Державний вищий навчальний заклад
Українська академія банківської справи
Національного банку України

Кафедра економічної кібернетики

ЗАТВЕРДЖУЮ
Декан факультету
банківських технологій
канд. екон. наук, доцент
_____
Т.І. Гончарук
07.05.2008

КОНСПЕКТ ЛЕКЦІЙ

з дисципліни «Економічна кібернетика. Програмування»
освітньо-професійної програми підготовки за напрямом
6.050100 «Економічна кібернетика»
галузі знань 0501 «Економіка і підприємництво»
для денної форми навчання

Укладач – канд. екон. наук _____ В. В. Колдовський
07.05.2008

Завідувач кафедри –
канд.техн.наук, доцент. _____ С.М. Новак
07.05.2008

Розглянуто та схвалено на засіданні кафедри, протокол від 07.05.2008 № 8

Суми – 2008

ЗМІСТ

ВСТУП

Дисципліна «Економічна кібернетика. Основи програмування» є складовою варіативної компоненти освітньо-професійної програми підготовки (ОПП) за напрямом 6.050102 – «Економічна кібернетика» галузі знань 0501 – «Економіка і підприємництво».

Основна мета вивчення дисципліни: формування системи теоретичних знань і практичних навичок з розробки комп'ютерних програм з використанням сучасних мов і технологій програмування.

Завдання дисципліни: вивчення теоретичних і практичних підходів до постановки і аналізу задач з програмування, розробки алгоритмів і комп'ютерних програм, набуття вмінь застосовувати знання і навички в діяльності, пов'язаній з інформатизацією економіки.

Предметом дисципліни: задачі, які потребують свого вирішення шляхом створення комп'ютерних програм.

Зміст дисципліни розкривається у наступних темах:

- об'єктно-орієнтоване програмування;
- графічний інтерфейс користувача;
- інтерфейс взаємодії з реляційними базами даних;
- графіка, діаграми та звіти;
- Web-рішення;
- мережі та сервіс-орієнтовані рішення;
- узагальнення та мова LINQ;
- модульне тестування, відладка, профілювання та основи створення закінченого програмного продукту.

ЛК.01 – ВВЕДЕННЯ ДО ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Перелік питань:

1. Передумови та історія виникнення об’єктно-орієнтованого підходу до створення програних систем.
2. Основні положення об’єктної моделі.
3. Класи та об’єкти.
4. Відмінність між класом та об’єктом.
5. Складові класу.
6. Поля класу.
7. Методи класу.
8. Позначення класу.
9. Задання видимості членів класу.
10. Інкапсуляція, наслідування та поліморфізм.
11. Реалізація класу та створення його екземпляру за допомогою об’єктно-орієнтованої мови програмування.

На самостійне вивчення:

1. Правила приведення типів в C# [1, С.240-242]

1. Передумови та історія виникнення об’єктно-орієнтованого підходу до створення програних систем.

Головний імператив створення програмного забезпечення за С. МакКонеллом – це боротьба зі складністю. Адже можемо мати не зримість задачі, відсутність ефективних способів відображення.

Складність ПЗ є характерною особливістю даного продукту цивілізації і проявляється у значно більш високій мірі, ніж у будь-якій іншій галузі людської діяльності. Ф. Брукс підкреслює, що складність є сутністю

програмних об’єктів і нелінійно прискорюючими темпами зростає разом зі зростанням їх обсягів.

На відміну від продуктів інших галузей людської діяльності, програмні об’єкти не складаються із повторювальних елементів, оскільки такі елементи в процесі розробки мають вилучатися і оформлюватися у вигляді допоміжних модулів. Складність ПЗ визначається великою кількістю можливих його станів, великим обсягом інформації, який містять вихідні коди ПЗ, що в результаті призводить до не лише до технічних, а й до адміністративних проблем.

Г. Буч виділив 4 головні причини складності:

- складність предметної області розробки ПЗ. Потрібно визначити предмет, основу та мету програмного забезпечення для попередньої його розробки;
- складність управління процесом розробки ПЗ. Досить важко управляти великими програмними кодами з високим рівнем диференціації його частин. У процесі розробки можуть виникати помилки, які потрібно швидко знаходити та корегувати. При чому чим більший код, тим важче ним керувати;
- необхідність забезпечення достатньої гнучкості ПЗ. Програмне забезпечення повинне працювати по призначенню і необхідно передбачити можливість його зміни для різних умов застосування;
- незадовільні способи опису великих дискретних систем. Дуже вжко знайти необхідний і зручний спосіб опису таких систем.

Розробник програмного забезпечення повинен орієнтуватися на користувача, тобто створювати відповідно прості за використанням та наглядні за інтерфейсом програми, що не потребують спеціальних навичок і знань у програмуванні. Сьогодні програми розробляються для людей зі знанням сучасних технологій, але в мірі необхідних їм знань. Користувач програми прагне виконати на ній певні дії, а не зрозуміти її сутність

створення. Тому необхідно створювати просте і зрозуміле програмне забезпечення (рис. 1.1):

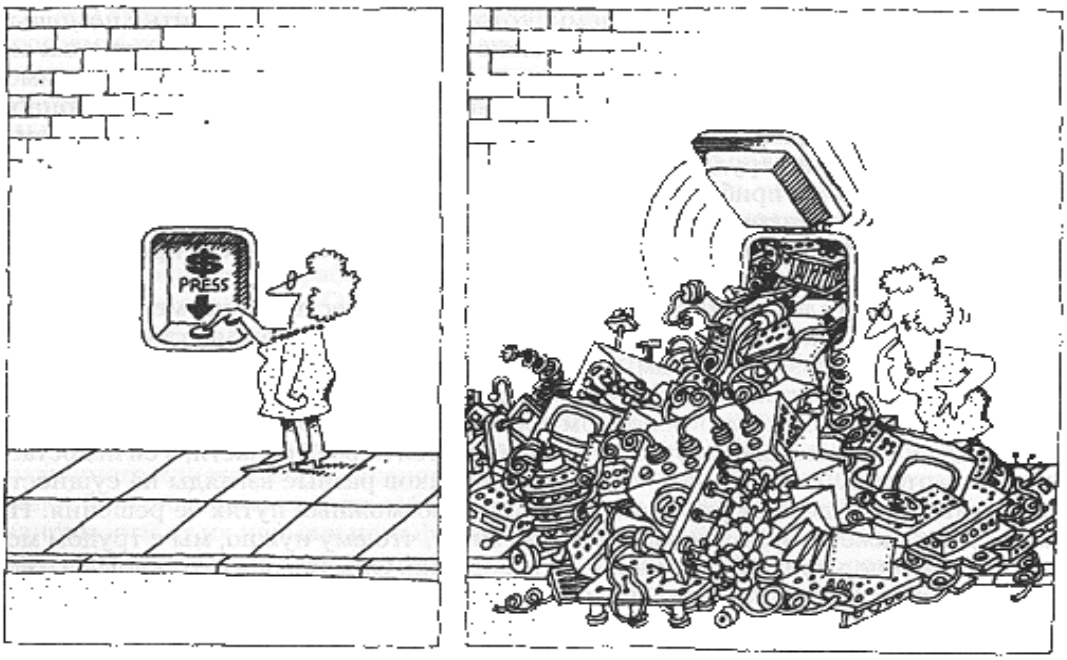


Рисунок 1.1 - Важлива задача розробника ПЗ – створити ілюзію простоти

Але крім зовнішньої простоти програміст повинен передбачити створення також і внутрішньої простоти, на скільки це можливо. У 1972 році Дейкстра сказав: «Компетентний програміст повністю усвідомлює розміри свого черепу, тому підходить до задач програмування з усією скромністю».

З 2 програм з однаковою функціональністю кращою буде та, що написана простішим алгоритмом і кодом. При чому розробка повинна слідувати поетапно та раціонально. Для цього використовують певні методології. Великого поширення сьогодні набув об’єктно-орієнтований підхід.

2. Основні положення об’єктної моделі

Об’єктно-орієнтоване програмування (ООП) – це методологія програмування, побудована на представленні програми у вигляді сукупності об’єктів, кожен із яких є екземпляром певного класу, а класи формують

ієрархію наслідування. Воно значною мірою відрізняється від алгоритмів і структур даних. Ключовим є об’єкти, через які безпосередньо і забезпечується взаємодія користувача та комп’ютера, а наслідком її стає створення програми, вирішення певної задачі. Така методологія програмування представлена далі (рис. 1.2):

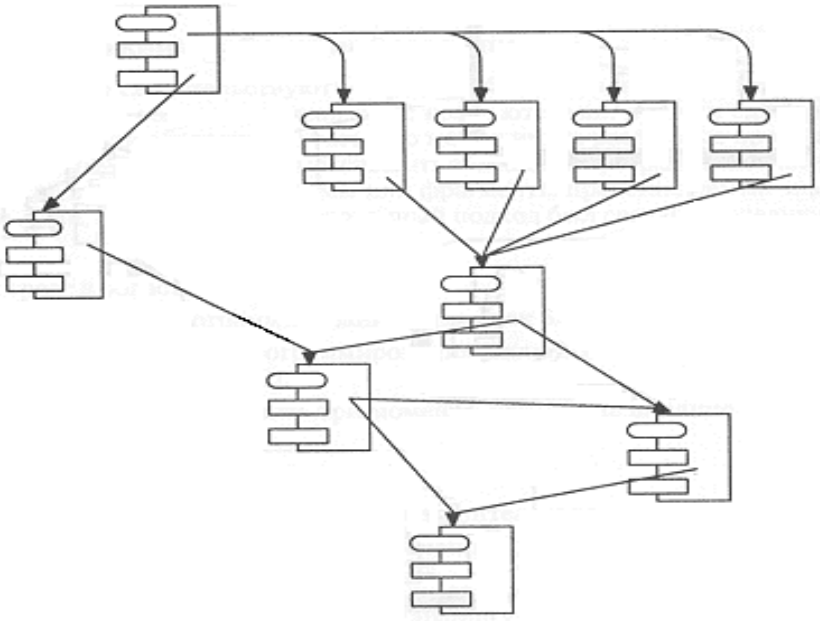


Рисунок 1.2 – Представлення об’єктно-орієнтованого програмування

Така методологія значно відрізняється від алгоритмів та структур даних і побудована на іншому принципі. Серед основних відмінностей та особливостей ООП у порівнянні зі структурним підходом виділяють:

– ООП використовує у якості базових елементів об’єкти, а не алгоритми;
– кожен об’єкт є екземпляром якого-небудь класу;
– класи організовані ієрархічно.

Взагалі об’єктно-орієнтоване програмування є частиною об’єктно-орієнтованого підходу. Основними структурними елементами цього підходу є такі етапи побудови вирішення задачі:

– ОО-аналіз – це методологія, за допомогою якої вимоги до системи сприймаються з точки зору класів і об’єктів, виявлених в предметній області;

– ОО-проектування – це методологія проектування, що поєднує в собі процес об’єктної декомпозиції та прийоми представлення логічної та фізичної, а також статичної та динамічної моделей системи, яка проектується.

І ключовим третім елементом виділяють якраз об’єктно-орієнтоване програмування. Тбто послідовність виконання проекту якраз і знаходиться в такому відношенні: ОО-аналіз -> ОО-проектування -> ОО-програмування.

3. Класи та об’єкти

На основі ООП програма складається з об’єктів. Кожен об’єкт екземпляром якого-небудь класу. В свою чергу класи організовані ієрархічно. Клас – це деяка множина об’єктів, що мають загальну структуру та загальну поведінку. Класи формуються в процесі ОО-декомпозиції і є певними абстракціями, які визначають певну поведінку, однак приховують деталі реалізації. Ідея контрактного програмування: “великі задачі розділити на багато маленьких і поручити їх дрібним субпідрядникам”.

4. Відмінність між класом і об’єктом

Але клас та об’єкт – не однорідні поняття. Між ними існує відмінність. Класи (певною мірою шаблони проектів) якраз і поділяються на менші задачі – об’єкти.

Об’єкт визначає лише певну сутність, що визначена в просторі і часі. Клас же - лише абстракцію суттєвого в об’єкті. Наглядно це демонструє подальший рисунок (рис. 1.3):

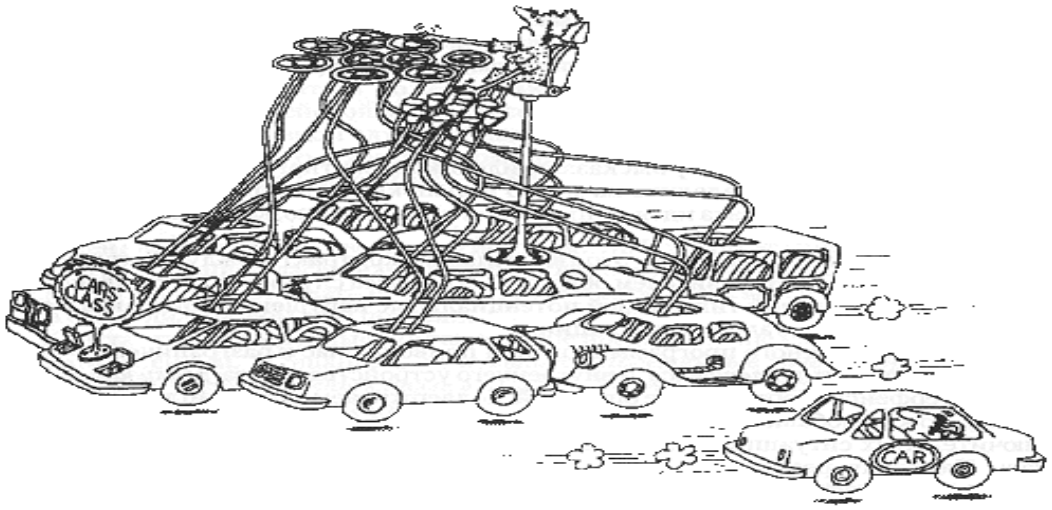


Рисунок 1.3 – Відмінність між класом та об’єктом

Клас фіксує можливості обмеження, об’єкт фіксує конкретні властивості певної задачі чи предмету.

5. Складові класу

Клас не являє собою однорідну сукупність. Це комплексна множина об’єктів. Тобто серед класу можна виділити його частини, що безпосередньо взаємодіють і мають доступ до об’єктів, визначають їх характеристики та стан. Найголовнішими в даному переліку виділяють: атрибут (поле), метод.

Атрибут (Поле) – це змінна чи константа, яка описана в самому класі і доступна в конкретному екземплярі класу (об’єкті).

Метод – визначає поведінку класу і є процедурою чи функцією, яка описана в класі і доступна для виклику в конкретному об’єкті.

6. Поля класу

Поля класу – аналог змінних у процедурних мовах програмування. Вони зберігають дані об’єкта. Вони можуть мати різну розмірність та містити різноманітні дані.

Набір певних значень полів становить інформацію про стан об’єкту, тобто це означає, що стан об’єкту визначається полями і тільки полями.

Як правило, правила доброго тону у ОО-програмуванні передбачають, що доступ до полів обмежений лише методами класу, а ззовні доступ до них відсутній.

7. Методи класу

Методи класу – аналог процедур і функцій у процедурних мовах програмування. Методи визначають поведінку об’єктів класу. На відміну від полів, значення яких відрізняються для різних екземплярів класів, методи є однаковими. Вони є постійними і відповідають за виконання певних команд і завдань з кодом. Вони можуть безпосередньо мати доступ до полів класу та оперувати ними.

8. Позначення класу

В загальному випадку клас позначається через ключове слово class, за яким слідує безпосередньо назва класу та дужки. Клас може містити всередині певні атрибути та методи. Доступ до них безпосередньо в класі не обмежений. Якщо ж потрібно звернутися до членів даного класу у іншому класі, слід створити посилання на даний клас. Тобто слід створити об’єкт даного класу, через який отримаємо доступ до атрибутів та методів класу

Приклад опису загальної структури класу зображає рисунок 1.4:

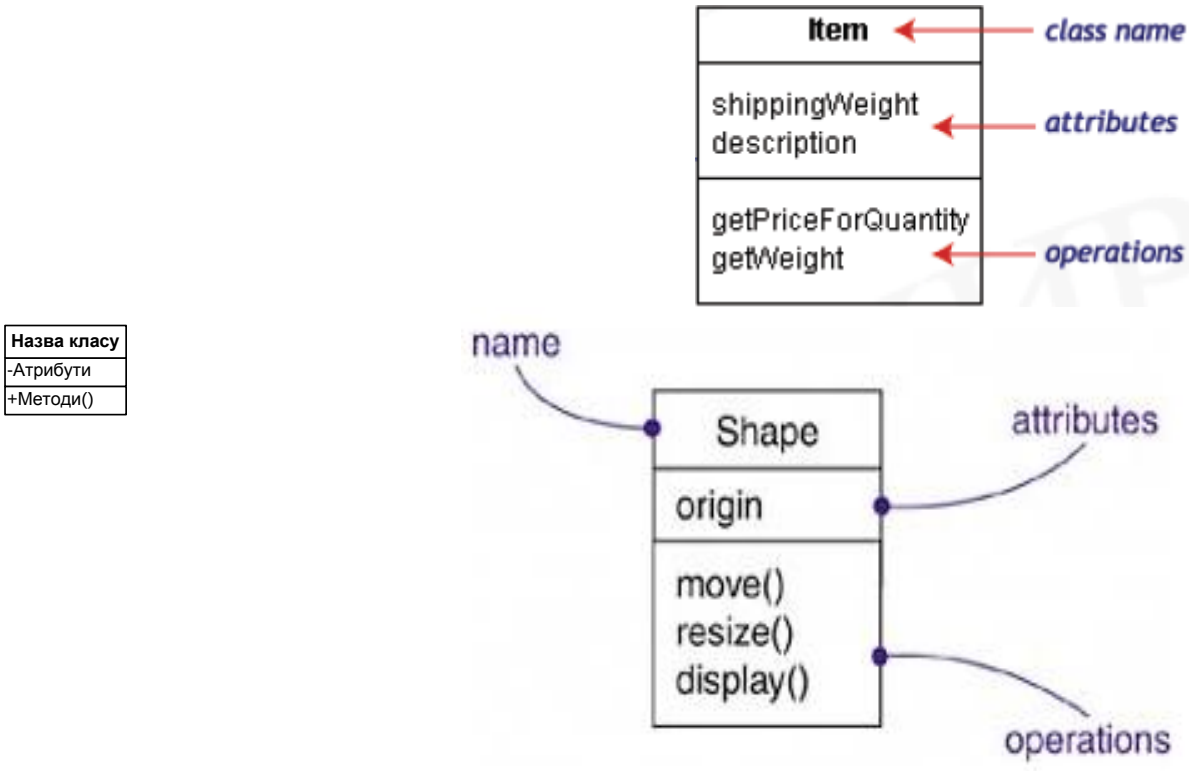


Рисунок 1.4 – Структура класу

9. Задання видимості членів класу

Члени класу можуть бути доступні у будь-якій часині коду програми або мати певний рівень захисту (обмеження до використання). Видимість членів класу задається користувачем. У разі відсутності задання – встановлюється автоматично на рівні доступу лише вередині класу.

Це задання видимості використовується для розмежування атрибутів у програмі, деякі роблять доступними у будь-якому класі, інші – лише у власному.

Для задання видимості мова програмування використовує певні ключові слова. Основними з них є:

+ (public) – доступ не обмежується. Члени класу видимі і доступні в усіх класах програми.

(protected) – доступ надається лише нащадкам класу.

- (private) – доступ наданий лише всередині класу, де створений певний атрибут чи метод.

~ (package) – доступ лише всередині одного пакету.

Приклад використання даних індикаторів видимості (рис. 1.5):

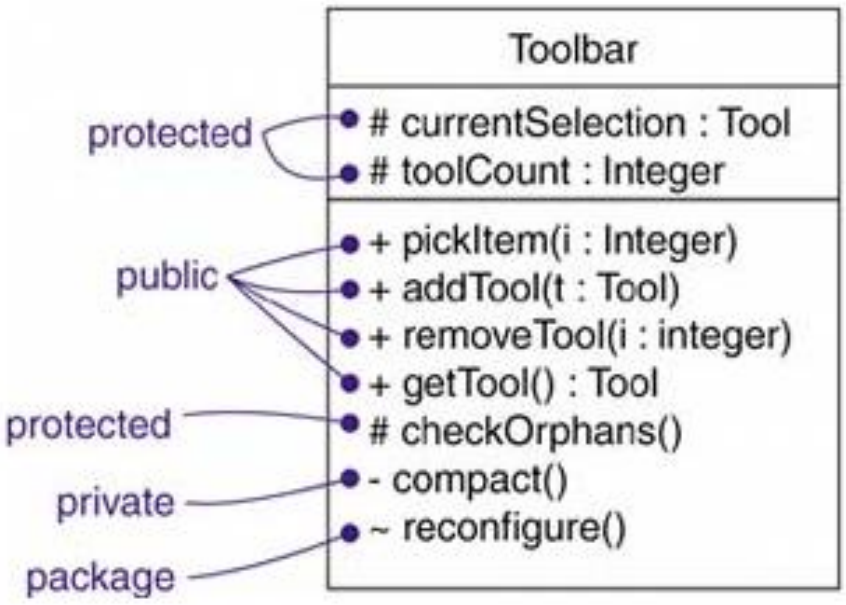


Рисунок 1.5 – Задання видимості членів класу

10.Інкапсуляція, наслідування та поліморфізм.

Обєктно-орієнтований підхід в своїй основі спирається на три фундаментальні принципи:

Інкапсуляція - означає поєднання даних і коду, “приховування” властивостей всередині об’єкта. Інкапсуляція – це процес відокремлення елементів об’єкту, які визначають його поведінку. Служить для того, щоб ізолювати контрактні обов’язки абстракції від їх реалізації (рис. 1.6).

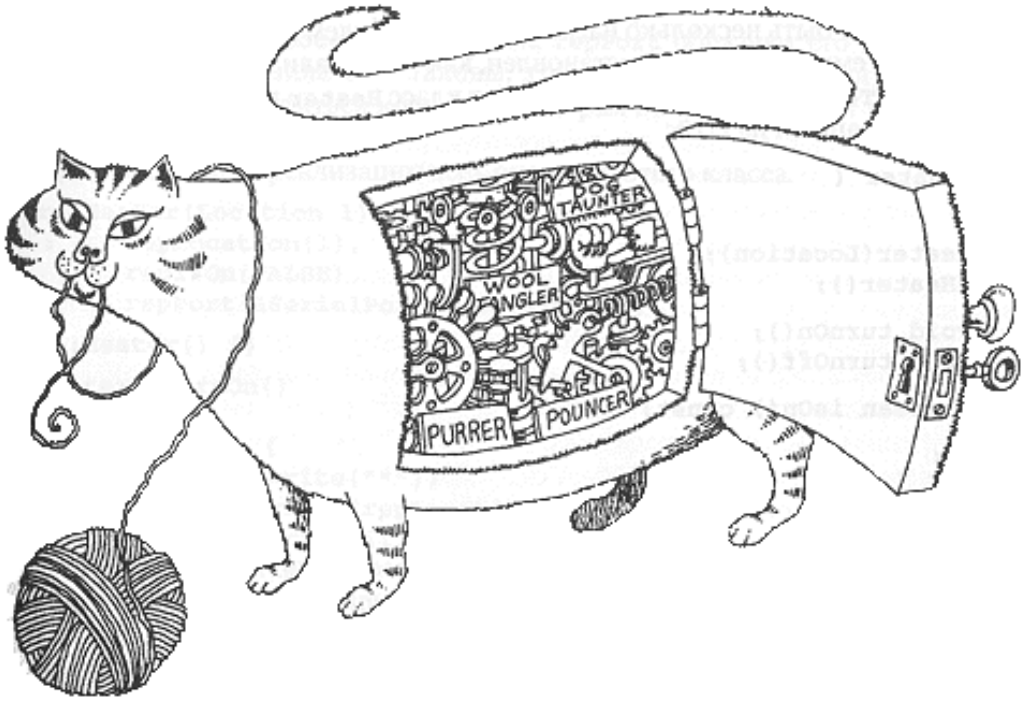


Рисунок 1.6 – Інкапсуляція

Наслідування - це здатність класів формувати ієрархії, коли класи-потомки наслідують властивості і поведінку класів-предків (рис. 1.7).

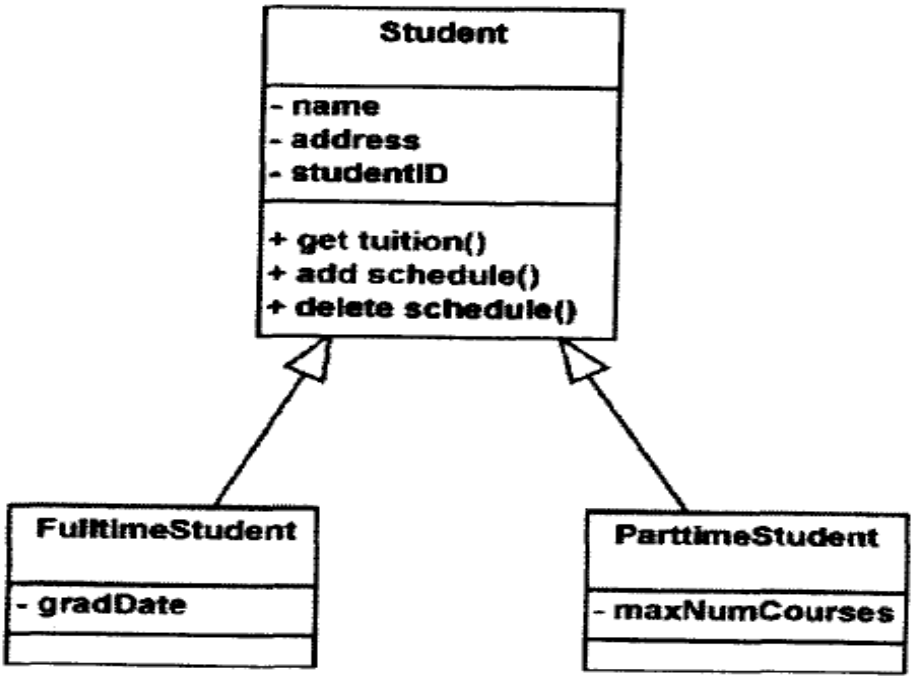


Рисунок 1.7 - Наслідування

Поліморфізм - здатність приховувати за однаковими назвами різну поведінку. Яскравим прикладом можна назвати метод «іти». Класи можуть утворювати діаграми. Діаграми класів можуть містити значну кількість різноманітних класів, пов’язаних певним відношенням (рис. 1.8).

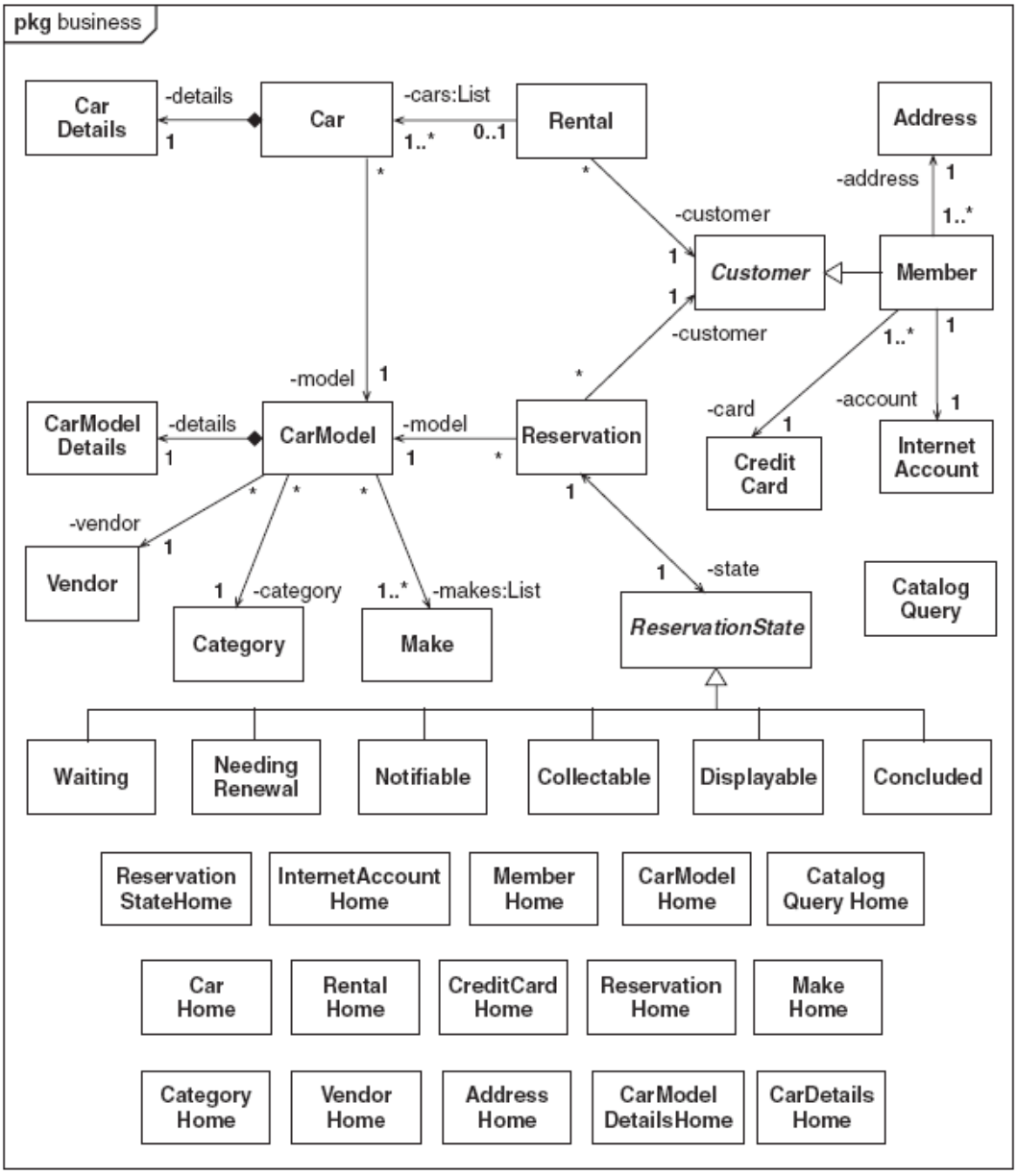


Рисунок 1.8 – Приклад діаграми класів

3. Реалізація класу та створення його екземпляру за допомогою об’єктно-орієнтованої мови програмування

Лістинг 1.1:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
  namespace ConsoleApplication1
{
class Student
    {
        public string Name;
        public string Group;
        public DateTime BirthDate;
        public Student(string TheName, string TheGroup, DateTime TheBirthDate)
        {
            Name = TheName;
            Group = TheGroup;
            BirthDate = TheBirthDate;
        }
        public int Age()
        {
           return (int)(DateTime.Now.Subtract(BirthDate).Days / 365.25);
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Student IvanovVasya = new Student("Іванов Василь", "ЕК-63", new DateTime(1989, 9, 13, 0, 0, 0));
            Console.WriteLine("Вік студента {0} складає {1} років", IvanovVasya.Name, IvanovVasya.Age());
            Console.ReadKey();
        }
    }
}
```

ЛК.02 – ОСНОВИ ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ У C#

Перелік питань:

1. Декларація класу
2. Створення екземпляру класу
3. Модифікатори доступу
4. Методи
5. Параметри
6. Перевантаження методів.
7. Члени класу.
8. Члени екземпляру класу.
9. Статичні поля.
10. Статичні методи.
11. Інші статичні члени класу.
12. Константи.
13. Властивості.
14. Конструктори об’єктів.
15. Статичні конструктори.
16. Фіналізатори.
17. Порівняння конструкторів і фіналізаторів.

На самостійне вивчення:

1. Інтерфейси [1, С.299-314]

1. Декларація класу

Декларація класу в мові програмування C# відбувається за допомогою використання ключового слова class, після якого слідує назва класу. Програма може мати декілька класів.

Клас – це визначений користувачем тип, який скомпоновано з полів даних і функцій, що впливають на ці дані. Багато полів даних в сукупності представляють «стан» екземпляра класу (рис. 2.1).

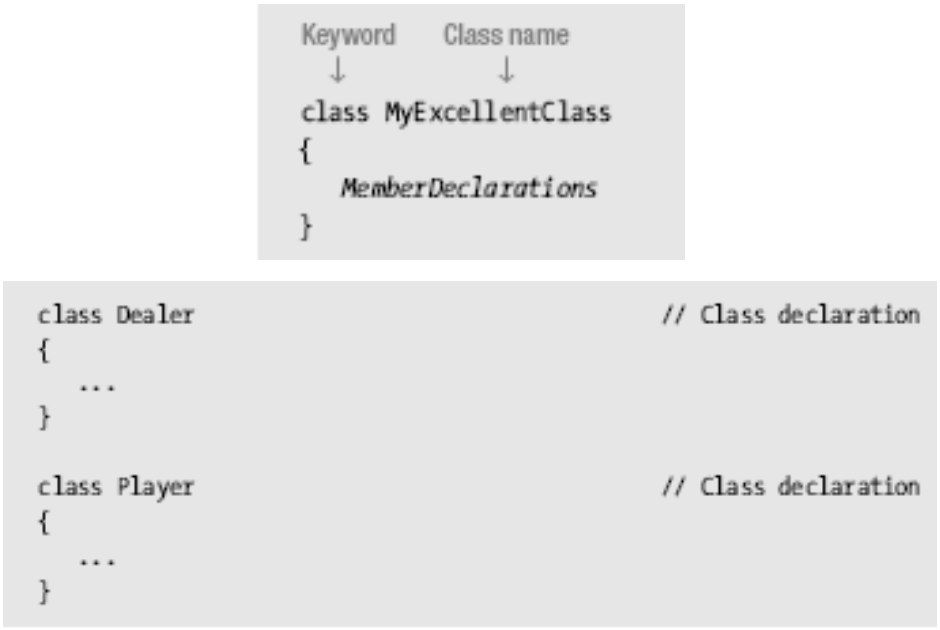


Рисунок 2.1 – Декларація класу

Декларація полів відбувається всередині класу. При чому можуть задаватися модифікатори доступу до них, а також тип даних (яку конкретну інформацію та розмірності міститиме дане поле). Обов’язковою умовою є назва поля, яка повинна бути унікальна в межах класу чи методу. Приклад декларації показано на рисунку 2.2:

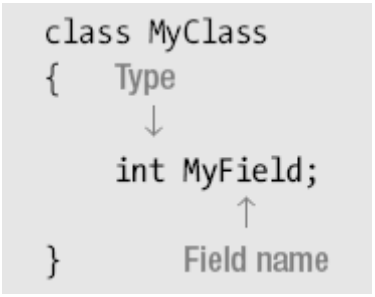
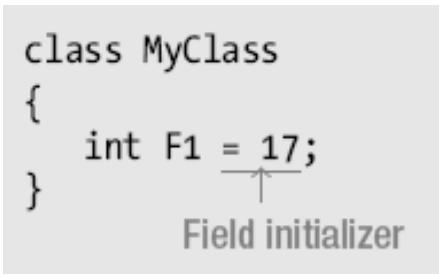


Рисунок 2.2 – Декларація поля

Ініціалізацією поля називається присвоєння йому певного значення. Це може бути конкретне значення, відповідне до заданого типу даних або початкове невизначене значення (0 для чисел, null для рядкових величин).

Ініціалізація полів як і декларація здійснюється в межах певного класу. Клас може містити довільну кількість полів. Ініціалізація полів з різними типами даних та з присвоєнням певних значень показана на рисунку 2.3:




```
int    F1, F3 = 25;
string F2, F4 = "abcd";
```

```
class MyClass
{
    int    F1;                 // Initialized to 0    - value type
    string F2;                 // Initialized to null - reference type

    int    F3 = 25;            // Initialized to 25
    string F4 = "abcd";        // Initialized to "abcd"
}
```

Рисунок 2.3 – Ініціалізація полів

В класі можуть декларуватися методи, що відповідають за певну функцію програмного коду. Методи представляють собою нестатичні функції всередині класу. Вони декларуються з використанням певного типу даних, що показує на повертання значення методом, відповідної назви та дужками, в яких містяться вхідні параметри для обробки даних в методі. Приклад такої декларації методу має такий вигляд (рис. 2.4):

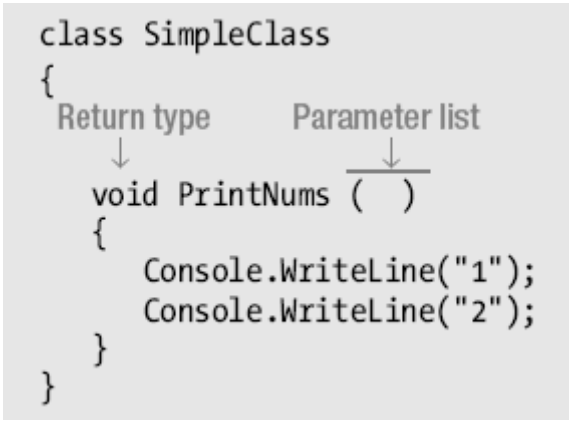
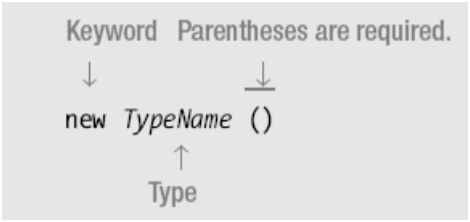


Рисунок 2.4 – Декларація методу

2. Створення екземпляру класу

Щоб звернутися до полів та методів іншого класу, потрібно створити екземпляр даного клас у. Принцип створення екземпляру показано на рисунку 2.5:




```
Dealer TheDealer;              // Declare variable for the reference.
TheDealer = new Dealer();      // Allocate memory for the class object.
```

Object-creation expression

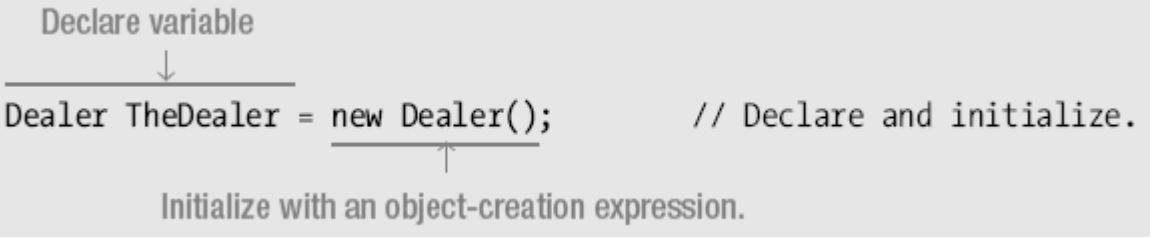


Рисунок 2.5 – Створення екземпляру класу

При цьому відбувається виділення пам’яті певного розміру, в яку буде записуватися всі дані про поля та методи. При створенні екземпляру класу відбувається виділення пам’яті у стеку (рис. 2.6).

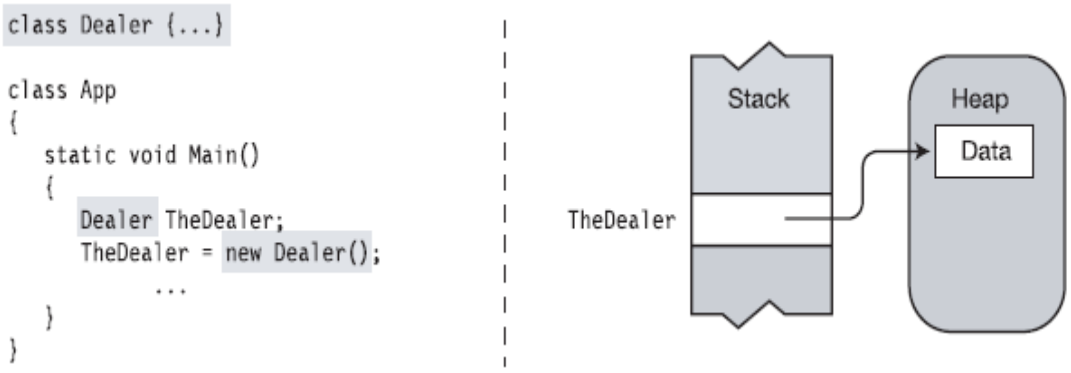


Рисунок 2.6 – Виділення пам’яті

Мова програмування C# дозволяє працювати з різними даними. Можна створювати декілька екземплярів певного класу і працювати з ними окремо та незалежно.

При чому у стеку буде виділено окремо пам'ять для кожного екземпляру класу, де окремо зберігатимуться дані полів та проводитимуться операції над ними за допомогою методів. Створення декількох екземплярів класу та рисунок виділення пам’яті показано на рисунках 2.7, 2.8:

```
class Dealer { ... }                                    // Declare class
class Player {                                          // Declare class
   string Name;                                         // Field
      ...
}

class Program {
   static void Main()
   {
      Dealer TheDealer = new Dealer();
      Player Player1 = new Player();
      Player Player2 = new Player();
      Player Player3 = new Player();
      ...
   }
}
```

Рисунок 2.7 – Створення декількох екземплярів класу

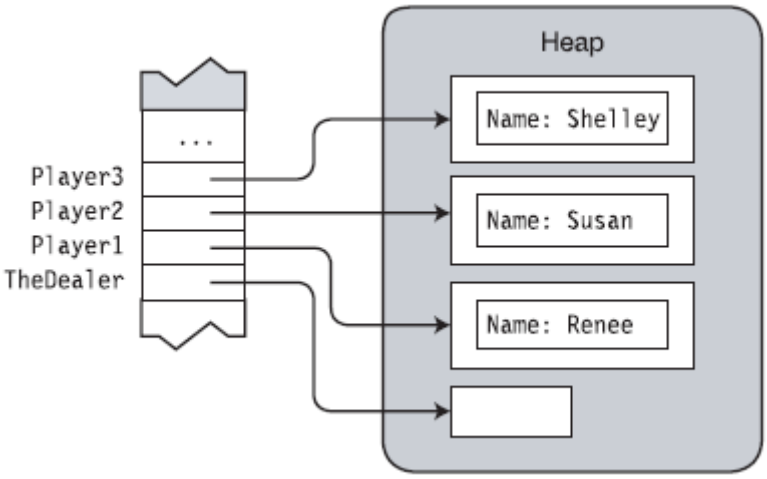


Рисунок 2.7 – Виділення пам’яті для кожного екземпляру класу

3. Модифікатори доступу

Модифікатори доступу забезпечують рівень доступності до атрибутів певного класу (полів та методів, конструкторів). Вони можуть задаватися користувачем як для полів, так і для методів, за замовчуванням програма встановлює доступність на рівні доступності лише в межах класу (рис. 2.9).

```
Fields
  AccessModifier Type Identifier;

Methods
  AccessModifier ReturnType MethodName ()
  {
     ...
  }
```

Рисунок 2.9 – Модифікатори доступу для полів і методів

Основними модифікаторами доступу є:

- public (даний елемент класу доступний всім зовнішнім споживачам);
- protected (до такого елементу класу можуть звертатися тільки класи, успадковані від даного);
- private (елемент недоступний за межами опису класу, тобто недоступний навіть нащадкам даного класу; цей модифікатор ставиться за замовчуванням);
- internal (елемент доступний тільки для класів, визначених в тій же збірці, що і даний клас);
- protected internal (доступ із класу з його описом та підкласів, а також збірки з описом об’єкту).

Основними серед вищеназваних є модифікатори private та public. Різниця у використанні показана на рисунках 2.10, 2.11:

```
        int MyInt1;                 // Implicitly declared private
private int MyInt2;                 // Explicitly declared private
   ↑
Access modifier
```

```
Access modifier
   ↓
public int MyInt;
```

Рисунок 2.10 – Декларування полів

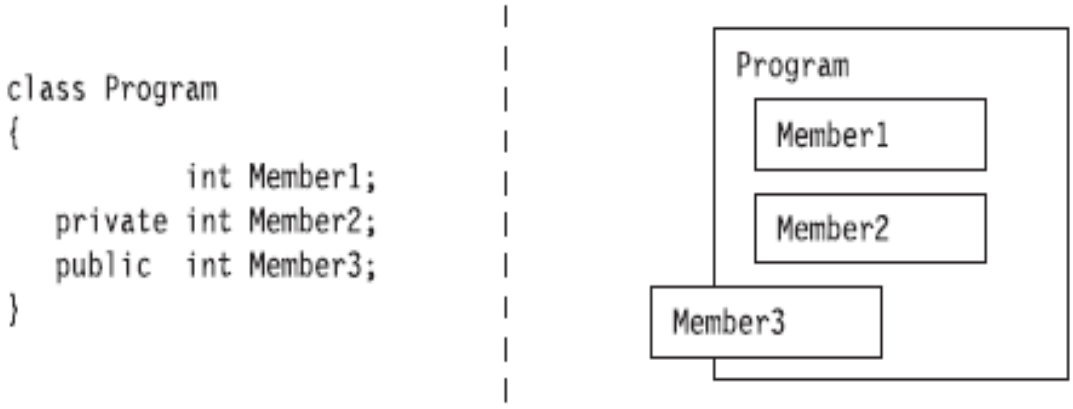


Рисунок 2.11 – Різниця між модифікаторами

Доступність можна декларувати й для методів. Якщо поле або метод є приватними, то доступ і видимість їх обмежена лише всередині даного класу, де вони задекларовані. В іншому випадку до таких полів та методів можна звертатись у будь-якому класі програми (рис. 2.12):

```
class C1
{
   int         F1;                      // Implicit private field
   private int F2;                      // Explicit private field
   public  int F3;                      // Public field

   void DoCalc()                       // Implicit private method
   {
      ...
   }

   public int GetVal()                 // Public method
   {
      ...
   }
}
```

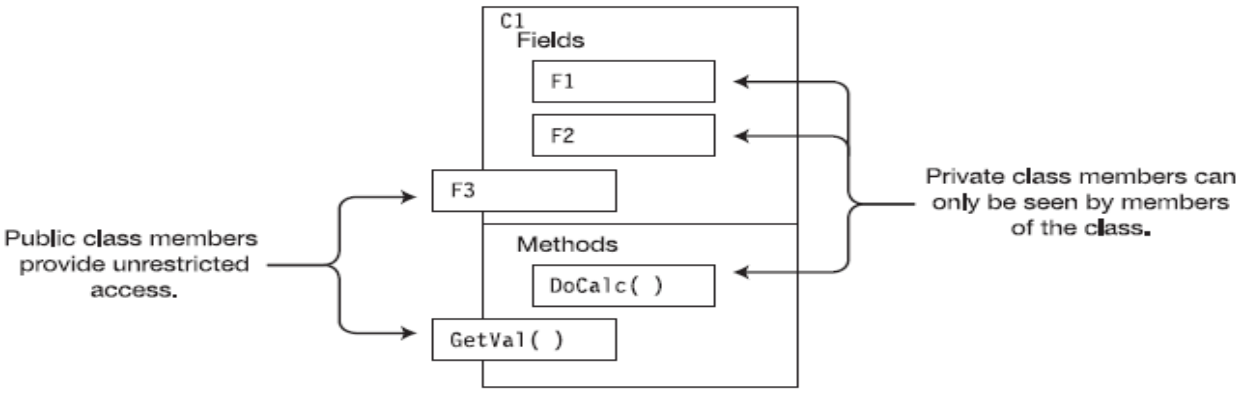


Рисунок 2.12 – Приклад доступності членів класу

Розрізняють доступ до членів класу всередині класу. Фактично всі задекларовані члени класу будуть доступними в даному класі. Це демонструє рисунок 2.13:

```
class DaysTemp
{
   // Fields
   private int High = 75;
   private int Low  = 45;

   // Methods
   private int GetHigh()
   {
      return High;                          // Access private field
   }

   private int GetLow()
   {
      return Low;                           // Access private field
   }

   public float Average ()
   {
      return (GetHigh() + GetLow()) / 2;    // Access private methods
   }
}
```

Accessing the private methods

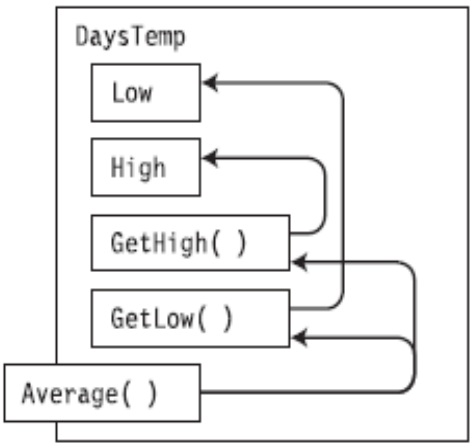


Рисунок 2.13 – Приклад доступу до членів класу всередині даного класу

Доступ до членів класу ззовні обмежується в залежності від модифікаторів доступу. Якщо члени класу публічні, то до них можна звертатися безпосередньо з будь-якого іншого класу (рис. 2.14):

```
class DaysTemp                              // Declare the class.
{
   public int High, Low;                    // Declare the instance fields.
   public int Avg()                         // Declare the instance method.
   {
      return (High + Low) / 2;
   }
}

class Class1
{
   static void Main()
   {
      DaysTemp T1 = new DaysTemp();
      // Create 2 instances of DaysTemp.
      DaysTemp T2 = new DaysTemp();

      // Write to the fields of each instance.
      T1.High = 76;  T1.Low  = 57;
      T2.High = 75;  T2.Low  = 53;

      // Read from the fields of each instance and call a method of
      // each instance.
      Console.WriteLine("T1: {0}, {1}, {2}", T1.High, T1.Low, T1.Avg ());
      Console.WriteLine("T2: {0}, {1}, {2}", T2.High, T2.Low, T2.Avg ());
                                                  ↑       ↑       ↑
   }                                            Field   Field   Method
}
```

Рисунок 2.14 – Приклад доступності членів класу ззовні

4. Методи

Декларація методів включає в себе визначення типу даних, що будуть повертатися методом, присвоєння певного імені та виокремлення параметрів, що будуть вхідними у метод. Основи декларації методу показує рисунок 2.15:

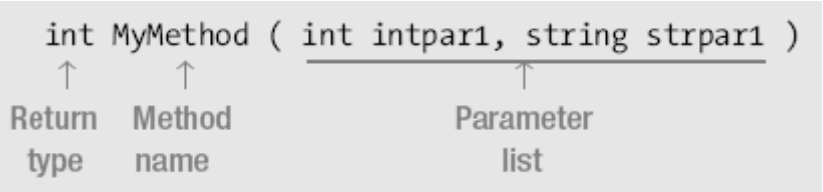


Рисунок 2.15 – Декларування методу

В структурі методу можуть проводитися різноманітні операції, ініціалізація та декларація змінних, приведення циклів тощо (рис. 2.16).

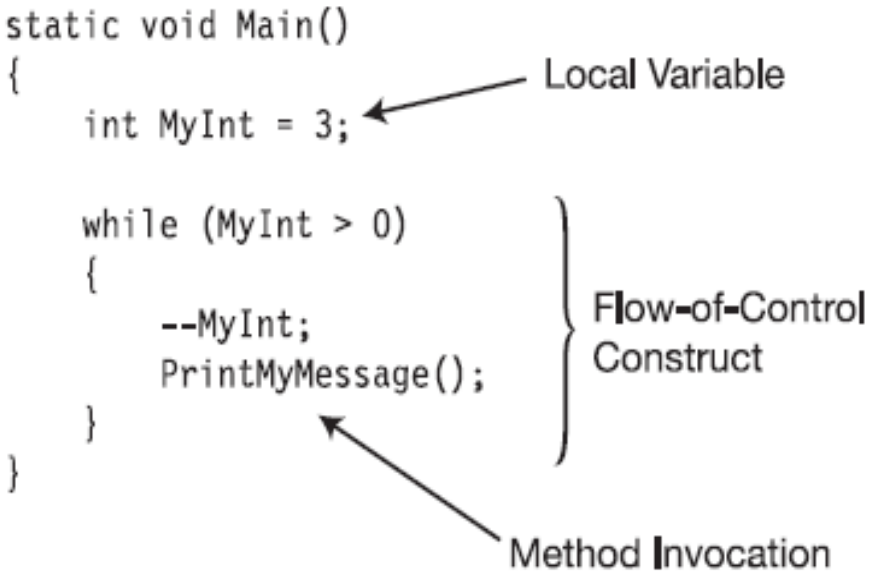


Рисунок 2.16 – Структура методу

Поряд з полями існують локальні змінні, що існують в 1 блоці. Їх декларування представлене на рисунку 2.17, а різниця – в таблиці 1. Як

правило, вони використовуються у методах для проведення допоміжних обчислень. Можуть повертати певне значення на виході з методу.

```
static void Main( )
{
   int FirstInt  = 15;
   int SecondInt = 13;

   int Total = FirstInt + SecondInt;
   ...
}
```

Рисунок 2.17 – Декларування локальних змінних

Таблиця 1 – Порівняння полів та локальних змінних

	Поля	Локальні змінні
Життя	Починається, коли створюється екземпляр класу. Закінчується після останнього звертання до об’єкту.	Починається в точці, де була декларована. Закінчується, коли блок, де була задекларована змінна, закінчує виконання.
Неявна ініціалізація	Присутня значенням за замовчуванням для даного типу	Відсутня. Значення залишається невизначеним до першого

		присвоювання.
Місце зберігання	Завжди зберігаються у купі	Типи за значенням зберігаються у стеку, за посиланням – у купі

Також локальні змінні існують всередині вкладених блоків в методах. При декларуванні та одночасній ініціалізації локальних змінних виділяється частина пам’яті у стеку (рис. 2.18).

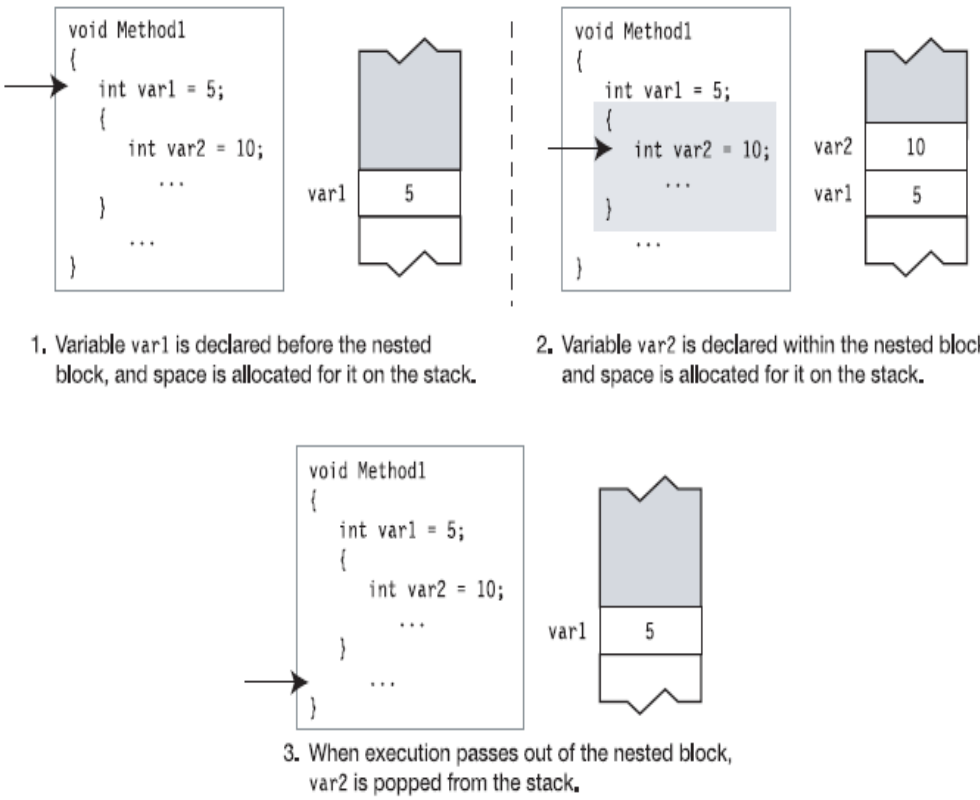


Рисунок 2.18 – Локальні змінні усередині вкладених блоків

Для управління ходом виконання програми C# надає такий синтаксис програмування:

Умови:

- if
- if...else
- switch

Цикли:

- for
- while
- do
- Foreach

Команди переходу:

- break
- continue
- goto
- return

Для виклику методів у іншому класі потрібно спочатку створити екземпляр класу, що містить метод, а потім безпосередньо звернутися до даного методу через цей екземпляр (рис. 2.19).

```
class MyClass
{
   void PrintDateAndTime( )                  // Declare the method.
   {
      DateTime dt = DateTime.Now;            // Get the current date and time.
      Console.WriteLine("{0}", dt);          // Write it out.
   }

   static void Main()                        // Declare the method.
   {
      MyClass mc = new MyClass();
      mc.PrintDateAndTime( );                // Invoke the method.
   }          ↑           ↑
}         Method name  Empty parameter list
```

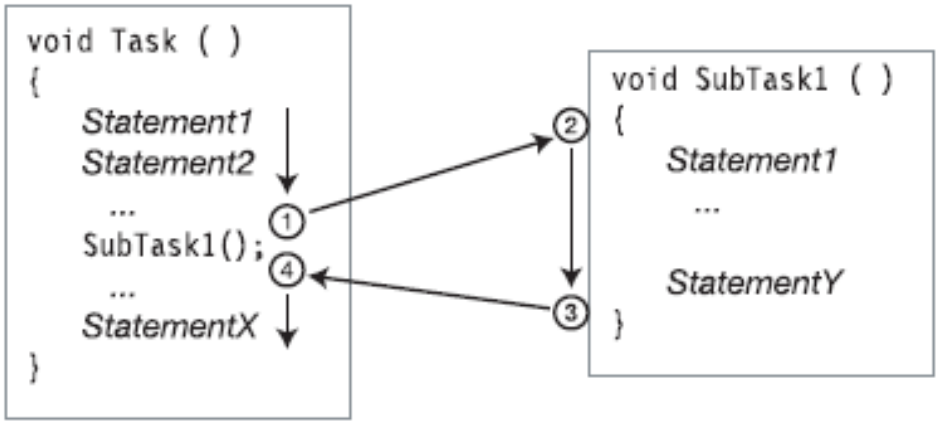


Рисунок 2.19 – Виклик методів

Тип даних, який вказується при декларуванні методів, автоматично визначає і повернені значення цих методів. На рисунках 2.20, 2.21 показано різницю між повернеными значеннями:

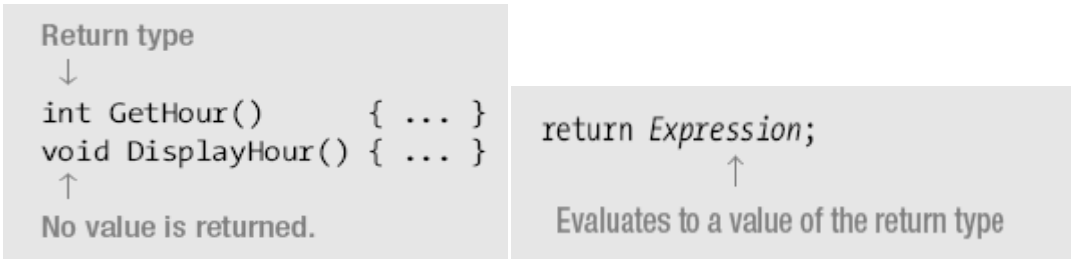


Рисунок 2.20 – Повернення значень

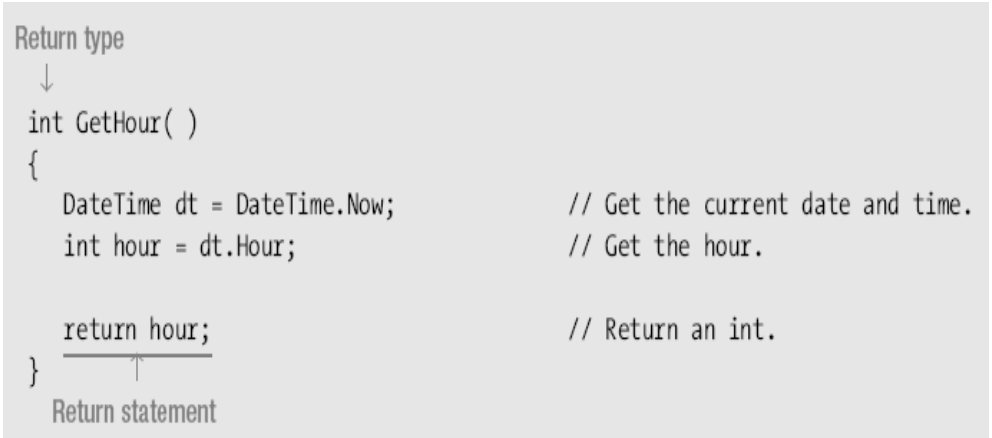


Рисунок 2.21 – Приклад повернення значення

Також можливе повернення не лише конкретного певного значення, а й повернення об’єктів (рис. 2.22).

