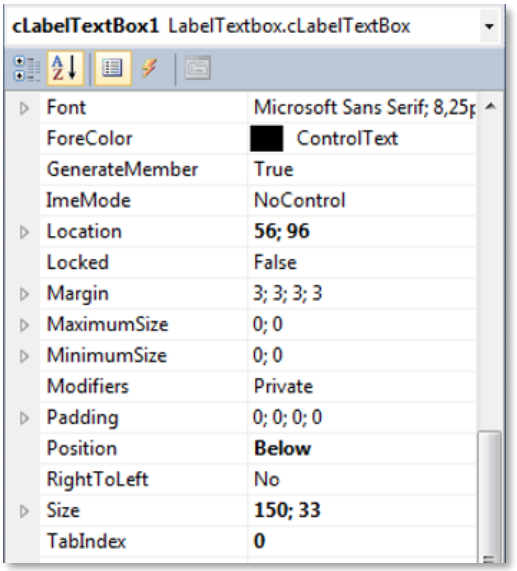


На панелі компонентів з'явиться вкладка з компонентом, що розробляється.

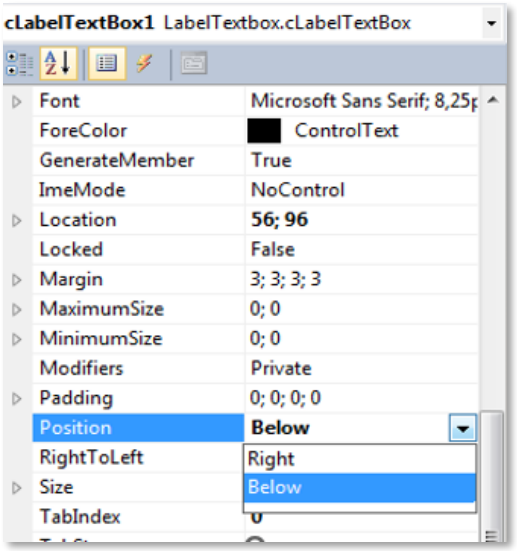
Додайте до форми керуючий елемент **clLabelTextbox**. Ви бачите, що за замовчанням текстове вікно розташовується під написом і вигляд компонента відрізняється від виду в процесі проектування. Це спрацював обробник події **Load,** на яку ми підписалися.



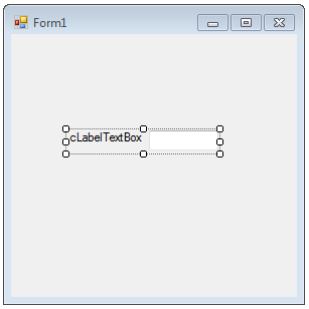
Можна змінити властивості компонента, наприклад, властивість **Position**, яку ми вже спроектували для компонента:



Змінимо значення **Position**:



Вибираємо значення **Right**, тоді зовнішній вигляд компонента зміниться:



Можна перевірити роботу властивості textBoxMargin, яка за замовчанням дорівнює 0. Пам'ятаємо, що ця властивість відіграє роль тільки при значенні **Position** рівним **Right.**

Крім того, спробуйте змінити розмір компонента на етапі проектування, щоб перевірити роботу події компонента **SizeChanged**, на яку ми підписалися.

14.3 Додавання нових властивостей

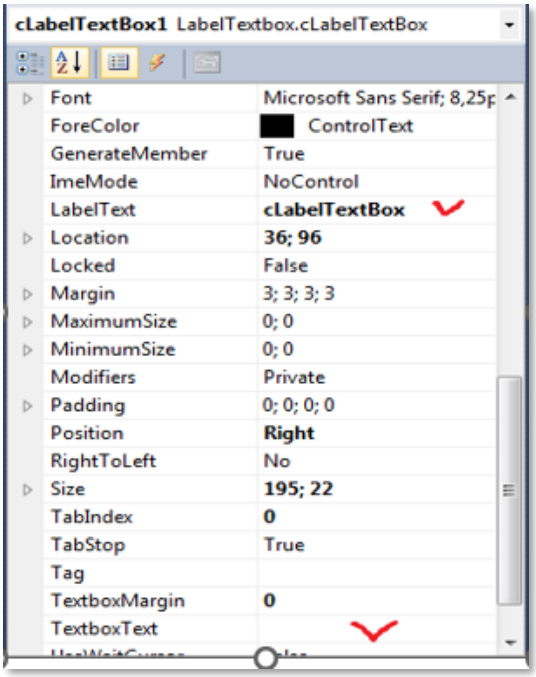
На цьому етапі практично нічого не можна робити за допомогою створеного керуючого елемента, оскільки у нас відсутня можливість змінювати текст напису і текст у вікні. Для роботи з цим текстом ми додамо дві властивості: **LabelText** та **TextBoxText**. Ці властивості додаються так само, як і дві попередні властивості:

```
public string LabelText
    {
      get
      {   return title.Text;   }
      set
      {   title.Text = value;
          MoveControls(); // Метод MoveControls викликається для зміни
          // розмірів Textbox за наявності такої необхідності
      }
    }
    public string TextboxText
    {
      get
      {   return text.Text;    }
      set
      {   text.Text = value;   }
    }
```

Якщо виникає необхідність увімкнути певний текст, ми просто присвоюємо

відповідний текст напису та властивості Text текстового вікна і повертаємо значення названих властивостей. Якщо текст напису буде змінено, нам потрібно звернутися до методу **MoveControls()**, оскільки цей текст може впливати на те, яким чином буде розміщено текстове вікно. З іншого боку, текст, що включається для виведення в текстовому вікні, не призводить до переміщення керуючих елементів, і в тому випадку, якщо він виявиться довшим, ніж текстове вікно, то він просто перестане бути видимим.

Після цього необхідно перебудувати програму, тоді внесені зміни відобразяться для компонента, доданого на форму:



14.4 Додавання нових обробників подій

Оскільки цей компонент є похідним від класу **userControl**, він успадковує від нього велику кількість функціональних можливостей. Однак, існує певна кількість подій, які ми не бажаємо передавати користувачеві стандартним способом. Серед них можна назвати такі події, як **KeyDown, KeyPress** та **KeyUp**. Причина, через яку нам необхідно внести певні зміни в ці події, полягає в тому, що користувач припускає, що вони будуть відправлятися в той момент, коли він натискає будь-яку клавішу текстового вікна. А в тому вигляді, в якому вони існують зараз, ці події будуть відправлятися тільки в тому випадку, якщо компонент знаходиться у фокусі і користувач натискає клавішу.

Для того, щоб змінити таку поведінку, ми повинні самостійно обробляти події,

що надсилаються текстовим вікном, а потім переправляти їх далі користувачеві. Спочатку нам необхідно підписатися на відповідні події у конструкторі класу компонента:

```
public cLabelTextBox()
    {
      InitializeComponent();
      this.SizeChanged += new System.EventHandler(this.OnSizeChanged);
      // Обробка події Load
      this.Load += new EventHandler(this.OnLoad);
      // Обробка подій у Textbox, пов'язаних із клавіатурою
      // додали команди
      this.text.KeyDown += new KeyEventHandler(this.text_KeyDown);
      this.text.KeyUp += new KeyEventHandler(this.text_KeyUp);
      this.text.KeyPress += new KeyPressEventHandler(this.text_KeyPress);
    }
```

Потім слід додати три обробники подій:

```
    private void text_KeyDown(object sender, KeyEventArgs e)
    {   OnKeyDown(e);   }
    private void text_KeyUp(object sender, KeyEventArgs e)
    {   OnKeyUp(e);       }
    private void text_KeyPress(object sender, KeyPressEventArgs e)
    {   OnKeyPress(e);    }
```

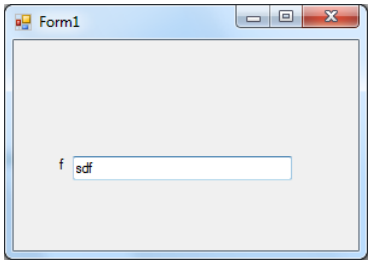
Виклик методу **OnKeyXXX** формує звернення до ланцюжка делегатів, підписаних на дану подію.

Тестування доданих подій

Додамо компоненту cLabelTextBox1 подієву функцію виду:

```
private void cLabelTextBox1_KeyPress(object sender, KeyPressEventArgs e)
    {
      cLabelTextBox1.LabelText = Convert.ToString(e.KeyChar);
    }
```

Результат роботи програми наступний:



14.5 Додавання власного оброблювача подій

Якщо потрібно створити подію, якої не існує в жодному з базових класів, то

доведеться виконати дещо серйознішу роботу. Ми створимо подію з ім'ям **PositionChanged** (позиція змінена), яка виникатиме, коли змінюється значення властивості **Position.**

Для створення такої події потрібно три речі:

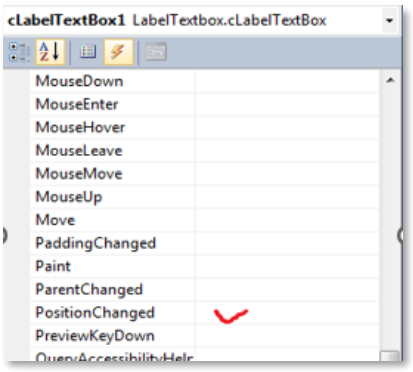
- Нам буде необхідний відповідний делегат, який можна використовувати для виклику методів, що згодом прив'язуються користувачем до події.
- Користувач повинен мати можливість підписатися на подію за допомогою прив'язки до нього деякого методу.
- Ми повинні викликати метод, який користувач прив'язав до цієї події.

Будемо використовувати делегат **EventHandler,** який надається нам системою **.NET Framework**. Наступний код оголошує подію та надає користувачеві можливість на неї передплатити:

```
public event EventHandler PositionChanged;
```

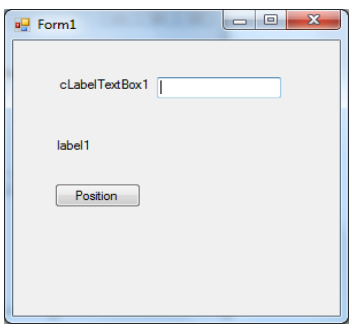
Тепер залишилося лише згенерувати подію. Оскільки вона має виникати в момент зміни значення властивості **Position,** ми генеруємо подію в аксесорі доступу **set** до властивості **Position**.

```
public PositionEnum Position
{  get
   {    return mPosition;     }
   set
   {    mPosition = value;  MoveControls();
      if (PositionChanged != null) // Переконуємось у наявності підписників
         PositionChanged(this, new EventArgs());
   }   }   }
}
```



Спочатку потрібно переконатися в наявності будь-яких підписників на цю подію, перевіряючи, чи не дорівнює значення **PositionChanged null**. Якщо ні, то ми викликаємо цю подію з усім ланцюжком подієвих функцій.

Тестування доданої події



Для кнопки додамо подієву функцію

```
private void button1_Click(object sender, EventArgs e)
      {
         if (fl)
cLabelTextBox1.Position=LabelTextbox.cLabelTextBox.PositionEnum.Below;
         else
cLabelTextBox1.Position=LabelTextbox.cLabelTextBox.PositionEnum.Right;
         fl = !fl;
      }
```

Тут використається прапор **bool fl = true;** - член класу форми. Щоб побачити роботу створеної події, необхідно створити відповідну функцію подій для компонента

```
private void cLabelTextBox1_PositionChanged(object sender, EventArgs e)
      {   label1.Text = label1.Text + "Privet";    }
```

Кожне натискання на кнопці призводить до зміни позиції напису по відношенню до текстового вікна, тобто зміну властивості **Position**, що викликає виконання аксесора **set** цієї властивості, в якому відбувається перевірка ланцюжка делегата події **PositionChanged** і виклик усіх обробників, підписаних на цю подію.

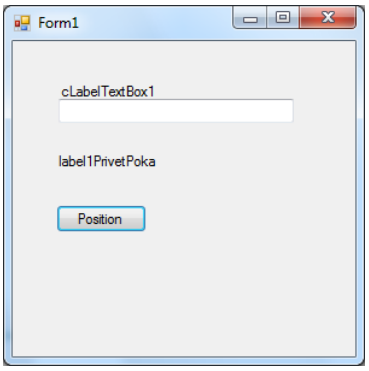
Для того щоб організувати ланцюжок делегатів, додамо ще функцію класу форми з такою самою сигнатурою

```
void my(object sender, EventArgs n)
      {    label1.Text = label1.Text+"Poka";      }
```

І додамо цю подієву функцію в ланцюжок у конструкторі форми:

```
public Form1()
      {    InitializeComponent();
           cLabelTextBox1.PositionChanged += new EventHandler(my);
      }
```

В результаті маємо:



Запитання: поясніть порядок слів?

Зауваження. Необхідно пам'ятати, якщо Ви хочете розроблений компонент використовувати надалі в інших застосунках, то він має бути підключений до панелі компонентів. Для цього вибираємо закладку, на яку Ви хочете додати компонент, потім необхідно клацнути правою кнопкою на панелі у новій вкладці та вибрати команду контекстного меню **Choose Items**… Натисніть на кнопці **Browse** вікна **Choose Toolbox Items** та вкажіть шлях до файлу **LabelTextBox.dll.** Після закриття вікна **Choose Toolbox Items** на панелі інструментів у створеній вкладці з'явиться піктограма компонента **cLabelTextBox.**

15. ЕЛЕМЕНТ УПРАВЛІННЯ DATAGRIDVIEW

Елемент управління **DataGridView** надає потужний та гнучкий спосіб відображення даних у табличному форматі. Елемент керування **DataGridView** можна використовувати для відображення уявлень невеликого обсягу даних тільки для читання, або можна масштабувати його для відображення редагованого представлення дуже великих наборів даних.

Для того, щоб реалізувати поведінку користувача в застосунках, елемент керування **DataGridView** можна розширити кількома способами. Наприклад, можна програмно задати власні алгоритми сортування, можна створити власні типи осередків. Зовнішній вигляд елемента керування **DataGridView** легко налаштовується завданням кількох властивостей. Як джерело даних можуть використовуватися сховища даних різних типів, а також елемент керування **DataGridView** може працювати без прив'язаного до нього джерела даних.

Розглянемо базову поведінку компонента. Якщо помістити **DataGridView** на форму і не виробляти жодних спецналаштувань, то компонент буде:

- Автоматично відображати заголовки колонок та заголовки рядків. І ті, й інші залишаються видимими за будь-якого скролінгу.
- Ставити на одному із заголовків рядків маркер (чорний трикутник) поточного рядка.
- Вибирати цілий рядок, якщо користувач клацне по заголовку рядка.
- Вибирати відразу кілька рядків, якщо клацання по заголовку рядка виконується із затиснутим **Ctrl** або **Shift**. При цьому поточний рядок (позначений трикутником) завжди буде єдиним.
- Видалити всі вибрані рядки після натискання на **Delete.**
- Відображати осередок, що має фокус введення, особливим чином.
- Якщо користувач виконає подвійне клацання по роздільнику колонок, буде проведена автопідбірка ширини лівої колонки.
- Крім цього компонент підтримуватиме редагування вмісту.
- Якщо користувач виконає подвійне клацання по комірці (або натисне на ній **F2**), ця комірка буде переведена в режим редагування.
- Якщо користувач змінить хоча б один символ у редагованій комірці, на заголовку відповідного рядка з'явиться спецсимвол (олівець, що пише), і буде відображатися доти, поки фокус введення не залишить редаговану комірку, або поки користувач не натисне **Esc**. Остання дія відновить значення комірки, яке вона містила до входу в режим редагування.
- Якщо користувач прокрутить компонент вниз до останнього рядка, буде відображено додатковий, спеціальний рядок для внесення нового запису. Такий рядок завжди позначений символом зірочки на заголовку. Коли користувач у будь-який спосіб переміщується в цей рядок, **DataGridView** додає новий запис зі значеннями за промовчанням. Якщо фокус введення знаходиться в цьому рядку, і користувач натискає **Esc**, новий запис пропадає, а фокус введення переміщається на рядок вище.

