МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ ІМЕНІ СЕМЕНА КУЗНЕЦЯ

Методичні рекомендації
до виконання лабораторних робіт
з навчальної дисципліни

"ПРОГРАМУВАННЯ ЗАСОБІВ МУЛЬТИМЕДІА",

розділ "Мультимедійні об'єктно-орієнтовані додатки"
для студентів напряму підготовки
6.051501 "Видавничо-поліграфічна справа"
денної форми навчання

Харків
ХНЕУ ім. С. Кузнеця
2016

Затверджено на засіданні кафедри комп'ютерних систем і технологій. Протокол № 12 від 07.04.2016 р.

Укладач В. В. Браткевич

М 54 **Методичні** рекомендації до виконання лабораторних робіт з навчальної дисципліни "Програмування засобів мультимедіа", розділ "Мультимедійні об'єктно-орієнтовані додатки" для студентів напряму підготовки 6.051501 "Видавничо-поліграфічна справа" денної форми навчання / уклад. В. В. Браткевич. – Харків : ХНЕУ ім. С. Кузнеця, 2016. – 132 с.

Подано методичні рекомендації до виконання лабораторних робіт з розроблення типових об'єктно-орієнтованих програм мовою C#. Надано основний навчальний матеріал, наведено порядок виконання лабораторних робіт, запропоновано структуру контенту, який необхідно включати у звіт. В якості середовища розроблення обрано Microsoft Visual Studio .NET.

Рекомендовано для студентів напряму підготовки 6.051501 "Видавничо-поліграфічна справа" денної форми навчання.

Вступ

Навчальна дисципліна "Програмування засобів мультимедіа" вивчається студентами напряму підготовки 6.051501 "Видавничо-поліграфічна справа" денної форми навчання протягом третього і четвертого семестрів.

Метою викладання навчальної дисципліни "Програмування засобів мультимедіа" є:

формування у студентів системи теоретичних знань і прикладних умінь в області застосування сучасних об'єктно-орієнтованих мов програмування для інструментальної підтримки технологічного процесу виробництва видавничо-поліграфічних та мультимедійних продуктів;

підготовка студентів до самостійного освоєння вмонтованих об'єктно-орієнтованих програмних засобів (скриптів) сучасних серед розробки мультимедіа і Web-дизайну.

Основними завданнями вивчення навчальної дисципліни "Програмування засобів мультимедіа" є оволодіння навичками в області застосування:

сучасних об'єктно-орієнтованих мов програмування для інструментальної підтримки технологічного процесу виробництва видавничо-поліграфічних та мультимедійних продуктів;

сучасних базових інструментальних програмних засобів, призначених для створювання та налагодження об'єктно-орієнтованих додатків.

Предметом навчальної дисципліни є: основні принципи об'єктно-орієнтованого програмування; зміст і структура типових бібліотек класів і шаблонів; технологія створення об'єктно-орієнтованих додатків.

Результатами виконання лабораторних робіт є засвоєння студентами особливостей:

організації інтерфейсу Designer Forms Microsoft Visual Studio .NET і оволодіння навичками його застосування для розробки та налагодження об'єктно-орієнтованих додатків;

структури типових бібліотек класів та шаблонів;

принципів створення і перетворення графічних зображень.

Студенти оволодіють навичками складати та налагоджувати відповідні програми мовою C#, які забезпечують:

визначення і дослідження найпростіших класів;

розроблення програм з ієрархією класів;

вивчення особливостей застосування конструкторів;

розроблення і дослідження простих методів;

розроблення програм з ієрархією класів;

застосування делегатів для оброблення подій;

розроблення і дослідження типового каркасу графічного Windows-додатка;

розроблення і дослідження Windows-додатків з елементами анімаційної графіки.

Розділ 2. Мультимедійні об'єктно-орієнтовані додатки

Змістовий модуль 3
Базові концепції об'єктно-орієнтованого програмування

Лабораторна робота 8
Використання простих класів

Мета роботи – отримати практичні навички розроблення програм із простішими класами.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

основних концепцій об'єктно-орієнтованого програмування;

загальної форми визначення класу;

правил створення об'єктів;

особливостей застосування змінних типу посилання;

типової структури C # - програми з кількома класами;

уміння:

розробити програму, що містить клас (класи), який інкапсулює інформацію про заданий об'єкт (предметна область задається індивідуально);

розробити програму, що обробляє інформацію (елементарні обчислення і виведення на екран вмісту елементів-даних) про декілька об'єктів одного класу;

комунікації:

аргументована взаємодія з клієнтами та замовниками під час вибору парадигми проектування, що забезпечує здатність мислити в термінах об'єктно-орієнтованої моделі і базується на знанні сучасних інформаційних технологій та вмінні використовувати інтернет-ресурси;

автономність і відповідальність:

самостійне формулювання рекомендацій щодо декомпозиції проекту на сукупність окремих блоків;

здатність обґрунтувати структуру класу та доцільність застосування певних типів даних для його опису.

Основні положення

Усі C# -програми оформлюються у вигляді класу. **Клас** – це шаблон, який визначає форму об'єкта. Він задає як дані, так і код, який оперує цими даними.

Об'єкти – це екземпляри класу. Важливо чітко розуміти, що клас – це логічна абстракція. Про її реалізацію немає сенсу говорити доти, поки не буде створено об'єкт класу, а в пам'яті не з'явиться фізичне його подання.

Методи і змінні, складові класу, називаються елементами (членами) класу.

Загальна форма визначення класу.

Незважаючи на те, що дуже прості класи можуть включати тільки код або тільки дані, більшість реальних класів містять те й інше. Дані містяться в змінних екземплярах, що визначаються класом, а код – у методах. Треба зазначити, що клас визначає також ряд спеціальних елементів – даних і елементів – методів, наприклад статичні змінні, константи, конструктори, деструктори, індексатори, події, оператори та властивості. Поки ми обмежимося розглядом змінних екземплярів і методів класу, а потім ознайомимося з конструкторами і деструкторами. Клас створюється за допомогою ключового слова class. Загальна форма визначення класу, який містить тільки змінні екземплярів і методи, має такий вигляд:

```
class ім'я_класу
{
// Оголошення змінних примірників.
        доступ тип зміна1;
        доступ тип зміна2;
        //. . .
// Оголошення методів.
        доступ тип_що повертається ім'я_методу1 (параметри)
        {
                // Тіло методу
        }
        доступ тип_що повертається ім'я_методу2 (параметри)
        {
                // Тіло методу
        }
        //. . .
}
```

Класи, які ми використовували досі у навчальній дисципліні "Основи програмування", містили тільки один метод – Main ( ). Незабаром ми дізнаємося, як створювати й інші методи. Слід зазначити, що в загальній формі визначення класу метод Main ( ) не заданий. Він потрібен тільки в тому випадку, якщо клас, що створюється, є відправною точкою програми.

Приклад визначення класу.

Створимо клас, який інкапсулює інформацію про будівлі (будинки, складські приміщеннях, офіси і т. д.). У цьому класі (назвемо його Building) будуть зберігатися три елементи інформації про будівлі (кількість поверхів, загальна площа і кількість мешканців).

Далі подано першу версію класу Building. У ньому визначено три змінні екземпляра: floors, area і occupants. Зверніть увагу на те, що клас Building не містить жодного методу. Тому поки його можна вважати класом даних. (У наступних лабораторних роботах ми доповнимо його методами).

```
class Building
{
        public int floors;          // кількість поверхів
        public int area;            // загальна площа основи будівлі
        public int occupants;       // кількість мешканців
}
```

Змінні екземпляру, які визначені в класі Building, ілюструють загальний спосіб їх оголошення. Формат оголошення змінної примірника такий:

доступ тип ім'я_змінної;

Тут елемент "доступ" представляє специфікатор доступу, елемент "тип" – тип змінної екземпляру, а елемент "ім'я_змінної" – ім'я цієї змінної. Таким чином, якщо не вважати специфікатор доступу, то змінна екземпляра оголошується так само, як локальна змінна.

У класі Building всі змінні екземпляра оголошені з використанням модифікатора доступу public, який дозволяє отримувати до них доступ з боку коду, розташованого навіть поза класом Building.

Визначення class створює новий тип даних. У даному випадку цей новий тип даних називається Building. Це ім'я можна використовувати для оголошення об'єктів типу Building.

Оголошення class ─ це лише опис типу; воно не створює реальних об'єктів. Таким чином, попередній код не означає існування об'єктів типу Building.

Щоб реально створити об'єкт класу Building можливо використовувати таку інструкцію:

```
Building house = new Building ( );
```

Після виконання цієї інструкції house стане екземпляром класу Building, тобто знайде "фізичну" реальність. Детально цю інструкцію ми розглянемо далі.

Під час кожного створення екземпляра класу створюється об'єкт, який містить власну копію кожної змінної екземпляра, що визначена цим класом.

Таким чином, кожен об'єкт класу Building буде утримувати власні копії змінних екземпляра floors, area і occupants. Для доступу до цих змінних використовується оператор "крапка" (.). Він пов'язує ім'я об'єкта з ім'ям його елемента.

Загальний формат цього оператора має такий вигляд:

```
об'єкт.елемент
```

Об'єкт вказується зліва від оператора "крапка", а його елемент – справа.

Наприклад, щоб привласнити змінній floors значення 2, використовується така інструкція:

```
house.floors = 2;
```

У загальному випадку оператор "крапка" можна використовувати для доступу як до змінних екземплярів, так і методам.

Приклад 1. Лістинг програми, в якій використовується клас Building.

```
using System;
class Building
{
        public int floors;          // Кількість поверхів
        public int area;            // Загальна площа основи будівлі
        public int occupants;       // Кількість мешканців
}
class BuildingDemo
{
public static void Main ()
{
```

```
Building house = new Building (); // створення об'єкта типу Building
int areaPP;
// Площа, що припадає на одного мешканця
// Присвоюється значення полям у об'єкті house
house.occupants = 4;
house.area = 2500;
house.floors = 2;
// Обчислюємо площу, що припадає на одного мешканця будинку
areaPP = house.area / house.occupants;
Console.WriteLine ("Будинок має: \n" +
house.floors + " поверхи \n" +house.occupants + " мешканці \n" +
house.area + " квадратних метрів загальної площі, з них \n" +
areaPP + " припадає на одну людину");
}
}
```

Результат виконання програми.

Будинок має:
2 поверхи
4 мешканці
2 500 квадратних метрів загальної площі, з них
625 припадає на одну людину

Приклад роботи з двома об'єктами одного класу.

Згідно з основним принципом програмування класів, кожен об'єкт класу має власні копії змінних екземпляра, які визначені у цьому класі. Таким чином, вміст змінних в одному об'єкті може відрізнятися від вмісту аналогічних змінних в іншому.

Між двома об'єктами одного класу немає зв'язку, за винятком того, що вони є об'єктами одного і того ж типу. Наступна програма демонструє це.

Приклад 2. Робота з двома об'єктами класу Building.

```
using System;
class Building
{
        public int floors;          // кількість поверхів
        public int area;            // загальна площа основи будівлі
        public int occupants;       // кількість мешканців
}
class BuildingDemo
{
```

```
public static void Main ()
{
Building house = new Building ();
Building office = new Building ();
int areaPP; // площа, яка припадає на одного мешканця
// Присвоюється значення полям у об'єкті house
house.occupants = 4;
house.area = 2500;
house.floors = 2;
// Присвоюється значення полям у об'єкті office
office.occupants = 25;
office.area = 4200;
office.floors = 3;
// Обчислюється площа, що припадає на одного мешканця.
areaPP = house.area / house.occupants;
Console.WriteLine (" Будинок має: \n" +
house.floors + " поверхи \n" +
house.occupants + " мешканці \n" +
house.area + " кв.м. загальної площі, з них \n" +
areaPP + " припадає на одну людину");
Console.WriteLine ();
// Обчислюється площа, що припадає на одного працівника офісу.
areaPP = office.area / office.occupants;
Console.WriteLine (" Офіс має: \n" +
office.floors + " поверхи \n" +
office.occupants + " працівників \n" +
office.area + " кв.м. загальної площі, з них \n" +
areaPP + " припадає на одну людину");
}
}
```

Результат виконання програми.

Будинок має:
2 поверхи
4 мешканці
2 500 кв. м загальної площі, з них
625 припадає на одну людину

Офіс має:
3 поверхи

25 працівників

4 200 кв. м загальної площі, з них

168 припадає на одну людину

Звернути увагу на особливості операторів визначення класів і доступу до елементів-даних екземплярів класу.

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання приклади програм, які були наведені в розділі "Основні положення" даної лабораторної роботи.

2. Проекспериментуйте з програмами:

змініть вихідні дані;

досліджуйте, як впливають синтаксичні помилки на результат компіляції програми. Які під час цього виникають помилки компіляції?

Індивідуальна частина

1. З табл. 1 вибрати індивідуальний варіант предметної області.

Розробити відповідно предметної області клас (за аналогією з прикладом 1). Створити два екземпляри поточного класу і здійснити їх оброблення (з аналогією з прикладом 2).

2. На базі табл. 1 сформуйте індивідуальний варіант завдання з двома предметними областями.

Для кожної предметної області розробити відповідні два класи. Створити два об'єкта кожного класу і здійснити їх оброблення.

Таблиця 1

Індивідуальні варіанти предметної області

Варіанти	Предметна область
1	Поліграфічні верстати
2	Папір для друку
3	Фарби
4	Принтери
5	Автомобілі
6	Літаки
7	Домашні тварини
8	Домашні птахи
9	Готельний бізнес
10	Викладачі
11	Студенти
12	Туризм
13	Квіти

Кожна програма з індивідуального завдання повинна:

здійснювати елементарні операції (в методі Main ()) з елементами-даними екземплярів відповідних об'єктів. Формули для обчислень визначити самостійно, вони повинні відповідати обраній предметній області;

виводити на екран результати обчислень і вміст елементів-даних всіх екземплярів визначених об'єктів.

3. Проаналізуйте вирази для оброблення елементів-даних: визначіть допустимі діапазони зміни вхідних величин, їх розмірність і тип.

4. Підготуйте контрольні приклади, які повною мірою характеризують аналізовані вирази.

5. Розробіть алгоритм обчислення і намалюйте його графічну схему (блок-схему).

6. Набрати і відкомпілювати текст програми, усуваючи у разі необхідності помилки.

Зміст звіту

1. Титульний лист.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти налагоджених програм загальної частини лабораторного заняття з необхідними коментарями і результатом виконання.

4. Аналіз предметної області індивідуального завдання й докладного опису спроектованого класу класів.

5. Тексти налагоджених програм із результатом виконання всіх контрольних прикладів індивідуального завдання.

6. Висновки.

Контрольні запитання

1. У чому відмінність процедурного та об'єктно-орієнтованого стилю програмування?

2. Дайте визначення класу. Опишіть можливі елементи класу.

3. Що розуміється під об'єктною моделлю, які її властивості?

4. Опишіть структури найпростішої об'єктно-орієнтованої програми.

5. Перелічіть основні етапи розроблення об'єктно-орієнтованих програм.

Лабораторна робота 9
Використання простих методів

Мета роботи – отримати практичні навички розроблення програм із класами, що містять найпростіші методи.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

формату запису методів;

способів повернення з методів;

правил передачі параметрів – значень;

уміння:

для заданої наочної області визначити клас, який інкапсулює елементи-методи та елементи-дані, що потребують подальшого оброблення;

написати програму, яка обробляє інформацію (обчислення і виведення на екран результатів обчислень і вмісту елементів-даних) про декілька об'єктів одного класу заданої наочної області;

комунікації:

обґрунтування рекомендацій команді учасників проекту щодо доцільності застосування методів згідно з певним сценарієм взаємодії відповідних об'єктів;

робота в команді над окремими класами, які містять перелік методів;

автономність і відповідальність:

прийняття рішення щодо доцільності включення в структуру класу певного методу;

самостійне обґрунтування можливих варіантів реалізацій відповідних методів.

Основні положення

Мова C# є об'єктно-орієнтованою і тому всі програми на її основі містять класи. Класи необхідні для опису об'єктів, на базі яких створюються їх певні екземпляри. Реалізація сценарію (алгоритму) здійснюється як взаємодія об'єктів. Взаємодія відбувається за допомогою виклику методів і подальшому обміну відповідною інформацією між ними.

Методи мають багато спільного з функціями, основні принципи роботи з якими були розглянуті під час вивчення дисципліни "Основи програмування". Тому перед виконанням даної лабораторної роботи слід повторити

матеріал пов'язаний із практичним застосуванням функцій за процедурного стилю програмування.

Якщо функції — це засіб, що дозволяє виконувати деякі ділянки коду в довільному місці додатка, то метод – це функціональний елемент класу, який реалізує обчислення або інші дії, що виконуються класом або його екземпляром (об'єктом).

Методи класу, як правило, маніпулюють даними, визначеними в класі, і забезпечують доступ до цих даних.

Метод Main () в попередній лабораторній роботі (приклад 1) обчислював площу, що припадає на одну людину, шляхом ділення загальної площі будівлі на кількість мешканців.

Незважаючи на формальну коректність, ці обчислення виконані не найвдалішим чином. Адже з обчисленням площі, що припадає на одну людину, цілком може впоратися сам клас Building, оскільки ця величина залежить тільки від значень змінних area і occupants, які інкапсульовані в класі Building.

Якщо обчислення площі буде "закріплено" за цим класом, то в іншої програмі, яка його використовує, не доведеться виконувати цю дію "вручну". Тут у наявності не просто зручність для "інших" програм, а запобігання невиправданого дублювання коду.

Нарешті, вносячи в клас Building метод, який обчислює площу, що припадає на одну людину, ви покращуєте його об'єктно-орієнтовану структуру, інкапсуліруя всередині розглянутого класу величини, пов'язані безпосередньо з будівлею.

Щоб додати в клас Building метод, необхідно визначити його всередині оголошення класу.

Приклад 1. Додавання методу в клас Building.

Наступна версія класу Building (в порівнянні с прикладом 1 попередньої лабораторної роботи 1) містить метод з ім'ям areaPerPerson (), який відображає значення площі конкретної будівлі, що припадає на одну людину.

```
using System;
class Building
{
        public int floors;          // кількість поверхів
        public int area;            // загальна площа будівлі
        public int occupants;       // кількість мешканців
// Відображаємо значення площі, що припадає на одну людину.
        public void areaPerPerson ()
```

```
        {
        Console.WriteLine ("" + area / occupants +
        "  Припадає на одну людину");
        }
}
// Використовуємо метод areaPerPerson ().
class BuildingDemo
{
        public static void Main ()
        {
        Building house = new Building ();
        Building office = new Building ();
        // Привласнюємо значення полям у об'єкті house
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;
        // Привласнюємо значення полям у об'єкті office
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;
        // Вивід результатів
        Console.WriteLine ("Будинок має: \n" +
        house.floors + " поверхи \n" +
        house.occupants + " мешканці \n" +
        house.area + " кв.м. загальної площі, з них");
        house.areaPerPerson ();
        Console.WriteLine ("Офіс має: \n" +
        office.floors + " поверхи \n" +
        office.occupants + " працівників \n" +
        office.area + " кв.м. загальної площі, з них");
        office.areaPerPerson ();
        }
}
```

Результат виконання програми.

```
Будинок має:
2 поверхи
4 мешканці
```

2 500 кв. м загальної площі, з них
625 припадає на одну людину
Офіс має:
3 поверхи
25 працівників
4 200 кв. м загальної площі, з них
168 припадає на одну людину

Розглянемо ключові елементи цієї програми, починаючи з самого методу areaPerPerson ().

Перший рядок цього методу виглядає так:

public void areaPerPerson ()

У цьому рядку оголошується метод з ім'ям areaPerPerson (), який не має параметрів. Цей метод визначений з використанням специфікатора доступу public, тому його можуть використовувати всі інші частини програми. Метод areaPerPerson () повертає значення типу void, тобто не повертає ніякого значення.

Тіло методу areaPerPerson () складається з єдиної інструкції:

```
{
Console.WriteLine ("" + area / occupants +
"  Припадає на одну людину");
}
```

Ця інструкція відображає площу будівлі, яка припадає на одну людину, шляхом ділення значення змінної area на значення змінної occupants.

Оскільки кожен об'єкт типу Building має власну копію значень area і occupants, то під час виклику методу areaPerPerson () використовуватимуться копії цих змінних.

Метод areaPerPerson () завершується фігурною дужкою, тобто управління програмою передається конкретному об'єкту, який цей метод визвав.

Розглянемо рядок коду з методу Main ():

```
house.areaPerPerson ();
```

Ця інструкція викликає метод areaPerPerson () для об'єкта house. Для цього використовується ім'я об'єкта, за яким слід оператор "крапка". Під час виклику методу управління виконанням програми передається

тілу методу, а після його завершення управління повертається автору виклику, і виконання програми поновлюється з рядка коду, яка розташована відразу за викликом методу.

У даному випадку в результаті виклику house.areaPerPerson () відображається значення площі, яка припадає на одну людину, для будівлі, що визначено об'єктом house.

Точно так же в результаті виклику office.areaPerPerson () відображається значення площі, яка припадає на одну людину, для будівлі, що визначено об'єктом office.

Змінні екземпляра area і occupants використовуються всередині методу areaPerPerson () без яких би то не було атрибутів, тобто їм не передує ні ім'я об'єкта, ні оператор "крапка". Це обумовлено тим, що метод обробляє змінну екземпляра, яка визначена в його класі, він робить це безпосередньо, без явного посилання на об'єкт і без оператора "крапка". Адже метод завжди викликається для деякого об'єкта конкретного класу. І якщо вже виклик відбувся, об'єкт, стало бути, відомий. Таким чином, немає необхідності вказувати всередині методу об'єкт вдруге. Це значить, що значення area і occupants всередині методу areaPerPerson () неявно вказують на копії цих змінних, які належать об'єкту, який викликає метод areaPerPerson ().

Повернення з методу.

У загальному випадку існує два варіанти умов для повернення з методу.

Перший варіант пов'язано з виявленням фігурної дужки, що позначає кінець тіла методу (як продемонстровано на прикладі методу areaPerPerson ()).

Другий варіант полягає у виконанні інструкції return. Можливі дві форми використання інструкції return: одна призначена для void-методів (які не повертають значень), а інша — для повернення значень.

Негайне завершення void-методу можна організувати за допомогою наступної форми інструкції return:

```
return;
```

Під час виконання цієї інструкції управління програмою передається автору виклику методу, а код, що залишився опускається.

Приклад 2. Припинення виконання void-методів.

```
using System;
class myReturn
{
```

```
    static public void myMeth()
    {
        int i;
        for (i = 0; i < 10; i++)
        {
            if (i == 5) return; // припинення виконання методу при i = 5
            Console.Write(" " + i);
        }
    }
}
    class Program
    {
        static void Main(string[] args)
        {
            myReturn.myMeth();
            Console.WriteLine(); // для паузи

        }
    }
```

Результат виконання програми.

0 1 2 3 4

Тут цикл for працюватиме під час значення змінної "i" в діапазоні тільки від 0 до 5, оскільки, як тільки значення "i" стане рівним 5, буде виконано повернення з методу myMeth ().

Метод може мати декілька інструкцій return.

Хоча void-методи – не рідкість, більшість методів усе ж повертають значення.

Значення, що повертаються методами, використовуються в програмуванні по-різному. В одних випадках повертається значення, яке є результатом обчислень, в інших – воно просто означає, успішно чи ні виконані дії, що становлять метод, а в третіх — воно може бути код-станом. Однак незалежно від мети застосування, використання значень, що повертаються методами, є невід'ємною частиною C#-програмування. Вони використовують таку форму інструкції return:

return значення;

Тут елемент "значення" представляє значення, що повертається методом.

Здатність методів повертати значення можна використовувати для поліпшення реалізації методу areaPerPerson (). Замість того, щоб відображати значення площі, яка припадає на одну людину, метод areaPerPerson () буде тепер повертати це значення, яке можна використовувати в інших обчисленнях.

У наступному прикладі подано модифікований варіант методу areaPerPerson (), який повертає значення площі, що припадає на одну людину, а не відображає його (як в попередньому варіанті).

Приклад 3. Повернення значення методом areaPerPerson ().

```
using System;
class Building
{
        public int floors;          // кількість поверхів
        public int area;            // загальна площа будівлі
        public int occupants;    // кількість мешканців
        // Повернення значення площі, яка припадає на одну людину
        public int areaPerPerson ()
        {
                return area / occupants;
        }
}
// Використання значення від методу areaPerPerson ().
class BuildingDemo
{
        public static void Main ()
        {
        Building house = new Building ();
        Building office = new Building ();
        int areaPP; // площа, яка припадає на одну людину
        // Привласнюємо значення полям у об'єкті house
        house.occupants = 4;
        house.area = 2500;
        house.floors = 2;
        // Привласнюємо значення полям у об'єкті office
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;
```

```
        // Отримуємо для об'єкта house площу, яка припадає на одну людину
        areaPP = house.areaPerPerson ();
        Console.WriteLine ("Будинок має: \n" +
        house.floors + " поверхи \n" +
        house.occupants + " мешканці \n" +
        house.area + " кв.м. загальної площі, з них \n" +
    areaPP + " припадає на одну людину");
        Console.WriteLine ();
        // Отримуємо для об'єкта office площу, яка припадає на одну людину
        areaPP = office.areaPerPerson ();
        Console.WriteLine ("Офіс має: \n" +
        office.floors + " поверхи \n" +
        office.occupants + " працівників \n" +
        office.area + " кв.м. загальної площі, з них \n" +
        areaPP + " припадає на одну людину");
        }
}
```

Результат виконання програми співпадає з попереднім результатом (див. приклад 1).

Використання параметрів.

Під час виклику методу можна передати одне або кілька значень, які називаються аргументами.

Змінна всередині методу, яка набуває значення аргумента, називається параметром.

Параметри оголошуються всередині круглих дужок, які слідують за ім'ям методу. Синтаксис оголошення параметрів аналогічний синтаксису, який застосовується для змінних.

Параметр знаходиться в області видимості свого методу, і, крім спеціального завдання отримання аргументу, діє подібно будь-якої локальної змінної.

Для додавання в клас Building нового засобу обчислення максимально допустимої кількості мешканців будівлі можна використовувати метод з параметрами. Передбачається, що площа, яка припадає на кожну людину, не повинна бути менше певного мінімального значення. Назвемо цей новий метод maxOccupant () і наведемо його визначення.

```
	public int maxOccupant (int minArea)
	{
		return area / minArea;
	}
```

Під час виклику методу maxOccupant () параметр minArea отримує значення мінімальної площі, необхідної для життєдіяльності кожної людини. Результат, що повертається методом maxOccupant (), виходить як частка від ділення загальної площі будівлі на це значення.

Приклад 4. Повне визначення класу Building, що включає метод maxOccupant ().

```
using System;
class Building
{
	public int floors;		// кількість поверхів
	public int area;		// загальна площа будівлі
	public int occupants;	// кількість мешканців

	public int areaPerPerson()
	{
		return area / occupants;
	}

	public int maxOccupant (int minArea)
	{
		return area / minArea;
	}
}
// Використання методу maxOccupant ().
class BuildingDemo
{
	public static void Main ()
	{
	Building house = new Building ();
	Building office = new Building ();
	// Привласнюємо значення полям у об'єкті house
	house.occupants = 4;
	house.area = 2500;
```

```
        house.floors = 2;
        // Привласнюємо значення полям у об'єкті office
        office.occupants = 25;
        office.area = 4200;
        office.floors = 3;
        // Отримуємо для об'єкта house площу, яка припадає на одну
людину
        int areaPP = house.areaPerPerson();
        Console.WriteLine("Будинок має: \n" +
        house.floors + " поверхи \n" +
        house.occupants + " мешканці \n" +
        house.area + " кв.м. загальної площі, з них \n" +
        areaPP + " припадає на одну людину");
        // Отримуємо для об'єкта office площу, яка припадає на одну
людину
        areaPP = office.areaPerPerson();
        Console.WriteLine("Офіс має: \n" +
        office.floors + " поверхи \n" +
        office.occupants + " працівників \n" +
        office.area + " кв.м. загальної площі, з них \n" +
        areaPP + " припадає на одну людину");
        Console.WriteLine ("Максимальна кількість осіб для дому, \n" +
"якщо на кожного повинно доводиться " +
300 + " квадратних метрів:" +
house.maxOccupant (300));
        Console.WriteLine ("Максимальна кількість осіб для офісу, \n" +
"якщо на кожного повинно доводиться " +
200 + " квадратних метрів:" +
office.maxOccupant (200));
        }
}
```

Результат виконання програми.

```
Будинок має:
2 поверхи
4 мешканці
2 500 кв. м загальної площі, з них
625 припадає на одну людину
```

Офіс має:
3 поверхи
25 працівників
4 200 кв. м загальної площі, з них
168 припадає на одну людину
Максимальна кількість осіб для дому,
якщо на кожного повинно доводиться 300 квадратних метрів: 8
Максимальна кількість осіб для офісу,
якщо на кожного повинно доводиться 200 квадратних метрів: 21

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання приклади програм, які були наведені в розділі "Основні положення" даної лабораторної роботи.

2. Проекспериментуйте з програмами:

змініть вихідні дані;

досліджуйте, як впливають синтаксичні помилки на результат компіляції програми. Які у ході цього виникають помилки компіляції?

Індивідуальна частина

1. З табл. 1 (вона наведена в методичних вказівках до лабораторної роботи 1) вибрати індивідуальний варіант предметної області.

2. Розробити відповідно предметної області клас, який інкапсулює елементи-дані і елементи-методи (без параметрів). Написати і налагодити програму, що демонструє роботу з об'єктом (або об'єктами), визначеного вище класу.

3. Визначити додатковий метод (з параметрами) і включити його до раніш розробленого класу. Написати і налагодити програму, що демонструє роботу з об'єктом (або об'єктами), визначеного вище класу.

Зміст звіту

1. Титульний лист.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти налагоджених програм загальної частини лабораторного заняття з необхідними коментарями і результатом виконання.

4. Аналіз предметної області індивідуального завдання й докладний опис усіх спроектованих класів.

5. Тексти налагоджених програм із результатом виконання всіх контрольних прикладів індивідуального завдання.

6. Висновки.

Контрольні запитання

1. Чим відрізняються методи від функцій?
2. Що таке сигнатура методу? Наведіть приклад.
3. Які типи даних може повертати метод?
4. Які варіанти повернення з методу ви знаєте? У чому їх відмінності і коли їх доцільно застосовувати?
5. Опишіть алгоритм виклику та виконання методу.

Лабораторна робота 10
Визначення особливостей застосування конструкторів

Мета роботи – набуття практичних навичок щодо роботи з конструкторами.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

призначення конструктора і форматів його запису;

особливості застосування оператора new до змінних типу значень;

призначення системи "збору сміття" в C# і формат запису деструктора;

призначення ключового слова this;

уміння:

для заданої наочної області визначити клас, який інкапсулює елементи-дані, елементи-методи, а також конструктори;

написати програму, яка оброблює і виводить на екран інформацію щодо взаємодії декількох об'єктів, які створюються за допомогою конструкторів;

комунікації:

обґрунтування рекомендацій команді учасників проекту щодо доцільності застосування відповідних конструкторів під час розроблення класів;

робота в команді над окремими класами, які містять конструктори;

автономність і відповідальність:

прийняття рішення щодо доцільності включення в клас певного типу конструктора;

самостійне обґрунтування можливих варіантів реалізацій відповідних конструкторів.

Основні положення

У попередніх лабораторних роботах змінні кожного Building-об'єкта встановлювалися "вручну" за допомогою такої послідовності інструкцій:

```
house.occupants = 4;
house.area = 2500;
house.floors = 2;
```

Професіонал ніколи б не використовував подібний підхід. І справа не стільки в тому, що таким чином можна попросту "забути" про один або декількох даних, скільки в тому, що існує набагато більш зручний спосіб це зробити. Цей спосіб – використання конструктора.

Конструктор ініціалізує об'єкт під час його створення. Він має таке ж ім'я, що і сам клас, а синтаксично подібний до методу. Однак у визначенні конструкторів не вказується тип значення.