```
Return type -- MyClass
   ↓
MyClass method3{ )
{
   MyClass mc = new MyClass();
      ...
   return mc;                                  // Return a MyClass object.
}        ↑
```

Рисунок 2.22 – Приклад повернення об’єкту

А за допомогою спеціально реалізованої команди return можливе повернення значення всередині виразу (рис. 2.23):

```
class MyClass
{          ↓ Return type
   public int GetHour()
   {
      DateTime dt = DateTime.Now;           // Get the current date and time.
      int hour = dt.Hour;                   // Get the hour.

      return hour;                          // Return an int.
   }          ↑
}           Return value


class Program
{
   static void Main()
   {                                    Method invocation
      MyClass mc = new MyClass();          ↓
      Console.WriteLine("Hour: {0}", mc.GetHour());
   }                                       ↑     ↑
}                                  Instance name  Method name
```

Рисунок 2.23 – Повернення значення всередині виразу

Команда return може використовуватися і для повернення значень з void-методів. Адже цей тип методу сам не повертає значення, тому ця команда робить це примусово при необхідності (рис. 2.24).

```
class MyClass {
    ↓ Void return type
   void TimeUpdate() {
      DateTime dt = DateTime.Now;          // Get the current date and time.
      if (dt.Hour < 12)                    // If the hour is less than 12,
         return;                           // then return.
           ↑
      Return to calling method.
      Console.WriteLine("It's afternoon!");   // Otherwise, print message.
   }

   static void Main() {
      MyClass mc = new MyClass();         // Create an instance of the class.
      mc.TimeUpdate();                    // Invoke the method.
   }
}
```

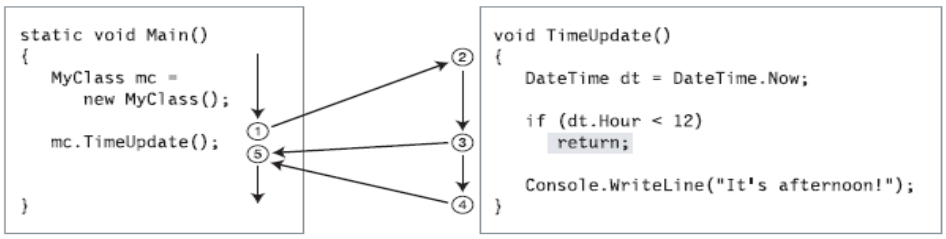


Рисунок 2.24 – Використання return для void-методів

5. Параметри

Розрізняють формальні, актуальні та параметри за значенням і параметри за посиланням. Формальні параметри представлені на рисунку 2.25 і мають свої особливості декларації. Актуальні параметри використовуються для актуалізації формальних. Рисунок 2.26 показує способи використання актуальних параметрів:

```
public void PrintSum( int x, float y )
{ ... }
                      ↑
              Formal parameter declarations
```

Рисунок 2.25 – Формальні параметри

```
PrintSum( 5, SomeInt );
          ↑     ↑
   Expression   Variable of type int
```

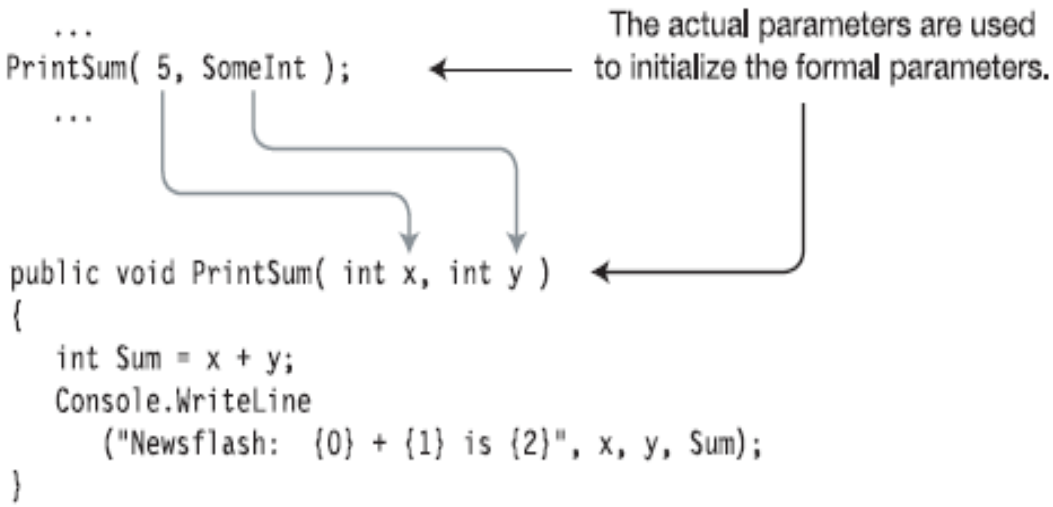


Рисунок 2.26 – Актуальні параметри

Виділяється окремо ще група параметрів за значенням. Вони задаються з попереднім урахуванням значень і тримають ці значення за замовчуванням. Приклад їх використання поряд з іншими видами параметрів такий (рис. 2.27):

```
class MyClass
{ public int Val = 20; }                    // Initialize the field to 20.

class Program                  Formal parameters
{                                    ↓
   static void MyMethod( MyClass f1, int f2 )
   {
      f1.Val = f1.Val + 5;                  // Add 5 to field of f1 param.
      f2     = f2 + 5;                      // Add 5 to second param.
   }

   static void Main( )
   {
      MyClass A1 = new MyClass();
      int     A2 = 10;

      MyMethod( A1, A2 );                   // Call the method.
   }                 ↑
}          Actual parameters
```

Рисунок 2.27 – Параметри за значенням

Виділення пам’яті для параметрів за значенням відбувається через звернення (посилання) до купи, в якій і замінюється певне значення внаслідок операцій над ним (рис. 2.28):

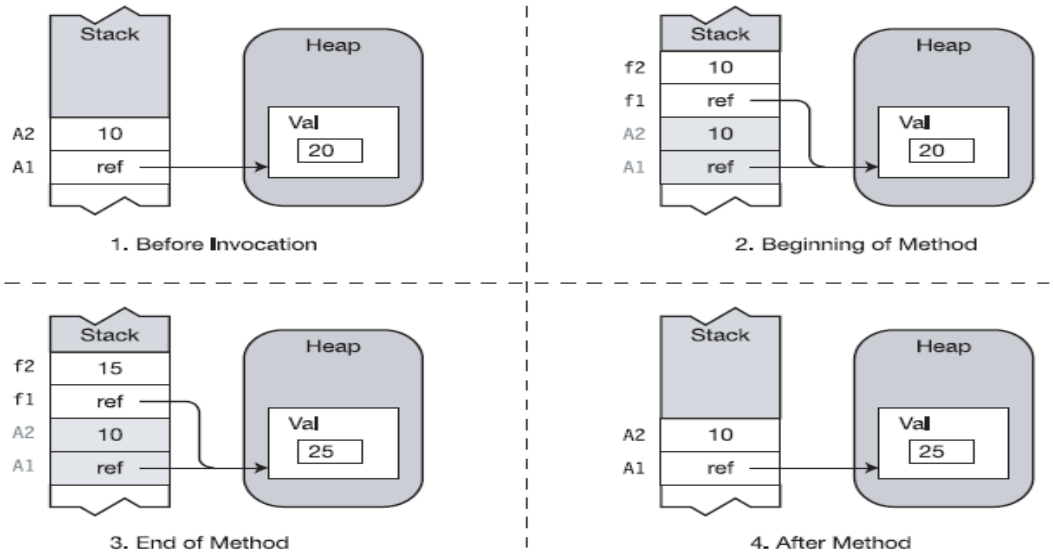


Рисунок 2.28 – Виділення пам’яті для параметрів за значенням

Параметри за посиланням використовують ключове слово ref. Приклад декларації таких параметрів показано на рисунку 2.29, а використання – на рисунку 2.30:

```
          Include the ref modifier.
                    ↓
void MyMethod( ref int val )          // Method declaration
{ ... }

int y = 1;                            // Variable for the actual parameter
MyMethod ( ref y );                   // Method call
             ↑
    Include the ref modifier.

MyMethod ( ref 3+5 );                 // Error!
               ↑
          Must use a variable.
```

Рисунок 2.29 – Параметри за посиланням

```
class MyClass
{ public int Val = 20; }                    // Initialize field to 20.

class Program          ref modifier        ref modifier
{                          ↓                   ↓
   static void MyMethod(ref MyClass f1, ref int f2)
   {
      f1.Val = f1.Val + 5;                  // Add 5 to field of f1 param.
      f2 = f2 + 5;                          // Add 5 to second param.
   }

   static void Main()
   {
      MyClass A1 = new MyClass();
      int A2 = 10;

      MyMethod(ref A1, ref A2);                          // Call the method.
   }           ↑       ↑
}              ref modifiers
```

Рисунок 2.30 - Приклад використання параметрів за посиланням

Управління пам’яттю при використанні параметрів за посиланням через посилання значень кількох параметрів на 1 частину стеку здійснюється поступово, проходячи кроки звернення зі стеку до купи, запиту значення, проведення певних операцій, збереження результату через процес посилання до купи та приведення результату. Це показує рисунок 2.31:

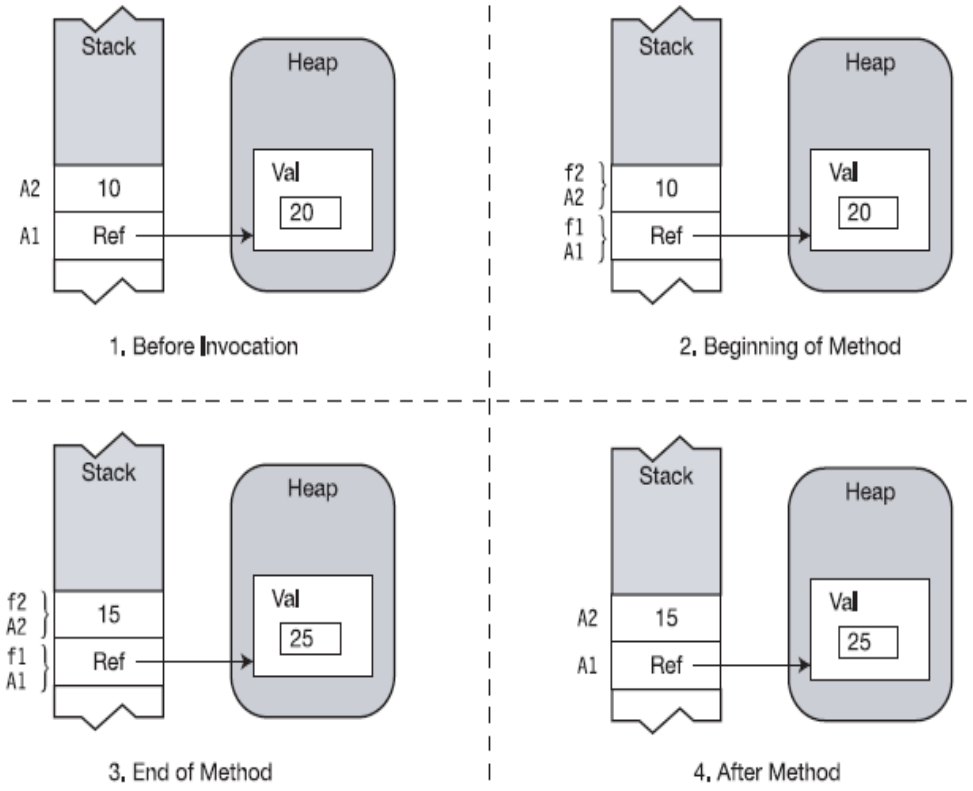


Рисунок 2.31 – Управління пам’яттю при параметрах за посиланням

Існують параметри, які повертають значення. Здебільшого вони використовуються в методах і мають такий вигляд (рис. 2.32, 2.33):

```
                out modifier
                     ↓
void MyMethod( out int val )          // Method declaration
{ ... }

...
int y = 1;                            // Variable for the actual parameter
MyMethod ( out y );                   // Method call
                ↑
          out modifier
```

Рисунок 2.32 – Вигляд параметрів, які повертають значення

```
class MyClass
{ public int Val = 20; }                    // Initialize field to 20.

class Program          out modifier          out modifier
{                            ↓                     ↓
   static void MyMethod(out MyClass f1, out int f2)
   {
      f1 = new MyClass();                   // Create an object of the class.
      f1.Val = 25;                          // Assign to the class field.
      f2     = 15;                          // Assign to the int param.
   }

   static void Main()
   {
      MyClass A1 = null;
      int A2;

      MyMethod(out A1, out A2);             // Call the method.
   }            ↑      ↑
}               out modifiers
```

Рисунок 2.33 – Використання параметрів за значенням

Управління пам’яттю при використанні параметрів, які повертають значення також зображає розглянута схема (рис. 2.34).

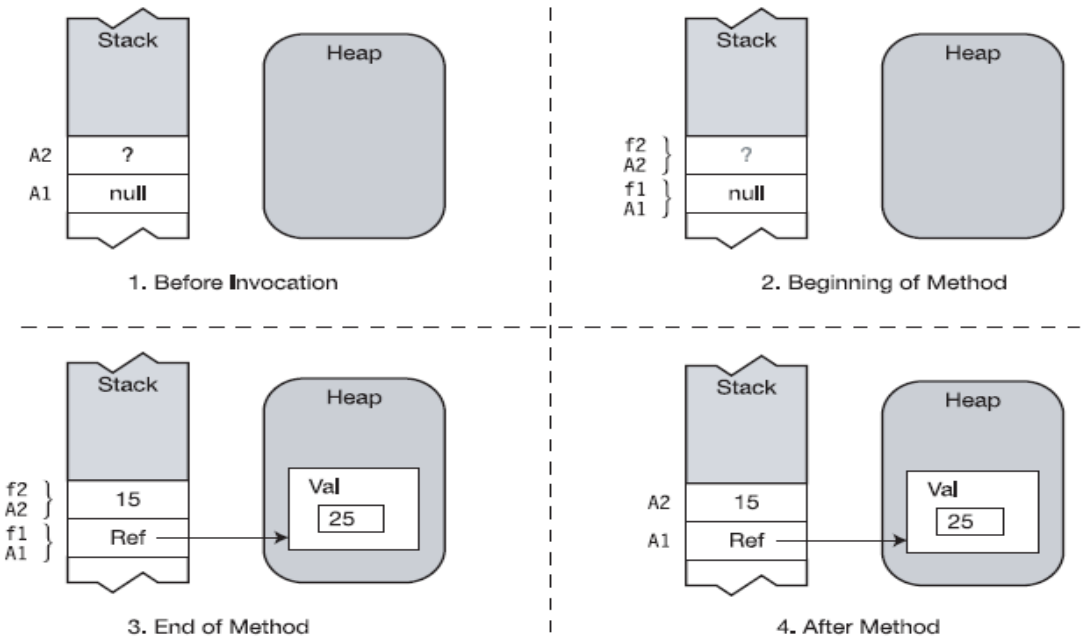


Рисунок 2.34 – Використання пам’яті при роботі методів

Параметри можуть викликатися, створюватися не тільки поодинці, а й утворювати масиви. Ці масиви відповідно можуть передаватися як параметри для певних методів з їх подальшим використанням. Варіант використання масиву параметрів показує рисунок 2.35:

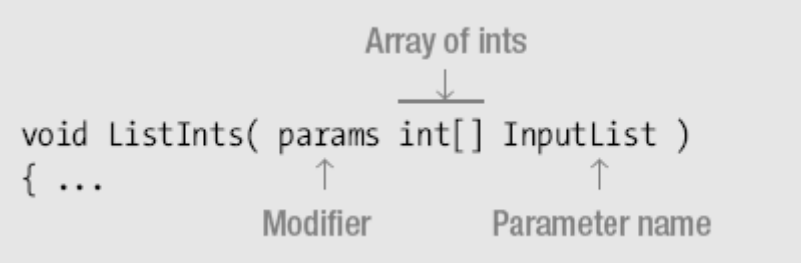


Рисунок 2.35 – Використання масивів параметрів

Існують різні варіанти виклику методів, якщо в якоті їх параметрів служать масиви. Декілька прикладів показано таким чином (рис. 2.36):

```
ListInts( 10, 20, 30 );
```

```
int[] IntArray = {1, 2, 3};
ListInts( IntArray );
```

Рисунок 2.36 – Виклик методів з масивами параметрів

Можливе також використання рекурсії. В такому випадку параметри використовуються таким чином (рис. 2.37):

```
class Program
{
   public void Count(int InVal)
   {
      if (InVal == 0)
         return;
      Count(InVal - 1);              // Invoke this method again.
        ↑
     Calls itself
      Console.WriteLine("{0} ", InVal);
   }

   static void Main()
   {
      Program pr = new Program();
      pr.Count(3);
   }
}
```

Рисунок 2.37 – Параметри у випадку рекурсії

В такому випадку управління пам’ятю зміне свій вигляд і набуде його таким, як показано на рисунку 2.38:

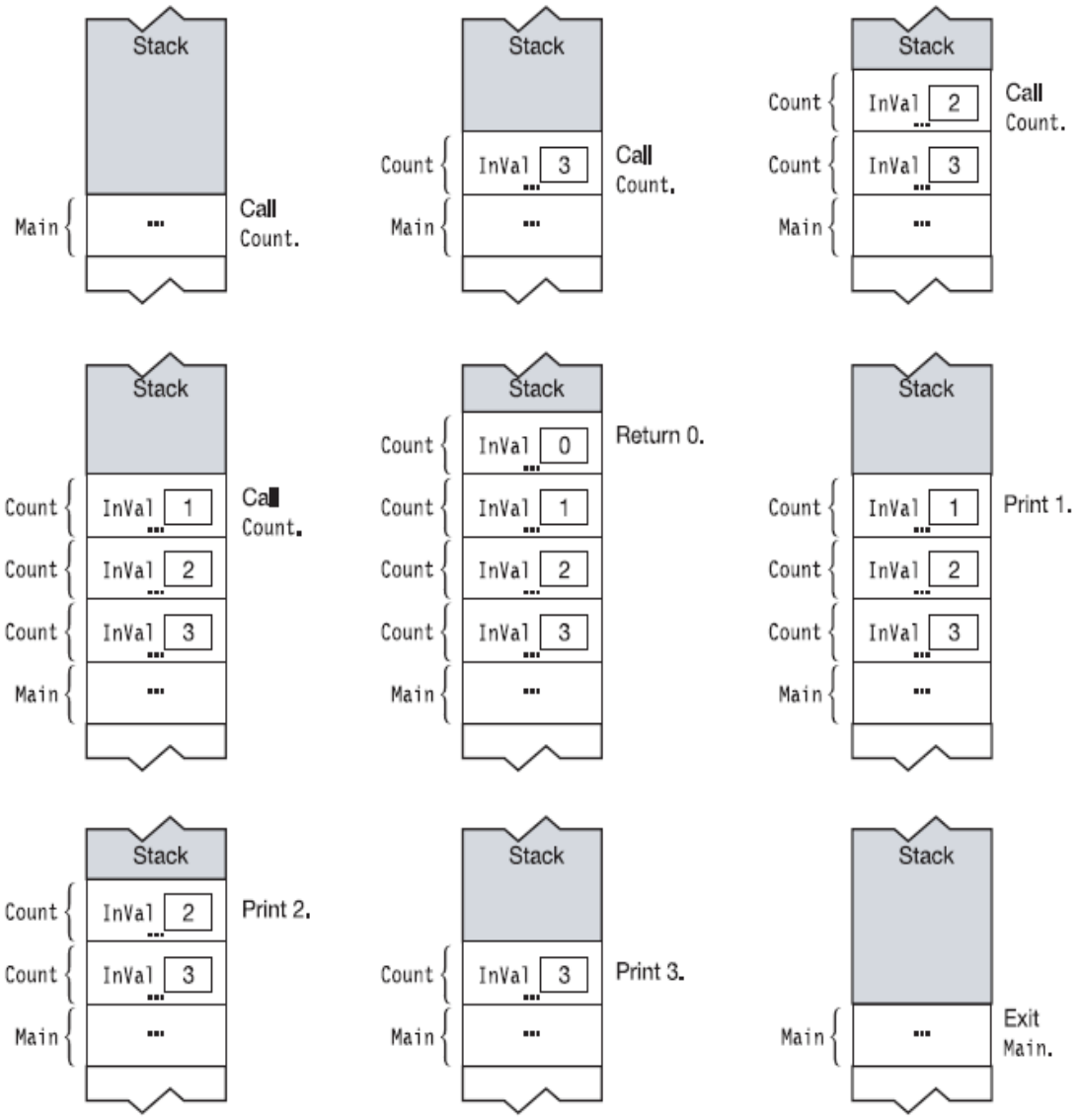


Рисунок 2.38 – Управління пам’ятю при рекурсії

6. Перевантаження методів

Клас може мати більш ніж один метод з однаковим іменем, проте кожен з цих методів повинен відрізнятися сигнатурою. Така ситуація називається “перевантаження методів”. В даному випадку можна зробити помилки. Тому рисунок 2.39 демонструє нам приклад вірного використання перевантажених методів, а рисунок 2.40 – невірне.

```
class A
{
   long AddValues( int   a, int   b)          { return a + b; }
   long AddValues( int   a, int   b, int c)   { return a + b + c; }
   long AddValues( float a, float b)          { return a + b; }
   long AddValues( long  a, long  b)          { return a + b; }
}
```

Рисунок 2.39 – Вірне використання перевантажених методів

```
class B          Signature
{                    ↓
   long AddValues( long  a, long  b) { return a+b; }
   int  AddValues( long  c, long  d) { return a+b; }
}                    ↑
                 Signature
```

Рисунок 2.40 – Невірне використання перевантажених методів

7. Члени класу

Мова C# всього налічує 9 можливих членів класу:

- поля;
- константи;
- методи;
- властивості;
- конструктори;
- фіналізатори;
- оператори;
- індексатори;
- події.

Поля і константи зберігають дані, усі інші члени класу пов’язані з програмним кодом. Тобто вони використовують поля та константи для

проведення певних дій та досягнення необхідного результату. Почнемо розгляд почергово з всіх членів екземпляру класу.

8. Члени екземпляру класу

Члени класу можуть бути асоційовані із екземпляром, чи із самим класом. Тобто вони можуть відноситися до екземпляру класу або викликатися при використанні самого класу.

По замовчуванню вони асоціюються із екземпляром – це означає, що кожен об’єкт має власну копію і вони називаються членами екземпляру класу. Відповідно, зміни до значень у одному екземплярі класу не впливають на значення у інших. Це важливо для збереження даних, станів, загальної безпеки всього програмного коду через всю програму. Приклад використання членів екземпляру класу показано на рисунку 2.41:

```
class D
{
   public int Mem1;
}

class Program
{
   static void Main()
   {
      D d1 = new D();
      D d2 = new D();
      d1.Mem1 = 10; d2.Mem1 = 28;

      Console.WriteLine("d1 = {0}, d2 = {1}", d1.Mem1, d2.Mem1);
   }
}
```

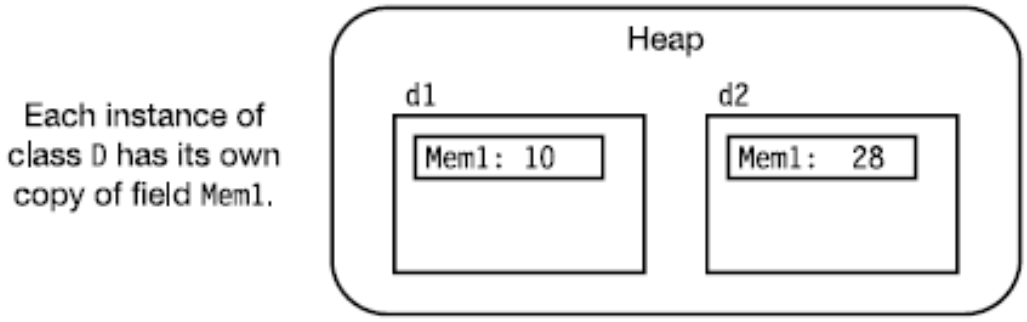


Рисунок 2.41 – Члени екземпляру класу

9. Статичні поля

Класи можуть мати також статичні поля, які є загальними для всіх екземплярів одного класу, оскільки займають одну область пам’яті. Для їх декларації використовується ключове слово “static”. Вони використовуються безпосередньо всередині класу. Приклад використання показано на рисунку 2.42, разом із зв’язком декларованих змінних із пам’ятю купи:

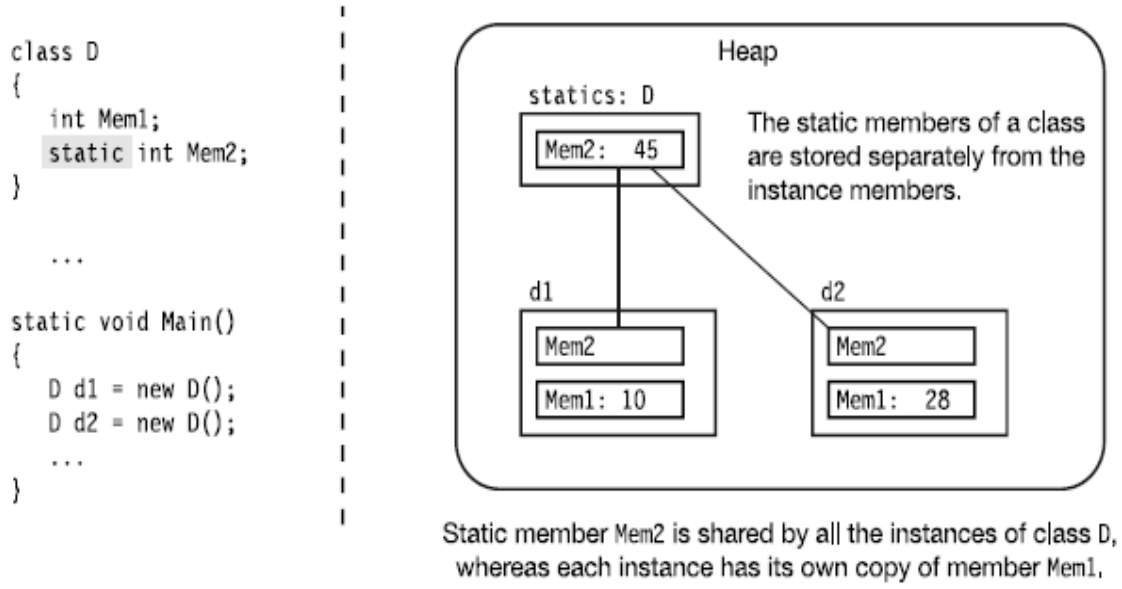


Рисунок 2.42 – Статичні поля

Життєвий цикл статичних членів класу довготривалий і незалежний від екземпляру класу. На відміну від членів екземпляру класу, статичні члени

існують навіть тоді, коли не було створено жодного екземпляру класу (рис. 2.43).

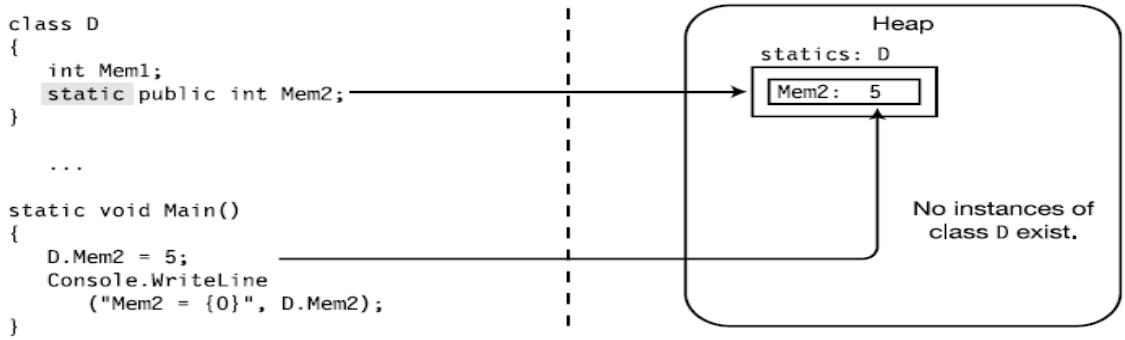


Рисунок 2.43 – Життєвий цикл статичних членів класу

10.Статичні методи

Класи можуть мати також і статичні методи, які можна викликати не створюючи екземпляру класу;

Статичні методи можуть мати доступ лише до статичних полів і не мають доступу до членів екземпляру класу. Використання і схема виклику статичних методів показана на рисунку 2.44:

```
class X
{
   static public int A;                              // Static field
   static public void PrintValA()                    // Static method
   {
      Console.WriteLine("Value of A: {0}", A);
   }                                          ↑
}                                  Accessing the static field
```

```
class Program
{
   static void Main()
   {
      X.A = 10;              // Use dot-syntax notation
      X.PrintValA();         // Use dot-syntax notation
   } ↑
} Class name
```

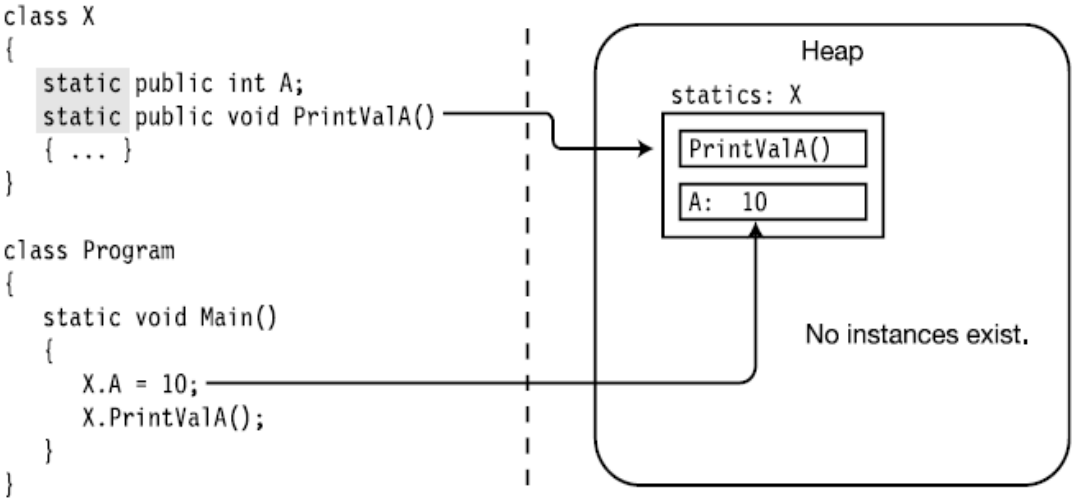


Рисунок 2.44 – Статичні методи, їх схем виклику

11.Інші статичні члени класу

Не всі члени класу можуть бути статичними. До таких, що дійсно можуть реалізовувати статичні операції, відносяться поля, методи, конструктори, оператори та події всередині класу. Статичні методи викликають, не створюючи екземпляр класу. Звернення можливе лише до статичних полів класу.

Статичні члени класу потрібні для оптимального використання ресурсів. Вони зберігаються в одному місці і посилаються на об’єкти. До них можна отримати доступ без створення екземпляру класу. Статичні члени класу є саме такими (рис. 2.45):

Data Members (Store Data)	Function Members (Execute Code)	
✓ Fields Constants	✓ Methods ✓ Properties ✓ Constructors Finalizers	✓ Operators Indexers ✓ Events

Рисунок 2.45 – Статичні члени класу

12. Константи

Характеристика констант:

- константа містить значення, яке не може бути змінене при виконанні програми;
- при декларації константи обов’язковою є її ініціалізація початковим значенням;
- видимість констант відповідає статичним змінним;
- константи можуть бути членами класу, а можуть бути задекларовані локально.

Константа декларується за допомогою ключового слова const. Приклад константи та присвоєння певних значень показано далі (рис. 2.46):

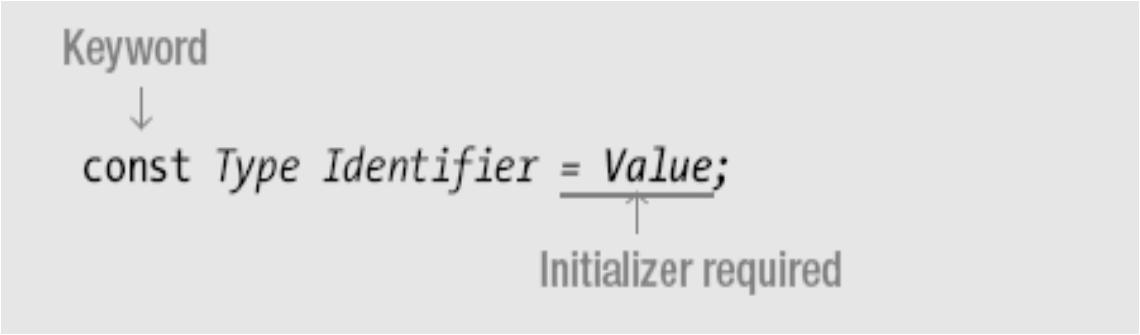


Рисунок 2.46 – Позначення константи

13. Властивості

Властивості – це члени класу, які забезпечують доступ до полів. За своєю поведінкою вони схожі до полів, однак, на відміну від них, вони самі не зберігають даних, а в процесі звертання до них виконується програмний код.

Формально властивість є іменованим набором двох методів, які мають назву аксцессори:

- аксцессор set призначений для запису значення до властивості;
- аксцессор get призначений для зчитування значення з властивості.

Особливостями властивостей є те, що вони дані не зберігають. Аксцессори дозволяють записувати дані до властивості та зчитувати їх. Також властивість може бути реалізована без 1 із методів. Приклад декларації властивості (рис. 2.47):

```
int MyValue
{
   set
   {
      SetAccessorCode
   }
   get
   {
      GetAccessorCode
   }
}
```

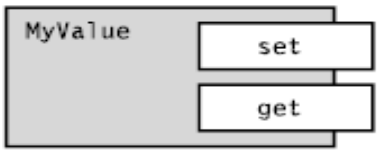


Рисунок 2.47 – Декларація властивості

Властивість та її аксцессори повинні декларуватися з врахуванням певних вимог. При створенні властивості необхідно зазначати її тип, тобто тип даних, які вона міститиме і на які посилатиметься до поля, і безпосередньо назву властивості.

Аксцессор set повинен відповідати вимогам:

– не повинен мати явних параметрів, має єдиний неявний параметр за значенням з назвою value такого ж типу, як і властивість;
– не повинен повертати значення (бути void);

Аксцессор get повинен відповідати вимогам:

– не приймати параметрів;
– тип значення, яке повертається, має відповідати типу властивості.

Приклад декларації властивості та її тіла показано на рисунку 2.48:

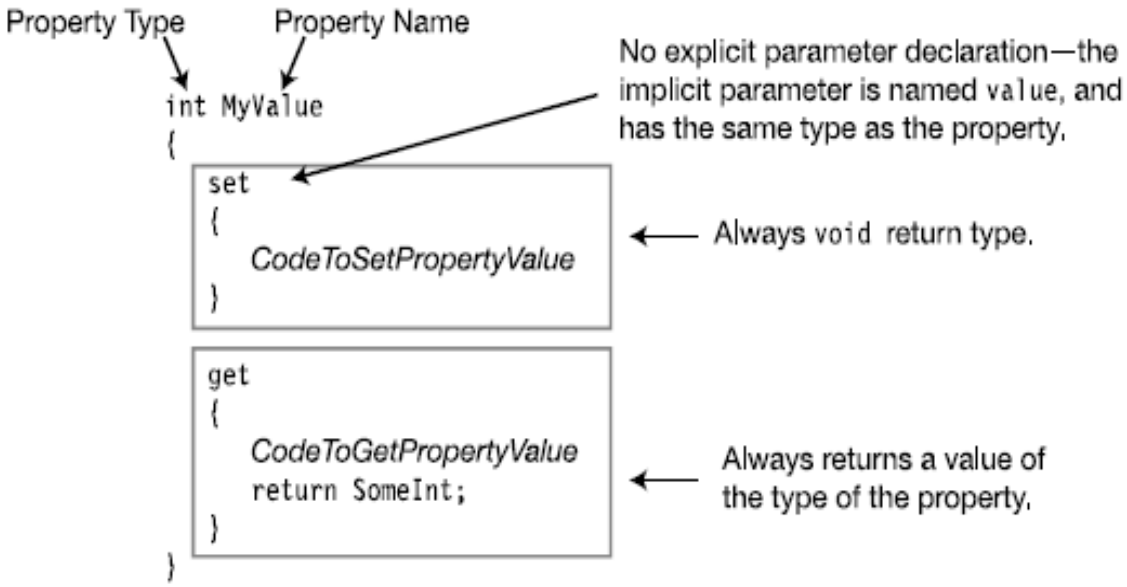


Рисунок 2.48 – Декларація властивості та аксцессорів

Відповідно приклад створення коду властивості на мові програмування C# та демонстрація взаємодії з полем, що містить саме значення, і до якого звертається властивість, показано на рисунку 2.49:

```
class C1
{
   private int TheRealValue;              // Field: memory allocated

   public int MyValue                     // Property: no memory allocated
   {
      set
      {
         TheRealValue = value;
      }

      get
      {
         return TheRealValue;
      }
   }
}
```

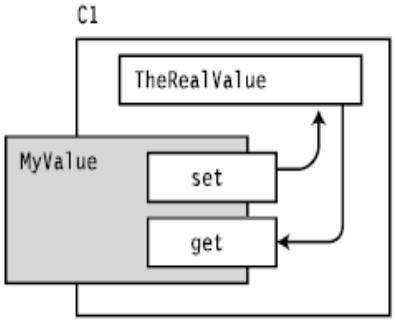


Рисунок 2.49 – Приклад створення властивості

Після створення до властивості можна звертатися, щоб дізнатися її значення. Найчастіше у певному класі поля роблять приватними, а властивості публічними. Це забезпечує збереження даних, доступ лише до властивості, з неможливою зміною зміною значення поля. Саме значення властивості можна задавати у процесі програми і звертатися до нього. Саме це забезпечують аксцессори.

При ініціалізації властивості певним значенням викликається метод set, а за звернення до значення властивості відповідає метод get (рис. 2.50).

```
int MyValue             // Property declaration
{
   set{ ... }
   get{ ... }
}
...
Property name
    ↓
MyValue = 5;            // Assignment: the set method is implicitly called
z = MyValue;            // Expression: the get method is implicitly called
        ↑
    Property name
```

Рисунок 2.50 – Приклад використання властивості

Властивість використовується подібно до поля, при цьому аксцессори викликаються неявно. Явний виклик аксцессорів не допускається. Тобто безпосереднє звернення до get і set за межами властивості не допустиме і неодмінно викличе помилку при компіляції (рис. 2.51).

```
y = MyValue.get();      // Wrong! Can't explicitly call get accessor.
MyValue.set(5);         // Wrong! Can't explicitly call set accessor.
```

Рисунок 2.51 – Неправильне використання властивості

Властивості асоціюються з полями. Вони реалізують доступ до приватних полів через екземпляри цього класу у інших. Також їм можна надавати певне значення. Необхідно створити екземпляр класу, у якому задекларована та ініціалізована властивість. Вона задекларована як видима у всіх нащадках та незалежних класах, тому можна звертатися до неї за допомогою екземпляру даного класу. Приклад роботи з властивістю, виклик її значення та присвоєння нового показано на рисунку 2.52. Реалізація відбувається на основі 2 незалежних класів за допомогою створення екземпляру класу C1.

```
class C1
{
   private int TheRealValue = 10;   // Field: memory allocated
   public  int MyValue              // Property: no memory allocated
   {
      set{ TheRealValue = value; }  // Sets the value of field TheRealValue
      get{ return TheRealValue; }   // Gets the value of the field
   }
}

class Program
{
   static void Main()
   {
                                    Read from the property as if it were a field
      C1 c = new C1();                          ↓
      Console.WriteLine("MyValue:  {0}", c.MyValue);

      c.MyValue = 20; ←Use assignment to set value of property
      Console.WriteLine("MyValue:  {0}", c.MyValue);
   }
}
```

Рисунок 2.52 – Реалізація властивості

У коді програми необхідно розрізняти поля та властивості. З метою цього була створена домовленість у найменуванні цих членів класу з різними початковими символами. Назва поля починається з малої літери, а властивості – з великої або знаку підкреслення перед назвою (рис. 2.53).

```
private int firstField;                  // Camel casing
public  int FirstField                   // Pascal casing
{
   get { return firstField; } set { firstField = value; }
}

private int _SecondField;                // Underscore
public  int SecondField
{
   get { return _SecondField; } set { _SecondField = value; }
}
```

Рисунок 2.53 – Іменування властивостей і полів

Властивості можуть використовуватися для здійснення обчислень перед присвоюванням значення. Для цього передбачена спеціальна система знаків. Програма проводить обчислення у властивості і серед деяких альтернативних варіантів обирає необхідний та присвоює це значення. Приклад такого використання властивості показано на рисунку 2.54:

```
int TheRealValue = 10;                    // The field
int MyValue                               // The property
{
   set {                                  // Sets the value of the field
      TheRealValue = value > 100          // But makes sure it's not > 100
                     ? 100
                     : value;
   }
   get {                                  // Gets the value of the field
      return TheRealValue;
   }
}
```

Рисунок 2.54 – Обчислення перед присвоюванням

Властивості можуть мати лише 1 аксцессор. В даному випадку відповідно за відсутності set її називають “тільки на читання”, а за наявності лише set - “тільки на запис”. Приклад властивості лише на читання показано на рисунку 2.55:

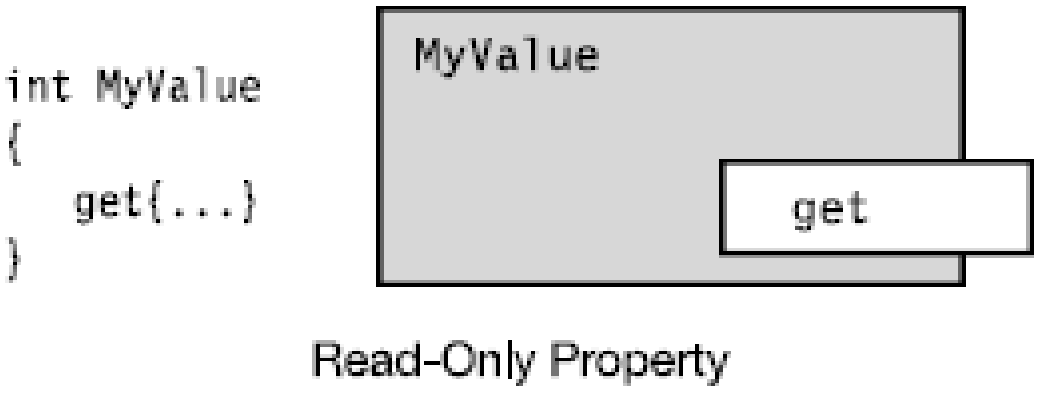


Рисунок 2.55 – Властивість «тільки на читання»

Властивість може бути обчислювана, тобто містити всередині певні математичні операції. В такому випадку її значення встановлюється при зміні величини поля чи проведення операцій над декількома полями. На рисунку 2.56 у класі задекларована властивість «тільки на читання», що містить значення гіпотенузи прямокутного трикутника після обчислення його за теоремою Піфагора, звертаючись до полів, що містять значення катетів.

```
class RightTriangle
{
   public double A = 3;
   public double B = 4;
   public double Hypotenuse                    // Read-only property
   {
      get{ return Math.Sqrt((A*A)+(B*B)); }    // Calculate return value
   }
}

class Program
{
   static void Main()
   {
      RightTriangle c = new RightTriangle();
      Console.WriteLine("Hypotenuse:  {0}", c.Hypotenuse);
   }
}
```

Рисунок 2.56 – Обчислювальна властивість

Також властивості можуть зберігати значення у базах даних. Для цього необхідно у аксцессорах організувати безпосереднє звернення до елементів бази даних. За допомогою аксцессора set властивості присвоюється певне значення комірки таблиці, а аксцессор get дозволяє доступ на читання до цього значення. Реалізовується властивість для зберігання значень у базі даних. У загальному випадку приклад використання властивості таким чином показано на рисунку 2.57:

```
int MyDatabaseValue
{
   set                              // Sets integer value in the database
   {
      SetValueInDatabase(value);
   }
   get                              // Gets integer value from the database
   {
      return GetValueFromDatabase();
   }
}
```

Рисунок 2.57 – Зберігання значень у базі даних з використанням властивості

Властивості можуть бути задекларовані як статичні. Аксцессори таких властивостей, як і всі статичні члени класу, не потребують створення екземпляру класу, можуть бути доступними через ім’я класу. Наприклад, наступний код показує клас із статичною властивістю MyValue, яка пов’язана з статичним полем, задекларованим раніше. Властивість у наступному класі доступна безпосередньо через його ім’я (рис. 2.58).

```
class Trivial
{
   static int myValue;
   public static int MyValue
   {
      set { myValue = value; }
      get { return myValue;  }
   }

   public void PrintValue()                    Accessed from inside the class
   {                                                      ↓
      Console.WriteLine("Value from inside: {0}", MyValue);
   }
}

class Program
{
   static void Main()                          Accessed from outside the class
   {                                                      ↓
      Console.WriteLine("Init Value: {0}", Trivial.MyValue);
      Trivial.MyValue = 10;    ← Accessed from outside the class
      Console.WriteLine("New Value : {0}", Trivial.MyValue);

      Trivial tr = new Trivial();
      tr.PrintValue();
   }
}
```

Рисунок 2.58 – Статичні властивості

14. Конструктори об’єктів

Конструктор призначений для ініціалізації стану об’єкту.

Для того, щоб була можливість створювати екземпляри класу із інших класів необхідно декларувати конструктор як public.

Ім’я конструктора має відповідати імені класу.

Конструктор не може повертати значення.

Конструктори можуть приймати параметри.

Клас може мати декілька конструкторів (можуть перевантажуватися) (рис. 2.59).

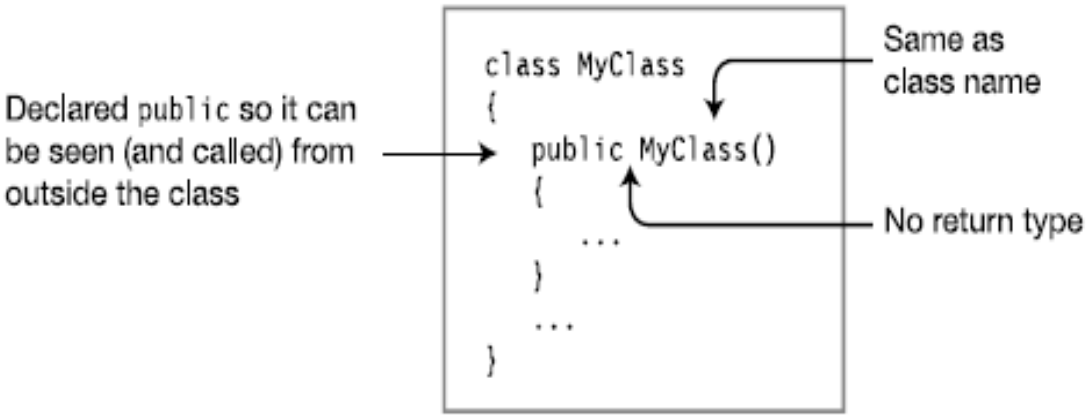


Рисунок 2.59 – Декларація конструктора

Конструктори можуть бути створені у кожному класі з різною сигнатурою, тобто перевантажуватися. Всередині них можна проводити певні обчислення, а також ініціалізацію полів. І тоді при створенні екземпляру класу потрібно в дужках перечислювати необхідні значення, які потім будуть присвоюватися полям у конструкторі і виконувати певні операції. До методу можна звертатися різним екземплярам, адже при створенні перевантаженого конструктора маємо можливість створювати різні екземпляри класу із різними значеннями у дужках. Це демонструє приклад використання конструкторів на рисунку 2.60:

```
class Class1
{
   int    MyNumber;
   string MyName;

   public Class1()            { MyNumber=28; MyName="Nemo"; } // Constructor 0
   public Class1(int Value){ MyNumber=Value; MyName="Nemo"; } // Constructor 1
   public Class1(String Name) { MyName=Name; }                // Constructor 2

   public void SoundOff()
      {Console.WriteLine("MyName {0}, MyNumber {1}", MyName, MyNumber); }
}

class Program
{
   static void Main()
   {
      Class1 a = new Class1(),                    // Call constructor 0.
             b = new Class1(7),                   // Call constructor 1.
             c = new Class1("Bill");              // Call constructor 2.

      a.SoundOff();
      b.SoundOff();
      c.SoundOff();
   }
}
```

Рисунок 2.60 – Приклад використання конструкторів

Клас у будь-якому випадку міститиме конструктор. Такі члени класів називаються конструкторами за замовчуванням. Якщо у класу не задано конструктор, то компілятор автоматично генерує неявний конструктор за замовчуванням, що не приймає жодних параметрів. Тому при створенні екземпляру класу все одно звернення надходить до його конструктора, при чому в дужках не вказується жодне значення.

Якщо у класу є хоча б один явний конструктор, то у такому разі конструктор за замовчуванням не генерується і у разі створення класу без параметрів генерується помилка (рис. 2.61).

```
class Class2
{
   public Class2(int Value)    { ... }   // Constructor 0
   public Class2(String Value) { ... }   // Constructor 1
}

class Program
{
   static void Main()
   {
      Class2 a = new Class2();   // Error! No constructor with 0 parameters
      ...
   }
}
```

Рисунок 2.61 – Помилка використання конструктора за замовчуванням

15. Статичні конструктори

Конструктори можуть бути задекларовані як статичні – вони ініціалізують дані на рівні класу.

Статичний конструктор декларується за допомогою слова static.

Статичний конструктор може бути лише один у класу і не повинен приймати параметрів.

Статичний конструктор не має доступу до даних рівня об’єкту, також не може декларуватися з модифікаторами доступу.

Клас може мати як статичний конструктор, так і конструктори екземпляру класу.

Статичні конструктори не викликаються створюваною програмою, вони викликаються автоматично системою виконання перед створенням екземпляру класу чи перед спробою доступу до будь-якого статичного члену класу.

Приклад створення статичного конструктора всередині класу показано на рисунку 2.62:

```
class Class1
{
   static Class1 ()
   {
      ...                    // Do all the static initializations.
   }
   ...
```

Рисунок 2.62 – Статичний конструктор

16. Фіналізатори.

Ще одним членом класу є фіналізатори. Важливо знати, що без потреби його використовувати не варто. Це може підвищити ресурсозатратність програми. Фіналізатор повинен звертатися до ресурсів об’єкту, а не до інших об’єктів. Його декларацію бажано робити в межах класу, а не публічною.

Фіналізатори (деструктори) виконують дії, необхідні для вивільнення зайнятих ресурсів перед тим, як клас буде знищено.

Для класу може бути створений лише один фіналізатор, який не приймає параметрів і не може мати модифікаторів доступу.

Має ім’я таке ж як у класу, проте перед ім’ям ставиться тільда (~)

Не може бути викликаний явно програмним кодом, його викликає система збору сміття у той момент, коли встановлено, що закінчилися звернення до об’єкту (рис. 2.63).