Якщо **DataGridView** прив'язаний через властивість **DataSource** до джерела даних, за замовчуванням виконується таке:

- Кожна колонка, що отримується з джерела даних, викликає додавання відповідної колонки до компоненту.
- Назви колонок джерела з'являться у заголовках колонок.

Практично все з перерахованого вище може бути ***дозволено/заборонено/налаштовано.***

15.1 Проектування компонента

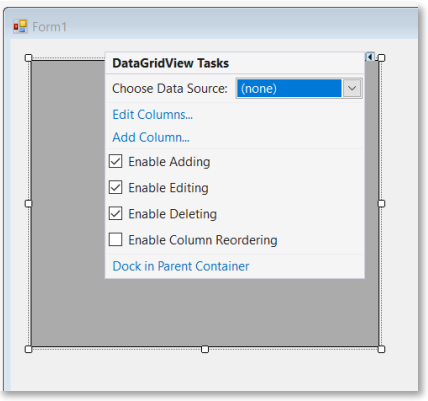
Додавання колонок – крок, абсолютно необхідний перед тим, як компонент буде пред'явлений кінцевому користувачеві. Якщо ми не додали жодної колонки, **DataGridView** буде тьмяно-сірим прямокутником на формі, зовсім позбавлений будь-якої функціональності. Автори компонента передбачили низку способів додавання колонок. У загальному випадку ми стикаємося з чотирма можливими сценаріями:

1. Є джерело даних, воно доступне під час розробки, і ми готові додавати колонки в цей час.
2. Немає джерела даних, але вже під час розробки ми знаємо склад і тип колонок і готові додавати їх.
3. Є джерело даних, але воно доступне лише під час виконання, а під час розробки нічого не відомо ні про нього, ні про склад колонок.
4. Немає джерела даних, а склад/тип колонок з'ясовується динамічно, під час виконання, а під час розробки невідомий тип і, можливо, навіть кількість колонок.

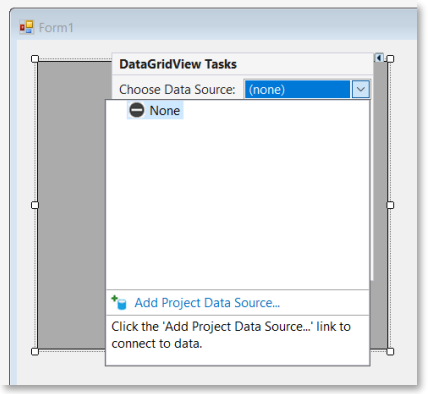
15.2 Джерело даних доступне під час розробки

Найпростіший варіант. Після завдання значень властивостей **DataSource** і **DataMember** компонент автоматично вивчає схему джерела і генерує по колонці для кожної колонки таблиці або властивості об'єкта, колекція яких використовується як джерело даних. Причому робить це "розумно", підбираючи не тільки відповідний заголовок колонки, але і тип колонки. Тобто. якщо тип колонки буде чимось на зразок **int/decimal/string**, то додасться колонка типу **DataGridViewTextBoxColumn**. А якщо така колонка матиме тип **boolean**, то додасться **DataGridViewCheckBoxColumn**. Зрозуміло, в нашій владі видалити "зайві" на наш погляд колонки, виправити текст заголовка, а також тип колонки. Ось як це робиться. Після завдання значень властивостей **DataSource** і, за необхідності, **DataMember**, ми маємо згенеровані за описаним вище алгоритмом колонки. Виділивши **grid** у дизайнері, натиснемо його "*розумний ярлик*" **(smart tag**). "Розумний ярлик" знаходиться у верхньому правому кутку компонента (причому майже будь-якого) і надає доступ до меню, склад елементів якого можна охарактеризувати як "налаштування, що

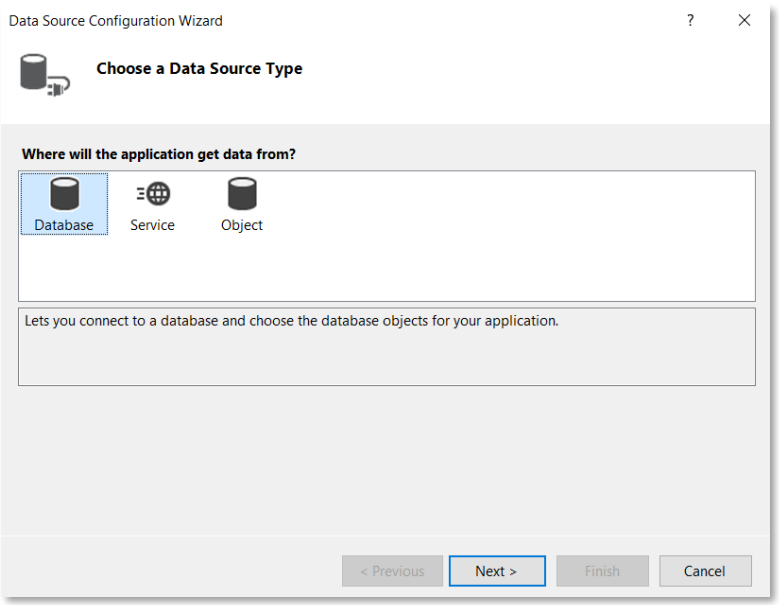
найчастіше використовуються".



За замовчанням джерела даних немає, якщо розкрити список:



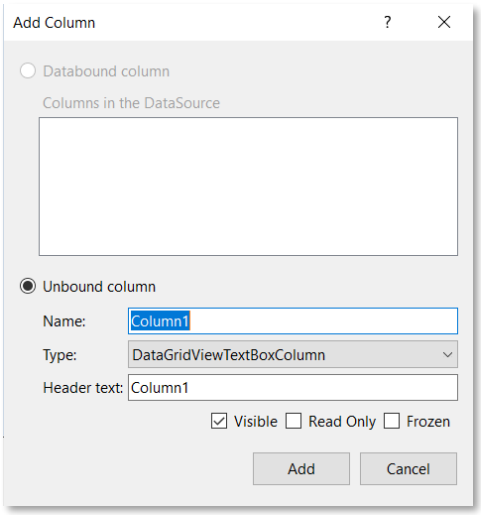
Вибір команди ***додати джерело даних*** проекту відкриває вікно виду:



Підключення БД Ви проводитимете при вивченні відповідних дисциплін.

15.3 Відсутність джерела даних у дизайн-таймі

У цьому сценарії ми не встановлюємо значення властивостей **DataSource** та **DataMember**, а готуємось працювати у вільному, неприв'язаному режимі. Відкриваємо все те ж ***smart tag-меню***, але вибираємо **Add Column**…



Оскільки джерела даних немає (**DataSource** та **DataMember** виставлені в **null**), перемикач **Databound column** не працює. Нам доступний лише перемикач **Unbound column** та підпорядковані йому поля. Задаючи значення у цьому вікні, ми фактично задаємо наступні властивості нової колонки:

1. **Name** (не плутайте із заголовком) – це ім'я створюваної у формі змінної (тип якої DataGridViewColumn або його спадкоємець), в яку міститься посилання на колонку, що додається.
2. **Type** колонки.
3. **Header text**, який буде показаний у заголовку цієї колонки.

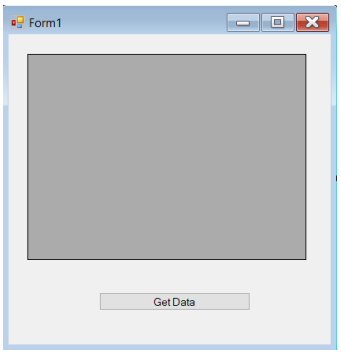
Три перемикачі, що залишилися, допомагають одразу задати відповідні властивості колонки.

Характерно, що після натискання на кнопку **Add** даний діалог не закривається, а пропонує нові значення за замовчуванням – Column2, Column3 і т.д., дозволяючи дуже швидко додати неабияку кількість.

15.4 Готове джерело даних, що підключається під час виконання

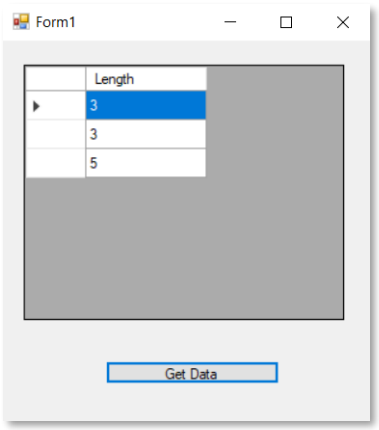
Цей сценарій змагається за простотою з першим найпростішим сценарієм. Якщо властивість **DataGridView.AutoGenerateColumns** виставлено в **true** (а за замовчанням так і є), то під час виконання будь-яка зміна властивостей **DataSource/DataMember** викликає генерацію колонок за алгоритмом сценарію 1. Можна також запустити (перезапустити) цей процес генерації та додавання, встановивши згадану властивість в **false**, а потім повернувши його в **true**.

Приклад 1




```
private void button1_Click(object sender, EventArgs e)
    {
       string[] stuff = new string[] { "One", "Two", "Three" };
       dataGridView1.DataSource = stuff;
    }
```

Після натискання кнопки:



Замість відображення рядків, визначених масивом, таблиця показує довжини цих рядків. Причина полягає в тому, що коли масив використовується як джерело даних для **DataGridView**, він шукає першу загальнодоступну властивість об'єкта, що міститься в масиві, замість рядкового значення. Перша (і єдина) загальнодоступна властивість рядка – її довжина, тому вона і

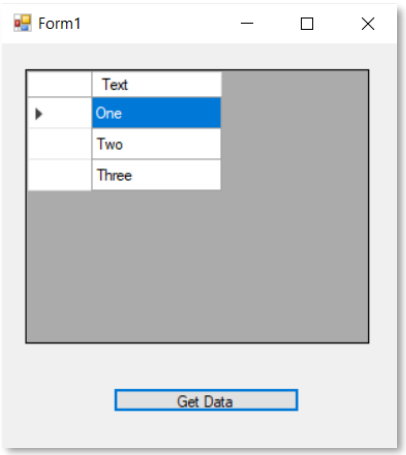
відображається. Список властивостей кожного класу можна отримати за допомогою методу **GetProperties()** класу **TypeDescriptor**. Він повертає колекцію об'єктів **PropertyDescriptor**, яка може використовуватися для відображення даних. Елемент керування **.NET PropertyGrid** застосовує цей метод для відображення довільних об'єктів.

Один із способів вирішення цієї проблеми з відображенням рядків у **DataGridView** полягає у створенні класу-оболонки:

Приклад 2.

```
public class Item
   {
      public Item(string text)
      {   _text = text;    }
      public string Text
      {   get { return _text; }   }
      private string _text;
   }
private void button1_Click(object sender, EventArgs e)
      {
         Item[] stuff = new Item[3];
         stuff[0] = new Item("One");
         stuff[1] = new Item("Two");
         stuff[2] = new Item("Three");
            dataGridView1.DataSource = stuff;
      }
```

Результат:

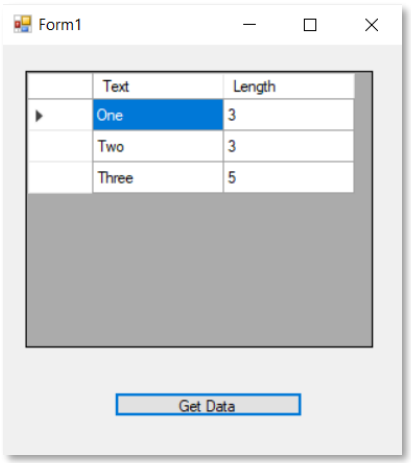


Додамо до класу ще одну властивість:

Приклад 3.

```
public class Item
   {
      public Item(string text)
      {    _text = text;   }
      public string Text
      {    get { return _text; }   }
      private string _text;
      public int Length
      {    get { return _text.Length; }    }
   }
```

Результат:

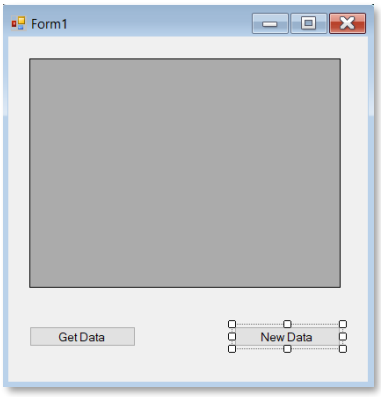


Приклад 4.

Додамо ще один клас з однією властивістю:

```
public class NoneText
  {
      public NoneText(Point A)
      {      t = A;       }
      public Point P
      {      get { return t; }      }
      private Point t;
  }
```

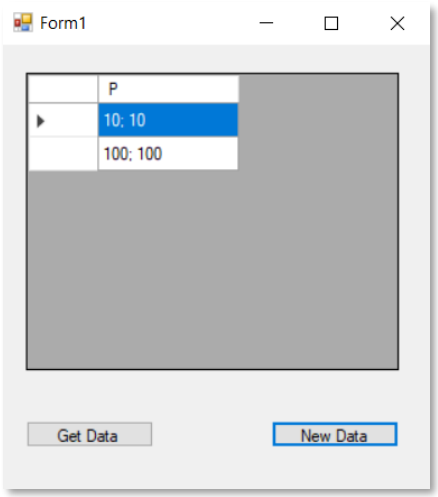
На формі розмістимо ще одну кнопку:



Функція клацання на другій кнопці має вигляд:

```
private void button2_Click(object sender, EventArgs e)
    {
        NoneText[] stuff = new NoneText[2];
        stuff[0] = new NoneText(new Point(10,10));
        stuff[1] = new NoneText(new Point(100, 100));
        dataGridView1.DataSource = stuff;
    }
```

Зробимо клацання на першій кнопці, а потім на другій:



15.5 Відсутність джерела даних під час виконання

В даному випадку немає іншого виходу, окрім як попрацювати ручками та скласти власну колекцію колонок. Перший крок – визначити тип колонок, які хотілося б бачити у **grid**. Оскільки колонки всіх типів (і вбудовані, і користувацькі) додаються в **grid** однаково, у цьому розділі буде розглянуто

роботу з найпоширенішим типом колонки **DataGridViewTextBoxColumn.** Робота з усіма іншими типами колонок є абсолютно аналогічною.

Один із способів – це встановити властивість **ColumnCount grid**-а в якесь значення більше нуля:
dataGridView1.ColumnCount = 3;

При цьому будуть створені колонки, ініціалізовані значеннями за замовчанням (порожніми рядками та нулями), але це можна налаштувати окремо.

Можна програмно додавати колонки методом Add(), а саме пару *ім'я-заголовок*:
dataGridView1.Columns.Add("MyColumnName", "MyColumnHeaderText");

Властивість є колекцією колонок, тому видалення можна здійснювати методами: **Remove(), RemoveAt()** та **Clear().**

15.6 Додавання нових рядків

Після додавання колонок додамо рядки. Відразу домовимося, що рядки розглядатимуться як смужки (**bands**), що поєднують осередки. Властивість **DataGridView.Rows** (типу **DataGridViewRowCollection**) забезпечує доступ до колекції рядків. Користуючись ним, можна додати рядки до **grid**. Але на відміну від чотирьох можливих сценаріїв додавання колонок, у разі додавання рядків сценарій всього один. Справа в тому, що додавати рядки під час розробки не можна в принципі, оскільки властивість **Rows** позначена атрибутом **Browsable (false)** і не відображається в **Property Grid**. Додати рядки в **DataGridView** можна або програмно, скориставшись методом **Add** колекції рядків, або підключивши до нього джерело даних.