Формат запису конструктора такий:

```
доступ ім'я_класу ()
        {
        // Тіло конструктора
        }
```

Зазвичай конструктор використовується, щоб надати змінному екземпляру, який визначено у класі, початкові значення або виконати початкові дії, необхідні для створення повністю сформованого об'єкта. Крім того, зазвичай в якості елемента доступу використовується модифікатор доступу public, оскільки конструктори, як правило, викликаються поза їх класу.

Усі класи мають конструктори незалежно від того, визначено їх чи ні, оскільки C# автоматично надає конструктор за замовчуванням, який ініціалізує всі змінні-елементи, що мають тип значень, – нулями, а змінні-елементи посилального типу – null-значеннями. Але якщо ви визначите власний конструктор, конструктор за замовчуванням більше не використовується.

Приклад 1. Використання простого конструктора.

```
class MyClass
{
   public int x;
   public MyClass( )
```

```
    {
        x = 10;
    }
}
class ConsDemo
{
    public static void Main( )
    {
        MyClass t1 = new MyClass();
        MyClass t2 = new MyClass();
        Console.WriteLine("t1.x = {0}  t2.x = {1}", t1.x, t2.x);
    }
}
```

Результат виконання програми.

```
t1.x = 10  t2.x = 10
```

У цьому прикладі програми конструктор класу MyClass має такий вигляд:

```
public MyClass ()
{
        x = 10;
}
```

Зверніть увагу на public-визначення конструктора, яке дозволяє викликати його з коду, визначеного поза класом MyClass. Цей конструктор присвоює змінної екземпляру значення 10.

Конструктор MyClass () викликається оператором new під час створення об'єкта класу MyClass.

Наприклад, під час виконання рядка

```
MyClass t1 = new MyClass ();
```

для об'єкта t1 викликається конструктор MyClass (), який присвоює змінної екземпляру t1.x значення 10. Те ж саме справедливо і щодо об'єкта t2, тобто в результаті створення об'єкта t2 значення змінної екземпляру t2.x також стане рівним 10.

Конструктори з параметрами.

У попередньому прикладі використовувався конструктор без параметрів. Але частіше доводиться мати справу з конструкторами, які беруть один або декілька параметрів.

Параметри вносяться в конструктор точно так само, як у метод: для цього достатньо оголосити їх усередині круглих дужок після імені конструктора.

Приклад 2. Використання конструктора з параметрами.

```
using System;
class MyClass
        {
        public int x;
        public MyClass (int i)
                {
                        x = i;
                }
        }
class ParmConsDemo
{
        public static void Main ()
        {
        MyClass t1 = new MyClass (10);
        MyClass t2 = new MyClass (88);
        Console.WriteLine("t1.x = {0}  t2.x = {1}", t1.x, t2.x);
         }
}
```

Результат виконання програми.

```
t1.x = 10  t2.x = 88
```

У конструкторі MyClass () цієї версії програми визначено один параметр з ім'ям "i", який використовується для ініціалізації змінної екземпляру. Таким чином, під час виконання рядка коду

```
MyClass  t1 = new  MyClass  (10);
```

параметру "i" передається значення 10, яке потім присвоюється змінній екземпляра x.

Приклад 3. Додавання конструктора з параметрами в клас Building.

```
using System;
class Building
{
        public int floors;          // Кількість поверхів
        public int area;            // Загальна площа основи будівлі
        public int occupants;       // Кількість мешканців
```

```
        public Building (int f, int a, int o)
                {
                floors = f;
                area = a;
                occupants = 0;
                }
        public int maxOccupant (int minArea)
        {
        return area / minArea;
        }
}
// Використовуємо конструктор Building з параметрами.
class BuildingDemo
{
        public static void Main ()
        {
        Building house = new Building (2, 2500, 4);
        Building office = new Building (3, 4200, 25);
        Console.WriteLine ("Максимальна кількість осіб для дому, \n" +
        "якщо на кожного повинно припадати" +
        300 + " квадратних метрів: " +
        house.maxOccupant (300));
        Console.WriteLine ("Максимальна кількість осіб для офісу, \n" +
        "якщо на кожного повинно припадати" +
        200 + " квадратних метрів: " +
        office.maxOccupant (200));
        }
}
```

Результат виконання програми.

Максимальна кількість осіб для дому, якщо на кожного повинно припадати 300 квадратних метрів: 8.

Максимальна кількість осіб для офісу, якщо на кожного повинно припадати 200 квадратних метрів: 21.

Використання оператора new.

Тепер, коли ви більше знаєте про класи і їх конструктори, можна детальніше ознайомитися з оператором new. Формат його такий:

змінна_типу_класу = new ім'я_класу ();

Тут елемент "змінна_типу_класу" означає ім'я створюваної змінної типу класу.

Під елементом "ім'я_класу" розуміється ім'я реалізованого в об'єкті класу. Ім'я класу разом з наступною за ним парою круглих дужок – це конструктор реалізованого класу.

Якщо в класі конструктор не визначено явним чином, оператор new буде використовувати конструктор за замовчуванням, який надається засобами мови C#.

Таким чином, оператор new можна використовувати для створення об'єкта будь-якого "класового" типу.

Оскільки обсяг пам'яті комп'ютера обмежений, імовірна ситуація, коли оператор new не зможе виділити область, необхідну для створюваного об'єкта, через її відсутність в достатній кількості. У цьому випадку виникне виняткова ситуація відповідного типу.

Застосування оператора new до змінних типу значень.

У C# змінна типу значення містить власне значення. Під час компіляції програми компілятор автоматично виділяє пам'ять для зберігання цього значення. Отже, немає необхідності використовувати оператор new для явного виділення пам'яті.

І навпаки, в змінних посилального типу зберігається посилання на об'єкт, а пам'ять для зберігання цього об'єкта виділяється динамічно, тобто під час виконання програми.

Наприклад: int i = new int ();

У цьому випадку викликається конструктор за замовчуванням для типу int, який ініціалізує змінну нулем.

У загальному випадку виклик оператора new для будь-якого несилочного типу означає виклик конструктора за замовчуванням для відповідного типу. Але в цьому випадку динамічного виділення пам'яті не відбувається. Більшість програмістів не використовують оператор new з типами без посилань.

Збирання "сміття" і використання деструкторів.

Під час використання оператора new об'єктам динамічно виділяється пам'ять з пулу вільної пам'яті.

Обсяг буфера динамічно виділеної пам'яті не нескінченний, і рано чи пізно вільна пам'ять може вичерпатися. Отже, результат виконання оператора new може бути невдалим через нестачу вільної пам'яті для створення бажаного об'єкта. Тому одним із ключових компонентів схеми динамічного виділення пам'яті є відновлення вільної пам'яті від невикористовуваних об'єктів, що дозволяє робити її доступною для створення наступних об'єктів.

У багатьох мовах програмування звільнення раніше виділеної пам'яті виконується вручну. Однак у C# ця проблема вирішується по-іншому, а саме з використанням системи збирання сміття.

Система збирання сміття C# автоматично повертає пам'ять для повторного використання, діючи непомітно і без втручання програміста. Її робота полягає в наступному.

Якщо не існує жодного посилання на об'єкт, то передбачається, що цей об'єкт більше не потрібен, і займана ним пам'ять звільняється.

Цю (відновлену) пам'ять знову можна використовувати для розміщення інших об'єктів.

Система збирання сміття діє тільки спорадично під час виконання окремої програми. Ця система може і не діяти: вона не "включається" лише тому, що існує один або декілька об'єктів, які більше не використовуються в програмі. Оскільки на збирання сміття потрібен певний час, динамічна система C# активізує цей процес тільки у разі необхідності або в спеціальних випадках. Таким чином, ви навіть не будете знати, коли відбувається збирання сміття, а коли — ні.

Деструктори.

Засоби мови C# дозволяють визначити метод, який повинен викликатися безпосередньо перед тим, як об'єкт буде остаточно зруйнований системою збирання сміття. Цей метод називається деструктором і його можна використовувати для забезпечення гарантії "чистоти" ліквідації об'єкта. Наприклад, ви могли б використовувати деструктор для гарантованого закриття файла, відкритого деяким об'єктом.

Формат запису деструктора такий:

```
~ ім'я_класу ()
{
        // Код деструктора
}
```

Елемент "ім'я_класу" тут означає ім'я класу. Таким чином, деструктор оголошується подібно конструктору за винятком того, що його імені передує символ "тильда" (~). Подібно конструктору, деструктор не повертає значення.

Щоб додати деструктор у клас, досить включити його як елемент. Він викликається в момент, що передує процесу утилізації об'єкта. У тілі деструктора ви вказуєте дії, які, на вашу думку, повинні бути виконані перед руйнуванням об'єкта.

Важливо розуміти, що деструктор викликається тільки перед початком роботи системи збирання сміття і не викликається, наприклад, коли об'єкт

виходить за межі області видимості. Це означає, що ви не можете точно знати, коли буде виконаний деструктор. Однак точно відомо, що всі деструктори будуть викликані перед завершенням програми.

Використання деструктора демонструється в наступній програмі, яка створює і руйнує велику кількість об'єктів. У певний момент виконання цього процесу буде активізований збір сміття, а значить, викликані деструктори об'єктів, що руйнуються.

Приклад 4. Демонстрація використання деструктора.

```
using System;
class Destruct
{
    public int x;

    public Destruct(int i)
    {
        x = i;
    }
    // Викликається при утилізації об'єкта.
    ~Destruct()
    {
        Console.WriteLine("деструктуризація  " + x);
    }
    // Метод створює об'єкт, який негайно руйнується.
    public void generator(int i)
    {
        Destruct obiekt = new Destruct(i);
    }
}
class DestructDemo
{
    public static void Main()
    {
        int count;
        Destruct ob = new Destruct(0);
       /*Тепер згенеруємо велику кількість об'єктів. У якийсь момент часу почнеться збирання сміття. Зауваження: можливо, для активізації цього процесу вам доведеться збільшити кількість об'єктів, що генеруються.
        */
```

```
        for (count = 1; count < 5; count++)
            ob.generator(count);
        Console.WriteLine("Готово");
    }
}
```

Результат виконання програми.

```
Готово
деструктуризація  4
деструктуризація  0
деструктуризація  3
деструктуризація  2
деструктуризація  1
```

Ключове слово this.

Під час виклику методу йому автоматично передається неявно заданий аргумент, який є посиланням на зухвалий об'єкт (тобто об'єкт, для якого викликається метод). Це посилання і називається ключовим словом this.

Щоб зрозуміти сенс посилання this, розглянемо спочатку програму, що створює клас Rect, який інкапсулює значення ширини і висоти прямокутника і включає метод area (), що обчислює площу прямокутника.

Приклад 5.

```
using System;
class Rect
{
    public int width;
    public int height;
    public Rect(int w, int h)
    {
        width = w;
        height = h;
    }
    public int area()
    {
        return width * height;
    }
}
class UseRect
```

```
{
    public static void Main()
    {
        Rect r1 = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);
        Console.WriteLine("Площа прямокутника r1:  " + r1.area());
        Console.WriteLine("Площа прямокутника r2:  " + r2.area());
    }
}
```

Результат виконання програми.

```
Площа прямокутника r1:  20
Площа прямокутника r2:  63
```

Як вам уже відомо, всередині методу можна отримати прямий доступ до інших членів класу, тобто без зазначення імені об'єкта або класу. Таким чином, усередині методу area () інструкція

```
return width * height;
```

означає, що буде виконана операція множення змінних width і height, які пов'язані із зухвалим об'єктом, і метод поверне їх здобуток. Але та ж сама інструкція може бути переписана таким чином:

```
return this.width * this.height;
```

Тут слово this посилається на об'єкт, для якого викликається метод area ().

Отже, вираз this.width посилається на копію змінної width цього об'єкта, а вираз this.height — на копію змінної height того ж об'єкта.

Приклад 5. Застосування повного класу Rect, який написано з використанням посилання this:

```
using System;
class Rect
{
    public int width;
    public int height;
    public Rect(int w, int h)
    {
        this.width = w;
```

```
        this.height = h;
    }
    public int area()
    {
        return this.width * this.height;
    }
}
class UseRect
{
    public static void Main()
    {
        Rect r1 = new Rect(4, 5);
        Rect r2 = new Rect(7, 9);
        Console.WriteLine("Площа прямокутника r1:  " + r1.area());
        Console.WriteLine("Площа прямокутника r2:  " + r2.area());

    }
}
```

Результат виконання програми.

```
Площа прямокутника r1:  20
Площа прямокутника r2:  63
```

Насправді жоден C#-програміст не використовує посилання this так, як показано в цій програмі, оскільки це не дає ніякого виграшу, та й стандартна форма виглядає простіше. Однак з this можна іноді отримати користь. Наприклад, синтаксис C# допускає, щоб ім'я параметра або локальної змінної збігалося з ім'ям змінної екземпляру. У цьому випадку локальне ім'я буде приховувати змінну екземпляру. І тоді доступ до прихованої змінної екземпляру можна отримати за допомогою посилання this.

Наприклад, наступний фрагмент коду (хоча його стиль написання не рекомендується до застосування) є синтаксично допустимим способом визначення конструктора Rect ().

```
public Rect (int width, int height)
{
        this.width = width;
        this.height = height;
}
```

У цій версії конструктора імена параметрів збігаються з іменами змінних екземпляра, у результаті чого за першими ховаються другі, а ключове слово this якраз і використовується для доступу до прихованих змінних екземпляра.

Як буде показано в наступних лабораторних роботах ключове слово this широко використовується під час опису графічних додатків.

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання приклади програм, які були наведені в розділі "Основні положення" даної лабораторної роботи.

2. Проекспериментуйте з програмами.

Зверніть увагу на особливості визначення і виклику конструкторів, а також на спосіб передачі в них параметрів.

Індивідуальна частина

Модифікувати раніше розроблену індивідуальну програму для попередньої лабораторної роботи 2 (її структура аналогічна програмі, наведеною в прикладі 4 (див. лабораторну роботу 2).

Для цього слід додати в раніше налагоджену програму параметризований конструктор, який під час створення об'єкта автоматично повинен ініціалізувати відповідні поля (тобто змінні екземпляра). У результаті має бути отримана програма, подібна прикладу 3 для поточної роботи.

Зміст звіту

1. Титульний аркуш.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти налагоджених програм загальної частини лабораторного заняття з необхідними коментарями і результатом виконання.

4. Текст налагодженої програми з результатом виконання контрольних прикладів індивідуального завдання.

6. Висновки.

Контрольні запитання

1. Чим відрізняються методи від конструкторів?

2. Коли доцільно використовувати конструктори?

3. Які особливості застосування оператора new до змінних типу значень?

4. Опішить призначення системи "збирання сміття" в C# і формат запису деструктора.

5. Коли доцільно застосовувати ключове слова this?

Лабораторна робота 11
Розроблення програм з ієрархією класів

Мета роботи — набуття практичних навичок щодо розроблення об'єктно-орієнтованих програм з ієрархією класів.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

призначення спадкування як фундаментального принципу ООП;

організації доступу до елементів класу під час спадкування;

механізмів використання захищеного доступу до елементів класу за допомогою модифікатора protected;

особливостей використання конструкторів під час спадкування;

уміння:

для заданої предметної області написати програму, яка:

складається з базового класу і виведеного з нього двох-трьох похідних класів;

здійснює елементарне оброблення і виведення інформації про властивості та результати оброблення щодо окремих об'єктів успадкованого класу;

комунікації:

обґрунтування рекомендацій команді учасників проекту щодо доцільності застосування базових та успадкованих класів;

робота в команді над класами, які створюють ієрархію;

автономність і відповідальність:

прийняття рішення щодо доцільності включення в клас певної ієрархії класів;

самостійне обґрунтування можливих варіантів створювання спадкоємних класів.

Основні положення

Спадкування — один з фундаментальних принципів об'єктно-орієнтованого програмування, оскільки саме завдяки йому можливо створення ієрархічних класифікацій.

Використовуючи спадкування, можливо створити загальний клас, який визначає характеристики відносно властивостей безлічі пов'язаних елементів.

Цей клас потім може бути успадкований іншими, вузькоспеціалізованими класами з додаванням у кожен з них своїх, унікальних особливостей.

У мові C# клас, який успадковується, називається базовим. Клас, який успадковує базовий клас, називається похідним.

Отже, похідний клас — це спеціалізована версія базового класу.

У похідний клас, що успадковує всі змінні, методи і властивості, які визначені в базовому класі, можуть бути додані унікальні елементи.

C# підтримує спадкування, дозволяючи в оголошення класу вбудовувати другий клас. Це реалізується за допомогою завдання базового класу при оголошенні прохідного класу.

Розглянемо клас TwoDShape, в якому визначаються атрибути "узагальненої" двовимірної геометричної фігури (наприклад, квадрата, прямокутника, трикутника і т. д.).

```
class TwoDShape
{
        public double width;
        public double height;
        public void showDim ()
        {
        Console.WriteLine ("Ширина і висота дорівнюють" +
        width + "і" + height);
        }
}
```

Клас TwoDShape можна використовувати в якості базового (тобто як стартовий майданчик) для класів, які описують специфічні типи двовимірних об'єктів.

Наприклад, у наступній програмі клас TwoDShape використовується для виведення класу Triangle.

Приклад 1. Проста ієрархія класів (клас двовимірних об'єктів).

```
using System;
// Клас двовимірних об'єктів
class TwoDShape
```

```
{
    public double width;
    public double height;

    public void showDim()
    {
        Console.WriteLine("Ширина і висота дорівнюють  " +
        width + "  і  " + height);
    }
}
// Клас Triangle виводиться з класу TwoDShape
class Triangle : TwoDShape
{
    public string style; // тип трикутника
    // Метод повертає площу трикутника
    public double area()
    {
        return width * height / 2;
    }
    // Відображаємо тип трикутника.
    public void showStyle()
    {
        Console.WriteLine("Трикутник  " + style);
    }
}
class Shapes
{
    public static void Main()
    {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = " рівнобедрений";
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = " прямокутний";
        Console.WriteLine("Інформація про t1:  ");
```

```
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Площа дорівнює  " + t1.area());
        Console.WriteLine();
        Console.WriteLine("Інформація про t2:  ");
        t2.showStyle();
        t2.showDim();
        Console.WriteLine("Площа дорівнює  " + t2.area());
    }
}
```

Результат роботи програми.

```
Інформація про t1:
Трикутник   рівнобедрений
Ширина і висота дорівнюють  4  і  4
Площа дорівнює  8

Інформація про t2:
Трикутник   прямокутний
Ширина і висота дорівнюють  8  і  12
Площа дорівнює  48
```

У класі створюється специфічний тип об'єкта класу TwoDShape, у даному випадку трикутник.

Клас Triangle містить всі елементи класу TwoDShape і, крім того, поле style, метод area () і метод showStyle ().

У змінній style зберігається опис типу трикутника, метод area () обчислює і повертає його площу, а метод showStyle () відображає дані про тип трикутника.

Далі наведено синтаксис, який використовується в оголошенні класу Triangle, щоб зробити його похідним від класу TwoDShape.

```
class Triangle: TwoDShape
    {
    // тіло класу
    }
```

Таким чином, якщо один клас успадковує інший, то ім'я базового класу вказується після імені похідного, причому імена класів розділяються двокрапкою.

Оскільки клас Triangle включає всі елементи базового класу, TwoDShape, він може звертатися до елементів width і height всередині методу агеа ().

Крім того, всередині методу Main () об'єкти t1 і t2 можуть прямо посилатися на елементи width і height, як би вони були частиною класу Triangle.

Незважаючи на те, що клас TwoDShape є базовим для класу Triangle, це абсолютно незалежний і автономний клас. Те, що його використовують як базовий похідний клас (класи), не означає неможливість використання його самого.

Наприклад, такий фрагмент коду абсолютно легальний:

```
TwoDShape shape = new TwoDShape ();
shape.width = 10;
shape.height = 20;
shape.showDim ();
```

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.

C# не підтримує спадкування декількох базових класів в одному похідному класі. І звичайно ж, жоден клас не може бути базовим (ні прямо, ні опосередковано) для самого себе.

У кожному похідному класі можна потім точно "налаштувати" власну класифікацію.

Ось, наприклад, як із базового класу TwoDShape можна вивести похідний клас, який інкапсулює прямокутники:

```
class Rectangle: TwoDShape
{
        // Метод повертає значення true, якщо прямокутник є квадратом.
        public bool isSquare ()
        {
```

```
        if (width == height) return true;
        return false;
        }
    // Метод повертає значення площі прямокутника.
        public double area ()
        {
        return width * height;
        }
    }
```

Клас Rectangle включає клас TwoDShape і додає метод isSquare (), який визначає, чи є прямокутник квадратом, і метод area (), що обчислює площу прямокутника.

Доступ до елементів класу та успадкування.

Щоб запобігти несанкціоноване використання та внесення змін елементи класу часто оголошуються закритими.

Спадкування класу не скасовує обмеження, пов'язані з закритим доступом. Таким чином, незважаючи на те, що похідний клас містить усі елементи базового класу, він не може отримати доступ до тих з них, які оголошені закритими.

Наприклад, як показано у наступному коді, якщо елементи width і height є private-елементами в класі TwoDShape, то клас Triangle не зможе отримати до них доступ.

Приклад 2. Доступ до закритих елементів не успадковується.

```
// Цей приклад не компілюється
using System;
// Клас двовимірних об'єктів
class TwoDShape
{
        double width;      //тепер це private-елемент
        double height;     // тепер це private-елемент
        public void showDim ()
        {
        Console.WriteLine ("Ширина і висота дорівнюють" +
        width + "and" + height);
        }
}
```

```
// Клас Triangle виводиться з класу TwoDShape
class Triangle: TwoDShape
        {
        public string style; // тип прямокутника
        // Метод повертає значення площі трикутника.
        public double area ()
        {
                return width * height / 2; // помилка, не можна отримати
        прямий доступ до закритого елемента
        }
        // Відображаємо тип трикутника.
        public void showStyle ()
        {
        Console.WriteLine ("Triangle is" + style);
        }
}
```

Повідомлення про помилку.

'TwoDShape.width' is inaccessible due to its protection level

Клас Triangle не компілюється, оскільки посилання на елементи width і height всередині методу area () викликає помилку порушення прав доступу.

Оскільки width і height – закриті елементи, вони доступні тільки для елементів їх власного класу. На похідні класи ця доступність не поширюється.

Закритий елемент класу залишається закритим у рамках цього класу. До нього не можна отримати доступ з коду, визначеного поза цього класу, включаючи похідні класи.

На перший погляд може здатися, що неможливість доступу до закритих елементів базового класу з боку похідного – серйозне обмеження. Однак це не так, оскільки в C# передбачені можливості вирішення цієї проблеми.

Одна з них – protected-елементи, друга можливість — використання відкритих властивостей і методів, що дозволяють отримати доступ до закритих даних.

Далі наведена нова версія класу TwoDShape, в якої колишні елементи width і height стали властивостями.

Приклад 3. Використання властивостей для запису і читання закритих елементів класу.

```
using System;
// Клас двовимірних об'єктів.
class TwoDShape
{
    double pri_width;     // тепер це private-елемент
    double pri_height;    // тепер це private-елемент
    // Властивості width і height.
    public double width
    {
        get { return pri_width; }
        set { pri_width = value; }
    }
    public double height
    {
        get { return pri_height; }
        set { pri_height = value; }
    }
    public void showDim()
    {
        Console.WriteLine("Ширина і висота дорівнюють " +
        width + " і " + height);
    }
}
// Клас трикутника - похідний від класу TwoDShape
class Triangle : TwoDShape
{
    public string style; // тип трикутника.
    // Метод повертає значення площі трикутника
    public double area()
    {
        return width * height / 2;
    }
    // Відображаємо тип трикутника
    public void showStyle()
    {
        Console.WriteLine("Трикутник  " + style);
    }
}
```

```
class Shapes2
{
    public static void Main()
    {
        Triangle t1 = new Triangle();
        Triangle t2 = new Triangle();
        t1.width = 4.0;
        t1.height = 4.0;
        t1.style = " рівнобедрений";
        t2.width = 8.0;
        t2.height = 12.0;
        t2.style = " прямокутний";
        Console.WriteLine("Інформація про t1  :");
        t1.showStyle();
        t1.showDim();
        Console.WriteLine("Площа дорівнює  " + t1.area());
        Console.WriteLine();
        Console.WriteLine("Інформація про t2  :");
        t2.showStyle();
        t2.showDim();
        Console.WriteLine("Площа дорівнює  " + t2.area());
    }
}
```

Результат роботи програми.

```
Інформація про t1  :
Трикутник   рівнобедрений
Ширина і висота дорівнюють 4 і 4
Площа дорівнює  8

Інформація про t2  :
Трикутник   прямокутний
Ширина і висота дорівнюють 8 і 12
Площа дорівнює  48
```

Використання захищеного доступу.

Закритий елемент базового класу недоступний для похідного класу. Здавалося б, це означає, що, якщо похідний клас повинен мати доступ до елементу базового класу, його потрібно зробити відкритим. При цьому

доведеться зміриться з тим, що відкритий елемент буде доступним для будь-якого іншого коду, що іноді небажано.

Однак, таких ситуацій можна уникнути, оскільки C# дозволяє створювати захищені елементи.

Захищеним є елемент, який відкритий для своєї ієрархії класів, але закритий поза цієї ієрархії.

Захищений елемент створюється за допомогою модифікатора доступу protected. Під час оголошення protected- елемента він за суттю є закритим, але з одним винятком. Виняток набирає чинності, коли захищений елемент успадковується. У цьому випадку захищений елемент базового класу стає захищеним елементом похідного класу, а отже, і доступним для цього класу.

Таким чином, використовуючи модифікатор доступу protected, можна створювати закриті (для "зовнішнього світу") елементи класу, але разом з тим вони будуть успадковуватися з можливістю доступу з боку похідних класів.

Приклад 4. Демонстрація використання захищених елементів класу.

```
using System;
class B
{
   protected int i, j;       // закрито усередині класу B, але
                             // доступно для класу D
   public void set(int a, int b)
   {
      i = a;
      j = b;
   }
   public void show()
   {
      Console.WriteLine(i + "  " + j);
   }
}
// Клас D отримує доступ до елементів i and j класу B
class D : B
{
   int k; // закритий елемент.
   public void setk()
   {
      k = i * j;
```

```
    }
    public void showk()
    {
        Console.WriteLine(k);
    }
}
class ProtectedDemo
{
    public static void Main()
    {
        D ob = new D();
        ob.set(2, 3);        // OK, так як D "бачить" B-елементи i та j
        ob.show();           // OK, так як D "бачить" B-елементи i та j
        ob.setk();           // OK, так як це частина самого класу D
        ob.showk();          // OK, так як це частина самого класу D
    }
}
```

Результат виконання програми.

```
2  3
6
```

Оскільки в цьому прикладі клас B успадковується класом D і елементи "i" та "j" були об'явлені захищеними в класі B (тобто з використанням модифікатора доступу protected), метод setk () може отримати до них доступ. Якби елементи "i" та "j" були оголошені в класі B закритими, клас D не мав би до них права доступу, і програма не була б скомпільована.

Подібно модифікаторам public і private модифікатор protected залишається зі своїм елементом незалежно від реалізованої кількості рівнів успадкування.

Таким чином, під час використання похідного класу в якості базового для створення другого похідного класу, будь-який захищений елемент вихідного базового класу, який успадковується першим похідним класом, також успадковується в статусі захищеного і другим похідним класом.

Конструктори і спадкування.

В ієрархії класів як базові, так і похідні класи можуть мати власні конструктори. Виникає важливе питання: який конструктор відповідає за створення об'єкта похідного класу? Конструктор базового або конструктор похідного класу, або обидва одночасно?

Відповідь така: конструктор базового класу створює частину об'єкта, відповідну базового класу, а конструктор похідного класу – частину об'єкта, яка відповідна похідному класу.

У попередніх прикладах класи спиралися на конструктори за умовчанням, які створювалися автоматично засобами C#, і тому ми не стикалися з подібною проблемою. Але на практиці більшість класів має конструктори.

Якщо конструктор визначається тільки в похідному класі, в цьому разі просто створюється об'єкт похідного класу. Частина об'єкта, яка відповідає базовому класу, створюється автоматично за допомогою конструктора за замовчуванням.

Далі наведена перероблена версія класу Triangle, в якій визначається конструктор. Тут елемент style оголошений private-елементом, оскільки тепер він установлюється конструктором.

Приклад 5. Додавання конструктора в клас Triangle.

```
using System;
class TwoDShape
{
   double pri_width;     // закритий елемент
   double pri_height;    // закритий елемент
   // Властивості width і height.
   public double width
   {
      get { return pri_width; }
      set { pri_width = value; }
   }
   public double height
   {
      get { return pri_height; }
      set { pri_height = value; }
   }
   public void showDim()
   {
      Console.WriteLine("Ширина і висота дорівнює   " +
      width + "  і  " + height);
   }
}
// Клас трикутника - похідний від класу TwoDShape
```

```
class Triangle : TwoDShape
{
   string style; // private
   // Конструктор
   public Triangle(string s, double w, double h)
   {
      width = w;   // ініціалізує елемент базового класу
      height = h;  // ініціалізує елемент базового класу
      style = s; //   ініціалізує елемент свого класу
   }
   // Метод повертає значення площі трикутника
   public double area()
   {
      return width * height / 2;
   }
   // Відображаємо тип трикутника
   public void showStyle()
   {
      Console.WriteLine(" Трикутник   " + style);
   }
}
class Shapes3
{
   public static void Main()
   {
      Triangle t1 = new Triangle(" рівнобедрений   ", 4.0, 4.0);
      Triangle t2 = new Triangle(" прямокутний   ", 8.0, 12.0);
      Console.WriteLine("Інформація про t1:    ");
      t1.showStyle();
      t1.showDim();
      Console.WriteLine("Площа дорівнює    " + t1.area());
      Console.WriteLine();
      Console.WriteLine("Інформація про t2:    ");
      t2.showStyle();
      t2.showDim();
      Console.WriteLine("Площа дорівнює    " + t2.area());
   }
}
```

Результат роботи програми.

```
Інформація про t1:
 Трикутник    рівнобедрений
Ширина і висота дорівнює   4  і  4
Площа дорівнює    8

Інформація про t2:
 Трикутник    прямокутний
Ширина і висота дорівнює   8  і  12
Площа дорівнює    48
```

У цьому прикладі конструктор класу Triangle ініціалізує успадковані їм елементи класу TwoDShape, а також власне поле style.

Якщо конструктори визначені і в базовому, і в похідному класі, процес створення об'єктів дещо ускладняється, оскільки повинні виконатися конструктори обох класів.

У цьому випадку необхідно використовувати ще одне ключове слово C# base, яке має два призначення:

• викликати конструктор базового класу;

• отримати доступ до елемента базового класу, який прихований за елементом похідного класу.

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання приклади програм, які були наведені в розділі "Основні положення" даної лабораторної роботи.

2. Проекспериментуйте з програмами:

Зверніть увагу на особливості синтаксису визначення похідних класів.

Індивідуальна частина

Для заданої предметної області написати програму, яка:

складається з розробленого базового класу і виведених з нього двох-трьох похідних класів;

здійснює елементарне оброблення і виведення інформації про властивості та результати оброблення для окремих об'єктів успадкованого класу.

Як прототип можна використовувати програму з прикладу 5.

Зміст звіту

1. Титульний аркуш.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти налагоджених програм загальної частини лабораторного заняття з необхідними коментарями і результатом виконання.

4. Текст налагодженої програми з результатом виконання контрольних прикладів індивідуального завдання.

6. Висновки.

Контрольні запитання

1. Коли доцільно використовувати ієрархію класів?

2. Як здійснюється організація доступу до елементів класу під час спадкування?

3. Опішить механізм використання захищеного доступу до елементів класу за допомогою модифікатора protected.

4. Наведіть особливості використання конструкторів під час спадкування.