```
Class1
{
   ~Class1()                                          // The finalizer
   {
      CleanupCode
   }
   ...
}
```

Рисунок 2.63 – Створення фіналізатора

17.Порівняння конструкторів і фіналізаторів

Див. табл. 2.2:

Таблиця 2.2 – Порівняння конструкторів та фіналізаторів

		Коли і як часто викликається
Екземпляр	Конструктор	Викликається один раз при кожному створенні нового об’єкту
	Фіналізатор	Викликається один раз при вивільненні пам’яті кожним об’єктом
Клас	Конструктор	Викликається лише один раз за весь час виконання програми, в залежності від того, яка подія відбудеться першою: чи буде створено екземпляр класу, чи буде звернення до статочного члену класу
	Фіналізатор	Не існує на рівні класу

ЛК.03 – МЕХАНІЗМ НАСЛІДУВАННЯ В ОБ’ЄКТНО-ОРІЄНТОВАНОМУ ПРОГРАМУВАННІ

Перелік питань:

1. Модифікатор readonly.
2. Ключове слово this.
3. Індексатори.
4. Модифікатори доступу на аксцессорах.
5. Часткові класи.
6. Наслідування класів.
7. Доступ до наслідуваних методів.
8. Приховування методів базового класу.
9. Доступ до базового класу.
10. Використання посилань на базовий клас.
11. Віртуальні методи.
12. Виконання конструктора.
13. Модифікатори доступу до класу.
14. Наслідування між збірками.
15. Модифікатори доступу до членів класу.

На самостійне вивчення:

1. Делегати [1, С.323-361].

1. Модифікатор readonly

Поля можна позначати модифікатором readonly (тільки для читання). На відміну від константи, поле “тільки для читання” може бути ініціалізоване у конструкторі, не обов’язково лише в момент декларації. Поля readonly можуть бути як статичними, так і належати до екземпляру класу.

Цей модифікатор зручно використовувати, коли потрібно створити незмінне поле, початкове значення якого буде відоме лише в середовищі

виконання. Поля, доступні лише для читання, дозволяють створювати елементи даних, значення яких залишаються невідомими у процесі трансляції, але про які відомо, що вони ніколи не будуть змінюватися після їх створення. Приклад використання поля readonly показано на рисунку 3.1:

```
class Shape
{  Keyword              Initialized
      ↓                    ↓
   readonly double PI = 3.1416;
   readonly int    NumberOfSides;
      ↑                    ↑
    Keyword          Not initialized

   public Shape(double side1, double side2)                  // Constructor
   {
      // Shape is a rectangle
      NumberOfSides = 4;
            ↑
      ... Set in constructor
   }

   public Shape(double side1, double side2, double side3)    // Constructor
   {
      // Shape is a triangle
      NumberOfSides = 3;
            ↑
      ... Set in constructor
   }
}
```

Рисунок 3.1 – Використання поля readonly

Поля, доступні тільки для читання, відрізняються від констант також тим, що таким полям можна присвоювати значення в контексті конструктора.

2. Ключове слово this

Ключове слово this означає екземпляр класу і може використовуватися лише в наступних блоках:

- конструкторах екземпляру класу;
- методах екземпляру класу;
- аксцессорах властовостей екземпляру класу та індексаторах.

Не може використовуватися у статичних методах класу. Основна ціль використання цього ключового слова – розрізняти імена параметрів методу і назви членів класу. На відміну від інших аргументів, поточний об’єкт до списку аргументів функції не потрапляє, а значить, програміст не назначає йому ніякого імені. Приклад використання ключового слова this показано на рисунку 3.2:

```
class MyClass
{
   int Var1 = 10;
        ↑    Both are called "Var1"   ↓
   public int ReturnMaxSum(int Var1)
   {        Parameter   Field
              ↓          ↓
      return Var1 > this.Var1
                  ? Var1                  // Parameter
                  : this.Var1;            // Field
   }
}

class Program
{
   static void Main()
   {
      MyClass mc = new MyClass();
      Console.WriteLine("Max: {0}", mc.ReturnMaxSum(30));
      Console.WriteLine("Max: {0}", mc.ReturnMaxSum(5));
   }
}
```

Рисунок 3.2 – Використання this

3. Індексатори

Індексатори створюються для того, щоб маніпулювати із полями екземпляру класу подібно до масиву. Мова C# надає змогу будувати користувацькі класи і структури, які можуть індексуватися подібно стандартним масивам. Індексатори забезпечують лише функціональність індексування. І користувацькі типи колекції краще відповідають структурі бібліотек базових класів (рис. 3.3).

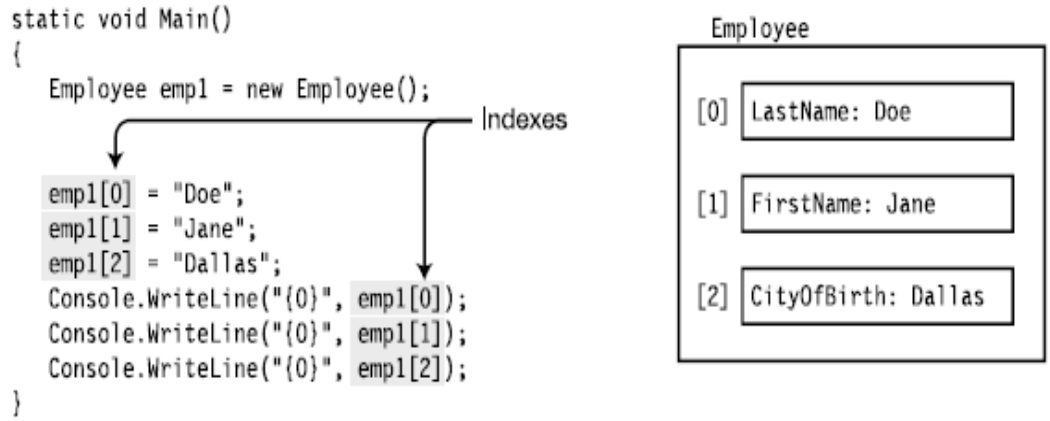


Рисунок 3.3 – Використання індексаторів

Індексатор фактично представляє собою набір get і set аксцессорів, схожих до таких, які використовуються у властивостях (рис. 3.4).

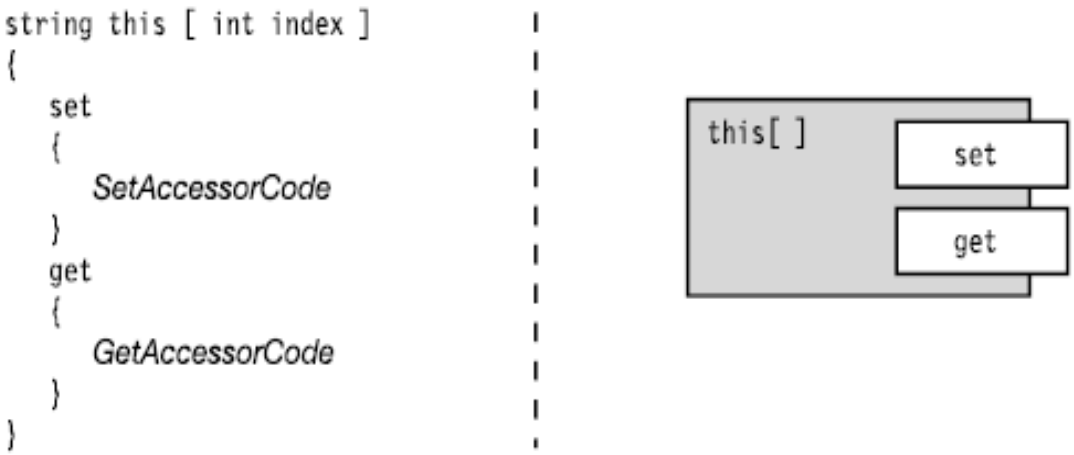


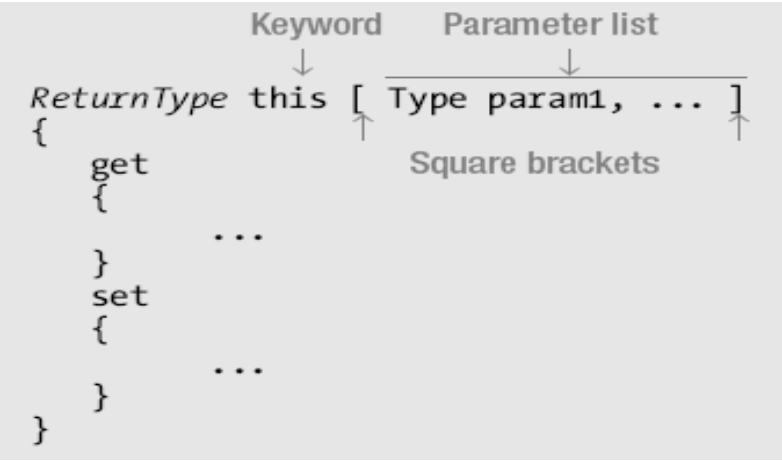
Рисунок 3.4 – Внутрішній вигляд індексатора

Можна порівняти індексатори та властивості у їх спільних та відмінних моментах. Індексатори і властивості схожі у багатьох моментах:

– як і для властивості, для індексатора не виділяється пам’ять для зберігання даних;
– як і властивості, індексатори використовуються переважно для того, щоб надавати доступ до інших членів даних, з якими вони пов’язані.

Проте існують принципові відмінності. Властивості асоціюються, як правило, із одним елементом, що зберігає дані, а індексатори – із багатьма. Індексатори не можуть бути статичними і можуть мати лише один аксцессор. У якості індексу можуть використовуватися будь-які типи даних – не обов’язково лише числа. Також індексатори можуть перевантажуватися.

Декларація індексатору включає зазначення типу для повернення значення, ключового слова this та у квадратних дужках перечислення необхідних параметрів із їх типами. Теоретичний зразок показує рисунок 3.5:



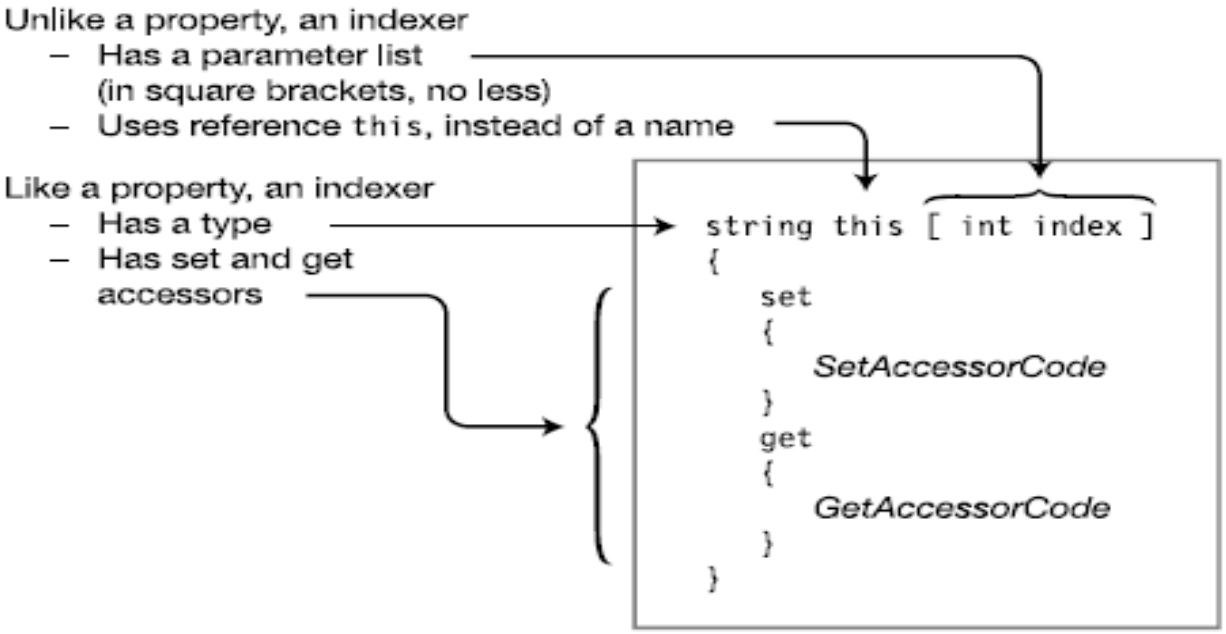
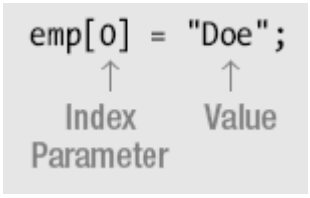


Рисунок 3.5 – Декларація індексатору

Код в аксцессорі set повинен перевіряти параметри індексу, визначати, де необхідно зберігати дані, і потім зберігати їх. Його синтаксис та значення показано на рисунку 3.6. Ліва частина саме показує синтаксис аксцессору, права – його значення та сигнатуру. Тобто він має тип повернення значень void, також використовує такий самий список параметрів як і при декларації індексатору, містить значення за замовчуванням value.



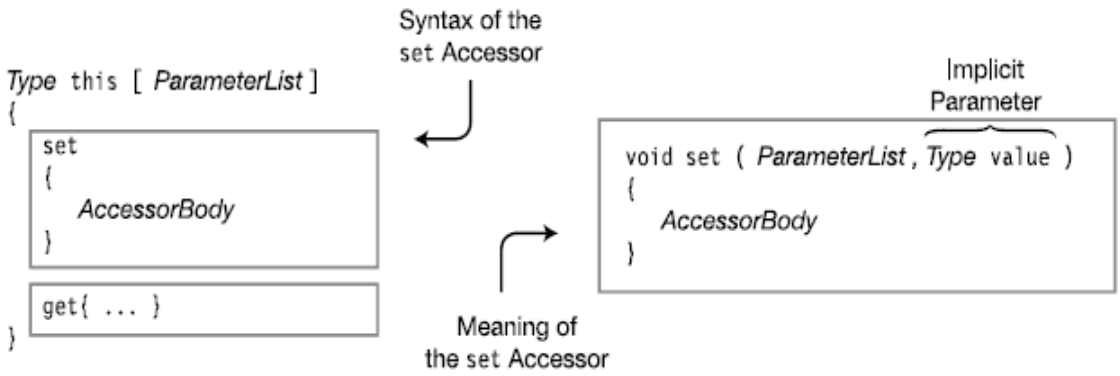


Рисунок 3.6 - Реалізація аксцессору set

Реалізацію аксцессора get аналогічним чином можна побачити на рисунках 3.7, 3.8. Головними відмінностями є відсутність значення value, але використовується ключове слово return для повернення значень.

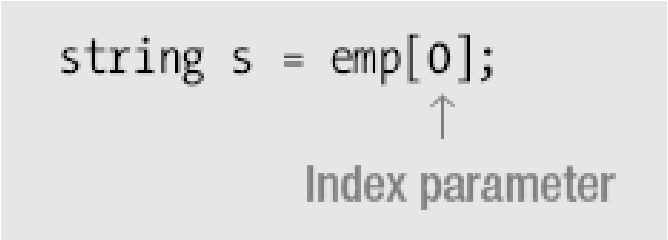


Рисунок 3.7 – Доступ до індексатору

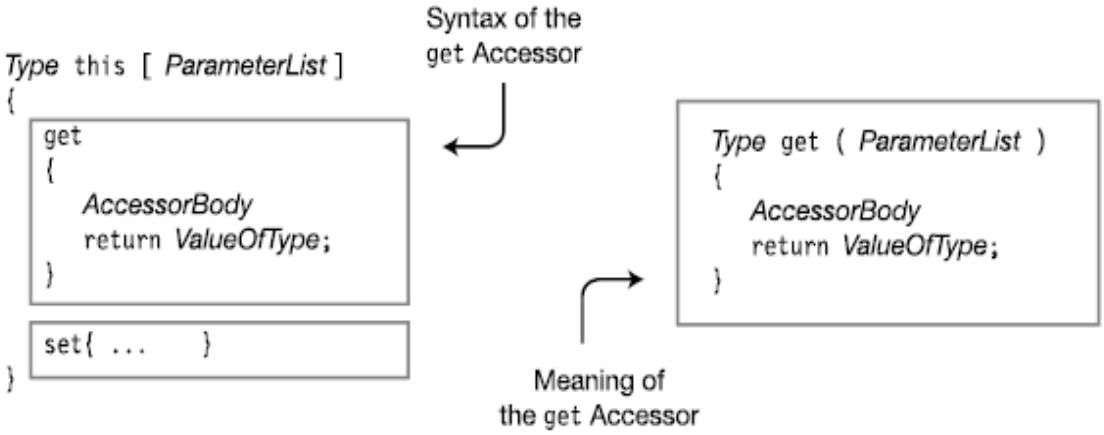


Рисунок 3.8 - Реалізація аксцессору get

Загальний приклад використання індексатору та заповнення його тіла показано на рисунку 3.9:

```
class Employee
{
   public string LastName;                          // Call this field 0.
   public string FirstName;                         // Call this field 1.
   public string CityOfBirth;                       // Call this field 2.

   public string this[int index]                    // Indexer declaration
   {
      set                                          // Set accessor declaration
      {
         switch (index)
         {
            case 0: LastName = value;
               break;
            case 1: FirstName = value;
               break;
            case 2: CityOfBirth = value;
               break;
         }
      }

      get                                          // Get accessor declaration
      {
         switch (index)
         {
            case 0: return LastName;
            case 1: return FirstName;
            case 2: return CityOfBirth;
            default:
               return "";
         }
      }
   }
}
```

Рисунок 3.9 – Використання індексатору

4. Модифікатори доступу на аксцессорах

За замовчуванням аксцессори мають такий же рівень доступу, як і властивість, яку вони реалізують. В окремих випадках можливо обмежувати доступ на рівні аксцессорів, однак слід керуватися ієрархією модифікаторів – доступ можна обмежувати лише від вищого до нижчого рівня (рис. 3.10).

Існують в даному випадку деякі обмеження. Аксцессор може мати модифікатор доступу лише в тому випадку, якщо члени класу (Властивості чи індексатори) мають обоє аксцессорів. Крім того модифікатор доступу може мати лише один із них.

```
class MyClass
{
   private string _Name = "John Doe";
   public string Name
   {
      get { return _Name; }
      protected set { _Name = value; }
   }
}
```

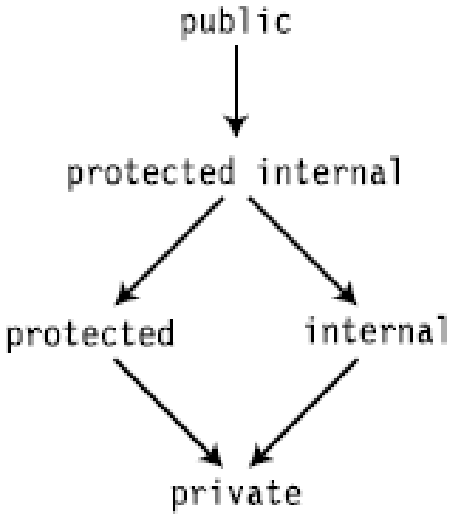


Рисунок 3.10 - Модифікатори доступу на аксцессорах, їх ієрархія

5. Часткові класи

Декларація класу може містити декілька окремих частин як в одному файлі, так і в різних файлах. Це є зручним при необхідності розбиття програми на окремі підчастини і розробку програмного коду цілої задачи окремо, незалежно. А потім в результаті обєднання часткових класів отримати повну програмну розробку. Приклад декларації та їх використання показано на рисунках 3.11, 3.12:

```
Type modifier
  ↓
partial class MyPartClass    // Same class name as following
{
   member1 declaration
   member2 declaration
      ...
}
Type modifier
  ↓
partial class MyPartClass    // Same class name as preceding
{
   member3 declaration
   member4 declaration
      ...
}
```

Рисунок 3.11 – Використання часткових класів

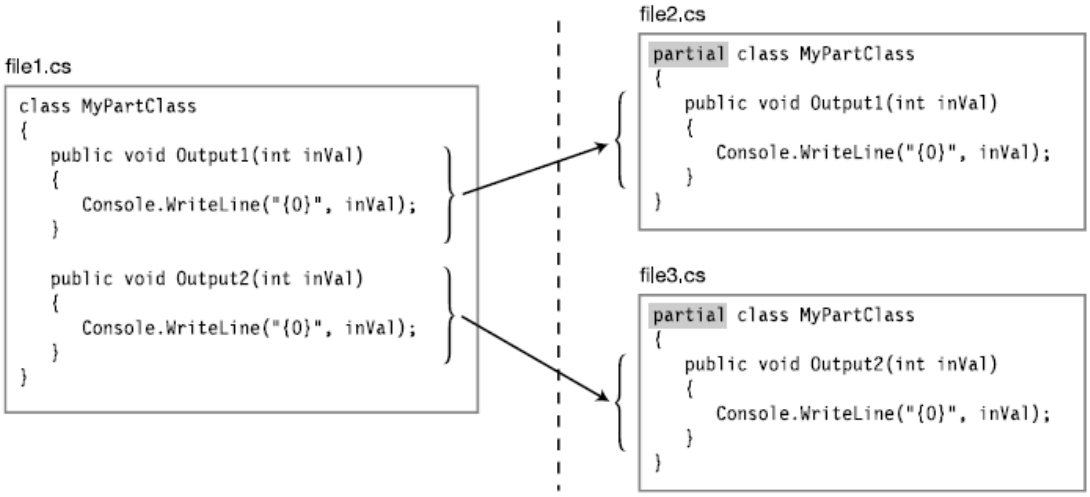


Рисунок 3.12 – Приклад використання часткових кдасів

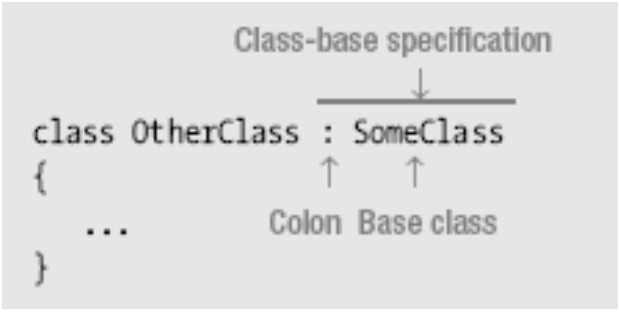
6. Наслідування класів

Можна використовувати існуючий клас як основу для створення нового класу. В такому разі існуючий клас називається базовим класом, а новий – наслідуваним класом.

Члени наслідуваного класу складаються з наступного:

– члени класу, які були задекларовані у ньому самому;
– члени базового класу.

Наслідуваний клас не може видаляти існуючі члени базового класу. Щоб задекларувати наслідуваний клас, потрібно додати специфікацію (двокрапку) після імені наслідуваного класу та вказати назву головного. Наслідуваний клас містить усю головну функціональність базового, а також додаткові операції у власному тілі (рис. 3.13).



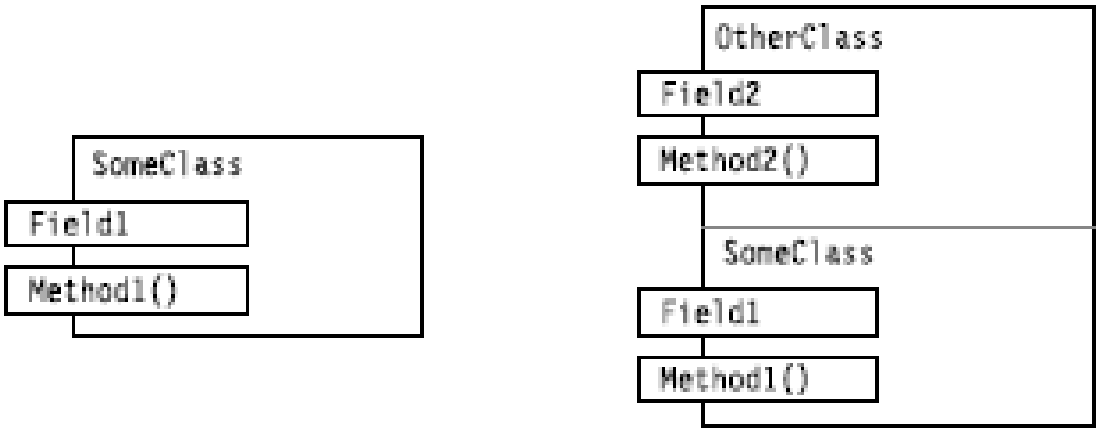


Рисунок 3.13 – Наслідування класів

7. Доступ до наслідуваних методів.

Приклад доступу показано на рисунку 3.14. Наслідуваний клас має доступ до власних методів, а також до неприватних методів головного класу.

```
class SomeClass {                          // Base class
   public string Field1 = "base class field";
   public void Method1( string value ) {
      Console.WriteLine("Base class -- Method1:     {0}", value);
   }
}

class OtherClass: SomeClass {              // Derived class
   public string Field2 = "derived class field";
   public void Method2( string value ) {
      Console.WriteLine("Derived class -- Method2:  {0}", value);
   }
}

class Program {
   static void Main() {
      OtherClass oc = new OtherClass();

      oc.Method1( oc.Field1 );         // Base method with base field
      oc.Method1( oc.Field2 );         // Base method with derived field
      oc.Method2( oc.Field1 );         // Derived method with base field
      oc.Method2( oc.Field2 );         // Derived method with derived field
   }
}
```

Рисунок 3.14 – Доступ до наслідуваних методів

В результаті отримаємо такі дані на екрані (рис. 3.15):

```
Base class -- Method1:     base class field
Base class -- Method1:     derived class field
Derived class -- Method2:  base class field
Derived class -- Method2:  derived class field
```

Рисунок 3.15 – Результат виконання програми

Всі класи за виключенням спеціального класу object є наслідуваними, навіть якщо вони не мають специфікації приналежності до певного головного класу. Вони напряму наслідуються від вказаного головного класу. А

відсутність наслідування розроблена для спрощення коду за замовчуванням. Клас може наслідуватися лише від головного класу. Ієрархія наслідування класів покроково може бути громіздкою і великою. Ці положення демонструють рисунки 3.16, 3.17:

Рисунок 3.16 - Всі класи наслідувані від класу “object”

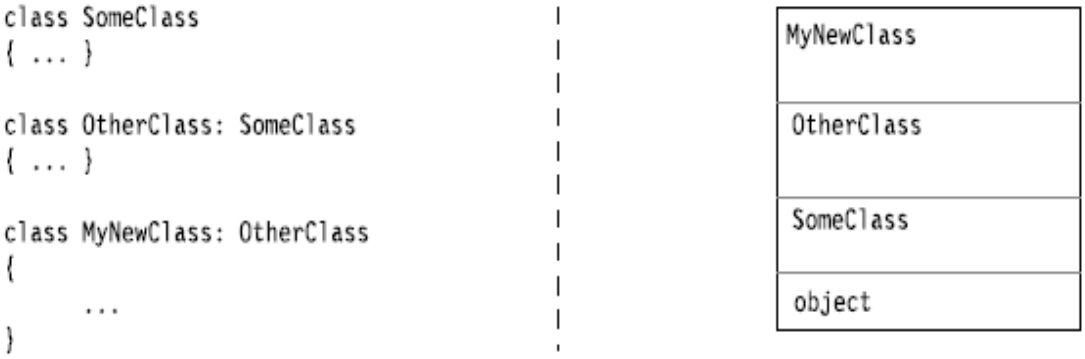


Рисунок 3.17 - Ієрархія класів

8. Приховування членів базового класу

Мова програмування надає можливість приховувати члени базового класу у наслідуваних. Але наслідуваний клас не може видалити члени та значення, які він наслідує. Їх якраз можна приховати чи замаскувати через декларацію нового метода чи поля (чи іншого члену класу) з ідентичною сигнатурою та назвою. Для цього використовується модифікатор new. Без нього програма скомпілюється вдало, але на екран виведеться спостереження про приховування. Приклад приховування показано на рисунках 3.18, 3.19:

```
class SomeClass                          // Base class
{
       string Field1;
   ...
}

class OtherClass : SomeClass            // Derived class
{
   new string Field1;                   // Mask base member with same name.
    ↑
 Keyword
```

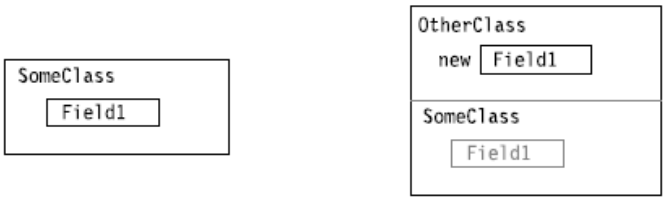


Рисунок 3.18 – Приховування членів базового класу

```
class SomeClass                                              // Base class
{
   public string Field1 = "SomeClass Field1";
   public void   Method1(string value)
       { Console.WriteLine("SomeClass.Method1:  {0}", value); }
}


class OtherClass : SomeClass                              // Derived class
{ Keyword
    ↓
   new public string Field1 = "OtherClass Field1";    // Mask the base member.
   new public void   Method1(string value)            // Mask the base member.
    ↑    { Console.WriteLine("OtherClass.Method1:  {0}", value); }
} Keyword


class Program
{
   static void Main()
   {
      OtherClass oc = new OtherClass();       // Use the masking member.
      oc.Method1(oc.Field1);                  // Use the masking member.
   }
}
```

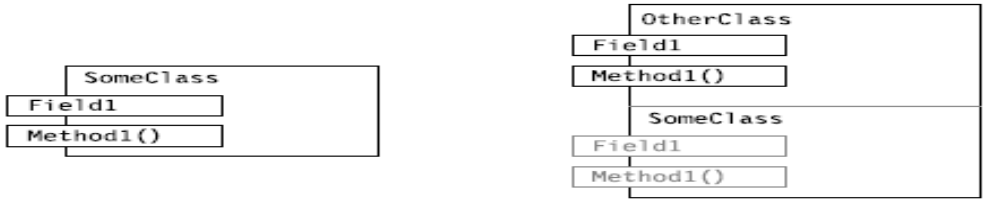


Рисунок 3.19 – Приклад реалізації

9. Доступ до базового класу

Іноді виникає необхідність звернутися до члену базового класу, який був прихований у наслідуваному класі. Для цього потрібно вказати ключове слово base, і після крапки безпосередньо звернутися до членів базового класу (навіть якщо вони є прихованими). Теоретичні основи показує рисунок 3.20, а приклад використання – рисунок 3.21:

```
Console.WriteLine("{0}", base.Field1);
                         ↑
                    Base access
```

Рисунок 3.20 – Доступ до базового класу

```
class SomeClass {                                          // Base class
   public string Field1 = "Field1 -- In the base class";
}

class OtherClass : SomeClass {                             // Derived class

   new public string Field1 = "Field1 -- In the derived class";
    ↑                  ↑
  Hides the field in the base class
   public void PrintField1()
   {
      Console.WriteLine("{0}", Field1);        // Access the derived class.
      Console.WriteLine("{0}", base.Field1);   // Access the base class.
   }
}                                  ↑
                              Base access

class Program {
   static void Main()
   {
      OtherClass oc = new OtherClass();
      oc.PrintField1();
   }
}
```

```
Field1 -- In the derived class
Field1 -- In the base class
```

Рисунок 3.21 – Приклад використання доступу

10.Використання посилань на базовий клас

У наслідуваному класі існує доступ до власних членів та додатково до членів отриманого (базового) класу. До них можна дістатися через посилання об’єкта. Наступні рисунки саме й показують цей процес через посилання об’єкта на частини базового класу (рис. 3.22, 3.23).

```
MyBaseClass mybc = (MyBaseClass) derived;
```

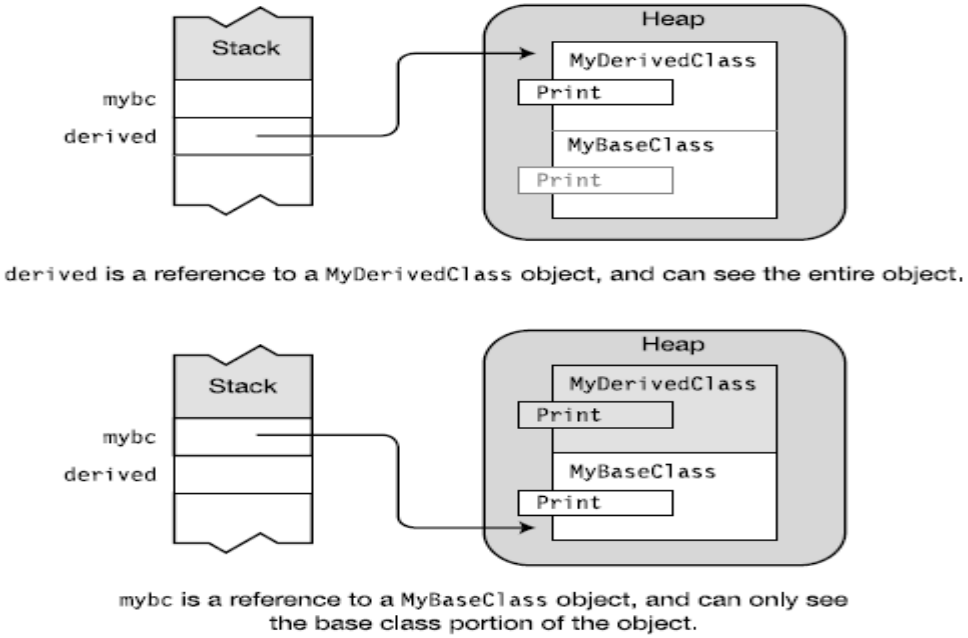


Рисунок 3.22 – Посилання на базовий клас

```
class MyBaseClass {
   public void Print() {
      Console.WriteLine("This is the base class.");
   }
}

class MyDerivedClass : MyBaseClass {
   new public void Print() {
      Console.WriteLine("This is the derived class.");
   }
}

class Program {
   static void Main() {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc = (MyBaseClass)derived;
                                ↑
                        Cast to base class
      derived.Print();           // Call Print from derived portion.
      mybc.Print();              // Call Print from base portion.
   }
}
```

Рисунок 3.23 – Використання посилань

Результатом отримаємо такі значення на рисунку 3.24. Також можна спостерігати взаємодію стеку з купою через посилання до значень відповідних необхідних членів класу.

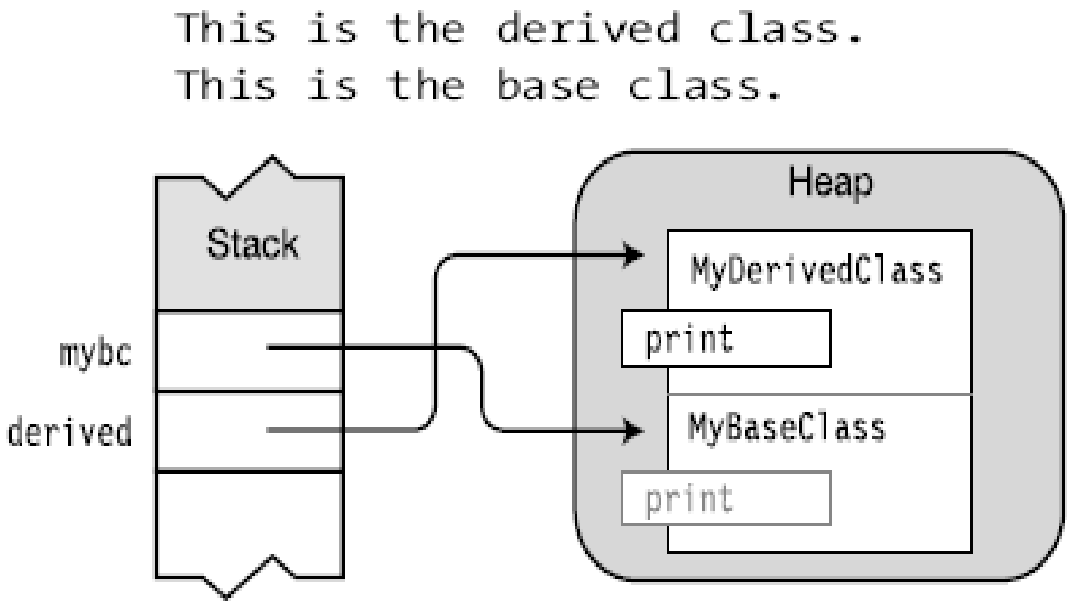


Рисунок 3.24 – Результат доступу до базового класу

11.Віртуальні методи

Віртуальні методи дозволяють базовому класу отримати доступ до членів наслідуваного класу. Поліморфізм забезпечує підкласам створити власну реалізацію методів, які визначені у базовому класі. Якщо у базовому класі потрібно визначити метод, який допускає пере визначення підкласом, то цей метод повинен бути віртуальним. У головному класі використовується при заданні методу ключове слово virtual. Щоб у підкласі перевизначити віртуальний метод, використовується ключове слово override (рис. 3.25, 3.26).

```
class MyBaseClass                                    // Base class
{
   virtual public void Print()
      ↑
 ...
class MyDerivedClass : MyBaseClass                   // Derived class
{
   override public void Print()
      ↑
```

Рисунок 3.25 – Використання віртуального методу

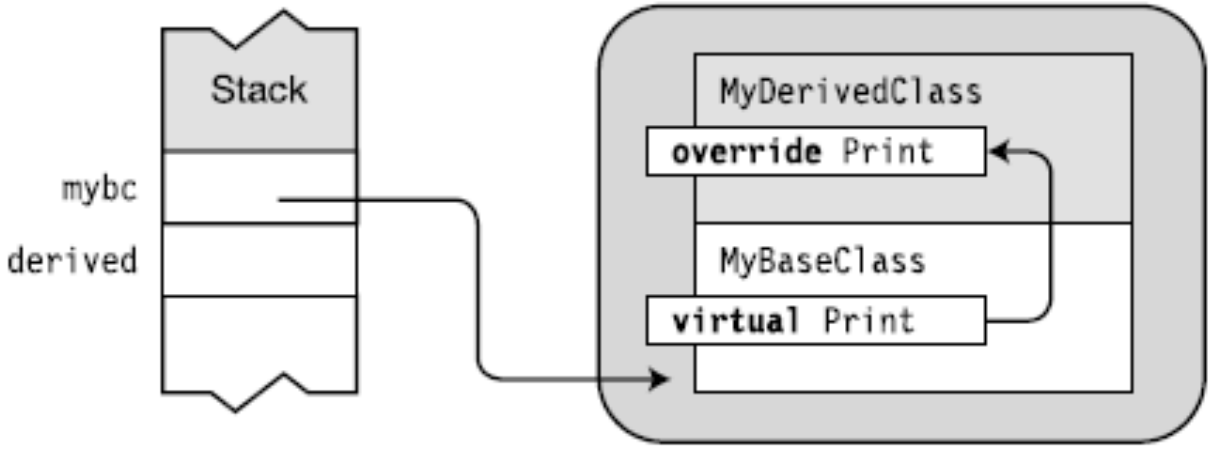


Рисунок 3.26 – Управління пам’яттю

В даному випадку реалізацію коду і відповідну зміну в результатах виводу можна проглянути на рисунку 3.27.

```
class MyBaseClass {
   virtual public void Print()
   {
      Console.WriteLine("This is the base class.");
   }
}

class MyDerivedClass : MyBaseClass {
   override public void Print()
   {
      Console.WriteLine("This is the derived class.");
   }
}

class Program {
   static void Main()
   {
      MyDerivedClass derived = new MyDerivedClass();
      MyBaseClass mybc = (MyBaseClass)derived;
                         ↑
      derived.Print();   Cast to base class
      mybc.Print();
   }
}
```

```
This is the derived class.
This is the derived class.
```

Рисунок 3.27 – Використання віртуальних методів і результат

При використанні віртуальних методів потрібно враховувати їх особливості. Віртуальні методи у базовому та наслідуваних класах повинні мати однакову видимість. Не можна перекривати (override) статичні методи, чи методи, які не були позначені ключовим словом virtual. Крім методів віртуальними можуть бути властивості, індексатори та події. Перекривання віртуальних методів можливе навіть на багатьох рівнях ієрархії (рис. 3.28).

```
class MyBaseClass                                        // Base class
{
   virtual public void Print()
   { Console.WriteLine("This is the base class."); }
}

class MyDerivedClass : MyBaseClass                       // Derived class
{
   override public void Print()
   { Console.WriteLine("This is the derived class."); }
}
```

```
class SecondDerived : MyDerivedClass {
   override public void Print() {
      ↑    Console.WriteLine("This is the second derived class.");
   }
}

class Program {
   static void Main()
   {
      SecondDerived derived = new SecondDerived(); // Use SecondDerived.
      MyBaseClass mybc = (MyBaseClass)derived;     // Use MyBaseClass.

      derived.Print();
      mybc.Print();
   }
}
```

```
This is the second derived class.
This is the second derived class.
```

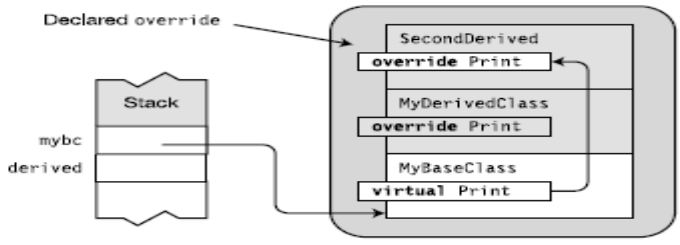


Рисунок 3.28 – Перекриття віртуальних методів на багатьох рівнях ієрархії

Віртуальний метод також може приховуватися. Для цього також використовується ключове слово new. Приклад програми, її результат та процес доступу до змінних у пам’яті показує рисунок 3.29.

```
class SecondDerived : MyDerivedClass {
   new public void Print() {
      Console.WriteLine("This is the second derived class.");
   }
}

class Program {
   static void Main()                                    // Main
   {
      SecondDerived derived = new SecondDerived();       // Use SecondDerived.
      MyBaseClass mybc = (MyBaseClass)derived;           // Use MyBaseClass.

      derived.Print();
      mybc.Print();
   }
}
```

```
This is the second derived class.
This is the derived class.
```

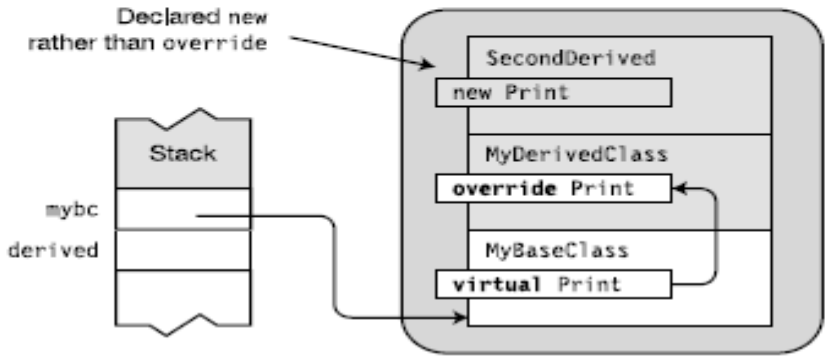


Рисунок 3.29 – Приховування віртуального методу

12.Виконання конструктора

Конструктор виконується при створенні класу, клас може мати декілька перевантажених конструкторів. Тобто це передбачає існування в класі декількох конструкторів з різною сигнатурою. Але для загального виконання конструктора повинні відбутися певні кроки. Це включає ініціалізацію членів екземпляру класу. Потім через них відбувається виклик конструкторів базового класу і безпосереднє виконання тіла конструктору

екземпляру класу. Створення конструктору наслідуваного класу та теоретичні етапи його виконання демонструє рисунок 3.30:

```
class MyDerivedClass : MyBaseClass
{
   MyDerivedClass()       // Constructor uses base constructor MyBaseClass().
   {
      ...
   }
```

Рисунок 3.30 – Конструктор наслідуваного класу

Так як реалізоване посилання на базовий клас, безпосередньо можна звертатися також і до конструкторів базового класу. При цьому при створенні конструктора наслідуваного класу вказується тип доступу, назва конструктору, перечислюються вхідні параметри, вказуються двокрапка та ключове слово base з передачею відповідних параметрів у дужках (рис. 3.31).

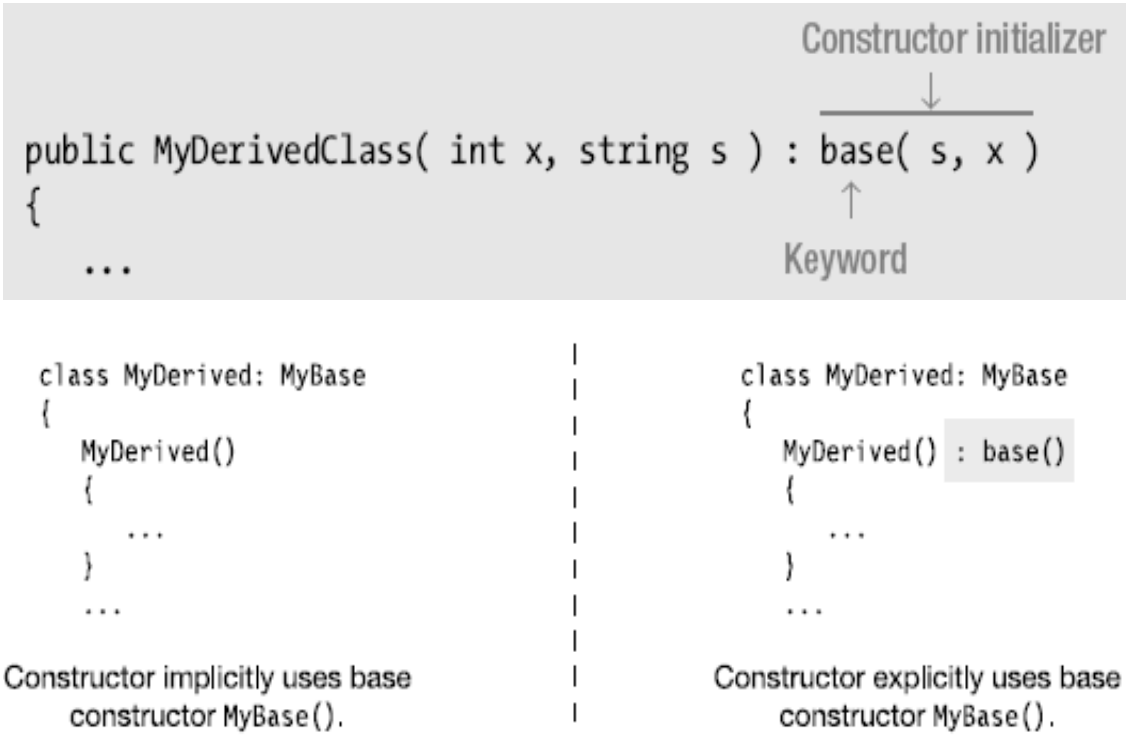


Рисунок 3.31 – Посилання на конструктор базового класу

Також можливе посилання на інший конструктор даного екземпляру класу, використовуючи ключове слово this з передачею параметрів (рис. 3.32).

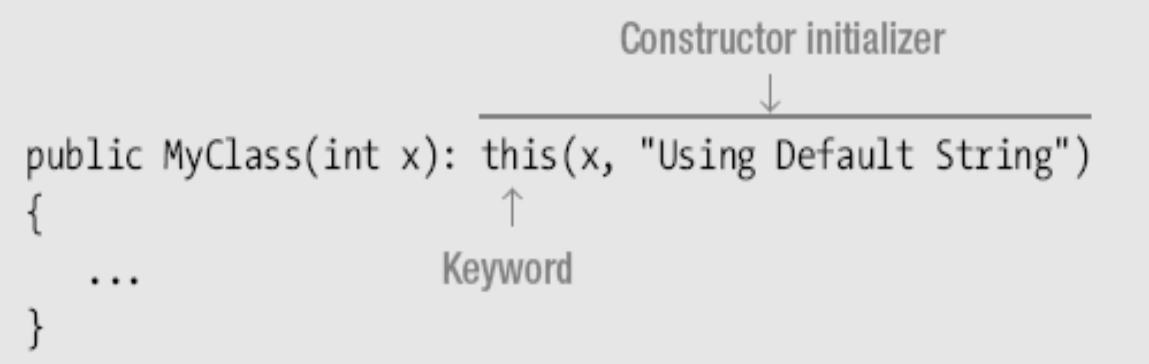


Рисунок 3.32 – Посилання на інший конструктор

13.Модифікатори доступу до класу

Класи можуть об’єднуватися у збірки (assemblies). Для класів передбачено два модифікатори доступу:

– public – видимий (доступний) із будь-якої збірки у системі;
– internal – видимий лише всередині збірки, у якій був задекларований.

В даному випадку класи, які об’єднуються у збірці і реалізовують доступ до себе саме у них. Якщо публічний – доступний по всій програмі. Приклад декларації класів та графічного доступу показує рисунок 3.33:

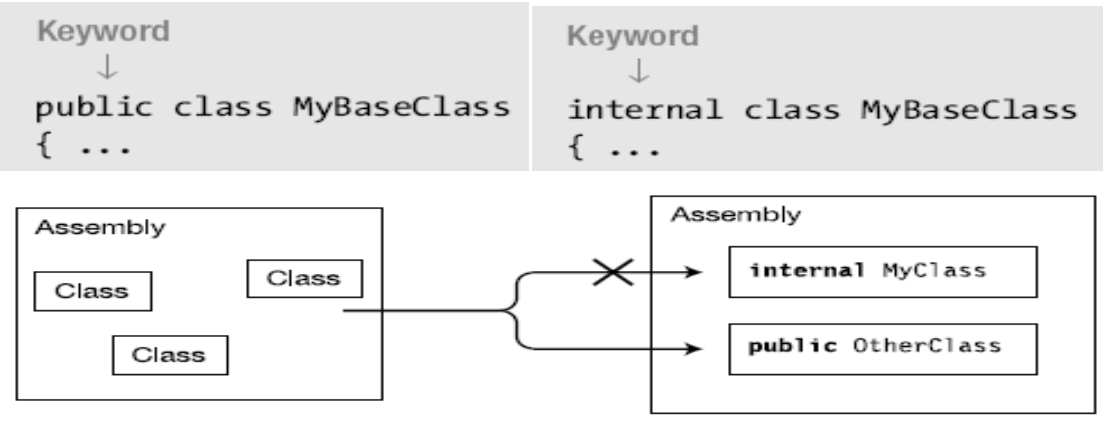


Рисунок 3.33 – Доступ до класів

14.Наслідування між збірками

Для того, щоб наслідувати клас з іншої збірки необхідно виконання наступних умов:

– базовий клас має бути задекларований як public;
– необхідно додати посилання в проекті Visual Studio на збірку, яка зберігає базовий клас.

В даному випадку реалізація такого виконання показана на рисунку 3.34 за таким кодом:

```
// Source file name BaseClass.cs
using System;
 Namespace containing declaration of base class
                ↓
namespace BaseClassNS
{
 Declare the class public, so it can be seen outside the assembly.
      ↓
   public class MyBaseClass {
      public void PrintMe() {
         Console.WriteLine("I am MyBaseClass");
      }
   }
}
```

Рисунок 3.34 – Наслідування між збірками

Наслідування між збірками відбувається через попередню декларацію простору імен базового класу. Потім декларація наслідуваного класу, що унаслідується від головного класу в іншій збірці. І потім безпосередньо створюємо екземпляр даного класу та можемо звертатися до його методів і полів та методів базового класу. Дана реалізація показана на рисунку 3.35:

```
// Source file name Program.cs
using System;
using BaseClassNS;
            ↑
 Namespace containing declaration of base class
namespace UsesBaseClass
{                  Base class in other assembly
                                ↓
   class DerivedClass: MyBaseClass {
      // Empty body
   }

   class Program {
      static void Main( )
      {
         DerivedClass mdc = new DerivedClass();
         mdc.PrintMe();
      }
   }
}
```

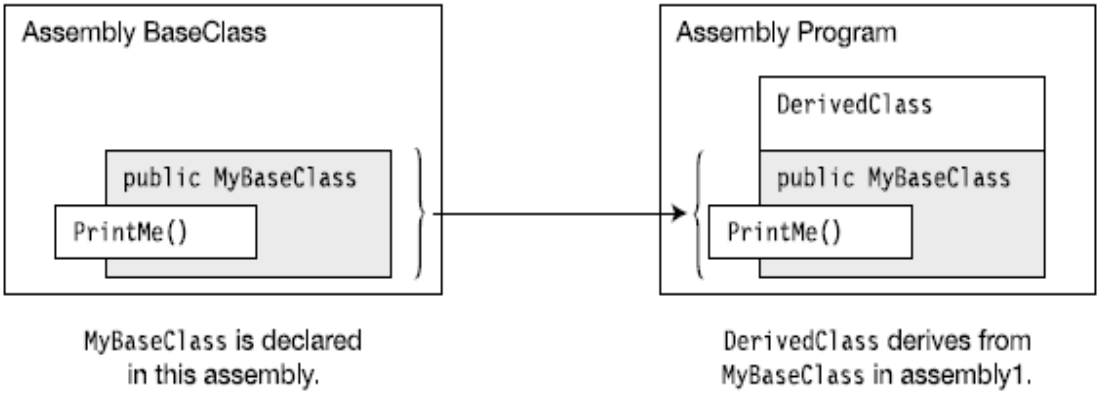
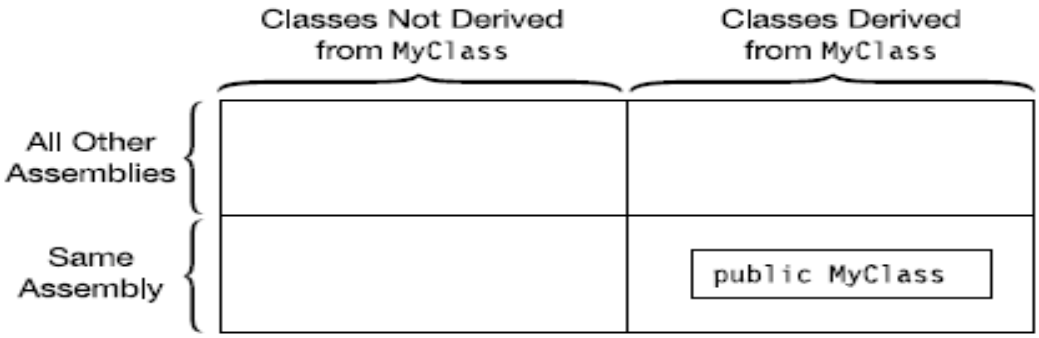


Рисунок 3.35 – Приклад наслідування між збірками

15.Модифікатори доступу до членів класу

В даному питанні буде надано докладнішу та наочніше інформацію про модифікатори доступу до членів класу.

Одним із найпоширеніших є публічний доступ видимості в усій програмі. Він дозволяє працювати з членами класу будь-де при їх виклику та використовувати їх властивості, операції чи значення, а також присвоювати власні (рис. 3.36). Вони одночасно доступні як для збірок, так і наслідуваних класів.




```
Keyword
   ↓
public int MyMember1;
```

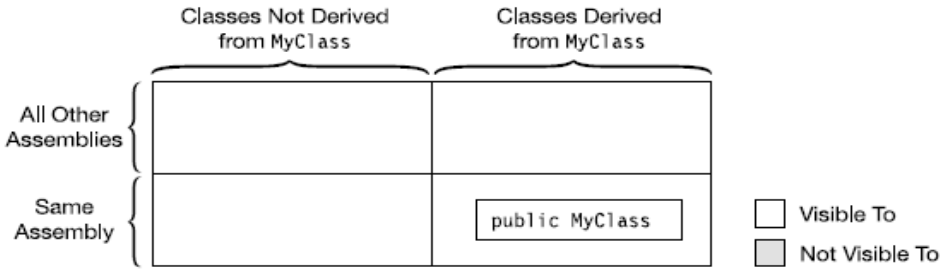
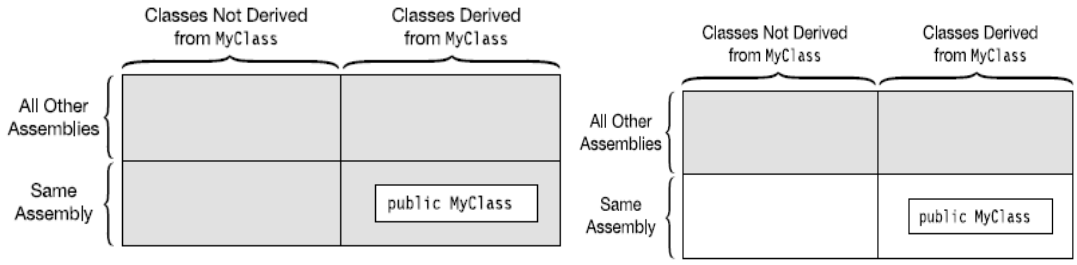


Рисунок 3.36 – Модифікатор public

Різницю та відмінності між іншими модифікаторами доступу демонструє рисунок 3.37:



Private та internal

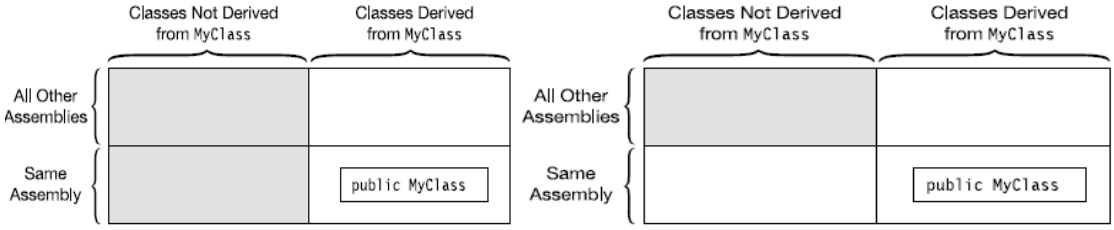


Рисунок 3.37 – Також protected та protected internal

Взагалі загальний опис модифікаторів доступу можна представити на рисунку 3.38:

Modifier	Meaning
`private`	Accessible only within the class
`internal`	Accessible to all classes within this assembly
`protected`	Accessible to all classes derived from this class
`protected internal`	Accessible to all classes that are either derived from this class or are declared within this assembly
`public`	Accessible to any class

	Classes in Same Assembly		Classes in Different Assembly	
	Non-derived	Derived	Non-derived	Derived
`private`				
`internal`	✓	✓		
`protected`		✓		✓
`protected internal`	✓	✓		✓
`public`	✓	✓	✓	✓

Рисунок 3.38 – Резюме по модифікаторам видимості членів класу

ЛК.04 – ПОГЛИБЛЕНІ ПІДХОДИ ДО ОБ’ЄКТНО-ОРІЄНТОВАНОГО ПРОГРАМУВАННЯ

Перелік питань

1. Абстрактні члени класу
2. Абстрактні класи
3. Запечатані класи
4. Зовнішні методи
5. Виключні ситуації
6. Події

На самостійне вивчення:

1. Перевантаження операцій [1, С.382-390].

1. Абстрактні члени класу

Абстрактний член класу призначений лише для того, щоб бути перевизначеним:

- позначається модифікатором abstract;
- не має блоку реалізації (заміняється крапкою з комою);
- декларується лише в абстрактних класах.

Дані члени класу використовуються тоді, коли потрібно їх визначити без реалізації, яка задана за замовчуванням. Створення абстрактного методу та властивості показано на рисунку 4.1:

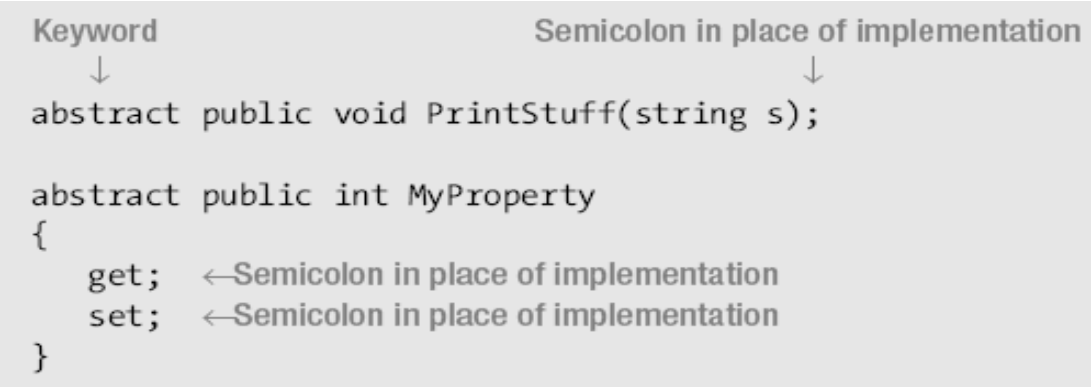


Рисунок 4.1 – Абстрактні члени класу

З точки зору покладених на них задач абстрактні і віртуальні члени класу дуже близькі, тому доцільно їх порівняти. Але мова програмування не допускає існування різних інструментів з однаковою реалізацією. Але існують 3 принципові відмінності, що суттєво впливають на вибір їх використання. По-перше, віртуальні члени класу декларуються за допомогою ключового слова virtual, в той час як абстрактні – abstract. Віртуальні члени класу мають тіло реалізації, а абстрактні ні.

Віртуальні члени класу можуть бути перевизначені у наслідуваному класі за допомогою ключового слова override. А абстрактні члени обов’язково повинні бути перевизначені за такими ж правилами. Віртуальними і абстрактними можуть бути методи, властивості, події та індексатори.

2. Абстрактні класи

Абстрактні класи можуть бути використані лише як основ для наслідування, не можна створювати екземпляри абстрактних класів. Такі класи використовуються безпосередньо для декларації полів та членів для всіх підкласів у загальному вигляді.

Абстрактні класи можуть мати декілька рівнів ієрархії. Клас, який наслідується від абстрактного, має перевизначити усі його абстрактні члени класу. Абстрактні класи є основою для наслідування. Приклад декларації абстрактного класу та правила його наслідування показано на рисунку 4.2:

```
abstract class AbClass                          // Abstract class
{
   ...
}

abstract class MyAbClass : AbClass              // Abstract class derived from
{                                               // an abstract class
   ...
}
```

Рисунок 4.2 – Абстрактні класи

Приклад використання абстрактного класу у програмі та наслідування приведено на рисунку 4.3:

```
abstract class AbClass                                  // Abstract class
{
   public void IdentifyBase()                           // Normal method
   { Console.WriteLine("I am AbClass"); }
    Keyword
       ↓
   abstract public void IdentifyDerived();              // Abstract method
}

class DerivedClass : AbClass                            // Derived class
{  Keyword
      ↓
   override public void IdentifyDerived()               // Implementation of
   { Console.WriteLine("I am DerivedClass"); }          // abstract method
}

class Example
{
   static void Main()
   {
      // AbClass a = new AbClass();          // Error.  Cannot instantiate
      // a.IdentifyDerived();                // an abstract class.

      DerivedClass b = new DerivedClass();   // Instantiate the derived class.
      b.IdentifyBase();                      // Call the inherited method.
      b.IdentifyDerived();                   // Call the "abstract" method.
   }
}
```

Рисунок 4.3 – Приклад використання абстрактних класів

3. Запечатані класи

Запечатані (sealed) класи не можуть виступати у якості основи для наслідування. Запечатані класи фактично є протилежністю абстрактним. Фактично відбувається ізолювання класу. В даному випадку це є корисним для збереження даних членів класу, які він містить (рис. 4.4).

Рисунок 4.4 – Декларація запечатаного класу

4. Зовнішні методи

Зовнішній метод – це такий метод, який не має реалізації під час декларації. Як правило реалізація таких методів виконується на іншій мові програмування. Спосіб реалізації зовнішнього методу позначається у атрибуті, яким помічається метод. Він позначається ключовим словом extern.

Приклад декларації показано на рисунку 4.5:

Рисунок 4.5 – Декларація зовнішнього методу

Приклад його використання показує рисунок 4.6. Відбувається імпортування ресурсів і відповідно робота програми на основі зовнішнього методу:

```
using System;
using System.Text;
using System.Runtime.InteropServices;

namespace ExternalMethod
{
   class MyClass
   {
      [DllImport("kernel32", SetLastError=true)]
      public static extern int GetCurrentDirectory(int a, StringBuilder b);
   }

   class Program
   {
      static void Main( )
      {
         const int MaxDirLength = 250;
         StringBuilder sb = new StringBuilder();
         sb.Length = MaxDirLength;

         MyClass.GetCurrentDirectory(MaxDirLength, sb);
         Console.WriteLine(sb);
      }
   }
}
```

Рисунок 4.6 – Приклад використання зовнішніх методів

5. Виключні ситуації

Виключна ситуація (exception) – помилка у програмі, яка призводить до переривання нормального ходу виконання алгоритму. Наприклад, найпростішою помилкою, що викличе звіт про виключну ситуацію при компіляції, може бути ділення на 0 (рис. 4.7):

```
static void Main()
{
   int x = 10, y = 0;
   x /= y;               // Attempt to divide by zero--raises an exception
}
```

```
Unhandled Exception: System.DivideByZeroException: Attempted to divide by zero.  at
Exceptions_1.Program.Main() in C:\Progs\Exceptions\Program.cs:line 12
```

Рисунок 4.7 – Приклад виключної ситуації

У мові програмування C# передбачена боротьба з виключними ситуаціями. Це конструкція try…catch. Вона дозволяє перевірити код на виключні ситуації і визначити код для їх рішення чи виведення повідомлення користувачеві в разі такої проблеми. Блок try вміщає код, що перевіряється на помилки. Блоки catch можуть бути декілька в залежності від передбачуваних помилок і є вмістилищами виключних ситуацій. Блок finally містить код, який повинен обов’язково виконатися незалежно від знаходження виключної ситуації (рис. 4.8).

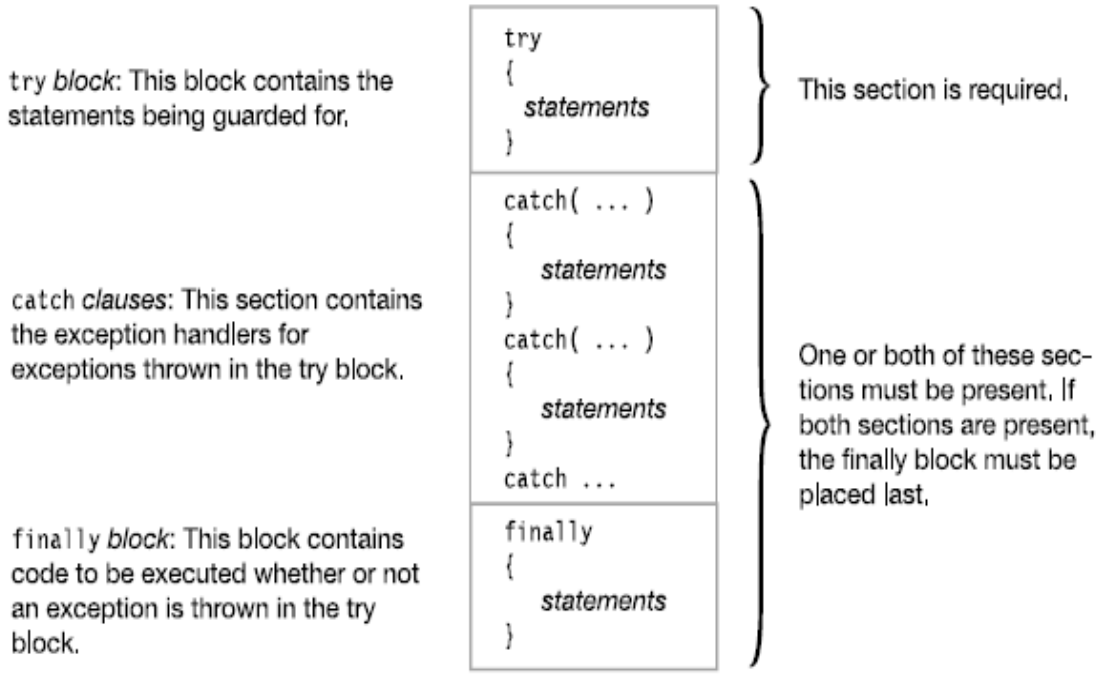


Рисунок 4.8 – Виявлення виключних ситуацій

Приклад використання конструкції для знаходження найпростішої виключної ситуації демонструє рисунок 4.9. Обробляється фактично помилка в коді програми:

```
static void Main()
{
   int x = 10;

   try
   {
      int y = 0;
      x /= y;                    // Raises an exception
   }
   catch
   {
         ...                     // Code to handle the exception

      Console.WriteLine("Handling all exceptions - Keep on Running");
   }
}
```

Рисунок 4.9 – Обробка виключної ситуації

Існує ієрархія класів виключних ситуацій. В даному випадку мова програмування містить головний клас помилок, що поділяються на виключні ситуації, передбачені мовою, та користувацькі виключні ситуації. Користувач може сам заздалегідь обробляти код, передбачати помилки і створювати відповідно класи виключних ситуацій (рис. 4.10):

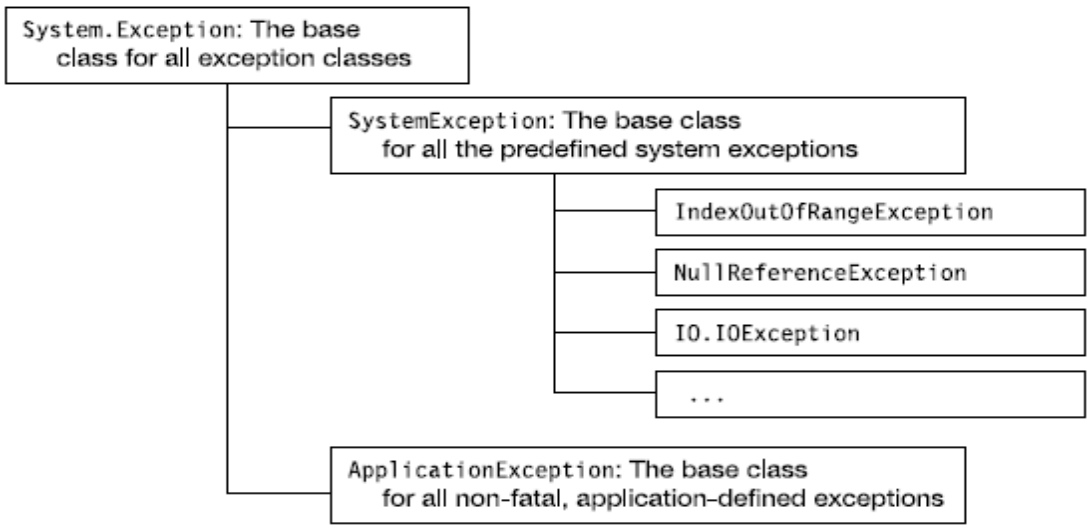


Рисунок 4.10 – Класи виключних ситуацій

Всі класи виключних ситуацій наслідуються від головного – Exeption. Його можна використовувати в коді програми і за допомогою його стандартних методів та членів виявляти та усувати помилки програми (рис. 4.11):

Property	Type	Description
Message	string	This property contains an error message explaining the cause of the exception.
StackTrace	string	This property contains information describing where the exception occurred.
InnerException	Exception	If the current exception was raised by another exception, this property contains a reference to the previous exception.
HelpLink	string	This property can be set by application-defined exceptions to give a URN or URL for information on the cause of the exception.
Source	string	If not set by an application-defined exception, this property contains the name of the assembly where the exception originated.

Рисунок 4.11 – Члени головного класу виключних ситуацій

Оператор catch може мати різний синтаксис. Якщо ловиться узагальнена випадкова і наперед невідома помилка, то він використовується без дужок та додаткової сигнатури. Якщо досліджується код на специфічну відому заздалегідь помилку, то оператор вказується з дужками і назвою цієї помилки. Фактично нею може слугувати певний клас виключної помилки. Можна також в дужках вказувати наперед передбачуваний ідентифікатор (змінну). Даний вибір з подальшим прикладом показують рисунки 4.12, 4.13:

```
catch
{
  Statements
}
```

General `catch` Clause
- Does not have a parameter list after the `catch` keyword.
- Matches any type of exception raised in the `try` block.