Метод Add() має чотири варіанти:

```
// додає один рядок, заповнюючи його значеннями за умовчанням
int Add();

// додає один рядок, заповнюючи його значеннями з масиву values
int Add(params object[] values);

// додає кілька рядків, заповнюючи їх значеннями за замовчанням
int Add(int count);
```

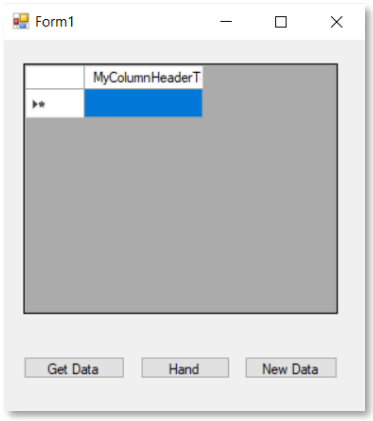
```
// додає заздалегідь створений рядок
int Add(DataGridViewRow dataGridViewRow);
```

DataGridView припускає наявність в одній колонці осередків різних типів! Що для цього потрібно зробити? Для цього спочатку об'єкт типу **DataGridViewRow** повинен бути створений і заповнений окремо. І тільки потім доданий до **grid.** При цьому кількість колонок у рядку має відповідати числу колонок **grid**-а.

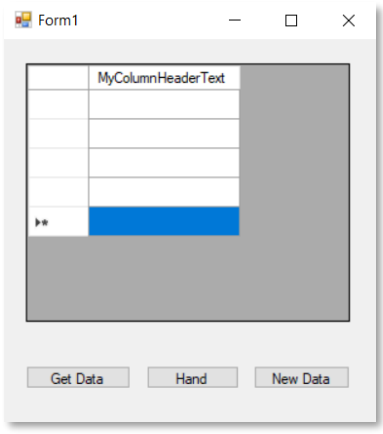
Приклад 6.

```
public Form1()
      {
         InitializeComponent();
         dataGridView1.Columns.Add("MyColumnName", "MyColumnHeaderText");
      }
private void button3_Click(object sender, EventArgs e)
      {   dataGridView1.Rows.Add();     }
```

У момент запуску застосунок має вигляд:



Кожне натискання на кнопці **Hand** призводить до додавання нового рядка:



15.7 Значення осередків.

Вільними даними називаються дані, що заносяться в комірки вручну. ***Пов'язаними*** даними називаються дані, що заносяться в комірки автоматично з прив’язаного джерела даних. Чи будуть дані вважатися пов'язаними або вільними, визначається лише на рівні колонок.

Колонка стає пов'язаною, якщо в її властивість **DataGridViewColumn.DataPropertyName (**типу **string)** заноситься назва колонки або властивості об'єкта з джерела даних. Колонка вважається вільною, якщо згадана властивість стає рівною порожньому рядку.

Якщо властивість **AutoGenerateColumns** виставлена в **true, DataGridView** не тільки генерує новий список колонок при підключенні джерела даних, але й прив'язує ці колонки колонкам або властивостям джерела даних.

Головне в осередках – їх значення. У вільних колонках значення зберігаються в екземплярах осередків. У разі осередків, пов'язаних із джерелом даних, осередки взагалі нічого не знають про значення і не зберігають їх. Якщо потрібно значення комірки, **grid** звертається до джерела даних значення.

За тип даних, що зберігаються в комірці (тобто її властивості **Value**), відповідає властивість **ValueType**, що вказує комірці, дані якого типу вона повинна зберігати. За умовчанням вважається, що у властивості **Value** комірки зберігається об'єкт невизначеного типу (**ValueType i object**). Якщо змінити це значення на значення конкретного типу, осередок почне контролювати значення, конвертуючи його в/з типу, вказаного у властивості **ValueType.**

При зв'язуванні колонки з джерелом даних її властивість **ValueType** отримує значення (відповідне значення типу даних колонки/властивості джерела даних), що призводить до зміни значення **ValueType** осередків.

Для малювання осередку grid має набути значення її властивості **FormattedValue.** Властивості **FormattedValueType** колонок та осередків визначають тип, що використовується для екранного відображення. Найчастіше використовується тип **string.** На рівні комірки формат контролюється методом **DataGridViewCell.GetFormattedValue().**

Важливо розуміти, що потенційно осередки можуть зберігати значення будь-яких типів даних. Має значення лише можливість конвертування цих типів даних у тип, зазначений у **FormattedValueType.** За умовчанням **DataGridView** для цього перетворення використовує **TypeConverter.** Вибір потрібного **TypeConverter** ґрунтується на значеннях властивостей **ValueType** та **FormattedValueType.**

У разі комірки **FormattedValue** запитується багато разів. При кожній промальовці комірки або автоматичній зміні розмірів колонки за вмістом комірки, і навіть при визначенні, чи курсор миші над цією коміркою. При кожному запиті **FormattedValue DataGridView** генерує подію **CellFormatting**. Це дає можливість змінювати форматування комірки.

Якщо осередок не може отримати значення FormattedValue, генерується подія **DataError.**

*Приклад 7***.** Розглянемо програму, яка містить інформацію про особисту бібліотеку користувача. У бібліотеці зберігаються книги, які задаються як об'єкти класу:

```
public class Book
  {
    String author;
    String title;
    uint year;
    uint count;

     public string Author
     {  get   { return author; }  }
     public string Title
     {  get   { return title; }    }
     public uint Year
     {  get   { return year; }    }
     public uint Count
     {  get   { return count; }   }
```

```
        public Book(string author, string title, uint year, uint count)
        {
            this.author = author;
            this.title = title;
            this.year = year;
            this.count = count;
        }
    }
```

Бібліотека – це клас, що інкапсулює у собі список книг:

```
public class Library
    {
        List<Book> books;
        public List<Book> Books
        {
            get
            { return books; }
        }
        public Library()
        {
            books = new List<Book>();
        }
        public void Add(Book b)
        {
            books.Add(b);
        }
        public void RemoveAt(int i)
        {
            books.RemoveAt(i);
        }
    }
```

Компонент **DataGridView** повинен відображати інформацію про книги бібліотеки, тому має містити 4 стовпці: Автор, Назва, Рік видання та кількість сторінок у книзі.

У класі форми оголосимо поле посилання бібліотеки:

```
public Library myLibrary;
```

Ініціалізуємо посилальну змінну в конструкторі форми:
myLibrary = new Library();

Зв'яжемо таблицю з ім'ям dataLibrary зі списком книг бібліотеки:
dataLibrary.DataSource = myLibrary.Books;

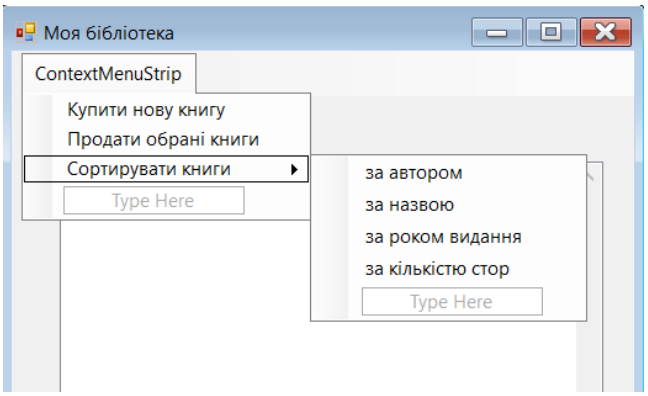
За замовчуванням назви стовпців прописуються назви властивостей класу Book. Програмним шляхом змінимо текст:

```
dataLibrary.Columns[0].HeaderText = "Автор";
dataLibrary.Columns[1].HeaderText = "Назва";
dataLibrary.Columns[2].HeaderText = "Рік";
dataLibrary.Columns[3].HeaderText = "Стор";
```

Для більш естетичного оформлення таблиці додамо до списку книг один порожній рядок, щоб відображався хоча б один рядок таблиці (якщо бібліотека порожня):

```
if (myLibrary.Books.Count == 0)
        {
           myLibrary.Add(new Book("", "", 0, 0));
        }
```

Додамо в застосунок контекстне меню таблиці з наступним функціоналом:



При натисканні на пункті меню «Купити нову книгу» виконується відкриття другі форми програми. Після натискання на кнопку «Додати до моєї бібліотеки» книга додається до списку та таблиці.
За виконання цього функціоналу відповідає подієва функція:

```
private void купитиНовуКнигуToolStripMenuItem_Click(object sender, EventArgs
e)
      {
         Form2 f2 = new Form2();
         f2.Owner = this;
         if (f2.ShowDialog() == DialogResult.OK)
         {
            if (emptyLibrary)
            { myLibrary.RemoveAt(0); emptyLibrary = false; }
            dataLibrary.DataSource=null;
            LoadSource();
         }
      }

      void LoadSource()
      {
         dataLibrary.DataSource = myLibrary.Books;
         dataLibrary.Columns[0].HeaderText = "Автор";
         dataLibrary.Columns[1].HeaderText = "Назва";
         dataLibrary.Columns[2].HeaderText = "Рік";
         dataLibrary.Columns[3].HeaderText = "Стор";
      }
```

Примітка 1. У коді є булевий прапор **emptyLibrary** , який належить класу форми і приймає значення true, якщо список книг порожній. Якщо додаються книги, прапор приймає значення **false.**

***Примітка 2*.** Додавання книги відбувається в коді другої форми в функції функції кнопки «Додати в мою бібліотеку»**:**

```
private void button1_Click(object sender, EventArgs e)
      {
         if (textBox1.Text=="" || textBox2.Text=="")
            {  MessageBox.Show("Введіть данні");
               DialogResult=DialogResult.None;
                return;     }
         try
           {   (Owner as Form1).myLibrary.Add(new Book(textBox1.Text,
textBox2.Text, (uint)numericUpDown1.Value, (uint)numericUpDown2.Value));
            }
```

```
        catch (NullReferenceException)
        {   MessageBox.Show("Створіть бібліотеку ");       }
    }
```

При натисканні на пункті меню «Продати вибрану книгу» виконується подієва функція:

```
private void продатиToolStripMenuItem_Click(object sender, EventArgs e)
    {
        myLibrary.RemoveAt(dataLibrary.CurrentRow.Index);
        if (myLibrary.Books.Count == 0)
        {
            emptyLibrary = true;
            NullBook();
        }
        dataLibrary.DataSource = null;
        LoadSource();
    }
 void NullBook()
    {
        myLibrary.Add(new Book("", "", 0, 0));
    }
```

Потрібно також забезпечити функціонал сортування даних. Цей функціонал організований за допомогою методу **Sort()** списку **List<Book> myLibrary**.

Застосунок додає чотири класи для сортування за різними полями книг. Розглянемо клас

```
  public class A : IComparer<Book>
  {
    public int Compare(Book x, Book y)
    {
        return x.Year.CompareTo(y.Year);
    }
  }
```

Він реалізує інтерфейс **IComparer** типу **Book.** В інтерфейсі **IComparer** визначається лише один метод **Compare():**

```
int Compare(object x, object y);
```

У методі **Compare()** порівнюються об'єкти **x** та **y**. Для сортування об'єктів за наростаючою конкретна реалізація даного методу має повертати нульове значення, якщо значення об'єктів, що порівнюються, рівні; позитивне - якщо значення об'єкта **x** більше, ніж у об'єкта **y**; і негативне - якщо значення об'єкта **x** менше, ніж у об'єкта **y**. А для сортування за спадною можна звернути результат порівняння об'єктів. Якщо ж тип об'єкта **x** не підходить для порівняння з об'єктом **y**, то в методі **Compare()** може бути згенеровано виключення **ArgumentException**.

При реалізації методу **Compare()** використовується метод **CompareTo()** із сигнатурою: `public int CompareTo (object value);`

Метод порівнює поточний екземпляр із зазначеним об'єктом та повертає ціле значення, як описано вище.

При натисканні на пункті меню «Сортувати книги | за роком видання» виконується наступний код:

```
private void заРокомВиданняToolStripMenuItem_Click(object sender, EventArgs e)
    {
      myLibrary.Books.Sort(new A());
      dataLibrary.DataSource = null;
      LoadSource();
    }
```

Природно, що для збереження інформації треба проводити запис у файли, щоб при наступному запуску програми бібліотека вже складалася з книг, які були в ній при попередньому завершенні роботи програми.

16. ЛАБОРАТОРНІ РОБОТИ

16.1 Лабораторна робота №7: Вивчення принципів візуального програмування

1. Форма повинна мати назву ***"Кінематика точки".***
2. Розміри форми мають бути незмінними.
3. Колір форми, властивий їй за замовчуванням, замінити довільним.
4. Кольори міток мають бути ідентичні кольору форми.
5. Напис на мітках із формулами закону руху має розташовуватися по центру.

6. Змінити шрифт напису на цих же мітках.

7. Введення користувачем у текстових полях не припустиме.

8. Введення констант **a** і **b** відбувається за допомогою горизонтальних прокручувань. При цьому напис на мітках з формулами, який містить константи **a** і **b**, безперервно змінюється. А так само безперервно змінюються числові значення для швидкості та прискорення, якщо ці текстові поля висвічуються на екрані.

9. Текстові поля спочатку залишаються невидимими. Перше клацання по відповідній командній кнопці робить відповідне текстове вікно видимим (кнопка ***"Швидкість"*** керує текстовим вікном для виведення значення швидкості, кнопка ***"Прискорення"*** управляє текстовим вікном для виведення значення прискорення), повторне клацання по відповідній командній кнопці робить відповідне текстове вікно невидимим тощо.

10. Після клацання на командній кнопці ***"Вихід"*** відбувається вихід із програми.

16.2 Лабораторна робота №8: Робота з текстовою інформацією

1. Форма повинна мати назву "Редагування та оформлення тексту".

2. Розміри форми мають бути незмінними.

3. На формі розташовано два компоненти типу Textbox у багаторядковому режимі. В перше поле вводиться користувачем довільний текст, а в друге поле текст виводиться згідно з пунктами меню, при цьому при кожному новому виборі пункту меню друге поле попередньо очищається. Введення в друге поле заборонено.

4. На формі розташований контейнер із компонентами для вибору оформлення тексту.

5. Три радіокнопки, розташовані на формі, спочатку невидимі.

6. Форма має містити меню згідно з умовою варіанта лабораторної роботи.

7. Спочатку пункти меню оформлення тексту не доступні для користувача. 8. Дві комірки стану мають написи, що відповідають підпунктам меню "Оформлення тексту". Усі спочатку не вибрані і мають два режими вибору.

9. При виборі комірки стану (у комірці стає галочка) відповідний пункт меню стає доступним для користувача.

10. При повторному клацанні на комірці стану (комірка очищається)

відповідний пункт меню стає знову не доступним.

11. При виборі підпунктів меню "Оформлення тексту" (коли він доступний для користувача) радіокнопки повинні мати написи, оформлені згідно з пунктом меню.

- ✓ Пункт меню ***"Колір шрифту"***: тексти написів радіокнопок Червоний, Зелений, Синій, оформлені відповідним кольором, при цьому розмір і шрифт тексту за замовчуванням, текст не оформлений додатковими накресленнями.
- ✓ Пункт меню ***"Шрифт"***: тексти написів радіокнопок Microsoft Sans Serif, Impact, Courier New, оформлені відповідним шрифтом, при цьому розмір і колір тексту за замовчуванням, текст не оформлений додатковими накресленнями.
- ✓ Пункт меню ***"Розмір"***: тексти написів радіокнопок 8, 10, 12 оформлені відповідним розміром, при цьому колір і шрифт тексту за замовчуванням, текст не оформлений додатковими накресленнями.
- ✓ Пункт меню ***"Підкреслення"***: тексти написів радіокнопок Напівжирний, Курсив, Підкреслений оформлені відповідним накресленням, при цьому колір, шрифт і розмір тексту за замовчуванням, текст не оформлений додатковими накресленнями.
- ✓ Пункт меню ***"Колір заливки"***: текст написів радіокнопок Жовтий, Помаранчевий, Фіолетовий, поля тексту оформлені відповідним кольором, при цьому розмір, колір і шрифт тексту за замовчуванням, текст не оформлений додатковими накресленнями.