5. Що таке "властивості" екземпляра об'єкта? Яким чином вони задаються в класі? Опішить синтаксис завдання.

Змістовий модуль 4
Організація мультимедійних програм і даних

Лабораторна робота 12
Розроблення типового каркаса графічного додатка

Мета роботи – отримати практичні навички розроблення багатовіконних Windows-додатків, що містять основне меню в головному вікні.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

особливостей взаємодії користувача з Windows;

типової структури Windows-програми;

технології створення керуючих елементів;

уміння:

використовувати стандартні вікна повідомлень;

розробити Windows-додаток, що складається з декількох вікон і здійснює виклик з головного меню основного вікна відповідні програми;

комунікації:

аргументована взаємодія з клієнтами та замовниками під час вибору технології розроблення Windows-програми;

робота в команді над окремими фрагментами основного та контекстного меню Windows-додатка;

автономність і відповідальність:

самостійне формулювання рекомендацій щодо вибору ієрархічної структури основного та контекстного меню Windows-додатка.

здатність обґрунтувати доцільності застосування певної технології розроблення Windows-програми.

Основні положення

Дві технології створення Windows-додатків.

Пакет Visual Studio включає великий набір засобів розроблення, які автоматизують більшу частину процесу створення Windows-програми. За допомогою цих засобів можна створювати та поміщати в потрібне місце різні елементи управління і меню, які будуть використовувані додатком.

Visual Studio допомагає також створювати класи і методи, які необхідні для кожного керуючого елемента. Це дуже зручний інструмент для створення більшості Windows-додатків, хоча і не єдино можливий.

Windows-програми можна створювати і за допомогою текстового редактора з подальшою компіляцією вихідного коду подібно до того, як це робиться з консольними додатками.

Відносно прості Windows-програми, які розглядаються в цієї лабораторної роботі, досить невеликі за розміром, тому їх створення дається тут у формі, яка підходить для використання текстового редактора. Але загальна структура, розроблення та організація програм залишаються такими ж, як і під час використання автоматизованих засобів розроблення, які будуть розглянути у таких лабораторних роботах.

Таким чином, матеріал цієї лабораторної роботи може бути застосовано до будь-якого способу створення програм.

Особливості взаємодії користувача з Windows.

Перш ніж приступати до Windows-програмування, необхідно зрозуміти, як користувач взаємодіє з Windows, оскільки саме цей фактор визначає архітектуру всіх Windows-програм.

Робота користувача з Windows в корені відрізняється від взаємодії, які реалізовані в консольних програмах.

У разі консольної програми саме ваша програма ініціює взаємодію з операційною системою. Прикладом може слугувати програма, яка запитує вхідні дані і виводить результати шляхом виклику методів Read () або WriteLine (). Таким чином, програми, написані "традиційним способом", самі звертаються до операційної системи, а не операційна система до них.

Але відносно "своїх" програм Windows передбачає зовсім іншу модель відношень: саме Windows повинна звертатися до вашої програмі.

Процес взаємодії організований таким чином: програма очікує до того часу, поки не отримає повідомлення від Windows. Отримавши його, програма повинна вжити відповідну дію. Відповідаючи на повідомлення, вона може викликати метод, який визначено в Windows, але головне тут те, що ініціатором взаємодії все-таки є Windows.

Таким чином, загальний формат всіх Windows-програм продиктований механізмом спілкувань, який і лежить в основі взаємодії з Windows.

Існує безліч різних повідомлень, які Windows може послати програмі. Наприклад, під час кожного клацання кнопкою мишки у вікні вашої програми буде послано повідомлення, пов'язане з клацанням кнопкою мишки. З точки зору програми повідомлення надходять випадковим чином. Ось чому Windows-програми нагадують програми, які керуються перериваннями.

Windows-форми.

Ядром Windows-програм, написаних на C#, є форма.

Форма інкапсулює основні функції, необхідні для створення вікна, його відображення на екрані й отримання повідомлень. Форма може бути

вікном будь-якого типу, включаючи основне вікно програми, дочірнє або навіть діалогове вікно.

Спочатку вікно створюється порожнім. Потім у нього додаються меню і елементи управління, наприклад екранні кнопки, списки і прапорці. Таким чином, форму можна подати у вигляді контейнера для інших Windows-об'єктів.

Коли вікну надсилається повідомлення, то воно перетворюється на подію. Отже, щоб обробити Windows-повідомлення, достатньо для нього зареєструвати обробник подій. Під час отримання цього повідомлення обробник подій буде викликатися автоматично.

Клас Form.

Форма створюється за допомогою реалізації об'єкта класу Form або класу, який є похідним від Form.

Клас Form крім поведінки, обумовленої власними елементами, демонструє поведінку, що успадкована від предків. Серед його базових класів виділяється своєю значущістю клас Control.

Клас Control.

Клас Control визначає риси, властиві всім Windows-елементам управління. Той факт, що клас Form виведений з класу Control, дозволяє використовувати форми для створення елементів управління.

Використання деяких елементів класів Form і Control демонструється в прикладах, які наведені нижче.

Приклад 1. Типова структура найпростішої Windows-програми.

```
using System;
using System.Windows.Forms;
// Клас WinSkel, похідний від класу Form
class WinSkel : Form
{
   public WinSkel()
   {
      // Привласнюємо вікна ім'я
      Text = "Скелет (шаблон) Windows-вікна";
   }
   // Метод Main, який використовується тільки для запуску програми
   [STAThread]
   public static void Main()
   {
```

```
            WinSkel skel = new WinSkel(); // створюємо форму
            // Запускаємо механізм функціонування вікна
            Application.Run(skel);
        }
    }
```

Щоб скомпілювати цю програму за допомогою Visual Studio можна діяти за двома варіантами.

Варіант 1. Створити консольний додаток. Після чого перетворити його в графічний додаток шляхом відповідної настройки середовища розробки і підключення до нього простору імен System.Windows.Forms. Проте надалі ми будемо використовувати більш простий шлях (варіант 2), який спочатку є орієнтований на графічні додатки.

Варіант 2. Створіть новий проект Windows Form Application. Як результат автоматично буде получено шаблон Windows-додатка (рис. 1).

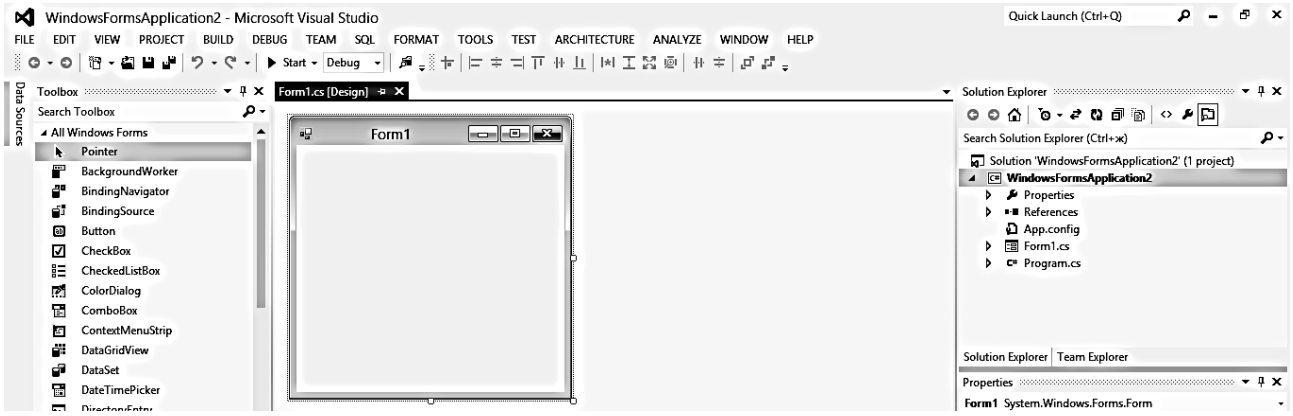


Рис. 1. **Шаблон Windows-додатка, який створюється автоматично**

З цим проектом буде пов'язаний файл Forml.cs. Оскільки наша задача створити цей додаток "вручну" за допомогою відповідного коду, треба видалити цей файл.

Потім, клацнувши лівою кнопкою мишки на імені файла Program.cs, ви повинні побачити таке діалогове вікно (рис. 2).

Далі слід замінити вміст вікна редактора тексту на код, який надано в прикладі 1. Після чого виконати компіляцію і запуск програми на виконання. Результат наведено далі (рис. 3).

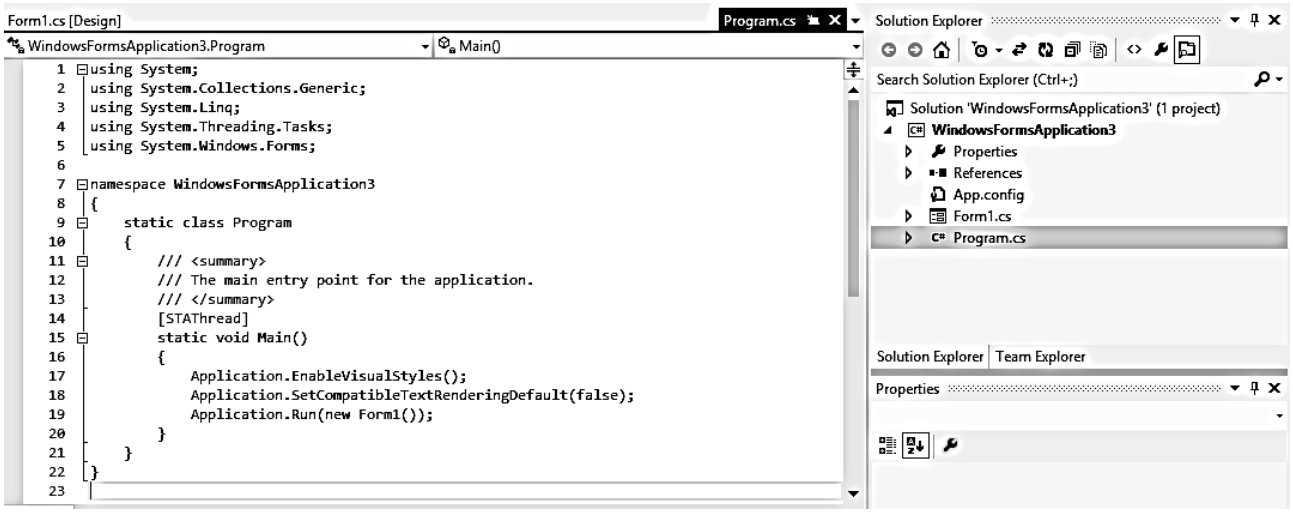


Рис. 2. **Вміст файла Program.cs, який Visual Studio створює автоматично**

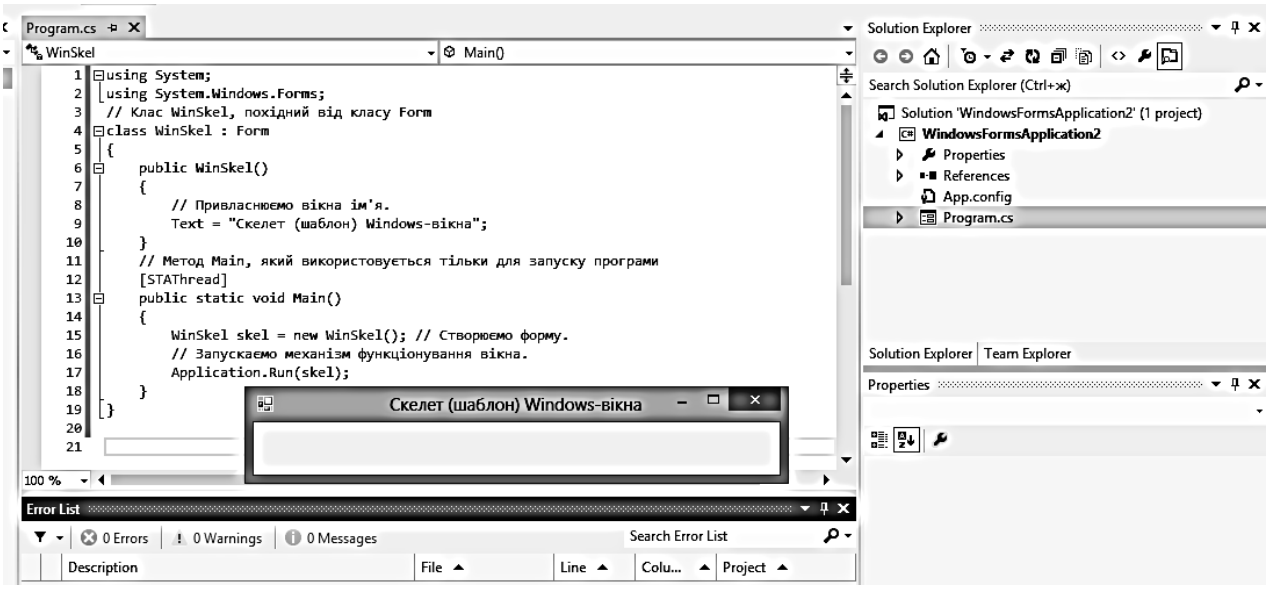


Рис. 3. **Типова структура найпростіший Windows-програми і результат її виконання**

Ця програма створює і відображає вікно, яке не містить інших елементів управління. Тим не менше, воно дозволяє показати дії, необхідні для побудови повнофункціональної Windows-програми. Іншими словами, вікно є стартовий майданчик, на якій можна побудувати більшість Windows-додатків.

Розглянемо код прикладу 1 за строками.

Перш за все, зверніть увагу на те, що програма включає два простори імен: System і System.Windows.Forms.

Простір імен System необхідне для використання атрибута STAThread, який передує методу Main (), а System.Windows.Forms призначене для підтримки підсистеми Windows.Forms.

Потім створюється клас WinSkel, який успадковує клас Form. Отже, клас WinSkel визначає тип форми. У даному випадку це найпростіша (мінімальна) форма на відміну від тієї, яку автоматично генерує майстер WindowsForms (див. рис. 1, рис. 2).

У тілі конструктора класу WinSkel міститься тільки один рядок коду:

```
Text = "Скелет (шаблон) Windows-вікна";
```

Тут встановлюється властивість Text, яка містить назву вікна. Таким чином, після виконання інструкції присвоєння рядок заголовка вікна буде містити текст "Скелет (шаблон) Windows-вікна".

Властивість Text успадкована від класу Control і визначається так:

```
public virtual string Text {get; set; }
```

Метод Main () оголошено подібно іншим методам Main (), що входять до складу програм.

Заголовку методу Main () передує властивість STAThread.

Microsoft заявляє, що ця властивість повинна мати метод Main () в кожній Windows-програмі.

Властивість STAThread встановлює модель організації потокової обробки (threading model). У цьому випадку йдеться про модель з однопоточним управлінням, тобто таке оброблення даних, коли всі об'єкти виконуються в єдиному процесі (single-threaded apartment ─ STA).

Розгляд моделей організації потокового оброблення виходить за рамки даної лабораторної роботи.

У методі Main () створюється об'єкт класу WinSkel з ім'ям skel. Цей об'єкт потім передається методу Run (), визначеним в класі Application:

```
Application.Run (skel);
```

Ця інструкція запускає механізм функціонування вікна. Клас Application визначається в просторі імен System.Windows.Forms і інкапсулює можливості, які властиві всім Windows-додаткам.

Ось як визначається використовуваний тут метод Run ():

```
public static void Run (Form ob)
```

Як параметр він приймає посилання на форму. Оскільки клас WinSkel виведено з класу Form, об'єкт типу WinSkel можна передати методу Run ().

Результат виконання програми – Windows-вікно (див. рис. 3). Воно має стандартний розмір (300 пікселів по ширині і 300 пікселів по висоті).

Це вікно повністю функціонально. Можна змінити його розміри (на рис. 3 розмір вікна зменшено), перемістити, згорнути, відновити і закрити. Таким чином, основні властивості, притаманні практично всім вікнам, були досягнуті написанням всього декількох рядків програмного коду.

Для порівняння: така ж програма, але написана на мові C++ безпосередньо викликає інтерфейс Windows API, вона зажадала б приблизно в п'ять разів більше програмних рядків!

Попередній приклад продемонстрував основні принципи створення Windows додатків, заснованих на застосуванні вікон.

Отже, щоб створити форму необхідно:

створити клас, похідний від класу Form;

ініціалізувати цю форму відповідно до вимог програми;

створити об'єкт похідного класу;

викликати метод Application.Run () для цього об'єкта.

Створення керуючих елементів (на прикладі кнопки).

У загальному випадку функціональність вікна забезпечується елементами двох типів: елементами управління і меню. Саме за допомогою цих елементів і взаємодіє користувач з програмою.

У Windows визначені різні типи керуючих елементів, включаючи екранні кнопки, прапорці, перемикачі та вікна списків. Незважаючи на відмінності між ними, технології їх створення приблизно однакові. Для прикладу розглянемо технологію створення кнопки.

Екранна кнопка інкапсулювана в класі Button, який виведено з абстрактного класу ButtonBase.

Оскільки клас ButtonBase призначено для реалізації поведінки віконного елемента управління, він успадковує клас Control.

У класі Button визначений тільки один конструктор:

```
public Button ();
```

Цей конструктор створює кнопку стандартного розміру, розташовану всередині вікна. Ця кнопка не містить опису, тому, перш ніж використовувати її, необхідно описати її, присвоївши властивості Text відповідний текстовий рядок.

Для вказівки місця розташування кнопки у вікні необхідно привласнити властивості Location координати її верхнього лівого кута.

Властивість Location успадкована від класу Control і визначається так:

```
public Point Location {get; set; }
```

Координати зберігаються в структурі Point, яка визначена в просторі імен System.Drawing. Вона містить такі властивості:

```
public int X {get; set; }
public int Y {get; set; }
```

Таким чином, щоб створити кнопку з написом "Клацніть" і прив'язати її до точки з координатами 100, 200, треба використати таку послідовність інструкцій:

```
MyButton = new Button ();
MyButton.Text = "Клацніть";
MyButton.Location = new Point (100, 200);
```

Розміщення кнопки на формі.

Після створення кнопки її необхідно помістити на форму. Це реалізується за допомогою методу Add (), який викликається з колекції елементів управління, пов'язаних з формою. Ця колекція доступна за допомогою властивості Controls, яка успадкована від класу Control. Ось як визначається метод Add ():

```
public virtual void Add (Control cnt)
```

Тут параметр cntl означає елемент керування, що додається. Після того, як елемент буде додано до складу форми, він стане видимим під час відображення самої форми.

Приклад 2. Розміщення кнопки на формі.

```
using System;
using System.Windows.Forms;
using System.Drawing;

class ButtonForm : Form
{
   Button MyButton = new Button();
```

```
    public ButtonForm()
    {
        Text = "Використання кнопки";
        MyButton.Text = "Клацніть";
        MyButton.Location = new Point(100, 200);
        Controls.Add(MyButton);
    }
    [STAThread]
    public static void Main()
    {
        ButtonForm skel = new ButtonForm();
        Application.Run(skel);
    }
}
```

У цій програмі створюється клас ButtonForm, який є похідним від класу Form. Він містить поле типу Button з ім'ям MyButton. У конструкторі класу ButtonForm кнопка створюється, ініціалізується і поміщається на форму. Під час виконання цієї програми відображається таке вікно (рис. 4).

Ви можете клацнути на кнопці, але нічого не станеться. Щоб змусити кнопку виконувати будь-які дії, необхідно додати в програму обробник подій.

Оброблення подій у Windows-додатках (на прикладі оброблення повідомлення від кнопки).

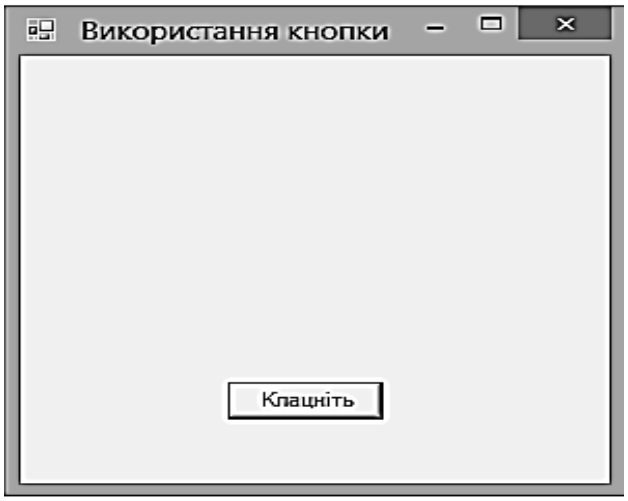


Рис. 4. **Розміщення кнопки на форму**

У загальному випадку, коли користувач впливає на елемент управління, його дія передається програмі у вигляді повідомлення.

У C#-програмі, яка заснована на застосуванні вікон, такі повідомлення обробляються обробниками подій.

Отже, щоб отримати повідомлення, в програму необхідно включити власний обробник подій, який слід додати в список обробників, що викликаються під час генерування повідомлення.

Для повідомлень, пов'язаних з клацанням на кнопці, це означає додавання обробника для події Click.

Подія Click визначається в класі Button. Вона успадкована від класу Control. Її загальний формат такий:

```
public Event EventHandler Click;
```

Делегат EventHandler визначається так:

```
public delegate void EventHandler (object who, EventArgs args)
```

Об'єкт, який згенерував подію, передається в параметрі who, а інформація, яка пов'язана з цією подією, – в параметрі args.

Для багатьох подій у якості параметра args буде служити об'єкт класу, який виведено з класу EventArgs. Оскільки клацання на кнопці не вимагає додаткової інформації, тому під час оброблення події клацання не потрібно турбуватися про аргументи цієї події.

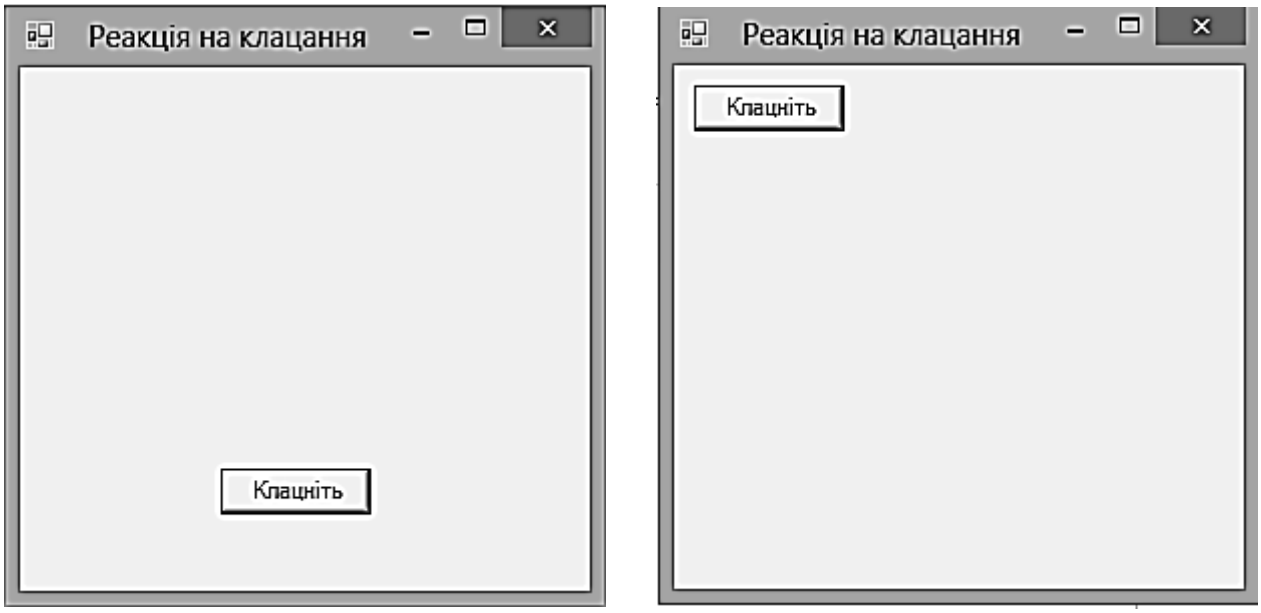
Наступна програма заснована на попередній, але з додаванням коду реакції на клацання. Під час кожного натискання на кнопці буде змінюватися її місце розташування.

Приклад 3. Оброблення повідомлень від кнопки.

```
using System;
using System.Windows.Forms;
using System.Drawing;
class ButtonForm : Form
{
   Button MyButton = new Button();  // створення екземпляру кнопки

   public ButtonForm()                    // конструктор
   {
   Text = "Реакція на клацання";          // напис на формі
   MyButton.Text = "Клацніть";            // напис на кнопці
   MyButton.Location = new Point (100, 200); // координати кнопки
   // Додаємо в список подій обробник подій кнопки
```

```
    MyButton.Click += new EventHandler (MyButtonClick);
    Controls.Add (MyButton); // додаємо кнопку на форму
    }
    [STAThread]
    public static void Main()
    {
       ButtonForm skel = new ButtonForm();
       Application.Run(skel);
    }
    // Оброблювач для кнопки MyButton.
    protected void MyButtonClick(object who, EventArgs e)
    {
       if (MyButton.Top == 200)
          MyButton.Location = new Point(10, 10);
       else
          MyButton.Location = new Point(100, 200);
    }
}
```

Результат виконання програми наведено на рис. 5.



Початковий стан вікна Стан після натискання на кнопку

Рис. 5. **Оброблення повідомлень від кнопки**

Оброблювач MyButtonClick () використовує таку ж сигнатуру, як і наведений делегат EventHandler, а це означає, що обробник можна додати в ланцюжок обробників для події Click.

Зверніть увагу на те, що в його визначенні використано модифікатор типу protected. І хоча це не є обов'язковою вимогою, така модифікація має сенс, оскільки обробники подій не призначені для виклику кодом, а використовуються лише для відповіді на події.

У коді обробника координати верхньої межі кнопки визначаються за допомогою властивості Top.

Наступні властивості визначаються для всіх елементів управління (вони задають координати верхнього лівого та нижнього правого кутів):

```
public int Top {get; set; }
public int Bottom {get; }
public int Left {get; set; }
public int Right {get; }
```

Зверніть увагу на те, що місце розташування елемента керування можна змінити за допомогою властивостей Top і Left, але не властивостей Bottom і Right, оскільки останні призначені тільки для читання. (Для зміни розміру елемента керування можна використовувати властивості Width і Height.)

Під час отримання події, яка пов'язана з клацанням на кнопці, перевіряється координата її верхньої межі, і, якщо вона дорівнює початковому значенню 200, для кнопки встановлюються нові координати: 10, 10. В іншому випадку кнопка повертається у вихідне положення з координатами 100, 200. Тому під час кожного натискання на кнопці її місце розташування змінюється.

Перш ніж обробник MyButtonClick () зможе отримувати повідомлення, його необхідно додати в ланцюжок обробників подій, які пов'язані з подією Click. Це реалізується в конструкторі класу ButtonForm за допомогою такої інструкції:

```
MyButton.Click + = new EventHandler (MyButtonClick);
```

Після виконання цієї інструкції під час кожного натискання на кнопці буде викликатися обробник подій MyButtonClick ().

Використання вікна повідомлень.

Одним із найбільш корисних вбудованих засобів Windows-додатків є вікно повідомлень. Воно дозволяє відображати повідомлення. З його допомогою можна також отримати від користувача такі прості відповіді (на поставлені питання), як Так, Ні або ОК.

У програмі, яка заснована на застосуванні вікон, вікно повідомлень підтримується класом MessageBox. Об'єкт класу створювати не потрібно. Для його досить викликати певний у цьому класі статичний метод Show ().

Метод Show () використовується в декількох форматах. Один із них виглядає так:

```
public static DialogResult Show (
                                    string     msg,
                                    string     caption,
                                    MessageBoxButtons     mbb
                                    )
```

Рядок, що відображається всередині вікна, передається в параметрі msg.

Тема вікна повідомлення – у параметрі caption.

Кнопки, які відображаються у вікні, задаються параметром mbb.

Метод повертає відповідь користувача.

Значення, що повертається методом Show (), означає, яка кнопка натиснута користувачем. Це може бути одне з таких значень:

Abort	Cancel	Ignore	No
None	OK	Retry	Yes

Наприклад: if (result == DialogResult.Yes) Application.Exit ();

MessageBoxButtons – це перерахування, яке визначає такі значення:

AbortRetrylgnore	OK	OKCancel
RetryCancel	YesNo	YesNoCancel

Кожне з цих значень описує кнопки, які будуть включені у вікно повідомлень.

Наприклад, якщо параметр mbb містить значення YesNo, то у вікні повідомлень будуть відображені кнопки Yes і No.

У програмі можна перевірити значення, що повертається методом Show (), і визначити лінію поведінки, яка обрана користувачем.

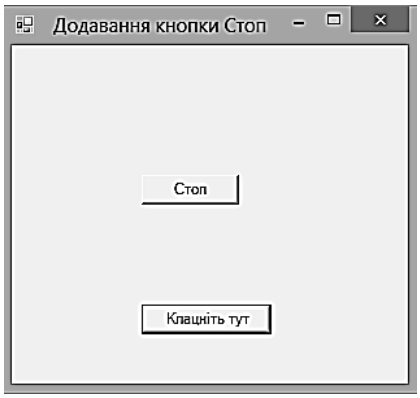
Наприклад, якщо у вікні повідомлення користувач попереджається про можливість перезапису файла, то програма запобіжить перезапис, якщо користувач клацне на кнопці Cancel, або виконає її, якщо користувач клацне на ОК.

Приклад 4. Наступна програма заснована на попередній, але з додаванням кнопки "Стоп" і вікна повідомлень.

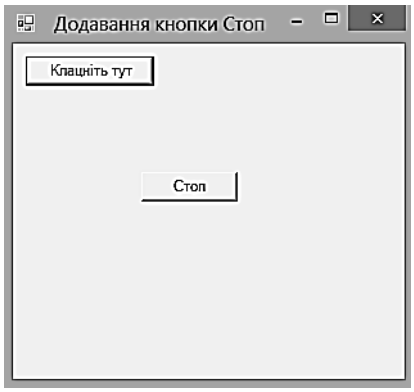
```
using System;
using System.Windows.Forms;
using System.Drawing;
class ButtonForm: Form
{
Button MyButton;
Button StopButton;

        public ButtonForm ()   // конструктор
        {
        Text = "Додавання кнопки Стоп";

        // Створюємо кнопки.
        MyButton = new Button ();
        MyButton.Text = "Клацніть тут";
        MyButton.Location = new Point (100, 200);
        MyButton.Width = 100;
        StopButton = new Button ();
        StopButton.Text = "Стоп";
        StopButton.Location = new Point (100, 100);

        // Додаємо обробники подій.
        MyButton.Click += new EventHandler (MyButtonClick);
        Controls.Add (MyButton);
        StopButton.Click+= new EventHandler (StopButtonClick);
        Controls.Add (StopButton);
        }
[STAThread]
public static void Main ()
        {
                ButtonForm skel = new ButtonForm ();
                Application.Run (skel);
        }
// Оброблювач подій для кнопки MyButton.
protected void MyButtonClick (object who, EventArgs e)
        {
        if (MyButton.Top == 200)
```

```
            MyButton.Location = new Point (10, 10);
        else
        MyButton.Location = new Point (100, 200);
        }
// Оброблювач подій для кнопки StopButton.
protected void StopButtonClick (object who, EventArgs e)
        {
        // Якщо користувач відповість клацанням на кнопці Yes,
        // програма буде завершена.
        DialogResult result = MessageBox.Show ("Зупинити програму?",
                        "Завершення", MessageBoxButtons.YesNo);
        if (result == DialogResult.Yes)
        Application.Exit ();
        }
}
```

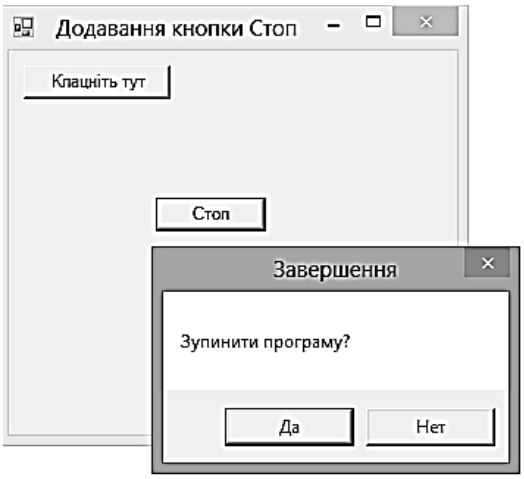
Результат виконання програми наведено на рис. 6.