```
catch( ExceptionType )
{
  Statements
}
```

Specific `catch` Clause
- Takes the name of an exception class as a single parameter.
- Matches any exception of the named type.

```
catch( ExceptionType InstID )
{
  Statements
}
```

Specific `catch` Clause with ID
- Includes an identifier after the name of the exception class.
- The identifier acts as a local variable in the block of the `catch` clause, and is called the *exception variable*.
- The exception variable references the exception object, and can be used to access information about the object.

Рисунок 4.12 – Варіанти використання catch

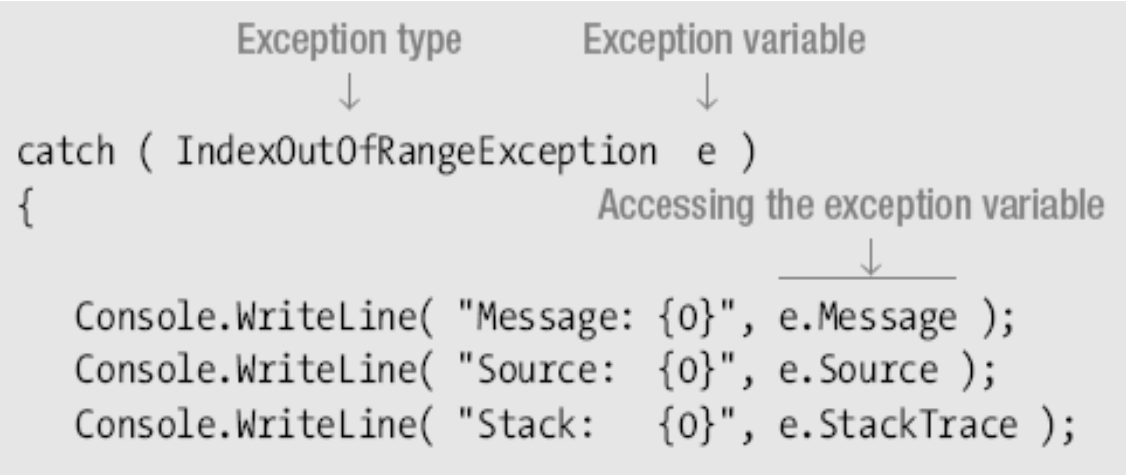


Рисунок 4.13 – Приклад використання catch

Рисунок 4.14 демонструє приклад обробки специфічної виключної ситуації. В даному випадку передбачена стандартна помилка ділення на 0:

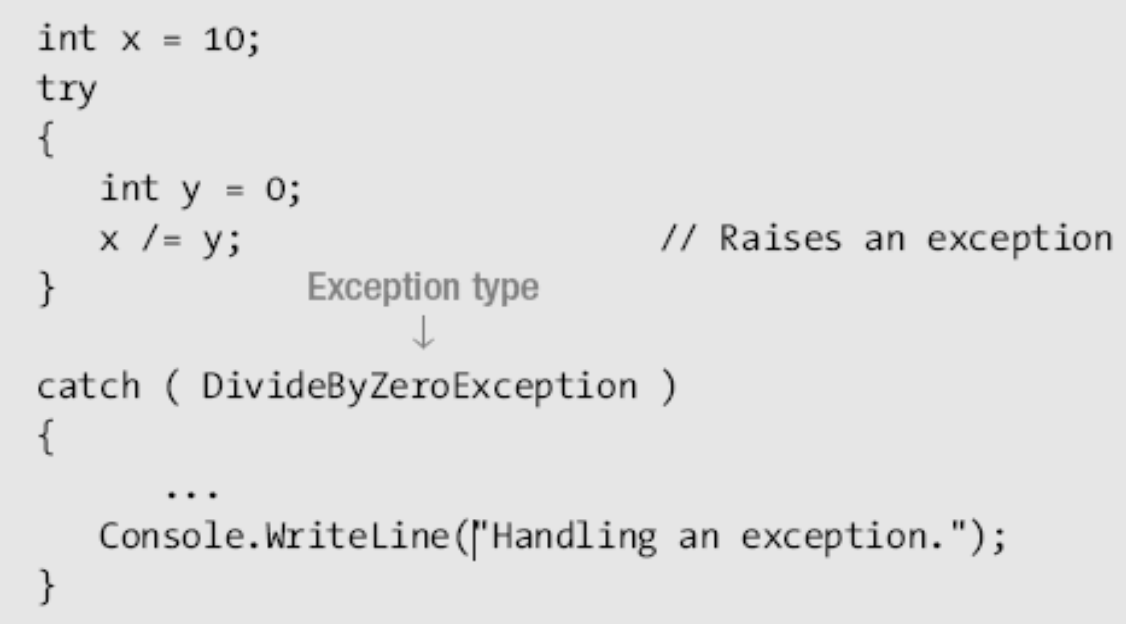


Рисунок 4.14 – Специфічна виключна ситуація

Приклад обробки виключної ситуації з доступом до змінної-об’єкта з інформацією про виключну ситуацію реалізується за допомогою наступного коду, як показано на рисунку 4.15:

```
int x = 10;
try
{
    int y = 0;
    x /= y;                          // Raises an exception
}          Exception type        Exception variable
                  ↓                      ↓
catch ( DivideByZeroException  e )
{                                  Accessing the exception variable
                                              ↓
    Console.WriteLine("Message: {0}", e.Message );
    Console.WriteLine("Source:  {0}", e.Source );
    Console.WriteLine("Stack:   {0}", e.StackTrace );
}
```

```
Message: Attempted to divide by zero.
Source:  Exceptions 1
Stack:       at Exceptions_1.Program.Main() in C:\Progs\Exceptions 1\Exceptions 1\
Program.cs:line 14
```

Рисунок 4.15 – Приклад обробки

При обробці виключних ситуацій можна використовувати декілька операторів catch. Вони виконуються по черзі в залежності від пошуку виключної ситуації, її виду. Тому потрібно створювати чергу виключних ситуацій. Найперше потрібно створити пошук найбільш специфічних типів помилок, а далі досліджувати найзагальніші. Завершення обробки (декларація та виконання блоку final) повинно слідувати після всіх виключних ситуацій (рис. 4.16).

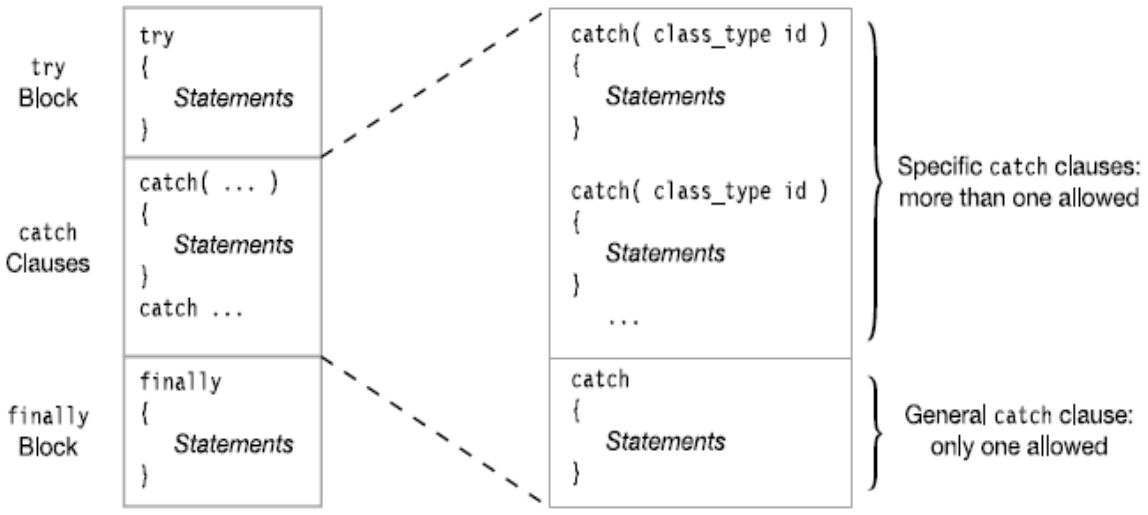


Рисунок 4.16 – Декілька операторів catch

Якщо необхідною умовою програми є кінцеве обов'язкове виконання певних операцій та коду, потрібно використовувати блок finally. Для цього він повинен слідувати після всіх оброблених виключних ситуацій та призводити до виконання коду. Ідея його використання показана на рисунку 4.17:

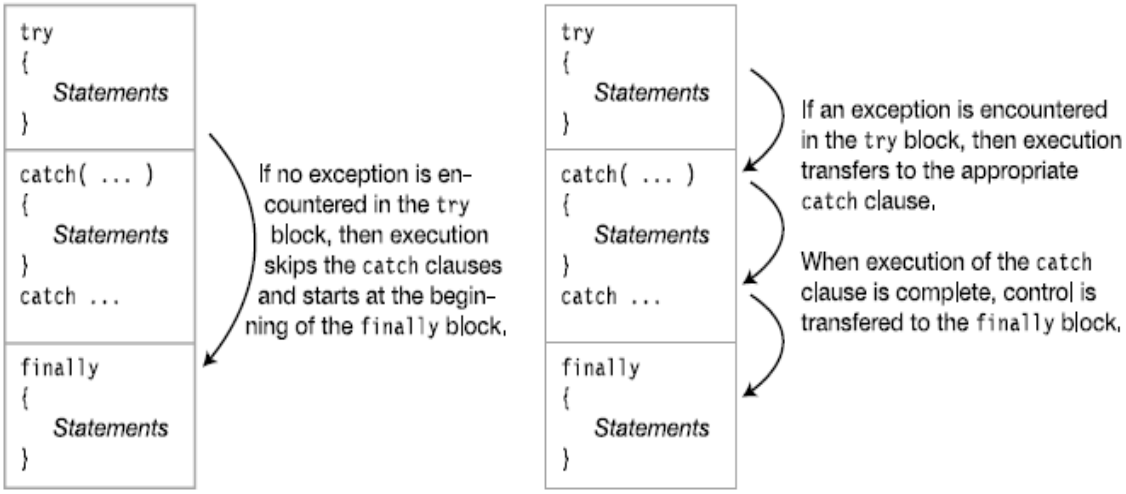


Рисунок 4.17 – Використання блоку finally

Приклад використання показує рисунок 4.18. В даному випадку відсутня конструкція catch. І незалежно від програмного коду в конструкції try у консольному вигляді на екран буде виведено неохідне повідомлення:

```
try
{
   if (inVal < 10) {
      Console.Write("First Branch  - ");
      return;
   }
   else
      Console.Write("Second Branch - ");
}
finally
{ Console.WriteLine("In finally statement"); }
```

Рисунок 4.18 – Приклад використання блоку finally

Але використання блоку finally не є обов'язковим. Тобто після перевірки виключної ситуації достатньо лише й блоків catch. В даному випадку відбувається пошук обробника виключної ситуації (рис. 4.19):

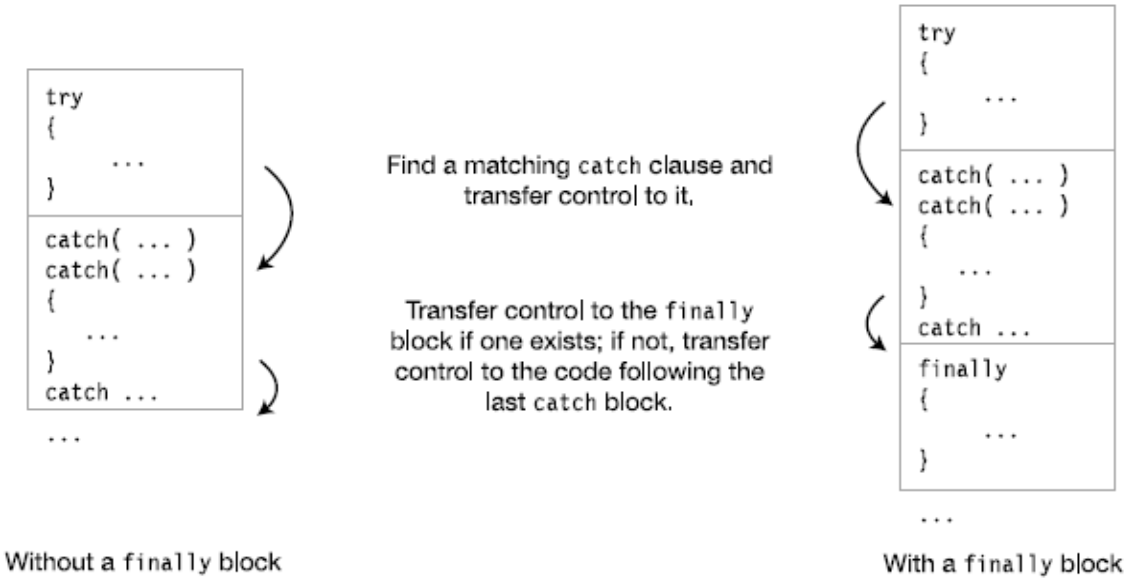


Рисунок 4.19 – Обробка виключної ситуації

Можна викликати виключні ситуації явно. Наприклад, рисунки 4.20, 4.21 демонструє викликання стандартної помилки throw ExceptionObject. Це використання ситуації перевірки пустого значення. Також відбувається порівняння з роботою правильного коду.

```
class MyClass
{
   public static void PrintArg(string arg)
   {
      try
      {
         if (arg == null)
         {
            ArgumentNullException MyEx = new ArgumentNullException();
            throw MyEx;
         }
         Console.WriteLine(arg);
      }
      catch (ArgumentNullException e)
      {
         Console.WriteLine("Message:  {0}", e.Message);
      }
   }
}
```

Рисунок 4.20 – Використання явних виключних ситуацій

```
class Program
{
    static void Main()
    {
        string s = null;
        MyClass.PrintArg(s);
        MyClass.PrintArg("Hi there!");
    }
}
```

Рисунок 4.21 – Результат на екрані

Можливе викликання виключних ситуацій повторно. Рисунок 4.22 демонструє у блоці catch. Для цього throw використовується після викликання виключної ситуації для перевірки повторних ситуацій такого ж типу. Маємо такий результат:

```
public static void PrintArg(string arg)
{
   try
   {
      try
      {
         if (arg == null)
         {
            ArgumentNullException MyEx = new ArgumentNullException();
            throw MyEx;
         }
         Console.WriteLine(arg);
      }
      catch (ArgumentNullException e)
      {
         Console.WriteLine("Inner Catch:  {0}", e.Message);
         throw;
      }    ↑
   }     Rethrow the exception—no additional parameters
   catch
   {
      Console.WriteLine("Outer Catch:  Handling an Exception.");
   }
}
```

Рисунок 4.22 – Повторний виклик виключних ситуацій

ЛК.05 – ВВЕДЕННЯ ДО ПРОГРАМУВАННЯ ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА

Перелік питань.

1. Основні принципи побудови графічного інтерфейсу користувача та програм, які керуються подіями.
2. Створення форм та управління ними.
3. Модальні і немодальні форми.
4. Основні елементи графічного інтерфейсу користувача.

На самостійне вивчення:

1. Клас Application [1, С.755-758].

1. Основні принципи побудови графічного інтерфейсу користувача та програм, які керуються подіями.

Існує 2 основні підходи розробки програм. Перший заснований на алгоритмах і передбачає послідовне виконання кроків у програмі. Тобто існує початок, безпосереднє тіло програми з реалізацією виконання та її кінець (рис. 5.1).

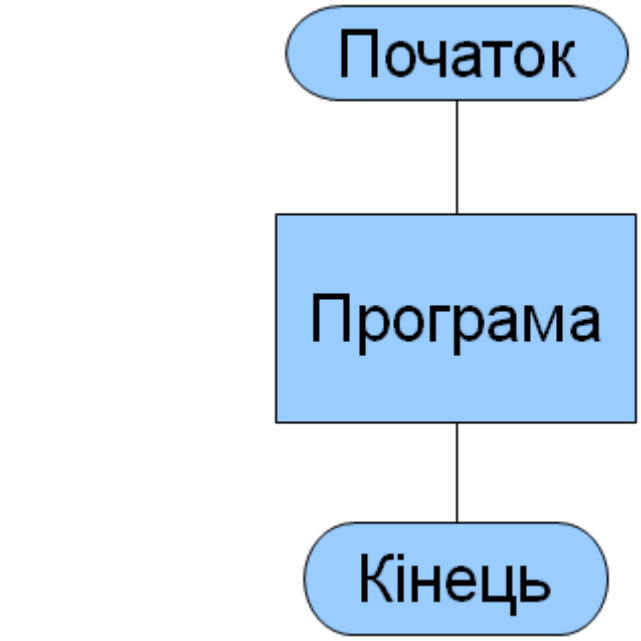


Рисунок 5.1 – Алгоритмічний підхід

Сьогодні більшої популярності і зручності набув підхід, заснований на подіях. Він передбачає існування програми. Вона не виконується поступово, а залежить від впливу на неї. Тобто програма приймає події та реагує на них в будь якій послідовності виконання таким чином, що передбачений програмним кодом (рис. 5.2).

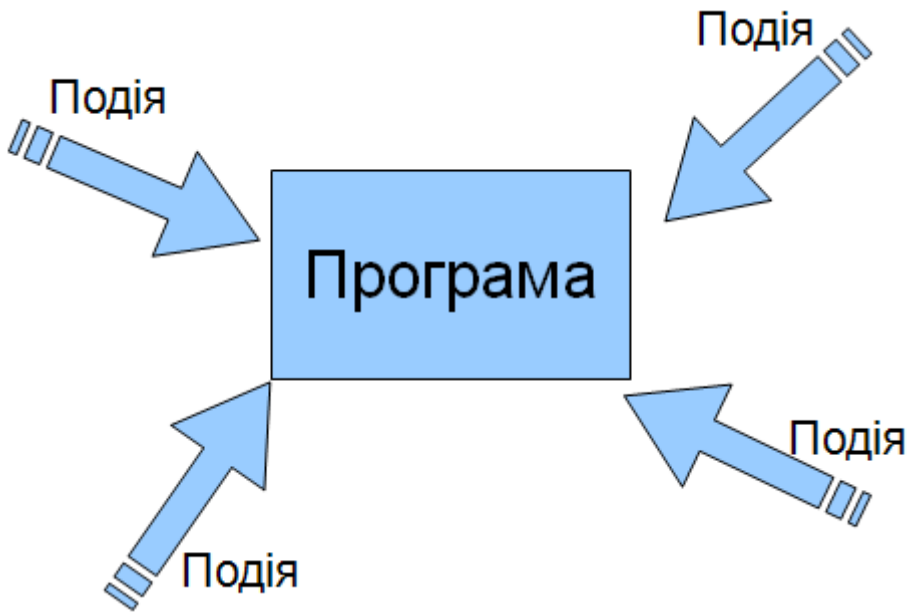


Рисунок 5.2 – Підхід, що оснований на подіях

В даному випадку буде використовуватися програмування під операційну систему. У мові програмування C# для цього реалізовано окремий проект – Windows Forms Application. Приклад його створення показано на рисунку 5.3. Можливе попереднє вказання проекту, його назви та розташування.

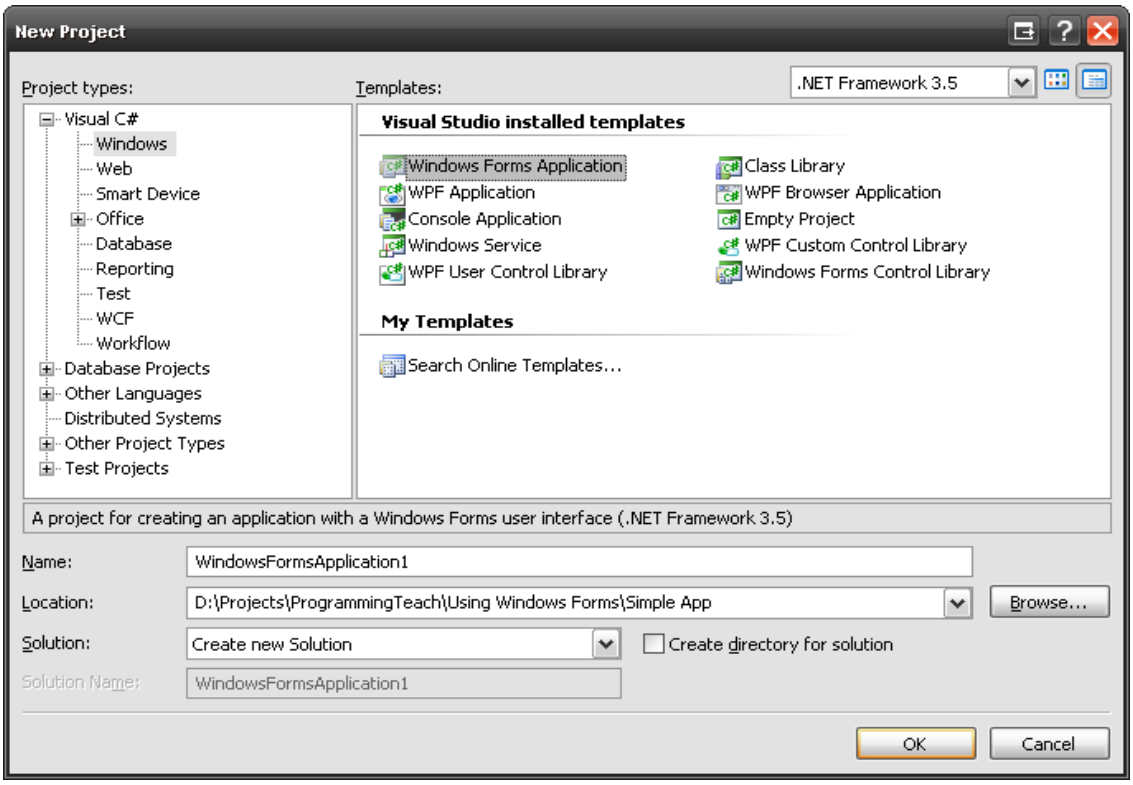


Рисунок 5.3 – Створення нового проекту

В даному випадку безпосередньо переходимо до середовища розробки. Воно містить пусте діалогове вікно, саме з яким і передбачається робота. Також для нього присутні набір властивостей і подій, які можна змінювати. Панель інструментів містить стандартні інструменти вікна, які можна додавати до форми. Вигляд середовища розробки демонструє рисунок 5.4:

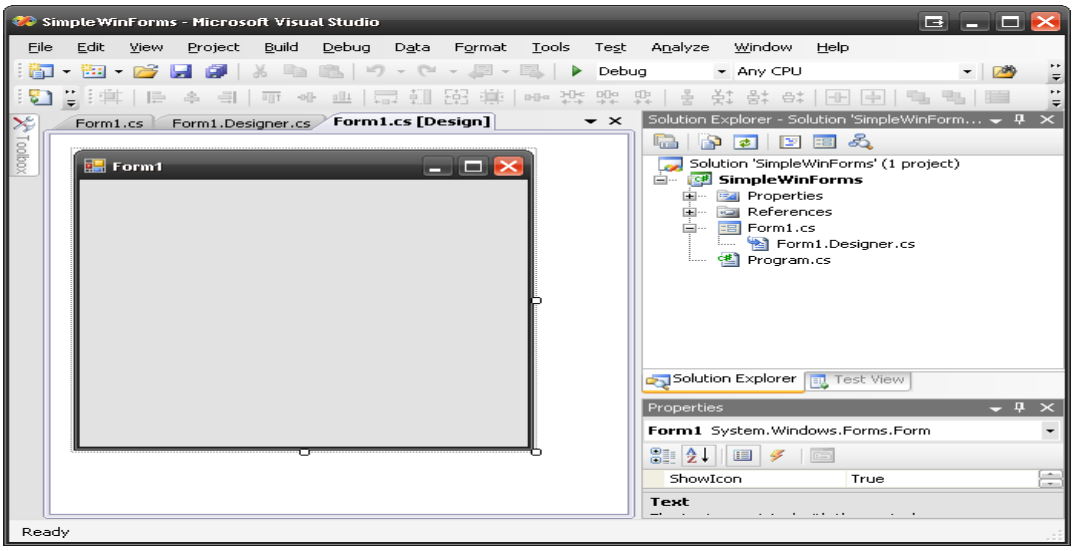


Рисунок 5.4 – Вигляд середовища розробки

Але вже передбачається виконання програми. Якщо не вдосконалювати середовище розробки жодним чином, то програма все одно буде компілюватися і виводити на екран пусту форму – діалогове вікно, створене засобами операційної системи (рис. 5.5).

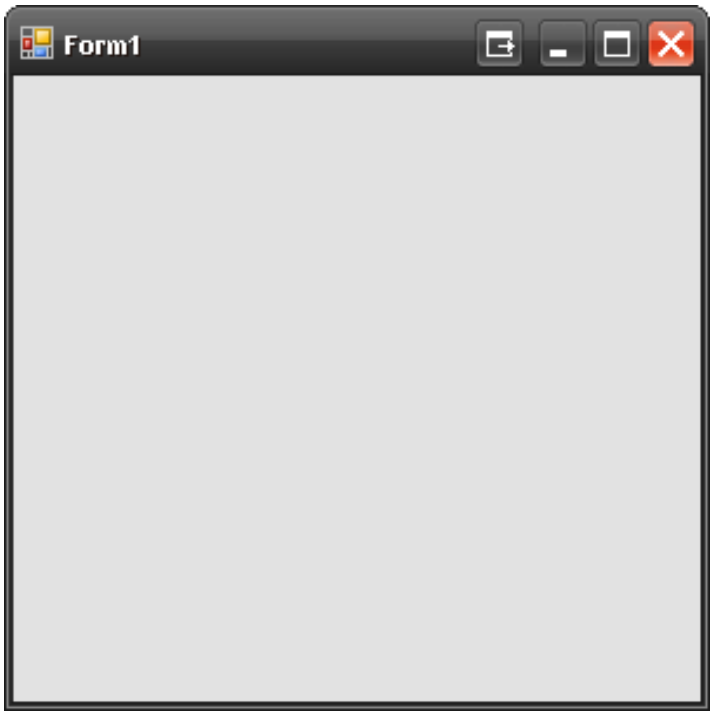


Рисунок 5.5 – Діалогове вікно, створене засобами операційної системи

Але крім візуального середовища розробки існує також і програмне. Тобто, щоб задати функціональність кожному елементу на формі, потрібно

передбачити відповідний програмний код. Мова програмування передбачає спеціальний файл його розробки. При створенні нового проекту автоматично у ньому генеруються необхідні простори імен та відбувається ініціалізація форми (лістинг 5.1):

Лістинг 5.1 - Вміст файлу Form1.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;
```

Продовження лістингу 5.1:

```
namespace SimpleWinForms
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }
    }
      }
```

Ще одним файлом, що створюється автоматично, є Form1.Designer.cs, який забезпечує вигляд форми і її взаємодію з системою. Він також створюється автоматично при створенні проекту і містить певний програмний код на початку (лістинг 5.2):

Лістинг 5.2 - Вміст файлу Form1.Designer.cs

```
namespace SimpleWinForms
{
    partial class Form1
    {
        /// <summary>
        /// Required designer variable.
        /// </summary>
        private System.ComponentModel.IContainer components = null;

        /// <summary>
        /// Clean up any resources being used.
        /// </summary>
        /// <param name="disposing">true if managed resources should be
disposed; otherwise, false.</param>
        protected override void Dispose(bool disposing)
```

```
        {
            if (disposing && (components != null))
            {
                components.Dispose();
            }
            base.Dispose(disposing);
        }

        #region Windows Form Designer generated code
        /// <summary>
        /// Required method for Designer support - do not modify
        /// the contents of this method with the code editor.
        /// </summary>
        private void InitializeComponent()
        {
            this.components = new System.ComponentModel.Container();
            this.AutoScaleMode = System.Windows.Forms.AutoScaleMode.Font;
            this.Text = "Form1";
        }
        #endregion
    }
}
```

Попередні 2 форми фактично були створені на основі часткових класів з певною реалізацією. Існує ще 1 файл, завдяки якому і відбувається безпосереднє виконання програми - Program.cs. У ньому автоматично створюється метод Main з певною специфікацією та атрибутами, що є точкою входу до виконання програми. Він саме і забезпечує виконання програми. Вміст його демонструє лістинг 5.3:

Лістинг 5.3 - Вміст файлу Program.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Windows.Forms;

namespace SimpleWinForms
{
    static class Program
    {
        /// <summary>
        /// The main entry point for the application.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Application.Run(new MainForm());
        }
    }
}
```

Для показу відображення простої програми варто додати певну функціональність до форми. Маємо виконання програми за кодом (лістинг 5.4).

Лістинг 5.4 - Програма, яка виводить привітання під час запуску

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleWinForms
{
    public partial class Form1 : Form
```

Продовження лістингу 5.4:

```
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            MessageBox.Show("Hello, World!");
        }
    }
}
```

В даному випадку при завантаженні даної форми і натисненні на ній кнопкою миші на екран буде виводитися повідомлення (рис. 5.7), що і є результатом виконання програми.

Рисунок 5.7 – Виконання програми

2. Створення форм та управління ними

Мова програмування передбачає створення багатьох пов’язаних форм в одному проекті, які викликаються за допомогою реалізації програмного коду з певною взаємодією. Але обов’язково необхідна наявність головної форми, що компілюється спочатку Тобто вона працює постійно і забезпечує виклик інших форм. Її можна додати в проект як нову форму за допомогою меню Visual Studio або контекстного меню утвореного проекту (рис. 5.8).

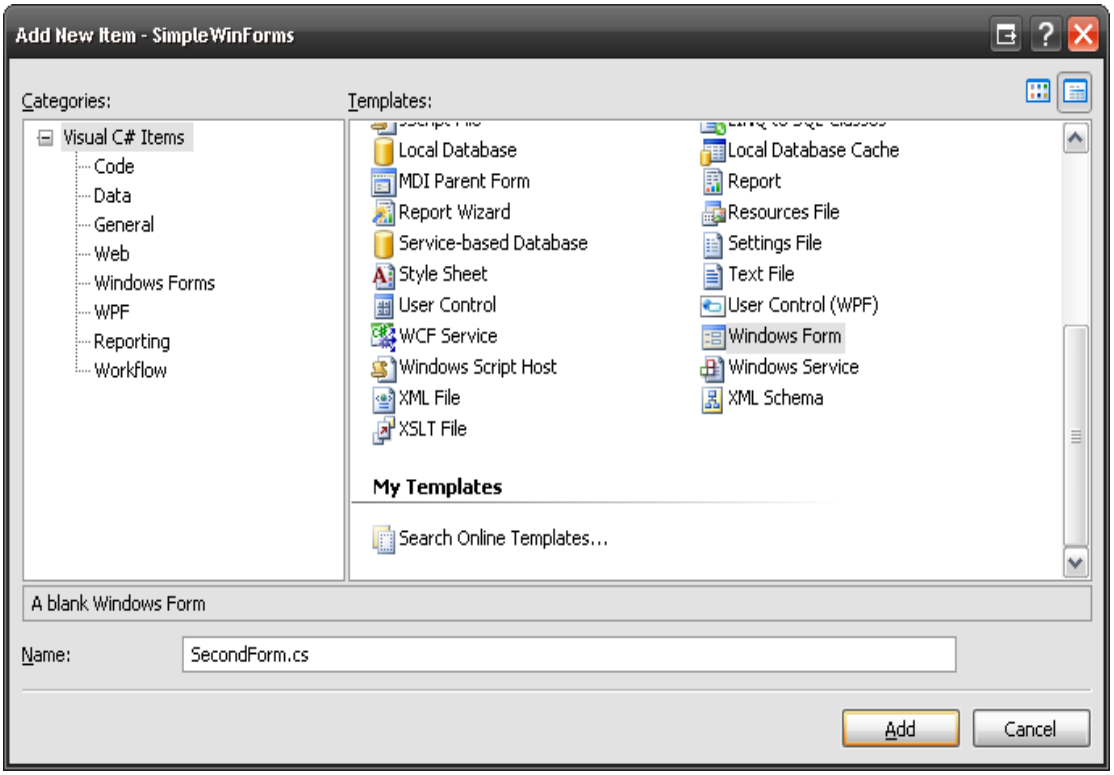


Рисунок 5.8 – Створення форм та управління ними

Взагалі форма – це клас, тому потрібно в будь-якому випадку створювати екземпляр даного класу. Можна реалізувати програму показу 1 форми завдяки натисненню на копці в іншій формі. Для цього на головну форму слід додати інструмент-кнопку та встановити її певні властивості (текст відображення, назву тощо). Ц е демонструє рисунок 5.9:

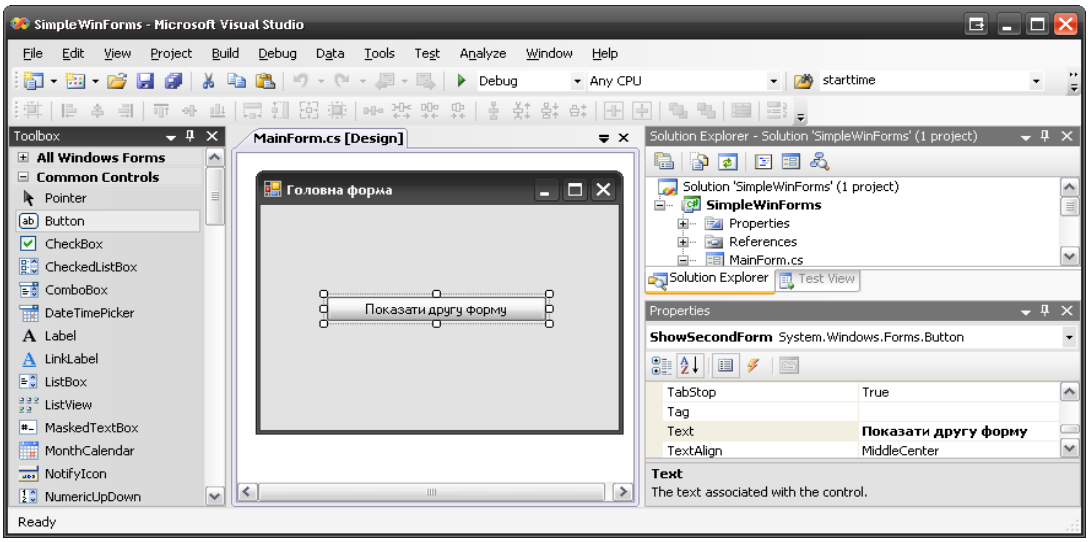


Рисунок 5.9 - Розміщуємо кнопку та задаємо властивості

При натисненні подвійним кліком мишки на кнопці переходимо до коду, де і задаємо функціональність кнопки за умови додання до проекту 2 форми (лістинг 5.5):

Лістинг 5.5 - Задаємо програмний код для показу другої форми

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;


namespace SimpleWinForms
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        private void SecondForm_Click(object sender, EventArgs e)
        {
            SecondForm TheSecondForm = new SecondForm();
            TheSecondForm.Show();
        }
    }
}
```

В результаті буде створена головна форма з заданою функціональністю. Тобто при натисненні на кнопку головної форми буде здійснюватися відкриття іншої (рис. 5.10).

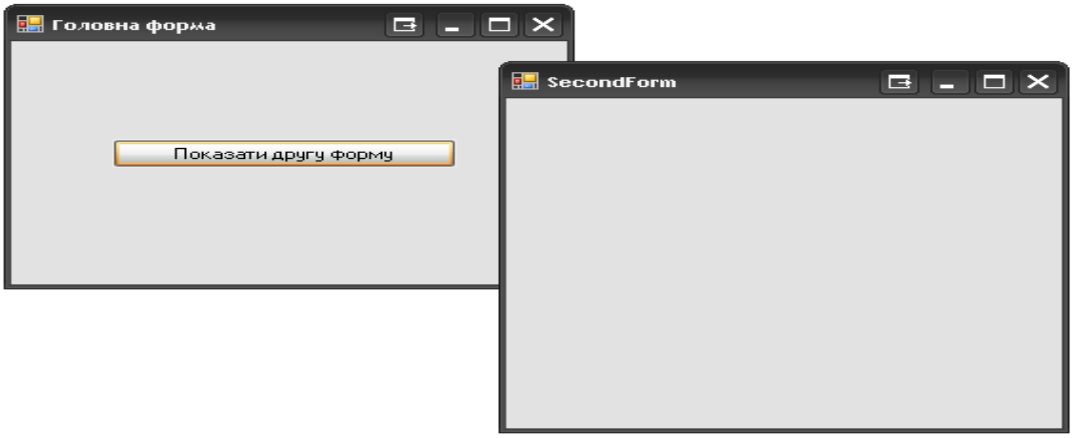


Рисунок 5.10 - Результат

3. Модальні і немодальні форми

Потрібно розрізняти принципові відмінності між цими видами форм. Модальні вікна не дозволяють перейти до інших вікон, доки вони не будуть закриті. Немодальні вікна ніяким чином не обмежують можливість переходу між вікнами.

Для показу форми у модальному режимі використовується метод ShowDialog(). Для показу форми у немодальному режимі використовується метод Show().

4. Основні елементи графічного інтерфейсу користувача

До основних елементів графічного інтерфейсу користувача відносяться всі елементи панелі інструментів розробки програми. Взагалі форми можуть створюватися багатофункціональні. Це передбачає можливість додавання будь-яких елементів до форми та присвоєння їх функціональності. Рисунок 5.11 показує створення комплексної форми з різними інструментами:

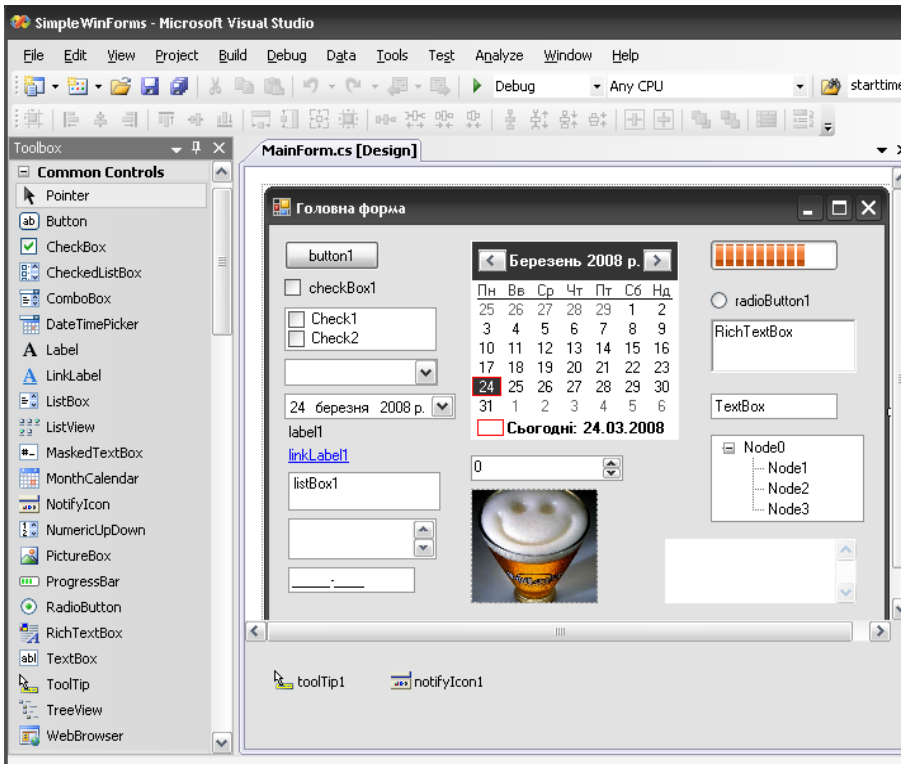


Рисунок 5.11 - Основні елементи графічного інтерфейсу користувача

Після розміщення всіх елементів і встановлення їх властивостей та прописання відповідного коду функціональності отримаємо результат виконання попередньої форми (рис. 5.12):

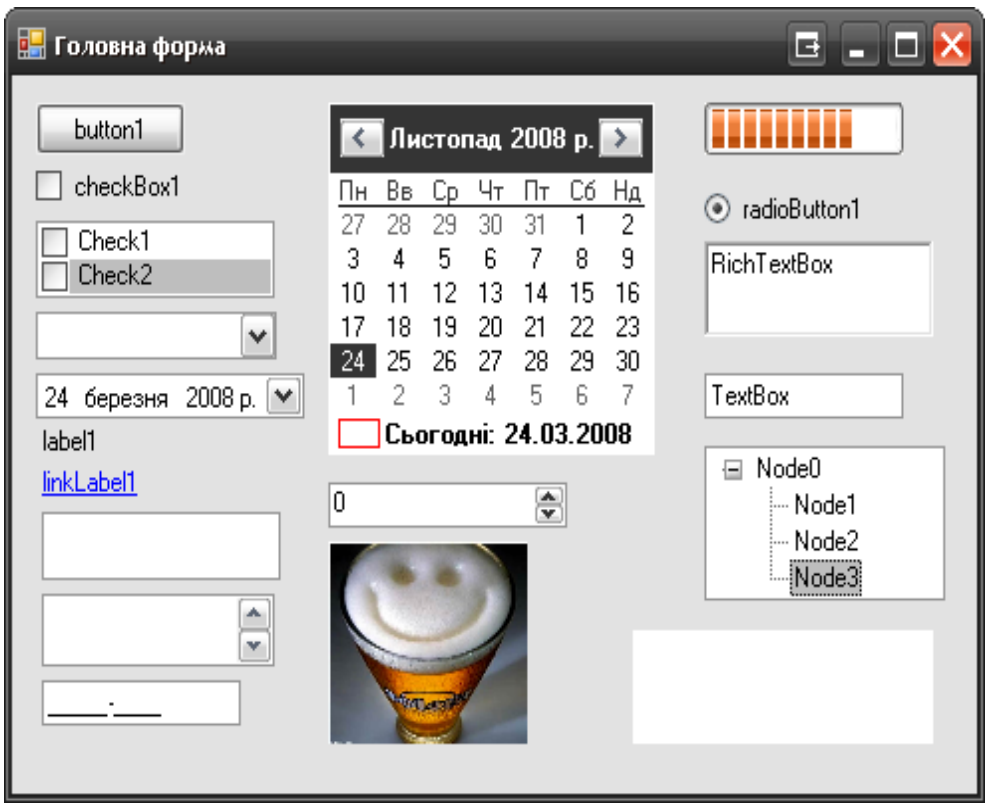


Рисунок 5.12 - Працююча програма з різними елементами управління

ЛК.06 – ОСНОВНІ ЕЛЕМЕНТИ УПРАВЛІННЯ ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА

Перелік питань

1. Важливі властивості і події елементів інтерфейсу.
2. Важливі властивості і події форми.
3. Елемент управління Button.
4. Елемент управління MenuStrip.
5. Елемент управління ConextMenuStrip.
6. Елемент управління CheckBox.
7. Елемент управління RadioButton.
8. Елемент управління CheckedListBox.
9. Елемент управління ListBox.
10. Елемент управління ComboBox.
11. Елемент управління Label.
12. Елемент управління TextBox.

На самостійне вивчення:

1. Елемент управління ToolStrip [1, С.792-798].

1. Важливі властивості і події елементів інтерфейсу.

Мова програмування містить набір властивостей та подій для кожного елементу управління графічного інтерфейсу користувача. Вони містяться в панелі інструментів і доступні для реалізації при кожному новому створенні елементу управління. Списки властивостей і подій представлено на рисунку 6.1 при обиранні необхідного переліку:

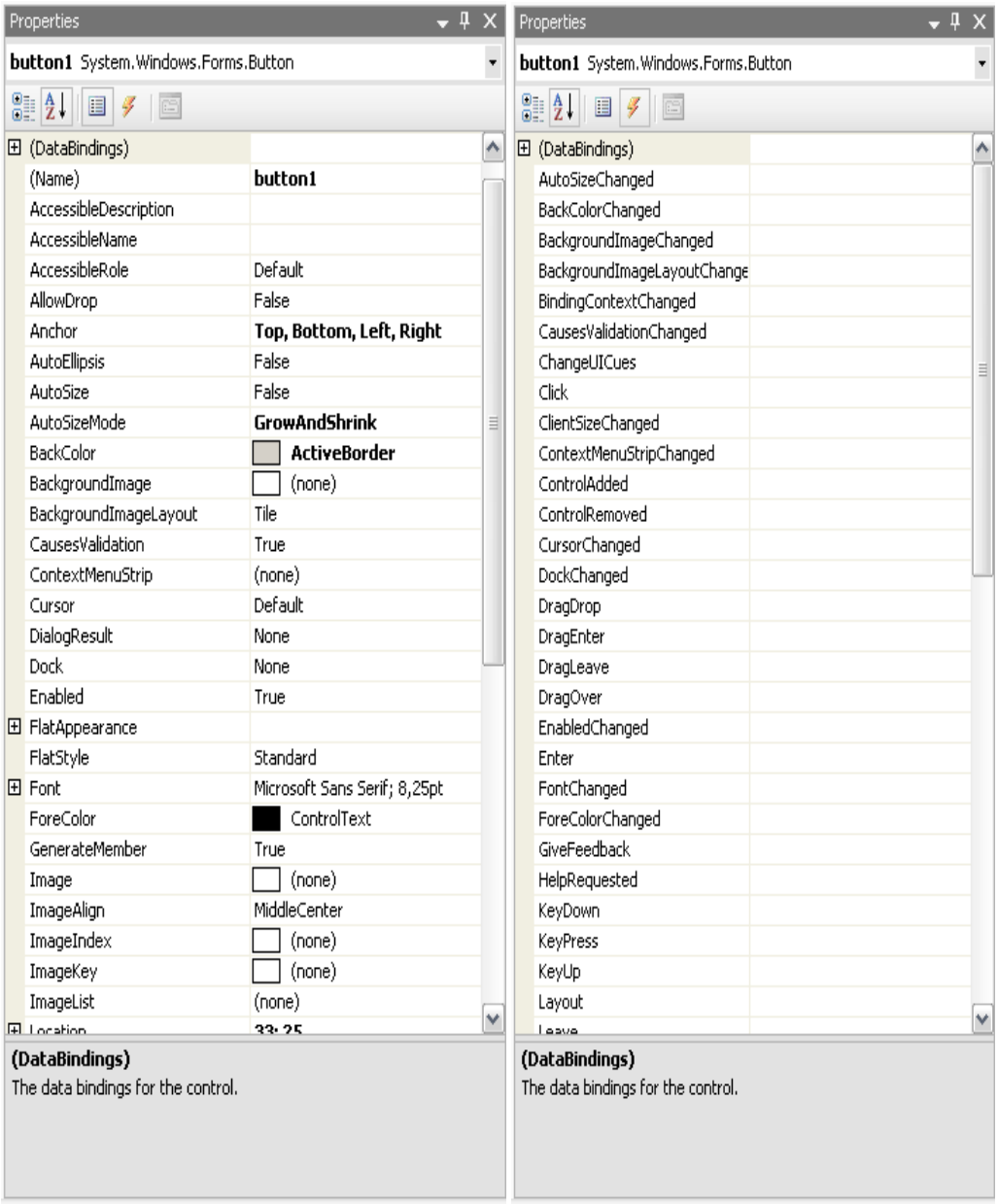


Рисунок 6.1 - Важливі властивості і події елементів інтерфейсу

Серед найголовніших властивостей варто приділити увагу саме таким:

- Name (string) – назва;
- Anchor (Top, Bottom, Left, Right) – якорь;
- AutoEllipsis (bool) – автоматичне обрізання тексту;
- AutoSize (bool) – автоматичний розмір;
- AutoSizeMode (GrowOnly, GrowAndShrink) – режим авто розміру;
- ContextMenuStrip – контекстне меню;
- Enabled (bool) – визначає, чи є дозволеним елемент управління;

- Font – шрифт;
- ImageList – список зображень, пов’язаний з елементом;
- Location (X, Y) – позиція елементу відносно контейнеру;
- Locked (bool) – забороняє рухати та змінювати розмір елементу;
- Margin – границя;
- MaximumSize, MinimumSize – максимальний та мінімальний розмір;
- Size (Width, Height) – розмір;
- Tag – через цю властивість будь-який об’єкт можна пов’язати з елементом управління;
- Text (string) – текст;
- TextAlign – вирівнювання тексту;
- Visible (bool) – видимий.

Події та властивості характерні як для самої форми, так і для конкретних елементів управління. Звичайно перелік є значно ширшим, при чому кожен елемент управління має свої додаткові властивості, що доповнюють загальні. Також найпоширенішими подіями, характерними для елементів управління, є такі:

– Click – відбулася подія “click”;
– DragDrop – закінчилася операція “drag-and-drop”;
– Enter – елемент управління отримує фокус;
– HelpRequested – відбувається, коли користувач натиснув F1;
– KeyDown – натиснута клавіша на клавіатурі;
– KeyUp – відпущена клавіша на клавіатурі;
– KeyPress – натиснута та відпущена клавіша на клавіатурі;
– Leave – елемент управління втрачає фокус;
– MouseClick – відбувся “клік” мишою;
– MouseDown – натиснута кнопка миші;
– MouseUp – відпущена кнопка миші;
– TextChanged – змінена властивість “text”.

2. Важливі властивості і події форми

Для форми характерні власні властивості та події,за допомогою яких їй можна задавати необхідну функціональність. До переліку властивостей варто віднести саме такі:

- Name – назва форми;
- Text – заголовок форми;
- BackColor – колір робочої області форми;
- CancelButton – кнопка, яка “натискається”, коли користувач натискає ESC;
- ContextMenuStrip – контекстне меню форми;
- ControlBox (bool) – показувати системне меню;
- Enabled – дозволена;
- FormBorderStyle – стиль рамки;
- HelpButton – кнопка допомоги на заголовку;
- Icon – іконка;
- KeyPreview (bool) – обробляти клавіатурні події раніше за компоненти;
- Location, Size – позиція, розмір;
- MainMenuStrip – головне меню форми;
- MaximizeBox (bool) – показувати кнопку “Maximize”;
- MaximumSize, MinimumSize – максимальний, мінімальний розмір;
- Opacity (%) – непрозорість;
- StartPosition – начальна позиція;
- TopMost – розміщати поверх інших;
- WindowState – початковий стан форми.

Важливі події форми:

- Load – завантаження форми;
- Activated – форма стає активною;
- Deactivate – форма стає неактивною;
- PreviewKeyDown – натиснута та відпущена клавіша на клавіатурі;

– Resize – відбувається зміна розміру;
– Shown – відбувається, коли форма була вперше показана;
– FormClosing – відбувається перед закриттям форми;
– FormClosed – відбувається після закриття форми.

3. Елемент управління “Button”

Цей елемент управління реалізує вигляд та функціонування кнопки. Призначення – виконувати дії після натискання за допомогою кнопок миші чи з клавіатури.

Серед найважливіших властивостей необхідно виділити:

– Text – текст на кнопці;
– DialogResult – результат, який автоматично повертається при натисканні кнопки, якщо форма показана у модальному режимі.

Звичайно, найголовнішою подією для кнопки є її натиснення, а подія, яка це реалізує - Click – натискання кнопки.

Створимо найпростішу програму її використання. Додамо до форми кнопку, введемо її властивість тексту та подвійним кліком перейдемо до файлу реалізації кнопки. Встановимо такий код для кнопки (лістинг 6.1):

Лістинг 6.1 - Приклад використання “Button”

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace SimpleWinForms
{
    public partial class MainForm : Form
    {
        public MainForm()
        {
            InitializeComponent();
        }

        private void Hello_Click(object sender, EventArgs e)
        {
            MessageBox.Show("Hello, World!");
```

```
        }
    }
      }
```

Результатом буде виведення на екран повідомлення із заданою текстовою інформацією, а вигляд форми та реалізація завдання виглядатимуть наступним чином (рис. 6.2, 6.3):

Рисунок 6.2 – Вигляд головної форми

Рисунок 6.3 – Виконання програми

4. Елемент управління MenuStrip

Фактично це є стандартне головне меню форми та програми. Використовується для побудови меню програми. Назву відповідно кожного пункту мова програмування надає прописувати користувачеві за бажанням або додати стандартні елементи управління.