12. При клацанні на кнопці "Застосувати" текст у лівому текстовому вікні оформляється відповідно до обраної користувачем комбінації кольору, шрифту, розміру, накреслення або кольору заливки.

13. Під час вибору підпунктів меню "Редагування тексту" текст у лівому текстовому вікні форматується обраним чином.

14. Під словом мається на увазі будь-який набір символів, відокремлених один від одного пробілами або символами переходу на новий рядок.

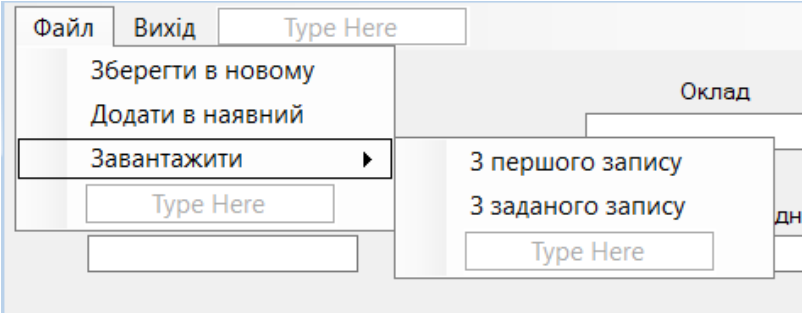
15. Для отримання інформації про символ, який треба видалити або символ, що замінює, або необхідної додаткової інформації для виконання завдання рекомендується застосовувати додаткові форми-діалоги, відкриті в модальному режимі.

16.3 Лабораторна робота №9: Динамічні компоненти

1. На формі в процесі проектування розміщена тільки група з двома радіокнопками.
2. Мітки і кнопки створюються програмним шляхом. Текстові поля так само створюються програмним шляхом, причому передбачається, що є три масиви компонент – для координат, швидкості та прискорення.
3. Закон руху задається варіантом завдання.
4. Група використовується для вибору режиму роботи програми: розрахунку або перевірки. За замовчуванням вважається обраним режим ***"Розрахунок".***
5. Якщо користувач обрав режим ***"Розрахунок"***, то стає видимою кнопка з написом ***"Розрахунок"*** (у разі першого вибору в групі) або змінюється напис (у разі повторного вибору), також очищаються текстові поля від наявної інформації і введення в них стає неможливим. Клацання на цій кнопці ***"Розрахунок"*** призведе до заповнення текстових полів відповідними числами (при цьому поле має бути стандартного кольору).
6. Якщо користувач обрав режим ***"Перевірка"***, то змінюється напис на кнопці, так само очищуються текстові поля від наявної інформації і фокус передається на перше текстове поле. Клацання на цій кнопці ***"Перевірка"*** призведе до перевірки інформації в текстових полях. Поля, у яких відповідь правильна, забарвлюються в червоний колір.
7. Клацання на мітці з моментами часу виводить на екран модальне вікно, у якому користувач може задати нову величину збільшення часу, тобто величину, на яку відрізняються один від одного моменти часу на мітці. Початковий момент часу завжди залишається рівним нулю. Для введення нового інтервалу часу необхідно використовувати компонент відображення цілих чисел. Під час відкриття модального вікна – значення відповідає поточному значенню збільшення.
8. При підведенні курсору миші до текстових полів у режимі ***"Перевірка"*** з'являються спливаючі підказки формул, за якими обчислюються відповідні величини.

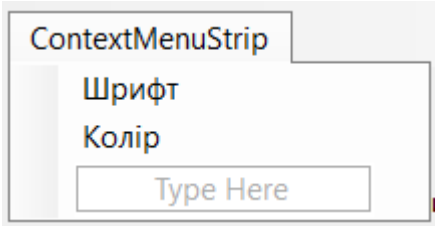
16.4 Лабораторна робота №10: Обробка інформації, що зберігається у файлах

1. Для організації роботи додатка використовувати клас лабораторної роботи №3, доповнивши його полем **Name**, у якому зберігатиметься ім'я об'єкта.
2. На формі організувати меню виду:



3. На формі розташувати список (для імені об'єкта з можливими варіантами вибору імені), мітки і вікна редагування в кількості достатній для створення об'єкта Вашого класу (на малюнку вказано можливий інтерфейс). Усі характеристики, що задаються, повинні мати відповідні написи для зручності користувача. Крім того, обов'язково мають бути присутніми поля (принаймні одне) *"Обчислювальна характеристика"* доступне тільки для читання і заповнюється після клацання на кнопці *"Обчислити"*.
При клацанні на пункті меню ***"Зберегти в новому"*** організовується збереження об'єкта в новий файл за допомогою діалогового вікна збереження файлів.
5. При клацанні на пункті меню ***"Додати в наявний"*** об'єкт записується в кінець файлу, зазначеного користувачем за допомогою діалогового вікна збереження файлів.
6. При клацанні на пункті меню ***"Завантажити з першого запису"*** у вікна редагування завантажується перший об'єкт із файлу, обраного користувачем за допомогою діалогового вікна відкриття файлів.
7. При клацанні на пункті меню ***"Завантажити з заданого запису"*** у вікна редагування завантажується об'єкт (з файлу, обраного за допомогою діалогового вікна відкриття файлів) з номером, зазначеним користувачем (організовується діалог з використанням модальної форми вибору номера запису). Максимум у вікні вибору номера має відповідати кількості записів у файлі.
8. На формі є також дві кнопки ***"Попередній запис"*** і ***"Наступний запис"***, які під час запуску програми невидимі. Якщо користувач завантажив інформацію з файлу, то залежно від номера запису, з якого починалося читання файлу,

стають видимими дві або одна із зазначених кнопок.
9. При клацанні на кнопці ***"Наступний запис"*** користувач переміщається по інформації з файлу, поки не дійде до останнього запису, при цьому кнопка ***"Наступний запис"*** стає невидимою.
10. При клацанні на кнопці ***"Попередній запис"*** користувач переміщається по інформації з файлу, поки не дійде до першого запису, при цьому кнопка ***"Попередній запис"*** стає невидимою.
11. На формі є контекстне меню виду:



12. При клацанні на пункті меню **"Шрифт"** користувач за допомогою відповідного діалогового вікна може вибрати шрифт для тексту форми. При клацанні на пункті меню **"Колір"** користувач за допомогою відповідного діалогового вікна може вибрати колір для тексту форми.
13. При клацанні на пункті меню ***"Вихід"*** здійснюється завершення роботи програми.

Рекомендації:

1. Під час запису бази даних рекомендується створювати два файли: 1-й містить інформацію про об'єкти, 2-й (індексний файл) - про кількість записів у першому файлі.
2. Для читання і збереження інформації рекомендується використовувати клас FileStream.

16.5 Лабораторна робота №11: Технологія Drag&Drop

1. Форма має містити текстове вікно, в якому буде відображатися текст завантажений з файлу.

2. Користувач може перетягувати слова з цього вікна у два інші текстові вікна, кожне з яких приймає слова за певним критерієм (згідно варіанту).

3. Реалізувати лічильники, що відображають кількість загальних спроб сортування та кількість успішних спроб (коли об'єкти сортувалися правильно).

4. Програма повинна мати інтуїтивно зрозумілий інтерфейс, який дозволяє користувачеві легко взаємодіяти з об'єктами та спостерігати за статистикою сортування.

16.6 Лабораторна робота №12: Створення власного пакету компонента

Розробити пакет власного компонента, клас розробленого компонента повинен бути спадкоємцем наявного в **C#** класу. У новому компоненті мають бути передбачені власні властивості, метод і подія.

Варіант №1.

Розробити компонент, який є спадкоємцем класу **Button**. Компонент повинен мати нову властивість: **Caption**. **Caption** – це рядок символів. У компонента є метод **ChangeTitle**, який поміщає **Caption** у текст батька. Крім того, компонент має подію **OnChangeTitle**, яка викликається в той момент, коли викликається метод **ChangeTitle.**

Варіант №2.

Розробити компонент, який є спадкоємцем класу **TextBox**. Компонент повинен мати нову властивість: **Caption**. **Caption** – це рядок символів. У компонента є метод **ChangeTitle,** який поміщає **Caption** у текст батька. Крім того, компонент має подію **OnChangeTitle**, яка викликається в той момент, коли викликається метод **ChangeTitle**.

Варіант №3.

Розробити компонент, який є спадкоємцем класу **Label**. Компонент повинен мати нову властивість: **Caption. Caption** – це рядок символів. У компонента є метод **ChangeTitle,** який поміщає **Caption** у текст батька. Крім того, компонент має подію **OnChangeTitle**, яка викликається в той момент, коли викликається метод **ChangeTitle**.

Варіант №4.

Розробити компонент, який є спадкоємцем класу **CheckBox**. Компонент повинен мати три нові властивості: **Value1, Value2, Operators. Value1 і Value2** – це числові значення. Властивість **Operators** має тип **char**, оскільки може містити тільки один символ – математичний оператор. У компонента є метод **Raschet**, який виконує дію типу **Value1 Operators Value2.** Отриманий результат відображається у властивості **Text** компонента. Крім того, компонент має подію **OnRaschet,** яка викликається в той момент, коли викликається метод **Raschet**.

Варіант №5.

Розробити компонент, який є спадкоємцем класу **RadioButton.** Компонент повинен мати три нові властивості: **Value1,Value2, Operators. Value1 і Value2** – це числові значення. Властивість **Operators** має тип char, оскільки може містити тільки один символ - математичний оператор. У компонента є метод **Raschet**, який виконує дію типу **Value1 Operators Value2**. Отриманий результат відображається у властивості **Text** компонента. Крім того, компонент має подію **OnRaschet,** яка викликається в той момент, коли викликається метод **Raschet.**

Варіант №6.

Розробити компонент, який є спадкоємцем класу **Label.** Компонент повинен мати три нових: **Value1, Value2, Operators. Value1 і Value2** - це числові значення. Властивість **Operators** має тип char, оскільки може містити тільки один символ – математичний оператор. У компонента є метод **Raschet**, який виконує дію типу **Value1 Operators Value2.** Отриманий результат відображається у властивості **Text** компонента. Крім того, компонент має подію **OnRaschet,** яка викликається в той момент, коли викликається метод **Raschet**.

Варіант №7.

Розробити компонент, який є спадкоємцем класу **TextBox.** Компонент повинен мати три нові властивості: **Value1, Value2, Operators. Value1 і Value2** – це числові значення. Властивість **Operators** має тип **char**, оскільки може містити тільки один символ - математичний оператор. У компонента є метод **Raschet**, який виконує дію типу **Value1 Operators Value2**. Отриманий результат відображається у властивості **Text** компонента. Крім того, компонент має подію **OnRaschet**, яка викликається в той момент, коли викликається метод **Raschet**.

Варіант №8.

Розробити компонент, який є спадкоємцем класу **ComboBox**. Компонент повинен мати три нові властивості: **Value1, Value2, Operators. Value1 і Value2** – це числові значення. Властивість **Operators** має тип **char**, оскільки може містити тільки один символ – математичний оператор. У компонента є метод **Raschet**, який виконує дію типу **Value1 Operators Value2**. Отриманий результат відображається у властивості **Text** компонента. Крім того, компонент має подію **OnRaschet**, яка викликається в той момент, коли викликається метод **Raschet**.

Варіант №9.

Розробити компонент, який є спадкоємцем класу **ListBox.** Компонент повинен мати нову властивість: **Stroka**. **Stroka** має тип **String** і містить у собі текст. У компонента є метод **Slova**, який підраховує кількість слів у **Stroka**. Крім того, компонент має подію **OnSlova**, яка викликається в той момент, коли викликається метод **Slova**.

Варіант №10.

Розробити компонент, який є спадкоємцем класу **ScrollBar**. Компонент повинен мати нову властивість: **Komponent**. Komponent має тип **Control**. У компонента є метод **Ruhaius**, який задає координати **Komponent** рівними **Value** бігунка. Крім того, компонент має подію **OnRuhaius**, яка викликається в той момент, коли викликається метод **Ruhaius**.

Варіант №11.

Розробити компонент, який є спадкоємцем класу **GroupBox**. Компонент повинен мати нові властивості: **Komponent** і **Caption**. Komponent має тип **Control. Caption** – рядок символів типу **String**. У компонента є метод **Sozdanie**, який створює на контейнері компонент **Komponent** із текстом рядка **Caption.** Крім того, компонент має подію **OnSozdanie**, яка викликається в той момент, коли викликається метод **Sozdanie.**

Варіант №12.

Розробити компонент, який є спадкоємцем класу **Timer**. Компонент повинен мати нову властивість: **Komponent. Komponent** має тип **Control.** У компонента є метод **Visibles**, який робить **Komponent** видимим або невидимим через інтервал часу заданий властивістю **Interval** таймера. Крім того, компонент має подію **OnVisibles**, яка викликається в той момент, коли викликається метод **Visibles**.

ЗМІСТОВИЙ МОДУЛЬ 3. Узагальнення знань

1. СЕРІАЛІЗАЦІЯ

Серіалізацією (serialization) називається процес перетворення об'єкта чи графа пов'язаних об'єктів у потік байтів. Відповідно, зворотне перетворення називається **десеріалізацією (deserialization).** Декілька прикладів застосування цього надзвичайно корисного механізму:

- Стан програми (граф об'єкта) можна легко зберегти у файлі на диску або базі даних і відновити при наступному запуску програми **ASP.NET** зберігає та відновлює стан сеансу шляхом серіалізації та десеріалізації.
- Набір об'єктів можна скопіювати в буфер і вставити в ту чи іншу програму. Цей підхід використовується у програмах **Windows Forms** та **Windows Presentation Foundation (WPF)**.
- Можна клонувати набір об'єктів і зберегти як резервну копію, поки користувач працює з основним набором об'єктів.
- Набір об'єктів можна легко передати через мережу в процес, запущений іншою машиною. Механізм віддаленої взаємодії платформи **.NET Framework** серіалізує та десеріалізує об'єкти, що просуваються за значенням.

Крім того, після серіалізації об'єктів у потік байтів у пам'яті з'являється можливість додаткової обробки даних – наприклад, шифрування та стискування.

У **.NET Framework** існує вбудований механізм серіалізації та десеріалізації. Розробник може використовувати об'єкти до серіалізації і після десеріалізації, а всю турботу про те, що відбувається між цими двома процедурами, бере на себе **.NET Framework.** Ці процедури визначені практично для всіх типів даних, а отже, вам не доведеться робити додаткових зусиль, щоб зробити свої типи, що серіалізуються.

Припустимо, що у вас є певний об'єкт, який потрібно зберегти у файлі і який описується так:

```
public class Person
{
   public Person()  { }
   public int Age;
   public int W;
}
```

У **C#** є можливість серіалізувати члени даного екземпляра без написання будь-якого коду. Все, що вам потрібно, це включити до класу атрибут **Serializable**, а все інше зробить за вас система виконання програм **.NET**.

Коли в названу систему надходить запит на серіалізацію якогось об'єкта, то вона перевіряє, чи реалізований у класі даного об'єкта інтерфейс **ISerializable.** І якщо ні, то чи є в цьому класі атрибут **Serializable.**

Якщо в класі виявляється атрибут **Serializable,** то **.NET** використовує відображення для того, щоб отримати всі дані конкретного екземпляра — незалежно від того, чи вони описані як загальні, захищені чи приватні, і зберегти їх як подання цього об'єкта. Десеріалізація являє собою процес, зворотний описаному, дані зчитуються з носія інформації і присвоюються змінним екземплярам даного класу.