Після запуску програми



Після натискання кнопки "Клацніть тут"



Після натискання кнопки "Стоп"

Рис. 6. **Додавання кнопки "Стоп"**

У обробнику подій, що пов'язано з кнопкою "Стоп", організовано відображення вікна повідомлень, в якому користувачеві пропонується відповісти на питання, чи бажає він зупинити програму. Якщо користувач відповість клацанням на кнопці "Да", програма зупиниться. Якщо ж він клацне на кнопці "Нет", виконання програми буде продовжено.

Розроблення елементів основного меню Windows-додатка.

Головне вікно практично всіх Windows-додатків включає меню, яке розташовано вздовж верхньої його межі. Воно називається основним.

Основне меню зазвичай містить такі категорії верхнього рівня, як Файл, Виправлення і Сервіс. З основного меню можна отримати меню, що розкриваються, які в свою чергу містять команди, пов'язані з відповідною категорією.

Під час вибору елемента меню генерується повідомлення. Отже, щоб виконати команду меню, програма повинна присвоїти кожному елементу меню відповідний обробник подій.

Основне меню створюється шляхом комбінації двох класів. Перший клас – MainMenu – інкапсулює загальну структуру меню, а другий – Menultem – окремий його елемент.

Елемент меню може становити або кінцеву дію (наприклад, закрити), або активізувати інше спадне меню.

Обидва класи – MainMenu і Menultem – успадковують клас Menu.

Під час вибору елемента меню генерується подія Click, яку визначено в класі Menultem. Отже, щоб обробити вибір елемента меню, у список обробників події Click, які пов'язані з цим елементом, необхідно додати відповідний обробник.

Кожна форма має властивість Menu, яка визначена таким чином:

```
public MainMenu Menu {get; set; }
```

За замовчуванням цій властивості не присвоїли ніяке меню. Щоб відобразити основне меню, цю властивість необхідно "налаштувати" відповідним чином.

Для створення основного меню виконайте такі дії.

1. Створіть об'єкт класу MainMenu.

2. У об'єкт класу MainMenu додайте об'єкти класу Menultem, які описують категорії верхнього рівня. Ці елементи меню додаються в колекцію типу Menultems, яка пов'язана з основним меню.

3. Для кожного MenuItem-об'єкта верхнього рівня додайте список MenuItem-об'єктів, який визначає спливаюче меню що пов'язане з елементом меню верхнього рівня. Ці елементи меню додаються в колекцію MenuItems, яка пов'язана з кожним елементом меню верхнього рівня.

4. Додайте обробники подій для кожного елемента меню.

5. Дайте об'єкту класу MainMenu властивості Menu, які пов'язані з формою.

У наступному фрагменті коду показано, як створити меню Файл, який містить три команди: "Відкрити", "Закрити" та "Вийти".

```
MyMenu = new MainMenu (); // створюємо об'єкт основного меню

// Додаємо в це меню елемент верхнього рівня.
MenuItem m1 = new MenuItem ("Файл");
MyMenu.MenuItems.Add (m1);

// Створення підміню "Файл".
MenuItem subm1 = new MenuItem ("Відкрити");
m1.MenuItems.Add (subm1);
 MenuItem subm2 = new MenuItem ("Закрити");
m1.MenuItems.Add (subm2);
 MenuItem subm3 = new MenuItem ("Вийти");
m1.MenuItems.Add (subm3);
```

Наведена послідовність інструкцій починається зі створення об'єкта класу MainMenu з ім'ям MyMenu. Цей об'єкт буде знаходитися на верхньому рівні структури меню.

Потім створюється елемент меню m1 із заголовком "Файл". Він додається безпосередньо до об'єкта MyMenu.

Після цього створюється спливаюче меню, яке пов'язане з командою "Файл" основного меню. Зверніть увагу на те, що елементи меню, що розкривається, додаються до об'єкта m1, який є елементом "Файл" основного меню.

Якщо один MenuItem-об'єкт додається до іншого, то об'єкт що додається стає частиною спадного меню, яке пов'язане з елементом, до якого додається MenuItem-об'єкт.

Отже, після того як елементи subm1 - subm3 будуть додані до елементу m1, під час вибору команди "Файл" відобразиться спадне меню, що містить команди "Відкрити", "Закрити" та "Вийти".

Створивши меню, для кожного його елемента необхідно створити пов'язані з ним обробники подій.

Під час вибору користувачем команди меню генерується подія Click. Тому у процесі виконання наступної послідовності інструкцій елементам subm1 ─ subm3 будуть призначені відповідні обробники подій.

```
// Додаємо обробники подій для елементів меню.
subm1.Click += new EventHandler (MMOpenClick);
subm2.Click += new EventHandler (MMCloseClick);
subm3.Click += new EventHandler (MMExitClick);
```

Таким чином, якщо користувач вибере команду "Вийти", виконається обробник подій MMExitClick ().

Нарешті, властивості Menu форми потрібно присвоїти об'єкт класу MainMenu:

```
Menu = MyMenu; // призначаємо меню формі
```

Після виконання цієї інструкції вікно буде відображатися разом із меню, під час вибору команд якого будуть викликатися відповідні обробники подій.

Приклад 5. Програма демонструє створення основного меню та обробку подій, які пов'язані з вибором відповідних команд.

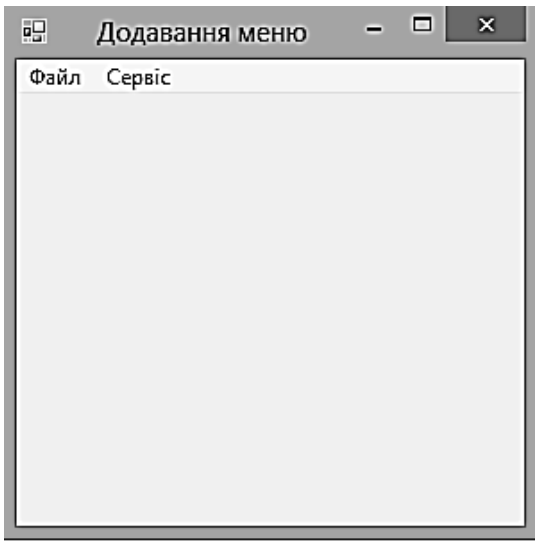
```
using System;
using System.Windows.Forms;
class MenuForm: Form
{
        MainMenu MyMenu;    // оголошуємо ім'я об'єкта основного меню
        public MenuForm ()    // конструктор
        {
        Text = "Додавання меню";
        // Створюємо об'єкт основного меню.
        MyMenu = new MainMenu ();
        // Додаємо в це меню елементи (m1, m2) верхнього рівня.
        MenuItem m1 = new MenuItem ("Файл");
        MyMenu.MenuItems.Add (m1);
        MenuItem m2 = new MenuItem ("Сервіс");
        MyMenu.MenuItems.Add (m2);
```

```
// Створення підміню "Файл".
MenuItem subm1 = new MenuItem ("Відкрити");
m1.MenuItems.Add (subm1);
MenuItem subm2 = new MenuItem ("Закрити");
m1.MenuItems.Add (subm2);
MenuItem subm3 = new MenuItem ("Вийти");
m1.MenuItems.Add (subm3);

// Створюємо підміню "Сервіс".
MenuItem subm4 = new MenuItem ("Координати");
m2.MenuItems.Add (subm4);
MenuItem subm5 = new MenuItem ("Змінити розмір");
m2.MenuItems.Add (subm5);
MenuItem subm6 = new MenuItem ("Відновити");
m2.MenuItems.Add (subm6);

// Додаємо обробники подій для елементів меню.
subm1.Click+= new EventHandler (MMOpenClick);
subm2.Click+= new EventHandler (MMCloseClick);
subm3.Click+= new EventHandler (MMExitClick);
subm4.Click+= new EventHandler (MMCoordClick);
subm5.Click+= new EventHandler (MMChangeClick);
subm6.Click+= new EventHandler (MMRestoreClick);

// Призначаємо меню формі.
Menu = MyMenu;
}  // закінчення конструктора
[STAThread]
public static void Main ()
{
MenuForm skel = new MenuForm ();
Application.Run (skel);
}

// Оброблювач для команди меню "Координати".
protected void MMCoordClick (object who, EventArgs e)
{
// Створюємо рядок, який містить три координати.
string size =
```

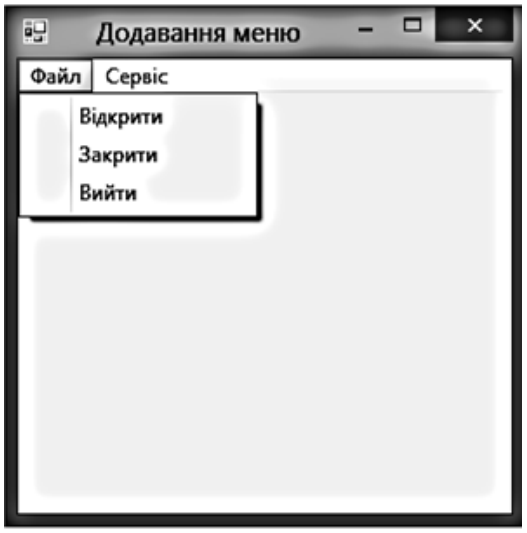
```
        String.Format ("{0}: {1}, {2} \n {3}: {4}, {5}",
        "Вгорі, Ліворуч", Top, Left,
        "Унизу, Ліворуч", Bottom, Right);
        // Відображаємо вікно повідомлень.
        MessageBox.Show (size, "Координати вікна",
        MessageBoxButtons.OK);
        }

        // Оброблювач для команди меню "Змінити розмір".
        protected void MMChangeClick (object who, EventArgs e)
        {
        Width = Height = 200;
        }

        // Оброблювач для команди меню "Відновити".
        protected void MMRestoreClick (object who, EventArgs e)
        {
        Width = Height = 300;
        }

        // Оброблювач для команди меню "Відкрити".
        protected void MMOpenClick (object who, EventArgs e)
        {
        MessageBox.Show ("Неактивна команда", "Заглушка",
        MessageBoxButtons.OK);
        }
// Оброблювач для команди меню "Закрити".
protected void MMCloseClick (object who, EventArgs e)
{
MessageBox.Show ("Неактивна команда", "Заглушка",
MessageBoxButtons.OK);
}

// Оброблювач для команди меню "Вийти".
protected void MMExitClick (object who, EventArgs e)
{
DialogResult result = MessageBox.Show ("Зупинити програму?",
"Завершення",
```

```
        MessageBoxButtons.YesNo);
        if (result == DialogResult.Yes) Application.Exit ();
        }
}
```

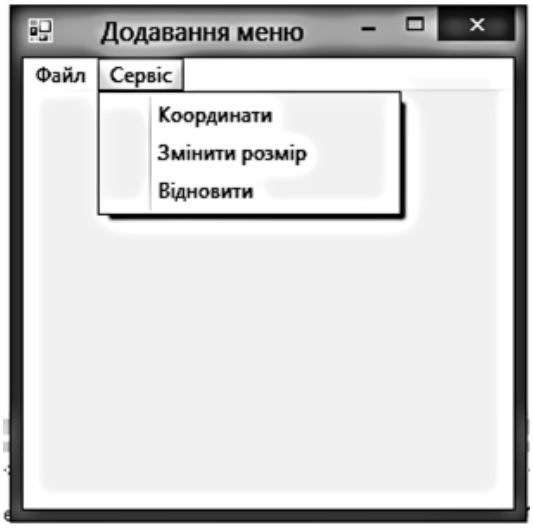
Результат виконання програми наведено на рис. 7.



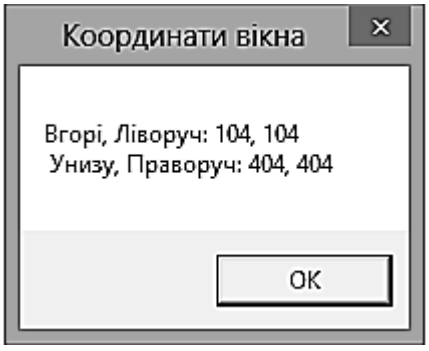
Після запуску програми



Виконана команда "Файл"



Виконана команда "Сервіс"



Виконані команди "Сервіс" / "Координати"

Рис. 7. **Результат виконання програми, яка створює основне меню**

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання приклад програми, який був наведений в розділі "Основні положення" даної лабораторної роботи.

2. Проекспериментуйте з програмою.

Індивідуальна частина

Модифікувати програму з прикладу 5 таким чином, щоб головне меню мало N пунктів, а кожне спадне меню – по M опцій. Значення N і M, а також найменування відповідних пунктів погодити з викладачем.

Під час вибору кожного з пунктів меню, що розкривається, повинна бути сформована відповідна програмна заглушка.

Зміст звіту

1. Титульний аркуш.
2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.
3. Текст налагодженої програми загальної частини лабораторного заняття з необхідними коментарями і результатом виконання.
4. Текст налагодженої програми з результатом виконання індивідуального завдання.
6. Висновки.

Контрольні запитання

1. У чому відмінність графічних додатків від консольних?
2. Опишіть структуру найпростішого графічного додатка.
3. Які оператори необхідно використовувати для створення і розміщення на формі кнопки?
4. Як здійснюється оброблення подій у Windows-додатку?
5. Перерахуйте можливі варіанти інтерфейсу вікна повідомлень.
6. Опишіть алгоритм створення головного меню Windows - додатка.

Лабораторна робота 13
Розроблення MDI- і SDI-додатків за допомогою компонентів Designer Forms

Мета роботи – отримати практичні навички роботи щодо створення MDI- і SDI-додатків за допомогою компонентів Designer Forms.

Дана лабораторна робота сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

складу і призначення типових компонентів Designer Forms;

технології створення MDI- і SDI-додатків;

особливостей застосування діалогових вікон;

уміння:

використовувати стандартні компоненти Designer Forms;

розробляти MDI- і SDI- Windows-додатки;

комунікації:

аргументована взаємодія з клієнтами та замовниками під час вибору технології розроблення MDI- і SDI- Windows додатків;

робота в команді над окремими фрагментами MDI- і SDI- Windows додатків;

автономність і відповідальність:

самостійне формулювання рекомендацій щодо вибору компонентів Designer Forms для створення інтерфейсу користувача;

здатність обґрунтувати доцільності застосування певної технології для створення MDI- і SDI- Windows-додатків.

Основні положення

Елементи управління – це компоненти, що забезпечують взаємодію між користувачем і програмою. Середа Visual Studio.NET надає велику кількість елементів, які можна згрупувати за кількома функціональними групами.

Група командних об'єктів

Елементи управління Button, LinkLabel, ToolBar реагують на натискання кнопки мишки і негайно запускають яку-небудь дію. Найбільш поширена група елементів.

Група текстових об'єктів

Більшість додатків надають можливість користувачеві вводити текст і, у свою чергу, виводять різну інформацію у вигляді текстових записів. Елементи TextBox, RichTextBox приймають текст, а елементи Label, StatusBar виводять інформацію. Для оброблення введеного користувачем тексту, як правило, слід натиснути на один або декілька елементів із групи командних об'єктів.

Група перемикачів

Додаток може містити кілька визначених варіантів виконання дії або завдання; елементи управління цієї групи надають можливість вибору користувачеві. Це одна із самих великих груп елементів, в яку входять ComboBox, ListBox, ListView, TreeView, NumericUpDown і багато інших.

Група контейнерів

З елементами цієї групи дії додатки практично ніколи не зв'язуються, але вони мають велике значення для організації інших елементів управління, їх угруповання і загального дизайну форми. Як правило, елементи цієї

групи, розташовані на формі, служать підкладкою кнопок, текстових полів, списків – тому вони й називаються контейнерами. Елементи Panel, GroupBox, TabControl, крім усього іншого, поділяють можливості програми на логічні групи, забезпечуючи зручність роботи.

Група графічних елементів

Навіть найпростіше додаток Windows містить графіку – іконки, заставку, вбудовані зображення. Для розміщення і відображення їх на формі використовуються елементи для роботи з графікою – Image List, Picture Box.

Діалогові вікна

Виконуючи різні операції з документом — відкриття, збереження, друк, попередній перегляд, — ми стикаємося з відповідними діалоговими вікнами. Розробникам .NET не доводиться займатися створенням вікон стандартних процедур: елементи OpenFileDialog, SaveFile Dialog, ColorDialog, PrintDialog містять вже готові операції.

Група меню

Багато користувачів налаштовують інтерфейс додатків на свій смак: одним подобається наявність певних панелей інструментів, іншим – індивідуальне розташування вікон. Але в будь-якому додатку буде присутнє меню, що містить доступ до всіх можливостей і налаштувань програми. Елементи MainMenu, ContextMenu є готовими формами для внесення заголовків і пунктів меню.

Далі наведено декілька прикладів, які дозоляють залучити практичні навички розроблення Windows додатків за допомогою Designer Forms.

Приклад 1. Дослідження керуючого елемента Button

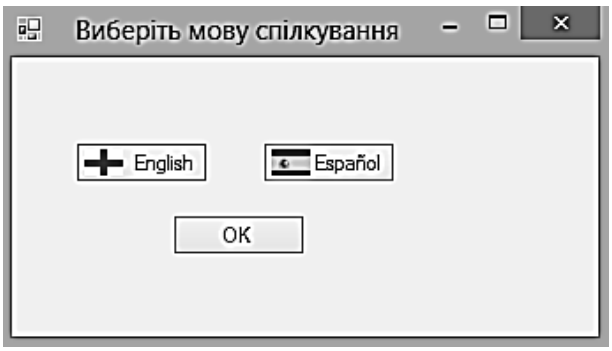
У прикладі буде створено діалог, у якому використовуються три кнопки.

Перші дві кнопки (рис. 8) будуть змінювати мову спілкування з англійської на іспанську і навпаки (можна використовувати будь-яку іншу мову на свій розсуд). Остання кнопка (OK) буде використовуватися для завершення діалогу.

Технологія створення користувальницького інтерфейсу.

Крок 1. Відкрийте Visual Studio.NET і створіть новий додаток C# Windows Application. Назвіть це додаток ButtonTest.

Крок 2. Розгорніть Toolbox і виконайте клацання мишкою на елементі Button. Потім пересуньте кнопки і встановіть відповідний розмір форми, як подано на рис. 9.



Початковий стан інтерфейсу

Натиснута кнопка English

Натиснута кнопка Español

Рис. 8. **Результат виконання програми**

Крок 3. Клацніть правою кнопкою мишки на вкладці Properties. Змініть властивість Name для всіх трьох кнопок таким чином:
button1 → btnEnglish;
button2 → btnEspan;
button3 → btnOK.

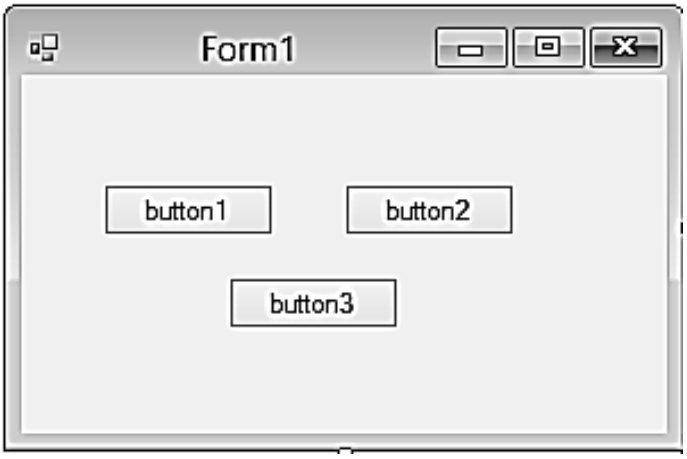


Рис. 9. **Початкова форма додатка**

Крок 4. Змініть властивість Text кожної з трьох кнопок на відповідний текст:

button1 → English;

button2 → Español

button3 → OK.

Змініть ім'я форми "Form1" на "Виберіть мову спілкування" (рис. 10).

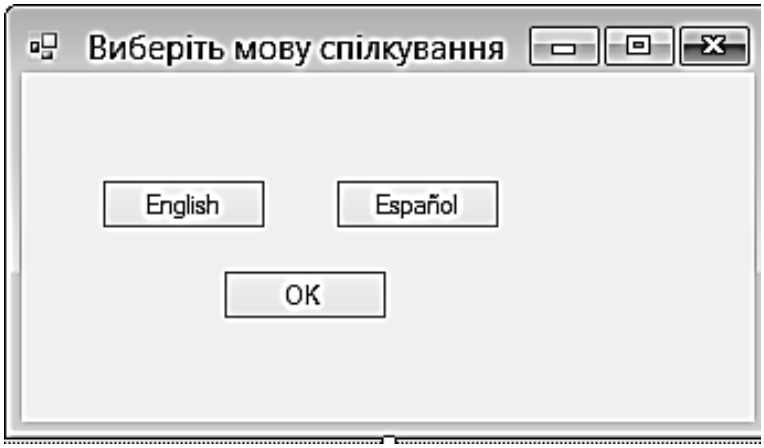


Рис. 10. **Результат виконання кроку 4**

Крок 5. Перед текстом необхідно вивести відповідний прапорець, щоб було зрозуміло, про що йдеться.

Виберіть кнопку English і знайдіть властивість image (рис. 11).

Клацніть мишкою праворуч від image, для того щоб перейти в діалогове вікно, в якому можна вибирати малюнки.

Іконки з різними прапорами поставляються разом з Visual Studio.NET. Якщо ви встановили Visual Studio.NET на стандартне місце (мається на увазі англомовна версія), то вони повинні розташовуватися в директорії C \ Program Files \ Microsoft Visual Studio.NET \ Common7 \ Graphics \ icons \ Flags.

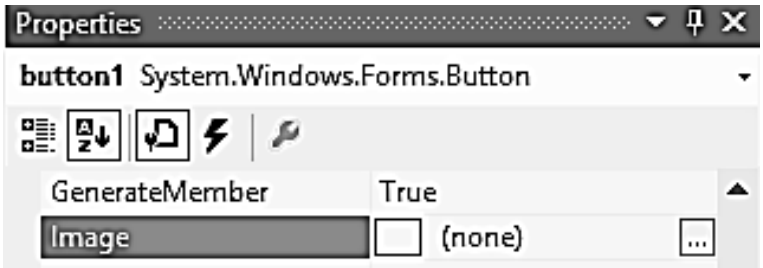


Рис. 11. **Знаходження властивості image на панелі Properties**

У іншому разі необхідно імпортувати іконку з папки, яка додається до лабораторної роботи (рис. 12).

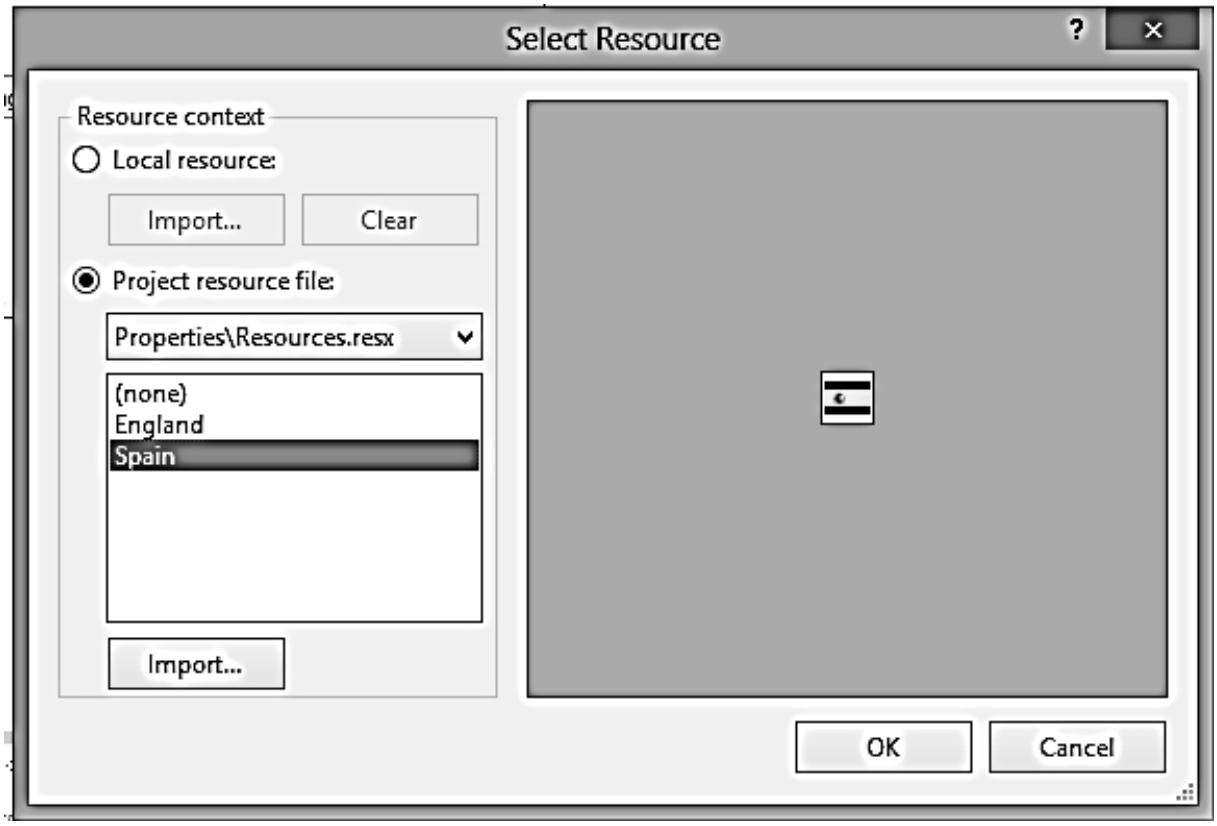


Рис. 12. **Вікно імпортування іконки**

Виберіть відповідну іконку. Повторіть ту ж саму процедуру для кнопки Español (за бажанням можна вибрати будь-який інший прапор з наявних у даній директорії). Результат надано на рис. 13.

Рис. 13. **Результат виконання кроку 5**

Крок 6. На даний момент текст і іконка, які розташовані на кнопці, накладаються один на одного, тому необхідно змінити місця їх розташування. Для обох кнопок змініть (рис. 14):

значення властивості ImageAlign на MiddleLeft;

значення властивості TextAlign на MiddleRight.

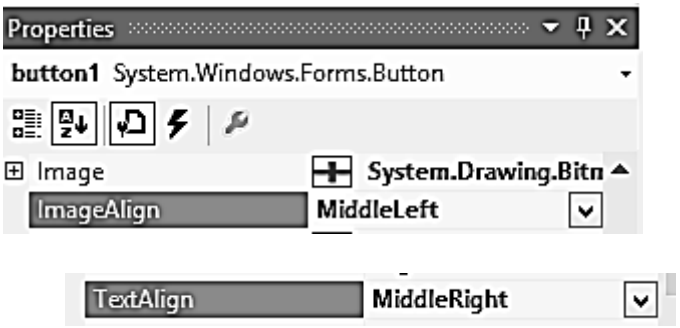


Рис. 14. **Налаштування властивостей ImageAlign та TextAlign**

Крок 7. У цей момент можна скорегувати ширину кнопок, з тим, щоб текст не починався безпосередньо в тому місці, де закінчується зображення. Для цього виберіть кожну кнопку і розтягніть її правий край.

Результат повинен виглядати таким чином (рис. 15).

Додавання обробників подій.

Крок 8. Клацніть мишкою два рази на кнопці English ─ ви потрапите безпосередньо на обробник події.

Click є подією, яка використовується для кнопок за замовчуванням; це саме та подія, яка створюється у разі подвійного натискання мишкою на кнопці.

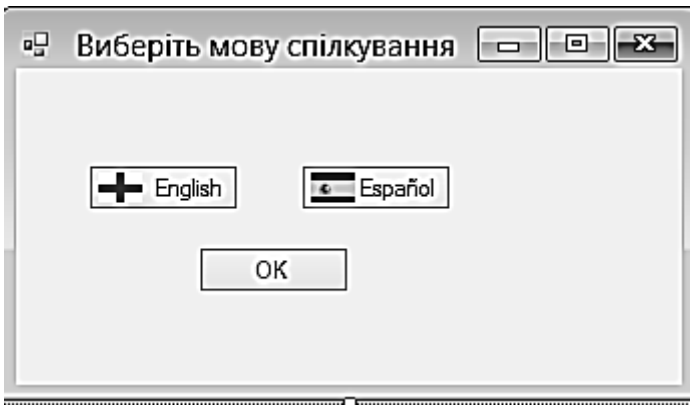


Рис. 15. **Інтерфейс користувача**

Під час подвійного натискання мишкою на керуючому елементі в коді, який лежить в основі даної форми, відбуваються дві речі.

По-перше, в методі InitializeComponent () створюється підписка на подію (додається обробник btnEnglish_Click в список подій):

this.btnEnglish.Click + = new System.EventHandler (this.btnEnglish_Click);

Якщо виникає необхідність підписатися на подію, відмінну від події за замовчуванням, то доведеться заносити відповідний код самостійно.

Необхідно пам'ятати, що код методу InitializeComponent () перезаписується кожен раз, коли ви перемикаєтеся з режиму розроблення і переходите до коду. З цієї причини ніколи не виконуйте підписку в цьому методі. Замість цього слід застосовувати конструктор даного класу.

Друге, що відбувається, — це додається власне обробник події:

```
private void btnEnglish_Click (object sender, System.EventArgs e)
{

}
```

Ім'я методу утворюється за допомогою об'єднання імені керуючого елемента, символу підкреслення та імені події, яка повинна обробляться.

Перший параметр – object sender – містить обраний керуючий елемент. У даному прикладі, це завжди буде елемент, який входить в ім'я методу, хоча в деяких інших випадках кілька керуючих елементів можуть використовувати один і той же метод для оброблення події, і в такому випадку за значенням цього параметра можна точно визначити, який саме елемент викликає даний метод.

У другому параметрі – System.EventArgs e – міститься інформація про те, що саме сталося. У даному випадку немає необхідності використовувати інформацію, що зберігається в обох параметрах.

Ключове слово this, позначає поточний екземпляр даного класу. Оскільки клас, з яким ми працюємо зараз, є саме таким екземпляром, тому можна отримувати доступ до його властивостей і керуючим елементам за допомогою цього слова.

Задаємо значення властивості Text поточного екземпляра форми:

```
private void btnEnglish_Click (object sender, System.EventArgs e)
{
        this.Text = "Do you speak English?";
}
```

Крок 9. Поверніться в Form Designer і клацніть мишкою два рази на кнопці Español для переходу до обробника подій для цієї кнопки. Ось його код:

```
private void btnEspan_Click (object sender, System.EventArgs e)
{
        this.Text = "Usted puede hablar español ?";
}
```

Цей метод ідентичний методу btnEnglish_Click за винятком тексту іспанською мовою.

Крок 10. Оброблювач подій для кнопки ОК додається таким же засобом, як і два попередніх. Однак код в цьому випадку буде трохи відрізнятися:

```
private void btnOK_Click (object sender, System.EventArgs e)
{
        Application.Exit ();
}
```

За допомогою цього методу завершується виконання програми. Повний текст файла ButtonTest.Form1 наведено далі:

```
using System;
using System.Windows.Forms;

namespace ButtonTest
{
   public partial class Form1: Form
   {
      public Form1()
      {
         InitializeComponent();
      }
      private void btnEnglish_Click(object sender, EventArgs e)
      {
         this.Text = "Do you speak English?";
      }
      private void btnEspan_Click(object sender, EventArgs e)
      {
```

```
            this.Text = "Usted puede hablar español  ?";
        }
        private void btnOK_Click(object sender, EventArgs e)
        {
            Application.Exit();
        }
    }
}
```

Результат виконання програми відповідає рис. 8.