Для позначення літер швидкого доступу перед ними необхідно використати знак “&”. Подвійний пункт на пункті меню – створюється обробник Click. Створення такого елементу управління показує рисунок 6.4:

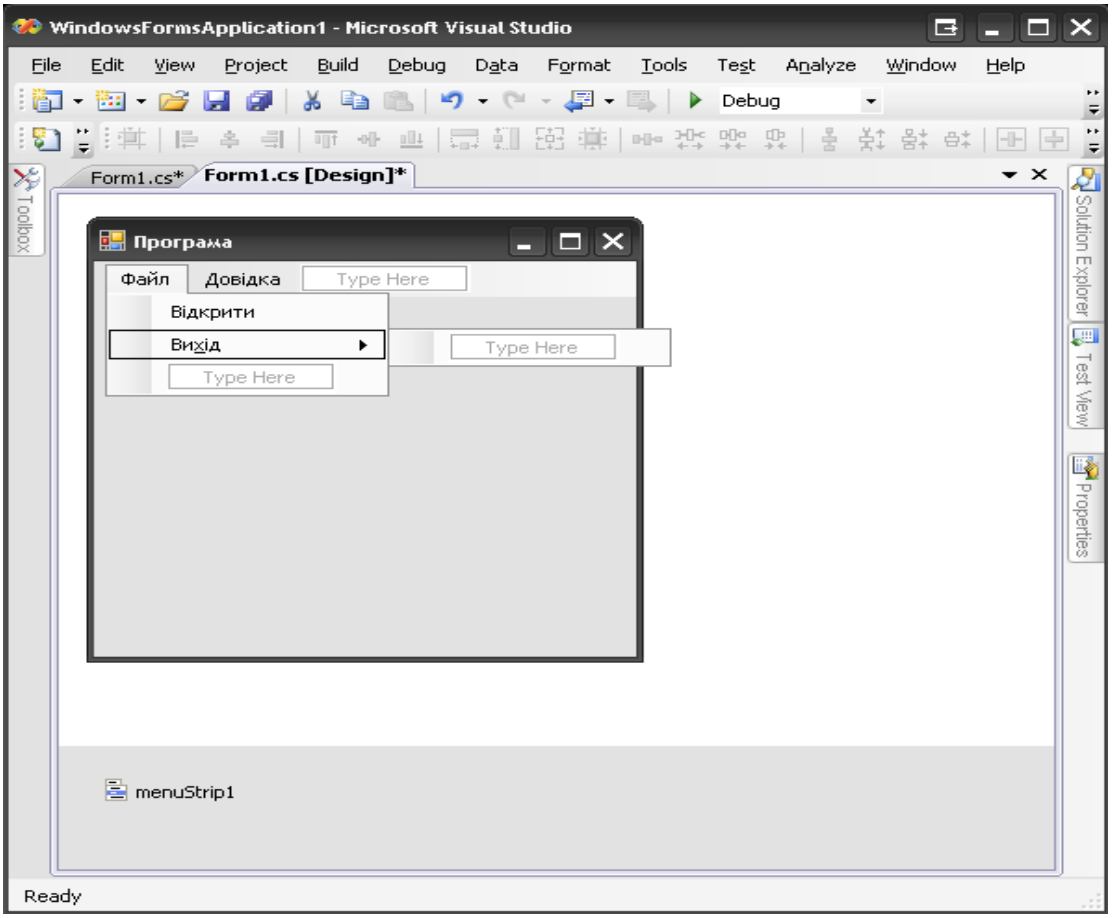


Рисунок 6.4 – Створення елементу управління MenuStrip

Вставка стандартних елементів забезпечується завдяки контекстному меню даного елементу управління. Внаслідок цього до форми уде додано стандартні елементи управління і програма набуде вигляду (рис. 6.5):

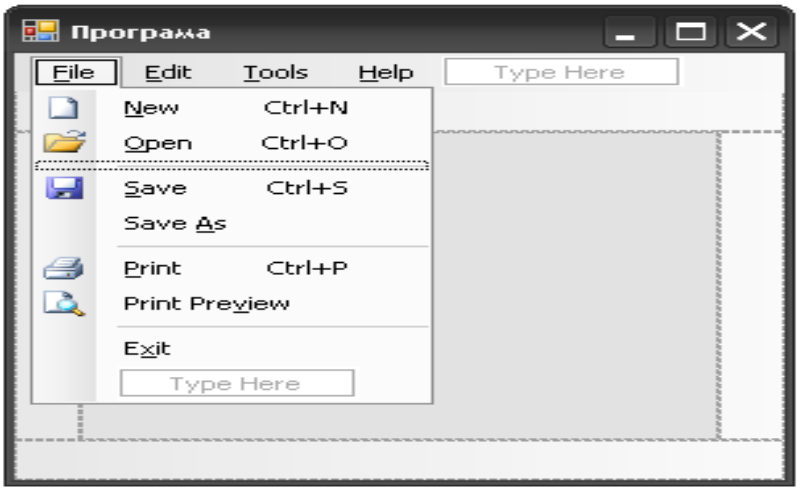


Рисунок 6.5 – Вставка стандартних елементів меню

Можна безпосередньо працювати з редактором меню, який забезпечує створення нових пунктів меню і керування його елементами. В ньому ж можна відразу задавати властивості кожного пункту меню (рис. 6.6):

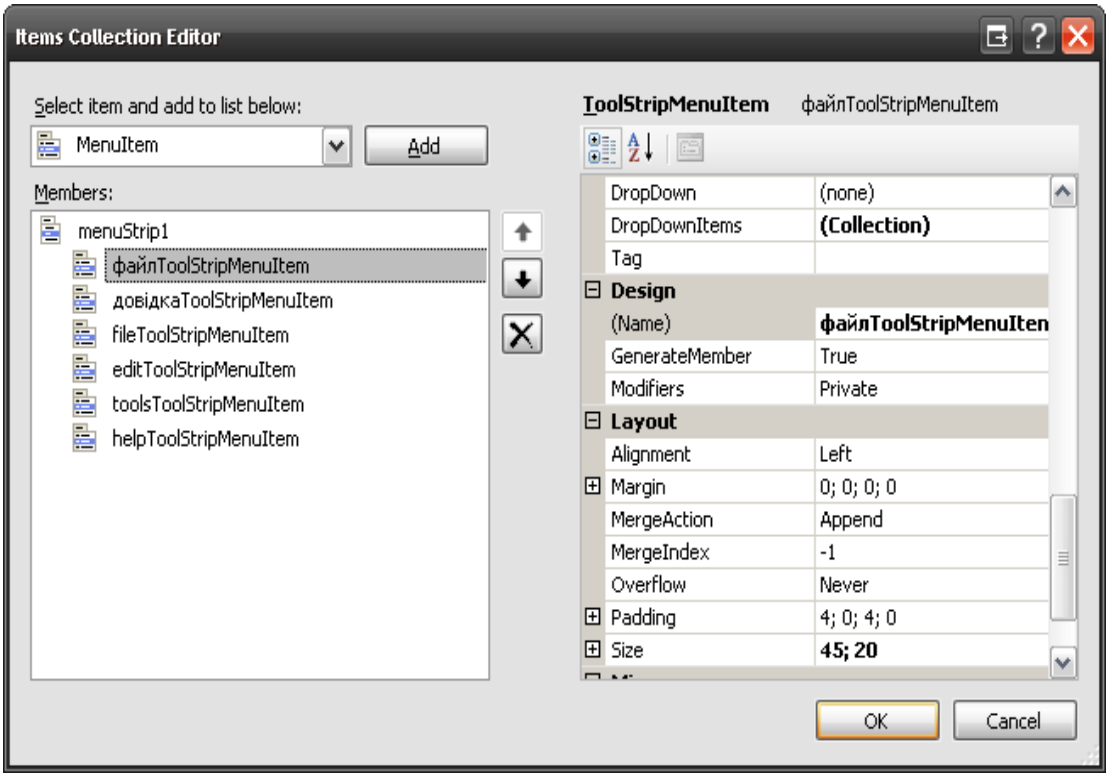


Рисунок 6.6 – Вигляд редактора меню

Також елементи меню можуть бути різних типів. А точніше сказати 4 типів, які саме й показані для обрання на рисунку 6.7:

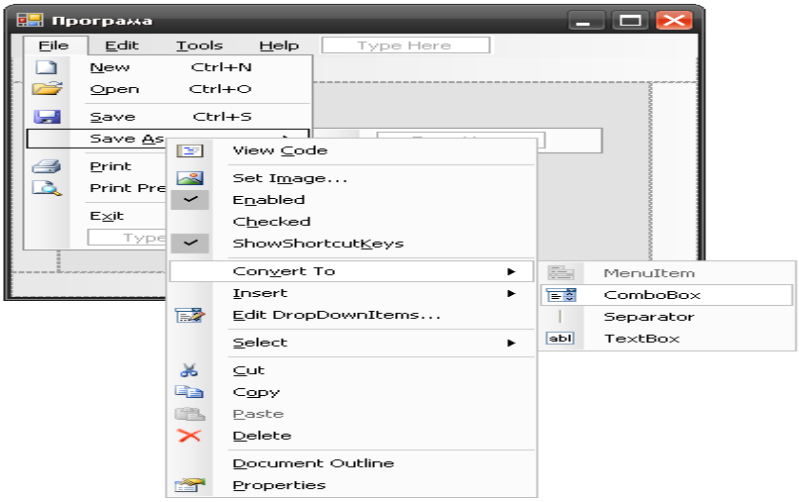


Рисунок 6.7 - Різні типи елементів меню

Для елементів меню також передбачена наявність властивостей і подій. У даному випадку до властивостей варто віднести такі найважливіші:

– Checked (bool) – позначає елемент;
– CheckOnClick (bool) – автоматично позначати під час кліку;
– Enabled (bool) – доступність елементу.

Найголовніша подія меню – реагування на мишу та кнопки клавіатури. Тобто це є якраз Click – виконання дії.

5. Елемент управління “ConextMenuStrip”

Крім головного меню надається змога створення власного контекстного меню. Для цього його слід перетягнути на форму з панелі інструментів та назвати його пункти і задати їх функціональність.

“ConextMenuStrip” за своєю поведінкою схожий до MenuStrip. Головна властивість даного елементу є та, що він може бути пов’язаний з будь-яким елементом управління через властивість ContextMenuStrip.

6. Елемент управління “CheckBox”

Для зрозуміння, який елемент пояснюється, його графічний вигляд приведено на рисунку 6.8:

Рисунок 6.8 - Елемент управління “CheckBox”

Використовується для вибору не виключаючих опцій. Для цього у мові програмування реалізовані такі властивості:

– Appearance (Normal, Button) – може мати вигляд кнопки;
– AutoCheck (bool) – позначати автоматично;
– Checked (bool) – повертає чи встановлює стан;
– ThreeState (bool) – підтримка трьох станів;
– CheckState (Unchecked, Checked, Indeterminate) – повертає чи встановлює стан.
– CheckStateChanged – змінено стан.

7. Елемент управління RadioButton

Даний елемент управління використовується для вибору серед виключаючих опцій. Він є дуже схожим до CheckBox як за поведінкою та властивостями, але не підтримує ThreeState (рис. 6.9).

Рисунок 6.9 - Елемент управління RadioButton

8. Елемент управління CheckedListBox

Призначений для розміщення списку елементів управління CheckBox. Тоді вданому випадку елементи доступні за індексом у списку в ньому при безпосередньому зверенні до них у програмі (рис. 6.10).

Важливою властивостю є Items – містить список елементів.

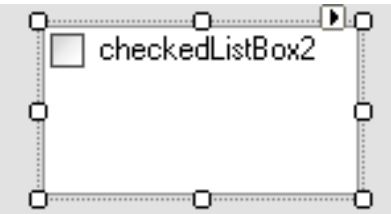


Рисунок 6.10 - Елемент управління CheckedListBox

9. Елемент управління ListBox

Він призначений для виводу на екран списку елементів. Користувач може обирати (виділяти один чи декілька елементів) (рис. 6.11).

Важливі властивості:

– SelectedIndex – містить індекс виділеного елементу (чи -1, якщо жоден елемент не виділено);
– SelectionMode – задає режим виділення елементів.
– Важливі методи:
– GetSelected – перевіряє, чи виділений елемент за індексом;
– SetSelected – виділяє певний елемент за індексом;
– Add, Insert, Clear, Remove – для управління списком елементів.

Можна проглянути приклади коду ініціалізації ListBox та коду інвертування виділених елементів. В даному випадку це демонструють лістинг 6.2, та рисунок 6.11:

Лістинг 6.2 - Приклад - код ініціалізації ListBox

```
// Add items to the ListBox.
            listBox1.Items.Add("A");
            listBox1.Items.Add("C");
            listBox1.Items.Add("E");
            listBox1.Items.Add("F");
            listBox1.Items.Add("G");
            listBox1.Items.Add("D");
            listBox1.Items.Add("B");

            // Sort all items added previously.
```

```
        listBox1.Sorted = true;

        // Set the SelectionMode to select multiple items.
        listBox1.SelectionMode = SelectionMode.MultiExtended;

        // Select three initial items from the list.
        listBox1.SetSelected(0, true);
        listBox1.SetSelected(2, true);
        listBox1.SetSelected(4, true);

        // Force ListBox to scroll back to the top of the list.
              listBox1.TopIndex = 0;


// Loop through all items the ListBox.
for (int x = 0; x < listBox1.Items.Count; x++)
{
     // Determine if the item is selected.
     if (listBox1.GetSelected(x) == true)
          // Deselect all items that are selected.
          listBox1.SetSelected(x, false);
     else
          // Select all items that are not selected.
          listBox1.SetSelected(x, true);
}
// Force the ListBox to scroll back to the top of the list.
listBox1.TopIndex = 0;
```

Рисунок 6.11 – Код інвертування виділених елементів

10. Елемент управління ComboBox

Призначений для вибору однієї опції із переліку доступних. Важливою властивістю є SelectedIndex – індекс виділеного елементу (-1, якщо жодного не виділено) (рис. 6.12).

Рисунок 6.12 - Елемент управління ComboBox

11.11. Елемент управління Label

Призначення – виводити текст (рис. 6.13).

Важливі властивості:

– Text – текст, який виводиться;
– AutoSize – підстроюватися під розмір тексту;
– TextAlign – вирівнювати текст.

Рисунок 6.13 - Елемент управління Label

12.Елемент управління TextBox.

Призначений для зчитування введення з клавіатури (рис. 6.14). Важливими властивостями є:

- Text – текст, який міститься у TextBox;
- CharacterCasing (Normal, Upper, Lower) – чи змінювати реєстр символів;
- PasswordChar (char) – символ, яким будуть замінятись усі символи при вводі;
- MultiLine (bool) – чи можна вводити багато рядків;
- ScrollBars (Horizontal, Vertical, Both) – показувати полоси прокрутки;
- AcceptsReturn (bool) – сприймати натискання Enter;
- TextAlign – яким чином вирівнювати текст.

Рисунок 6.14 – Елемент управління TextBox

ЛК.07 – ДОДАТКОВІ ЕЛЕМЕНТИ УПРАВЛІННЯ ГРАФІЧНОГО ІНТЕРФЕЙСУ КОРИСТУВАЧА

Перелік питань

1. Настроювання переходів по табуляції.
2. Елемент управління MonthCalendar.
3. Елемент управління DateTimePicker.
4. Елемент управління ToolTip.
5. Елемент управління TabControl.
6. Елемент управління TrackBar.
7. Елемент управління Panel.
8. Елемент управління SplitContainer.
9. Елементи управління UpDown.
10. Елемент управління ErrorProvider.

На самостійне вивчення:

1. Побудова MDI-програм [1, С.799-801].

1. Настроювання переходів по табуляції

Розглянемо рисунок 7.1. У ньому створено 4 текстбокси. При необхідному одночасному заповненні даних текст боксів користувачу не зручно використовувати мишу. Тому для переходів передбачається використання табуляції.

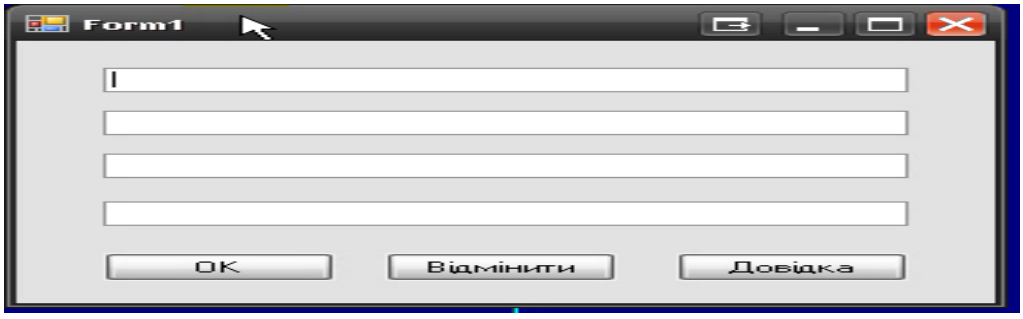


Рисунок 7.1 – Настроювання переходів по табуляції

Для настроювання табуляції необхідно скористатися майстром переходів по табуляції. Також слід задати властивості, які визначають послідовність переходів. В даному випадку можливі 2 варіанти:

- TabStop – чи зупинятись на даному елементі при натисканні Tab;
- TabIndex – номер елементу у послідовності (починаючи з 0, елементи з однаковими номерами обираються у порядку створення).

2. Елемент управління MonthCalendar

Дозволяє обирати дату чи діапазон дат (рис. 7.2). Найважливішими його властивостями є:

- ShowToday – показувати сьогоднішню дату;
- ShowTodayCircle – виділяти сьогоднішню дату;
- ShowWeekNumbers - показувати номери тижнів;
- FirstDayOfWeek – перший день тижня;
- CalendarDimensions – розмір календаря (у кількості місяців, максимум 12);
- BoldedDates, AnnuallyBoldedDates, MonthlyBoldedDates – виділяти певні дати;
- MaxDtate, MinDate – максимальна і мінімальна дати, які може обрати користувач;
- MaxSelectionount – максимальна кількість днів, які можна виділити;
- SelectionRange (Start, End) – зберігає виділений діапазон дат (для однієї дати початок і кінець діапазону збігаються).

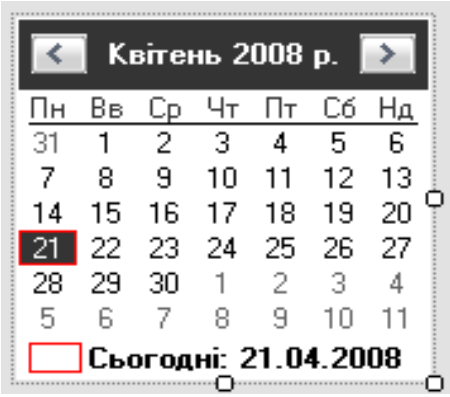


Рисунок 7.2 - Елемент управління MonthCalendar

3. Елемент управління DateTimePicker

Дозволяє обрати дату чи час (рис. 7.3). Найважливішими Властивостями є:

- ShowUpDown – показувати кнопки вгору/вниз замість стрілки вниз;
- Format - спосіб представлення дати чи Time для часу;
- MaxDate, MinDate – максимальна та мінімальна дати;
- ShowCheckBox – показувати CheckBox перед цим елементом управління, змінювати значення дати чи часу можна буде лише тоді, коли CheckBox.Checked == true;
- Value – поточне значення.

Наприклад, для того, щоб використовувати цей елемент управління для вибору часу, необхідно встановити ShowUpDown=true та Format=Time.

Рисунок 7.3 - Елемент управління DateTimePicker

4. Елемент управління ToolTip

Використовується для створення “спливаючих” підказок (рис. 7.4). Він розміщується на формі в області для невізуальних елементів управління. Для

того, щоб зв’язати підказку з елементом управління на формі необхідно задати текст підказки для цього елемента (ця властивість з’являється автоматично).

Рисунок 7.4 - Елемент управління ToolTip

Також можна задавати декілька підказок для одного й того ж об’єкта. Така реалізація демонструється рисунком 7.5:

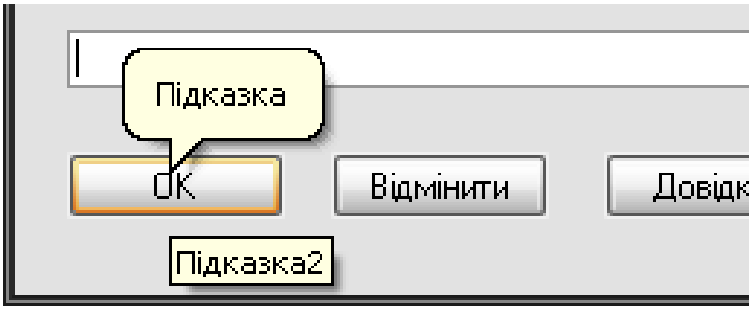


Рисунок 7.5 – Задання декількох підказок

5. Елемент управління TabControl

Використовується для створення інтерфейсу, який складається із декількох вкладок (сторінок). Сторінки задаються у колекції TabPages (рис. 7.6).

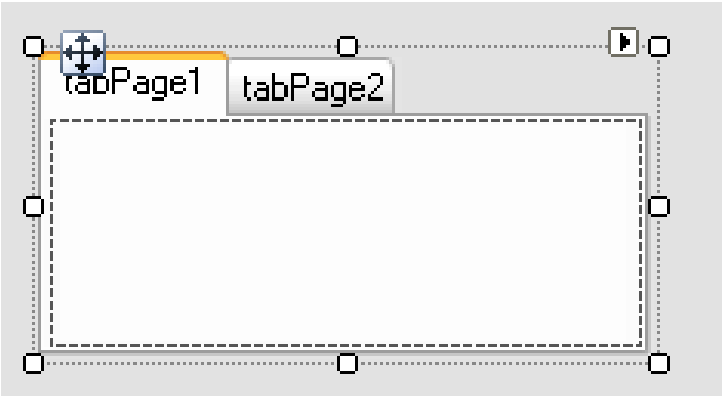


Рисунок 7.6 – Вигляд елементу управління TabControl

Цей елемент управління може також керуватися майстром переходів по табуляції. В даному випадку для кожної сторінки у зручному швидкому переході можна встановлювати властивості, а також додавати чи видаляти сторінки. Дана можливість демонструється на рисунку 7.7, якщо використати контекстне меню заданого елементу управління:

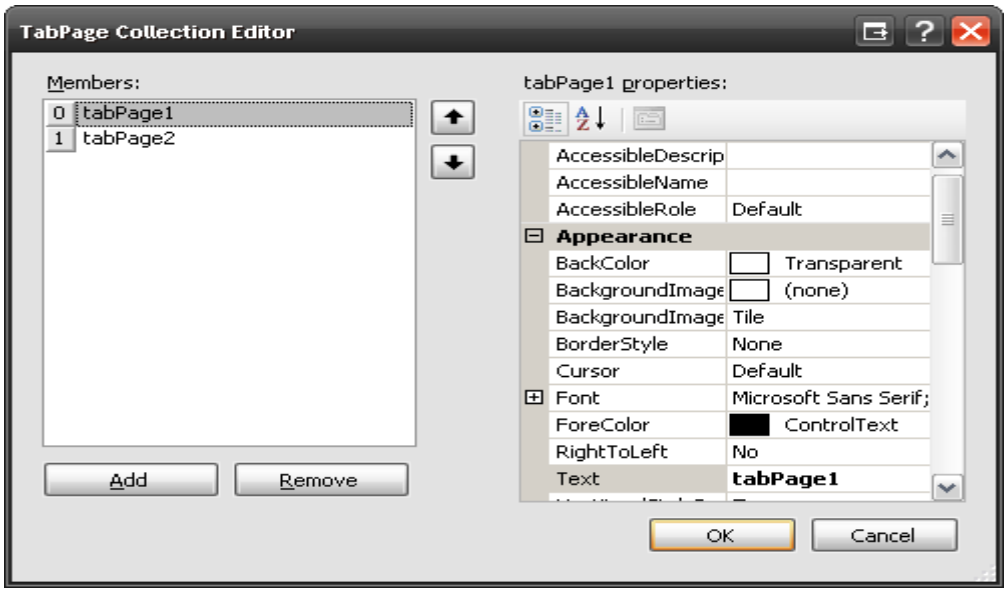


Рисунок 7.7 – Майстер табуляції для TabControl

6. Елемент управління TrackBar

Він собою реалізує «Повзунок» для вибору із діапазону значень (рис. 7.8). Важливими властивостями даного елементу управління є:

– Value – поточне значення;
– TickFrequency – частота міток;
– Minimum, Maximum – мінімальне та максимальне значення;
– SmallChange, LargeChange – зміна значень з клавіатури (клавішами управління курсором та Page Up/Down);
– Orientation – горизонтальне чи вертикальне розміщення.

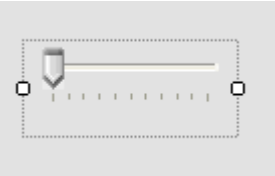


Рисунок 7.8 - Елемент управління TrackBar

7. Елемент управління Panel

Використовується для логічного групування елементів управління (рис. 7.9). За рахунок зміни властивості Visible дозволяє приховувати групи елементів управління, розміщених на панелі.

Рисунок 7.9 - Елемент управління Panel

8. Елемент управління SplitContainer

Представляє собою дві панелі, для яких можна змінювати область, яку займає одна та інша під час виконання програми (рис. 7.10).

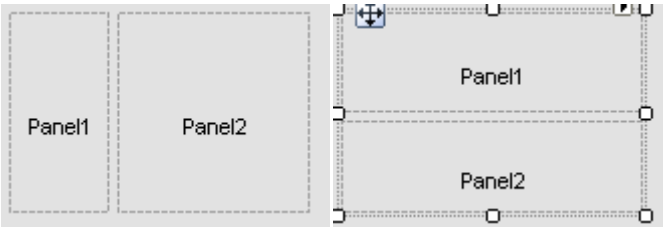


Рисунок 7.10 - Елемент управління SplitContainer

9. 9. Елементи управління UpDown

Для вибору серед набору значень чи введення чисел шляхом перебору значень є два елементи управління: DomainUpDown та NumericUpDown (рис. 7.11).

Для NumericUpDown властивість Value містить поточне значення.

Для DomainUpDown можливі варіанти необхідно задати до властивості Items, поточне значення доступне через SelectedItem чи SelectedIndex.

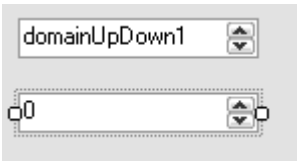


Рисунок 7.11 - Елементи управління UpDown

10.Елемент управління ErrorProvider

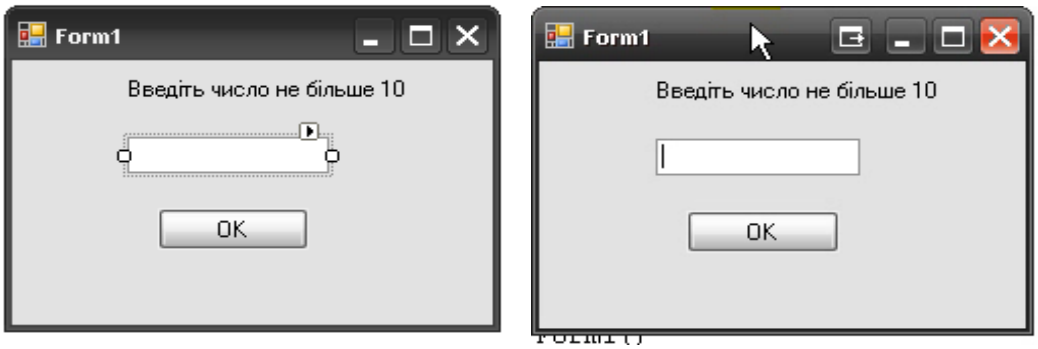
Використовується для перевірки введення інформації користувачем. Фактично є обробником помилки в коді програми та формі (рис. 7.12). Важливими властивостями можна назвати:

– AlwaysBlink – мигання завжди при помилці;
– BlinkIfDifferentError – мигання, якщо з’являється нова помилка;
– NeverBlink – відключити мигання.

Для того, щоб реалізувати його дію, необхідно знати, що даний обробник помилки з елементом управління пов’язується у програмному коді за рахунок використання властивості Validating.

Рисунок 7.12 - Елемент управління ErrorProvider

Приклад використання даного елементу управління можна показати на простій помилці введення числа більше 10 у текст бокс. Цедемонструє рисунок 7.13:




```
private void textBox1_Validating(object sender, CancelEventArgs e)
{
    if (int.Parse(textBox1.Text) > 10)
    {
        errorProvider1.SetError(textBox1, "Не більше 10!");
    }
}
```

Рисунок 7.13 - Приклад використання ErrorProvider

ЛК.08 – ПРОГРАМУВАННЯ ІНТЕРФЕЙСУ СУБД НА ОСНОВІ ADO.NET

Перелік питань:

1. База даних.
2. Основи реляційної СУБД.
3. Основні принципи роботи з СУБД.
4. Введення до технології ADO.NET.
5. Створення найпростішої програми для роботи з СУБД.
6. Послідовність команд для читання даних з БД.
7. Використання класу SqlConnection.
8. Використання класу SqlConnectionStringBuilder.
9. Файли конфігурації.
10. Робота з об’єктами читання даних.
11. Зміна вмісту таблиць за допомогою об’єктів команд.
12. Робота з об’єктами параметризованих команд.

На самостійне вивчення:

1. Виконання збережених процедур за допомогою DbCommand [1, С.965-967].

1. База даних

База даних (БД) – структурований організованих набір даних, який використовується для опису характеристик певних систем. Головне завдання бази даних – надійне збереження даних та надання доступу до них користувачам та прикладним програмам.

Система управління базами даних (СУБД) – програмне забезпечення, яке забезпечує доступ до даних у БД та надає можливість додавання, оновлення, видалення, пошук та ін. операції по роботі з даними. Основними характеристиками СУБД є:

- контроль за надлишковістю даних;

– непротирічність даних;
– підтримка цілісності бази даних (коректність та непротирічність);
– цілісність описується за допомогою обмежень;
– незалежність прикладних програм від даних;
– спільне використання даних;
– підвищений рівень безпеки.

2. Основи реляційної СУБД

Реляційна база даних – це тип СУБД, який оснований на двовимірних таблицях, зв'язаних між собою. Стовпчики називаються атрибутами, рядки – кортежами. Домени – це типи даних, які присвоюються атрибутам. Кількість атрибутів визначає ступінь таблиці , а рядків – її кардинальність.

Приклад реляційної СУБД демонструє рисунок 8.1:

BIN#	WINE	PRODUCER	YEAR	BOTTLES	READY
2	Chardonnay	Buena Vista	1997	1	1999
3	Chardonnay	Geyser Peak	1997	5	1999
6	Chardonnay	Simi	1996	4	1998
12	Joh. Riesling	Jekel	1998	1	1999
21	Fume Blanc	Ch. St. Jean	1997	4	1999
22	Fume Blanc	Robt. Mondavi	1996	2	1998
30	Gewurztraminer	Ch. St. Jean	1998	3	1999
43	Cab. Sauvignon	Windsor	1991	12	2000
45	Cab. Sauvignon	Geyser Peak	1994	12	2002
48	Cab. Sauvignon	Robt. Mondavi	1993	12	2004

Рисунок 8.1 – Реляційна СУБД

Графічно СУБД можна представити як проміжну ланку між взаємодією серверів з користувачами, де міститься доступна їм інформація, що зберігається довільним чином, але в цілісному вигляді (рис. 8.2).

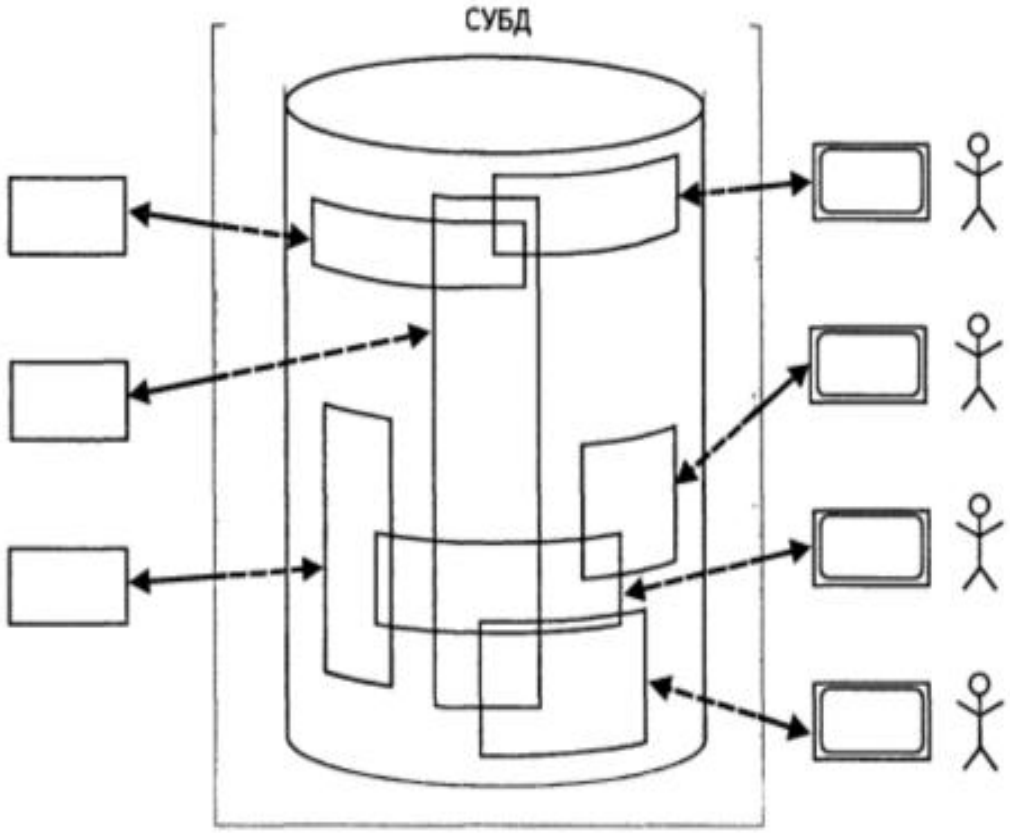


Рисунок 8.2 – Представлення СУБД

Терміни, які використовуються у базах даних для опису таблиці, були розглянуті раніше. Їх взаємодія та відображення на таблиці для прикладу можна подивитися на рисунку 8.3:

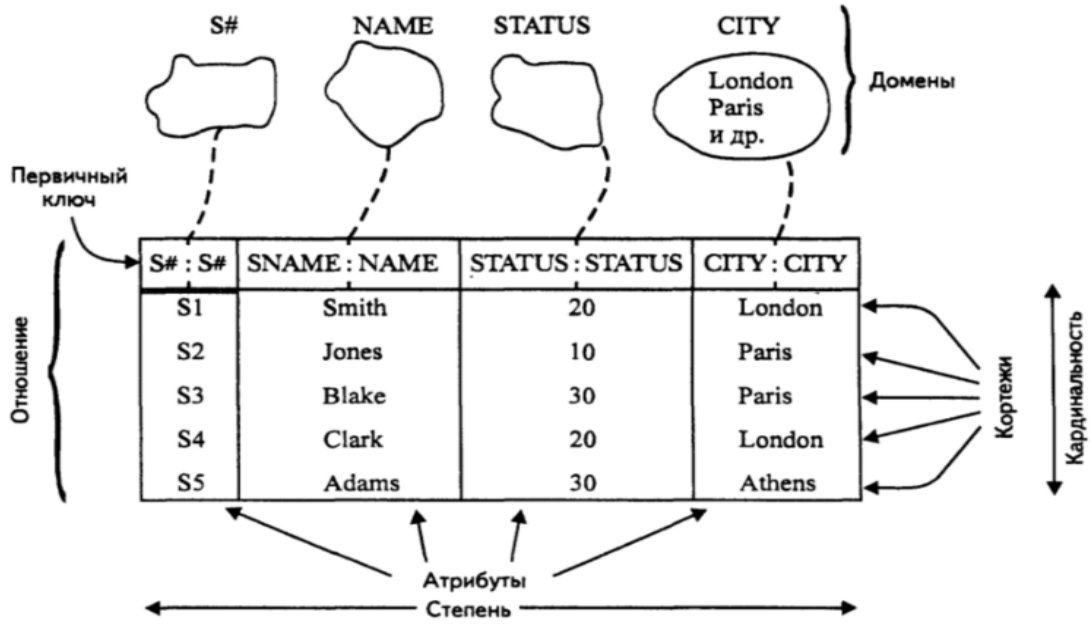


Рисунок 8.3 – Терміни опису структури даних

Можна створювати зв'язок між таблицями. По-перше, кожна окрема таблиця в базі даних повинна мати власний первинний ключ. Первинний ключ – це унікальний ідентифікатор рядків (кортежів) таблиці. Він може бути природнім або штучним. І використовується для посилання на кортеж інших таблиць.

Існує також зовнішній (вторинний) ключ – це посилання на первинний ключ даної таблиці з іншої (первинної).

Приклад створення зв’язку за 2 стовпцями 2 таблиць демонструє рисунок 8.4, де стовпчик «Група» головної таблиці доповнюється інформацією в підлеглій таблиці:

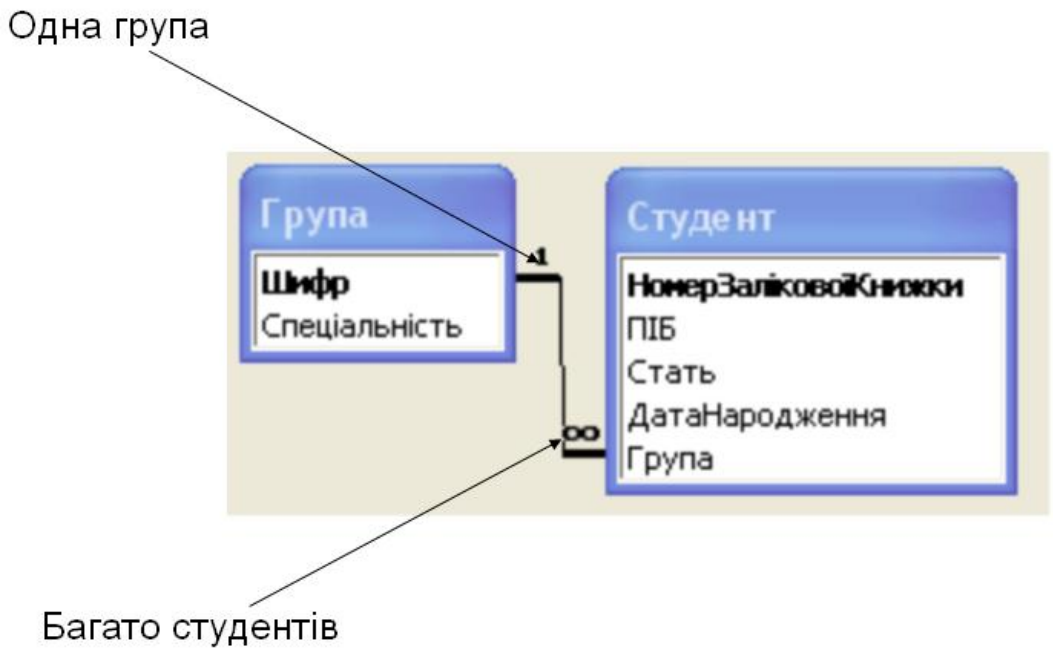


Рисунок 8.4 - Зв’язок між таблицями

3. Основні принципи роботи з СУБД

Щоб працювати з певною програмою, механізмом чи пристроєм, потрібно знати основні принципи його роботи. СУБД також мають свої принципи роботи, які варто виділити:

– використання уніфікованої мови SQL для доступу до даних;
– перманентність даних;

- розмежування доступу;
- контроль правильності введення даних;
- атомарність операцій (використання транзакцій);
- забезпечення одночасної роботи різних користувачів;
- незалежність від конкретної СУБД;
- оптимізація швидкості операцій з даними;
- здатність працювати з великими обсягами даних (які неможливо одночасно завантажити до оперативної пам’яті);
- можливість виконувати роботу у оффлайновому режимі;
- нормалізація даних;
- багаторівнева архітектура, необхідність поділу бізнес-логіки на ту, яка виконується на сервері, та ту, яка виконується на клієнті.

4. Введення до технології ADO.NET

Необхідно проаналізувати об’єктну модель ADO.NET. Вона розглянута на рисунку 8.5. В даному випадку .NET Data Provider – постачальник даних.

Компоненти ADO.Net спроектовані таким чином, щоб відділити доступ до даних від маніпуляцій із даними. Два центральні компоненти ADO.Net виконують це завдання: клас DataSet та провайдер даних, який являє собою набір компонентів, що включають об’єкти Connection, Command, DataReader та DataAdapter.

Клас DataSet є центральним компонентом для від’єднаної (disconnected) архітектури ADO.Net. Цей клас спроектовано для доступу до даних незалежно від джерела даних. В результаті він може бути використаний багатьма різними джерелами даних, з XML-файлами, або для роботи з локальними даними програми. DataSet містить колекцію з одного або більше об’єктів DataTable, які складаються з рядків та колонок із даними, а також первинних ключів, зовнішніх ключів, обмежень (constraint), та інформації про зв’язки між даними.

Іншим базовим компонентом архітектури ADO.Net є провайдер даних .Net, компоненти якого спроектовані для маніпуляцій з даними та швидкого доступу до даних для читання. Об’єкт Connection представляє під’єднання до джерела даних. Об’єкт Command надає доступ до команд бази даних для читання даних, модифікації та запуску збережених процедур. DataReader представляє високопродуктивний потік даних з джерела даних. І нарешті DataAdapter є мостом між об’єктом DataSet та джерелом даних. Він використовує об’єкти Command для виконання SQL-запитів до джерела даних як для завантаження даних у DataSet, так і для повернення змін у даних назад у джерело даних.

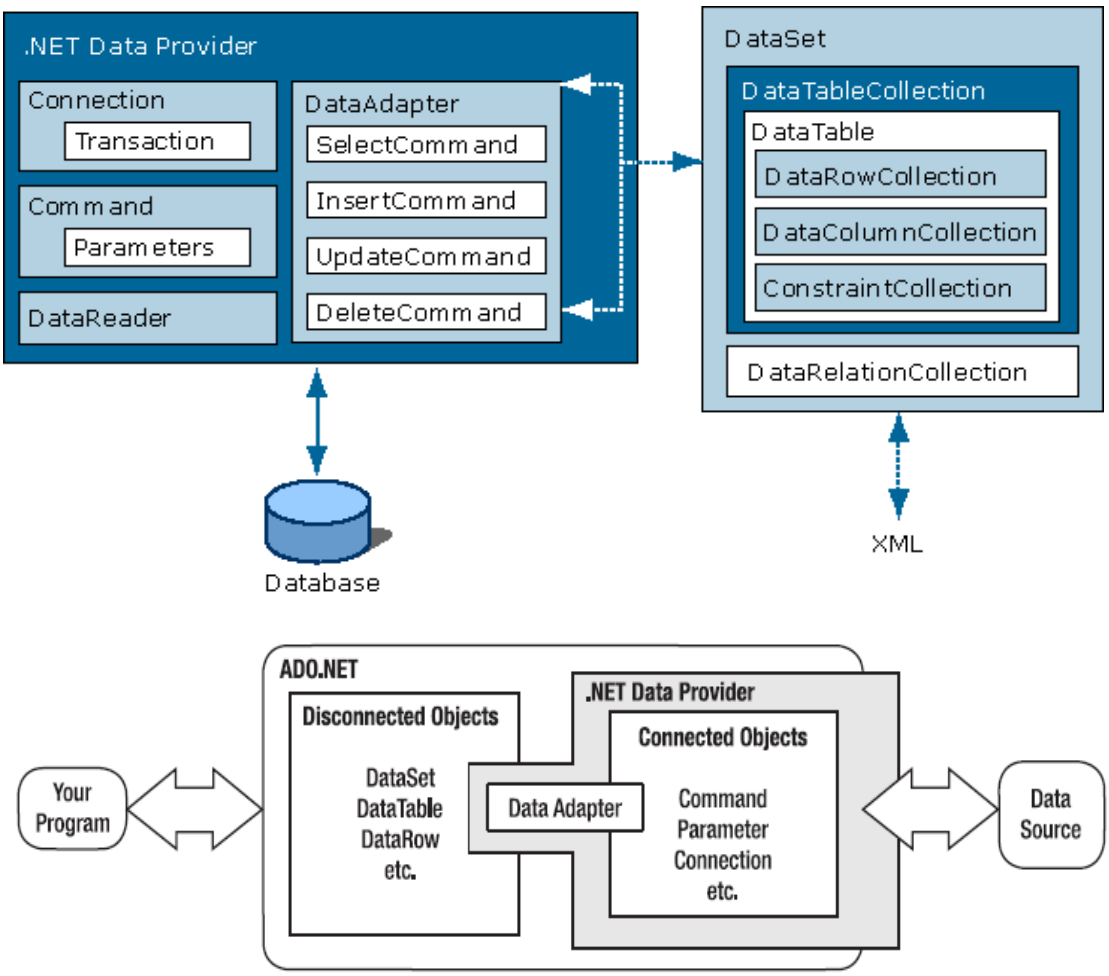


Рисунок 8.5 - Об’єктна модель ADO.NET

Відповідно для різних мов програмування створюються окремі провайдери доступу даних. Це демонструє рисунок 8.6:

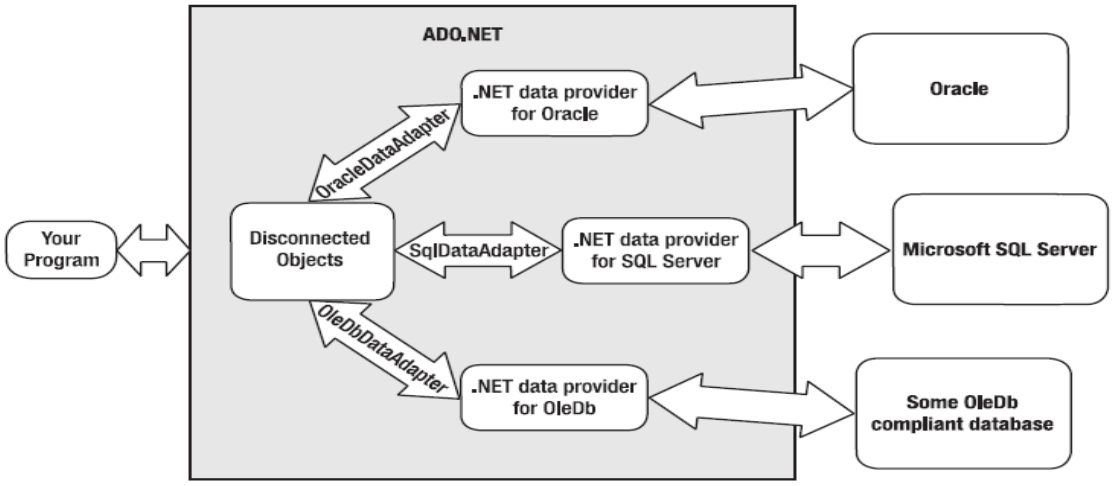


Рисунок 8.6 – Провайдери доступу даних

5. Створення найпростішої програми для роботи з СУБД

Створення найпростішої програми для роботи з СУБД у Visual Studio може бути здійснено без написання програмного коду (відео).

6. Послідовність команд для читання даних з БД

Для того, щоб зчитати дані з бази даних, необхідно виконати таку послідовність команд:

а) Створити об’єкт, який зберігатиме з’єднання:

SqlConnection testConnection = new SqlConnection(connectionString);

б) Створити об’єкт, який зберігатиме запит:

SqlCommand testCommand = testConnection.CreateCommand() ;

testCommand.CommandText = "Select DemoValue from Demo where DemoID = 1" ;

в) Відкрити з’єднання:

testConnection.Open() ;

г) Виконати запит і отримати результат:

string result = (string)testCommand.ExecuteScalar() ;

д) Закрити з’єднання:

testConnection.Close();

7. Використання класу SqlConnection

Лістинг 8.1 демонструє приклад і правила використання даного класу і перевірку стану підключення:

```
SqlConnection testConnection =
            new SqlConnection(
            "Data Source=(local);Initial Catalog=Test;Integrated
Security=SSPI");
```

Продовження лістингу 8.1:

```
 SqlConnection testConnection = new SqlConnection();
            string testConnectionString =
            "Data Source=(local);Initial Catalog=Test;Integrated
Security=SSPI";
            testConnection.ConnectionString = testConnectionString;
static void Main(string[] args)
        {
            SqlConnection testConnection = new SqlConnection(args[0]);
            try
            {
                testConnection.Open();
                if (testConnection.State == ConnectionState.Open)
                {
                    Console.WriteLine("Successfully opened a connection");
                }
            }
            catch (Exception)
            {
                if (testConnection.State != ConnectionState.Open)
                {
                    Console.WriteLine("Failed to open a connection");
                }
            }
            finally
            {
                // Closing a connection ensures connection pooling.
                if (testConnection.State == ConnectionState.Open)
                {
                    testConnection.Close();
                }
                testConnection.Dispose();
            }
        }
```

8. Використання класу SqlConnectionStringBuilder.

Правила використання даного класу наводяться в лістингу 8.2:

Лістинг 8.2

```
static void Main(string[] args)
        {
            SqlConnectionStringBuilder connstrBuilder = new
SqlConnectionStringBuilder();
            connstrBuilder.DataSource = "(local)";
            connstrBuilder.InitialCatalog = "Test";
            connstrBuilder.IntegratedSecurity = true;
            using (SqlConnection testConnection =
            new SqlConnection(connstrBuilder.ToString()))
            {
                try
                {
                    testConnection.Open();
                    if (testConnection.State == ConnectionState.Open)
                    {
```

Продовження лістингу 8.2:

```
                        Console.WriteLine("Connection successfully opened");
                        Console.WriteLine("Connection string used: " +
                        testConnection.ConnectionString);
                    }
                }
                catch (Exception)
                {
                    if (testConnection.State != ConnectionState.Open)
                    {
                        Console.WriteLine("Connection open failed");
                        Console.WriteLine("Connection string used: "
                        + testConnection.ConnectionString);
                    }
                }
            }
          Console.WriteLine("Press any key to continue ..");
            Console.Read();
          }
```

9. Файли конфігурації

На стороні клієнта можна використовувати файл *.config, в якому в рамках елементу <appSettings> можна вказувати послідовні пари ключів та значень (рис. 8.7).

```
<configuration>
   <appSettings>
      <add key="provider" value="SqlServer" />
      <add key="cnStr" value=
          "Data Source=localhost;uid=sa;pwd=;Initial Catalog=Pubs"/>
   </appSettings>
</configuration>
```

Рисунок 8.7 – Приклад використання файлів конфігурації

Щоб реалізувати доступ до файлу конфігурації, необхідно звикористати доступ до вмісту файлу конфігурації (здійснюється за допомогою збірки System.Configuration.dll та простору імен System.Configuration). Приклад такої реалізації показано на рисунку 8.8:

```
static void Main(string[] args)
{
   // Чтение значения ключа provider.
   string dpStr = ConfigurationManager.AppSettings["provider"];
   DataProvider dp =
      (DataProvider)Enum.Parse(typeof(DataProvider), dpStr);

   // Чтение значения cnStr.
   string cnStr = ConfigurationManager.AppSettings["cnStr"];

   // Получение соединения.
   IDbConnection myCn = GetConnection(dp);
   myCn.ConnectionString = cnStr;
...
}
```

Рисунок 8.8 – Доступ до файлу конфігурації

10.Робота з об’єктами читання даних

Для цього необхідно знати певні положення, що забезпечують таку функціональність бази даних:

Об’єкти читання даних (наприклад, DbDataReader) забезпечують простий і швидкий спосіб отримання інформації із БД.

Об’єкти читання даних створюють односпрямований та доступний лише для читання потік даних, який повертає лише по одному запису за один раз.

Ці об’єкти підтримують з’єднання відкритим до тих пір, доки явно не була отримана команда закрити його.

Вони є найзручніші у ситуаціях, коли необхідно запросити великий обсяг даних, який у іншому випадку міг би не поміститися до оперативної пам’яті. Наприклад, запросити 1 млн. записів із БД та зберегти їх до локального файлу.

Виконання об’єкту читання даних здійснюється за допомогою методу ExecuteReader(). Додатково у якості параметру можна передати інструкцію автоматично закрити з’єднання за допомогою CommandBehavior.CloseConnection.

Приклад такого використання обєкту читання даних реалізується через код на рисунку 8.9:

```
static void Main(string[] args)
{
...
   // Получение объекта чтения данных в стиле ExecuteReader().
   SqlDataReader myDataReader;
   myDataReader =
      myCommand.ExecuteReader(CommandBehavior.CloseConnection);

   // Цикл по результирующему набору.
   while (myDataReader.Read())
   {
     Console.WriteLine("-> Марка - {0}, название - {1}, цвет - {2}.",
       myDataReader ["Make"].ToString().Trim(),
       myDataReader["PetName"].ToString().Trim(),
       myDataReader["Color"].ToString().Trim());
   }
   myDataReader.Close();
   ShowConnectionStatus(cn);
}
```

Рисунок 8.9 - Приклад використання об’єкту читання даних

11.Зміна вмісту таблиць за допомогою об’єктів команд

Метод ExecuteNonQuery() для об’єкту sqlCommand дозволяє здійснювати виконання SQL-команди, які не повертають значення, а дозволяють змінити дані у БД (рис. 8.10):

```
private static void InsertNewCar(SqlConnection cn)
{
    // Сбор информации о новой машине.
    Console.Write("Введите номер машины: ");
    int newCarID = int.Parse(Console.ReadLine());
    Console.Write("Введите марку: ");
    string newCarMake = Console.ReadLine();
    Console.Write("Введите цвет: ");
    string newCarColor = Console.ReadLine();
    Console.Write("Введите название: ");
    string newCarPetName = Console.ReadLine();
    // Создание и выполнение оператора SQL.
    string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "('{0}', '{1}', '{2}', '{3}')", newCarID, newCarMake,
        newCarColor, newCarPetName);
    SqlCommand cmd = new SqlCommand(sql, cn);
    cmd.ExecuteNonQuery();
}
```

Рисунок 8.10 – Приклад зміни вмісту таблиць

12.Робота з об’єктами параметризованих команд

При роботі з параметризованими командами необхідно використовувати тип DbParameter.

Тип DbParameter дозволяє створювати параметризовані команди, які отримують параметри певного типу, що не є жорстко закодовані у програмному коді. Для sql-запитів використовується наслідуваний тип sqlParameter.

Основні ключові члени цього типу показано на рисунку 8.11:

Свойство	Описание
DbType	Читает или записывает информацию о "родном" типе данных для источника данных, представленную в виде соответствующего типа данных CLR
Direction	Читает или записывает значение, указывающее направление потока для данного параметра (только ввод, только вывод, двунаправленное движение, предусмотренное возвращение значения)
IsNullable	Читает или записывает значение, являющееся индикатором того, что параметр допускает значения null
ParameterName	Читает или устанавливает имя DbParameter
Size	Читает или устанавливает максимальный размер данных параметра
Value	Читает или устанавливает значение параметра

Рисунок 8.11 – Члени типу DbParameter

Приклад такого використання програмного коду з визначенням та подальшим заданням параметрів демонструють рисунки 8.12, 8.13:

```
private static void InsertNewCar(SqlConnection cn)
{
...
   // Обратите внимание на 'заполнители' в SQL-запросе.
   string sql = string.Format("Insert Into Inventory" +
      "(CarID, Make, Color, PetName) Values" +
      "(@CarID, @Make, @Color, @PetName)");
```

Рисунок 8.12 – Задання sqlParameter

```
// Наполнение коллекции параметров.
SqlCommand cmd = new SqlCommand(sql, cn);
SqlParameter param = new SqlParameter();
param.ParameterName = "@CarID";
param.Value = newCarID;
param.SqlDbType = SqlDbType.Int;
cmd.Parameters.Add(param);

param = new SqlParameter();
param.ParameterName = "@Make";
param.Value = newCarMake;
param.SqlDbType = SqlDbType.Char;
param.Size = 20;
cmd.Parameters.Add(param);
```

```
    param = new SqlParameter();
    param.ParameterName = "@PetName";
    param.Value = newCarPetName;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 20;
    cmd.Parameters.Add(param);
    cmd.ExecuteNonQuery();
}
```

Рисунок 8.13 - Приклад використання sqlParameter

ЛК.09 – ЕЛЕМЕНТИ УПРАВЛІННЯ ADO.NET

Перелік питань

1. Незв’язний рівень ADO.NET та тип DataSet.
2. Робота з DataColumn.
3. Робота з DataRow.
4. Робота з DataTable.
5. Збереження DataSet та DataTable в форматі XML.
6. Прив’язка DataTable до інтерфейсу користувача.
7. Робота з типом DataView.
8. Робота з адаптерами даних.
9. Об’єкти DataSet з багатьма таблиць та об’єкти DataRelation.

На самостійне вивчення:

1. Використання майстрів даних [1, С.1002-1005].

1. Незв’язний рівень ADO.NET та тип DataSet

У доповнення до роботи з об’єктами з’єднання, команд і читання даних, які виконуються на основі відкритого під’єднання до СУБД (так званий зв’язний рівень роботи з даними), технологія ADO.NET підтримує незв’язний рівень роботи з даними, який реалізується через спеціальний адаптер даних (DbDataAdapter), який, в свою чергу, використовує об’єкти DataSet, що являються контейнерами і можуть містити довільну кількість об’єктів DataTable, кожен із яких містить колекцію об’єктів DataRow та DataColumn.

Свою назву даний рівень отримав за рахунок того, що з’єднання з СУБД не є постійним і розривається у той час, коли клієнтська сторона отримує необхідні дані (рис. 9.1).

Рисунок 9.1 – аємодія клієнта з базою даних

Тип DataSet (рис. 9.2) являє собою представлення зовнішніх даних в пам’яті, зберігаючи три внутрішні строго типізовані колекції:

– DataTableCollection – таблиці;
– DataRelationCollection – зв’язки;
– PropertyCollection – додаткові властивості.

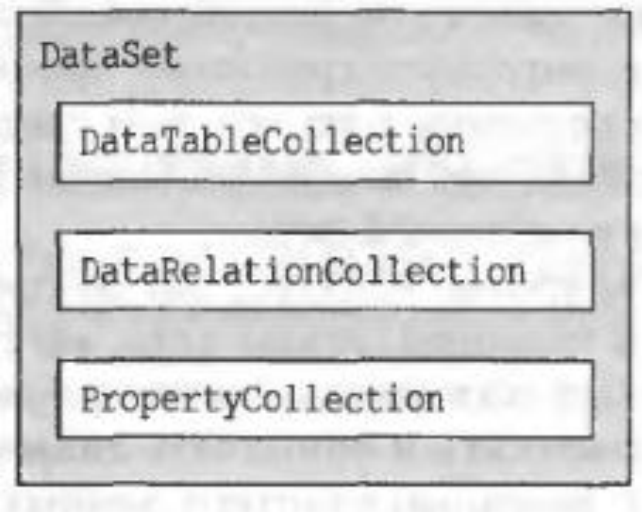


Рисунок 9.2 - Тип DataSet

Кожен тип має свої властивості. Не виключенням є і тип DataSet. Його основні властивості з описом дії представлені на рисунку 9.3. При цьому можлива чутливість до регістру символів, задання імені таблиці, перевірка помилок чи обмежень тощо.

Свойство	Описание
CaseSensitive	Индикатор чувствительности к регистру символов при сравнении строк в объектах DataTable
DataSetName	Представляет понятное имя данного объекта DataSet. Обычно это значение устанавливается с помощью параметра конструктора
EnforceConstraints	Получает или устанавливает значение, являющееся индикатором необходимости применения заданных ограничений при любой операции обновления
HasErrors	Получает значение, являющееся индикатором наличия ошибок в любой из строк объектов DataTable для объекта DataSet
RemotingFormat	Новое свойство .NET 2.0, позволяющее указать, как должна выполняться сериализация DataSet (в двоичном или XML-формате) для слоя удаленного взаимодействия .NET

Рисунок 9.3 - Важливі властивості DataSet

Також важливі методи DataSet перечислені та описуються на рисунку 9.4, включаючи як роботу з таблицями, так і потоками.

Методы	Описание
AcceptChanges()	Фиксирует все изменения, сделанные в данном объекте DataSet с момента его загрузки или последнего вызова AcceptChanges()
Clear()	Выполняет полную очистку данных DataSet путем удаления всех строк в каждом объекте DataTable
Clone()	Клонирует структуру DataSet, включая все объекты DataTable, а также все отношения и ограничения
Copy()	Копирует и структуру, и данные для имеющегося объекта DataSet
GetChanges()	Возвращает копию DataSet, содержащую все изменения, сделанные со времени последней загрузки или последнего вызова AcceptChanges()
GetChildRelations()	Возвращает коллекцию дочерних связей для указанной таблицы
GetParentRelations()	Возвращает коллекцию родительских связей для указанной таблицы
HasChanges()	Перегруженный метод, который возвращает значение, являющееся индикатором наличия модификаций у DataSet, учитывая новые, удаленные или измененные строки
Merge()	Перегруженный метод, который выполняет слияние данного объекта DataSet с указанным объектом DataSet
ReadXml() ReadXmlSchema()	Позволяют считывать XML-данные из действительного потока (файлового, размещенного в памяти или сетевого) в DataSet
RejectChanges()	Выполняет откат всех изменений, сделанных в DataSet с момента его создания или последнего вызова DataSet.AcceptChanges()
WriteXml() WriteXmlSchema()	Позволяют записать содержимое DataSet в действительный поток

Рисунок 9.4 - Важливі методи DataSet

Кодовий приклад використання DataSet зображено на рисунку 9.5:

```
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("***** Забавы с DataSet *****\n");
      // Создание объекта DataSet.
      DataSet carsInventoryDS = new DataSet("Inventory из Car");
      carsInventoryDS.ExtendedProperties["TimeStamp"] =
         DateTime.Now;
      carsInventoryDS.ExtendedProperties["Company"] =
         "Intertech Training";
   }
}
```

Рисунок 9.5 - Приклад використання DataSet

2. Робота з DataColumn

Тип DataColumn – окремий стовбець в базі даних, їх набір формує структуру таблиці. Серед найважливіших властивостей дадного типу є (рис. 9.6, 9.7).

Свойства	Описание
AllowDBNull	Индикатор того, что строка в этом столбце может содержать значение null. Значением по умолчанию является true
AutoIncrement AutoIncrementSeed AutoIncrementStep	Используются для настройки автоприращения для данного столбца, когда нужно гарантировать уникальность значений в данном объекте DataColumn (например, для первичного ключа). По умолчанию в DataColumn автоприращение не выполняется
Caption	Читает или устанавливает текст заголовка, который должен отображаться для данного столбца (например, текст, который увидит конечный пользователь в DataGridView)
ColumnMapping	Определяет представление DataColumn при сохранении DataSet в виде XML-документа с помощью метода DataSet.WriteXml()
ColumnName	Читает или устанавливает имя столбца в коллекции Columns (т.е. его представление в DataTable). Если не установить ColumnName явно, значением по умолчанию будет Column с номером столбца (т.е. Column1, Column2, Column3 и т.д.)
DataType	Определяет тип данных, хранимых в столбце (логический, строковый, числовой с плавающей точкой и т.д.)
DefaultValue	Читает или устанавливает значение, которое должно приписываться по умолчанию для данного столбца при вставке новых строк. Это значение используется тогда, когда не указано иное

Рисунок 9.6 - Властивості DataColumn

Свойства	Описание
Expression	Читает или устанавливает выражение, используемое для фильтрации строк, вычисления значений столбца или создания агрегированных столбцов
Ordinal	Возвращает числовую позицию столбца в коллекции Columns, поддерживаемой объектом DataTable
ReadOnly	Индикатор запрета изменения содержимого столбца после добавления строки в таблицу. Значением по умолчанию является false
Table	Возвращает объект DataTable, содержащий данный объект DataColumn
Unique	Индикатор требования уникальности значений в данном столбце. Если столбцу назначается ограничение первичного ключа, свойству Unique должно быть назначено значение true

Рисунок 9.7 – Додаткові властивості DataColumn

Приклад створення DataColumn безпосередньо програмним кодом показано на рис. 9.8:

```
static void Main(string[] args)
{
...
   // Создание объектов DataColumn, отображающих 'реальные'
   // столбцы таблицы Inventory из базы данных Cars.
   DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
   carIDColumn.Caption = "Номер";
   carIDColumn.ReadOnly = true;
   carIDColumn.AllowDBNull = false;
   carIDColumn.Unique = true;

   DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
   DataColumn carColorColumn = new DataColumn("Color",
      typeof(string));
   DataColumn carPetNameColumn = new DataColumn("PetName",
      typeof(string));
   carPetNameColumn.Caption = "Название";
}
```

Рисунок 9.8 - Приклад створення DataColumn

Можливе використання автоінкременту (приросту) для полів. Це реалізується за допомогою властивостей AutoIncrement, AutoIncrementSeed, AutoIncrementStep (рис. 9.9):

```
static void Main(string[] args)
{
...
   DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
   carIDColumn.ReadOnly = true;
   carIDColumn.Caption = "Номер";
   carIDColumn.AllowDBNull = false;
   carIDColumn.Unique = true;
   carIDColumn.AutoIncrement = true;
   carIDColumn.AutoIncrementSeed = 0;
   carIDColumn.AutoIncrementStep = 1;
}
```

Рисунок 9.9 - Дозволення автоінкременту для полів.