Нижче наведено приклад класу, позначеного атрибутом **Serializable:**

```
[Serializable]
public class Person
{
   public Person()
   { }
   public int Age;
   public int W;
}
```

Для того, щоб зберегти екземпляр класу **Person**, використовується об'єкт **Formatter**, який перетворює дані, що зберігаються в класі, у послідовність байтів.

Є три класи об'єктів **Formatter:**

• BinaryFormatter

• SoapFormatter

• XmlSerializer

Тип **BinaryFormatter** серіалізує стан об'єкта потік, використовуючи компактний двійковий формат. Цей тип визначено у просторі імен **System.Runtime.Serialization.Formatters.Binary**, яке входить до складання **mscorlib.dll.** Таким чином, щоб отримати доступ до цього типу, необхідно вказати наступну директиву **using**:

using System.Runtime.Serialization.Formatters.Binary;

Тип **SoapFormatter** зберігає стан об'єкта у вигляді повідомлення **SOAP** (стандартний формат **XML** для передачі та отримання повідомлень від веб-служб). Цей тип визначено у просторі імен **System.Runtime.Serialization.Formatters.Soap**, що знаходиться в окремій збірці. Тому для форматування графа об'єктів повідомлення **SOAP** необхідно

спочатку встановити посилання на **System.Runtime.Serialization.Formatters.Soap.dll**, використовуючи діалогове вікно **Add Reference** (додати посилання) у **Visual Studio** і потім вказати наступну директиву **using**:
using System.Runtime.Serialization.Formatters.Soap;

І, нарешті, для збереження дерева об'єктів у документі **XML** передбачено тип **XmlSerializer**. Щоб використовувати цей тип, потрібно вказати директиву **using** для простору імен **System.Xml.Serialization** і встановити посилання на збірку **System.Xml.dll**. Шаблони проектів **Visual Studio** автоматично посилаються на **System.Xml.dll**, тому досить просто вказати відповідний простір імен:
using System.Xml.Serialization;

1.1 Серіалізація та десеріалізація об'єктів за допомогою BinaryFormatter

Розглянемо приклад, який демонструє, як можна використовувати BinaryFormatter для збереження об'єкта класу **Person**:

```
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;



public static void Serialize()
  {
    // Створення об'єкту
    Person me = new Person();
    // Введення даних, що підлягають серіалізації
    me.Age = 20;
    me.W = 200;

    // Створення на диску файлу для зберігання даного об'єкта
    Stream s = File.Open("Me.dat", FileMode.Create);

    // Використання об'єкта для запису в потік
    BinaryFormatter bf = new BinaryFormatter();

    // Серіалізація об'єкту
    bf.Serialize(s, me);
```

```
        // Закриття потоку
        s.Close();
    }
```

Хоча зберігати об'єкти за допомогою механізму серіалізації об'єктів **.NET** досить просто, процес, що при цьому відбувається за лаштунками, досить складний. Наприклад, коли об'єкт зберігається в потоці, всі асоційовані з ним дані (тобто дані базового класу і об'єкти, що містяться в ньому) також автоматично серіалізуються. Тому, при спробі серіалізувати похідний клас, у гру вступають також усі дані щодо ланцюжка успадкування.

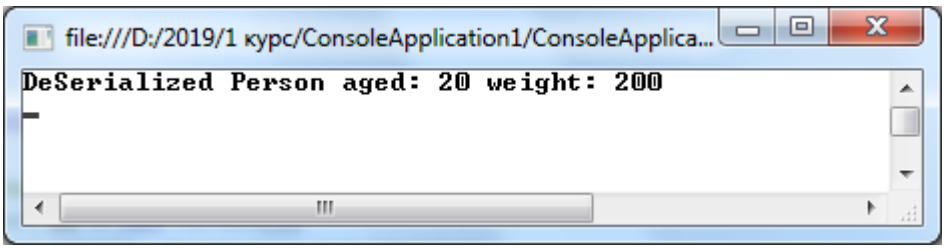
Десеріалізація, переважно, зворотня коду, наведеному вище. Наступний приклад відкриває потік у вигляді файлу **Me.dat**, створює **BinaryFormatter** для зчитування об'єкта та викликає метод **Deserialize,** що дозволяє вилучити цей об'єкт. Після цього лічені дані наводяться до типу **Person**, і на консоль виводяться вік та вага:

```
public static void DeSerialize()
    {
      // Цього разу спочатку відкривається файл
       Stream s = File.Open("Me.dat", FileMode.Open);
       // BinaryFormatted використовується для читання об'єкта (об'єктів) з потоку
       BinaryFormatter bf = new BinaryFormatter();
       // Десеріалізація об'єкту
       object o = bf.Deserialize(s);
       // Переконуємось у тому, що об'єкт потрібного нам типу
       Person p = o as Person;
       Console.WriteLine("DeSerialized Person aged: {0} weight: {1}", p.Age, p.W);
       // Закриття потоку
       s.Close();
    }
```

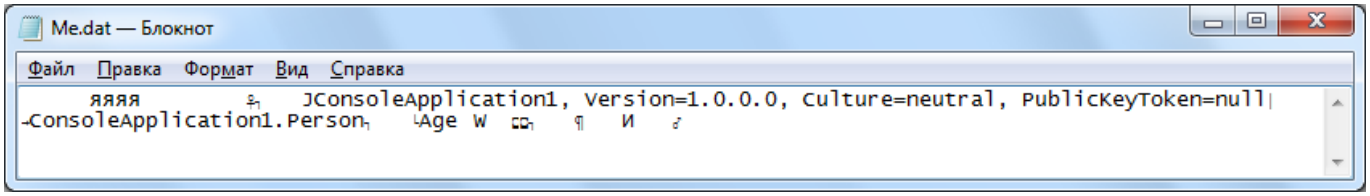
Головна функція має вигляд:

```
static void Main()
    {
       Serialize();
       DeSerialize();
       Console.ReadLine();
    }
```

*Результат***:**



Файл має вигляд:

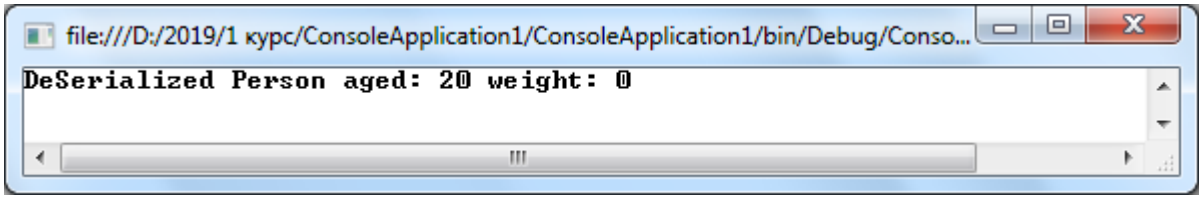


Код, який виконує серіалізацію, буде за промовчанням зберігати весь вміст об'єкта – в більшості випадків це і є необхідним. Однак за певних обставин може знадобитися описати одне або кілька полів, які не підлягають серіалізації.

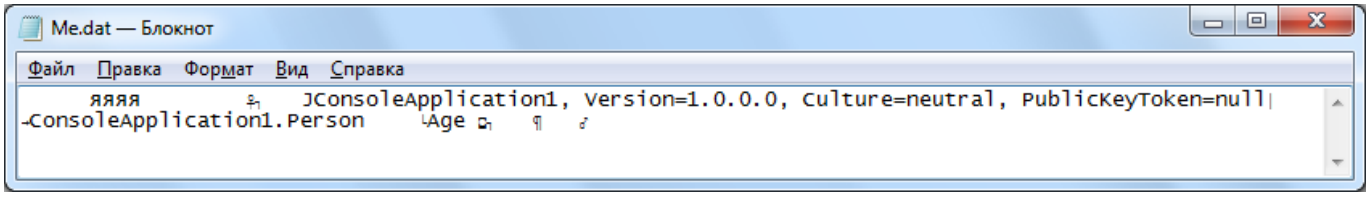
```
[Serializable]
public class Person
{
   public Person()   { }
   public int Age;
   [NonSerialized]
   public int W;
}
```

У процесі серіалізації такого класу зберігатиметься лише член **Age.** А член **W** зберігатися не буде, а отже, не буде вилучатися при десеріалізації.

Результат:



Файл має вигляд:



Зазвичай за допомогою **NonSerilized** (що не підлягають серіалізації) позначаються дані, які виходять в результаті обчислень, оскільки вони можуть бути обчислені в міру виникнення такої необхідності. Як приклад можна навести клас, у якому обчислюються прості числа; їх цілком можна зберігати для збільшення швидкості відповіді при зверненні до даного класу, проте серіалізація та десеріалізація списку простих чисел не є необхідною, оскільки вони можуть бути обчислені в момент звернення до класу. В інших випадках певний член може мати відношення лише до цього екземпляра документа. Наприклад, для об'єкта, який представляє документ, що обробляється редактором, потрібно серіалізувати вміст самого документа, але не поточну позицію, оскільки при черговому завантаженні документа в якості такої позиції просто використовується початок документа.

1.2 Серіалізація та десеріалізація об'єктів із використанням SoapFormatter

Наступний форматер, яким ми скористаємося, буде **SoapFormatter**, який серіалізує дані у відповідному конверті **SOAP**. Протокол **SOAP** (**Simple Object Access Protocol** — простий протокол доступу до об'єктів) визначає стандартний процес виклику методів незалежно від платформи та операційної системи манері.

Припускаючи, що посилання на складання **System.Runtime.Serialization.Formatters.Soap.dll** встановлено, а простір імен **System.Runtime.Serialization.Formatters.Soap** імпортовано, для збереження та вилучення Person у вигляді повідомлення **SOAP** можна просто замінити в попередньому прикладі всі входження **BinaryFormatter** на **SoapFormatter**.
Результат: аналогічний попередньому прикладу.
Файл має вигляд:

1.3 Серіалізація та десеріалізація об'єктів із використанням XmlSerializer

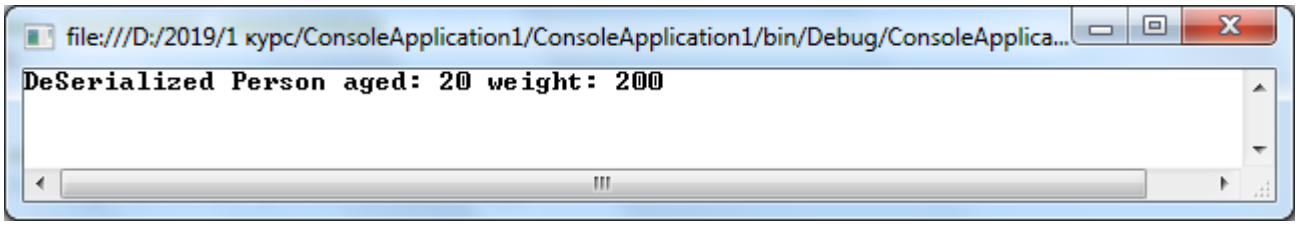
На додаток до двійкового форматора і **SOAP**, збірка **System.Xml.dll** пропонує третій клас форматера - **System.Xml.Serialization.XmlSerializer**. Цей форматор може використовуватися для збереження відкритого стану заданого об'єкта у вигляді чистої **XML**-розмітки, на противагу даним **XML** усередині повідомлення **SOAP**. Робота з цим типом дещо відрізняється від роботи з типами **SoapFormatter** чи **BinaryFormatter**. Ключова відмінність полягає в тому, що тип **XmlSerializer** вимагає вказівки інформації про тип, що представляє клас, який необхідно серіалізувати.

Розглянемо наступний код:

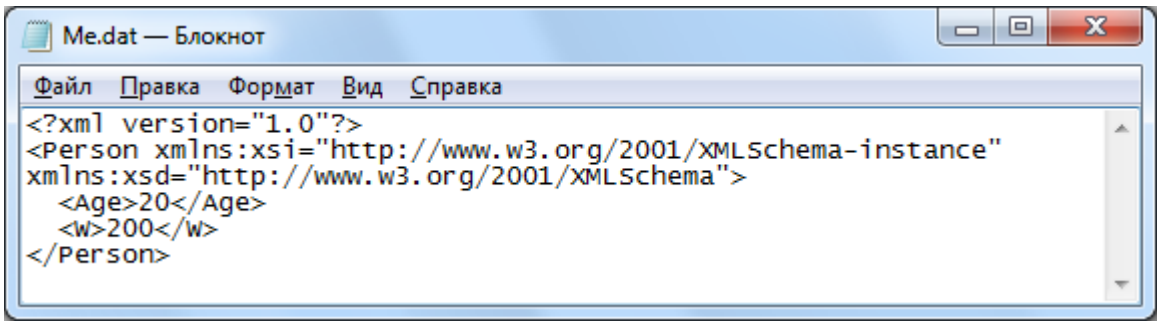
```
…
using System.Xml.Serialization;
namespace ConsoleApplication1
{ [Serializable]
   public class Person
   {
      public Person()    { }
      public int Age;
      public int W;          }
   class Demo
   {
      public static void Serialize()
      {
         // Створення об'єкту
         Person me = new Person();
         // Введення даних, що підлягають серіалізації
         me.Age = 20;
         me.W = 200;
```

```
        // Створення на диску файлу для зберігання даного об'єкта
        Stream s = File.Open("Me.dat", FileMode.Create);

        // Використання об'єкта для запису в потік
        XmlSerializer xmlFormat = new XmlSerializer(typeof(Person));

        // Серіалізація об'єкту
        xmlFormat.Serialize(s, me);

        // Закриття потоку
        s.Close();
    }
    public static void DeSerialize()
    {
        // Цього разу спочатку відкривається файл
        Stream s = File.Open("Me.dat", FileMode.Open);

        XmlSerializer xmlFormat = new XmlSerializer(typeof(Person));
        // Десеріалізація об'єкту
        object o = xmlFormat.Deserialize(s);
        // Переконуємось у тому, що об'єкт потрібного нам типу
        Person p = o as Person;

        Console.WriteLine("DeSerialized Person aged: {0} weight: {1}", p.Age,
p.W);
        // Закриття потоку
        s.Close();
    }
    public static void Main()
    {
        Serialize();
        DeSerialize();
        Console.ReadLine();
    }    }    }
```

Результат:



Файл має вигляд:



1.4 Точність типів серед форматерів

Найбільш очевидна відмінність між трьома форматорами пов'язана з тим, як граф об'єктів зберігається в потоці (двійковому, SOAP або XML). Слід також знати про деякі більш тонкі відмінності, зокрема, яким чином форматери досягають точності типів. Коли використовується тип **BinaryFormatter**, він зберігає не тільки дані полів об'єктів з графа, але також повністю задане ім'я кожного типу та повне ім'я визначальної його складання (ім'я, версія тощо). Ці додаткові елементи даних роблять **BinaryFormatter** ідеальним вибором, коли необхідно передавати об'єкти за значенням (тобто повні копії) для використання у **.NET**-застосунках.

Форматер **SoapFormatter** зберігає трасування збірок-джерел за рахунок застосування простору імен **XML**. Наприклад, якщо потрібно зберегти тип Person у повідомленні **SOAP**, ви виявите, що відкриваючий елемент Person забезпечений згенерованим параметром **xlmns**. Погляньте на таке часткове визначення, звернувши особливу увагу на простір імен **XML** під назвою **al.**

Однак, **XmlSerializer** не намагається захистити точну інформацію про тип, тому не записує його повністю задане ім'я або збірку, в якій він визначений. Хоча на перший погляд це може здатися обмеженням, причина полягає у відкритій природі представлення даних **XML**.