Приклад 2. Розроблення Windows-додатка "Блокнот"

Необхідно розробити додаток "Блокнот", який повторює в найпростішому варіанті функції стандартної однойменної Windows-програми.

Створення інтерфейсу головного меню

Більшість Windows-додатків має головне меню, яке є ієрархічною структурою функцій і команд. Практично всі функції, які можна здійснити за допомогою елементів управління, мають свій аналог у вигляді відповідних пунктів меню.

Далі наведена покрокова технологія розроблення основного меню додатка "Блокнот" за допомогою стандартних компонентів панелі Designer Forms.

Крок 1. Налаштування початкової форми та панелі Designer Forms.

Створіть новий додаток і назвіть його NotepadCSharp.

Установіть такі властивості форми:

Name → frmmain
Icon → …\Icon\README.ICO
Text → Notepad C#
WindowState → Maximized

Результат наведено на рис. 16.

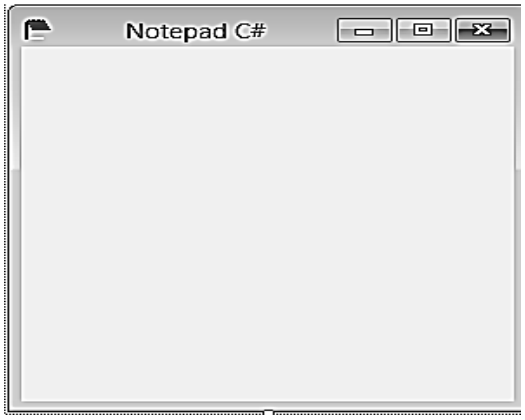


Рис. 16. **Початкова форма додатка Блокнот**

Перетягуємо елемент управління MainMenu, який знаходиться на панелі інструментів Toolbox, на форму (рис. 17).

Якщо елемент MainMenu відсутній (в версії Visual Studio 2012) можна застосовувати елемент MenuStrip, але йому бракує декілька необхідних надалі властивостей.

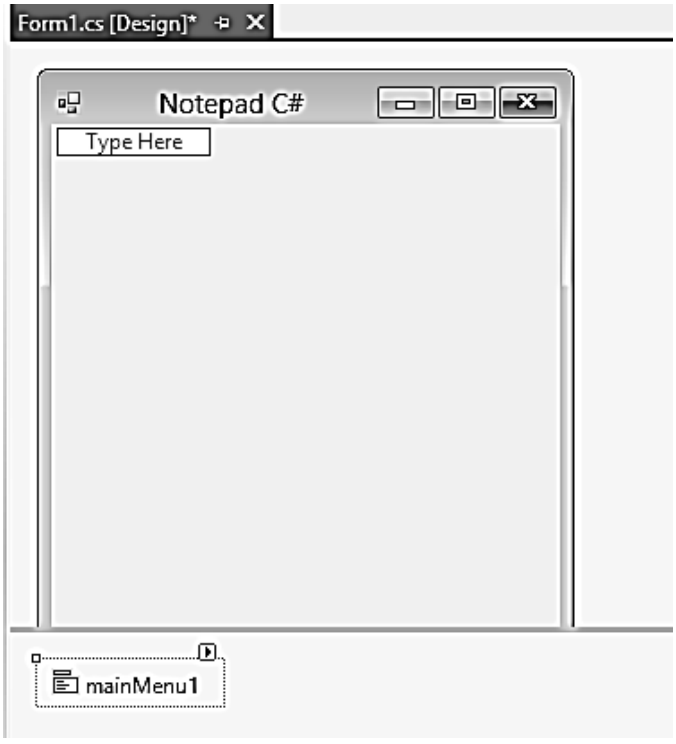


Рис. 17. **Результат виконання кроку 1**

Для додавання у панель Toolbox компонента MainMenu необхідно з контекстного меню вкладки Menu & Toolbars викликати команду Choose Items… Далі у вікні необхідно відмітити елемент Menu (рис. 18).

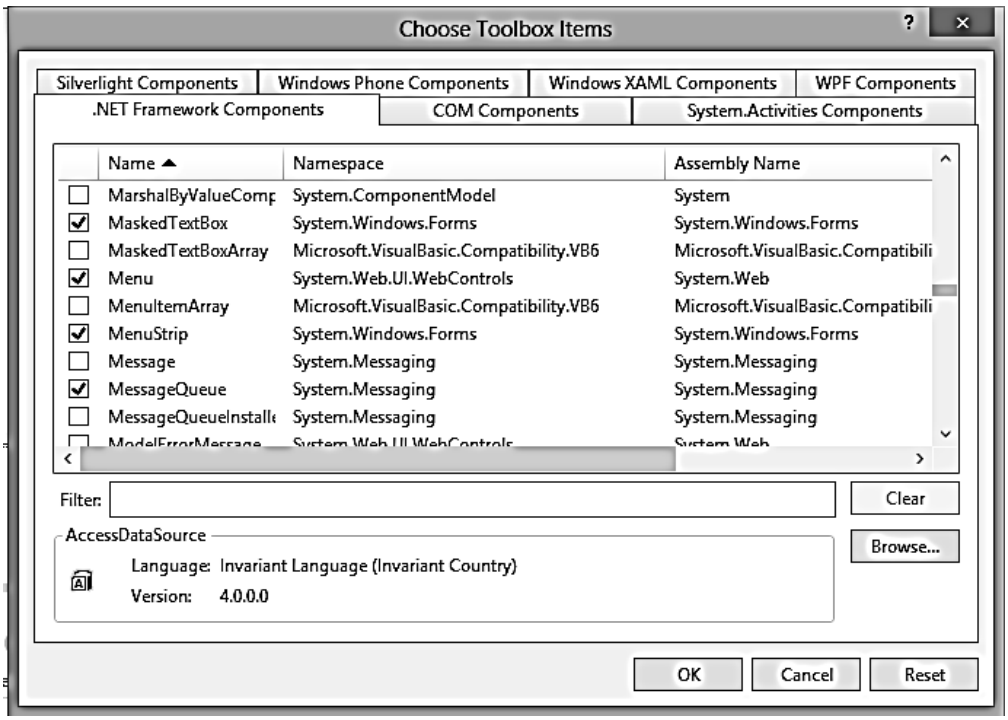


Рис. 18. **Вікно додавання елемента MainMenu в панель Toolbox**

Як результат у панелі Toolbox з'явиться необхідний нам компонент.

Крок 2. Створення інтерфейсу головного меню.

Необхідно заповнити рядки меню такими пунктами (рис. 19).

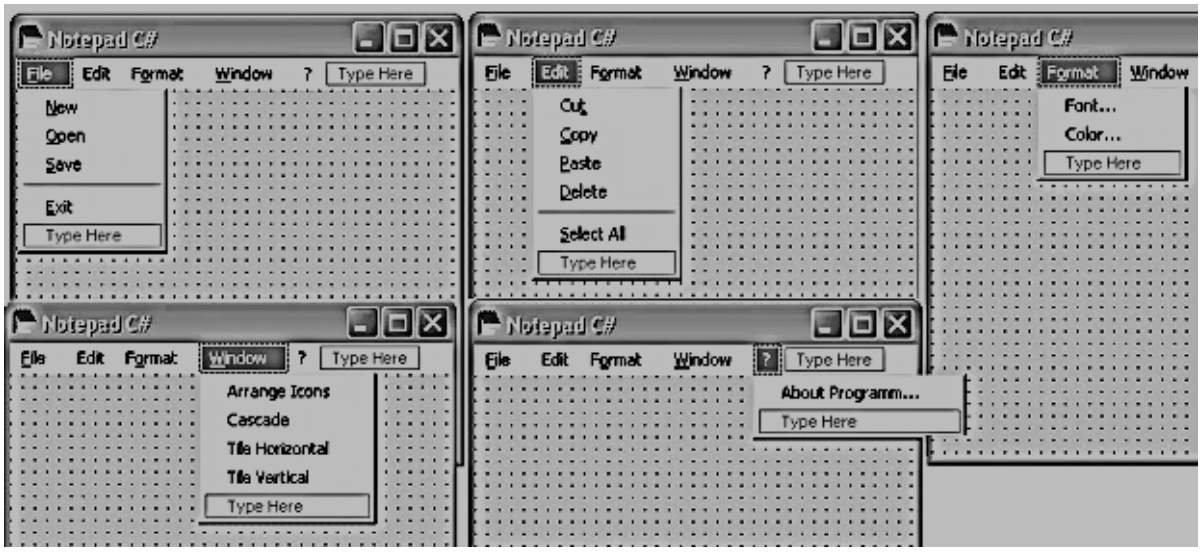


Рис. 19. **Пункти головного меню програми Notepad C#**

Кожен пункт головного меню має своє вікно властивостей, в якому, подібно іншим елементам управління, задаються значення властивостей Name, Text і Shortcut (рис. 20).

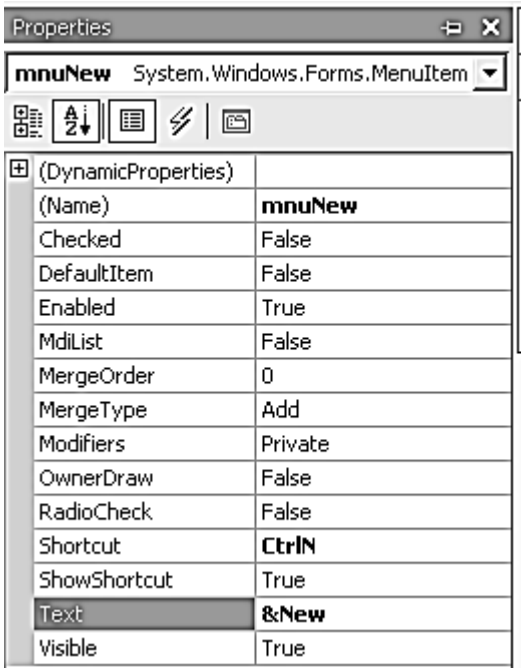


Рис. 20. **Властивості пункту меню New**

У полі Text перед словом New стоїть знак & – так званий амперсанд, який вказує, що N повинно бути підкреслена і буде частиною вбудованого клавіатурного інтерфейсу Windows. Коли користувач на клавіатурі натискає клавішу Ctrl і потім N, виводиться підменю New.

У Windows є ще інтерфейс для роботи з так званими швидкими клавішами, або акселераторами. Поєднання клавіш вказують із перерахування Shortcut.

Горизонтальна розділова лінія використовується в тих випадках, коли треба візуально відокремити подібні групи завдань.

Для використання пунктів меню в коді, їм також призначають імена (властивість Name).

Слід призначати стандартним пунктам загальноприйняті сполучення клавіш. Властивості пунктів меню в додатку Notepad C # наводяться в табл. 2.

Таблиця 2

Пункти головного меню додатку Notepad C#

Name	Text	Shortcut
1	2	3
mnuFile	File	
mnuNew	&New	CtrlN
mnuOpen	&Open	CtrlO

Закінчення табл. 2

1	2	3
mnuSave	&Save	CtrlS
mnuExit	&Exit	CtrlE
mnuEdit	Edit	
mnuCut	Cut	
mnuCopy	Copy	
mnuPaste	Paste	
mnuDelete	Delete	
mnuSelectAll	SelectAll	
mnuFormat	Format	
mnuFont	Font…	
mnuColor	Color…	
mnuWindow	Window	
mnuArrangeIcons	Arrange Icons	
mnuCascade	Cascade	
mnuTileHorizontal	Tile Horizontal	
mnuTileVertical	Tile Vertical	
mnuHelp	?	
mnuAbout	About Programm...	

Можна самостійно вибрати поєднання клавіш, які не зазначені в табл. 2, для відповідних пунктів меню.

Результат виконання програми у вигляді відповідного інтерфейсу наведено на рис. 21.

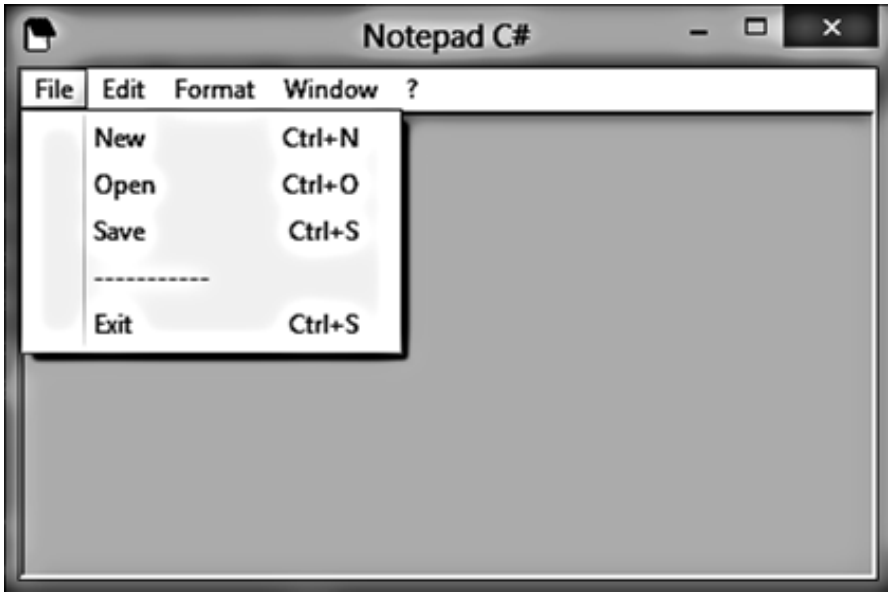


Рис. 21. **Інтерфейс програми "Блокнот"**

Створення MDI-додатків

Такі програми, як блокнот і Microsoft Paint, належать до SDI (Single – Document Interface) додаткам, здатним працювати тільки з одним документом. Інші, такі як, Microsoft Word або Adobe Photoshop, підтримують роботу відразу з декількома документами і називаються MDI (Multiple – Document Interface) додатками.

У MDI-додатках головна форма містить кілька документів, кожен з яких є полотном у графічних програмах або полем для тексту в редакторах.

Продовжимо роботу над додатком Notepad C#. Наше завдання – на базі вже розробленого інтерфейсу програми "Блокнот" створити багатовіконний (MD)I додаток.

Спочатку (перший етап) необхідно додати до основної форми додаткову (дочірню) форму і виконати відповідне налаштування її властивостей. Після чого (другий етап) потрібно підключити до кожного управляючого елемента меню відповідні обробники подій.

Етап 1. Створення і налаштування дочірній форми.

Крок 1. Створення пустої форми blank.

У вікні Solution Explorer клацаємо правою кнопкою на імені проекту і в контекстному меню вибираємо команду Add / Add Windows Form ....

У вікні, яке з'явилось, присвоюємо формі ім'я blank.cs (рис. 22).

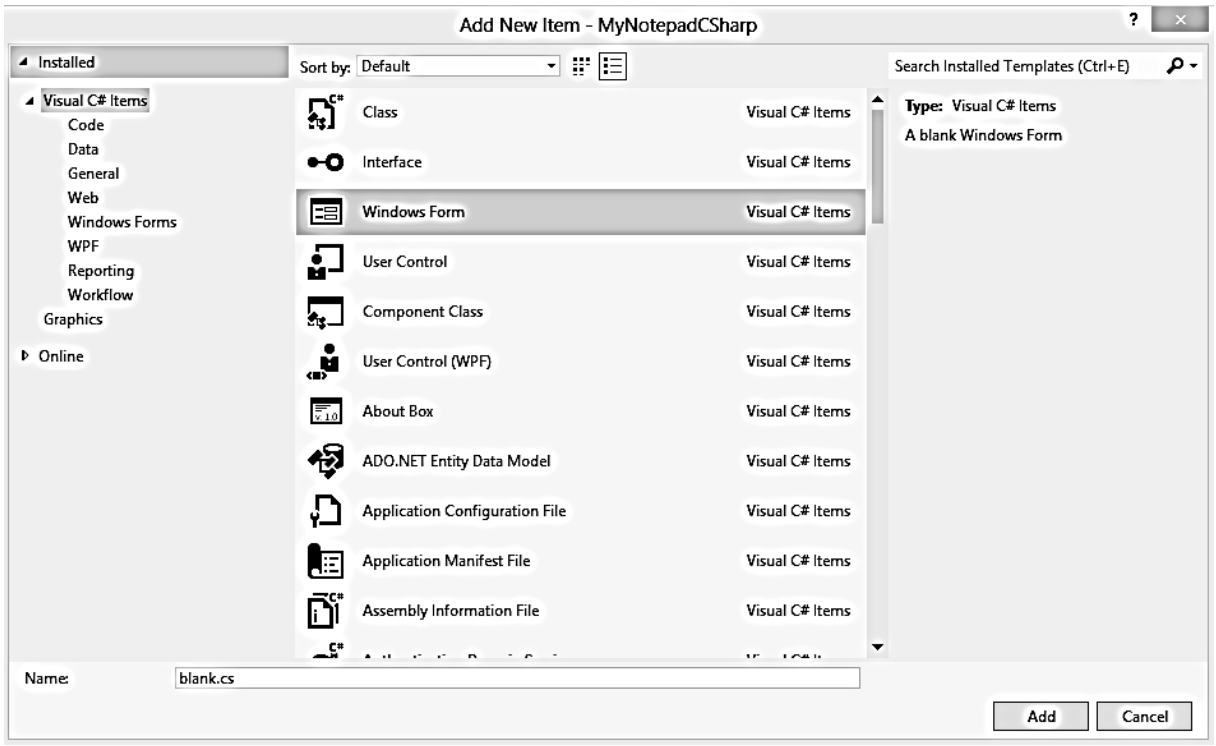


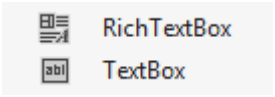
Рис. 22. **Вікно створення пустої форми blank**

У даному проекті з'явилася нова форма — будемо називати її дочірньою формою.

У вікні Solution Explorer ім'я головної форми Form1 замінимо на frmmain (рис. 23).

Крок 2. Налаштування властивостей форми blank.

У режимі дизайну перетягуємо на форму blank елемент управління RichTextBox:



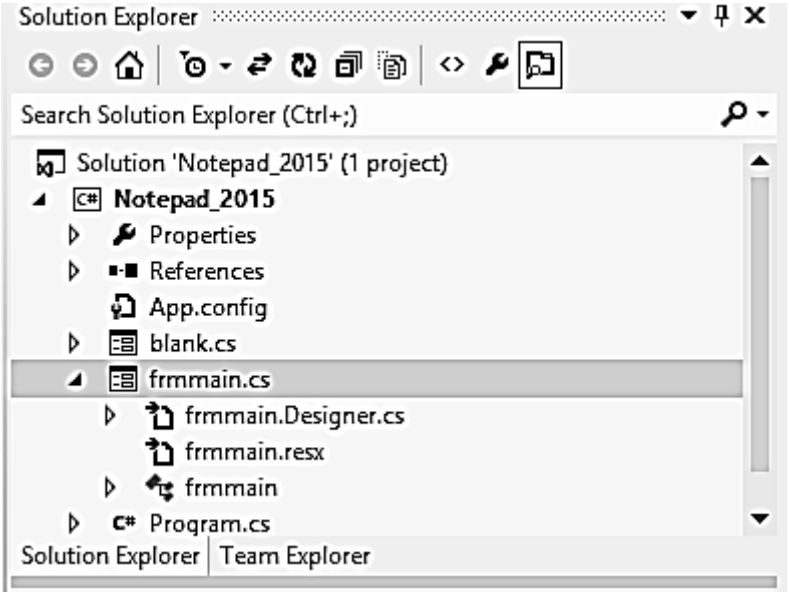


Рис. 23. **Вікно інспектора Solution Explorer**

На відміну від елемента textBox, розмір вмісту тексту в ньому не обмежується 64 Кб; крім того, RichTextBox дозволяє редагувати колір тексту та додавати зображення.

Властивість Dock цього елемента фіксуємо значенням Fill (рис. 24).

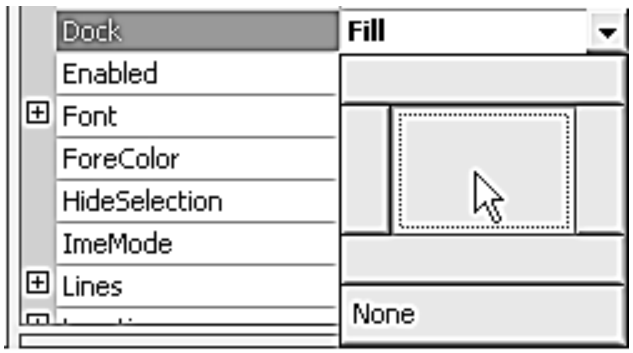


Рис. 24. **Властивість Dock елемента RichTextBox**

Крок 3. Налаштування кольору основної форми.

Переходимо в режим дизайну форми frmmain і встановлюємо властивості IsMdiContainer значення true (рис. 25). Колір форми при цьому стає темно-сірим.

Етап 2. Підключення обробників подій.

Крок 1. Обробник події кнопки New.

Нові екземпляри документів повинні з'являтися під час натискання пункту меню New (або поєднання клавіш Ctrl + N).

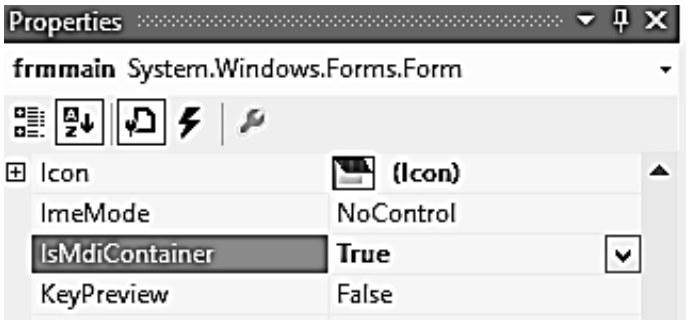


Рис. 25. **Налаштування кольору основної форми**

Двічі клацаємо на пункті New і переходимо в шаблон обробника події. Після чого записуємо в шаблон такий код:

```
private void mnuNew_Click(object sender, EventArgs e)
    {
       // Створюємо новий екземпляр форми frm
       blank frm = new blank();
       // Вказуємо, що батьківським контейнером нового екземпляру
       //буде ця, головна форма
       frm.MdiParent = this;
       // Викликаємо форму
       frm.Show();
    }
```

Запускаємо програму. Тепер кожен раз при натисканні клавіш Ctrl + N або виборі пункту меню New з'являється кілька вікон, розташованих каскадом. Однак заголовок у всіх однаковий – blank (рис. 26).

Крок 2. Привласнення кожному екземпляру документа відповідного номера.

Для того, щоб кожному екземпляру нового документа присвоювався свій номер необхідно виконати такі дії.

Перемикаємося в код форми blank, і в класі blank оголошуємо змінну DocName:

```
public partial class blank : Form
   {
      public string DocName = "";
      public blank()
      {
         InitializeComponent();
      }
   }
```

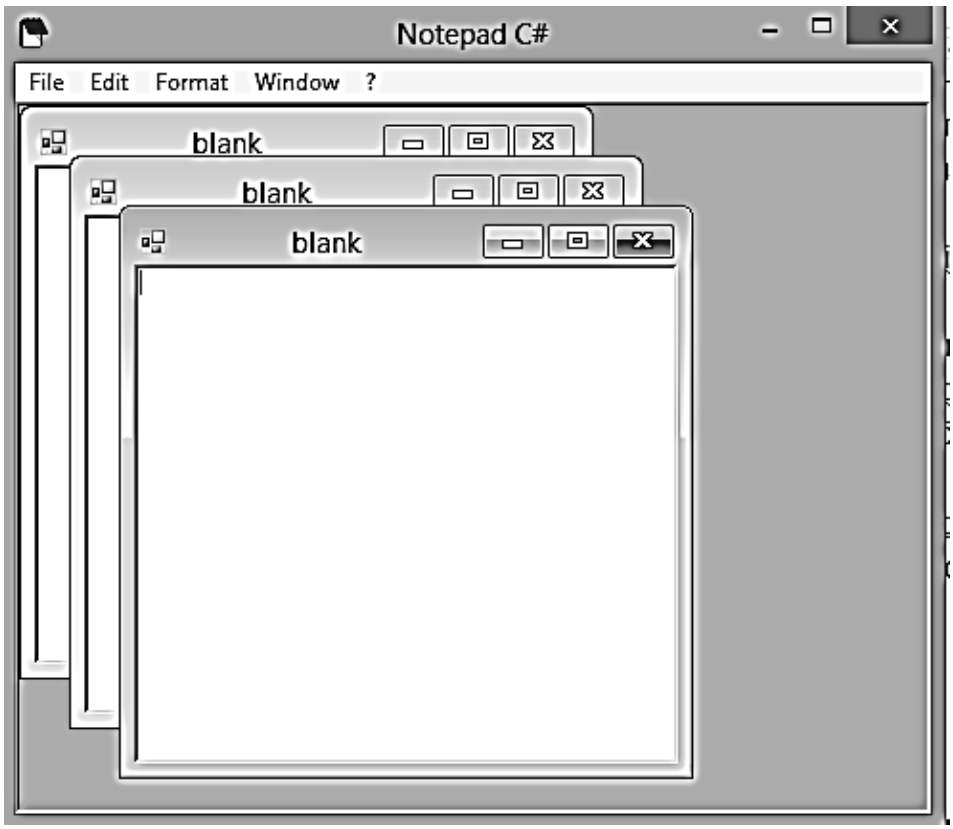


Рис. 26. **Результат виклику дочірніх вікон**

Перемикаємося в код форми frmmain і в класі frmmain оголошуємо змінну openDocuments:

```
public partial class frmmain : Form
   {
      private int openDocuments = 0;
      public frmmain()
      {
```

```
        InitializeComponent();
    }
...
```

Модифікуємо обробник події mnuNew_Click таким чином:

```
private void mnuNew_Click(object sender, EventArgs e)
    {
        // Створюємо новий екземпляр форми frm
        blank frm = new blank();
        frm.DocName = "Untitled " + ++openDocuments;
        // Вказуємо, що батьківським контейнером
        // нового примірника буде ця, головна форма.
        frm.MdiParent = this;
        frm.Text = frm.DocName;
        // Викликаємо форму
        frm.Show();
    }
```

Запускаємо програму. Тепер нові документи містять різні заголовки (рис. 27).

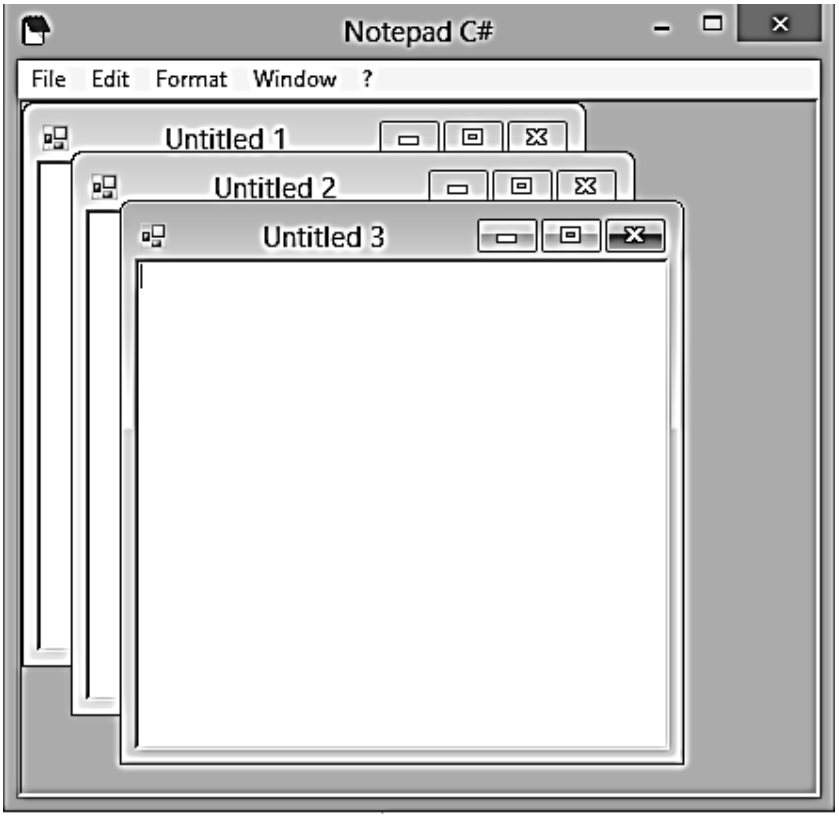


Рис. 27. **Нові документи, які мають впорядковані назви**

Крок 3. Упорядкування документів на екрані. Обробники подій пункту Window.

Під час роботи з декількома документами в MDI-додатках зручно впорядковувати їх на екрані. Можна, звичайно, розподіляти форми вручну, але під час роботи з великою кількістю документів це є скрутним. У пункті меню Window реалізуємо процедуру вирівнювання вікон, для чого створюємо відповідні обробники:

```
private void mnuArrangeIcons_Click(object sender, EventArgs e)
    {
       this.LayoutMdi(MdiLayout.ArrangeIcons);
    }

    private void mnuCascade_Click(object sender, EventArgs e)
    {
       this.LayoutMdi(MdiLayout.Cascade);
    }

    private void mnuTileHorizontal_Click(object sender, EventArgs e)
    {
       this.LayoutMdi(MdiLayout.TileHorizontal);
    }

    private void mnuTileVertical_Click(object sender, EventArgs e)
    {
       this.LayoutMdi(MdiLayout.TileVertical);
    }
```

Як слід з обробників, метод LayoutMdi містить перерахування MdiLayout, що містить чотири відповідні члени.

Під час вибору пункту ArrangeIcons фокус повинен перемикатись на обрану форму. Для чого у властивості MdiList пункту меню ArrangeIcons необхідно встановити значення true.