Тепер необхідно додати DataColumn в DataTable. На практиці тип DataColumn використовується у складі об’єктів типу DataTable (рис. 9.10):

```
static void Main(string[] args)
{
...
 // Добавление DataColumn в DataTable.
 DataTable inventoryTable = new DataTable("Inventory");
 inventoryTable.Columns.AddRange(new DataColumn[]
   { carIDColumn, carMakeColumn, carColorColumn, carPetNameColumn });
}
```

Рисунок 9.10 - Додання DataColumn в DataTable

3. Робота з DataRow

Тип DataRow відповідає за рядок даних в таблиці, їх колекція, фактично, становить набір даних таблиці. До основних членів цього типу відносять такі (рис. 9.11).

Члены	Описание
HasErrors GetColumnsInError() GetColumnError() ClearErrors() RowError	Свойство HasErrors возвращает булево значение, являющееся индикатором наличия ошибок. В этом случае можно использовать метод GetColumnsInError(), чтобы получить информацию о членах, порождающих проблемы, метод GetColumnError(), чтобы получить описание ошибки, и метод ClearErrors(), удаляющий ошибки для данной строки. Свойство RowError позволяет задать текстовое описание ошибки для данной строки
ItemArray	Свойство, возвращающее или устанавливающее значения для данной строки с помощью массива объектов
RowState	Свойство, используемое для выяснения текущего "состояния" DataRow с помощью значений из перечня RowState
Table	Свойство, используемое для получения ссылки на DataTable, содержащий данный объект DataRow
AcceptChanges() RejectChanges()	Эти методы, соответственно, фиксируют или отвергают все изменения, сделанные в данной строке с момента последнего вызова AcceptChanges()
BeginEdit() EndEdit() CancelEdit()	Эти методы, соответственно, начинают, завершают или отменяют операции редактирования для объекта DataRow.
Delete()	Метод, помечающий данную строку для удаления при вызове метода AcceptChanges()
IsNull()	Метод, возвращающий значение-индикатор того, что данный столбец содержит значение null

Рисунок 9.11 - Робота з DataRow

Робота з DataRow відрізняється від DataColumn тим, що ці об’єкти не створюються напряму, а через метод DataTable (рис. 9.12).

```
static void Main(string[] args)
{
...
    // Добавление строк в таблицу Inventory.

    DataRow carRow = inventoryTable.NewRow();
    carRow["Make"] = "BMW";
    carRow["Color"] = "черный";
    carRow["PetName"] = "Hamlet";
    inventoryTable.Rows.Add(carRow);

    carRow = inventoryTable.NewRow();
    carRow["Make"] = "Saab";
    carRow["Color"] = "красный";
    carRow["PetName"] = "Sea Breeze";
    inventoryTable.Rows.Add(carRow);
}
```

Рисунок 9.12 - Створення DataRow

DataRow має властивість RowState. Дана властивість необхідна для ідентифікації рядків таблиці за певним станом (створений, змінений і т.д.)

Під час програмних маніпуляцій з рядками об'єкту DataTable дана властивість встановлюється автоматично (рис. 9.13):

Значение	Описание
Added	Строка была добавлена в DataRowCollection, но метод AcceptChanges() не вызывался
Deleted	Строка была удалена с помощью метода Delete() объекта DataRow
Detached	Строка была создана, но не является частью коллекции DataRowCollection. Объект DataRow находится в этом состоянии после своего создания до того, как будет добавлен к коллекции (или же после удаления этого объекта из коллекции)
Modified	Строка была изменена, но метод AcceptChanges() не вызывался
Unchanged	Стока не изменялась со времени последнего вызова AcceptChanges()

Рисунок 9.13 - Властивість DataRow.RowState

Програмний код зміни властивості DataRow.RowState виглядає таким (рис. 9.14):

```
static void Main(string[] args)
{
...
    DataRow carRow = inventoryTable.NewRow();
    // Выводит 'Состояние строки: Detatched.'
    Console.WriteLine("Состояние строки: {0}.", carRow.RowState);
    carRow["Make"] = "BMW";
    carRow["Color"] = "черный";
    carRow["PetName"] = "Hamlet";
    inventoryTable.Rows.Add(carRow);

    // Выводит 'Состояние строки: Added.'
    Console.WriteLine("Состояние строки: {0}.",
        inventoryTable.Rows[0].RowState);
}
```

Рисунок 9.14 - Приклад зміни властивостей DataRow.RowState

4. Робота з DataTable

Тип DataTable відповідає безпосередньо за таблицю даних імає такі основні властивості (рис. 9.15)

Свойство	Описание
CaseSensitive	Индикатор необходимости учета регистра символов при сравнении строк в пределах таблицы. Значением по умолчанию является false (ложь)
ChildRelations	Возвращает коллекцию дочерних отношений для данного объекта DataTable (если таковые имеются)
Constraints	Возвращает коллекцию ограничений, поддерживаемых таблицей
DataSet	Возвращает объект DataSet, содержащий данную таблицу (если таковой имеется)
DefaultView	Возвращает пользовательское представление таблицы, которое может включать фильтр или позицию курсора
MinimumCapacity	Читает или устанавливает значение для начального числа строк данной таблицы (это значение по умолчанию равно 25)
ParentRelations	Возвращает коллекцию родительских отношений для данного объекта DataTable
PrimaryKey	Читает или устанавливает массив столбцов, функционирующих в качестве первичных ключей для таблицы данных
RemotingFormat	Позволяет определить, как объект DataSet должен выполнять сериализацию соответствующего содержимого (в двоичном или XML-формате) для слоя удаленного взаимодействия .NET. Это свойство появилось в .NET 2.0
TableName	Читает или устанавливает имя таблицы. Это же свойство может быть указано в качестве параметра конструктора

Рисунок 9.15 – Властивості типу DataTable

Приклад використання DataTable демонструють рисуноки 9.16, 9.17:

```
static void Main(string[] args)
{
...
   // Установка первичного ключа для таблицы.
   inventoryTable.PrimaryKey =
      new DataColumn[] { inventoryTable.Columns[0] };
}
```

```
static void Main(string[] args)
{
...
    // Наконец, добавление таблицы в DataSet.
    carsInventoryDS.Tables.Add(inventoryTable);
    // Теперь вывод данных DataSet.
    PrintDataSet(carsInventoryDS);
}
```

Рисунок 9.16 – Створення DataTable

```
static void PrintDataSet(DataSet ds)
{
   Console.WriteLine( "Таблицы в DataSet '{0}'.\n", ds.DataSetName);
   foreach (DataTable dt in ds.Tables)
   {
      Console.WriteLine("Таблица {0}.\n", dt.TableName);
      // Вывод имен столбцов.
      for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
      {
        Console.Write(dt.Columns[curCol].ColumnName.Trim() + "\t");
      }
      Console.Write Line ("\n--------------------------------");
      // Вывод DataTable.
      for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
      {
         for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
         {
        Console.Write (dt.Rows [curRow] [curCol].ToString() + "\t");
         }
         Console.WriteLine();
      }
   }
}
```

Рисунок 9.17 – Релізація методу PrintDataSet

5. Збереження DataSet (та DataTable) в форматі XML

За допомогою методів WriteXml() та ReadXml() вміст DataSet та DataTable можна зберігати та зчитувати з файлу у форматі XML (рис. 9.18):

```
static void Main(string[] args)
{
...
    // Сохранение DataSet в виде XML.
    carsInventoryDS.WriteXml("carsDataSet.xml");
    carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");

    // Очистка DataSet и вывод содержимого (должно быть пустым).
    carsInventoryDS.Clear();
    PrintDataSet(carsInventoryDS);

    // Загрузка и печать DataSet.
    carsInventoryDS.ReadXml("carsDataSet.xml");
    PrintDataSet(carsInventoryDS);
}
```

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>0</CarID>
    <Make>BMW</Make>
    <Color>черный</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
  <Inventory>
    <CarID>1</CarID>
    <Make>Saab</Make>
    <Color>красный</Color>
    <PetName>Sea Breeze</PetName>
  </Inventory>
</Car_x0020_Inventory>
```

Рисунок 9.18 – Збереження таблиці в форматі XML

6. Прив’язка DataTable до інтерфейсу користувача

Робота користувача з даними у графічному інтерфейсі здійснюється за допомогою компонента DataGridView. Рисунок вибору даного інструменту можна вибрати таким чином (рис. 9.19):

Рисунок 9.19 – Вибір DataGridView

Створена таблиця буде виглядати даним чином (рис. 9.20, 9.21):

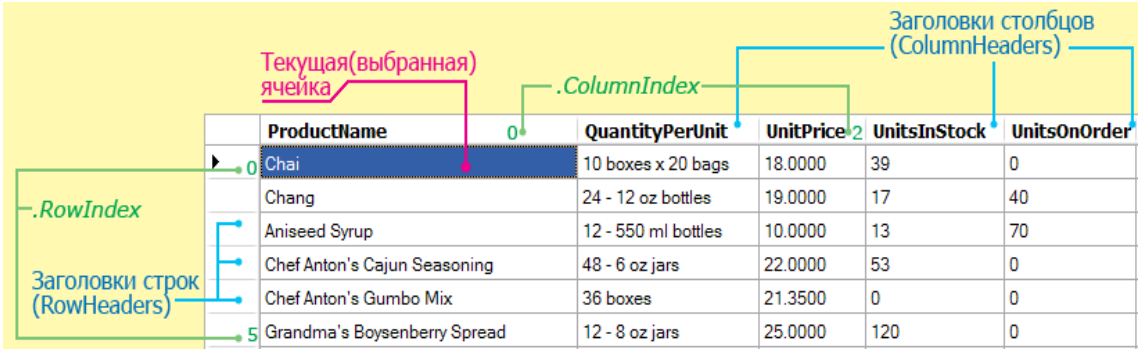


Рисунок 9.20 – Вигляд DataGridView

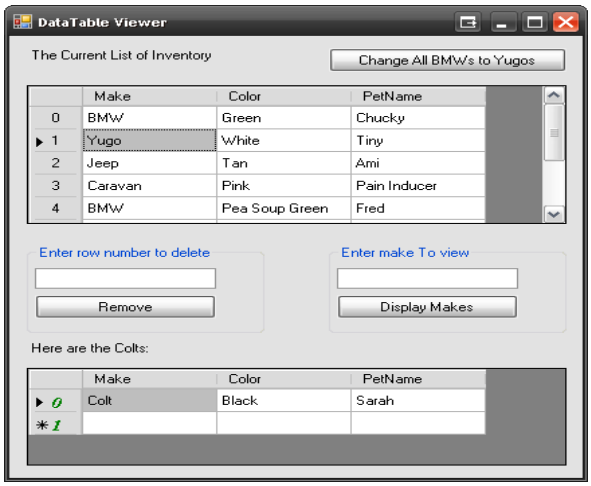


Рисунок 9.21 - Приклад використання DataGridView

На наступному лістингу 9.1 створимо клас автообіль.

Лістинг 9.1 – Створення класу

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace WindowsFormsDataTableViewer
{
    class Car
    {
        // Use automatic props.
        public string carPetName { get; set; }
        public string carMake { get; set; }
        public string carColor { get; set; }
        public Car(string petName, string make, string color)
        {
            carPetName = petName;
            carColor = color;
            carMake = make;
        }
    }
}
```

Далі створимо список автомобілів у конструкторі форми (лістинг 9.2):

Лістинг 9.2 – Створення списку автомобілів:

```
public partial class MainForm : Form
  {
    // A collection of Car objects.
    List<Car> listCars = new List<Car>();
    public MainForm()
    {
      InitializeComponent();

      // Fill the list with some cars.
      listCars.Add(new Car("Chucky", "BMW", "Green"));
```

```
      listCars.Add(new Car("Tiny", "Yugo", "White"));
      listCars.Add(new Car("Ami", "Jeep", "Tan"));
      listCars.Add(new Car("Pain Inducer", "Caravan", "Pink"));
      listCars.Add(new Car("Fred", "BMW", "Pea Soup Green"));
      listCars.Add(new Car("Sidd", "BMW", "Black"));
      listCars.Add(new Car("Mel", "Firebird", "Red"));
            listCars.Add(new Car("Sarah", "Colt", "Black"));
```

Потім створимо об’єкт таблиці і зв’яжемо його з DataGridView (лістинг 9.3):

Лістинг 9.3 – Зв'язок стореної таблиці з DataGridView:

```
private void CreateDataTable()
```

Продовження лістингу 9.3:

```
{
      // Create table schema
      DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
      DataColumn carColorColumn = new DataColumn("Color", typeof(string));
      DataColumn carPetNameColumn = new DataColumn("PetName",
typeof(string));
      carPetNameColumn.Caption = "Pet Name";
      inventoryTable.Columns.AddRange(new DataColumn[] { carMakeColumn,
        carColorColumn, carPetNameColumn });
      // Iterate over the array list to make rows.
      foreach (Car c in listCars)
      {
        DataRow newRow = inventoryTable.NewRow();
        newRow["Make"] = c.carMake;
        newRow["Color"] = c.carColor;
        newRow["PetName"] = c.carPetName;
        inventoryTable.Rows.Add(newRow);
      }
      // Bind the DataTable to the carInventoryGridView.
      carInventoryGridView.DataSource = inventoryTable;
    }
```

Пропишемо код для видалення рядків (лістинг 9.4):

Лістинг 9.4 – Код для видалення рядків

```
private void btnRemoveRow_Click(object sender, EventArgs e)
    {
      try
      {
        inventoryTable.Rows[(int.Parse(txtRowToRemove.Text))].Delete();
        inventoryTable.AcceptChanges();
      }
      catch (Exception ex)
      {
        MessageBox.Show(ex.Message);
      }
    }
```

7. Робота з типом DataView

Представлення (view) в базі даних дозволяє створити альтернативне представлення таблиці чи їх набору.

В ADO.NET тип DataView дозволяє у програмному режимі витягти інформацію із DataTable у окремий об’єкт, який може бути пов’язаний із окремим елементом GUI.

Розширимо попередній приклад (лістинг 9.5):

Лістинг 9.5 – Розширення прикладу з використанням DataView

```
// View of the DataTable.
DataView coltsOnlyView;
```

Продовження лістингу 9.5:

```
private void CreateDataView()
{
  // Set the table that is used to construct this view.
  coltsOnlyView = new DataView(inventoryTable);
  // Now configure the views using a filter.
  coltsOnlyView.RowFilter = "Make = 'Colt'";
  // Bind to the new grid.
  dataGridColtsView.DataSource = coltsOnlyView;
}
```

8. Робота з адаптерами даних

Адаптери даних використовуються для постачання даних до DataSet, а також для запису оновлених даних до СУБД. Тому члени типу DataSet розширюються за рахунок таких (рис. 9.22):

Члены	Описание
SelectCommand InsertCommand UpdateCommand DeleteCommand	Задают SQL-команды, которые будут отправлены хранилищу данных при вызове метода Fill() или Update()
Fill()	Заполняет данную таблицу в DataSet некоторым набором записей, зависящим от заданного объектом команды значения SelectCommand
Update()	Обновляет DataTable, используя объекты команд из свойств InsertCommand, UpdateCommand или DeleteCommand. Точная команда, которая при этом выполняется, зависит от значения RowState для данного DataRow в данном объекте DataTable (данного DataSet)

Рисунок 9.22 - Робота з адаптерами даних

Відповідно код заповнення DataSet за допомогою адаптера даних матиме таке представлення (рис. 9.23):

```
static void Main(string[] args)
{
   Console.WriteLine("***** Забавы с адаптерами данных *****\n");
   string cnStr =
      "uid=sa;pwd=;Initial Catalog=Cars;Data Source=(local)";

   // Заполнение DataSet новыми DataTable.
   DataSet myDS = new DataSet("Cars");
   SqlDataAdapter dAdapt =
      new SqlDataAdapter("Select * From Inventory", cnStr);
   dAdapt.Fill(myDS, "Inventory");

   // Отображение содержимого.
   PrintDataSet(myDS);
}
```

Рисунок 9.22 - Заповнення DataSet за допомогою адаптера даних

9. Об’єкти DataSet з багатьма таблицями та об’єкти DataRelation

У практичних задачах програмування баз даних часто виникає потреба працювати із багатьма зв’язаними таблицями. Технологія ADO.NET дозволяє реалізувати зв’язки на рівні програми за допомогою колекції DataRelation типу DataSet. Пов’язані таблиці можна представляти на 1 формі і їх дані можуть бути доступні один одному (тобто посилання на окремі дані зв’язаних таблиць), що показує рисунок 9.23:

Рисунок 9.23 – Вигляд зв’язаних таблиць

ЛК.10 – ПРОГРАМУВАННЯ ГРАФІКИ

Перелік питань

1. Введення до програмування графіки з використанням GDI+.
2. Тип Point.
3. Тип Rectangle.
4. Тип Region.
5. Клас Graphics.
6. Сеанси Paint.
7. Доступ до об’єкта Graphics поза обробником Paint.
8. Системи координат GDI+.
9. Визначення кольорових значень.
10. Маніпулювання шрифтами.
11. Простір імен System.Drawing.Drawing2D.
12. Робота з типами Pen.

На самостійне вивчення:

1. Робота з типами Brush [1, С.835-839].

1. Введення до програмування графіки з використанням GDI+

GDI+ - це технологія платформи .NET Framework, за допомогою якої реалізується виведення графіки, у тому числі і шрифтів (розшифровується як Graphical Device Interface). Платформа .NET Framework забезпечує цілий набір просторів імен для підтримки візуалізації двовимірної графіки. Дана технологія вміщує в собі функціональні можливості розробника, які звичайно пропонуються графічними пакетами (кольори, шрифти, пензлі тощо), а також геометричну трансформацію, згладжування, змішування палітр і друк документів.

Доступ до можливостей GDI+ реалізується через відповідні простори імен (рис. 10.1):

Пространство имен	Описание
System.Drawing	Базовое пространство имен GDI+, определяющее множество типов для основных операций визуализации (шрифты, перья, основные кисти и т.д.), а также "всемогущий" тип Graphics
System.Drawing.Drawing2D	Предлагает типы, используемые для более сложной двумерной/векторной графики (градиентные кисти, стили концов линий для перьев, геометрические трансформации и т.д.)
System.Drawing.Imaging	Предлагает типы, обеспечивающие обработку графических изображений (изменение палитры, извлечение метаданных изображения, работа с метафайлами и т.д.)
System.Drawing.Printing	Предлагает типы, обеспечивающие отображение графики на печатной странице, непосредственное взаимодействие с принтером и определение полного формата задания печати
System.Drawing.Text	Дает возможность управлять коллекциями шрифтов

Рисунок 10.1 – Базові простори імен GDI+

Більшість типів, що використовуються при створенні GDI+ програм, зберігаються в просторі імен System.Drawing. Тут реалізуються класи, що представляють собою зображення, пензлі, шрифти. Крім цього System.Drawing визначає ряд зв'язаних утилітарних типів, таких як колір (Color), крапка (Point), і прямокутник (Rectangle). На рисунках 10.2, 10.3 перечислюється опис деяких базових типів цього простору імен.

Тип	Описание
Bitmap	Тип, инкапсулирующий данные изображения (*.bmp или какого-то другого)
Brush Brushes SolidBrush SystemBrushes TextureBrush	Объекты Brush используются для заполнения внутренних областей графических форм, например, таких как прямоугольники, эллипсы и многоугольники
BufferedGraphics	Новый тип .NET 2.0, обеспечивающий графический буфер для двойной буферизации, которая используется для уменьшения или полного исключения влияния эффекта мелькания, возникающего при перерисовке изображений
Color SystemColors	Типы Color и SystemColors определяют ряд статических свойств, доступных только для чтения и используемых для получения нужного цвета при использовании различных перьев и кистей
Font FontFamily	Тип Font инкапсулирует характеристики данного шрифта (название, плотность, начертание, размер и т.д.). FontFamily предлагает абстракцию для группы шрифтов, имеющих аналогичный дизайн, но определенные вариации стиля
Graphics	Представляет реальную поверхность нанесения изображения, а также предлагает ряд методов для визуализации текста, изображений и геометрических шаблонов

Рисунок 10.2 – Базові типи простору імен System.Drawing

Icon SystemIcons	Представляют пользовательские пиктограммы, а также набор стандартных пиктограмм, предлагаемых системой
Image ImageAnimator	Тип Image — это абстрактный базовый класс, необходимый для поддержки функциональных возможностей типов Bitmap, Icon и Cursor. Тип ImageAnimator обеспечивает возможность выполнения цикла по набору типов Image из некоторого заданного интервала
Pen Pens SystemPens	Pens — это объекты, используемые для построения линий и кривых. Тип Pen определяет ряд статических свойств, возвращающих новый объект Pen заданного цвета
Point PointF	Структуры, представляющие отображение координаты (x, y) в соответствующее целое значение или значение с плавающей точкой, соответственно
Rectangle RectangleF	Структуры, представляющие размеры прямоугольника (снова с отображением в соответствующее целое значение или значение с плавающей точкой)
Size SizeF	Структуры, представляющие заданные высоту/ширину (снова с отображением в соответствующее целое значение или значение с плавающей точкой)
StringFormat	Тип, используемый для инкапсуляции различных характеристик размещения текста (выравнивание, промежутки между строками и т.д.)
Region	Тип, описывающий геометрический образ, скомпонованный из прямоугольников и траекторий

Рисунок 10.3 – Додатковий перелік базових типів

2. Тип Point

Тип Point визначений у просторі імен System.Drawing і призначений для збереження інформації і маніпулювання даними точки у двовимірній системі координат. За допомогою конструктора Point(x,y) можна створити екземпляр типу, передавши йому початкові координати точки. За допомогою методу Offset(x, y) можна змінювати координати точки.

Тип реалізує перевантажені арифметичні оператори і оператори порівняння, за допомогою яких можна здійснювати арифметичні операції над координатами точок та перевіряти відповідність координат різних екземплярів типу.

Даний тип є одним із утилітарних. Для ілюстрації його реалізації можна розглянути наступний консольний додаток, в якому використовується тип System.Drawing.Point (лістинг 10.1):

Лістинг 10.1 - Приклад використання типу Point

```
using System;
```

```
using System.Drawing;
namespace PointExample
```

Продовження лістингу 10.1:

```
{
    class Program
    {
        static void Main(string[] args)
        {
            Point pt = new Point(100, 72);
            Console.WriteLine(pt);
            pt.Offset(20, 20);
            Console.WriteLine(pt);
            Point pt2 = new Point(120, 92);
            if (pt == pt2)
                Console.WriteLine("Точки однакові");
            else
                Console.WriteLine("Точки різні");
        }
    }
}
```

В результаті матимемо консольну програму. Вона буде виводити на екран координати 2 точок (фактично 2 буде утворена з 1 внаслідок зміни координат) та порівняння утвореної точки із заданою (рис. 10.4):

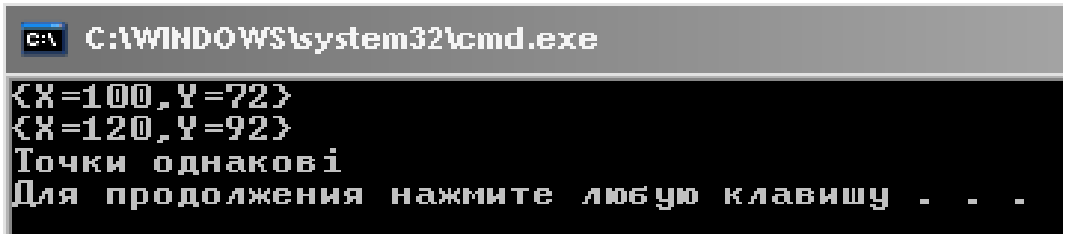


Рисунок 10.4 – Результат виконання програми

3. Тип Rectangle

Тип Rectangle визначений у просторі імен System.Drawing і призначений для маніпулювання прямокутниками. Екземпляр створюється за допомогою конструктора, якому передаються координати лівої верхньої та правої нижньої точок. Метод Contains() дозволяє перевірити наявність іншого об’єкту (типу Point чи Rectangle) всередині даного. Реалізація даного типу показана на наступному прикладі (лістинг 10.2):

Лістинг 10.2 - Приклад використання Rectangle

```
using System;
using System.Drawing;
namespace RectangleExample
```

Продовження лістингу 10.2:

```
{
    class Program
    {
        private static void CheckPoint(Rectangle rt, Point pt)
        {
            if (rt.Contains(pt))
                Console.WriteLine("Точка знаходиться всередині
прямокутника");
            else
                Console.WriteLine("Точка знаходиться за межами
прямокутника");
        }
        static void Main(string[] args)
        {
            Rectangle rt = new Rectangle(0, 0, 100, 100);
            Point pt = new Point(101, 101);
            CheckPoint(rt, pt);
            pt.Offset(-10, -10);
            CheckPoint(rt, pt);
        }
    }
}
```

Також програма матиме певний результат. У даному випадку використовується метод визначення за координатами, чи знаходиться задана точка у межах заданого прямокутника (побудованого програмою). Результат виконання демонструється на рисунку 10.5:

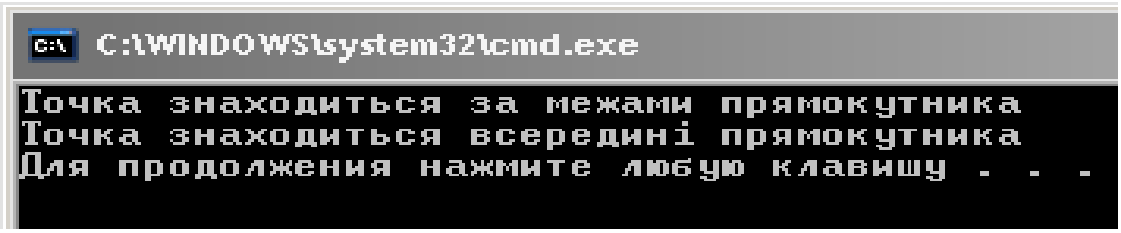


Рисунок 10.5 – Результат виконання програми

4. Тип Region

Тип Region представляє внутрішню частину геометричної фігури. Конструктор типу має отримати шаблон, на основі якого створюється фігура.

Маючи внутрішню частину фігури, можна маніпулювати нею з використанням різноманітних членів. Основні з них та базові наводяться на рисунку 10.6:

- `Complement()` — изменяет данный объект `Region` на часть указанного графического объекта, не пересекающуюся с данным объектом `Region`;
- `Exclude()` — изменяет данный объект `Region` на ту его часть, которая не пересекается с указанным графическим объектом;
- `GetBounds()` — возвращает `Rectangle(F)`, который представляет прямоугольный регион, ограничивающий данный объект `Region`;
- `Intersect()` — изменяет данный объект `Region` на его пересечение с указанным графическим объектом;
- `Transform()` — трансформирует данный объект `Region` с помощью указанного объекта `Matrix`;
- `Union()` — изменяет данный объект `Region` на его объединение с указанным графическим объектом;
- `Translate()` — сдвигает координаты данного объекта `Region` на указанную величину.

Рисунок 10.6 – Члени класу Region

5. Клас Graphics

Клас Graphics безпосередньо відповідає за побудову графіки у GDI+, він надає як поверхню, на якій відбувається формування графіки, а також методи, які виводять графіку. Клас не дозволяє безпосереднього створення своїх екземплярів.

Клас System.Drawing.Graphics є входом у функціональні можливості візуалізації GDI+. Частковий список членів даного класу представлено на рисунку 10.7

Крім ряду методів візуалізації клас також визначає додаткові члени, які дозволяють конфігурувати стан об’єкта Graphics. За допомогою присвоєння необхідних значень властивостям, показаним на рисунку 10.8, можна змінити поточні характеристики процесу візуалізації.

При створенні екземпляру класу ключове слово new не використовується. Адже клас не має відкритих конструкторів.

Методы	Описание
FromHdc() FromHwnd() FromImage()	Статические методы, обеспечивающие возможность получения действительного объекта Graphics из данного изображения (например, пиктограммы, точечного рисунка и т.п.) или GUI-элемента
Clear()	Заполняет объект Graphics заданным цветом, выполняя в процессе заполнения очистку поверхности рисования
DrawArc() DrawBezier() DrawBeziers() DrawCurve() DrawEllipse() DrawIcon() DrawLine() DrawLines() DrawPie() DrawPath() DrawRectangle() DrawRectangles() DrawString()	Эти методы используются для визуализации данного изображения или геометрического шаблона. Позже вы увидите, что методы DrawXXX() требуют использования объектов Pen GDI+
FillEllipse() FillPath() FillPie() FillPolygon() FillRectangle()	Эти методы используются для заполнения внутренности данной геометрической формы. Позже вы увидите, что методы DrawXXX() требуют использования объектов Brush GDI+

Рисунок 10.7 – Члени класу Graphics

Свойства	Описание
Clip ClipBounds VisibleClipBounds IsClipEmpty IsVisibleClipEmpty	Позволяют установить опции отсечения, используемые с текущим объектом Graphics
Transform	Позволяет трансформировать "мировые координаты" (подробнее об этом будет говориться позже)
PageUnit PageScale DpiX DpiY	Позволяют указать начало координат для операций визуализации, а также единицу измерения
SmoothingMode PixelOffsetMode TextRenderingHint	Позволяют задать параметры гладкости геометрических объектов и текста
CompositingMode CompositingQuality	Свойство CompositingMode задает режим визуализации: либо рисование поверх фона, либо сопряжение с фоном
InterpolationMode	Указывает режим интерполяции данных между конечными точками

Рисунок 10.8 - Властивості класу Graphics, які визначають стан

Об’єкт Graphics використовує ресурси системи, а тому він має звільнятися як тільки стає непотрібний для вирішення конкретної задачі. Адже він працює з самими різними некерованими ресурсами, то має зміст вивільнення вказаних ресурсів як можна швидше.

Якщо об’єкт Graphics отриманий у якості посилання на вже існуючий об’єкт, то його звільняти не потрібно (наприклад, об’єкт, отриманий для форми під час обробки події Paint), якщо об’єкт створено в результаті виконання певних операцій програмістом (наприклад, завантаження зображення із файлу), то він має бути звільнений за допомогою метода Dispose() як тільки стає непотрібним.

6. Сеанси Paint

Коли елемент графічного інтерфейсу, такий як, наприклад, форма, вимагає перерисування, то він генерує подію Paint, у відповідь на яку необхідно викликати код, що відповідає за рисування позначеного елементу інтерфейсу. Існує два способи обробки події:

– перевизначити метод OnPaint();
– реалізувати обробник події Paint.

В процесі обробки події Paint можна отримати доступ до об’єкту Graphics, який відповідає за даний елемент управління і за допомогою якого відбувається виведення графіки.

Можна розглянути приклад обробки події Paint для форми. Даний код (лістинг 10.3) необхідно помістити у обробник події Paint форми і отримати наступний результат графічної реалізації (рис. 10.9):

Лістинг 10.3 – Обробка події Paint для форми

```
private void Form1_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    g.DrawString("Обробка події Paint форми", new Font("Times New Roman",
20),
        Brushes.Red, 0, 0);
}
```

Рисунок 10.9 – Результат обробки події Paint для форми

Також передбачається оновлення області клієнта форми. Подія Paint викликається операційною системою автоматично у випадку, якщо форма вимагає перерисування, наприклад, коли змінюються її розміри, або вона була перекрита іншою формою. Цей стан (тобто момент між потребою в оновленні і фактичним оновленням) називається “брудним”. Програмно позначити необхідність перерисування форми (тобто фактично перевести її у “брудний стан”) можна за допомогою методу Invalidate(). Метод Invalidate() може бути викликаний без параметрів, у такому разі оновленню підлягає вся форма, а може бути викликаний з параметром типу Rectangle, який задає область, що підлягає оновленню. Цей метод може бути перевантаженим.

7. Доступ до об’єкта Graphics поза обробником Paint

В окремих ситуаціях необхідно отримати доступ до об’єкта Graphics за межами обробника події Paint. Зробити це можна за допомогою статичного методу Graphics.FromWnd (), який дозволяє отримати об’єкт на основі вікна системи. Необхідно передбачити, щоб уся інформація, яка виводиться за межами обробника Paint, зберігалася при повторному виклику, інакше вона буде втрачена.

Реалізація методу створення червоних кругів на формі при натисненні на кнопку миші демонструється в лістингу 10.4, а результат виконання програми – на рисунку 10.10:

Лістинг 10.4 – Метод створення доступу до об’єкта Graphics поза обробником Paint:

```
private void Form1_MouseDown(object sender, MouseEventArgs e)
{
   Graphics g = Graphics.FromHwnd(this.Handle);
   g.FillEllipse(Brushes.Firebrick, e.X, e.Y, 10, 10);
   g.Dispose();
}
```

Рисунок 10.10 – Реалізація завдання і результат виконання

8. Системи координат GDI+

GDI+ підтримує три системи координат:

– “світові координати” (зовнішні координати) – координати певного абстрактного полотна, незалежно від одиниць виміру;
– координати сторінки – визначаються по відношенню до світових координат;
– “координати пристрою” – визначаються як результат використання конкретних координат сторінки до певної точки відліку у світових координатах.

Для перетворення між різними системами координат та зміни початку координат використовується метод TranslateTransform(). Рисунок 10.11 відображатиме код зміщення верхнього кута прямокутника у певну задану точку (20, 20):

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    // Указание смещения (10 * 10) для страничных координат.
    Graphics g = e.Graphics;
    g.TranslateTransform(10, 10);
    g.DrawRectangle(10, 10, 100, 100);
}
```

Рисунок 10.11 – Приклад перетворення між системами координат

За замовчуванням у якості одиниці задається піксель, відлік іде згори вниз, зліва направо (рис. 10.12):

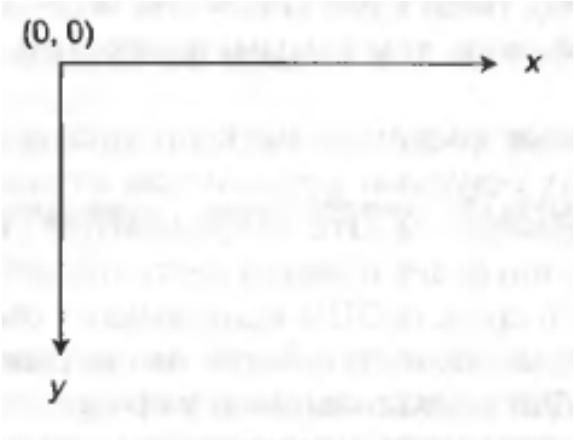


Рисунок 10.12 – Система координат GDI+, що пропонується за замовчуванням

Якщо необхідно, можна задати інші одиниці вимірювання, а не піксельні, із структури GraphicsUnit (World – часові координати, Display – пік сель для відео дисплея 1/100 дюйма, Pixel - піксель, Point – стандартна крапка принтера 1/72 дюйма, Inch - дюйм, Document – стандартна одиниця документу 1/300 дюйма, Millimeter - міліметр). Можна модифікувати код та отримати необхідну реалізацію на рисунку 10.13:

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
    // Отображение прямоугольника в дюймах, а не в пикселях...
    Graphics g = e.Graphics;
    g.PageUnit = GraphicsUnit.Inch;
    g.DrawRectangle(new Pen(Color.Red, 5), 0, 0, 100, 100);
}
```

Рисунок 10.13 – Вибір альтернативної одиниці виміру

9. Визначення кольорових значень

Багато методів класу Graphics вимагають позначення кольору, який має використовуватися під час рисування.

Структура System.Drawing.Color дозволяє сформувати необхідний колір:

– зазначення кольору по імені: Color c = Color.Red;
– формування кольору із простору ARGB: Color c = Color.FromArgb(0, 255, 128, 64);
– формування кольору із його назви у тексті: Color c = Color.FromName(“Red”).

Незалежно від методу отримання типу Color, з цим типом можна взаємодіяти за допомогою його таких членів (рисунок. 10.14):

- `GetBrightness()` — возвращает значение яркости типа `Color` на основании измерения HSB (Hue-Saturation-Brightness — оттенок, насыщенность, яркость).
- `GetSaturation()` — возвращает значение насыщенности типа `Color` на основании измерения HSB.
- `GetHue()` — возвращает значение оттенка типа `Color` на основании измерения HSB.
- `IsSystemColor` — индикатор того, что данный тип `Color` является зарегистрированным системным цветом.
- `A`, `R`, `G`, `B` — возвращают значения, присвоенные для альфа, красной, зеленой и синей составляющих типа `Color`.

Рисунок 10.14 – Важливі члени типу Color

Щоб забезпечити кінцевому користувачеві програми можливість конфігурувати тип Color, простір імен System.Windows.Forms пропонує вбудований клас діалогового вікна з ім’ям ColorDialog. Він призначений для вибору кольору за допомогою стандартного діалогового вікна операційної

системи. В даному випадку отримаємо вигляд задання кольору, як діалогове вікно налаштування кольору операційної системи (рис. 10.15). ЇЇ можна викликати модально, використовувати властивість Color.

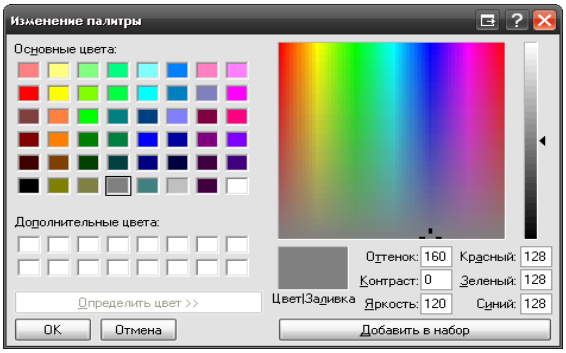


Рисунок 10.15 – Діалогове вікно налаштування кольорів

Приклад реалізації даного класу у коді програми реалізовано на рисунку 10.16:

```
public partial class MainForm : Form
{
    private ColorDialog colorDlg;
    private Color currColor = Color.DimGray;
    public MainForm()
    {
        InitializeComponent();
        colorDlg = new ColorDialog();
        Text = "Для изменения цвета щелкните здесь";
        this.MouseDown += new MouseEventHandler(MainForm_MouseDown);
    }
    private void MainForm_MouseDown(object sender, MouseEventArgs e)
    {
        if (colorDlg.ShowDialog() != DialogResult.Cancel)
        {
            currColor = colorDlg.Color;
            this.BackColor = currColor;
            string strARGB = colorDlg.Color.ToString();
            MessageBox.Show(strARGB, "Выбранный цвет:");
        }
    }
}
```

Рисунок 10.16 – Приклад використання ColorDialog

10. Маніпулювання шрифтами.

Тип System.Drawing.Font представляє шрифт, який встановлений на комп'ютері користувача. При створенні екземплярів цього типу можна використовувати декілька перевантажених конструкторів. Після створення об'єкту типу Font він має бути переданий методу Graphics.DrawString() у якості параметру (рис. 10.17):

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Аргументы (String, Font, Brush, Point).
   g.DrawString("Моя строка", new Font("WingDings", 25),
      Brushes.Black, new Point(0,0));
   // Аргументы (String, Font, Brush, int, int)
   g.DrawString("Другая строка", new Font("Times New Roman", 16),
      Brushes.Red, 40, 40);
}
```

Рисунок 10.17 – Приклад роботи з шрифтами

У мові програмування реалізована робота із сімействами шрифтів. Простір імен System.Drawing визначає також тип FontFamily, який може бути використаний для визначення групи шрифтів, які мають однаковий базовий дизайн, однак відрізняються певними варіаціями стилю.

Тип FontFamily може бути використаний для визначення числових характеристик шрифта. Тобто реалізується можливість зібрання статистики по відношенню певного сімейства шрифтів. Така змога надається через використання членів типу FontFamily (рис. 10.18):

Член	Описание
GetCellAscent()	Возвращает метрику надстрочного элемента для членов данного семейства
GetCellDescent()	Возвращает метрику подстрочного элемента для членов данного семейства
GetLineSpacing()	Возвращает расстояние между двумя последовательными строками текста для данного FontFamily с указанным FontStyle
GetName()	Возвращает имя данного FontFamily на указанном языке
IsStyleAvailable()	Индикатор доступности указанного FontStyle

Рисунок 10.18 – Основні члени типу FontFamily

Можливе вказування гарнітури шрифту, використовуючи вбудований набір гарнітур. В даному випадку в обробнику Paint слід помістити наступний код виконання програми (рис. 10.19). В даному випадку В даному випадку буде поміщатися та перемальовуватися повідомлення в центрі відповідного прямокутника.

```
private void MainForm_Paint(object sender, PaintEventArgs e)
{
   Graphics g = e.Graphics;
   // Размер шрифта должен находиться в диапазоне от 12 до 62,
   // в зависимости от swellValue.
   Font theFont = new Font(strFontFace, 12 + swellValue);
   string message = "Привет GDI+";

   // Вывод сообщения в центре прямоугольника.
   float windowCenter = this.DisplayRectangle.Width/2;
   SizeF stringSize = g.MeasureString(message, theFont);
   float startPos = windowCenter - (stringSize.Width/2);
   g.DrawString(message, theFont, new SolidBrush(Color.Blue),
      startPos, 10);
}
```

Рисунок 10.19 – Визначення гарнітури шрифту

Мова програмування містить список встановлених шрифтів, які можна безпосередньо обирати для використання. Доступ до списку встановлених у системі шрифтів можна отримати за допомогою класу InstalledFontCollection (рис. 10.20):

```
public partial class MainForm : Form
{
    // Содержит список шрифтов.
    private string installedFonts;

    // Обработчик меню для получения списка шрифтов.
    private void mnuConfigShowFonts_Clicked(object sender,
        EventArgs e)
    {
        InstalledFontCollection fonts = new InstalledFontCollection();
        for(int i = 0; i < fonts.Families.Length; i++)
            installedFonts += fonts.Families[i].Name + " ";

        // На этот раз нужно обновить всю область клиента,
        // поскольку обновляется строка installedFonts в нижней части
        // области клиента.
        Invalidate();
    }
}
```

Рисунок 10.20 – Використання списку встановлених шрифтів

Існує клас діалогового вікна для налаштування шрифтів (FontDialog). Він призначений для виклику стандартного діалогового вікна операційної системи для вибору шрифту. Для його відображення необхідно викликати метод ShowDialog, для отримання поточних характеристик шрифту та їх зміни використовується властивість Font. Для прикладу можна розглянути наступну форму, що задає необхідний шрифт. При натисненні користувача мишею на будь-якому місці вікна форми відображається діалогове вікно Шрифт та виводиться інформація про поточний вибір (рис. 10.21):

```
private FontDialog fontDlg = new FontDialog();
private Font currFont = new Font("Times New Roman", 12);
public MainForm()
{
    InitializeComponent();
    CenterToScreen();
}
private void MainForm_MouseDown(object sender, MouseEventArgs e)
{
    if (fontDlg.ShowDialog() != DialogResult.Cancel)
    {
        currFont = fontDlg.Font;
        this.Text = string.Format("Selected Font: {0}", currFont);
        Invalidate();
    }
}
```

Рисунок 10.21 – Код реалізації прикладу

11.Простір імен System.Drawing.Drawing2D

Даний простір імен містить ряд класів, які дозволяють обрати різні форми пера, яким здійснюється рисування, визначити текстуру і працювати з векторною графікою. Деякі базові типи, згруповані по функціональним можливостям, описані на рисунку 10.22.

Також варто знати, що простір імен System.Drawing.Drawing2D визначає набір переліків, які використовуються разом із вказаними на рисунку базовими типами.

Классы	Описание
AdjustableArrowCap CustomLineCap	Используются для изменения формы концов линий для перьев. Данные типы задают, соответственно, регулируемую стрелку и пользовательскую форму конца линии
Blend ColorBlend	Позволяют определить шаблон смешивания (и цвет) для использования с LinearGradientBrush
GraphicsPath GraphicsPathIterator PathData	Объект GraphicsPath представляет серию линий и кривых. Этот класс позволяет добавлять в траектории геометрические шаблоны практически любого вида (дуги, прямоугольники, линии, строки, многоугольники и т.д.). PathData содержит графические данные, формирующие траекторию
HatchBrush LinearGradientBrush PathGradientBrush	Экзотические типы кистей

Рисунок 10.22 - Типи простору імен

12.Робота з типами Pen

Тип Pen використовується для вибору ліній, за допомогою яких з’єднуються точки, що формують зображення. Даний тип визначає невеликий набір конструкторів, які дозволяють задати початковий колір та ширину пера. Опис деяких із цих властивостей пропонується на рисунку 10.23.

Свойства	Описание
Brush	Определяет тип Brush для использования с данным типом Pen
Color	Определяет тип Color для использования с данным типом Pen
CustomStartCap CustomEndCap	Читает или устанавливает параметры пользовательского стиля концов линий, создаваемых с помощью данного типа Pen. *Стиль концов линий* — это просто термин, используемый для обозначения того, как должен выглядеть начальный и заключительный "штрих" данного пера. Эти свойства позволяют строить пользовательские стили начала и конца линий для типов Pen
DashCap	Читает или устанавливает параметры стиля концов линий, используемого для прерывистых линий, создаваемых с помощью данного типа Pen
DashPattern	Читает или устанавливает массив пользовательской маски для рисования прерывистых линий. Соответствующие "тире" складываются из сегментов линий
DashStyle	Читает или устанавливает параметры стиля, используемого для прерывистых линий, создаваемых с помощью данного типа Pen
StartCap EndCap	Читает или устанавливает встроенный стиль концов линий, создаваемых с помощью данного типа Pen. Стиль концов линий Pen устанавливается в соответствии с перечнем LineCap, определенным в пространстве имен System.Drawing.Drawing2D
Width	Читает или устанавливает ширину данного Pen
DashOffset	Читает или устанавливает расстояние от начала линии до начала шаблона прерывистой линии

Рисунок 10.23 – Властивості Pen

Приклад використання даного типу демонструє лістинг 10.5, а саме вікно форми (результату виконання програми) можна побачити на рисунку 10.24.

Лістинг 10.5 – Приклад графічної програми

```
private void PenForm_Paint(object sender, PaintEventArgs e)
{
    Graphics g = e.Graphics;
    // Створення великого пера синього кольору
    Pen bluePen = new Pen(Color.Blue, 20);
    // Отримання готового пера із типу Pens
    Pen pen2 = Pens.Firebrick;
    // Нарисуємо декілька фігур
    g.DrawEllipse(bluePen, 10, 10, 100, 100);
    g.DrawLine(pen2, 10, 130, 110, 130);
    g.DrawPie(Pens.Black, 150, 10, 120, 150, 90, 80);
    // Рисуємо пупурний полігон з пунктирною границею
    Pen pen3 = new Pen(Color.Purple, 5);
    pen3.DashStyle = DashStyle.DashDotDot;
    g.DrawPolygon(pen3, new Point[]{new Point(30, 140),
        new Point(265, 200), new Point(100, 225),
        new Point(190, 190), new Point(50, 330),
        new Point(20, 180)});
    // Та прямокутник з текстом
    Rectangle r = new Rectangle(150, 10, 130, 60);
    g.DrawRectangle(Pens.Blue, r);
    g.DrawString("Вчимося рисувати з GDI+", new Font("Arial", 11),
Brushes.Black, r);
}
```

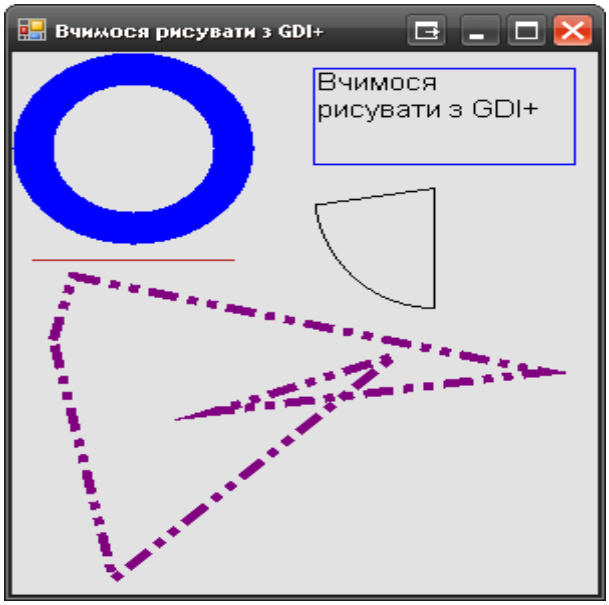


Рисунок 10.24 – Результат виконання програми

ЛК.11 – ВІЗУАЛІЗАЦІЯ ЗОБРАЖЕНЬ, ПОБУДОВА ДІАГРАМ ТА ЗВІТІВ

Перелік питань

1. Візуалізація зображень.
2. Побудова діаграм з програмним управлінням Microsoft Excel.
3. Побудова звітів з програмним управлінням Microsoft Word.

На самостійне вивчення:

1. Використання ресурсів в .NET [1, С.850-857].

1. Візуалізація зображень

Абстрактний тип System.Drawing.Image визначає ряд методів та властивостей для роботи із зображеннями. Конкретні екземпляри створюються для наслідуваних класів, які уособлюють певні типи графічних зображень. Даний тип використовує різноманітні властивості для представлення розмірів зображень чи отримання доступу до їх палітри тощо. Відображення базових членів класу Image приведено на рисунку 11.1:

Члены	Описание
FromFile()	Статический метод, создающий объект Image из указанного файла
FromStream()	Статический метод, создающий объект Image из указанного потока данных
Height Width Size HorizontalResolution VerticalResolution	Свойства, возвращающие информацию о размерах данного объекта Image
Palette	Свойство, возвращающее тип данных ColorPalette, который представляет палитру, используемую для данного объекта Image
GetBounds	Метод, возвращающий объект Rectangle, который представляет текущие размеры данного объекта Image
Save()	Метод, сохраняющий в файл данные, содержащиеся в производном от Image типе

Рисунок11.1 – Члени типу Image

Оскільки екземпляр класу Image не можна створити безпосередньо, то звичайно безпосередньо створюється екземпляр типу Bitmap. Припустимо,

що ми маємо певний клас Form, який відображає 3 точкові рисунки в області клієнта. Вказавши для кожного з типів Bitmap найбільш відповідний файл зображення, можна просто відобразити їх в обробнику події Paint, використовуючи метод Graphics.DrawImage(). Даний код, що реалізує необхідне виконання, показано в лістингу 11.1:

Лістинг 11.1 - Приклад завантаження зображень із файлів

```
public Form1()
{
  //
  // Required for Windows Form Designer support
  //
  InitializeComponent();
  // Fill the images with bitmaps.
  bMapImageA = new Bitmap("imageA.bmp");
  bMapImageB = new Bitmap("imageB.bmp");
  bMapImageC = new Bitmap("imageC.bmp");
  // Create an interesting region.
  myPath.StartFigure();
  myPath.AddLine(new Point(150, 10), new Point(120, 150));
  myPath.AddArc(200, 200, 100, 100, 0, 90);
  Point point1 = new Point(250, 250);
  Point point2 = new Point(350, 275);
  Point point3 = new Point(350, 325);
  Point point4 = new Point(250, 350);
  Point[] points = {point1, point2, point3, point4};
  myPath.AddCurve(points);
  myPath.CloseFigure();
  CenterToScreen();
}
```

Щоб створити візуалізацію різних зображень на формі, потрібно скористатися лістингом 11.2. Внаслідок прописаного коду буде утворено необхідний результат (рис. 11.2).

Лістинг 11.2 - Приклад візуалізація зображень на формі

```
private void Form1_Paint(object sender, System.Windows.Forms.PaintEventArgs
e)
{
    Graphics g = e.Graphics;
    // Render all three images.
    g.DrawImage(bMapImageA, rectA);
    g.DrawImage(bMapImageB, rectB);
    g.DrawImage(bMapImageC, rectC);
    // Draw the graphics path.
```

Продовження лістингу 11.2:

```
g.FillPath(Brushes.AliceBlue, myPath);
    // Draw outline (if clicked...)
```

```
    if(isImageClicked == true)
    {
        Pen outline = new Pen(Color.Red, 5);
        switch(imageClicked)
        {
            case 0:
                g.DrawRectangle(outline, rectA);
                break;
            case 1:
                g.DrawRectangle(outline, rectB);
                break;
            case 2:
                g.DrawRectangle(outline, rectC);
                break;
            case 3:
                g.DrawPath(outline, myPath);
                break;
            default:
                break;
        }
    }
}
```

Рисунок 11.2 – Результат візуалізації зображень

2. Побудова діаграм з програмним управлінням Microsoft Excel

Для того, щоб отримати доступ до управління встановленим на комп’ютері екземпляром Microsoft Excel, необхідно додати в проект

посилання на COM-об’єкт автоматизації. Це робиться за допомогою контекстного меню проекту та вибору додання необхідних посилань, як показано на рисунку 11.3:

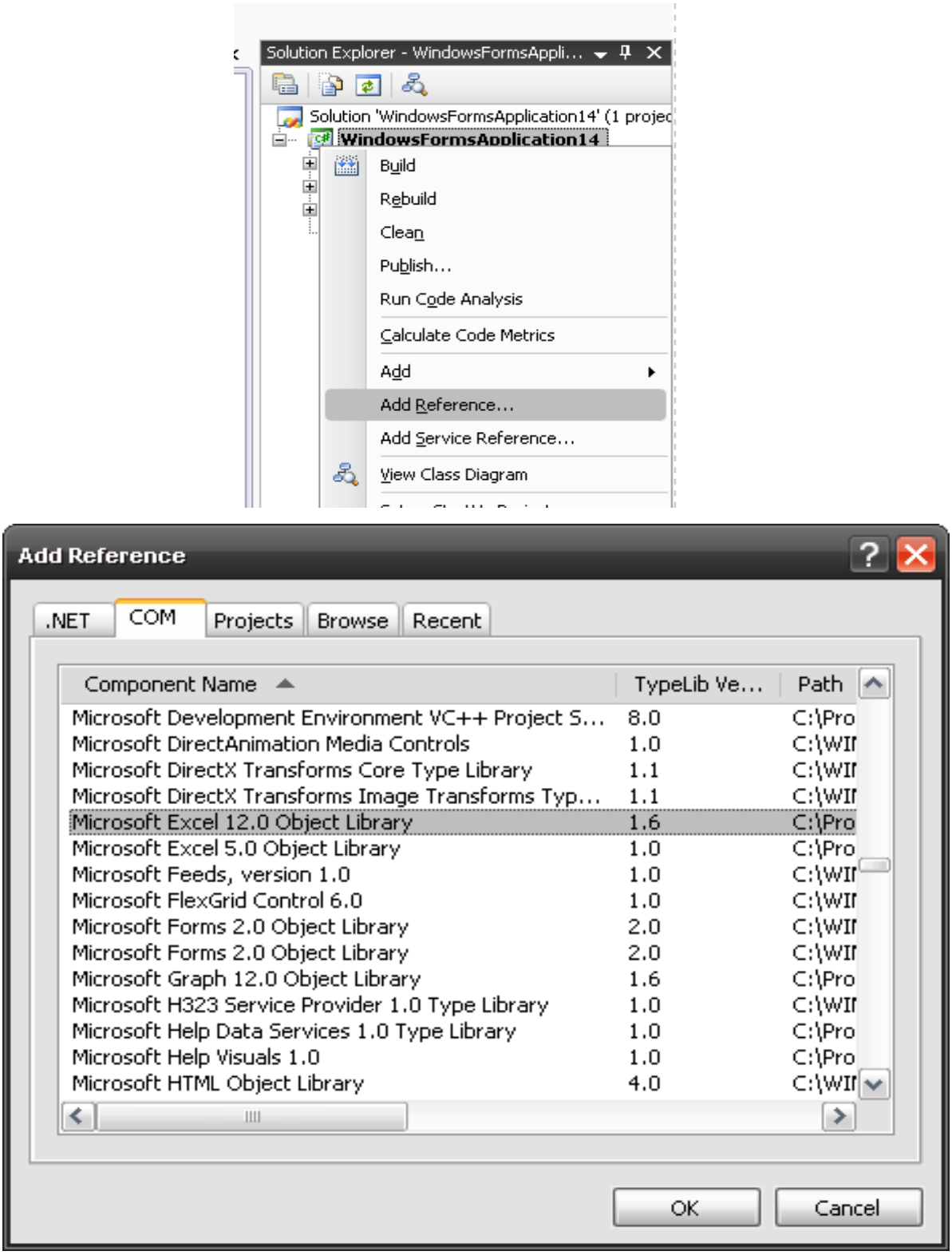


Рисунок 11.3 – Додання посилання на Microsoft Excel у програмі

В операційних системах Windows підтримується можливість встановлення регіональних параметрів, які впливають на спосіб представлення чисел, дат, виконання вбудованих алгоритмів сортування та ін. Хоча регіональні параметри задаються на рівні ОС, для окремих програм (точніше, їх потоків) під час виконання їх можна змінювати. У середовищі .NET Framework це здійснюється за допомогою властивості System.Threading.Thread.CurrentThread.CurrentCulture.

Якщо програма виконується на ПК, який має відмінні регіональні параметри від американських (культура «en-US»), то до програмного виклику MS Excel їх необхідно змінити на американські, а потім повернути назад. Приклад заміни регіональних параметрів і повернення до початкового значення показано в лістингу 11.3:

Лістинг 11.3 – Зміна регіональних параметрів

```
System.Threading.Thread thisThread = System.Threading.Thread.CurrentThread;
System.Globalization.CultureInfo originalCulture = thisThread.CurrentCulture;


// Використовуємо блок try..finally
try
{
    thisThread.CurrentCulture = new System.Globalization.CultureInfo(
        "en-US");

    // Здійснюємо виклик MS Excel
    ...