Якщо необхідно зберегти стан об'єкта так, щоб його можна було використовувати в середовищі будь-якої операційної системи **Windows, Mac OS X** та різних дистрибутивах **Linux**), на будь-якій платформі додатків (.**NET, Java Enterprise Edition, COM** тощо) або будь-якому мові програмування, дотримуватися повної точності типів годі було, оскільки не можна розраховувати, що це можливі адресати зможуть зрозуміти специфічні для .NET типи даних. З огляду на це **SoapFormatter** і **XmlSerializer** є ідеальним вибором, коли потрібно гарантувати якомога ширше поширення дерева

об'єктів.

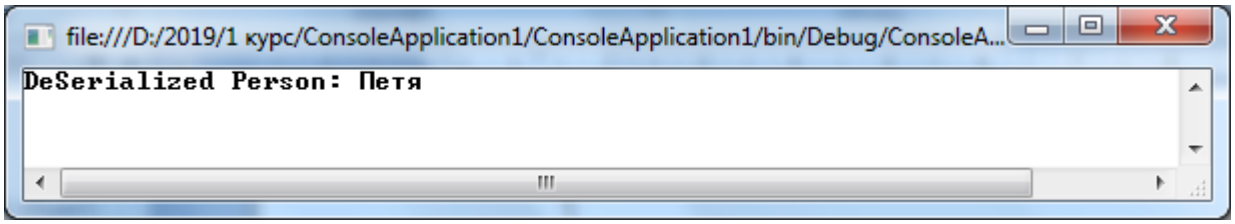
Зауваження. Однією з найважливіших відмінностей **XmlSerializer** та **BinaryFormatter** є те, що двійкова серіалізація може серіалізувати як публічні, так і приватні члени, тоді як **XmlSerializer** працює лише з публічними. Тип **XmlSerializer** серіалізуватиме лише відкриті поля даних або закриті поля, представлені відкритими властивостями. Закриті дані, які не представлені властивостями, ігноруватимуться.

Змінимо приклад:

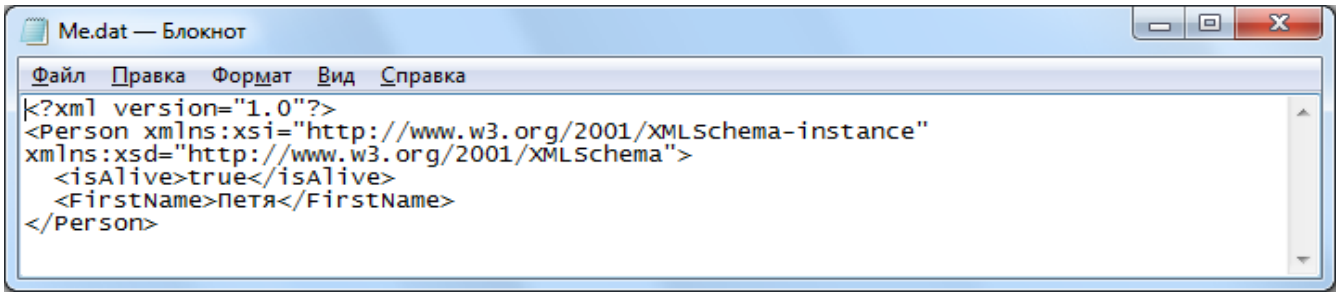
```
[Serializable]
public class Person
{
   // Открытое поле.
   public bool isAlive = true;
   // Закрытое поле.
   private int Age = 21;
   // Открытое свойство/закрытые данные.
   private string fName = string.Empty;
   public string FirstName
   {
      get { return fName; }
      set { fName = value; }
   }
}
   public static void Serialize()
   {
      // Створення об'єкту
      Person me = new Person();
      // Введення даних, що підлягають серіалізації
      me.FirstName = "Петя";

      // Створення на диску файлу для зберігання даного об'єкта
      Stream s = File.Open("Me.dat", FileMode.Create);

      // Використання об'єкта для запису в потік
      XmlSerializer xmlFormat = new XmlSerializer(typeof(Person));

      // Серіалізація об'єкту
      xmlFormat.Serialize(s, me);
```

```
        // Закриття потоку
        s.Close();
    }
    public static void DeSerialize()
    {
        // Цього разу спочатку відкривається файл
        Stream s = File.Open("Me.dat", FileMode.Open);

        XmlSerializer xmlFormat = new XmlSerializer(typeof(Person));
        // Десеріалізація об'єкту
        object o = xmlFormat.Deserialize(s);
        // Переконуємось у тому, що об'єкт потрібного нам типу
        Person p = o as Person;

        Console.WriteLine("DeSerialized Person: {0}", p.FirstName);
        // Закриття потоку
        s.Close();
    }
```

Результат:



Файл має вигляд:



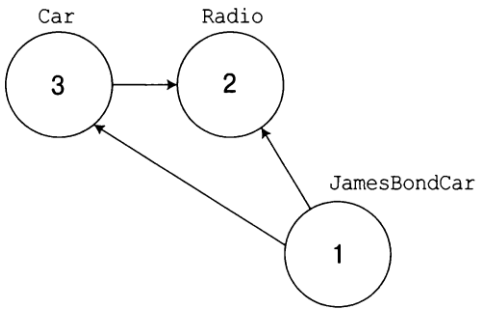
1.5 Роль графів об'єктів

Як згадувалося раніше, коли серіалізується об'єкт, середовище **CLR** враховує всі пов'язані об'єкти, щоб гарантувати коректне збереження даних. Цей набір пов'язаних об'єктів називається графом об'єктів. Графи об'єктів надають простий спосіб документування того, як між собою пов'язані елементи набору. Слід зазначити, що графи об'єктів не позначають відносини є і має

об'єктно-орієнтованого програмування. Натомість стрілки в графі об'єктів можна читати як "вимагає" або "залежить від".

Кожен об'єкт у графі набуває унікальне числове значення. Майте на увазі, що числа, призначені об'єктам у графі, є довільними та не мають жодного значення для зовнішнього світу. Після того, як всім об'єктам призначено числові значення, граф об'єктів може записувати набір залежностей кожного об'єкта.

Як простий приклад припустимо, що створено набір класів, що моделюють автомобілі. Існує базовий клас на ім'я **Car**, який має клас **Radio**. Інший клас на ім'я JamesBondCar розширює базовий тип **Car**. На малюнку показано можливий граф об'єктів, який моделює ці відносини.



При читанні графів об'єктів для опису сполучних стрілок можна використовувати вираз "залежить від" або "посилається на". Таким чином, на малюнку видно, що клас **Car** посилається на клас **Radio** (з огляду на відношення “має”), **JamesBondCar** посилається на **Car** (з огляду на відношення “є”), а також на **Radio** (оскільки успадковує цю захищену змінну-член).
Звичайно, середовище **CLR** не малює картинки у пам'яті для представлення графа пов'язаних об'єктів. Натомість відношення, документоване в попередній діаграмі, є формулою, яка виглядає приблизно так:
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]

Якщо ви проаналізуєте цю формулу, побачите, що **об'єкт 3 (Car)** має залежність від **об'єкта 2 (Radio)**. **Об'єкт 2 (Radio)** – це "самотній вовк", якому не потрібен ніхто. І, нарешті, **об'єкт 1 (JamesBondCar)** має залежність як від **об'єкта 3**, так і від **об'єкта 2**. У будь-якому випадку, при серіалізації або десеріалізації екземпляра **JamesBondCar** граф об'єктів гарантує, що типи **Radio** та **Car** також братимуть участь у процесі.

Зручність процесу серіалізації у тому, що граф, що представляє відносини між об'єктами, встановлюється автоматично, “за лаштунками”.

Зауваження. Строго кажучи, тип **XmlSerializer** не зберігає стан з використанням графа об'єктів; проте він серіалізує і десеріалізує пов'язані об'єкти в передбачуваній манері.

Для серіалізації цього графа об'єктів необхідно кожен клас позначити атрибутом **[Serializable].**

```
[Serializable]
   public class Radio
   {
      public bool hasTweeters;
      public bool hasSubWoofers;
      public double[] stationPresets;
      [NonSerialized]
      public string radioID = "XF-552RR6";
   }
[Serializable]
   public class Car
   {
      public Radio theRadio = new Radio();
      public bool isHatchBack;
   }
[Serializable]
   public class JamesBondCar : Car
   {
      public bool canFly;
      public bool canSubmerge;
   }
```

Майте на увазі, що атрибут **[Serializable]** не може успадковуватись від батьківського класу. Тому при наслідуванні типу, позначеного як **[Serializable],** дочірній клас також має бути позначений атрибутом **[Serializable]** або його не можна буде зберегти в потоці. Насправді всі об'єкти у графі об'єктів мають бути позначені атрибутом **[Serializable].** Спроба серіалізувати об'єкт, що не

серіалізується, з використанням **BinaryFormatter** або **SoapFormatter** призводить до генерації винятку **SerializationException** під час виконання.

Зверніть увагу, що у кожному з цих класів поля даних визначено як public, це зроблено спрощення прикладу. Звичайно, закриті дані, представлені відкритими властивостями, були б кращими з точки зору об'єктно-орієнтованого програмування. Також для простоти в цих типах не визначилися ніякі спеціальні конструктори, і тому всі неініціалізовані поля даних отримають очікувані стандартні значення.

Зауваження. Клас **XmlSerializer** вимагає, щоб усі серіалізовані типи в графі об'єктів підтримували стандартний конструктор (тому не забудьте додати його, якщо визначали спеціальні конструктори). Якщо цього не зробити, під час виконання згенерується виняток **InvalidOperationException.**

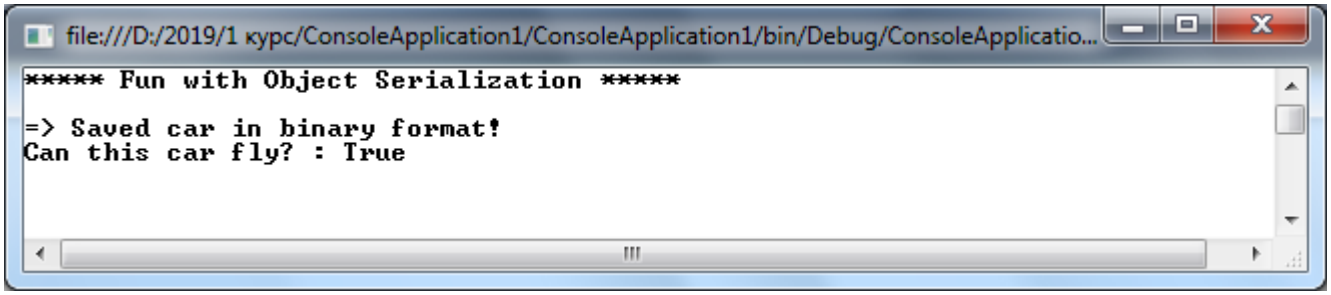
1.6 Серіалізація та десеріалізація графа об'єктів за допомогою BinaryFormatter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Runtime.Serialization.Formatters.Binary;
using System.IO;

namespace ConsoleApplication1
{
   [Serializable]
   public class Radio
   {
      public bool hasTweeters;
      public bool hasSubWoofers;
      public double[] stationPresets;
      [NonSerialized]
      public string radioID = "XF-552RR6";
   }
   [Serializable]
   public class Car
   {
      public Radio theRadio = new Radio();
```

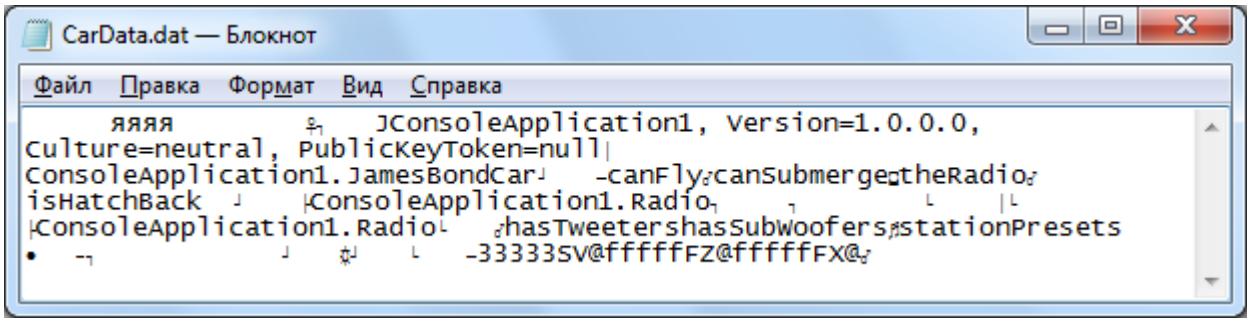
```
    public bool isHatchBack;
}
[Serializable]
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
class Demo
{
    static void SaveAsBinaryFormat(object objGraph, string fileName)
    {
        // Зберегти об'єкт у файл CarData.dat у двійковому вигляді.
        BinaryFormatter binFormat = new BinaryFormatter();
        // створення тимчасового потоку
        using (Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
        {
            binFormat.Serialize(fStream, objGraph);
        }
        Console.WriteLine("=> Saved car in binary format!");
    }
    static void LoadFromBinaryFile(string fileName)
    {
        BinaryFormatter binFormat = new BinaryFormatter();
        // Прочитати JamesBondCar з двійкового файлу.
        // створення тимчасового потоку
        using (Stream fStream = File.OpenRead(fileName))
        {
            JamesBondCar carFromDisk =
            (JamesBondCar)binFormat.Deserialize(fStream);
            Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
        }
    }
    public static void Main()
    {
        Console.WriteLine("***** Fun with Object Serialization *****\n");
        // Створити JamesBondCar та встановити стан.
        JamesBondCar jbc = new JamesBondCar();
```