Тепер під час відкриття кількох нових документів вікна розташовуються каскадом (рис. 28), їх можна розташувати горизонтально ─ значення TileHorizontal (рис. 29) або вертикально ─ значення TileVertical (рис. 30), а потім знову повернути каскадне розташування ─ Cascade

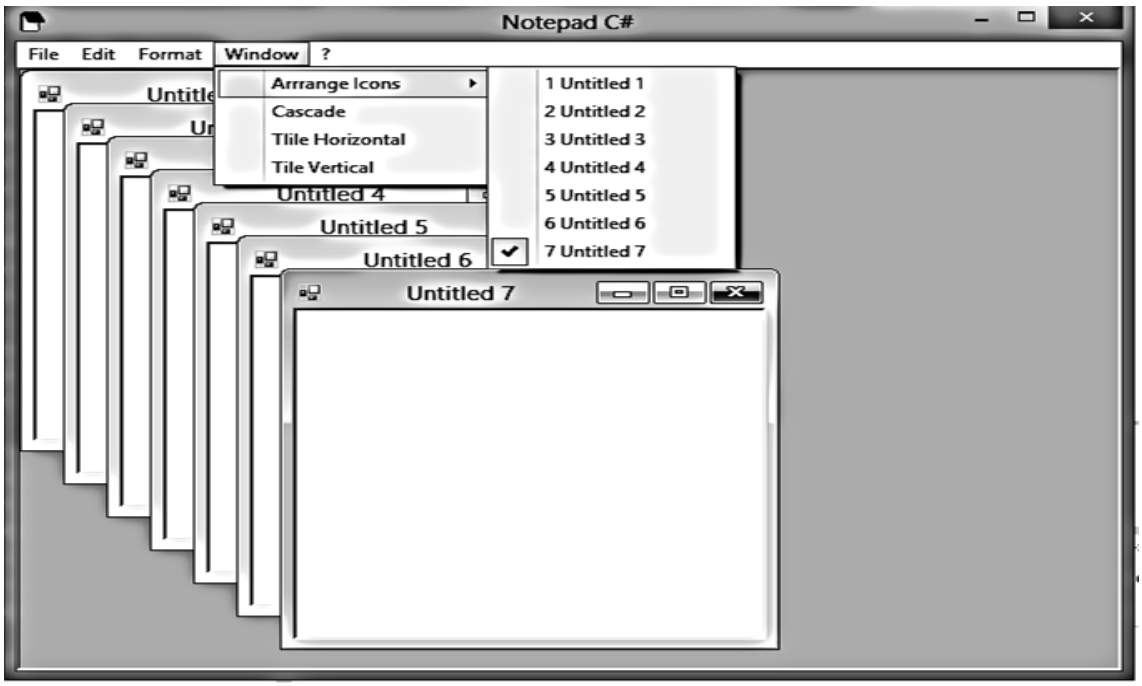


Рис. 28. **Під час відкриття кількох нових документів, вікна розташовуються каскадом**

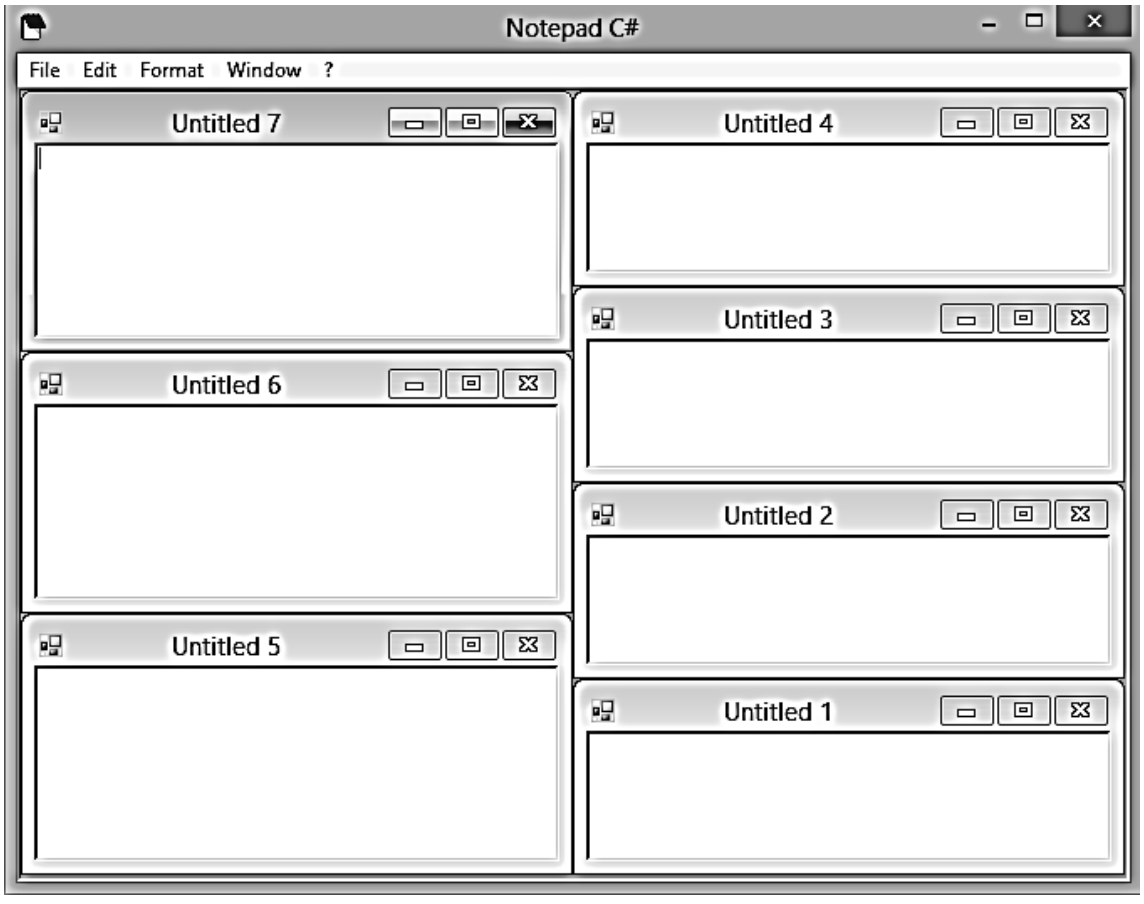


Рис. 29. **Вікна розташовуються горизонтально**

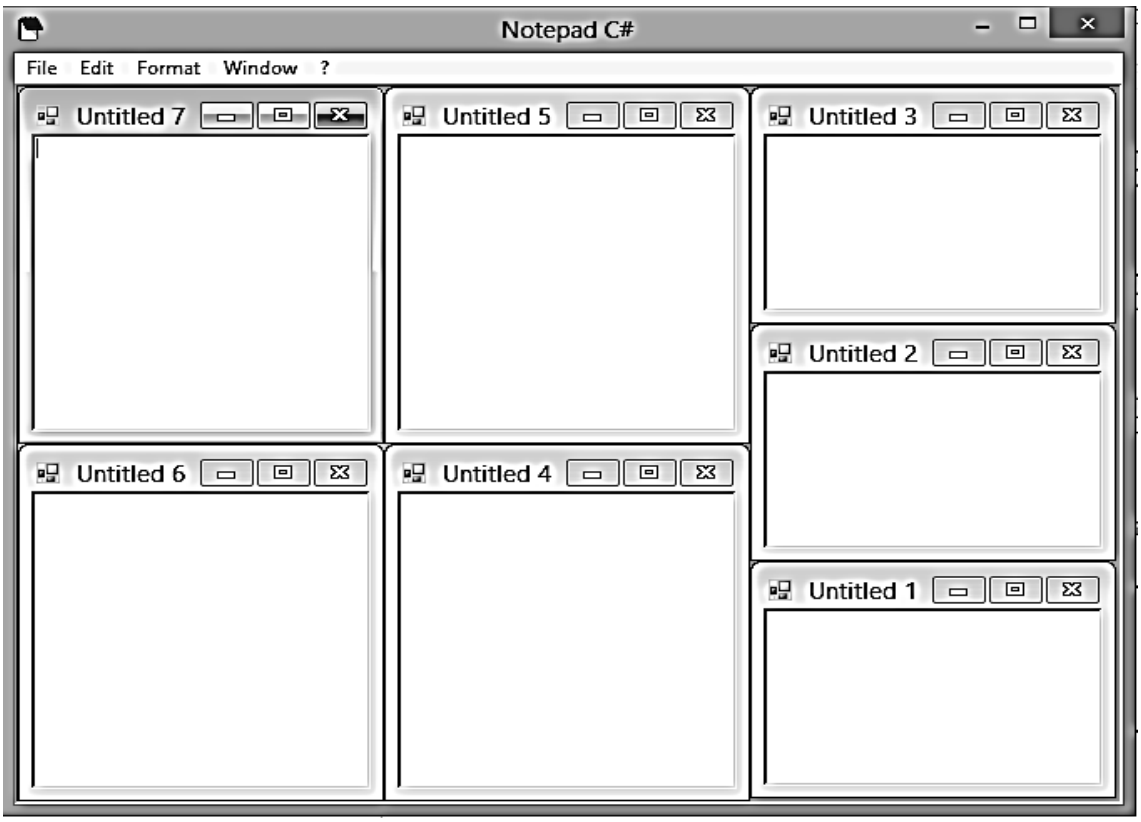


Рис. 30. **Вікна розташовуються вертикально**

Вирізання, копіювання і вставка текстових фрагментів

З додатком працювати буде зручніше, якщо під час створення нового документа він відразу буде займати всю область головної форми. Для цього встановимо властивість WindowState форми blank Maximized.

Приступимо до створення обробників для стандартних операцій вирізання, копіювання і вставки. Елемент управління RichTextBox має властивість SelectedText, яка містить виділений фрагмент тексту. На підставі цієї властивості і будуть реалізовані дії роботи з текстом.

Крок 1. Оголошення буферної зміни.

У коді форми **blank** оголошуємо змінну (буфер) BufferText, в якій буде зберігатися фрагмент тексту, який потрібно запам'ятати для подальшого оброблення:

```
public partial class blank : Form
   {
      public string DocName = "";
      private string BufferText = "";
      public blank()
```

```
        {
            InitializeComponent();
        }
    }
```

Крок 2. У коді форми **blank** створюємо відповідні методи пункту Edit головного меню.

```
        // Вирізання тексту
        public void Cut()
        {
            this.BufferText = richTextBox1.SelectedText;
            richTextBox1.SelectedText = "";
        }
        // Копіювання тексту
        public void Copy()
        {
            this.BufferText = richTextBox1.SelectedText;
        }
        // Вставка
        public void Paste()
        {
            richTextBox1.SelectedText = this.BufferText;
        }
        // Виділення всього тексту — використовуємо властивість SelectAll елемента управління RichTextBox
        public void SelectAll()
        {
            richTextBox1.SelectAll();
        }
        // Видалення
        public void Delete()
        {
            richTextBox1.SelectedText = "";
            this.BufferText = "";
        }
```

Крок 3. Перемикаємося в режим дизайну форми **frmmain** і створюємо обробників для пунктів меню:

```
private void mnuCut_Click (object sender, System.EventArgs e)
    {
      blank frm = (blank) this.ActiveMdiChild;
      frm.Cut ();
    }

private void mnuCopy_Click (object sender, System.EventArgs e)
    {
    blank frm = (blank) this.ActiveMdiChild;
      frm.Copy ();
    }

private void mnuPaste_Click (object sender, System.EventArgs e)
    {
      blank frm = (blank) this.ActiveMdiChild;
      frm.Paste ();
    }

private void mnuDelete_Click (object sender, System.EventArgs e)
    {
      blank frm = (blank) this.ActiveMdiChild;
      frm.Delete ();
    }

private void mnuSelectAll_Click (object sender, System.EventArgs e)
    {
      blank frm = (blank) this.ActiveMdiChild;
      frm.SelectAll ();
    }
```

Властивість ActiveMdiChild перемикає фокус на поточну форму, якщо їх відкрито декілька, і викликає один із методів, визначених у дочірньої формі. Запускаємо програму. Тепер можна виконувати всі стандартні операції з текстом.

Контекстне меню

Контекстне меню, яке дублює деякі дії основного меню, – звичний інструмент для користувача. Елемент управління TextBox містить найпростіше

контекстне меню, яке дублює дії підміню Edit. Для того щоб переконатися в цьому, досить нанести цей елемент управління на форму і запустити додаток (рис. 31):

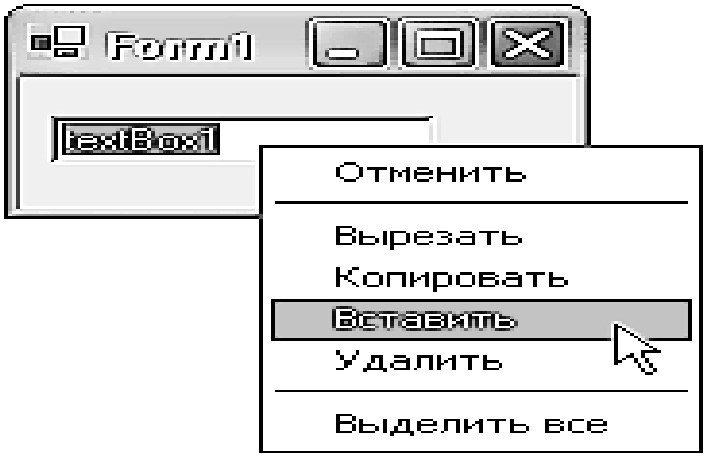


Рис. 31. **Контекстне меню елемента TextBox**

Крок 1.

У нашому додатку Notepad C# в якості текстового елемента ми використовуємо RichTextBox.

Перетягуємо елемент управління contextMenuStrip (у разі Visual Studio 2012) з вікна ToolBox на форму **blank**.

Крок 2. Додаємо пункти контекстного меню точно так само, як ми це робили для головного меню (рис. 32).

Не забуваємо перейменувати (властивість Name) відповідним чином кожен з пунктів меню:

Cut → cmnuCut;
Copy → cmnuCopy;
Paste → cmnuPaste;
Delete → cmnuDelete;
Select All → cmnuSelectAll;

Крок 3. Перемикаємося в **режим дизайну форми blank** і створюємо обробників для відповідних пунктів контекстного меню:

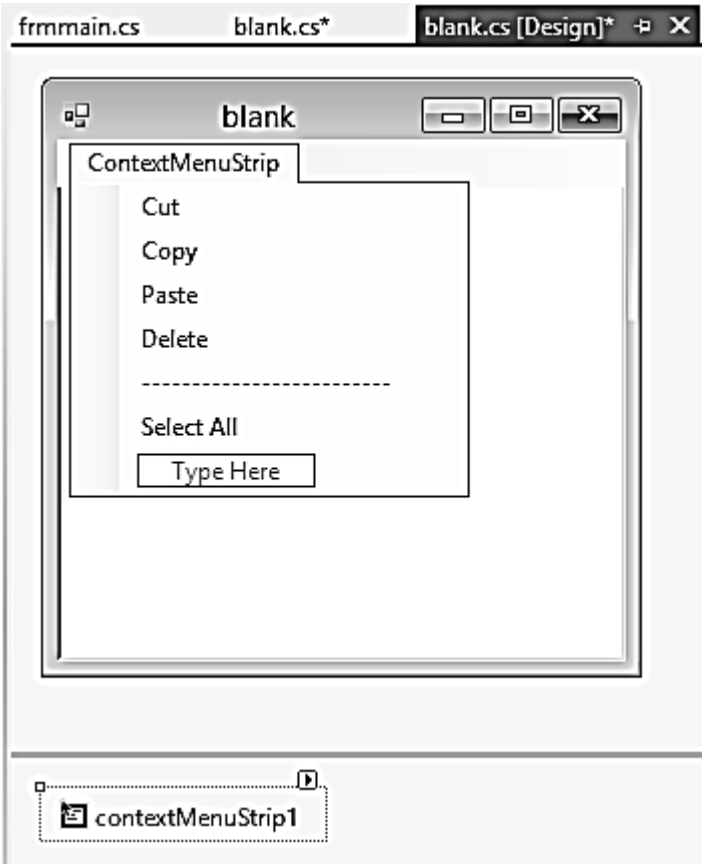


Рис. 32. **Пункти контекстного меню**

```
private void cmnuCut_Click(object sender, EventArgs e)
    {
       Cut();
    }

    private void cmnuCopy_Click(object sender, EventArgs e)
    {
       Copy();
    }

    private void cmnuPaste_Click(object sender, EventArgs e)
    {
       Paste();
    }

    private void cmnuDelete_Click(object sender, EventArgs e)
    {
       Delete();
    }
```

```
private void cmnuSelectAll_Click(object sender, EventArgs e)
{
    SelectAll();
}
```

Крок 4. Останнє, що нам залишилося зробити, – це визначити, де буде з'являтися контекстне меню.

Елемент RichTextBox, так само, як і форми frmmain і blank, має властивість ContextMenuStrip (рис. 33). Необхідно праворуч вибрати contextMenuStrip1, оскільки нам потрібно відображати меню саме в текстовому полі.

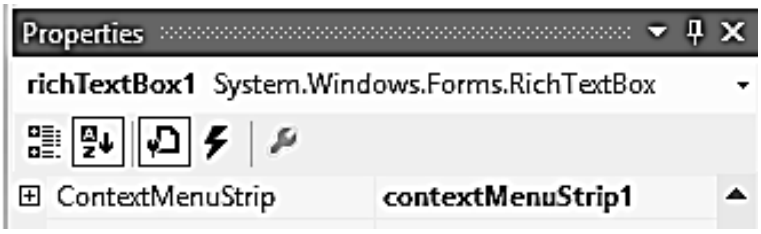


Рис. 33. **Налаштування місця, де буде з'являтися контекстне меню**

Запускаємо програму – тепер у будь-якій точці тексту доступно меню (рис. 34).

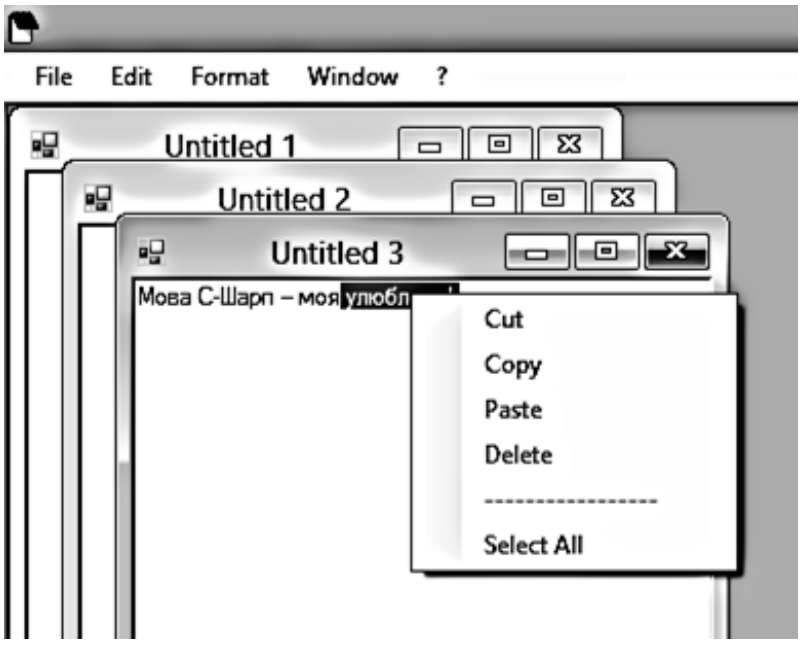


Рис. 34. **Виклик контекстного меню**

Діалогові вікна

Середа Visual Studio .NET містить готові діалогові вікна, що пропонують вибрати файл для відкривання або шлях до диску для збережень поточного файла. Розглянемо покрокові технології створення типових діалогових вікон.

OpenFileDialog

Крок 1. Налаштування властивостей вікна.

Додайте на форму frmmain елемент управління OpenFileDialog з вікна панелі інструментів ToolBox. Подібно елементу MainMenu, він буде розташовуватися на панелі невидимих компонентів.

В інспекторі властивостей елемента управління OpenFileDialog (рис. 35) властивість FileName задає назву файла, який буде знаходитися в полі "Ім'я файла" під час появи діалогу. На рисунку назва в цьому полі – "Текстові файли (.txt)", оскільки ми будемо вводити саме текст.

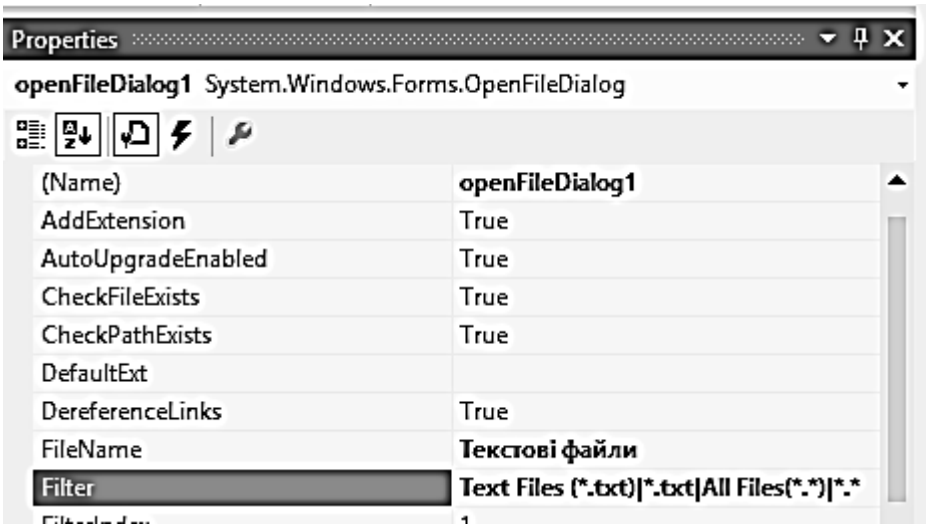


Рис. 35. **Налаштування властивостей вікна OpenFileDialog**

Властивість Filter задає обмеження файлів, які можуть бути обрані для відкриття – у вікні будуть показуватися тільки файли з заданим розширенням.

Тут введено Text Files (* .txt) | * .txt | All Files (*. *) | *. *, що означає огляд або текстових файлів, або всіх разом.

Властивість InitialDirectory дозволяє задати директорію, звідки буде починатися огляд. Якщо цю властивість не встановлено, вихідною директорією буде робочий стіл.

Крок 2. Робота з файловими потоками в коді форми **blank**.

2.1. Перемикаємося в режим дизайну форми blank. Для роботи з файловими потоками в коді форми blank підключаємо простір імен System.IO: using System.IO;

2.2. У методі Open записуємо вміст файла, який потрібно відкрити, в полі RichTextBox. Це робиться за допомогою такого методу:

```
// Створюємо метод Open, де як параметр оголошуємо рядок адресу файла.
    public void Open (string OpenFileName)
    {
      // Якщо файл не обраний, повертаємося назад (з'явиться вбудоване попередження)
      if (OpenFileName == "")
      {
        return;
      }
      else
      {
// Створюємо новий об'єкт StreamReader і передаємо йому змінну
                                                    // OpenFileName
        StreamReader sr = new StreamReader (OpenFileName);
        // Читаємо весь файл і записуємо його в richTextBox1
        richTextBox1.Text = sr.ReadToEnd ();
        // Закриваємо потік
        sr.Close ();
        // Перемінної DocName присвоюємо адресний рядок
        DocName = OpenFileName;
      }
    }
```

Крок 3. Додаємо обробник пункту меню Open форми frmmain:

Перемикаємося в режим дизайну форми **frmmain**. Двічі клацаємо на пункт меню Open. У шаблон методу, який з'явився, записуємо такий код:

```
private void mnuOpen_Click(object sender, EventArgs e)
      {
        // Можна програмно задавати доступні для огляду розширення файлів
```

```
//openFileDialog1.Filter = "Text Files (* .txt) | * .txt | All Files (*. *) | *. *";
        // Якщо обрано діалог відкриття файла, виконуємо умова
        if (openFileDialog1.ShowDialog() == DialogResult.OK)
        {
            // Створюємо новий документ
            blank frm = new blank();
            // Викликаємо метод Open форми blank
            frm.Open(openFileDialog1.FileName);
            // Вказуємо, що батьківською формою є форма frmmain
            frm.MdiParent = this;
            // Надаємо змінної DocName ім'я файла
            frm.DocName = openFileDialog1.FileName;
            // Властивості Text форми присвоюємо змінну DocName
            frm.Text = frm.DocName;
            // Викликаємо форму frm
            frm.Show();
    }
```

Крок 4. Запускаємо програму і відкриваємо текстовий файл, збережений у блокноті в форматі Unicod (або ANSI) (рис. 36).

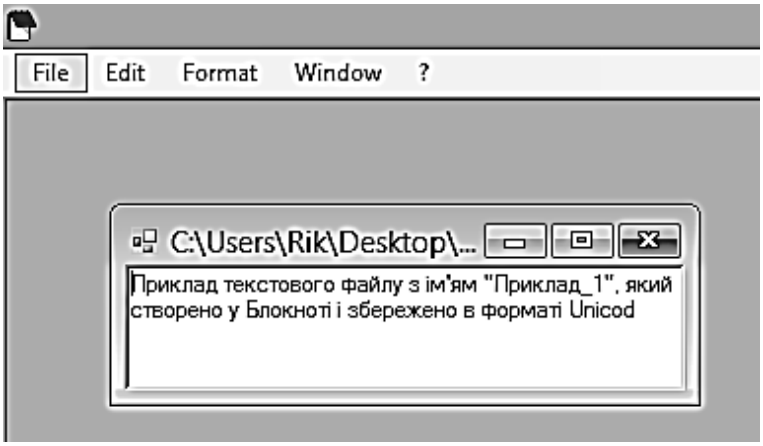


Рис. 36. **Назва вікна є адресою та ім'ям відкритого файла**

Для коректного читання кирилиці текст у блокноті повинен бути збережений у кодуванні Unicode або ANSI. На жаль, вбудовані діалогові вікна OpenFileDialog Visual Studio .NET не містять додаткового поля, що дозволяє вибирати кодування файла під час його відкриття або збереження, як це реалізовано, наприклад, у стандартному Блокноті.

SaveFileDialog

Крок 1. Налаштування властивостей вікна.

Для збереження файлів додаємо на форму **frmmain** елемент управління saveFileDialog1. Властивості цього діалогу в точності такі ж, як і у компонента OpenFileDialog (див. рис. 35).

Крок 2. Робота з файловими потоками в коді форми **blank**.

2.1. Перемикаємося в режим дизайну форми blank.

Переходимо в код форми blank:

```
// Створюємо метод Save, де як параметр оголошуємо рядок адреси файла.
        public void Save (string SaveFileName)
        {
          // Якщо файл не обраний, повертаємося назад (з'явиться вбудоване попередження)
          if (SaveFileName == "")
          {
            return;
          }
          else
          {
// Створюємо новий об'єкт StreamWriter і передаємо йому змінну // OpenFileName
            StreamWriter sw = new StreamWriter (SaveFileName);
            // Вміст richTextBox1 записуємо в файл
            sw.WriteLine (richTextBox1.Text);
            // Закриваємо потік
            sw.Close ();
// Встановлюємо в якості імені документа назву збереженого файла
            DocName = SaveFileName;
          }
        }
```

Крок 3. Додаємо обробник пункту меню Save форми frmmain:

Перемикаємося в режим дизайну форми **frmmain**. Двічі клацаємо на пункті меню Save. У шаблон методу, який з'явився, записуємо такий код:

```
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
    {
      // Можна програмно задавати доступні для огляду розширення файлів
```

```
        //openFileDialog1.Filter = "Text Files (* .txt) | * .txt | All Files (*. *) | *. *";
        // Якщо обрано діалог відкриття файлу, виконуємо умова
        if (saveFileDialog1.ShowDialog() == DialogResult.OK)
        {
           // Перемикаємо фокус на дану форму.
           blank frm = (blank)this.ActiveMdiChild;
           // Викликаємо метод Save форми blank
           frm.Save(saveFileDialog1.FileName);
           // Вказуємо, що батьківською формою є форма frmmain
           frm.MdiParent = this;
           // Надаємо змінної FileName ім'я файла
           frm.DocName = saveFileDialog1.FileName;
           // Властивості Text форми присвоюємо змінну DocName
           frm.Text = frm.DocName;
        }
    }
```

Крок 4. Запускаємо програму. Тепер файли можна відкривати, редагувати і зберігати.

FontDialog

Додаємо тепер можливість вибирати шрифт, його розмір і накреслення.

Крок 1. У режимі дизайну перетягнемо на форму **frmmain** з вікна ToolBox елемент управління FontDialog.

Крок 2. Переходимо в обробник пункту Font головного меню і заповнюємо його шаблон кодом, який наведено далі.

```
        private void mnuFont_Click (object sender, System.EventArgs e)
        {
          // Перемикаємо фокус на дану форму.
          blank frm = (blank) this.ActiveMdiChild;
          // Вказуємо, що батьківською формою є форма frmmain
          frm.MdiParent = this;
          // Викликаємо діалог
          fontDialog1.ShowColor = true;
          // Зв'язуємо властивості SelectionFont і SelectionColor елемента RichTextBox
          // з відповідними властивостями діалогу
          fontDialog1.Font = frm.richTextBox1.SelectionFont;
          fontDialog1.Color = frm.richTextBox1.SelectionColor;
```

```
    // Якщо обрано діалог відкриття файлу, виконуємо умова
    if (fontDialog1.ShowDialog () == DialogResult.OK)
    {
      frm.richTextBox1.SelectionFont = fontDialog1.Font;
      frm.richTextBox1.SelectionColor = fontDialog1.Color;
    }
    // Показуємо форму
    frm.Show ();
  }
```

Крок 3. Переходимо на форму **blank**. Встановлюємо властивість Modifiers згідно з рис. 37.

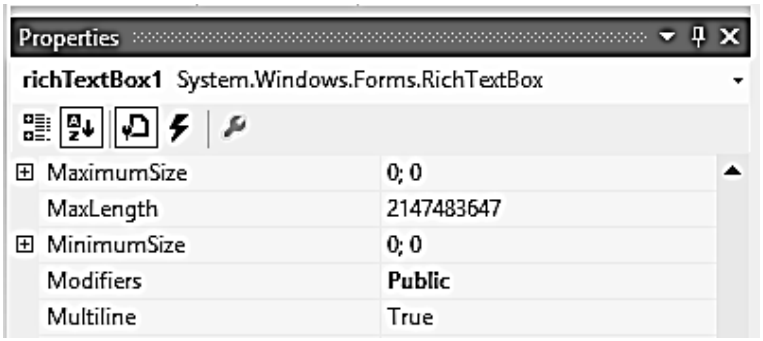


Рис. 37. **Налаштування властивості Modifiers**

Крок 4. Запускаємо програму. Тепер під час вибору пункту Font меню Format можна міняти параметри шрифту поточного тексту (рис. 38).

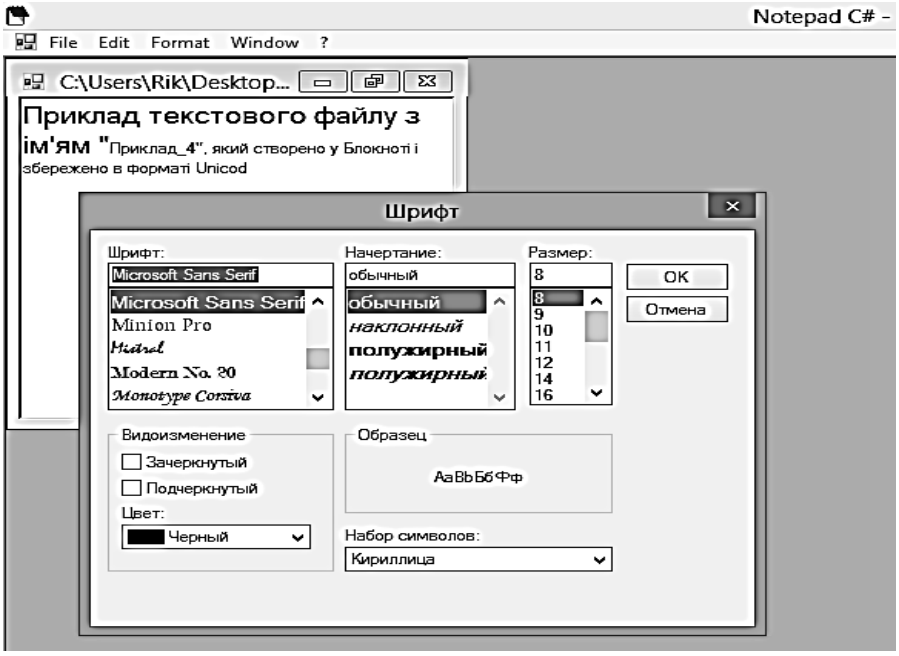


Рис. 38. **Вікно вибору шрифту (пункту Font меню Format)**

ColorDialog

Діалогове вікно FontDialog містить список кольорів, які можуть бути застосовані до тексту, але пропонований список обмежений. Більш цікавою видається можливість призначати користувальницький колір, який може бути визначений у великому діапазоні.

Крок 1. У режимі дизайну перетягнемо на форму **frmmain** з вікна ToolBox елемент управління ColorDialog

Крок 2. Переходимо в обробник пункту Color головного меню і заповнюємо його шаблон кодом, який наведено далі:

```
private void mnuColor_Click(object sender, EventArgs e)
      {
         blank frm = (blank)this.ActiveMdiChild;
         frm.MdiParent = this;
         colorDialog1.Color = frm.richTextBox1.SelectionColor;

         if (colorDialog1.ShowDialog() == DialogResult.OK)
         {
            frm.richTextBox1.SelectionColor = colorDialog1.Color;
         }

         frm.Show();
      }
```

Крок 3. Запускаємо програму. Тепер під час вибору пункту Color меню Format можна міняти колір шрифту поточного тексту (рис. 39) в значно більшому діапазоні порівняно з можливістю, яку дає пункт.

Зверніть увагу на те, що код для ColorDialog в точності такий же, як і частина коду для властивості Color діалогу FontDialog. Це й не дивно: адже ми пов'язуємо ці діалоги з властивостями одного і того ж об'єкта – RichTextBox.