}
finally
{
    // Відновлюємо регіональні параметри
    thisThread.CurrentCulture = originalCulture;
}
```

Для продовження створення звіту потрібно імпортувати спеціальний простір імен (лістинг 11.4).

Лістинг 11.4 – Імпортування простору імен для взаємодії з Microsoft Excel

```
using Microsoft.Office.Interop.Excel;
```

Наступним кроком є задання культури для потоку на “en-US”. Дану реалізацію демонструє лістинг 11.5, з використанням блоку обов’язкового виконання коду незалежно від помилок для уникнення проблем із параметрами.

Лістинг 11.5 - Задання культури для потоку на “en-US

```
System.Threading.Thread thisThread = System.Threading.Thread.CurrentThread;
System.Globalization.CultureInfo originalCulture = thisThread.CurrentCulture;
thisThread.CurrentCulture = new System.Globalization.CultureInfo("en-US");
try
{
    BuildTheDiagram();
}
finally
{
    thisThread.CurrentCulture = originalCulture;
}
```

У попередньому лістингу викликався метод BuildTheDiagram(). Він саме і містить всю кодову частину побудови звіту. Для того, щоб його реалізувати, потрібно здійснити такі кроки:

а) Створити об’єкт ApplicationClass та задекларувати необхідні змінні (лістинг 11.6):

Лістинг 11.6 – Реалізація першого кроку

```
ApplicationClass excelApplication = new ApplicationClass();
Workbook newWorkbook = null;
Worksheet targetSheet = null;
Range dataRange = null;
ChartObjects chartObjects = null;
ChartObject newChartObject = null;
```

б) Задати шлях до документа (лістинг 11.7):

Лістинг 11.7 – Кодова реалізація другого кроку

```
string paramWorkbookPath = @"C:\Temp\Test.xlsx";
```

в) Створити зміну для необ’язкових параметрів (лістинг 11.8):

Лістинг 11.8 – Створення спеціальної змінної для параметрів

```
object paramMissing = Type.Missing;
```

г) Задати властивості діаграми (лістинг 11.9):

Лістинг 11.9 – Властивості діаграми

```
object paramChartFormat = 1;
object paramCategoryLabels = 0;
object paramSeriesLabels = 0;
bool paramHasLegend = true;
object paramTitle = "Sales by Quarter";
object paramCategoryTitle = "Fiscal Quarter";
object paramValueTitle = "Billions";
```

д) Створити метод, який відповідає за вставку даних (лістинг 11.10):

Лістинг 11.10 – Метод вставки даних

```
static void SetCellValue(Worksheet targetSheet, string Cell,
    object Value)
{
    targetSheet.get_Range(
        Cell, Cell).set_Value(XlRangeValueDataType.xlRangeValueDefault,
        Value);
}
```

е) Створити нову сторінку в книзі Excel (лістинг 11.11):

Лістинг 11.11 – Створення нової сторінки

```
newWorkbook = excelApplication.Workbooks.Add(XlWBATemplate.xlWBATWorksheet);
targetSheet = (Worksheet)(newWorkbook.Worksheets[1]);
targetSheet.Name = "Quarterly Sales";
```

є) Заповнити лист даними (лістинг 11.12):

Лістинг 11.12 – Заповнення фрагменту даними

```
SetCellValue(targetSheet, "A2", "N. America");
SetCellValue(targetSheet, "A3", "S. America");
```

```
SetCellValue(targetSheet, "A4", "Europe");
SetCellValue(targetSheet, "A5", "Asia");
SetCellValue(targetSheet, "B1", "Q1");
SetCellValue(targetSheet, "B2", 1.5);
SetCellValue(targetSheet, "B3", 2);
SetCellValue(targetSheet, "B4", 2.25);
SetCellValue(targetSheet, "B5", 2.5);
SetCellValue(targetSheet, "C1", "Q2");
SetCellValue(targetSheet, "C2", 2);
SetCellValue(targetSheet, "C3", 1.75);
SetCellValue(targetSheet, "C4", 2);
SetCellValue(targetSheet, "C5", 2.5);
SetCellValue(targetSheet, "D1", "Q3");
SetCellValue(targetSheet, "D2", 1.5);
SetCellValue(targetSheet, "D3", 2);
SetCellValue(targetSheet, "D4", 2.5);
SetCellValue(targetSheet, "D5", 2);
SetCellValue(targetSheet, "E1", "Q4");
SetCellValue(targetSheet, "E2", 2.5);
SetCellValue(targetSheet, "E3", 2);
SetCellValue(targetSheet, "E4", 2);
SetCellValue(targetSheet, "E5", 2.75);
```

Після заповнення фрагменту листа даною інформацією отримаємо такий вигляд листа у програмі Excel (рис. 11.4):

	A	B	C	D	E
1		Q1	Q2	Q3	Q4
2	N. America	1.5	2	1.5	2.5
3	S. America	2	1.75	2	2
4	Europe	2.25	2	2.5	2
5	Asia	2.5	2.5	2	2.75

Рисунок 11.4 – Результат створення таблиці даних

ж) Визначити діапазон для діаграми (лістинг 11.13):

Лістинг 11.13 – Задання діапазону діаграми

```
dataRange = targetSheet.get_Range("A1", "E5");
```

з) Створити об’єкт діаграми (лістинг 11.14):

Лістинг 11.14 – Створення діаграми

```
chartObjects = (ChartObjects)(targetSheet.ChartObjects(paramMissing));
newChartObject = chartObjects.Add(0, 100, 300, 300);
```

і) Далі за допомогою методу ChartObject.Chart.ChartWizard необхідно створити саму діаграму (лістинг 11.15):

Лістинг 11.15 – Безпосереднє створення діаграми за даними

```
newChartObject.Chart.ChartWizard(dataRange, XlChartType.xl3DColumn,
paramChartFormat, XlRowCol.xlRows, paramCategoryLabels, paramSeriesLabels,
paramHasLegend, paramTitle, paramCategoryTitle, paramValueTitle,
paramMissing);
```

Метод ChartObject.Chart.ChartWizard вміщує список певних параметрів, за допомогою яких можна створювати діаграму по даним, вибору її типу, передачі даних, створення легенди, назви тощо.

Параметр Source (джерело) вміщує у собі дані для нової діаграми (гістограми). Попередній код використовує змінну dataRange для специфікації комірок, що містять дані гістограми.

Параметр Gallery (галерея) дозволяє вибрати вид гістограми для створення. Його властивість XlChartType.xl3DColumn визначає трьохвимірну стовпчасту діаграму.

Параметр Format (формат) визначає число опцій для створення діаграми з подальшим її форматуванням. Це число варіації від 1 до 10, залежно від типу галереї.

Параметр PlotBy визначає, чи дані для кожної серії знаходяться у стовпцях чи рядках. Такий параметр задається і в попередньому коді для визначення даних з рядків.

Параметр CategoryLabels визначає кількість стовпців чи рядків у діапазоні ресурсів для побудови діаграми. Наприклад, код вище

використовує paramCategoryLabels для значення 0, щоб визначити необхідні рядки у нульовому рядку діапазону.

Параметр SeriesLabels є числом, яке визначає кількість рядків чи стовпців у діапазоні ресурсів, що містять послідовні ярлики. Він використовує paramSeriesLabels – змінну, що початково встановлена на значенні 0, яка визначає, що послідовні ярлики містяться в нульовому стовпці з діапазону ресурсів.

Параметр HasLegend визначає, чи включається легенда до діаграми.

Параметр Title визначає текстову назву діаграми.

Параметри CategoryTitle, CategoryTitle визначає показ назви заголовків осей графіка.

й) Зберегти файл і коректно вивільнити об’єкти (лістинг 11.16):

Лістинг 11.16 – Збереження до файлу утворених результатів

```
newWorkbook.SaveAs(paramWorkbookPath, paramMissing, paramMissing,
    paramMissing, paramMissing, paramMissing,
    XlSaveAsAccessMode.xlNoChange, paramMissing, paramMissing,
    paramMissing, paramMissing, paramMissing);

// Release the references to the Excel objects.
newChartObject = null;
chartObjects = null;
dataRange = null;
targetSheet = null;
// Close and release the Workbook object.
if (newWorkbook != null)
{
    newWorkbook.Close(false, paramMissing, paramMissing);
    newWorkbook = null;
}
// Quit Excel and release the ApplicationClass object.
if (excelApplication != null)
{
    excelApplication.Quit();
    excelApplication = null;
}
GC.Collect();
GC.WaitForPendingFinalizers();
GC.Collect();
GC.WaitForPendingFinalizers();
```

В результаті буде побудована задана діаграма (рис. 11.5) і збережена на жорсткому диску з подальшим зверненням до неї:

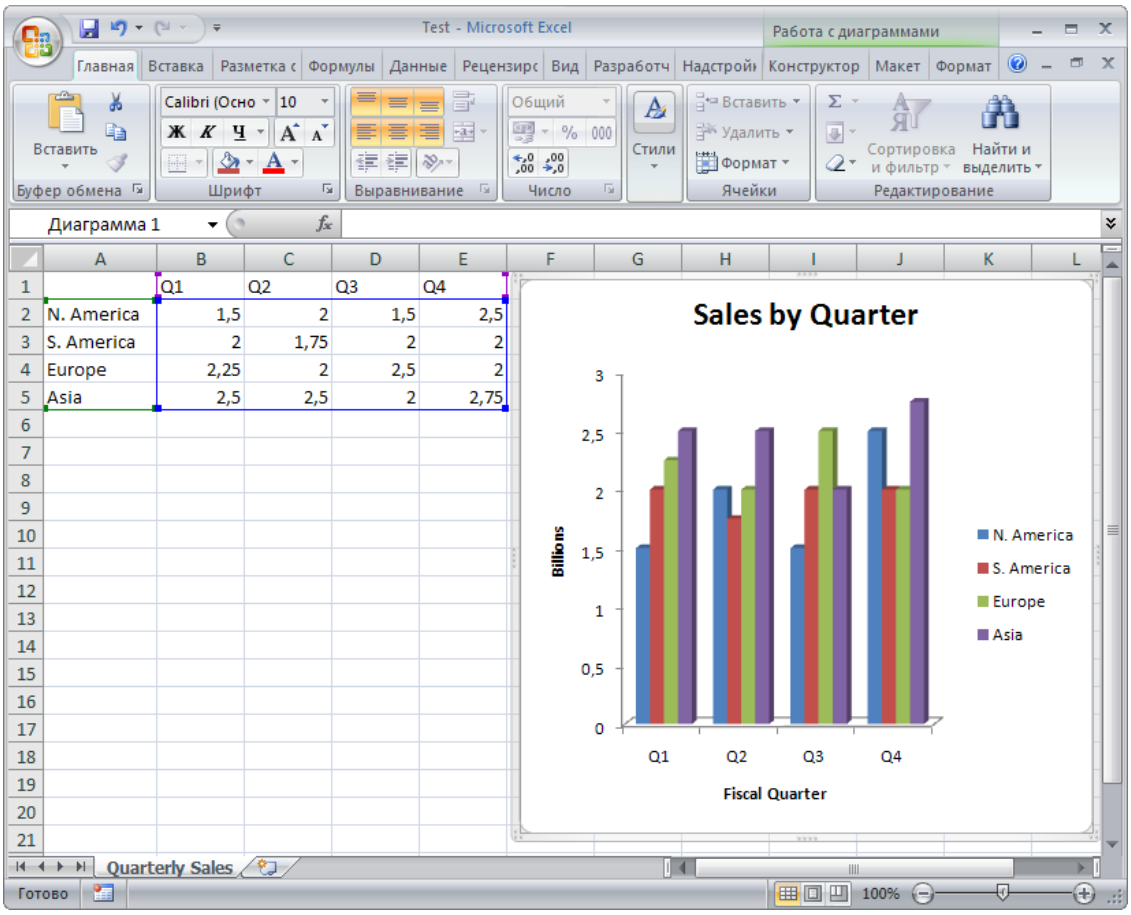


Рисунок 11.5 – Вигляд побудованої діаграми

и) Доступ до побудованої діаграми (лістинг 11.17):

Лістинг 11.17 – Відкриття створеної діаграми

```
// Create an instance of the Excel ApplicationClass object.
excelApplication = new ApplicationClass();

// Open the source workbook.
Workbook excelWorkBook =
    excelApplication.Workbooks.Open(paramWorkbookPath,
    paramMissing, paramMissing, paramMissing, paramMissing,
    paramMissing, paramMissing, paramMissing, paramMissing,
    paramMissing, paramMissing, paramMissing, paramMissing,
    paramMissing, paramMissing);

// Get the worksheet that contains the chart.
Worksheet targetSheet =
    (Worksheet)(excelWorkBook.Worksheets["Quarterly Sales"]);

// Get the ChartObjects collection for the sheet.
```

Продовженн лістингу 11.17:

```
ChartObjects chartObjects =
    (ChartObjects)(targetSheet.ChartObjects(paramMissing));

// Get the chart to modify.
```

```
ChartObject existingChartObject =
    (ChartObject)(chartObjects.Item("Sales Chart"));

// Use the ChartWizard method to modify the chart's title and legend
// properties.
existingChartObject.Chart.ChartWizard(paramMissing, paramMissing,
    paramMissing, paramMissing, paramMissing, paramMissing, false,
    "Quarterly Sales", paramMissing, paramMissing, paramMissing);
```

Для побудови такої ж діаграми і відкриття її в Excel без збереження до файлу всі попередні пункти та кодову реалізацію можна значно спростити таким чином:

1. Не задається ім’я файлу (змінна paramWorkbookPath не потрібна).
2. Відсутній код, який зберігає файл і закриває книгу, а також звільняє об’єкт ApplicationClass.
3. Для того, щоб показати вікно Excel на екрані необхідно задати властивість Visible:

excelApplication.Visible = true;

3. Побудова звітів з програмним управлінням Microsoft Word

Для того, щоб отримати доступ до управління встановленим на комп’ютері екземпляром Microsoft Word, необхідно додати в проект посилання на COM-об’єкт автоматизації. Це робиться за допомогою контекстного меню проекту та вибору додання необхідних посилань, як показано на рисунку 11.6. В даному випадку у коді програми декларується простір імен (using Microsoft.Office.Interop.Word;) і виробляється необхідний план дій.

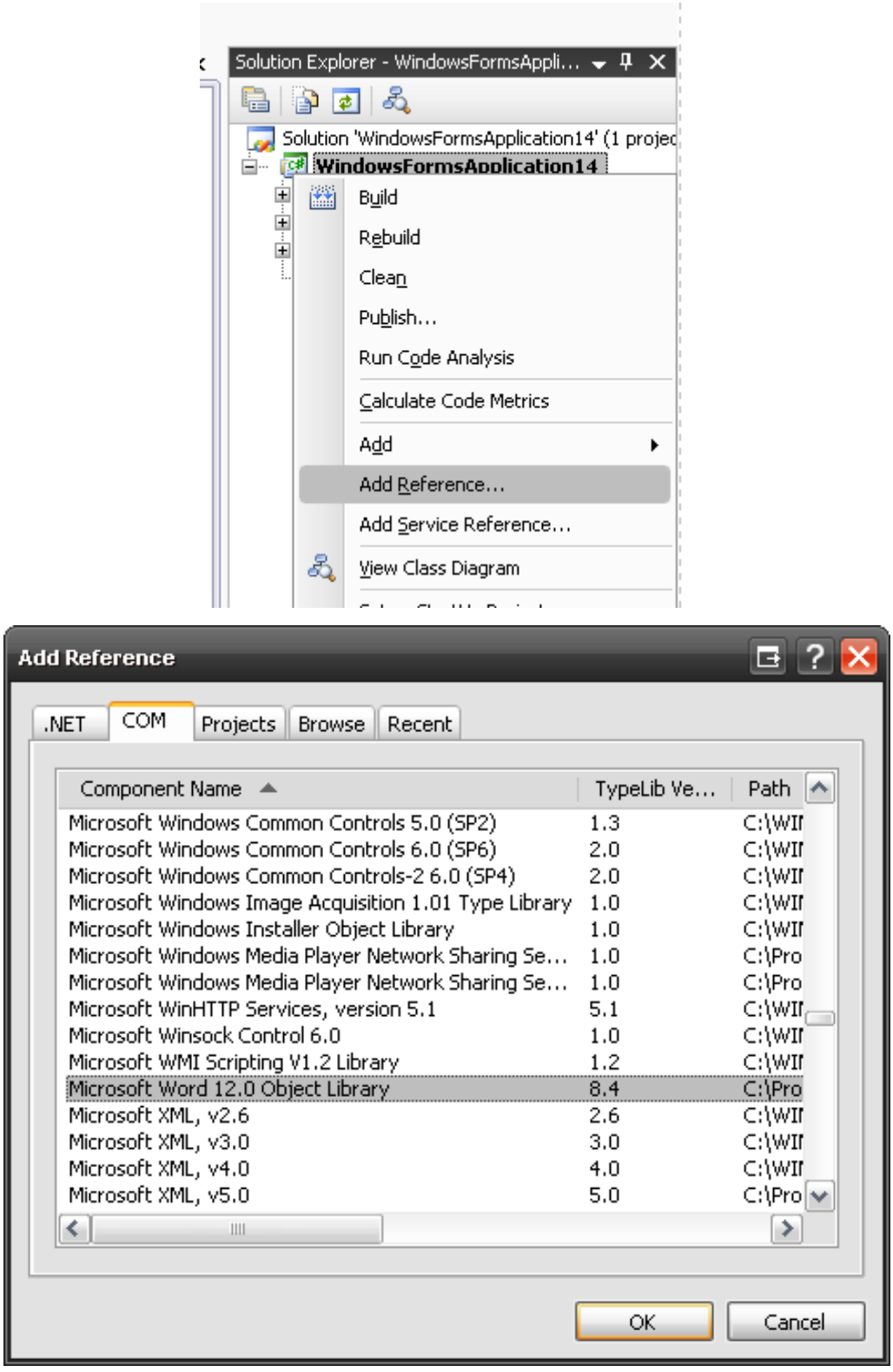


Рисунок 11.6 - Додання посилання на Microsoft Word у програмі

До плану дій слід віднести такі кроки:

1. Задати допоміжні змінні.
2. Відкрити об’єкт автоматизації Microsoft Word.

3. Створити новий документ.

4. Заповнити документ даними.

5. Зберегти документ та закрити Word чи залишити відкрите вікно на екрані.

Розглянемо розробку по даному плану дій на прикладі коду створення звіту та його виклику у програмі. Для цього необхідно:

а) Задати допоміжні змінні (лістинг 11.18):

Лістинг 11.18 – Задання допоміжних змінних

```
object oMissing = System.Reflection.Missing.Value;
object oEndOfDoc = "\\endofdoc"; // Закладка, яка означає кінець документу
```

б) Відкрити об’єкт автоматизації Word і створити новий документ (лістинг 11.19):

Лістинг 11.19 – створення нового документу

```
Word._Application oWord;
Word._Document oDoc;
oWord = new Word.Application();
oWord.Visible = true;
oDoc = oWord.Documents.Add(ref oMissing, ref oMissing,
    ref oMissing, ref oMissing);
```

в) Заповнити документ даними можна за допомогою використання коду по вставці тексту і розривів сторінки (лістинг 11.20), вставки та використання таблиць (лістинг 11.21), вставки спеціальної діаграми документу MSGraph (лістинг 11.22), вставки тексту з перевіркою відстані від верхнього краю документу (лістинг 11.23) тощо.

Лістинг 11.20 - Вставка тексту та розриву сторінки

```
//Вставити абзац на початку документа
Word.Paragraph oPara1;
oPara1 = oDoc.Content.Paragraphs.Add(ref oMissing);
oPara1.Range.Text = "Програмно згенерований документ";
oPara1.Range.Font.Bold = 1;
oPara1.Format.SpaceAfter = 24;    //24 pt відстань після параграфу
oPara1.Range.InsertParagraphAfter();
```

Продовження лістингу 11.20:

```
//Вставити абзац в кінці документа
Word.Paragraph oPara2;
object oRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
oPara2 = oDoc.Content.Paragraphs.Add(ref oRng);
oPara2.Range.Text = "Заголовок";
oPara2.Format.SpaceAfter = 6;
oPara2.Range.InsertParagraphAfter();

//Вставити інший абзац в кінці документа
Word.Paragraph oPara3;
oRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
oPara3 = oDoc.Content.Paragraphs.Add(ref oRng);
oPara3.Range.Text = "Це речення звичайного тексту.";
oPara3.Range.Font.Bold = 0;
oPara3.Format.SpaceAfter = 24;
oPara3.Range.InsertParagraphAfter();

//Вставити розрив сторінки та текст на початку наступної сторінки
object oCollapseEnd = Word.WdCollapseDirection.wdCollapseEnd;
object oPageBreak = Word.WdBreakType.wdPageBreak;
wrdRng.Collapse(ref oCollapseEnd);
wrdRng.InsertBreak(ref oPageBreak);
wrdRng.Collapse(ref oCollapseEnd);
wrdRng.InsertAfter("Зараз ми на наступній сторінці ");
wrdRng.InsertParagraphAfter();
```

Лістинг 11.21 – Вставка таблиць

```
//Вставити таблицю розміром 3 на 5 комірок, заповнити її даними,
//виділити перший рядок жирним шрифтом та курсивом
Word.Table oTable;
Word.Range wrdRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
oTable = oDoc.Tables.Add(wrdRng, 3, 5, ref oMissing, ref oMissing);
oTable.Range.ParagraphFormat.SpaceAfter = 6;
int r, c;
string strText;
for (r = 1; r <= 3; r++)
    for (c = 1; c <= 5; c++)
    {
        strText = "Ряд." + r + "Стовб." + c;
        oTable.Cell(r, c).Range.Text = strText;
    }
oTable.Rows[1].Range.Font.Bold = 1;
oTable.Rows[1].Range.Font.Italic = 1;

//Вставити таблицю 5 х 2, заповнити даними, змінити ширину стовбчиків
wrdRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
oTable = oDoc.Tables.Add(wrdRng, 5, 2, ref oMissing, ref oMissing);
oTable.Range.ParagraphFormat.SpaceAfter = 6;
for (r = 1; r <= 5; r++)
    for (c = 1; c <= 2; c++)
    {
        strText = "Ряд." + r + "Стовб." + c;
        oTable.Cell(r, c).Range.Text = strText;
    }
//Змінити ширину стовбчиків 1 і 2
oTable.Columns[1].Width = oWord.InchesToPoints(2);
oTable.Columns[2].Width = oWord.InchesToPoints(3);
```

Лістинг 11.22 - Вставка діаграми MSGraph

```
//Вставити діаграму
Word.InlineShape oShape;
object oClassType = "MSGraph.Chart.8";
wrdRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
oShape = wrdRng.InlineShapes.AddOLEObject(ref oClassType, ref oMissing
    ref oMissing, ref oMissing, ref oMissing,
    ref oMissing, ref oMissing, ref oMissing);
// Доступ до об'єкту діаграми
object oChart;
object oChartApp;
oChart = oShape.OLEFormat.Object;
oChartApp = oChart.GetType().InvokeMember("Application",
    BindingFlags.GetProperty, null, oChart, null);
//Змінити тип діаграми
object[] Parameters = new Object[1];
Parameters[0] = 4; //xlLine = 4
oChart.GetType().InvokeMember("ChartType", BindingFlags.SetProperty,
    null, oChart, Parameters);
//Закрити MSGraph.
oChartApp.GetType().InvokeMember("Quit",
    BindingFlags.InvokeMethod, null, oChartApp, null);
//Задати ширину діаграми
oShape.Width = oWord.InchesToPoints(6.25f);
oShape.Height = oWord.InchesToPoints(3.57f);
```

Лістинг 11.23 - Вставка тексту з перевіркою відстані від верхнього краю документа

```
object oPos;
double dPos = oWord.InchesToPoints(7);
oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range.InsertParagraphAfter();
do
{
    wrdRng = oDoc.Bookmarks.get_Item(ref oEndOfDoc).Range;
    wrdRng.ParagraphFormat.SpaceAfter = 6;
    wrdRng.InsertAfter("Рядок тексту");
    wrdRng.InsertParagraphAfter();
    oPos = wrdRng.get_Information
                (Word.WdInformation.wdVerticalPositionRelativeToPage);
}
while (dPos >= Convert.ToDouble(oPos));
```

ЛК.12 – ВВЕДЕННЯ ДО РОЗРОБКИ WEB-РІШЕНЬ НА ОСНОВІ ASP.NET

Перелік питань

1. Архітектура і основні принципи побудови Web-рішень.
2. Представлення інформації за допомогою мови HTML.
3. Основні обмеження і особливості побудови Web-рішень.
4. Серверна частина Web-рішення.
5. Простір імен ASP.NET.
6. Модель програмного коду Web-сторінки ASP.NET.
7. Структура папки ASP.NET.
8. Цикл компіляції сторінки ASP.NET.
9. Ланцюг наслідування типу Page.
10. Взаємодія з вхідним HTTP-запитом.
11. Доступ до властивостей браузера.
12. Доступ до даних форми.
13. Властивість IsPostBack.
14. Взаємодія з результуючою HTTP-відповіддю.
15. Цикл існування Web-сторінки ASP.NET.
16. Елементи управління ASP.NET.

На самостійне вивчення:

1. Подія Error [1, С.1051-1052].

1. Архітектура і основні принципи побудови Web-рішень

Web – рішення дуже сильно відрізняються від традиційних програм для настільних систем. Першою очевидною відміною є те, що будь-яке реальне Web – рішення має на увазі використання, як мінімум, двох з’єднаних у мережу комп’ютерів. Залучені машини повинні погоджувати використання певного мережного протоколу для успішного виконання відправки та прийняття даних. Мережним протоколом, який з’єднує комп’ютери, є

протокол HTTP (протокол передачі гіпертексту). Він забезпечує безпосередній зв'язок між клієнтом та сервером (рис. 12.1):

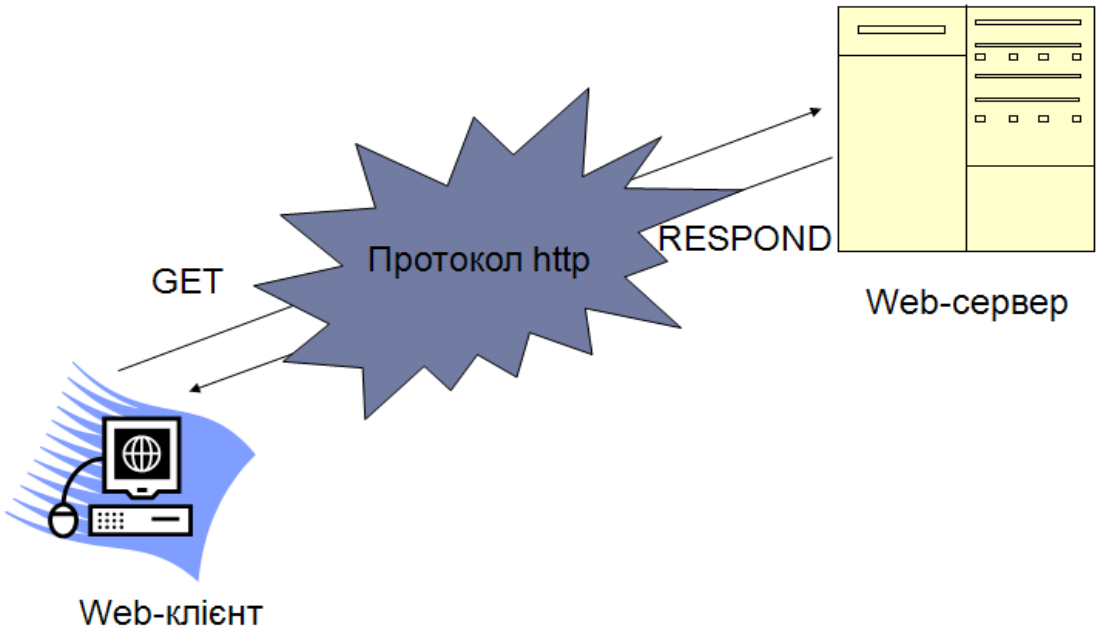


Рисунок 12.1 - Архітектура Web-рішень

З 1997 року використовується протокол RFC 2068: Протокол HTTP 1.1. Його пояснення та призначення демонструються на рисунку 12.2. Він увібрав у себе більше вимог та надійності, порівнюючи з попередниками:

RFC 2068 HTTP/1.1 January 1997

1 Introduction

1.1 Purpose

The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990. The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945 [6], improved the protocol by allowing messages to be in the format of MIME-like messages, containing metainformation about the data transferred and modifiers on the request/response semantics. However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, and virtual hosts. In addition, the proliferation of incompletely-implemented applications calling themselves "HTTP/1.0" has necessitated a protocol version change in order for two communicating applications to determine each other's true capabilities.

This specification defines the protocol referred to as "HTTP/1.1". This protocol includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features.

Рисунок 12.2 – Ціль використання RFC 2068: Протокол HTTP 1.1

Коли машина-клієнт запускає Web-браузер, генерується HTTP-запит доступу до конкретного ресурсу на віддаленій машині-сервері. Протокол HTTP – це текстовий протокол, створений на стандартній парадигмі запитів та відповідей.

Для початку роботи необхідно сформувати HTTP-запит. Це необхідно для звернення до ресурсів на сервер. У даному випадку використовується така синтаксична модель: ‹Метод› ‹URI› HTTP/‹Версія›. Серед методів, що входять до запиту, можуть бути OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE та CONNECT.

В даному випадку приклад реалізації HTTP-запиту можна побачити у наступних рядках реалізації:

Запит:

GET /wiki/HTTP HTTP/1.1

Host: ru.wikipedia.org

User-Agent: Mozilla/4.0 (compatible; MSIE 6.0; Windows NT 5.1)

Connection: close

Відповідь:

HTTP/1.0 200 OK

Server: Apache

Content-Language: ru

Content-Type: text/html;

charset=utf-8 Content-Length: 1234

(далі йде текст сторінки).

Код стану HTTP - це тризначний код результату запиту. Код стану HTTP складається з числової коди стану і пояснюючої фрази. Код стану призначений для використання автоматами, а пояснююча фраза призначена для користувачів.

Перша цифра коду стану визначає клас відповіді:

1xx: Інформаційні коди - запит отриманий, продовжується обробка. 2xx: Успішні коди - дія була успішно отримана, зрозуміла і оброблена.

3xx: Коди перенаправлення - для виконання запиту повинні бути зроблені подальші дії. 4xx: Коди помилок клієнта - запит має поганий синтаксис або не може бути виконаний. 5xx: Коди помилок сервера - сервер не в змозі виконати допустимий запит.

GET-запит повністю передається у адресному рядку, його розмір обмежений, а параметри доступні користувачеві. Приклад GET-запита у адресному полі браузера:

http://example.com/data.cgi?id=17:30576-10&action=5.

У POST-запиті параметри передаються окремо від адресного рядку, розмір цього запиту може бути як завгодно великим.

2. Представлення інформації за допомогою мови HTML

HTML - HyperText Mark-up Language – мова розмітки гіпертексту, яка служить для створення гіпертекстових сторінок.

Мова HTML розроблена для створення WWW-документів у 1991 році в Європейській лабораторії фізики елементарних частинок (CERN) на основі специфікацій стандартної метамови описів *SGML* (Standard Generalized Markup Language, ISO-8879).

Вона має свій особливий синтаксис, за допомогою якого виконуються різноманітні передбачені команди, відбувається посилання запиту на сервер. Ключеві слова (текстові маркери, мітки, теги) мови розмітки сторінок гіпертексту записуються згідно з наступними правилами:

1) теги HTML заключаються в кутові дужки:

<МАРКЕР>;

2) теги можуть записуватися великими або малими латинськими літерами:

<МАРКЕР> або <маркер>;

3) кожен тег може мати свою сферу дії, котра відміняється таким же (парним) тегом с косою рискою попереду ("слешем")- кінцевим тегом:

<МАРКЕР> сфера дії команди </МАРКЕР>;

4) деякі теги не мають парних кінцевих маркерів, наприклад
,<HR>, <META> або <IMG ...>;

5) тег складається з ключового слова, за яким можуть слідувати атрибути (параметри), можливо, з значеннями атрибутів:

<МАРКЕР АТРИБУТ> або <МАРКЕР АТРИБУТ = значення >;

6) деякі теги можуть застосовуватися лише в сфері дії других тегів, які називаються контейнерами. Наприклад пара <TR> і </TR> лише всередині тегів <TABLE> і </TABLE>, а теги <TD> і </TD> можуть бути вкладені лише в теги <TR> і </TR>.

Теги поділяються на групи в залежності від їх призначення. Деякі з них можна віднести до таких груп:

1. Ті, що відносяться до логічної структури документа, такі як:

<ADDRESS> Адрес </ADDRESS>;

<AUTHOR> Автор </AUTHOR>;

<DIV> Розділ </DIV>;

<H1> Заголовок </H1>;

<QUOTE> Цитата </QUOTE>;

<SAMP> Приклад </SAMP>.

2. Ті, що відносяться до оформлення (представлення) документа, наприклад:

<B> жирний </B> ;

<FONT COLOR="red"> червоний </FONT>;

<FONT SIZE=+1> розмір +1 </FONT>;

<STIKE> перекреслений </STRIKE>.

Гіпертекстова сторінка знаходиться між парними тегами:

<HTML> і </HTML>

Коментарі записуються всередині тегів коментаря і не відображаються браузером:

<!-- коментар -->.

Коментарі спеціального вигляду визначають для браузера версію мови HTML, застосованої в документі. В цьому випадку коментар <!DOCTYPE ...> є першим елементом HTML-документа. Він визначає версію HTML, наприклад: <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 3.2//EN">

Сторінка складається з титульної частини <HEAD> і тіла сторінки <BODY>.

Для кращого розуміння структури сторінки варто розглянути приклад розмітки пустої Web-сторінки:

```
<HTML>
 <HEAD>
  <TITLE> Заголовок сторінки </TITLE>
 </HEAD>
 <BODY>
  <!-- Тіло сторінки -->
 </BODY>
</HTML>.
```

Текст сторінки розглядається браузером як один довгий рядок, повторюючи переводи строки. Для форматування тексту сторінки використовуються наступні теги:

 <! перехід на нову строку >;

<P> Новий абзац в тексті </P>;

<HR> <! горизонтальна лінія >;

<CENTER> Вирівняний по центру текст </CENTER>.

Якщо необхідно попередньо відформатоване розміщення тексту: <PRE>. Заздалегідь відформатований текст з збереженням відступів і пропусків: </PRE>

Для виділення в тексті розділів використовується пара тегів <DIV> з можливим зазначенням вирівнювання за допомогою атрибута ALIGN="LEFT| RIGHT | CENTER":

<DIV ALIGN= "вирівнювання" > Текст розділу </DIV>.

Форматування фрагментів різного нарису можна здійснити за допомогою таких тегів:

<B> жирний нарис </B>;

<I> курсив </I>;

<S> перекреслений текст </S>;

<U> підкреслений текст </U>;

<TT> моноширинний текст </TT>;

_{нижній індекс};

^{верхній індекс}.

За допомогою тегів також можна задавати різні шрифти тексту:

<FONT FACE=" назва шрифту" > ... </FONT>

Наприклад, так задається відображення тексту визначеної гарнітури:

<FONT FACE="Times New Roman"> текст </FONT>

Для визначення відносного розмірів букв в тексті використовуються теги:

<BIG> Букви більшого розміру </BIG>

<SMALL> Букви меншого розміру </SMALL>

Можна також указувати абсолютний розмір букв числами от 1 до 7 або відносний розмір букв: додатне або від’ємне відхилення відносно базового розміру:

<FONT SIZE=n> </FONT> <!-- n=1,2,...,7-->

<FONT SIZE="+|-n"> </FONT> <!-- n=-1,2,...|+1,2,...-->

Базовий розмір тексту указується тегом <BASEFONT SIZE= "розмір" >.

Для визначення кольорів окремих фрагментів тексту використовується тег: <FONT COLOR=#rrggbb|"цвет"> ... </FONT>.

Кольори задаються шістнадцятирічними числами по схемі RRGGBB ("червоний - зелений - синій") або назвою кольору. Наприклад, зелений колір можна указати двома способами:

<FONT COLOR=00FF00> зелений </FONT>;

<FONT COLOR="green"> зелений </FONT>.

Глобальні кольори тексту и фона на сторінці, а також кольори посилань (невідвіданих, відвіданих і виділених) указуються в маркері опису тіла сторінки:

<BODY BGCOLOR=#rrggbb ;

TEXT=#rrggbb;

LINK=#rrggbb;

VLINK=#rrggbb;

ALINK=#rrgbb>.

Нумерований список задається парними тегами <OL> і </OL>, котрі обрамляють теги <LI> і </LI>, які містять в собі окремі елементи списку. Допускається для короткості опускати кінцевий маркер </LI>.

Елементи списку в тексті розміщуються стовпцями з нумерацією, тип яких можна задавати атрибутом TYPE="1 | i | I |a | A".

<OL TYPE="тип">;

<LI> Перший елемент </LI>;

<LI> Другий елемент </LI>...

</OL>.

Маркірований список задається парними маркерами <UL> та </UL> с вкладеними маркерами <LI> та </LI>.

За допомогою атрибута TYPE="CIRCLE| DISC | SQUARE".

<UL TYPE="тип">;

<LI> Перший елемент </LI>;

<LI> Другий елемент </LI>...

</UL>.

Можна також використовувати картинки. Для цього вказується її місце знаходження, ширина, висота, вирівнювання тощо:

<IMG SRC="URL файла зображення"

HEIGHT="висота"

WIDTH="ширина"

ALIGN="LEFT|CENTER|RIGHT"

BORDER=n

ALT="альтернативний текст“/>

Для прикладу можна розглянути такий код:

<img
border=0
src=“/images/x2.gif”
width=12 height=12
alt="Close menu"
align=right/>

Можна оформляти та створювати гіперпосилання у HTML:

– <A HREF="гіперпосилання"> *Опис в тексті* </A>;
– <A HREF ="гіперпосилання“><*IMG* .../></A> .

Для організації посилань в межах сторінки потрібно об’явити мітки для переходів (так звані "якоря") за допомогою тега: <A NAME= "ім’я якоря">.

Після цього можна указувати гіперпосилання на ці якоря, розміщені на поточній або іншій сторінці, при цьому перед іменем якоря ставиться символ #:

<A HREF="#end"> на тій же сторінці </A> ;

<A HREF="faq.html#index"> в тім же каталозі </A>;

<A HREF="http://www.what.org/path/page.htm#anchor"> зовнішній </A>.

Сторінка може містити програму на одній з мов сценаріїв, частіше за все на JavaScript, розміщену між парою маркерів:

<SCRIPT LANGUAGE="JavaScript">

програма на мові сценарію

</SCRIPT>

Гіперпосилання на інші ресурси оформляється у вигляді уніфікованого покажчика ресурсів - URL (Uniform Resource Locator) за наступними правилам: "протокол: адреса“.

Протокол доступу, визначаючий вид ресурсу, і мережева адреса, яка описує його положення, можуть приймати різні значення для різних

інформаційних ресурсів Internet (зірочкою * позначені нестандартні визначення ресурсів):

- ресурс FTP - ftp://им’я_користувача:пароль@internet_им’я/шлях;

- гіпертекст WWW - http://internet_им’я/шлях/програма?параметр+параметр;

- ресурс e-mail - mailto:им’я_користувач@internet_им’я/шлях;

- локальний ресурс - file:диск:\шлях\файл.

Мова JavaScript дозволяє застосовувати замість URL вирази мови сценаріїв динамічно формуючі результат показу на посилання. При цьому гіперпосилання оформлюються наступним чином:

<A HREF="javascript:вираз"> обчислене посилання </A>

Наприклад:

<A HREF="javascript:alert('Hello!')"> Привітання </A>

В сторінці можна вбудовувати таблиці, опис кожної із яких задається маркерами:

<TABLE>

<! опис таблиці >

</TABLE>

Заголовок таблиці розміщується між маркерами <CAPTION> і </CAPTION>. Наприклад:

<TABLE ALIGN="center" BORDER=1>

<CAPTION> Заголовок </CAPTION>

</CAPTION>

<! ...опис строк таблиці... >

</TABLE>

Таблиця складається з строк, які повинні знаходитися між маркерами <TR> і <TR>, в яких описані комірки таблиці.

<TABLE>

<TR>

<! ...опис комірки 1-й строки... >

```
</TR>
<TR>
<! ...опис комірки 2-й строки... >
</TR>
<TR>
<! ...опис комірки 3-й строки... >
</TR>
</TABLE>
```

Кожен рядок таблиці складається з заголовків комірок, які знаходяться між тегами <TH> і </TH> або комірок даних між маркерами <TD> і <TD>. Ось так виглядає типова таблиця з 3-х рядків (з них перша - заголовок) і 4-х колонок:

```
<TABLE>
<TR>
<TH>One</TH>
<TH>Two</TH>
<TH>Three</TH>
<TH>Four</TH>
</TR>
<TR>
<TD>1.1</TD>
<TD>1.2</TD>
<TD>1.3</TD>
<TD>1.4</TD></TR>
<TR>
<TD>2.1</TD>
<TD>2.2</TD>
<TD>2.3</TD>
<TD>2.4</TD>
</TR>
```

</TABLE>

3. Основні обмеження і особливості побудови Web-рішень

Проте Web-рішення не є ідеальним інструментом використання. Вони містять ряд обмежень, що значно звужують можливості роботи клієнтів та серверів. Серед них основними можна виділити такі:

- проблеми сумісності браузерів та клієнтських пристроїв і середовища;
- недоліки інтерфейсу на основі HTML;
- обмеження стандартних протоколів і технологій;
- навантаження на серверну складову.

У такому випадку можна використовувати альтернативні інструменти розробки. Це є багатоланкові рішення, Silverlight або Adobe AIR.

4. Серверна частина Web-рішення

Традиційний підхід до створення Web-рішення передбачає, що основна робота по обробці інформації здійснюється на сервері, клієнт лише запитує необхідну інформацію та надсилає, якщо потрібно, певні дані, наприклад, заповнюючи форми. Сучасні Web-рішення, основані на технології AJAX, можуть частину роботи перекладати на клієнта. У будь-якому разі серверна частина залишається основною складовою Web-рішення.

Для функціонування серверної частини динамічного сайту потрібен Web-сервер (наприклад, Apache чи IIS), а також певна технологія, яка дозволяє генерувати HTML-сторінки динамічно, наприклад, CGI, ASP.NET чи PHP. CGI (Common Gateway Interface) – перша технологія, яка дозволяла динамічно генерувати HTML-сторінки. Є надзвичайно простою у реалізації, однак відрізняється неефективністю використання ресурсів та трудомісткістю кодування у порівнянні з сучасними спеціалізованими технологіями, такими, як, наприклад, ASP.NET.

Можна розглянути виконання найпростішої програми на мові C# (лістинг 12.1) та результат її виконання у браузері (рис. 12.3):

Лістинг 12.1 - Найпростіша CGI-программа на C#

```
using System;
using System.Collections.Generic;
using System.Text;
namespace CGIApplication
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            string HTMLTemplate = @"Привіт, світ!<br> Дата та час на сервері:
{0}";
            Console.Write(HTMLTemplate, DateTime.Now.ToString());
        }
    }
}
```

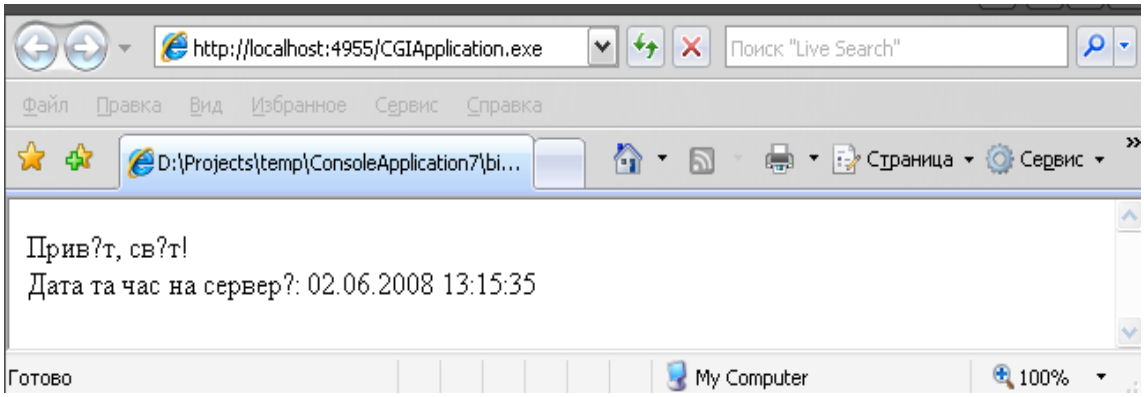


Рисунок 12.3 – Результат виконання програми

5. Простір імен ASP.NET

Технологія ASP.NET (Active Server Pages) дозволяє створювати Web-рішення, використовуючи мови програмування та можливості платформи .NET Framework. Рішення ASP.NET виконуються під управлінням Web-серверу IIS (Internet Information Services), який функціонує на операційних системах Windows Server.

Побудова рішень ASP.NET здійснюється з використанням спеціального простору імен .NET Framework (System.Web), який надає доступ до Web-

елементів управління, а також функціональності, специфічної для Web-рішень. У Microsoft Visual Studio для створення ASP.NET-рішень доступний спеціальний тип проекту (Web-site), а також локальний Web-сервер, який дозволяє запускати і відлажувати Web-рішення локально.

До основних складових простору імен ASP.NET належать (рис. 12.4):

Пространства имен	Описание
System.Web	Определяет типы, обеспечивающие коммуникацию браузера и Web-сервера (в частности, возможности запроса и ответа, обработки файлов cookie и передачи файлов)
System.Web.Caching	Определяет типы, обеспечивающие возможность кэширования для Web-приложения
System.Web.Hosting	Определяет типы, позволяющие строить пользовательские хосты для среды выполнения ASP.NET
System.Web.Management	Определяет типы, обеспечивающие управление и контроль правильности функционирования Web-приложения ASP.NET
System.Web.Profile	Определяет типы, используемые для работы с пользовательскими профилями ASP.NET
System.Web.Security	Определяет типы, позволяющие программно обеспечить безопасность узла
System.Web.SessionState	Определяет типы, обеспечивающие поддержку информации состояния для каждого пользователя (например, на основе использования сеансовых переменных состояния)
System.Web.UI System.Web.UI.WebControls System.Web.UI.HtmlControls	Определяют ряд типов, позволяющих создавать для Web-приложений программы клиента с графическим пользовательским интерфейсом

Рисунок 12.4 – Складові простору імен ASP.NET

Для переходу до створення найпростішого ASP.NET-рішення у мові програмування C# необхідно спочатку створити проект за стандартною процедурою, після чого з’явиться діалогове вікно вибору типу проекту, задання імені та розміщення (рис. 12.5):

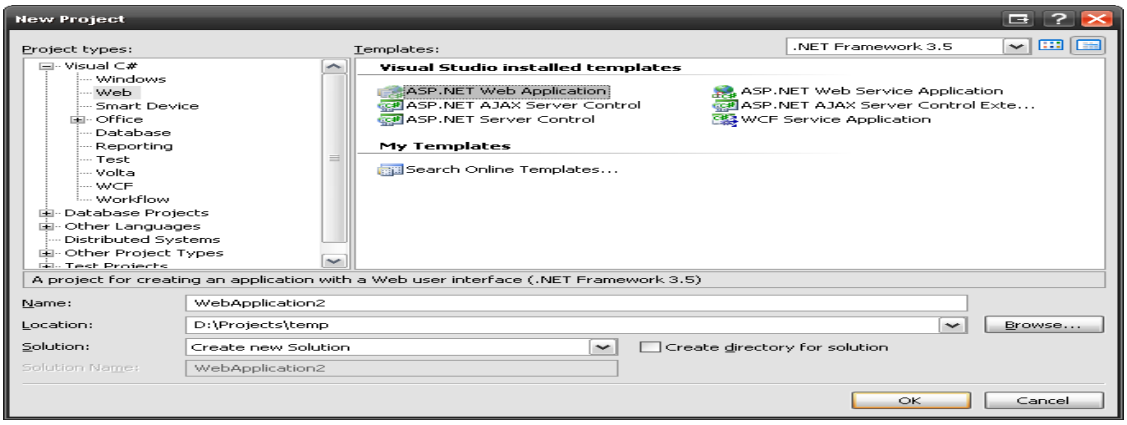


Рисунок 12.5 – Створення проекту ASP.NET-рішення

Тепер для прикладу на утвореній формі з панелі інструментів розмістимо на дизайнерській частині елемент управління Label. В даному випадку автоматично буде згенеровано код у частині, що відповідає за джерело програми (так звана Web частина) і з’явиться панель властивостей, де можна безпосередньо задати назву використовуваного інструменту. Таких елементів можна розміщувати за бажанням розробника (рис. 12.6).

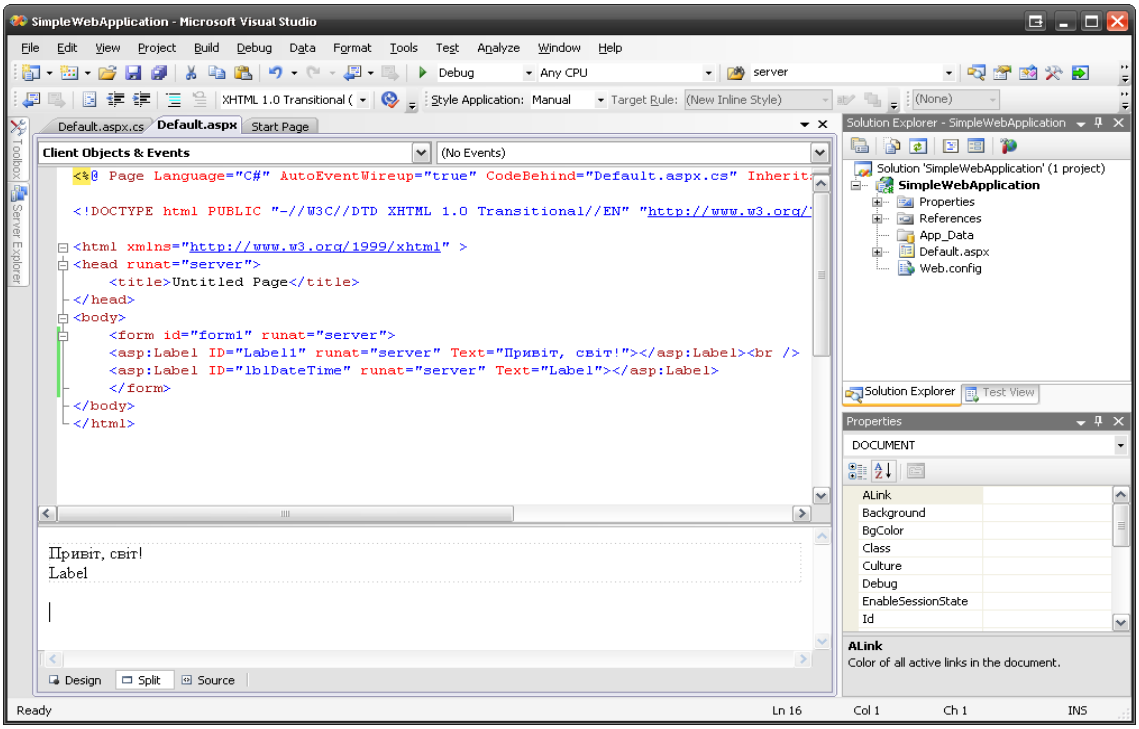


Рисунок 12.6 - Розміщення елемента управління Label

Але дана функціональність не обмежується попередньо вказаними частинами коду. Щоб забезпечити повну коректну роботу програми,

необхідно перейти до коду безпосередньо на мові програмування C#. Це можна зробити подвійним натисненням на дизайнерській частині форми та написанням відповідного необхідного коду. Приклад простої програми з розширенням попереднього (лістинг 12.2) та результати його виконання у браузері (рис. 12.7) наведено далі.

Лістинг 12.2 - Задання коду на завантаження сторінки

```
using System;
using System.Collections;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
namespace SimpleWebApplication
{
    public partial class _Default : System.Web.UI.Page
    {
        protected void Page_Load(object sender, EventArgs e)
        {
            lblDateTime.Text = "Дата та час на сервері: " +
DateTime.Now.ToString();
        }
    }
}
```

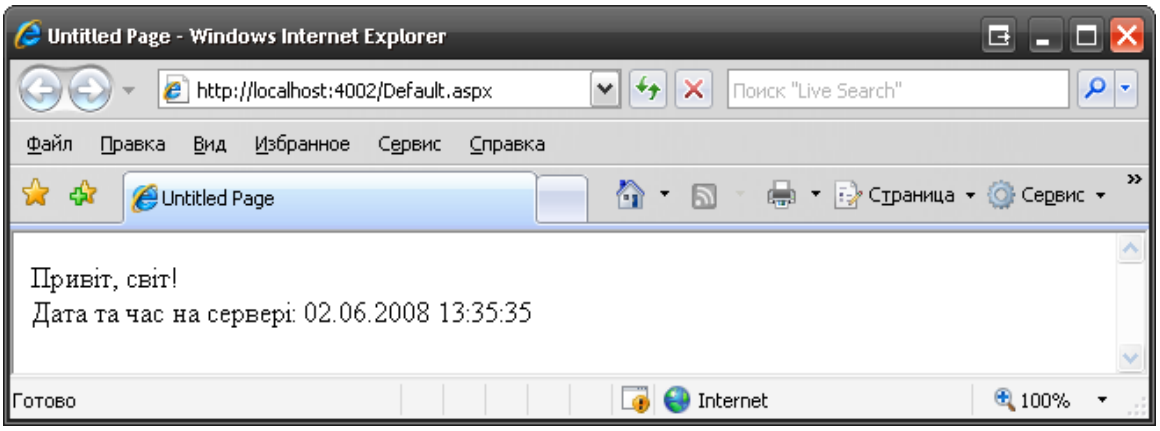


Рисунок 12.7 – Результат виконання програми

6. Модель програмного коду Web-сторінки ASP.NET

Існує два варіанти створення Web-сторінок у ASP.NET:

– одномодульна модель – сторінка у файлі .aspx містить як HTML-код зі спеціальними тегами, так і код на C#;
– модель із зовнішнім кодом підтримки (файлами code-behind) – сторінка у файлі .aspx містить лише HTML-код (зі спеціальними тегами), а код на C# зберігається у окремих файлах , які називаються файлами code-behind (рис. 12.8).

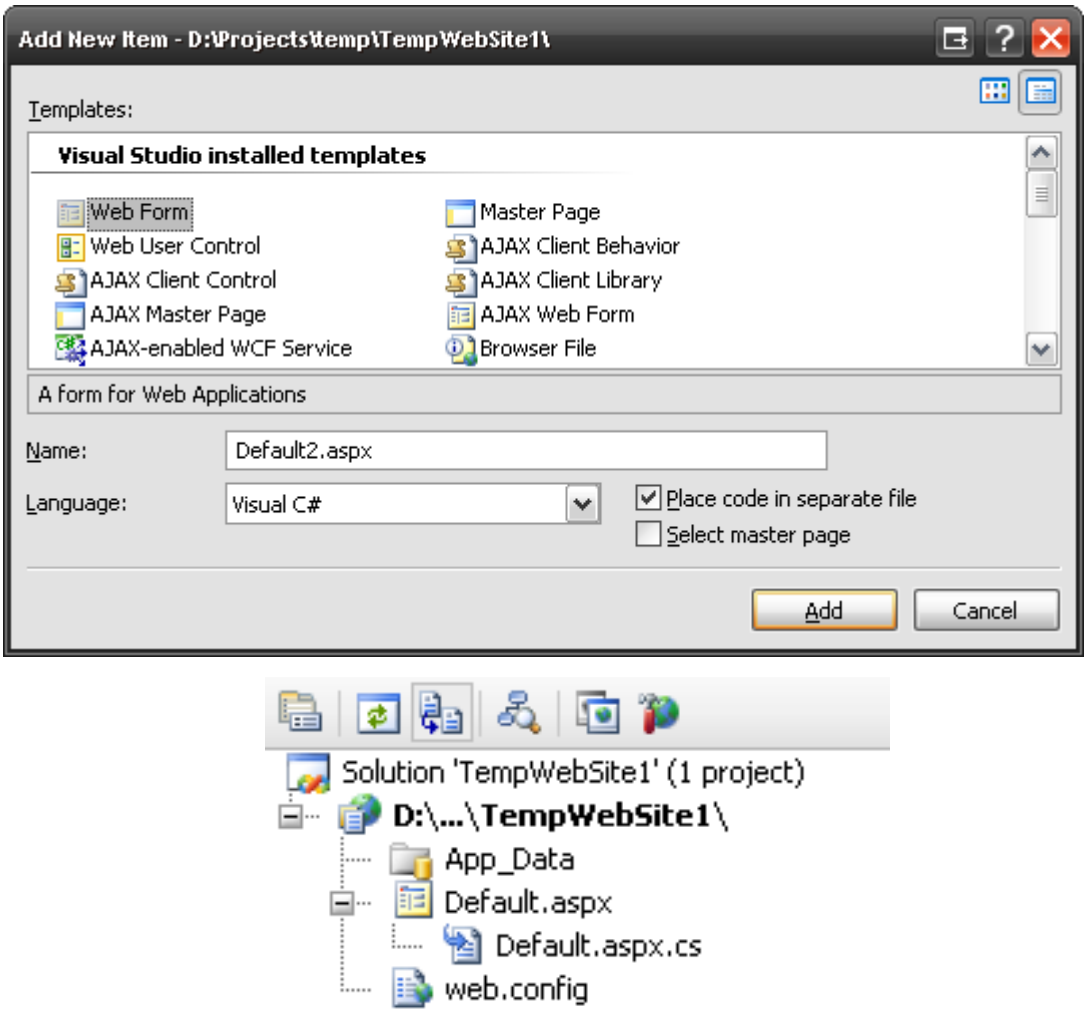


Рисунок 12.8 – Розмежування коду HTML та C#

Можна розглянути приклад коду одно модульної сторінки, який саме і реалізований за допомогою мови HTML (лістинг 12.3).

Лістинг 12.3 - Приклад коду одномодульної сторінки

```
<%@ Page Language="C#" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<script runat="server">

</script>
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

    </div>
    </form>
```

Продовження лістингу 12.3:

```
</body>
</html>
```

А тепер відповідно можна встановити різницю, порівнюючи попередній код з наступним кодом сторінки з файлом code-behind (лістинг 12.4, 12.5).

Лістинг 12.4 – Реалізація в мові HTML

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>


    </div>
    </form>
</body>
</html>
```

Лістинг 12.5 – Реалізація на мові C#

```
using System;
using System.Configuration;
using System.Data;
using System.Linq;
```

```
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
public partial class _Default : System.Web.UI.Page
{
    protected void Page_Load(object sender, EventArgs e)
    {
    }

}
```

Директива <%@Page%> визначає ASP.NET сторінку, у тому числі й вказує на те, яка модель програмного коду використовується. Основні атрибути даної директиви з їх подальшим описом демонструє рисунок 12.9:

Атрибут	Описание
CompilerOptions	Позволяет определить любые флаги командной строки (представленные одной строкой), передаваемые компилятору при обработке страницы
CodePage	Указывает имя соответствующего файла с внешним кодом поддержки
EnableTheming	Индикатор поддержки тем ASP.NET 2.0 элементами управления данной страницы *.aspx
EnableViewState	Индикатор поддержки состояния представления между запросами страницы (более подробно об этом говорится в главе 24)
Inherits	Определяет класс страницы, из которой получается данный файл *.aspx; может быть любым классом, полученным из System.Web.UI.Page
MasterPageFile	Указывает шаблон страницы, используемый в паре с текущей страницей *.aspx
Trace	Индикатор разрешения трассировки

Рисунок 12.9 – Набір атрибутів директиви <%@Page%>

Директива <%@Import%> використовується для позначення простору імен, до якого має доступ сторінка.

У більшості випадків ця директива для сторінки не застосовується, оскільки всі ASP.NET-сторінки мають доступ до основних просторів імен, зокрема (лістинг 12.6).

Лістинг 12.6 – Основні простори імен в ASP.NET-сторінці

```
System;
System.Configuration;
System.Collections.Generic;
System.IO;
System.Text;
System.Text.RegularExpressions;
System.Data;
System.Linq;
System.Web;
System.Web.Security;
System.Web.UI;
System.Web.UI.HtmlControls;
System.Web.UI.WebControls;
System.Web.UI.WebControls.WebParts;
System.Xml.Linq;
```

Блок <script> передбачає програмний код, який виконується у контексті сторінки і може бути:

– кодом на C#, який виконується на стороні сервера (якщо задано атрибут runat=“server”);
– кодом на JavaScript, який виконується на стороні клієнта у браузері (якщо атрибут runat=“server” не задано).

Приклад використання цього блоку, що автоматично генерується при створенні сторінки для управління виконанням на певній стороні взаємодії показано в лістингу 12.7:

Лістинг 12.7 - Блок <script>

```
<script runat="server">

    protected void Button1_Click(object sender, EventArgs e)
    {

    }

</script>
```

7. Структура папки ASP.NET

Проект з використанням ASP.NET має спеціальну структуру папок (не всі папки обов’язкові для проекту, вони створюються автоматично під час додавання до проекту відповідних елементів), що показані на рисунку 12.10:

Подкаталог	Описание
App_Browsers	Папка для файлов определений, которые используются для идентификации браузеров и выявления их возможностей
App_Code	Папка для исходного кода компонентов или классов, которые вы хотите компилировать, как часть вашего приложения. Программный код из этого подкаталога компилируется при запросе страниц и автоматически будет доступен вашему приложению
App_Data	Папка для хранения файлов *.mdb Access, файлов *.mdf SQL Express, XML-файлов и других наборов данных
App_GlobalResources	Папка для файлов *.resx, которые доступны из программного кода приложения
App_LocalResources	Папка для файлов *.resx, которые привязаны к конкретной странице
App_Themes	Папка с набором файлов, определяющих внешний вид Web-страницы и элементов управления ASP.NET
App_WebReferences	Папка для классов агентов, схем и других файлов, связанных с использованием Web-сервисов в приложении
Bin	Папка для скомпилированных приватных компоновочных блоков (файлы *.dll). На компоновочные блоки из папки Bin приложение ссылается автоматически

Рисунок 12.10 – Спеціальні підкаталоги ASP.NET

Додати будь-яку із цих підкаталогів можна явно, вибравши відповідне меню (WebSite /Add Folder). Але в багатьох випадках це робить саме середовище розробки при доданні до нього відповідного файлу.