```
            jbc.canFly = true;
            jbc.canSubmerge = false;
            jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };
            jbc.theRadio.hasTweeters = true;
            // Зберегти об'єкт у зазначеному файлі у двійковому форматі.
            SaveAsBinaryFormat(jbc, "CarData.dat");
            LoadFromBinaryFile("CarData.dat");
            Console.ReadLine();
        }
    }
}
```

Результат:



Файл має вигляд:

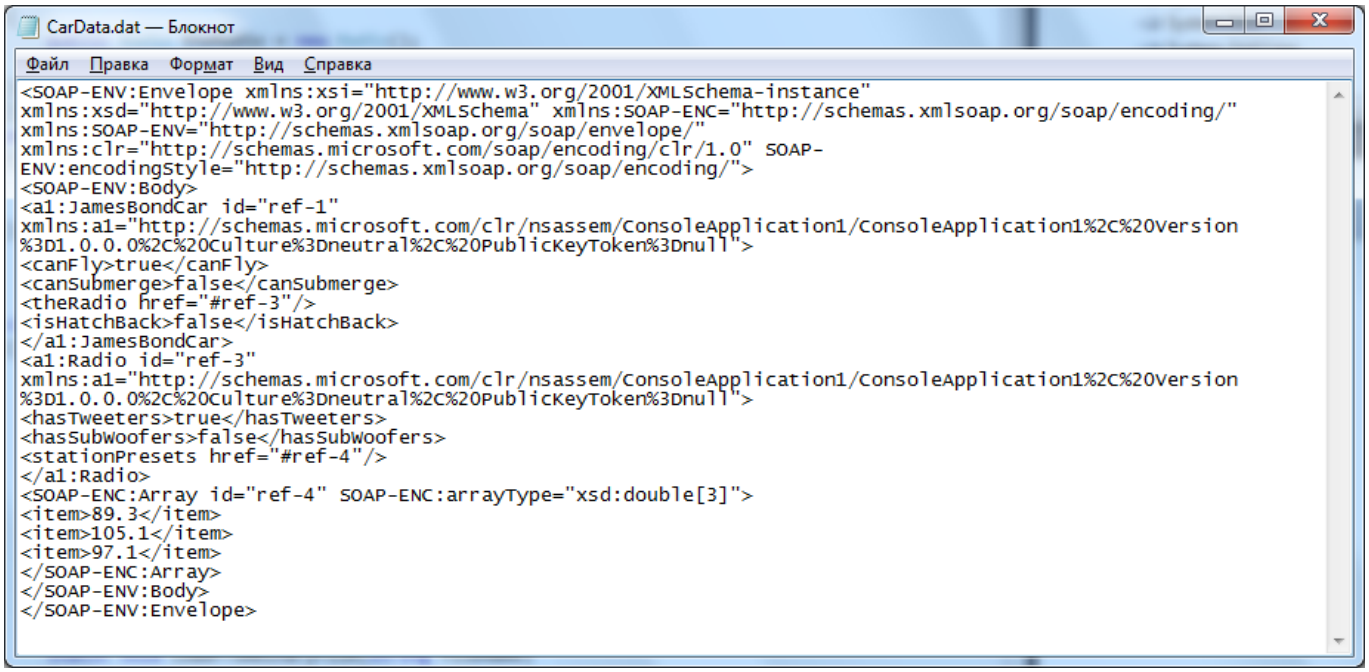


1.7 Серіалізація та десеріалізація графа об'єктів із використанням SoapFormatter

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
using System.Runtime.Serialization.Formatters.Soap;
//using System.Xml.Serialization;

namespace ConsoleApplication1
{
   [Serializable]
   public class Radio
   {
      public bool hasTweeters;
      public bool hasSubWoofers;
      public double[] stationPresets;
      [NonSerialized]
      public string radioID = "XF-552RR6";
   }

   [Serializable]
   public class Car
   {
      public Radio theRadio = new Radio();
      public bool isHatchBack;
   }
   [Serializable]
   public class JamesBondCar : Car
   {
      public bool canFly;
      public bool canSubmerge;
   }
```

```
class Demo
 {
   static void SaveAsBinaryFormat(object objGraph, string fileName)
   {
     // Зберегти об'єкт у файл CarData.dat у двійковому вигляді.
     SoapFormatter binFormat = new SoapFormatter();
     // створення тимчасового потоку
     using (Stream fStream = new FileStream(fileName,
     FileMode.Create, FileAccess.Write, FileShare.None))
     {     binFormat.Serialize(fStream, objGraph);          }
     Console.WriteLine("=> Saved car in binary format!");
   }
   static void LoadFromBinaryFile(string fileName)
   {
     SoapFormatter binFormat = new SoapFormatter();
     // Прочитати JamesBondCar з двійкового файлу.
     // створення тимчасового потоку
     using (Stream fStream = File.OpenRead(fileName))
     {
       JamesBondCar carFromDisk =
       (JamesBondCar)binFormat.Deserialize(fStream);
       Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
     }
   }
   public static void Main()
   {
     Console.WriteLine("***** Fun with Object Serialization *****\n");
     // Створити JamesBondCar та встановити стан.
     JamesBondCar jbc = new JamesBondCar();
     jbc.canFly = true;
     jbc.canSubmerge = false;
     jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };
     jbc.theRadio.hasTweeters = true;
     // Зберегти об'єкт у зазначеному файлі у двійковому форматі.
     SaveAsBinaryFormat(jbc, "CarData.dat");
     LoadFromBinaryFile("CarData.dat");
     Console.ReadLine();          }
 }
}
```

Файл має вигляд:



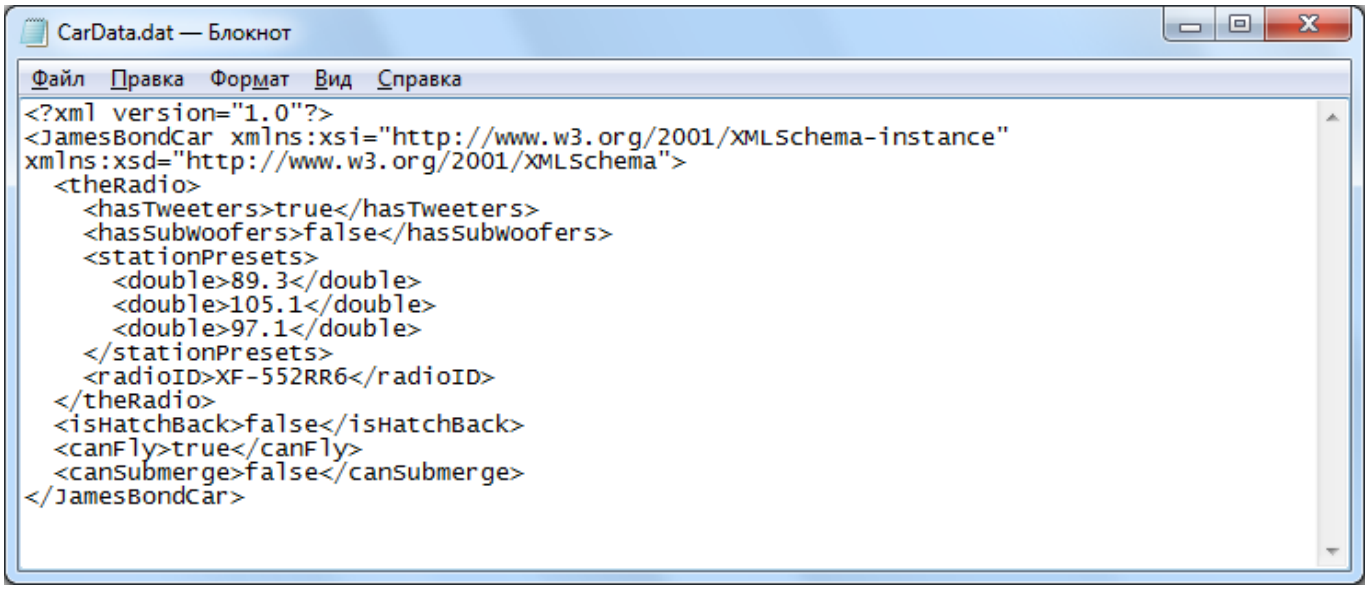
1.8 Серіалізація та десеріалізація графа об'єктів із використанням XmlSerializer

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
//using System.Runtime.Serialization.Formatters.Binary;
using System.IO;
//using System.Runtime.Serialization.Formatters.Soap;
using System.Xml.Serialization;

namespace ConsoleApplication1
{
  [Serializable]
  public class Radio
  {
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;
    [NonSerialized]
    public string radioID = "XF-552RR6";
  }
```

```
[Serializable]
public class Car
{
   public Radio theRadio = new Radio();
   public bool isHatchBack;
}
[Serializable]
public class JamesBondCar : Car
{
   public bool canFly;
   public bool canSubmerge;
}
class Demo
{
   static void SaveAsBinaryFormat(object objGraph, string fileName)
   {
      // Зберегти об'єкт у файлі CarData.xm3. у форматі XML.
      XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar), new
Type[] { typeof(Radio), typeof(Car) });
      // створення тимчасового потоку
      using (Stream fStream = new FileStream(fileName,
      FileMode.Create, FileAccess.Write, FileShare.None))
      {   xmlFormat.Serialize(fStream, objGraph);          }
      Console.WriteLine("=> Saved car in XML format!");
   }
   static void LoadFromBinaryFile(string fileName)
   {
      XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar), new
Type[] { typeof(Radio), typeof(Car) });
      // Прочитати JamesBondCar з двійкового файлу.
      // створення тимчасового потоку
      using (Stream fStream = File.OpenRead(fileName))
      {
         JamesBondCar carFromDisk =
         (JamesBondCar)xmlFormat.Deserialize(fStream);
         Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
      }
   }
```

```
        public static void Main()
        {
            Console.WriteLine("***** Fun with Object Serialization *****\n");
            // Створити JamesBondCar та встановити стан.
            JamesBondCar jbc = new JamesBondCar();
            jbc.canFly = true;
            jbc.canSubmerge = false;
            jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };
            jbc.theRadio.hasTweeters = true;
            // Зберегти об'єкт у зазначеному файлі у двійковому форматі.
            SaveAsBinaryFormat(jbc, "CarData.dat");
            LoadFromBinaryFile("CarData.dat");
            Console.ReadLine();
        }
    }
}
```

Файл має вигляд:



1.9 Серіалізація колекцій об'єктів

Тепер, коли ви знаєте, як зберігати єдиний об'єкт у потоці, давайте подивимося, як можна зберегти безліч об'єктів. Як ви, можливо, помітили, метод **Serialize()** інтерфейсу **IFormatter** не надає способу вказати довільну кількість об'єктів як введення (допускається лише єдиний об'єкт **System.Object**). До того ж значення **Deserialize(),** що повертається, також являє собою одиночний об'єкт **System.Object** (те ж базове обмеження стосується і **XmlSerializer**):

```
public interface IFormatter
   {
      object Deserialize(Stream serializationStream);
      void Serialize(Stream serializationStream, object graph);
   }
```

Згадайте, що **System.Object** представляє ціле дерево об'єктів. З урахуванням цього, якщо передати об'єкт, який позначений атрибутом [Serializable] і містить інші об'єкти [Serializable], то за допомогою єдиного виклику даного методу буде зберігатися весь набір об'єктів. На щастя, більшість типів із просторів імен System.Collections і System.Collections.Generic вже позначені атрибутом [Serializable]. Таким чином, щоб зберегти безліч об'єктів, просто додайте цю безліч до контейнера (такого як звичайний масив, ArrayList або List<T>) і серіалізуйте даний об'єкт у вибраний потік.

Повернемося, наприклад, з бібліотекою. Об'єкт бібліотеки містить у собі список книг. Позначимо атрибутом [Serializable] класи Book та LibraryClass.

Додамо для форми подію функцію закриття, в якій проведемо серіалізацію об'єкта бібліотеки:

```
private void Form1_FormClosing(object sender, FormClosingEventArgs e)
      {
         // Створення на диску файлу для зберігання даного об'єкта
         Stream s = File.Open("Me.dat", FileMode.Create);
         // Використання об'єкта для запису в потік
         BinaryFormatter bf = new BinaryFormatter();
         // Серіалізація об'єкту
         bf.Serialize(s, myLibrary);
         // Закриття потоку
         s.Close();
      }
```

Змінимо код подієвої функції завантаження форми, додамо код десеріалізації бібліотеки:

```
private void Form1_Load(object sender, EventArgs e)
      {
     // тут має бути десеріалізація моєї бібліотеки
         if (File.Exists("Me.dat"))
         {
            Stream s = File.Open("Me.dat", FileMode.Open);
     // BinaryFormatted використовується для читання об'єкта (об'єктів) з потоку
```