Закривання форми

У обробнику пункту Clos головного меню форми **frmmain** додаємо код:

```
private void mnuExit_Click(object sender, EventArgs e)
      {
         this.Close();
      }
```

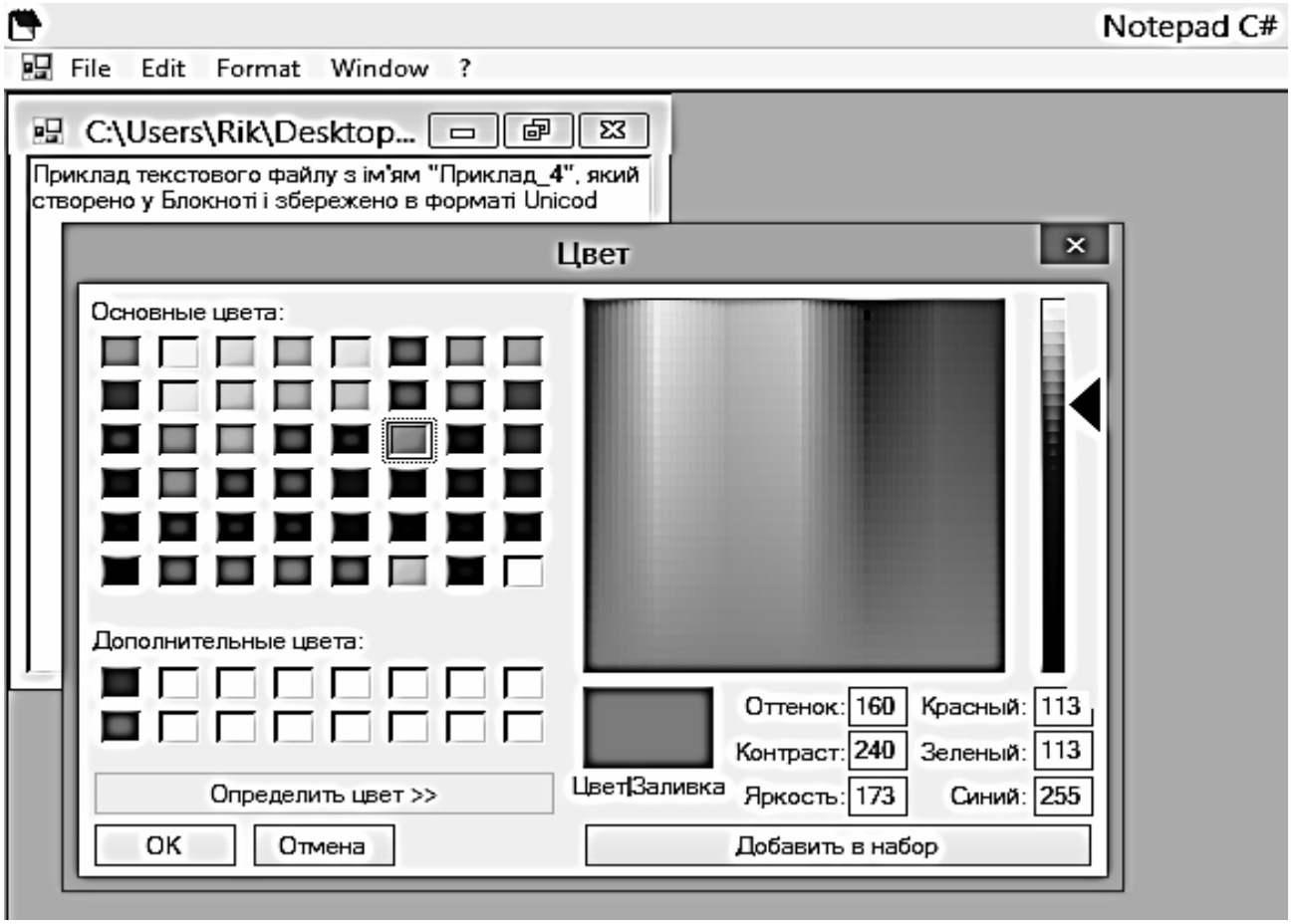


Рис. 39. **Вікно вибору кольору користувача**

Метод Close закриває форму і може бути призначений іншим елементам управління ─ наприклад, кнопці.

Порядок виконання лабораторної роботи

Загальна частина

1. Набрати, відкомпілювати і запустити на виконання першої програми (приклад 1), яка була наведена в розділі "Основні положення" даної лабораторної роботи.

Індивідуальна частина

Повторити всі кроки розроблення програми "Блокнот" (приклад 2). Слід до імен (властивість Name) відповідних пунктів головного меню додати префікс із перших трьох букв свого прізвища. Наприклад, було – mnuFile, стало – mnuFile_Fan, якщо прізвище студента є Фандорін.

Зміст звіту

1. Титульний аркуш.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти відповідних обробників подій індивідуального завдання, які пов'язані з пунктами головного і контекстного меню.

4. Висновки.

Контрольні запитання

1. Які групи елементів надає середа Visual Studio.NET, щоб забезпечить взаємодію між користувачем і програмою, яка розробляється? Опишіть групу командних об'єктів.

2. Дайте порівняння елементів TextBox, RichTextBox.

3. Опишіть технологію створення головного меню MDI-додатка.

4. Для чого і яким чином застосовується компонент OpenFileDialog?

5. Опишіть технологію створення контекстного меню MDI-додатка.

Лабораторна робота 14
Розроблення Windows-додатків з елементами графіки

Мета роботи – отримати практичні навички роботи щодо створення MDI- і SDI-додатків з елементами графіки.

Вона сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

взаємодії форм;

модальних та немодальних форм;

передачі інформації між формами;

запуску програм з програми;

уміння:

використовувати стандартні компоненти Designer Forms;

розробляти MDI- додатки з стандартним Windows інтерфейсом;

комунікації:

аргументована взаємодія з клієнтами та замовниками під час вибору технології розроблення елементів інтерфейсу Windows додатків;

робота в команді, яка проектує інтерфейс користувача певного Windows-додатка;

автономність і відповідальність:

самостійне формулювання рекомендацій щодо вибору технології створення інтерфейсу користувача;

здатність обґрунтувати доцільності застосування певного набору стандартних компонентів для створення відповідного інтерфейсу Windows-додатка.

Основні положення

Взаємодія форм

Звичайний Windows-додаток завжди містить кілька форм. Одні відкриваються у процесі роботи, інші закриваються. У кожний поточний момент на екрані може бути відкрита одна або декілька форм, користувач може працювати з однією формою або перемикатися по ходу роботи з однієї на іншу.

Слід чітко розрізняти процес створення форми – відповідного об'єкта, що належить класу Form або спадкоємцю цього класу, – і процес показу форми на екрані.

Для показу форми слугує метод Show цього класу, що викликається відповідним об'єктом; для приховування форми використовується метод Hide. Реально методи Show і Hide змінюють властивість Visible об'єкта, так що замість виклику цих методів можна міняти значення цієї властивості, встановлюючи його або в true, або в false.

Зауважте різницю між приховуванням і закриттям форми – між методами Hide і Close. Перший із них робить форму невидимою, але сам об'єкт залишається живим і неушкодженим. Метод Close відбирає у форми її ресурси, роблячи об'єкт відтепер недоступним; викликати метод Show після виклику методу Close неможливо, якщо тільки не створити об'єкт заново. Відкриття та показ форми завжди означає одне і те ж – виклик методу Show. У форми є метод Close, але немає методу Open. Форми, як і всі об'єкти, створюються при виклику конструктора форми під час виконання операції new.

Форма, що відкривається в процедурі Main під час виклику методу Run, називається головною формою проекту. Її закриття призводить до закриття всіх інших форм і завершення Windows-програми. Завершити додаток можна і програмно, викликавши в потрібний момент статичний метод Exit класу Application. Закриття інших форм не приводить до завер-

шення проекту. Найчастіше головна форма проекту завжди відкрита, в той час як інші форми відкриваються і закриваються (ховаються). Якщо ми хочемо, щоб в кожний поточний момент була відкрита тільки одна форма, то потрібно вжити певних заходів, щоб під час закриття (приховування) форми відкривалася інша. Інакше можлива клінчевая ситуація – всі форми закриті, зробити нічого не можна, а додаток не завершено. Звичайно, вихід завжди є – завжди можна натиснути магічну трійку клавіш CTRL + ALT + DEL і завершити будь-який додаток.

Можна створювати форми як об'єкти класу Form. Однак такі об'єкти досить рідкісні. Найчастіше створюється спеціальний клас FormX – спадкоємець класу Form. Так, зокрема, відбувається в Windows-додатку, створюваному за замовчуванням, коли створюється клас Form1 – спадкоємець класу Form. Так відбувається в режимі проектування, коли в проект додається нова форма з використанням пункту меню Add Windows Form. Як правило, кожна форма в проекті – це об'єкт власного класу. Можлива ситуація, коли знову створювана форма багато в чому повинна бути схожою на вже існуючу, і тоді клас нової форми може бути зроблений спадкоємцем класу форми існуючої. Спадкування форм ми розглянемо докладніше трохи пізніше.

Модальні та немодальні форми

Первинним є поняття модального і немодального вікна. Вікно називається модальним, якщо не можна закінчити роботу у відкритому вікні доти, поки воно не буде закрито. Модальне вікно не дозволяє, якщо воно відкрито, тимчасово переключитися на роботу з іншим вікном. Вийти з модального вікна можна, тільки закривши його. Немодальні вікна допускають паралельну роботу у вікнах. Форма називається модальною або немодальною залежно від того, яке її вікно.

Метод Show відкриває форму як немодальну, а метод ShowDialog – як модальну. Назва методу відображає основне призначення модальних форм – вони призначені для організації діалогу з користувачем, і поки діалог не завершиться, покидати форму забороняється.

Передача інформації між формами

Часто форми повинні працювати з одним і тим же об'єктом, роблячи з ним різні операції. Як це реалізується? Звичайна схема така: об'єкт створюється в одній з форм, найчастіше, у головній. Під час створення

наступної форми глобальний об'єкт передається конструктору нової форми в якості аргументу. Природно, одне з полів нової форми має представляти посилання на об'єкт відповідного класу, так що конструктору залишиться тільки зв'язати посилання з переданим йому об'єктом. Зауважте, все це ефективно реалізується, оскільки об'єкт створюється лише один раз, а різні форми містять посилання на цей єдиний об'єкт.

Якщо такий глобальний об'єкт створюється у головній формі, то можна передавати не об'єкт, необхідний іншим формам, а його контейнер – головну форму. Це зручніше, оскільки можна передати кілька об'єктів, можна не замислюватися над тим, який об'єкт передавати тій чи іншій формі. Мати посилання на головну форму часто необхідно, хоча б для того, щоб при закритті будь-якої форми можна було б відкривати головну, якщо вона була попередньо прихована.

Уявімо, що кілька форм повинні працювати з об'єктом класу Books. Нехай в головній формі такий об'єкт оголошений:

public Books myBooks;

У конструкторі головної форми такий об'єкт створюється:

myBooks = new Books (max_books);

де max_books – задана константа. Нехай ще в головній формі оголошена форма – об'єкт класу NewBook:

public NewBook form2;

Під час створення об'єкта form2 його конструктору передається посилання на головну форму:

form2 = new NewBook (this);

Клас NewBook містить поля:

private Form1 mainform;

private Books books;

а його конструктор такий код:

mainform = form;

books = mainform.myBooks;

Тепер об'єкту form2 доступні раніше створені об'єкти, що задають книги і головну форму, так що в обробнику події Closed, що виникає під час закриття форми, можна задати код:

```
private void NewBook_Closed (object sender, System.EventArgs e)
{
mainform.Show ();
}
```

що відкриває головну форму.

Продовжимо роботу над додатком "Блокнот" (див. приклад 2 попереднього лабораторного заняття).

Додавання пункту SaveAs… в меню File програми Блокнот.

Запускаємо програму "Блокнот". Тепер файли можна відкривати, редагувати і зберігати. Однак, під час збереження внесених змін у вже раніш збереженому файлі і знову відкритому, замість його перезапису знову з'являється вікно SaveFileDialog.

Змінимо програму так, щоб можна було зберігати і перезаписувати файл.

Крок 1. У конструкторі форми **frmmain** після InitializeComponent відключимо доступність пункту меню Save:

mnuSave.Enabled = false;

Крок 2. Перемикаємося в режим дизайну форми **frmmain** і додаємо пункт меню Save As після пункту Save.

Установлюємо наступні властивості цього пункту: Name – mnuSaveAs, Shortcut – CtrlShiftS, Text – Save &As.

Крок 3. У обробнику Save As вставляємо вирізаний обробник пункту Save і додаємо включення доступності Save:

mnuSave.Enabled = true;

Крок 4. Зберігати зміни потрібно як у щойно збережених документах, так і в документах, створених раніше і відкритих для редагування. Тому додаємо в метод Open включення доступності пункту меню Save:

```
private void mnuOpen_Click (object sender, System.EventArgs e)
{
  mnuSave.Enabled = true;
}
```

Крок 5. У обробнику пункту Save додаємо виклик методу Save форми blank:

```
private void mnuSave_Click (object sender, System.EventArgs e)
  {
      . . .
      // Викликаємо метод Save форми blank
      frm.Save (frm.DocName);
  }
```

Запускаємо програму. Тепер, якщо ми працюємо з незбереженим документом, пункт Save неактивний, після збереження він стає активним і,

крім того, працює сполучення клавіш Ctrl + S. Можна зберегти копію поточного документа, знову скориставшись пунктом меню Save As.

Збереження файлу при закритті форми

Кожного разу, коли ми закриваємо документ Microsoft Word, в який внесли зміни, повинно з'являється вікно попередження, що пропонує зберегти документ. Додаймо аналогічну функцію у додаток.

Крок 1. У класі **blank**: System.Windows.Forms.Form форми **blank** створюємо змінну, яка буде фіксувати факт внесення зміни в поточний документа:

```
public bool IsChanged = false;
```

Крок 2. У вікні властивостей перемикаємо на події форми, клацнувши на значок із блискавкою.

У поле події TextChanged двічі клацаємо і переходимо в шаблон обробника події, в який записуємо такий код, який фіксує зміну в поточному документі:

```
private void richTextBox1_TextChanged(object sender, EventArgs e)
      {
         IsChanged = true;
      }
```

Крок 3. В обробник методів Save і Save As форми **frmmain** додаємо значення цієї змінної, яка дорівнює – false.

Таким чином, якщо в поточному файлі були якісь зміни, а потім він був збережений, то під час повторного його відкриття ознака зміни IsChanged має бути в протилежному стані.

```
private void mnuSave_Click (object sender, System.EventArgs e)
    {
...
        frm. IsChanged = false;
    }

  private void mnuSaveAs_Click (object sender, System.EventArgs e)
    {
...
frm. IsChanged = false;
    }
```

Крок 4. Переходимо в режим дизайну форми **blank** і у вікні властивостей перемикаємо на події форми, клацнувши на значок з блискавкою.

У поле події FormClosing двічі клацаємо і переходимо в шаблон обробника події, в який записуємо такий код:

```
private void blank_FormClosing(object sender, FormClosingEventArgs e)
        {
          // Якщо змінна IsSaved має значення true, тобто новий документ
           // був збережений (Save As) або у відкритому документі були
           // збережені зміни (Save), то виконується умова
           if (IsSaved == true)
             // З'являється діалогове вікно, що пропонує зберегти документ.
              if (MessageBox.Show("Ви бажаєте зберегти зміни у фалі " 
+ this.DocName + "?",
                  "Message", MessageBoxButtons.YesNo,
                  MessageBoxIcon.Question) == DialogResult.Yes)
             // Якщо була натиснута кнопка Yes, викликаємо метод Save
             {
                this.Save(this.DocName);
             }
        }
```

Крок 5. Запускаємо програму. Створюємо новий документ (рис. 40), зберігаємо його, далі закриваємо форму. Ніякого попередження не виникає, тому що застосовувався пункт меню Save As.

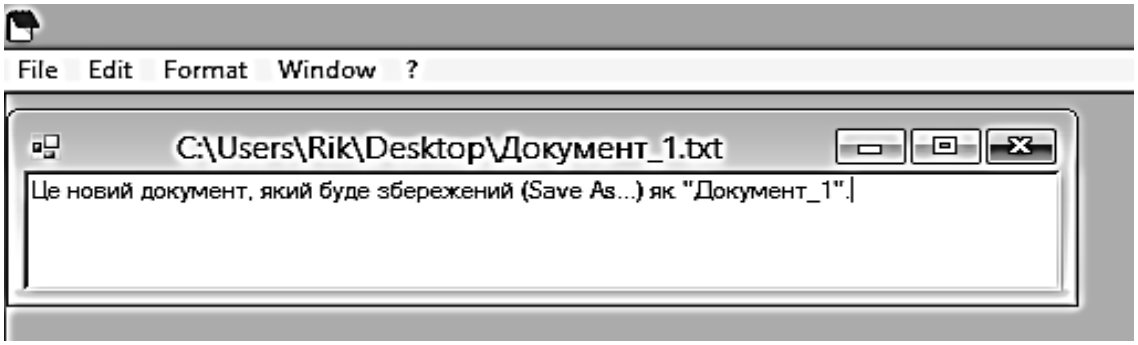


Рис. 40. **Новий документ**

Відкриваємо документ (пункт Open) і зразу ж закриваємо форму. Ніякого попередження не виникає, тому що в документі відсутні зміни.

Знову відкриваємо документ і дописуємо в нього текст "Виконувач – Студент Фандорін". Після чого натискаємо кнопку закрити форми. Як результат – з'являється вікно попередження (рис. 41).

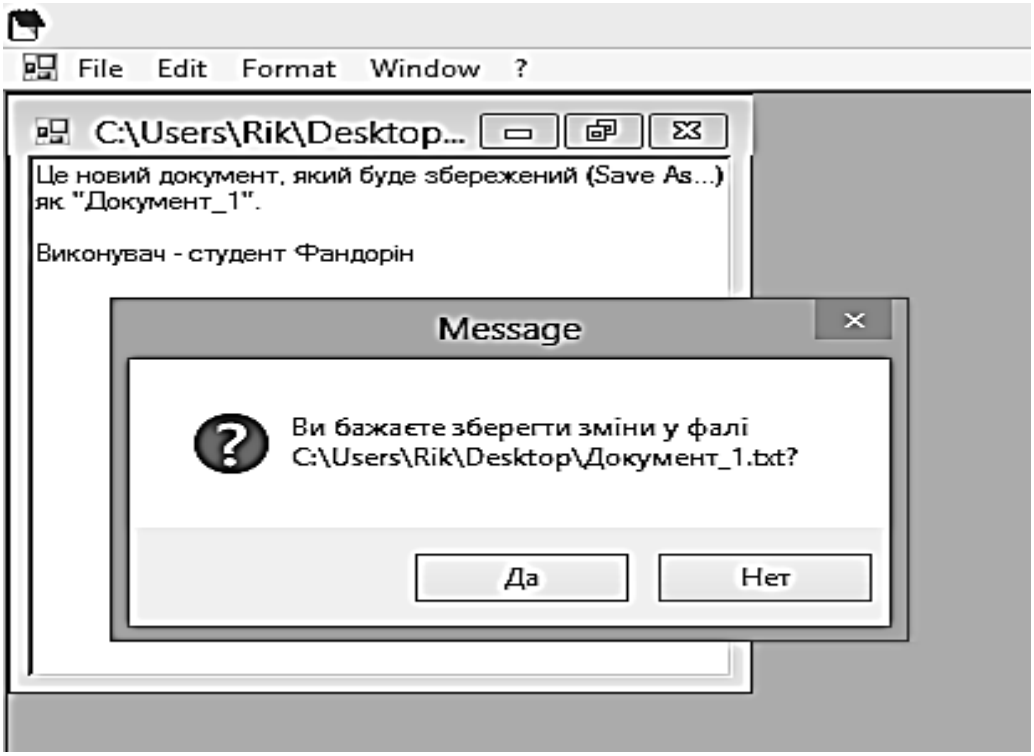


Рис. 41. **Вікно попередження**

Натискуємо кнопку "Да" і знову відкриваємо документ. Бачимо, що останні зміни в документі були збережені.

StatusBar

Елемент управління StatusBar застосовується в програмах для виведення інформації в рядок стану – невелику смужку, розташовану унизу програми.

Додамо до додатка Notepad C # рядок стану, на якій здійснюється підрахунок символів, що вводять і виводиться системний час.

Крок 1. Додаємо на форму blank елемент управління StatusBar (рис. 42).

Видаляємо вміст поля властивості Text.

Крок 2. У полі властивості Panels (рис. 43) клацаємо на кнопку (...).

Відкривається StatusBarCollectionEditor, в якому ми створюємо панелі для відображення.

Крок 3. Створіть дві панелі, двічі клацаючи на кнопці Add, і встановіть їм такі властивості (змінені значення виділяються жирним шрифтом) (рис. 44; 45):

Значення деяких властивостей елемент управління StatusBar наведено в табл. 3.

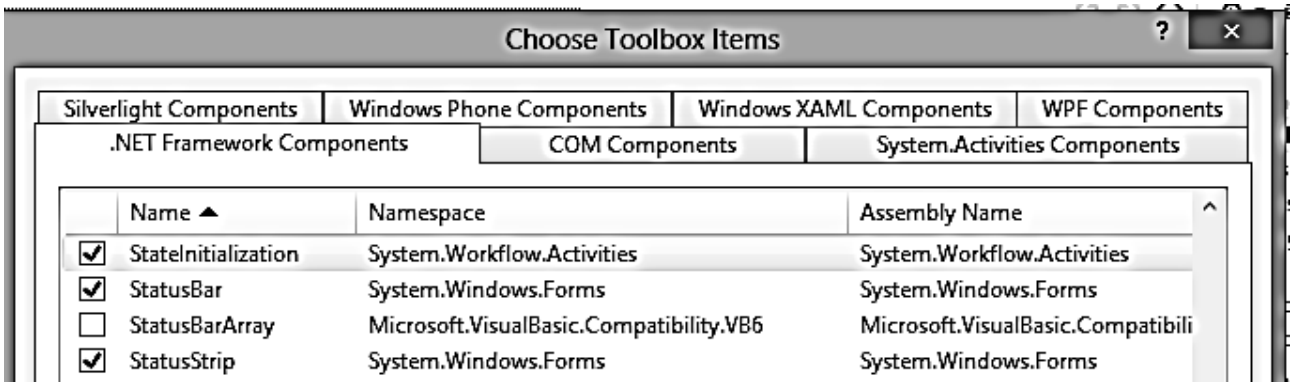


Рис. 42. **Додавання на форму елемента управління StatusBar**

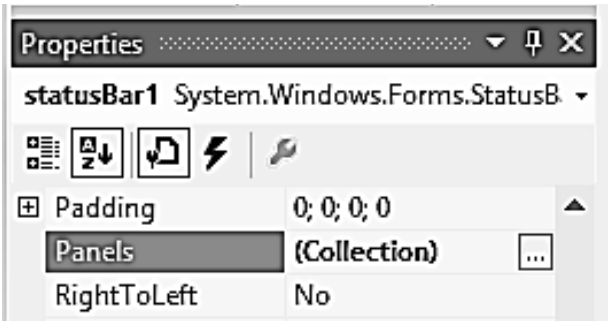


Рис. 43. **Поле властивості Panels**

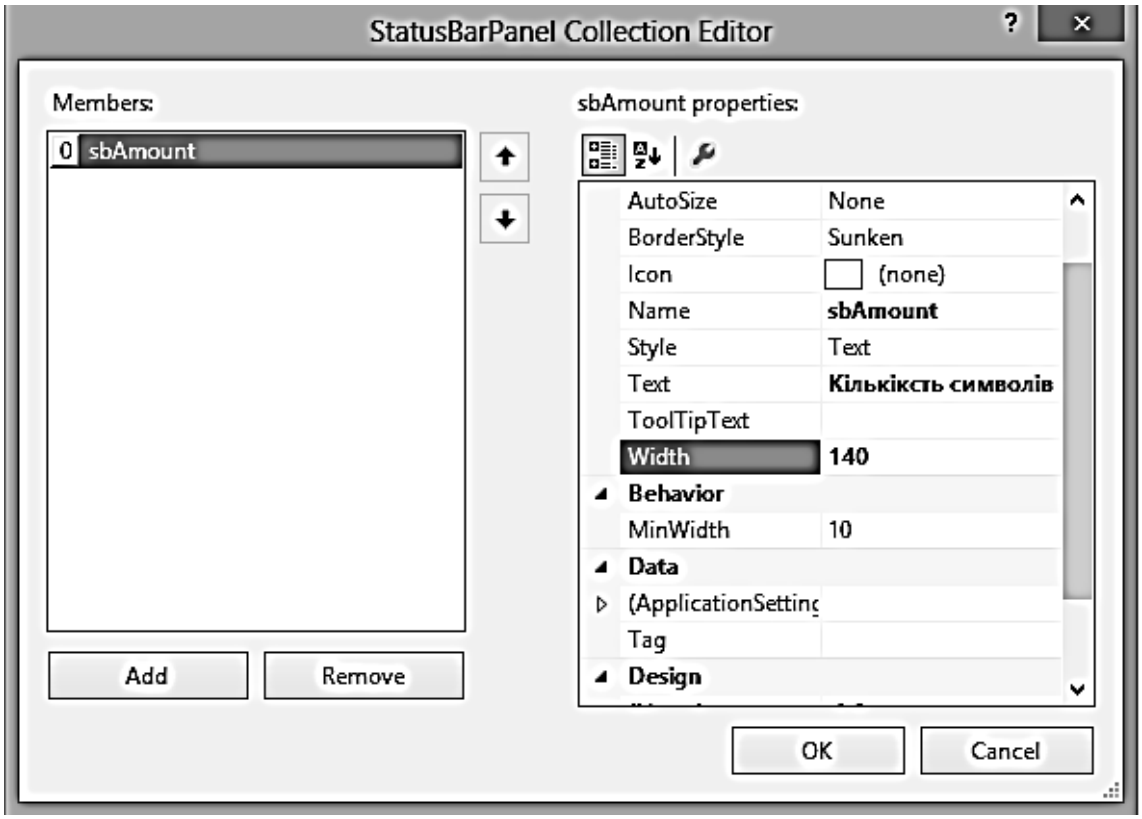


Рис. 44. **Властивості Name, Text, Width панелі sbAmount**

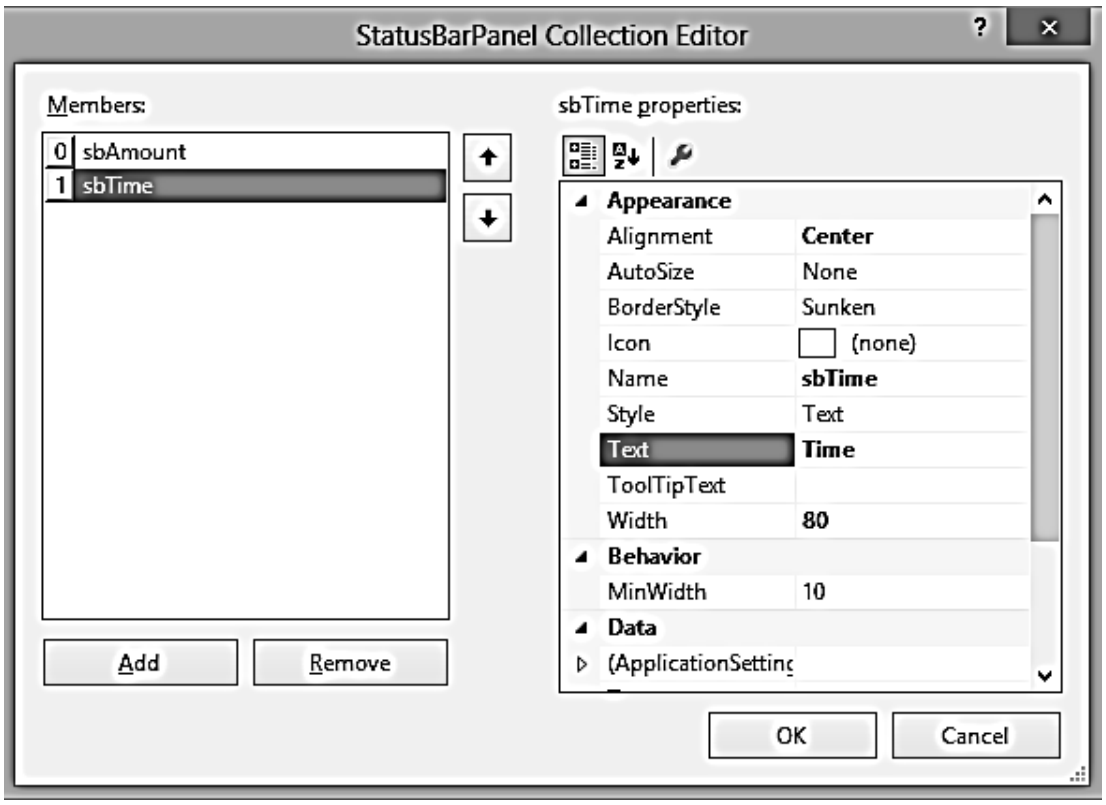


Рис. 45. **Властивості Alignment, Name, Text, Width панелі sbTime**

Таблиця 3

Властивості елемента управління StatusBar

Властивість	Значення
Alignment	Вирівнювання вмісту властивості Text на панелі
AutoSize	Зміна розмірів панелі за вмістом
BorderStyle	Зовнішній вигляд панелі - втоплена, піднесена або без виділення
Icon	Додавання іконки
Style	Стиль панелі
Text	Текст, наявний на панелі
ToolTipText	Спливаюча підказка – з'являється під час наведення курсора на панель
Width	Ширина панелі в пікселях
Name	Назва панелі для звернення до неї в коді

Властивості панелі, призначені у вікні редактора StatusBarCollection Editor, можна змінювати в коді (саме так буде зроблено далі у крокові 5).

Після завершення роботи над панелями закриваємо редактор.

Крок 4. Властивості ShowPanels елемента управління StatusBar встановлюємо значення True. На формі негайно відображаються дві панелі (рис. 46).

Крок 5. Виділяємо елемент управління RichTextBox форми **Blank**, у вікні його властивостей перемикаємо на подію TextChanged і доповнюємо раніш створений обробник новим кодом (виділено жирним шрифтом):

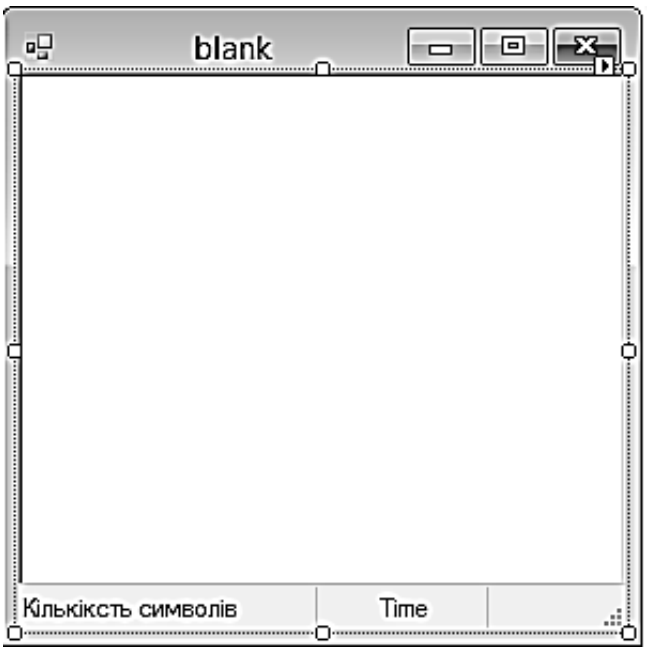


Рис. 46. **Панель StatusBar**

```
private void richTextBox1_TextChanged(object sender, EventArgs e)
  {
  IsChanged = true;
// Властивості Text панелі sbAmount програмно змінюємо
// на "Кількість символів: "
 sbAmount.Text = "Кількість символів: " +
richTextBox1.Text.Length.ToString();
}
```

Властивість Text панелі sbAmount ми змінюємо програмно: навіть якби ми нічого не написали у вікні редактора StatusBarCollectionEditor, при виникненні події TextChanged на панелі з'явиться відповідний напис.