```
            BinaryFormatter bf = new BinaryFormatter();
            // Десеріалізація об'єкту
            object o = bf.Deserialize(s);
            //Убеждаемся в том, что объект нужного нам типа
            myLibrary = o as LibraryClass;
            // Закриття потоку
            s.Close();
        }
        else
            myLibrary = new LibraryClass();
        if (myLibrary.Books.Count == 0)
        {
            NullBook();
            emptyLibrary = true;
        }
        LoadSource();
    }
```

2. UML-ДІАГРАМИ

2.1 Діаграми класів

UML (уніфікована мова моделювання Unified Modeling Language) – це мова графічного опису для об'єктного моделювання в галузі розробки програмного забезпечення.

Відносини на рівні класів.

Успадкування:

Імплементація: Реалізація інтерфейсу, реалізація методів, тобто. те, що фактично працює у класі.

Відносини на рівні об'єктів.

Асоціація: означає відношення між класами об'єктів, яке дозволяє одному екземпляру об'єкта викликати іншого, щоб виконати дію від його імені.

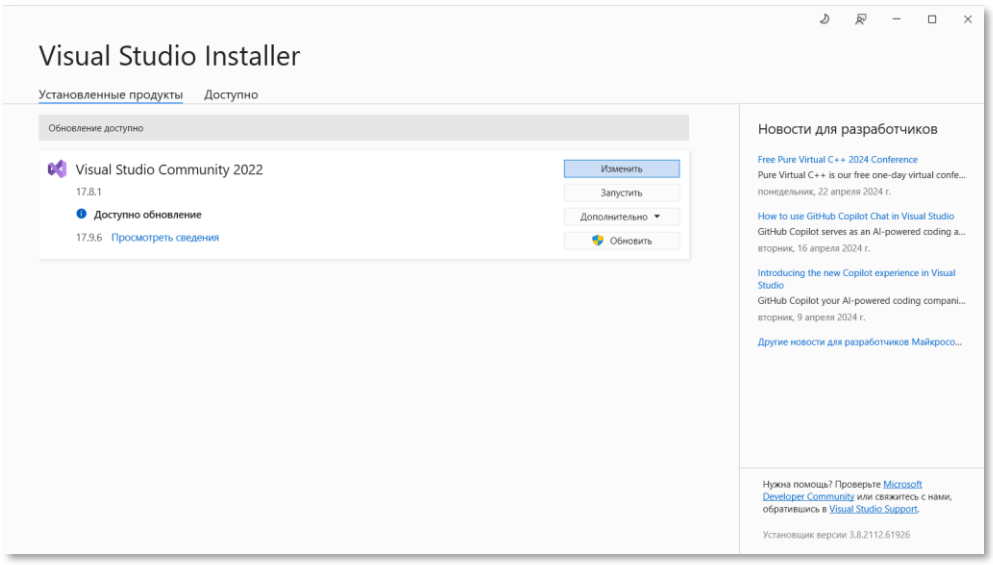
Агрегація: Підтип асоціації. Наприклад, один клас містить (агрегують) об'єкти іншого класу.

Композиція: Схоже на агрегацію лише сильніший зв'язок. Тому зафарбований ромб. Наприклад: якщо знищується композитор, його об'єкти класів, куди він посилається, також перестають існувати.

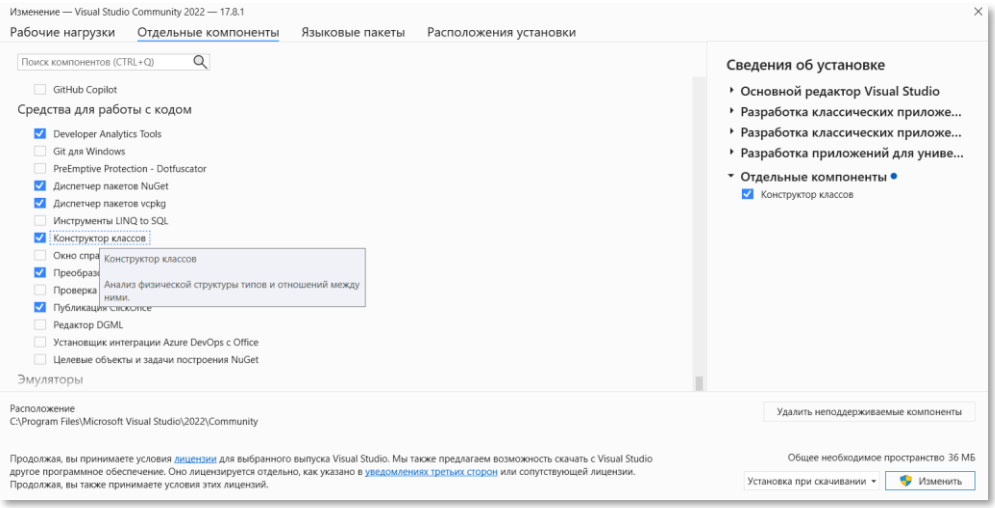
Просто ставлення.

Залежність: Класи "яким чином" залежать один від одного. Наприклад, якщо один клас змінюються методи, конструктори чи поля, і тому доводиться переписувати інший клас, отже вони залежні. Одна з найслабших зв'язків. Наприклад, об'єкти одного класу передаються як параметр методи іншого класу і т.д.

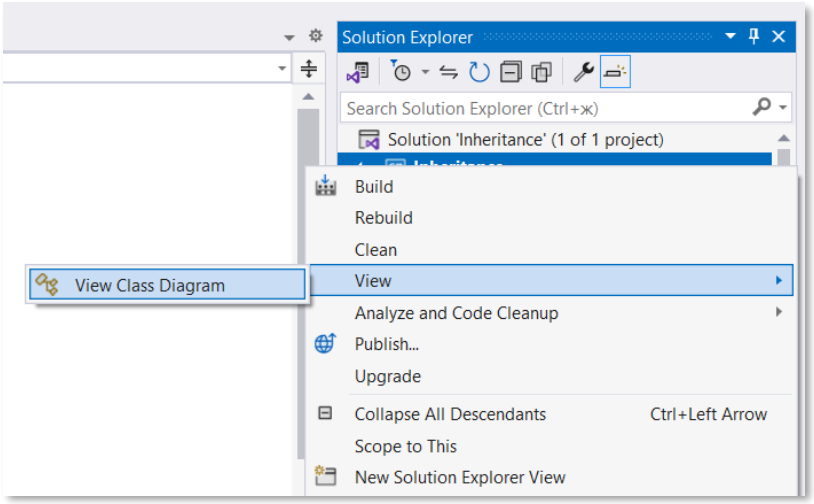
Перш ніж перейти до створення діаграми перевірте чи всі компоненти встановлені у Visual Studio Installer. Для перевірки або встановлення натисніть на кнопку «***Змінити»***

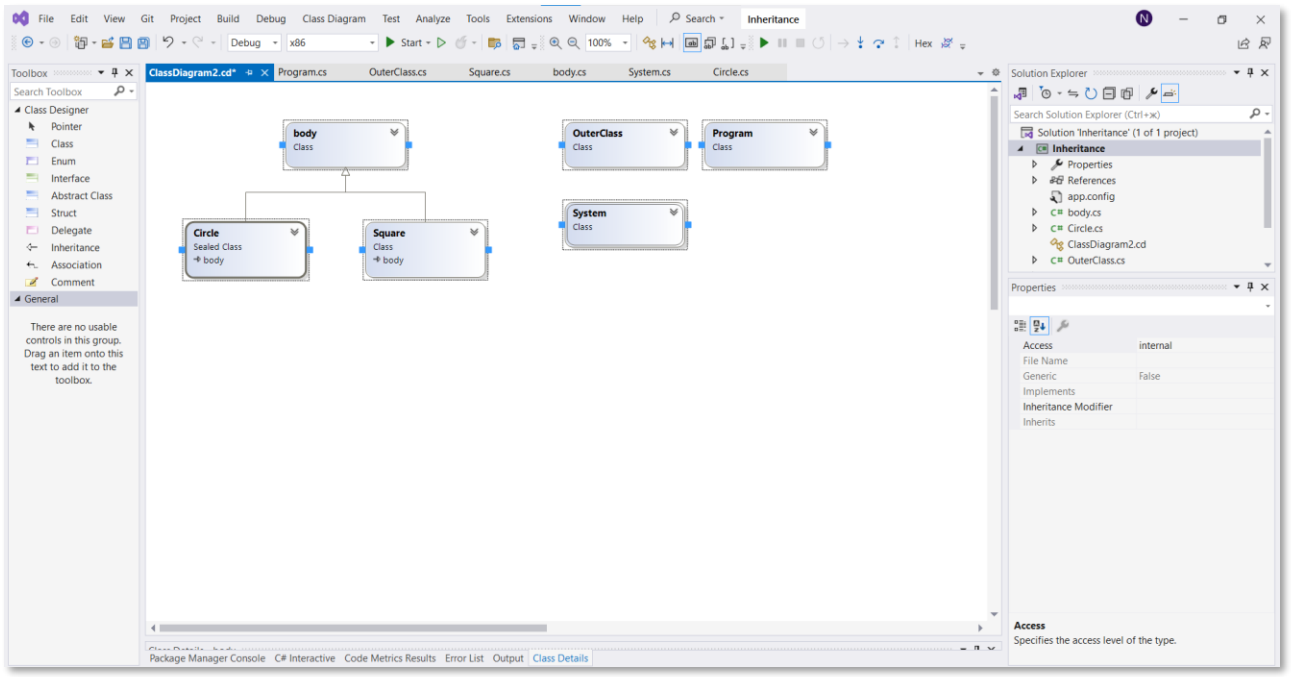


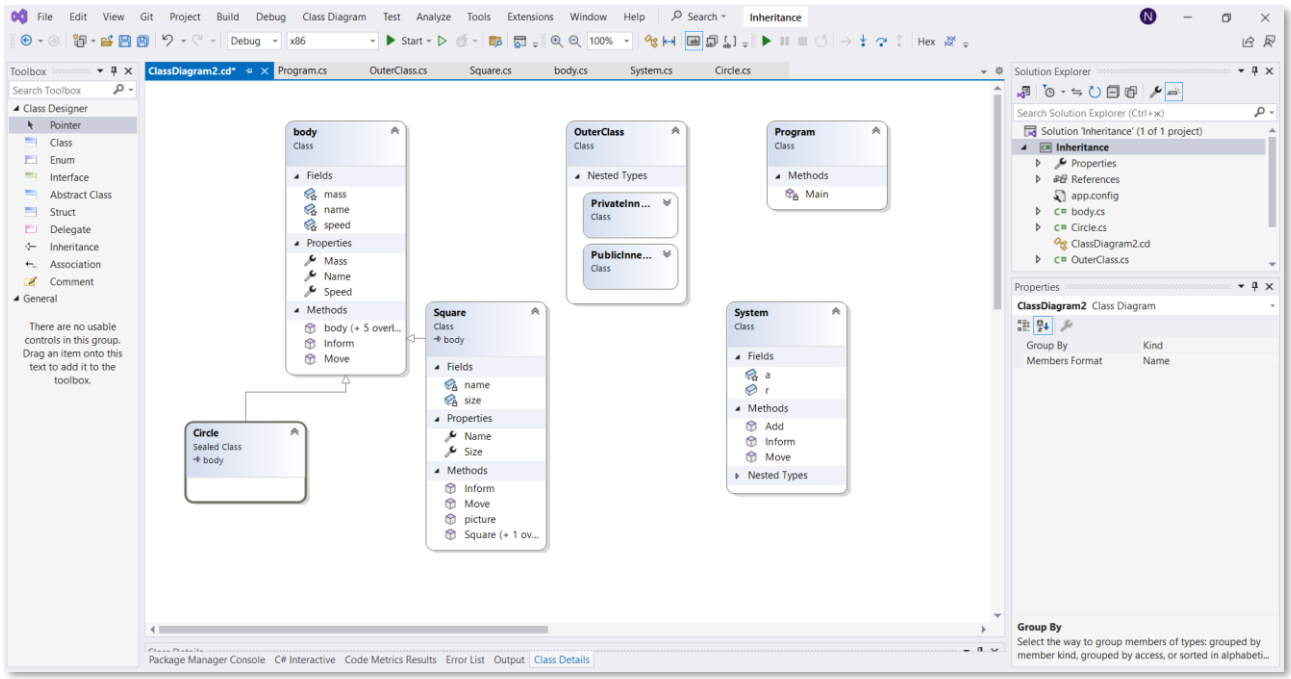
Перейдіть на вкладку ***«Окремі компоненти»(*** **Individual components**). Знайдіть групу **Code Tools** та оберіть **Class Designer** (якщо встановлено прапорець, то нічого робити не треба). Натисніть ***«Змінити»***



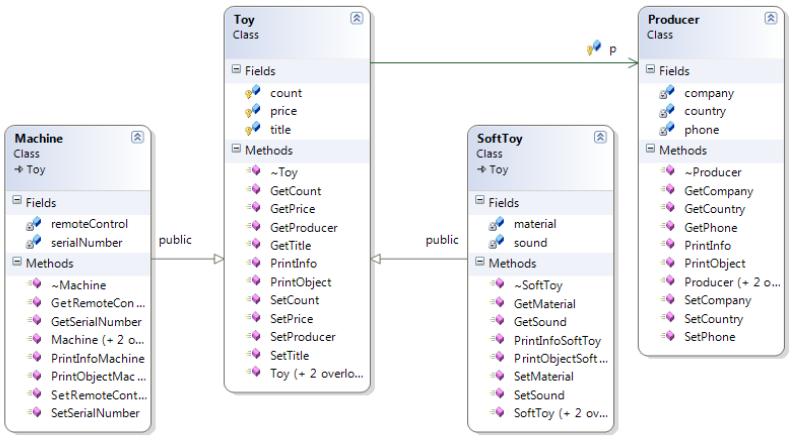
Після цих дій Ви зможете створювати UML-діаграми. Для цього оберіть необхідний проект в Solution Explorer та натисніть праву кнопку миші:




File Edit View Git Project Build Debug Class Diagram Test Analyze Tools Extensions Window Help
Search
Inheritance
Debug
x86
Start
100%
Hex
Toolbox
Search Toolbox
Class Designer
Pointer
Class
Enum
Interface
Abstract Class
Struct
Delegate
Inheritance
Association
Comment
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
ClassDiagram2.cd*
Program.cs
OuterClass.cs
Square.cs
body.cs
System.cs
Circle.cs
body
Class
OuterClass
Class
Program
Class
System
Class
Circle
Sealed Class
body
Square
Class
body
Solution Explorer
Search Solution Explorer (Ctrl+ж)
Solution 'Inheritance' (1 of 1 project)
Inheritance
Properties
References
app.config
body.cs
Circle.cs
ClassDiagram2.cd
OuterClass.cs
Properties
Access
internal
File Name
Generic
False
Implements
Inheritance Modifier
Inherits
Access
Specifies the access level of the type.
Package Manager Console
C# Interactive
Code Metrics Results
Error List
Output
Class Details


File Edit View Git Project Build Debug Class Diagram Test Analyze Tools Extensions Window Help
Search
Inheritance
Debug
x86
Start
100%
Hex
Toolbox
Search Toolbox
Class Designer
Pointer
Class
Enum
Interface
Abstract Class
Struct
Delegate
Inheritance
Association
Comment
General
There are no usable controls in this group. Drag an item onto this text to add it to the toolbox.
ClassDiagram2.cd*
Program.cs
OuterClass.cs
Square.cs
body.cs
System.cs
Circle.cs
body
Class
Fields
mass
name
speed
Properties
Mass
Name
Speed
Methods
body (+ 5 overl...
Inform
Move
OuterClass
Class
Nested Types
PrivateInn...
Class
PublicInne...
Class
Program
Class
Methods
Main
Square
Class
body
Fields
name
size
Properties
Name
Size
Methods
Inform
Move
picture
Square (+ 1 ov...
System
Class
Fields
a
r
Methods
Add
Inform
Move
Nested Types
Circle
Sealed Class
body
Solution Explorer
Search Solution Explorer (Ctrl+ж)
Solution 'Inheritance' (1 of 1 project)
Inheritance
Properties
References
app.config
body.cs
Circle.cs
ClassDiagram2.cd
OuterClass.cs
Properties
ClassDiagram2 Class Diagram
Group By
Kind
Members Format
Name
Group By
Select the way to group members of types: grouped by member kind, grouped by access, or sorted in alphabeti...
Package Manager Console
C# Interactive
Code Metrics Results
Error List
Output
Class Details

Приклад


Toy
Class
Fields
count
price
title
Methods
~Toy
GetCount
GetPrice
GetProducer
GetTitle
PrintInfo
PrintObject
SetCount
SetPrice
SetProducer
SetTitle
Toy (+ 2 overlo...
p
Producer
Class
Fields
company
country
phone
Methods
~Producer
GetCompany
GetCountry
GetPhone
PrintInfo
PrintObject
Producer (+ 2 o...
SetCompany
SetCountry
SetPhone
Machine
Class
Toy
Fields
remoteControl
serialNumber
Methods
~Machine
GetRemoteCon...
GetSerialNumber
Machine (+ 2 o...
PrintInfoMachine
PrintObjectMac...
SetRemoteCont...
SetSerialNumber
public
SoftToy
Class
Toy
Fields
material
sound
Methods
~SoftToy
GetMaterial
GetSound
PrintInfoSoftToy
PrintObjectSoft...
SetMaterial
SetSound
SoftToy (+ 2 ov...
public

СПИСОК РЕКОМЕНДОВАНОЇ ЛІТЕРАТУРИ

Основна

1. Бублик В.В. Об'єктно-орієнтоване програмування: підручник. Київ: ІТ-книга, 2015. 624 с.
2. Кравець П.О. Об'єктно-орієнтоване програмування: навч. посіб. Львів: Видавництво Львівської політехніки, 2012. 624 с.
3. Дібрівний О.А., Гребенюк В.В. Вступ до об’єктно-орієнтованого програмування C#: навч. посіб. Київ: Державний університет телекомунікацій, 2018. 190 с.
4. Коноваленко І.В. Програмування мовою C# 6.0: навч. посіб. Тернопіль: ТНТУ. 2016. 227 с.
5. Бурлаков А.А. Об’єктно-орієнтований аналіз і проектування: метод. реком. Хмельницький: ХНУ, 2017. 136 с.
6. Daniel Solis. Illustrated C# 2012. Berkeley: APress, 2012. 732 p.
7. Standard ECMA-334. C# Language Specification. 4th Edition. Ecma International. June 2006 URL: https://www.ecma-international.org/wp-content/uploads/ECMA-334_4th_edition_june_2006.pdf
8. C# Reference, сайт розробників MSDN. URL: https://code.visualstudio.com/docs/languages/csharp (дата звернення 11.02.2023)

Додаткова

9. International standard ISO/IEC 23270:2006. Information technology – Programming languages. C#. Second edition. ISO/IEC. 2006.
10. The C# Programming Language / A. Hejlsberg et al. Boston: Addison-Wesley Professional, 2010. 864 p.
11. Albahari J., Albahari B. C# 7.0 in a Nutshell. Sebastopol: O'Reilly Media, 2017. 1090 p.
12. Schildt H. C# 4.0. The Complete Reference. New York:McGraw Hill Education, 2017. 984 p.
13. Sharp J. Microsoft Visual C# Step by Step. London:Pearson Education, 2018. 878p.
14. Troelsen A., Pro C# 7. With .NET and .NET Core. Berkeley: Apress, 2017. 1372 p.

Навчальне видання

ОБ'ЄКТНО-ОРІЄНТОВАНЕ ПРОГРАМУВАННЯ

ЕЛЕКТРОННИЙ МЕТОДИЧНИЙ ПОСІБНИК

до лекційних та лабораторних занять
для студентів факультету математики,
фізики та інформаційних технологій
першого (бакалаврського) рівня освіти
спеціальності 122 Комп'ютерні науки

Електронне практичне видання

Укладачі:

Рачинська Алла Леонідівна

Недєва Ольга Анатоліївна

Палій Катерина Сергіївна

В авторській редакції

Затвердж. авт. 16.04.2023. Шрифт Times New Roman.
Системні вимоги: операційна система сумісна з програмним забезпеченням
для читання файлів формату PDF.
Обсяг 11,4 МБ. Зам. № 2785.

Видавець і виготовлювач
Одеський національний університет імені І. І. Мечникова
Свідоцтво суб'єкта видавничої справи ДК № 4215 від 22.11.2011 р.
65082, м. Одеса, вул. Єлісаветинська, 12, Україна
Тел.: (048) 723 28 39, e-mail: druk@onu.edu.ua