Крок 6. Тепер друга панель – та, на яку будемо виводити системний час. У конструкторі форми **blank** додаємо код (виділено жирним шрифтом):

```
public blank()
        {
            InitializeComponent();
            // Властивості Text панелі sbTime встановлюємо системний
час,
            // Конвертіровав його в тип String
            sbTime.Text                                                =
Convert.ToString(System.DateTime.Now.ToLongTimeString());
            // В тексті підказки виводимо поточну дату
            sbTime.ToolTipText                                         =
Convert.ToString(System.DateTime.Today.ToLongDateString());
        }
```

Крок 7. Запускаємо програму. Результат подано на рис. 47.

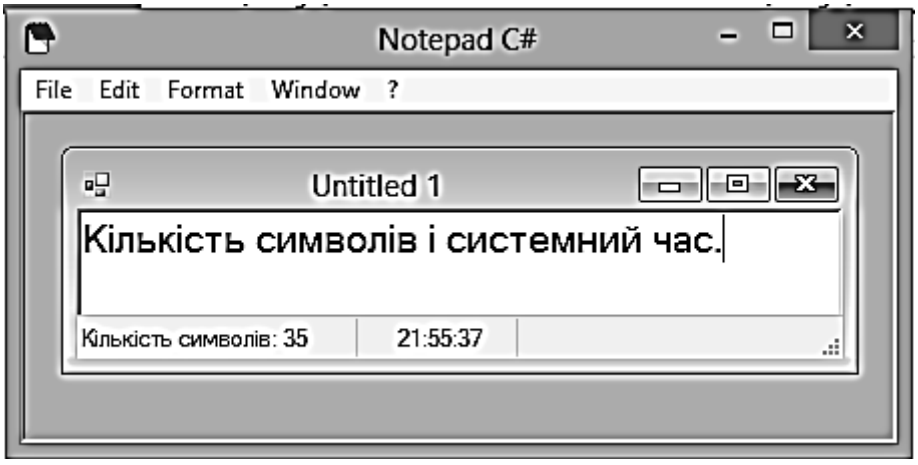


Рис. 47. **Панель StatusBar після введення тексту**

Компоненти Label, LinkLabel і PictureBox.

Програми, як правило, містять пункт головного меню "Про програму", де в окремому вікні поміщається логотип компанії, ліцензійна угода, гіпер-посилання на сайт розробника та інша інформація. Створимо подібну форму, використовуючи елементи управління – Label, LinkLabel і PictureBox.

Крок 1. Додаємо у даний проект нову форму і назвемо її About.cs. Установимо такі властивості форми:

Name → About

FormBorderStyle → FixedSingle

MaximizeBox → False

MinimizeBox → False

Size → 318; 214

Text → About Notepad C#

Крок 2. Додаємо на форму About елемент управління PictureBox – він є підкладкою, що розміщується на формі, яка може містити малюнки для відображення.

У полі властивості Image клацаємо на кнопку (...) і вибираємо малюнок (з файла роздаткового матеріалу, або з Інтернету)….\ Icon \ logo.gif.

Оскільки logo.gif є анімований малюнком, елемент PictureBox починає відтворювати анімацію відразу ж, навіть в режимі дизайну (рис. 48).

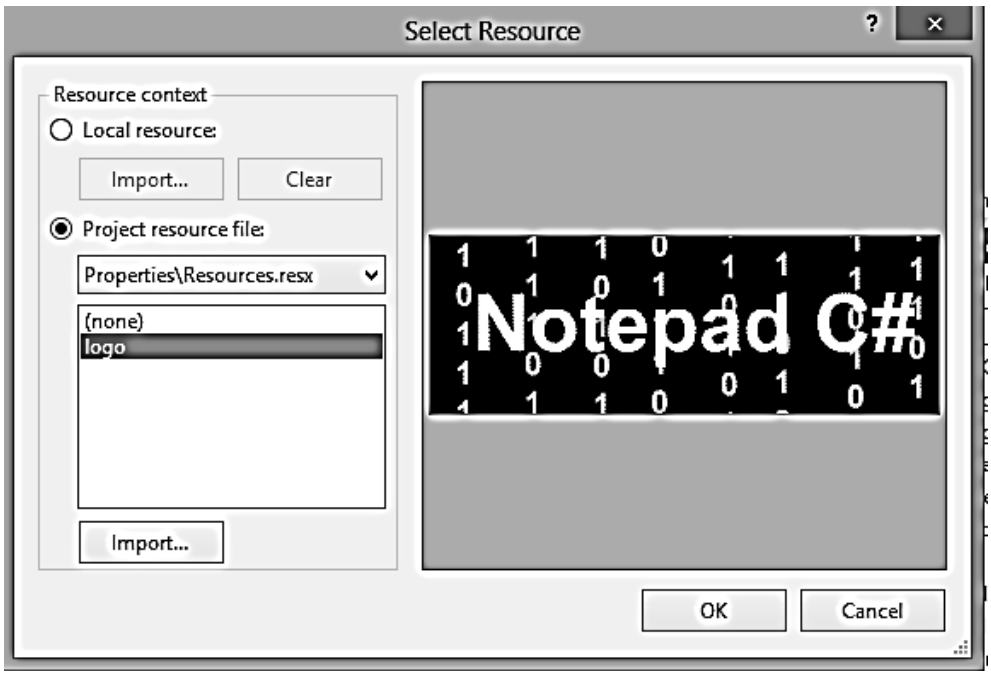


Рис. 48. **Вікно імпорту файла logo.gif з анімований малюнком**

Крок 3. З вікна ToolBox перетягнемо на форму елементи: Button, Label і LinkLabel.

У полі властивості Text кнопки введемо &OK.

Елемент Label призначений для розміщення на формі написів, які в готовому додатку будуть доступні тільки для читання. У полі властивості Text введемо "Notepad C# 2015. Розробник – Фандорін В. В.".

Крок 4. Елемент LinkLabel відображає текст на формі в стилі web-посилань і зазвичай використовується для створення навігації між формами або посилання на сайт. У полі Text цього елемента вводимо адресу гіпотетичного сайта – http://www.mo.hneu.edu.ua. Користувач буде переходити на сайт, натискаючи на це посилання, тому реалізуємо перехід за гіперпосиланням для події Click.

Крок 5. У вікні Properties елементу LinkLabel переходимо на вкладку з обробниками подій і клацаємо двічі на подію Click (рис. 49).

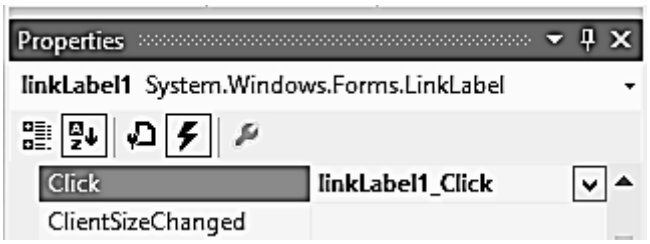


Рис. 49. **Виклик шаблону обробника події Click**

У шаблон, який з'явився, додаємо обробник:

```
private void linkLabel1_Click(object sender, EventArgs e)
        {
            // Додаємо блок для обробки виключень — з різних причин
            // користувач може не отримати доступу до ресурсу.
            try
            {
                // Викликаємо метод VisitLink, визначений нижче
                VisitLink();
            }
            catch (Exception ex)
            {
                MessageBox.Show(ex + "Неможливо відкрити сайт");
            }
        }

        // Створюємо метод VisitLink
        private void VisitLink()
        {
            // Змінюємо колір відвіданого посилання, програмно
            // звертаючись до властивості LinkVisited елемента LinkLabel
            linkLabel1.LinkVisited = true;
            // Викликаємо метод Process.Start method для запуску брау-
зера,
            // встановленого за замовчуванням, і відкриття посилання

System.Diagnostics.Process.Start("http://www.mo.hneu.edu.ua");
        }
```

Крок 6. Підключаємо обробник події кнопки OK (форми About), який буде забезпечувати закривання форми:

```
private void button1_Click(object sender, EventArgs e)
    {
       this.Close();
    }
```

Крок 7. У пункті головного меню About Programm .форми **frmmain** додаємо процедуру виклику форми About:

Для відкриття форми в модальному режимі використовується метод ShowDialog:

```
private void mnuAbout_Click(object sender, EventArgs e)
    {
       // Створюємо новий екземпляр форми About
       About frm = new About();
       frm.ShowDialog();
    }
```

Крок 8. Запускаємо програму (рис. 50).

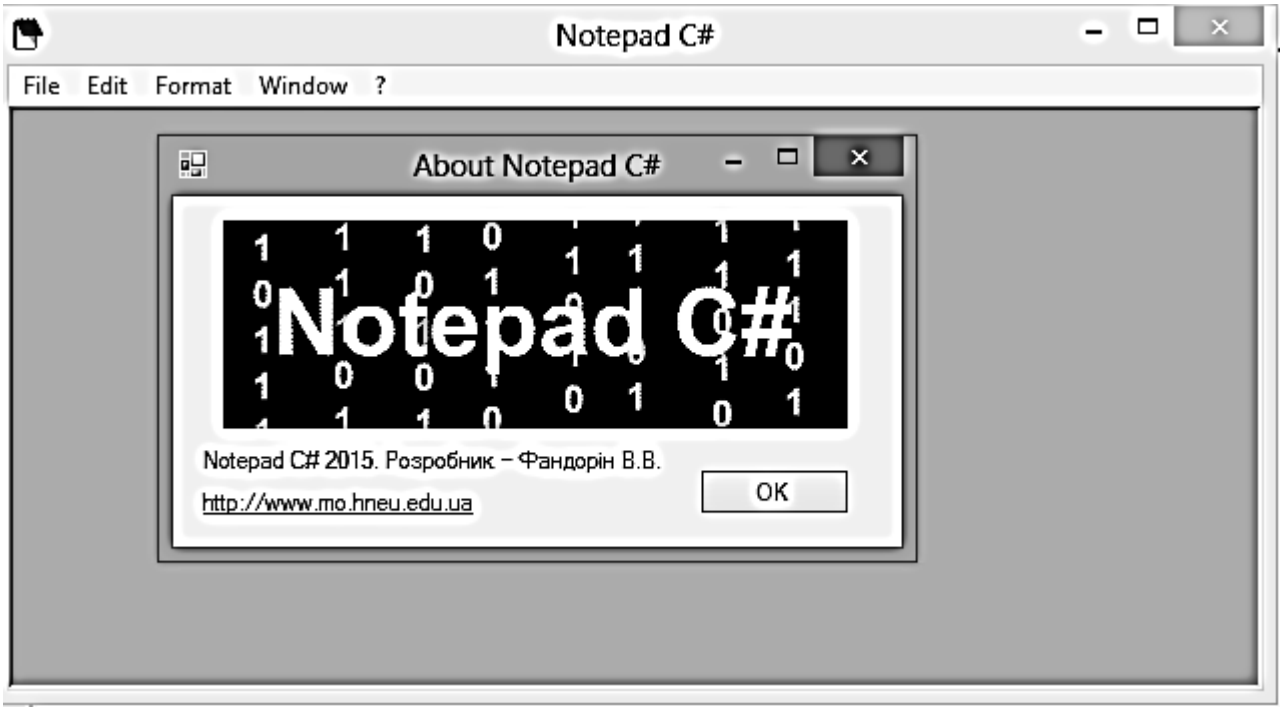


Рис. 50. **Вікно About Notepad C#**

Панель інструментів (компоненти ToolBar і ImageList)

Панелі інструментів ToolBar містять набори кнопок, як правило, дублюючих пункти головного меню. У графічних програмах панелі інструментів – це основний засіб робот. Далі наведена технологія створення подібних кнопок на прикладі поточного додатка.

Крок 1. Відкриємо додаток Notepad C# і перетягнемо з вікна ToolBox елемент управління ToolBar. У разі необхідності потрібно доповнити вікно ToolBox цим елементом.

На кнопках панелі зазвичай розташовуються іконки, тому, перш ніж ми почнемо займатися ними, слід заздалегідь подумати про малюнки на цих кнопках.

Крок 2. Додамо на форму **frmmain** елемент управління ImageList, який застосовується для зберігання малюнків, що можуть бути використані для оформлення елементів управління додатка.

Клацнемо в поле властивості Images елемента ImageList (рис. 51).

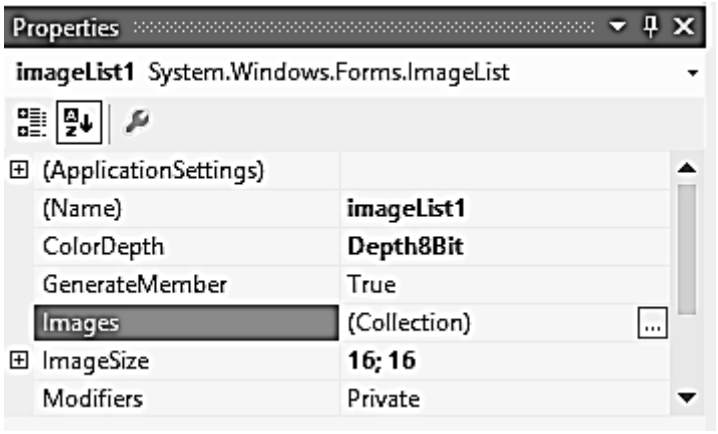


Рис. 51. **Властивість Images елемента ImageList**

Розкриємо вікно редактора колекцій (кнопка (Collection) ...).

Додамо файли іконок (рис. 52), послідовно натискаючи кнопку Add і вибираючи директорію - … \ Icon (дивись роздатковий файл).

Можна вибрати відповідні зображення на своєму комп'ютері або скачати з Інтернету. У будь-якому випадку, підібравши іконки, завершуємо роботу з редактором ImageCollectionEditor, натискаючи OK.

Крок 3. Створимо кнопки панелі інструментів, що дублюють дії пунктів меню New, Open, Save, ?.

Виділяємо елемент ToolBar (форма **frmmain**).

Властивості Name встановимо значення toolBarMain, а в поле властивості ImageList виберемо imageList1.

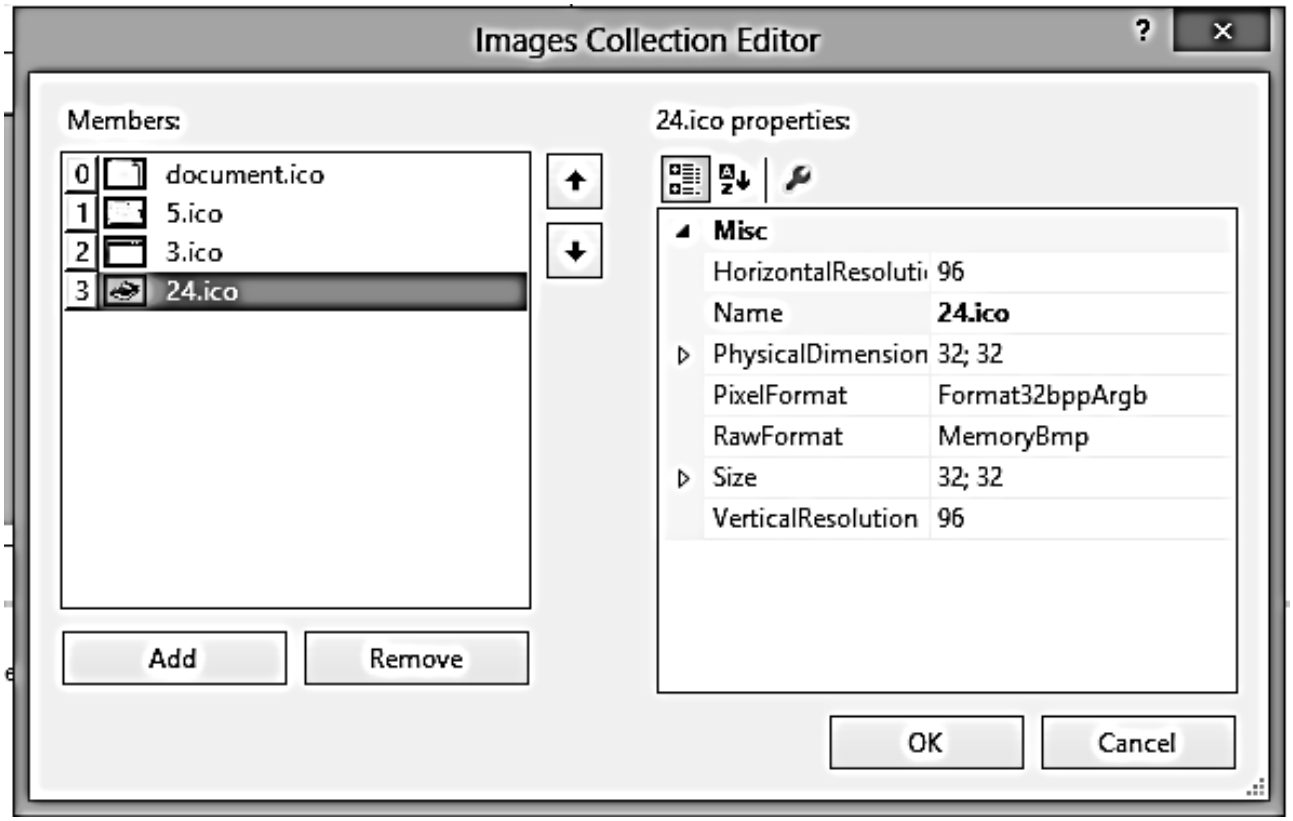


Рис. 52. **Вікно редактора колекцій**

Крок 4. Запустимо редактор ToolBarButton Collection Editor для створення кнопок. Для цього потрібно натиснути кнопку (...) в полі властивості Buttons (рис. 53).

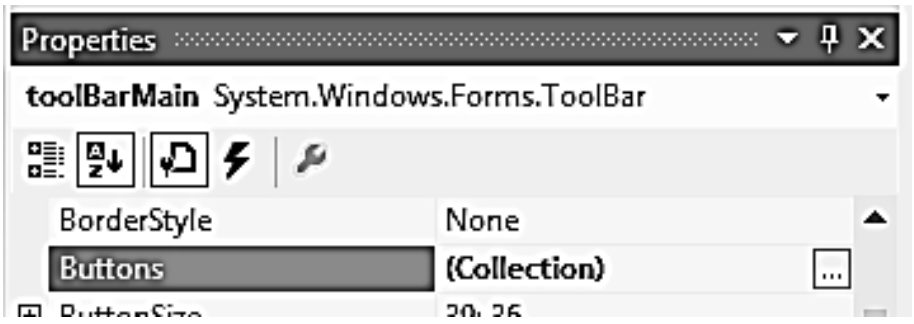


Рис. 53. **Запуск редактора ToolBarButton Collection Editor**

Як результат – з'являється вікно відповідного редактора. Для додавання кнопок у цьому редакторі теж слід натискати кнопку Add. Створимо чотири кнопки, встановивши для них такі властивості (табл. 4).

Таблиця 4

Властивості кнопок панелі інструментів ToolBarButton

Name	Image Index	ToolTipText
tbNew	0	Create New
tbOpen	1	Open
tbSave	2	Save
tbAbout	3	About

Властивість Name встановлює назву кнопки для звернення до неї в коді. Властивість Image Index визначає зображення на кнопці, а в поле ToolTipText вводимо текст підказки, яка буде з'являтися при наведенні курсору на кнопку.

Результат наведено на рис. 54.

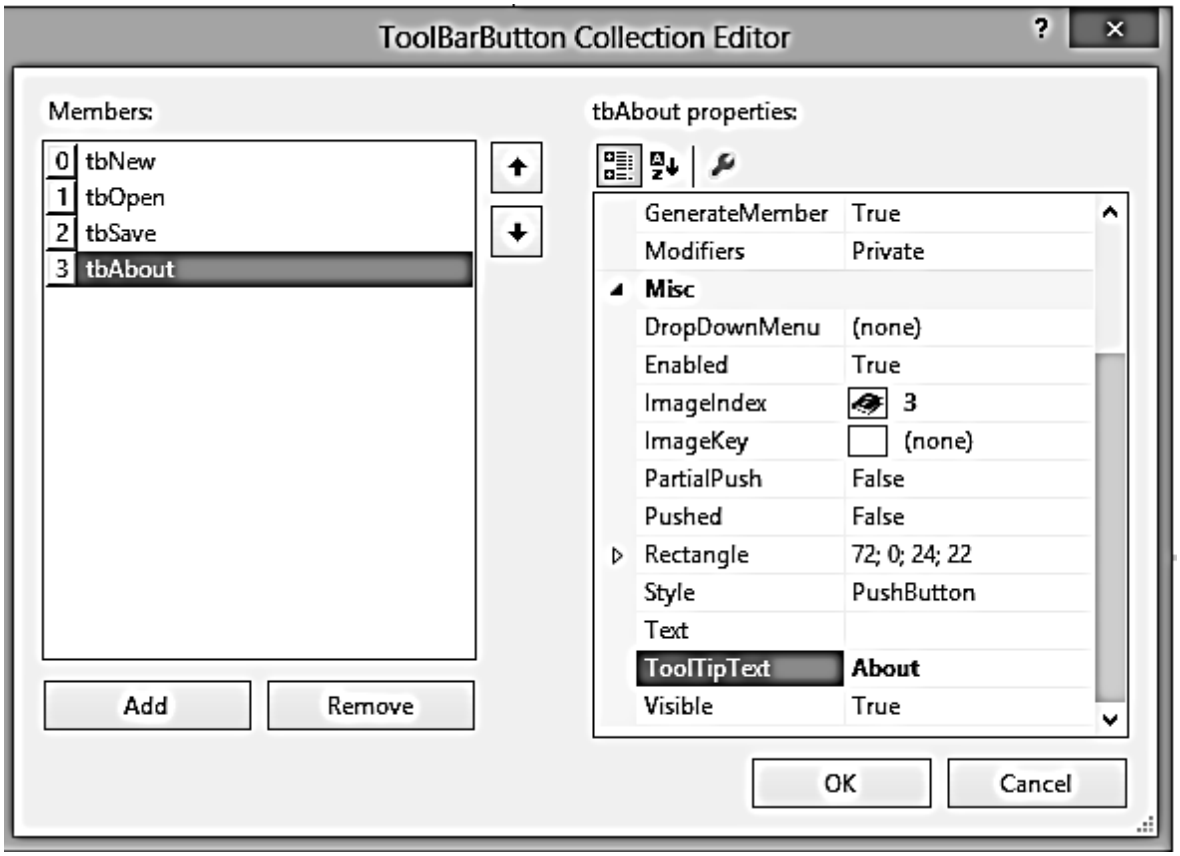


Рис. 54. **Редактор ToolBarButton Collection Editor**

Крок 5. Створення функціональності кнопок.

Завершивши роботу з редактором кнопок у режимі дизайну форми frmmain, двічі клацаємо на ToolBar і переходимо в шаблон коду.

```
private void toolBarMain_ButtonClick(object sender,
                                     ToolBarButtonClickEventArgs e)
    {

    }
```

Для створення функціональності кнопок пов'язуємо подію Click заданої кнопки з відповідним оброблювачем пунктів меню.

```
    private void toolBarMain_ButtonClick(object sender,
                                         ToolBarButtonClickEventArgs e)
        {
            // Create New
            if (e.Button.Equals(tbNew))
            {
                mnuNew_Click(this, new EventArgs());
            }
            // Open
            if (e.Button.Equals(tbOpen))
            {
                mnuOpen_Click(this, new EventArgs());
            }
            // Save
            if (e.Button.Equals(tbSave))
            {
                mnuSave_Click(this, new EventArgs());
            }
            // About
            if (e.Button.Equals(tbAbout))
            {
                mnuAbout_Click(this, new EventArgs());
            }
        }
```

Запускаємо програму. Кнопки панелі інструментів дублюють пункти меню, а під час наведення на них з'являються підказки (рис. 55).

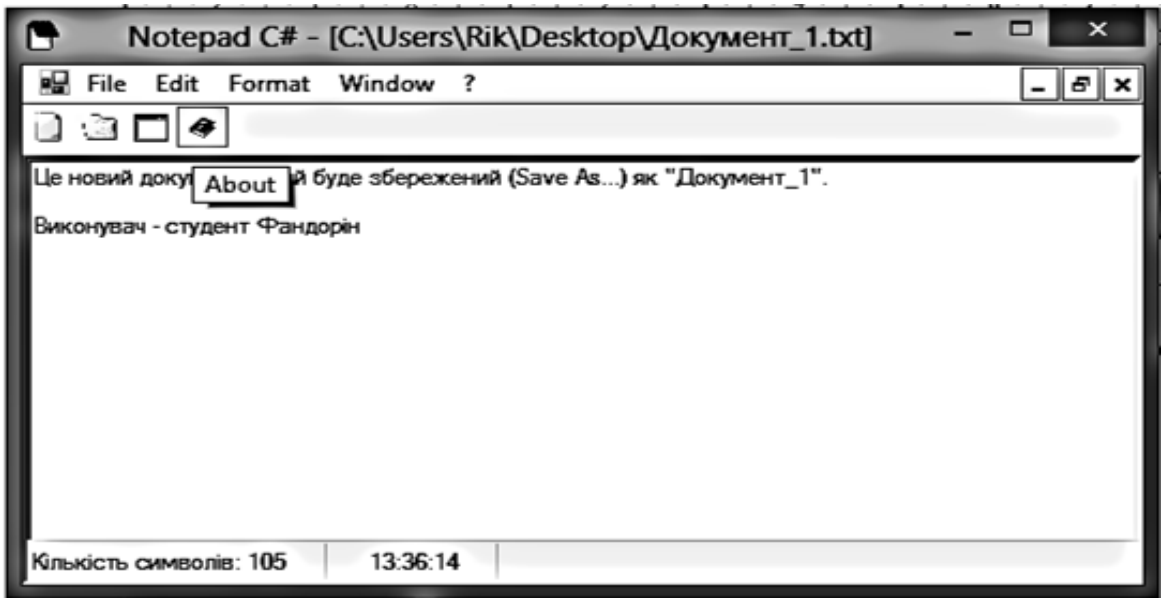


Рис. 55. **Заключний результат роботи програми**

Порядок виконання лабораторної роботи

Загальна частина

1. Повторити всі кроки продовження розроблення програми "Блокнот", які були наведені в розділі "Основні положення" даної лабораторної роботи.

Індивідуальна частина

Додати в головне меню програми пункт "Про автора" з двома – трьома підпунктами (назву вибрати самостійно), в яких розповісти про себе і свої захоплення. Текст проілюструвати фотографіями і посиланнями на відповідні Інтернет-сайти.

Зміст звіту

1. Титульний аркуш.
2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.
3. Тексти відповідних обробників подій індивідуального завдання, які пов'язані з пунктом "Про автора" головного меню.
4. Висновки.

Контрольні запитання

1. Як здійснюється взаємодія форм?
2. Дайте порівняння модальних і немодальних форм.
3. Як здійснюється передача інформації між формами?
4. Для чого і яким чином застосовується компонент SaveFileDialog?
5. Опишіть технологію створення елементів управління StatusBar.

Лабораторна робота 15
Розроблення мультимедійних Windows-додатків (звук, відео, анімація)

Мета роботи – отримати практичні навички роботи щодо створення Windows-додатків з елементами звуку, відео та анімації.

Вона сприяє напрацюванню таких **компетентностей** відповідно до Національної рамки кваліфікації:

знання:

особливостей розроблення інтерфейсів для управління звуком та відео;

методів запуску зовнішніх файлів відносно програми, що розробляється;

уміння:

розробляти графічні додатки зі звуком, відео та анімацією;

використовувати стандартні компоненти Designer Forms;

комунікації:

аргументована взаємодія з клієнтами та замовниками під час вибору технології розроблення елементів мультимедійного інтерфейсу Windows-додатків;

робота в команді, яка проектує мультимедійний інтерфейс користувача певного Windows-додатка;

автономність і відповідальність:

самостійне формулювання рекомендацій щодо вибору технології створення мультимедійного інтерфейсу користувача;

здатність обґрунтувати доцільності застосування певного набору стандартних компонентів для створення відповідного мультимедійного інтерфейсу Windows-додатка.

Порядок виконання лабораторної роботи

Загальна частина

Повторити лекційний матеріал теми 12. "Програмування додатків з елементами мультимедіа", уділяючи основну увагу прикладам розроблення графічних додатків зі звуком, відео та анімацією.

Індивідуальна частина

У головному меню програми "Блокнот", яка була створена в процесі виконання лабораторної роботи 14 "Розроблення Windows-додатків

з елементами графіки", модифікувати пункт "Про автора", в якому розповідаєтеся про себе і свої захоплення, таким чином:

додати елементи управління фоновою музикою;

передбачити можливість підключення відео з сюжетом про автора;

передбачити можливість підключення розважального анімаційного ролика.

Зміст звіту

1. Титульний аркуш.

2. Цілі лабораторного заняття і вказівка, які навички та вміння передбачається отримати в результаті його виконання.

3. Тексти відповідних обробників подій індивідуального завдання, які пов'язані з модифікованим пунктом "Про автора" головного меню.

4. Висновки.

Контрольні запитання

1. Дайте перелік особливості розроблення мультимедійного контенту в Windows Forms графічних додатках.

2. Опішить алгоритм додавання звука в графічний додаток.

3. Наведіть синтаксис виклику відеофайла в графічний додаток.

4. Як додати анімаційній ефекти в графічний додаток?

Рекомендована література

1. Гаврилов В. П. Основи програмування : конспект лекцій для студентів напряму підготовки 0927 "Видавничо-поліграфічна справа" усіх форм навчання / В. П. Гаврилов, В. В. Браткевич, І. О. Бондар. – Харків : Вид. ХНЕУ, 2007. – 172 с.

2. Ватсон К. Программист – программисту. C# / К. Ватсон ; пер. с англ. – Москва : Изд. "Лори", 2010. – 862 с.

3. Лабор В. В. Си Шарп. Создание приложений для Windows / В. В. Лабор. – Минск : Харвест, 2011. – 384 с.

4. Петцольд Ч. Программирование для Microsoft Windows на C#. в 2-х томах / Ч. Петцольд ; пер. с англ. – Москва : Издательско-торговый дом "Русская Редакция", 2009. – 576 с.

5. Просиз Д. Программирование для Microsoft NET / Д. Просиз. – пер. с англ. – Москва : Изд. "Русская Редакция", 2011. – 704 с.

6. Робинсон У. C# без лишних слов / У. Робинсон ; пер. с англ. – Москва: Пресс, 2010. – 352 с.

7. C# для профессионалов. В 2 томах / С. Робинсон, О. Корнес, Дж. Глинн и др. ; пер. с англ. – Москва : "Лори", 2012. – 734 с.

Зміст

НАВЧАЛЬНЕ ВИДАННЯ

Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни "ПРОГРАМУВАННЯ ЗАСОБІВ МУЛЬТИМЕДІА",

розділ "Мультимедійні об'єктно-орієнтовані додатки" для студентів напряму підготовки 6.051501 "Видавничо-поліграфічна справа" денної форми навчання

Укладач **Браткевич** Вячеслав Вячеславович

Відповідальний за видання *О. І. Пушкар*

Редактор *В. О. Бутенко*

Коректор *Т. А. Маркова*

План 2016 р. Поз. № 102.

Підп. до друку 28.11.2016 р. Формат 60 × 90 1/16. Папір офсетний. Друк цифровий.

Ум. друк. арк. 8,25. Обл.-вид. арк. 10,31. Тираж 40 пр. Зам. № 254.

Видавець і виготовлювач – ХНЕУ ім. С. Кузнеця, 61166, м. Харків, просп. Науки, 9-А

Свідоцтво про внесення суб'єкта видавничої справи до Державного реєстру ***ДК № 4853 від 20.02.2015 р.***