8. Цикл компіляції сторінки ASP.NET

Перед виконанням сервером проект ASP.NET компілюється схожим чином до того, як компілюються звичайні .NET-програми. Процес компіляції одномодульних сторінок показано на рисунку 12.11.

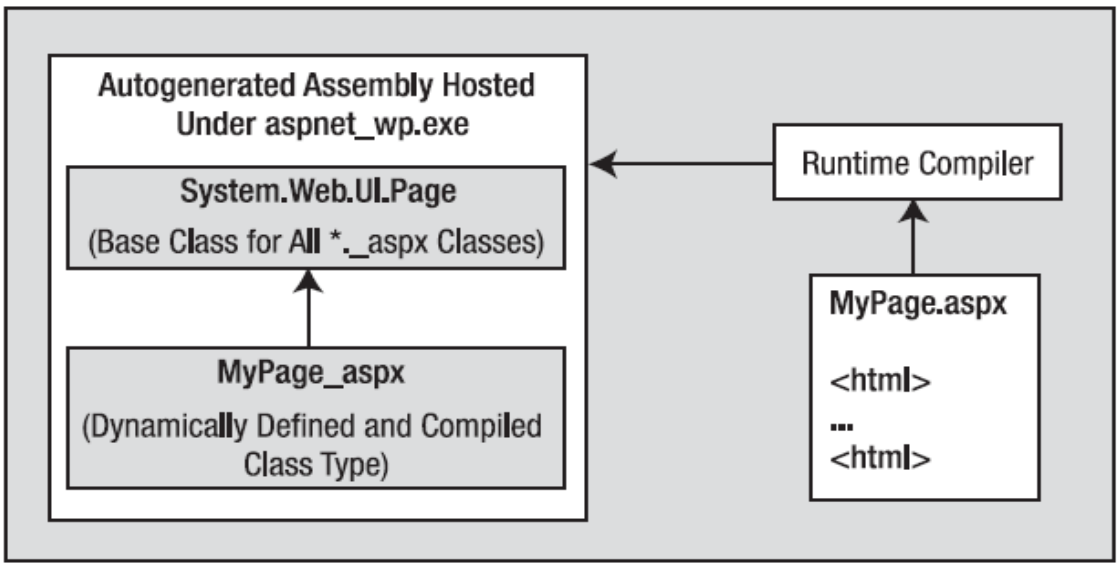


Рисунок 12.11 - Цикл компіляції сторінки

Цикл компіляції сторінок з файлами code-behind має свої відмінності, адже залежить від структури кодової розробки та складається з 2 частин коду, написаних на 2 різних мовах. Його вигляд є таким (рис. 12.12).

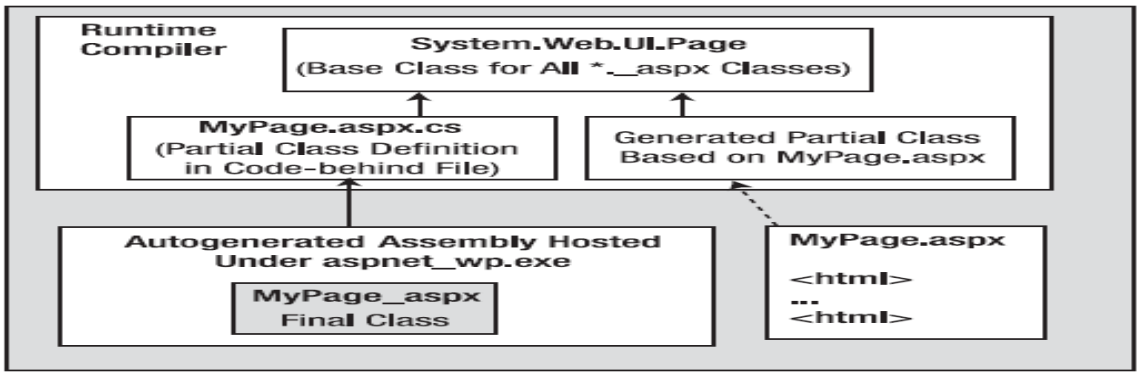


Рисунок 12.12 - Компіляція сторінок з файлами code-behind

9. Ланцюг наслідування типу Page

Кожна ASP.NET-сторінка наслідується від типу Page (повна назва System.Web.UI.Page), що наслідуваний від типу TemplateControl, який, в свою чергу, наслідуваний від типу Control, котрий напряму наслідуваний від Object. Кожен із цих базових класів вносить у файл розробки свої наслідувані частини. Рисунок 12.13 показує роль типу Page.

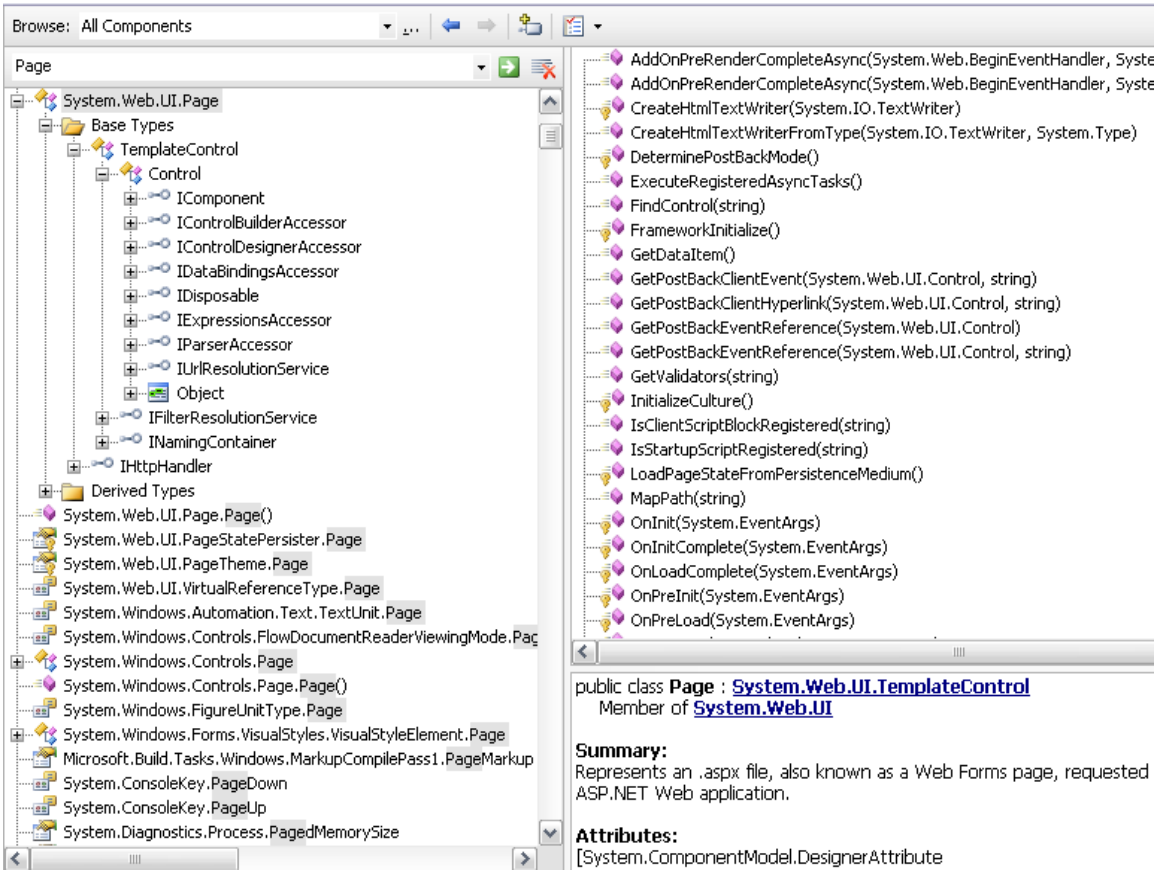


Рисунок 12.13 – Походження сторінки ASP.NET

Важливими властивостями даного типу є представлені на рисунку 12.14, які забезпечують можливість взаємодії з різними частинами сторінки, такими як Web-примітиви, змінні додатки та сеанси, запити і відповіді HTTP, теми тощо.

Свойство	Описание
Application	Позволяет взаимодействовать с переменными приложения для текущего Web-узла
Cache	Позволяет взаимодействовать с объектом кэша для текущего Web-узла
ClientTarget	Позволяет указать способ визуализации для данной страницы в зависимости от запрашивающего браузера
IsPostBack	Получает значение, являющееся индикатором загрузки страницы в ответ на вторичный запрос клиента (в отличие от первичной загрузки страницы)
MasterPageFile	Создает шаблон страницы для текущей страницы
Request	Обеспечивает доступ к текущему HTTP-запросу
Response	Позволяет взаимодействовать с исходящим HTTP-ответом
Server	Обеспечивает доступ к объекту HttpServerUtility, содержащему различные вспомогательные функции сервера
Session	Позволяет взаимодействовать с сеансовыми данными для текущего вызывающего объекта
Theme	Получает или устанавливает имя темы, используемой для текущей страницы
Trace	Обеспечивает доступ к объекту TraceContext, позволяющему записывать пользовательские сообщения в ходе сеанса отладки

Рисунок 12.14 - Властивості типу Page

10.Взаємодія з вхідним HTTP-запитом

Коли користувач здійснює запит до серверу, то програма ASP.NET може отримати доступ екземпляру типу HttpRequest, який зберігає інформацію про сам запит та клієнтське ПЗ.

Члени типа HttpRequest та їх опис показано на рисунку 12.15.

Член	Описание
Application Path	Получает путь к виртуальному каталогу приложения ASP.NET на сервере
Browser	Обеспечивает информацию о возможностях браузера клиента
Cookies	Получает коллекцию файлов cookie, отправленных браузером клиента
FilePath	Указывает виртуальный путь текущего запроса
Form	Получает коллекцию переменных формы
Headers	Получает коллекцию HTTP-заголовков
HttpMethod	Указывает метод передачи HTTP-данных, используемый клиентом (GET, POST)
IsSecureConnection	Индикатор защищенности HTTP-соединения (т.е. использования HTTPS)
QueryString	Получает коллекцию строковых переменных HTTP-запроса
RawUrl	Получает "сырой" URL текущего запроса
RequestType	Указывает метод передачи HTTP-данных, используемый клиентом (GET, POST)
ServerVariables	Получает коллекцию переменных Web-сервера
UserHostAddress	Получает IP-адрес хоста удаленного клиента
UserHostName	Получает DNS-имя удаленного клиента

Рисунок 12.15 – Члени типу HttpRequest

11.Доступ до властивостей браузера

Завдяки попередньо перекисленим властивостям, до них можна безпосередньо звертатися для отримання інформації щодо стану браузера та його вмісту. Приклад такого доступу демонструє лістинг 12.8, а результат написаної програми у браузері показано на рисунку 12.16:

Лістинг 12.8 - Доступ до властивостей браузера

```
protected void btnGetBrowserStats_Click(object sender, EventArgs e)
{
    string theInfo = "";
    theInfo += string.Format("<li>Is the client AOL? {0}</li>",
    Request.Browser.AOL);
    theInfo += string.Format("<li>Does the client support ActiveX? {0}</li>",
    Request.Browser.ActiveXControls);
    theInfo += string.Format("<li>Is the client a Beta? {0}</li>",
    Request.Browser.Beta);
    theInfo += string.Format("<li>Does the client support Java Applets?
{0}</li>",
    Request.Browser.JavaApplets);
    theInfo += string.Format("<li>Does the client support Cookies? {0}</li>",
    Request.Browser.Cookies);
    theInfo += string.Format("<li>Does the client support VBScript?
{0}</li>",
    Request.Browser.VBScript);
    lblOutput.Text = theInfo;
}
```

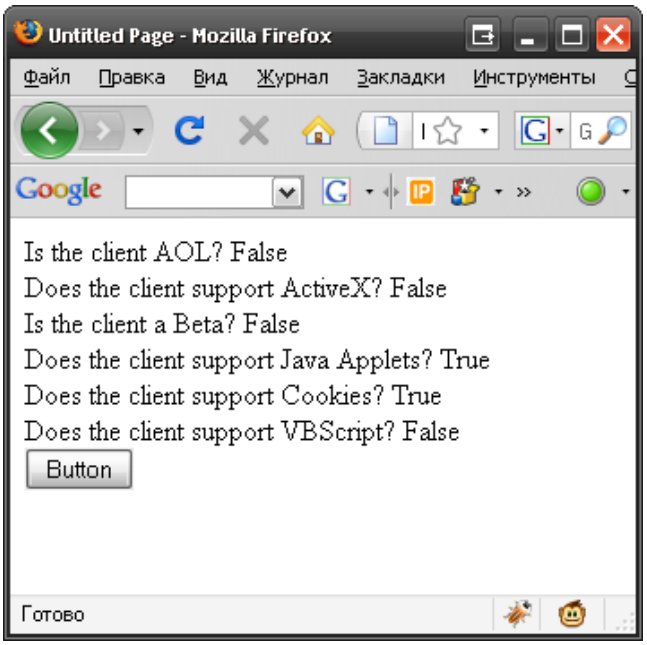


Рисунок 12.16 – Результат програми

12.Доступ до даних форми

ASP.NET дозволяє використовувати HTML форми, доступ до даних форми можна здійснити за допомогою наступного коду (лістинг 12.9).

Лістинг 12.9 – Реалізація доступу

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
// Get value for a widget with ID txtFirstName.
string firstName = Request.Form("txtFirstName");
// Use this value in your page...
}
```

Однак у більшості випадків доцільно використовувати вбудовані елементи управління ASP.NET, які дозволяють звертатися безпосередньо до властивостей (лістинг 12.10).

Лістинг 12.10 – Зверення безпосередньо до властивостей

```
protected void btnGetFormData_Click(object sender, System.EventArgs e)
{
// Get value for a widget with ID txtFirstName.
string firstName = txtFirstName.Text;
// Use this value in your page...
}
```

13.Властивість IsPostBack

Властивість типу Page IsPostBack необхідна для того, щоб визначити, чи завантажується сторінка користувачем вперше (приймає значення false), чи завантажується повторно у відповідь на певні дії користувача (значення true). Звичайно використовується у методі Page_Load() для виконання дій, які мають бути здійснені однократно при першому завантаженні сторінки за допомогою коду в лістингу 12.11.

Лістинг 12.11 – Метод Page_Load()

```
protected void Page_Load(object sender, System.EventArgs e)
{
// Fill DataSet only the very first time
```

```
// the user comes to this page.
if (!IsPostBack)
{
// Populate DataSet and cache it!
```

Продовження літингу 12.11:

```
}
// Use cached DataSet.
}
```

14.Взаємодія з результуючою HTTP-відповіддю

Тип HttpResponce призначений для зберігання відповіді, яку завантажує користувач у свій браузер. Цей тип визначає ряд властивостей, які дозволяють сформувати HTTP-відповідь, яка направляється знову браузеру клієнта. Опис базових властивостей пропонується на рисунку 12.17:

Свойство	Описание
Cache	Возвращает семантику кэширования Web-страницы (например, время ожидания, параметры конфиденциальности, различные описания)
ContentEncoding	Читает или устанавливает набор символов выходного потока HTTP
ContentType	Читает или устанавливает MIME-тип выходного потока HTTP
Cookies	Получает коллекцию HttpCookie, посланную текущим запросом
IsClientConnected	Читает значение, являющееся индикатором продолжающегося соединения клиента с сервером
Output	Разрешает пользовательский вывод в поле содержимого исходящего HTTP-сообщения
OutputStream	Разрешает двоичный вывод в поле содержимого исходящего HTTP-сообщения
StatusCode	Читает или устанавливает код состояния HTTP-ответа, возвращаемого клиенту
StatusDescription	Читает или устанавливает строку состояния HTTP-ответа, возвращаемого клиенту
SuppressContent	Читает или устанавливает значение, являющееся индикатором отмены отправки HTTP-содержимого клиенту

Рисунок 12.17 – Властивості типу HttpResponce

Можна переглянути також описи деяких методів цього типу, що представлені на рисунку 12.18.

Метод	Описание
AddCacheDependency()	Добавляет объект в кэш приложения (см. главу 24)
Clear()	Удаляет все заголовки и содержимое вывода из буфера потока
End()	Отправляет все содержимое буфера вывода клиенту, а затем завершает соединение для данного сокета
Flush()	Отправляет все содержимое буфера вывода клиенту
Redirect()	Выполняет перенаправление клиента по новому URL
Write()	Записывает значения в выходной поток HTTP-содержимого
WriteFile()	Записывает файл непосредственно в выходной поток HTTP-содержимого

Рисунок 12.18 – Методи типу HttpResponce

Приклад використання HttpResponce для програмного генерування HTML-вмісту показано в лістингу 12.12.

Лістинг 12.12 - Використання HttpResponce

```
protected void btnHttpResponse_Click(object sender, EventArgs e)
{
    Response.Write("<b>My name is:</b><br>");
    Response.Write(this.ToString());
    Response.Write("<br><br><b>Here was your last request:</b><br>");
    Response.WriteFile("MyHTMLPage.htm");
}
```

Приклад використання HttpResponce для перенаправлення користувачів (лістинг 12.13).

Лістинг 12.13 – Інший варіант використання HttpResponce

```
protected void btnSomeTraining_Click(object sender, EventArgs e)
{
    Response.Redirect("http://www.intertech.com");
}
```

15.Цикл існування Web-сторінки ASP.NET.

Сторінка ASP.NET створюється середовищем виконання у вигляді об’єкту і має свій життєвий цикл.

Події, які виникають при створенні, ініцалізації, завантаженні, функціонуванні та звільненні сторінки (рис. 12.19).

Событие	Описание
PreInit	Используется инфраструктурой .NET для размещения Web-элементов управления, применения тем, создания шаблона страницы и установки профиля пользователя. Вы можете перехватить это событие, чтобы внести изменения в соответствующий процесс
Init	Используется для установки свойств Web-элементов управления в предыдущее состояние с помощью вторичного запроса или просмотра данных состояния (подробнее об этом говорится в главе 24)
Load	Возникает тогда, когда страница и ее элементы управления полностью инициализированы, а их предыдущие значения восстановлены. С этого момента вполне безопасно начать взаимодействие с любым из Web-элементов
"Событие, вызвавшее вторичный запрос"	События с таким именем, конечно же, не существует. Так здесь обозначено любое событие, заставившее браузер отправить вторичный запрос Web-серверу (это может быть, например, щелчок на кнопке)
PreRender	Привязка данных и конфигурация пользовательского интерфейса завершена, и элементы управления готовы отправить свои данные в поток исходящего HTTP-ответа
Unload	Страница и ее элементы управления завершили процесс передачи данных, и объект страницы готов к уничтожению. Взаимодействие с исходящим HTTP-ответом в этот момент породит ошибку среды выполнения. Можно выполнить захват этого события для "уборки мусора" на уровне страницы (чтобы закрыть файлы и базы данных, выполнить процедуру выхода из системы, освободить ресурсы и т.д.)

Рисунок 12.19 – Події при використанні сторінки

Атрибут AutoEventWireUp , який задається для сторінки, призначений для того, щоб була можливість автоматичного виклику подій, пов’язаних із життєвим циклом сторінки, за замовчуванням його значення рівно “true” (лістинг 12. 14).

Лістинг 12.14 – Використання атрибуту AutoEventWireUp

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
```

Приклад коду, який буде викликатися автоматично (лістинг 12.15):

Лістинг 12.15 – Автоматичне використання коду

```
protected void Page_Load(object sender, EventArgs e)
{
    Response.Write("Load event fired!");
}
protected void Page_Unload(object sender, EventArgs e)
{
    // No longer possible to emit data to the HTTP
    // response, so we will write to a local file.
    System.IO.File.WriteAllText(@"C:\MyLog.txt", "Page unloading!");
}
```

16.Елементи управління ASP.NET

Взагалі, для розробки ASP.NET має власні елементи управління, загальний повний перелік яких демонструє рисунок 12.20.

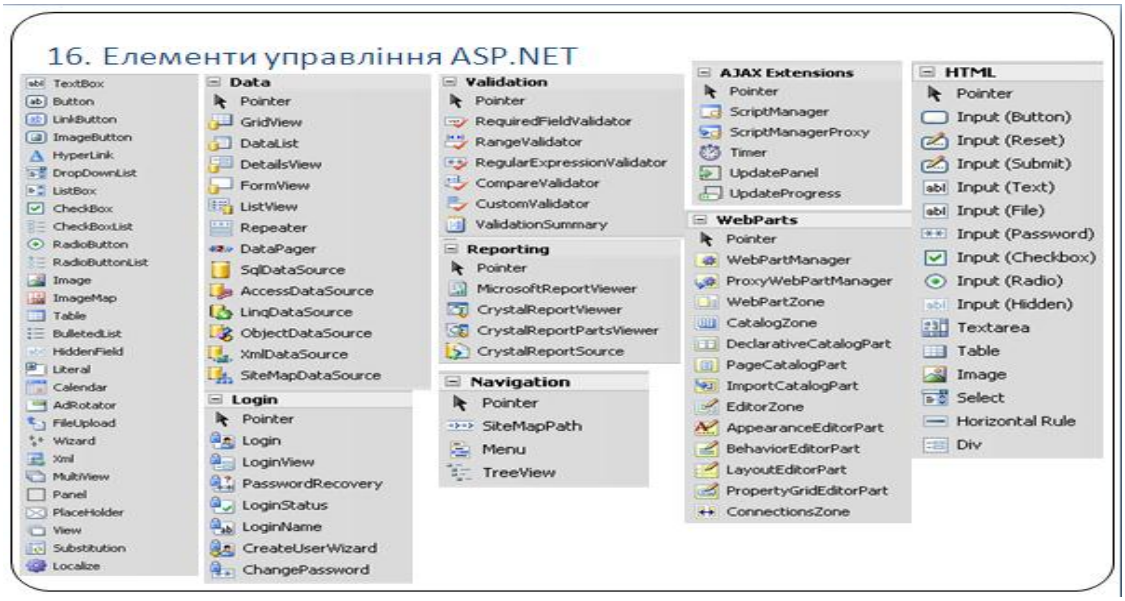


Рисунок 12.20 – Елементи управління ASP.NET

ЛК.13 – ПОГЛИБЛЕНІ ПИТАННЯ ВИКОРИСТАННЯ ASP.NET

Перелік питань

1. Властивість AutoPostBack.
2. Тип System.Web.UI.Control.
3. Тип System.Web.UI.WebControls.WebControl.
4. Категорії Web-елементів управління ASP.NET.
5. Використання шаблонів Web-сторінок.
6. Елемент управління Menu.
7. Елемент управління AdRotator.
8. Елементи управління контролю введення даних.
9. Клієнтська частина Web-рішення.
10. Елементи мови JavaScript.
11. Створення рішень на основі асинхронної передачі і обробки даних, JavaScript та XML (технологія AJAX).
12. Технології управління станом в ASP.NET.
13. Файл Global.asax.
14. Кешування даних.
15. Обробка сеансових даних.
16. Використання cookie.

На самостійне вивчення:

1. Налаштування властивостей за допомогою Web.config [1, С.1108-1115].

1. Властивість AutoPostBack

Кількість подій, які отримує серверна частина ASP.NET при виконанні дій на клієнті, обмежена лише найважливішими подіями, які передбачають відповідь на певну дію користувача.

В той же час деякі елементи управління, наприклад, TextBox, RadioButton, CheckBox підтримують властивість AutoPostBack, яка дозволяє розширити кількість подій, які вони надсилають до сервера, подіями, що виникають при зміні даних. Наприклад, наступний обробник події TextChanged для TextBox буде викликатися лише в тому разі, якщо властивість AutoPostBack встановлена в true (лістинг 13.1):

Лістинг 13.1 – Використання властивості AutoPostBack

```
protected void txtAutoPostback_TextChanged(object sender, System.EventArgs e)
{
    lstTextBoxData.Items.Add(txtAutoPostback.Text);
}
```

2. Тип System.Web.UI.Control

Тип System.Web.UI.Control є базовим для елементів управління ASP.NET. Він визначає різні властивості, методи та події, які дозволяють взаємодіяти з базовими членами Web-елементу управління (які звичайно не відносяться до графічного інтерфейсу). Основні члени типу показує рисунок 13.1.

Член	Описание
Controls	Свойство, получающее объект ControlCollection, представляющий дочерние элементы управления в рамках данного элемента управления
DataBind()	Метод, выполняющий привязку источника данных к вызванному серверному элементу управления и всем его дочерним элементам управления
EnableThemeing	Свойство, указывающее возможность поддержки тем для данного элемента управления
HasControls()	Метод для определения наличия дочерних элементов управления у данного серверного элемента управления
ID	Свойство, читающее или устанавливающее значение программного идентификатора для серверного элемента управления
Page	Свойство, получающее ссылку на экземпляр типа Page, содержащий серверный элемент управления
Parent	Свойство, получающее ссылку на родительский элемент управления данного серверного элемента управления в иерархии элементов управления страницы
SkinID	Свойство, читающее или устанавливающее параметры скиннинга элемента управления. Это дает возможность в ASP.NET 2.0 устанавливать внешний вид элемента управления динамически
Visible	Свойство, читающее или устанавливающее значение, указывающее необходимость обработки серверного элемента управления, как элемента пользовательского интерфейса страницы

Рисунок 13.1 – Набір членів System.Web.UI.Control

3. Тип System.Web.UI.WebControls.WebControl

Тип System.Web.UI.WebControls.WebControl є базовим для графічних елементів управління ASP.NET. Він забезпечує поліморфний графічний інтерфейс для всіх Web-елементів. До його основних властивостей відносяться (рис. 13.2).

Свойства	Описание
BackColor	Читает или устанавливает цвет фона Web-элемента управления
BorderColor	Читает или устанавливает цвет границы Web-элемента управления
BorderStyle	Читает или устанавливает стиль границы Web-элемента управления
BorderWidth	Читает или устанавливает ширину границы Web-элемента управления
Enabled	Читает или устанавливает значение, являющееся индикатором доступности Web-элемента управления
CssClass	Позволяет назначить Web-элементу управления класс, определенный в рамках CSS (Cascading Style Sheet — каскадная таблица стилей)
Font	Читает информацию о шрифте для Web-элемента управления
ForeColor	Читает или устанавливает цвет изображения (обычно цвет текста) для Web-элемента управления
Height Width	Читает или устанавливает высоту и ширину Web-элемента управления
TabIndex	Читает или устанавливает индекс перехода по табуляции для Web-элемента управления
ToolTip	Читает или устанавливает значение-индикатор, указывающее необходимость отображения подсказки при задержке указателя мыши на изображении Web-элемента управления

Рисунок 13.2 – Властивості базового класу WebControl

4. Категорії Web-елементів управління ASP.NET

Серед Web-елементів управління ASP.NET можна виділити наступні категорії:

- прості елементи управління (мають прямих відповідників серед стандартних HTML-тегів);
- елементи управління з розширеними можливостями (не мають прямих відповідників серед стандартних HTML-тегів, проте схожі до певних елементів управління Windows Forms);
- елементи управління для роботи із джерелами даних (призначені для роботи із джерелами даних);
- елементи управління для контролю введення даних (є серверними елементами управління, які генерують JavaScript-код клієнта для перевірки даних, що вводяться до форми);
- елементи управління для входу до системи (спеціальний набір елементів управління для забезпечення входу на сайт, отримання пароля, підтримки ролей користувачів).

5. Використання шаблонів Web-сторінок

Технологія ASP.NET підтримує можливість створення шаблонів сторінок, які фактично являють собою звичайні сторінки, однак зберігаються у файлах з розширенням .master, від яких наслідуються конкретні .aspx-сторінки. За допомогою шаблонів Web-сторінок можна створювати сайти, яка загальну систему меню, загальні елементи оформлення верхної, нижньої та бокових сторін сторінки та ін. (рис. 13.3).

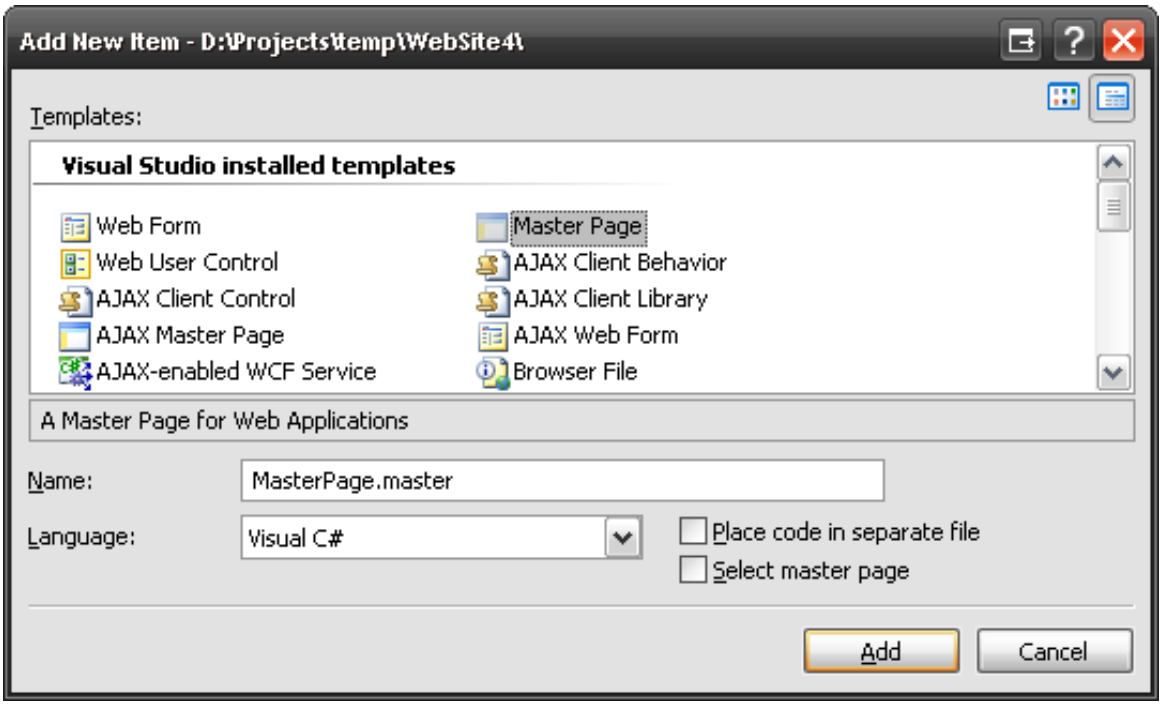


Рисунок 13.3 – Створення та вибір шаблону

6. Елемент управління Menu

Елемент управління Menu дозволяє створювати меню у ASP.NET-сайтах подібно до меню у Windows-програмах. При налаштуванні даного елементу управління слід задати сторінки, на які буде здійснюватися перехід при виборі відповідних пунктів меню. Він дозволяє переходити до інших сторінок вузла. Виконати додаткові дії у випадку вибору користувачем даного пункту меню можна за допомогою обробки події MenuItemClick. Вигляд цього елементу управління показано на рисунку 13.4.

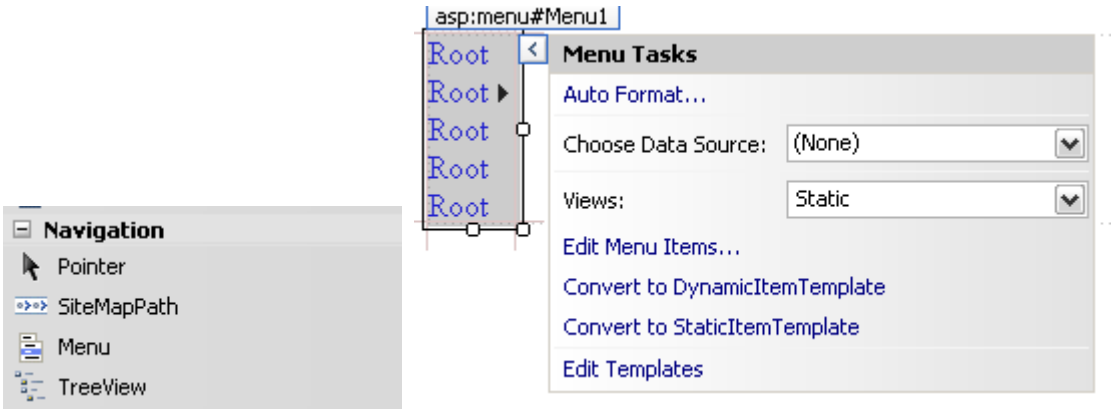


Рисунок 13.4 – Вигляд елементу управління Menu

7. Елемент управління AdRotator

Елемент управління AdRotator дозволяє у випадковій послідовності відображати зображення у певному місці сторінки. Перелік файлів зображень, які слід показувати, задається за допомогою властивості AdvertisementFile, що вказує на XML-файл певної структури (допускається використання інших джерел даних).

Приклад XML-файлу із зображеннями показано в лістингу 13.2.

Лістинг 13.2 – Використання елементу AdRotator

```
<Advertisements>
  <Ad>
    <ImageUrl>SlugBug.jpg</ImageUrl>
    <TargetUrl>http://www.Cars.com</TargetUrl>
    <AlternateText>Your new Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
  <Ad>
    <ImageUrl>car.gif</ImageUrl>
    <TargetUrl>http://www.CarSuperSite.com</TargetUrl>
    <AlternateText>Like this Car?</AlternateText>
    <Impressions>80</Impressions>
  </Ad>
</Advertisements>
```

8. Елементи управління контролю введення даних

Елементи управління контролю введення даних (група Validation) призначені для перевірки правильності даних, які вводяться користувачем. На рисунку 13.5 показано опис елементів управління ASP.NET , пов’язаних з контролем введення.

Элемент управления	Описание
CompareValidator	Выполняет проверку значения одного элемента управления на равенство фиксированной константе или значению другого элемента управления
CustomValidator	Позволяет построить функцию пользовательского контроля ввода для данного элемента управления
RangeValidator	Проверяет принадлежность значения заданному диапазону значений
RegularExpressionValidator	Проверяет соответствие значения соответствующего элемента управления заданному шаблону регулярного выражения
RequiredFieldValidator	Гарантирует, что данный элемент управления не останется пустым (т.е. будет содержать значение)
ValidationSummary	Отображает резюме всех ошибок проверки ввода на странице в формате списка, списка с буллитами или формате единого абзаца. Ошибки могут отображаться "на месте" и/или во всплывающем окне сообщения

Рисунок 13.5 – Елементи контролю введення даних

Також дані елементи мають власні властивості, що безпосередньо дозволяють управляти даними та їх перевіркою на коректність при введенні. Їх перелік та опис наводиться на рисунку 13.6.

Свойство	Описание
ControlToValidate	Читает или устанавливает имя элемента управления, который необходимо контролировать
Display	Читает или устанавливает значение, характеризующее вид отображения сообщения об ошибке для элемента контроля ввода
EnableClientScript	Читает или устанавливает признак активизации контроля ввода на стороне клиента
ErrorMessage	Читает или устанавливает текст сообщения об ошибке
ForeColor	Читает или устанавливает цвет сообщения, отображаемого при отрицательном исходе проверки ввода

Рисунок 13.6 - Важливі властивості елементів контролю введення

9. Клієнтська частина Web-рішення

Сучасні Web-рішення передбачають виконання певної частини обчислень на стороні клієнта, що реалізується з використанням мови програмування JavaScript.

У такому разі браузер виступає у якості не просто засобу, який надсилає запити до сервера та відображає HTML, а у якості справжньої платформи, яку необхідно програмувати подібно до того, як створюються програми для будь-якої іншої платформи.

Реалізація JavaScript для кожного браузеру має свої особливості, що суттєво ускладнює створення подібних рішень, які мають працювати як на звичайних десктопних браузерах, так і на браузерах мобільних пристроїв.

10.Елементи мови JavaScript

Код на JavaScript помічається у блоки <script> із атрибутом “JavaScript”, виконання цього коду для стандартного HTML елемента управління button здійснюється за допомогою атрибута “onclick” (лістинг 13.3):

Лістинг 13.3 – Використання мови JavaScript

```
<html>
  <body>
    <form>
      <script language="JavaScript">
        function displayPopup() {
            alert("Привіт, Світ!");
        }
      </script>

      <input type="button" value="Натисніть мене!" onclick="displayPopup()"
/>
    </form>
  </body>
</html>
```

Також мову JavaScript можна використовувати і при прив’язці до елементів управління ASP.NET. Наприклад, розглянемо використання кнопки. Для виконання JavaScript при натисканні ASP.NET-кнопки необхідно код на JavaScript додати в атрибут кнопки “onload” під час виконання завантаження сторінки (метод PageLoad). Код даного використання показано в лістингу 13.4.

Лістинг 13.4 - Приклад виконання JavaScript при натисканні кнопки

```
<%@ Page Language="C#" %>
<script runat="server">
   void Page_Load(object sender, EventArgs e) {
      HelloButton.Attributes.Add("onclick",
           "javascript:alert('Привіт, світ!')");
   }
</script>
<html>
<head>
</head>
<body>
    <form id="MainForm" runat="server">
        <asp:Button id="HelloButton" runat="server" Font-Bold="True"
         Font-Names="Verdana" Font-Size="Larger"
         Text="Click Me!"></asp:Button>
    </form>
</body>
</html>
```

11.Створення рішень на основі асинхронної передачі і обробки даних, JavaScript та XML (технологія AJAX)

ASP.NET підтримує можливість створення рішень на основі технології AJAX (Asynchronous JavaScript and XML), зберігаючи при цьому максимальну простоту і наглядність розробки.

Підтримка AJAX реалізується за допомогою спеціальної групи елементів управління із вкладки “AJAX Extensions” (рис. 13.7), які працюють сумісно із звичайними елементами управління, змінюючи їх поведінку таким чином, щоб отримати переваги асинхронної передачі і обробки даних.

Рисунок 13.7 – Елементи управління AJAX Extensions

Одним із найпоширеніших зручних інструментів AJAX Extensions є UpdatePanel. Він використовується для оновлення лише частини браузера саме в цій панелі без оновлення всього браузера. Створимо веб-сторінку, яка завантажується в браузер у з певною періодичністю (10 сек.) показує курс акцій у браузері, не оновлюючи при цьому сторінку повністю (лістинг 13.5).

Лістинг 13.5 - Приклад використання UpdatePanel

```
<%@ Page Language="C#" AutoEventWireup="true" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.1//EN" °
"http://www.w3.org/TR/xhtml11/DTD/xhtml11.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head id="Head1" runat="server">
    <title>Timer Example Page</title>
    <script runat="server">
        protected void Page_Load(object sender, EventArgs e)
        {
            OriginalTime.Text = DateTime.Now.ToLongTimeString();
        }
        protected void Timer1_Tick(object sender, EventArgs e)
        {
            StockPrice.Text = GetStockPrice();
            TimeOfPrice.Text = DateTime.Now.ToLongTimeString();
        }
        private string GetStockPrice()
        {
            double randomStockPrice = 50 + new Random().NextDouble();
            return randomStockPrice.ToString("C");
        }
    </script>
</head>
<body>
    <form id="form1" runat="server">
        <div>
        Приклад використання ASP.NET AJAX:
        </div>
        <div>
        Сторінка завантажена: <asp:Label ID="OriginalTime"
runat="server"></asp:Label>
        </div>
        <asp:ScriptManager ID="ScriptManager1" runat="server" />
        <asp:Timer ID="Timer1" OnTick="Timer1_Tick" runat="server"
Interval="10000" />
        <asp:UpdatePanel ID="StockPricePanel" runat="server"
UpdateMode="Conditional">
        <Triggers>
            <asp:AsyncPostBackTrigger ControlID="Timer1" />
        </Triggers>
        <ContentTemplate>
            Курс акцій: <asp:Label id="StockPrice"
runat="server"></asp:Label><BR />
            станом на <asp:Label id="TimeOfPrice" runat="server"></asp:Label>
        </ContentTemplate>
        </asp:UpdatePanel>
    </form>
</body>
```

```
</html>
```

12.Технології управління станом в ASP.NET

Протокол HTTP не підтримує управління станом, тому його необхідно реалізовувати окремо. Технологія ASP.NET надає цілий ряд механізмів для управління станом:

- використання даних стану представлень asp;net;
- використання даних стану елементів управління asp;net;
- задання змінних рівня додатку;
- використання об’єктів кешування;
- визначення змінних рівня сеансу;
- використання даних cookie.

Стан представлень ASP.NET можна пояснити наступним чином. Атрибут EnableViewState директиви <%@Page%> відповідає за активацію внутрішнього механізму управління станом представлення і за замовчуванням має значення true. Стан сторінки зберігається у службовому прихованому полі _VIEWSTATE. Цей механізм дозволяє зберігати вміст і стан елементів управління навіть у тому разі, якщо сторінку було перезавантажено. Кожен елемент управління також має властивість EnableViewState, використовуючи яку можна управляти збереженням стану на рівні сторінки.

Можна продемонструвати наступний приклад використання ViewState для збереження даних (лістинг 13.6, 13.7):

Лістинг 13.6 – Використання ViewState (код Default.aspx.cs)

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
  Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
  "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml" >
<body>
    <form id="form1" runat="server">
        <asp:TextBox runat="server" id="NameField" />
        <asp:Button runat="server" id="SubmitForm" onclick="SubmitForm_Click"
            text="Надіслати і встановити змінну" />
```

```
        <asp:Button runat="server" id="RefreshPage" text="Зчитати значення
змінної"
            onclick="RefreshPage_Click" />
        <br /><br />
```

Продовження лістингу 13.6:

```
   Змінна, отримана з ViewState: <asp:Label runat="server" id="NameLabel" />
    </form>
</body>
</html>
```

Лістинг 13.7 - Використання ViewState (код Default.aspx.cs)

```
using System;
using System.Data;
using System.Web;
public partial class _Default : System.Web.UI.Page
{
    const string SomeVar = "SomeVar";
    protected void Page_Load(object sender, EventArgs e)
    {
        if (ViewState[SomeVar] != null)
            NameLabel.Text = ViewState[SomeVar].ToString();
        else
            NameLabel.Text = "Поки що не задано...";
    }
    protected void SubmitForm_Click(object sender, EventArgs e)
    {
        ViewState[SomeVar] = NameField.Text;
    }
    protected void RefreshPage_Click(object sender, EventArgs e)
    {
        if (ViewState[SomeVar] != null)
            NameLabel.Text = ViewState[SomeVar].ToString();
    }
}
```

13.Файл Global.asax

Файл Global.asax є необов’язковим для проекту і дозволяє задати обробники, які можуть виконуватися на рівні всього проекту (рис. 13.8).

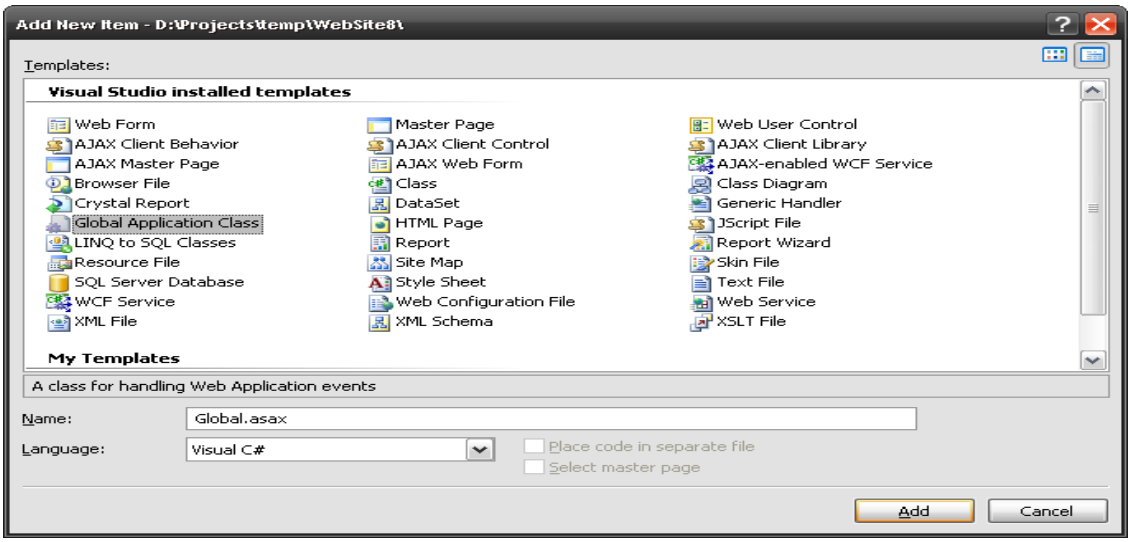


Рисунок 13.8 – Створення файлу Global.asax

Вміст файлу Global.asax після створення можна переглянути за допомогою лістингу 13.8:

Лістинг 13.8 - Файлу Global.asax

```
<%@ Application Language="C#" %>
<script runat="server">
    void Application_Start(object sender, EventArgs e)
    {
        // Code that runs on application startup
    }

    void Application_End(object sender, EventArgs e)
    {
        //  Code that runs on application shutdown
    }

    void Application_Error(object sender, EventArgs e)
    {
        // Code that runs when an unhandled error occurs
    }
    void Session_Start(object sender, EventArgs e)
    {
        // Code that runs when a new session is started
    }
    void Session_End(object sender, EventArgs e)
    {
        // Code that runs when a session ends.
        // Note: The Session_End event is raised only when the sessionstate
mode
        // is set to InProc in the Web.config file. If session mode is set to
StateServer
        // or SQLServer, the event is not raised.
    }

</script>
```

Файл Global.asax є останнім місцем, де можна спіймати необроблену виключну ситуацію до того, як вона призведе до падіння роботи веб-рішення. Приклад обробки виключної ситуації показано на рисунку 13.9:

Лістинг 13.9 - Обробка помилок в Global.asax

```
void Application_Error(object sender, EventArgs e)
{
    // Code that runs when an unhandled error occurs
    Exception ex = Server.GetLastError();
    EventLog ev = new EventLog("Application");
    ev.WriteEntry(ex.Message, EventLogEntryType.Error);
    Server.ClearError();
    Response.Write("Виникла помилка. Вибачте!");
}
```

14.Кешування даних

Необхідність у кешуванні даних є дуже актуальною для веб-рішень. Наприклад, якщо створюється програма, яка зчитує певну інформацію з БД (скажімо, курс валют), то при високій відвідуваності сайту навантаження на БД буде надзвичайно великим.

Для вирішення цієї проблеми доцільно створити кеш, у який дані слід зчитати один раз при першому зверненні, а потім усім новим клієнтам показувати збережені у кеші дані до тих пір, доки вони не втратять актуальності, після цього кеш слід оновити. Технологія ASP.NET реалізує гнучкий механізм кешування даних, у такому разі розробнику не потрібно створювати власну реалізацію.

Записати значення до кешу можна як найпростішим способом, звернувшись до індексатору з назвою значення, так і повним, використавши для цього спеціальний метод Add(). У другому разі можна зазначити час зберігання, залежність елементу від інших об’єктів, пріоритет елементу, спосіб обчислення часу для його видалення із кешу та делегат, який буде викликаний у разі видалення елементу із кешу. Приклад використання кешу міститься в лістингу 13.10:

Лістинг 13.10 - Приклад використання кешу у C#

```
// Простий спосіб зберегти значення до кешу.Це значення не застаріє.
Context.Cache["SomeStringItem"] = "This is the string item";

// Зчитати значення з кешу.
string s = (string)Context.Cache["SomeStringItem"];

// Повний спосіб збереження даних до кешу
Context.Cache.Add("AppDataTable", // Назва елементу в кеші
    theData, // Об'єкт, який слід помістити до кешу
    null, // Залежності для цього об'єкту
    DateTime.Now.AddSeconds(15), // Як довго об'єкт буде в кеші
    Cache.NoSlidingExpiration, // Фіксований чи плаваючий час
    CacheItemPriority.Default, // Пріоритет для елементу
    // Делегат, який буде викликаний у випадку видалення елементу із кешу
    new CacheItemRemovedCallback(UpdateCarInventory));
```

15.Обробка сеансових даних

Типова задача, яка виникає при створенні веб-рішень – це необхідність зберігати дані на рівні сеансу, який встановлює користувач, що звернувся до сайту. Наприклад, це може бути реалізація кошику замовлень для інтернет-магазину – у кожного із клієнтів має бути своя власна корзина. Сеанс у ASP.NET реалізується за допомогою типу HttpSessionState – об’єкт цього типу створюється для кожного відвідувача сайту. Приклад звернення до сеансових даних (лістинг 13.11):

Лістинг 13.11 - Обробка сеансових даних

```
// Додання сеансової змінної для певного користувача
        Session["ChoosenCarModel"] = "BMW740i";

        // Зчитування сеансової змінної для певного користувача
        string carModel = (string)Session["ChoosenCarModel"];
```

Для можливості зберігання об’єктів на сеансовому рівні необхідно передбачити можливість збереження/зчитування їх з рядку.

16.Використання cookie

Файли cookie – це текстові файли, які веб-рішення може зберегти на комп’ютері користувача. Використовуються для ідентифікації користувача

(для того, щоб не вводити пароль кожен раз при відвідуванні сайту), збереження певних неконфіденційних даних (наприклад, зовнішній вигляд чи розташування елементів сайту).

Файли cookie у ASP.NET можуть бути:

– постійними – фізично зберігаються на жорсткому диску користувача;
– тимчасовими – фактично зберігаються у оперативній пам'яті, при закритті браузера зникають.

Навіть постійні cookie мають певний термін актуальності – він задається у вигляді дати, до якої їх можна використовувати. Рисунок 13.9 демонструє використання cookie.

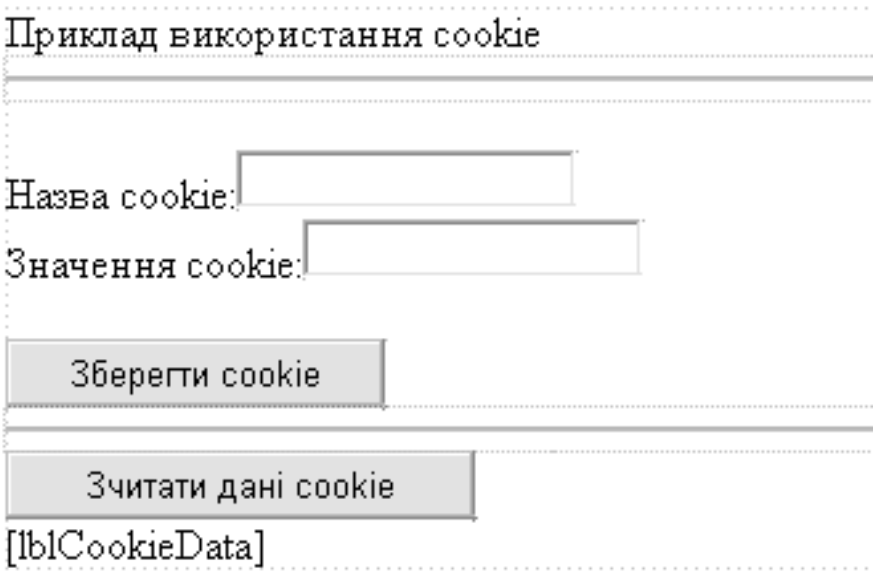


Рисунок 13.9 - Приклад використання cookie (сторінка)

Відповідно, до цієї сторінки у мові програмування розроблено код. Він наведений у лістингах 13.12 та 13.13:

Лістинг 13.12 - Приклад використання cookie (код сторінки)

```
<%@ Page Language="C#" AutoEventWireup="true"  CodeFile="Default.aspx.cs"
Inherits="_Default" %>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head runat="server">
```

```
    <title>Untitled Page</title>
</head>
<body>
    <form id="form1" runat="server">
    <div>

        <asp:Label ID="lblTitle" runat="server" Text="Приклад використання
cookie"></asp:Label>
        <hr />
        <br />
        <asp:Label ID="lblCookieName" runat="server" Text="Назва
cookie:"></asp:Label>
        <asp:TextBox ID="txtCookieName" runat="server"></asp:TextBox>
        <br />
        <asp:Label ID="lblCookieVal" runat="server" Text="Значення
cookie:"></asp:Label>
        <asp:TextBox ID="txtCookieValue" runat="server"></asp:TextBox>
        <br />
        <br />
        <asp:Button ID="btnWriteCookie" runat="server" Text="Зберегти cookie"
            onclick="btnWriteCookie_Click" />
        <hr />
        <asp:Button ID="btnReadCookie" runat="server"
onclick="btnReadCookie_Click"
```

Продовження лістингу 13.12:

```
 Text="Зчитати дані cookie" />
        <br />
        <asp:Label ID="lblCookieData" runat="server"></asp:Label>
        <br />

    </div>
    </form>
</body>
</html>
```

Лістинг 12.13 - Приклад використання cookie (код code-behind)

```
using System;
using System.Configuration;
using System.Data;
using System.Linq;
using System.Web;
using System.Web.Security;
using System.Web.UI;
using System.Web.UI.HtmlControls;
using System.Web.UI.WebControls;
using System.Web.UI.WebControls.WebParts;
using System.Xml.Linq;
public partial class _Default : System.Web.UI.Page
{
    protected void btnWriteCookie_Click(object sender, EventArgs e)
    {
        HttpCookie theCookie =
            new HttpCookie(txtCookieName.Text, txtCookieValue.Text);
        theCookie.Expires = DateTime.Now.AddYears(1);
        Response.Cookies.Add(theCookie);
    }
    protected void btnReadCookie_Click(object sender, EventArgs e)
    {
        string cookieData = "";
        foreach (string s in Request.Cookies)
```

```
        {
            cookieData +=
                string.Format("<li><b>Назва<b>: {0}, <b>Значення<b>:
{1}</li>",
                  s, Request.Cookies[s].Value);
        }
        lblCookieData.Text = cookieData;
    }
}
```

Результат виконання програми у браузері наведено на рисунку 13.10. Завантажується сторінка, яка дозволяє зберігати певні значення та повертати їх на екран.

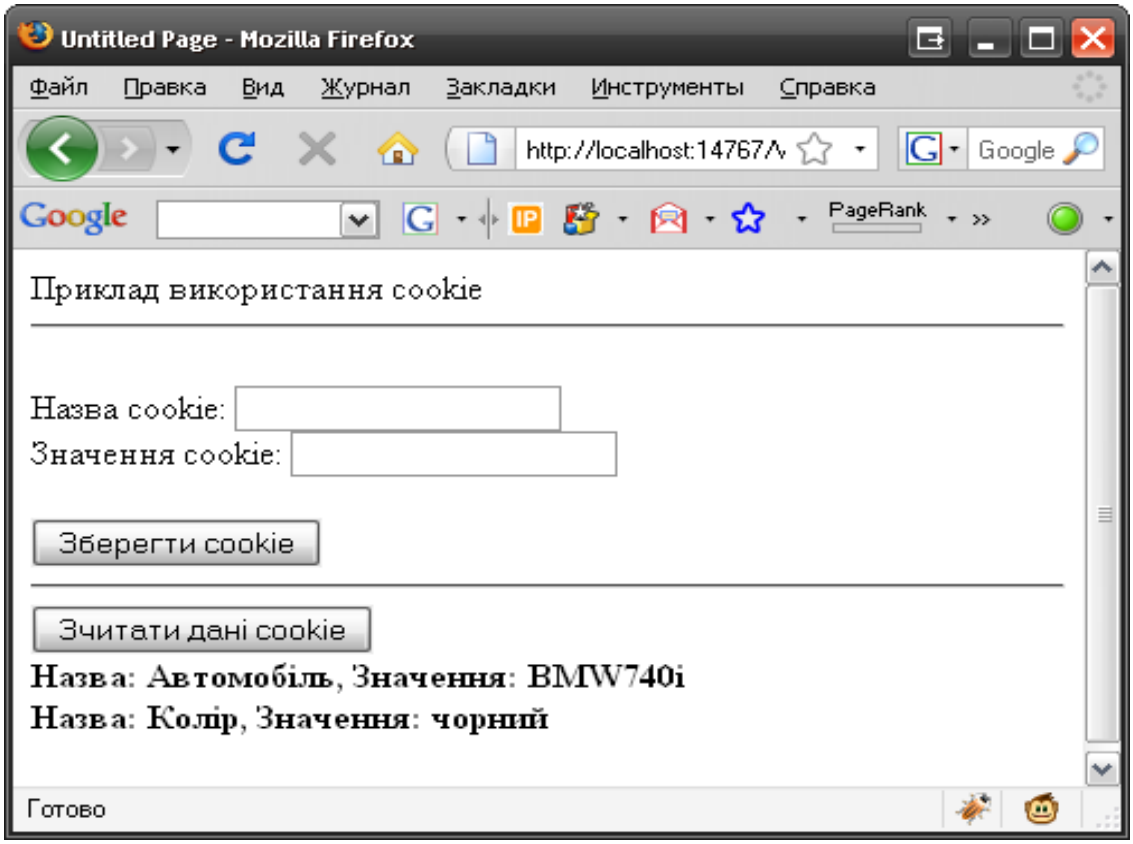


Рисунок 13.10 – Результат програми

ЛК.14 – ПРОГРАМУВАННЯ МЕРЕЖ НА ОСНОВІ TCP/IP

Перелік питань

1. Концепція мереж і протоколів.
2. Основи програмування мереж в .NET Framework.
3. Простір імен System.NET.
4. Робота з сокетами.
5. Програмування протоколу TCP.
6. Створення програми для обміну миттєвими повідомленнями.

На самостійне вивчення:

1. Створення розподіленого рішення [1, С.714-718].

1. Концепція мереж і протоколів

Мережа – це комп’ютер (Компанія SUN). Для того, щоб різні комп’ютери різних виробників могли взаємодіяти між собою були створені протоколи – домовленості, які регламентують, яким чином можна обмінюватися інформацією.

Для представлення стандартів і протоколів, які приймають участь у роботи комп’ютерної мережі використовуються модель OSI (Open System Interconnection), яка була створена Міжнародною організацією зі стандартизації у 1980-х роках. ЇЇ табличний вигляд з описами всіх семи рівнів представлений в таблиці 14.1. В основі знаходиться фізична частина, що відповідає за зв'язок із фізичним середовищем. Канальний передає дані між сусідніми вузлами, мережний відповідає за вибір маршруту та доставку пакетів даних, транспортний встановлює сполучення. Інші три рівня складають сеансовий (керування діалогом при з’єднанні), відображення (перетворення даних) та прикладний (взаємодія прикладних процесів).

Таблиця 14.1 – Структура моделі OSI

Номер та назва рівня за стандартом	Назва рівня українською мовою	Назва рівня у російських джерелах	Функціональне призначення рівня
7 Application layer	Прикладний	Прикладной	Взаємодія прикладних процесів
6 Presentation layer	Відображення	Представи-тельный	Перетворення даних (кодів, форматів)
5 Session layer	Сеансовий	Сеансовый	Керування діалогом
4 Transport layer	Транспортний	Транспортный	Встановлення наскрізного прозорого сполучення
3 Network layer	Мережний	Сетевой	Вибір маршруту та доставка пакетів
2 Data Link layer	Канальний	Канальный	Передавання даних між сусідніми вузлами
1 Physical layer	Фізичний	Физический	Зв’язування з фізичним середовищем

Яскраво демонструє мережне під’єднання комп’ютерів, їх взаємодію та всі необхідні пристрої рисунок 14.1. Для цього необхідна спеціальна розетка мережного виходу, певні з’єднувальні кабелі, конвектори та концентратори.

Для підєднання використовується спеціальний кабель. Він має назву «вита пара» та комплексну структуру, яка зображена на рисунку 14.2. Для його виготовлення використовується мідний провід і передбачається створення захисної оболонки і металевого екрану.

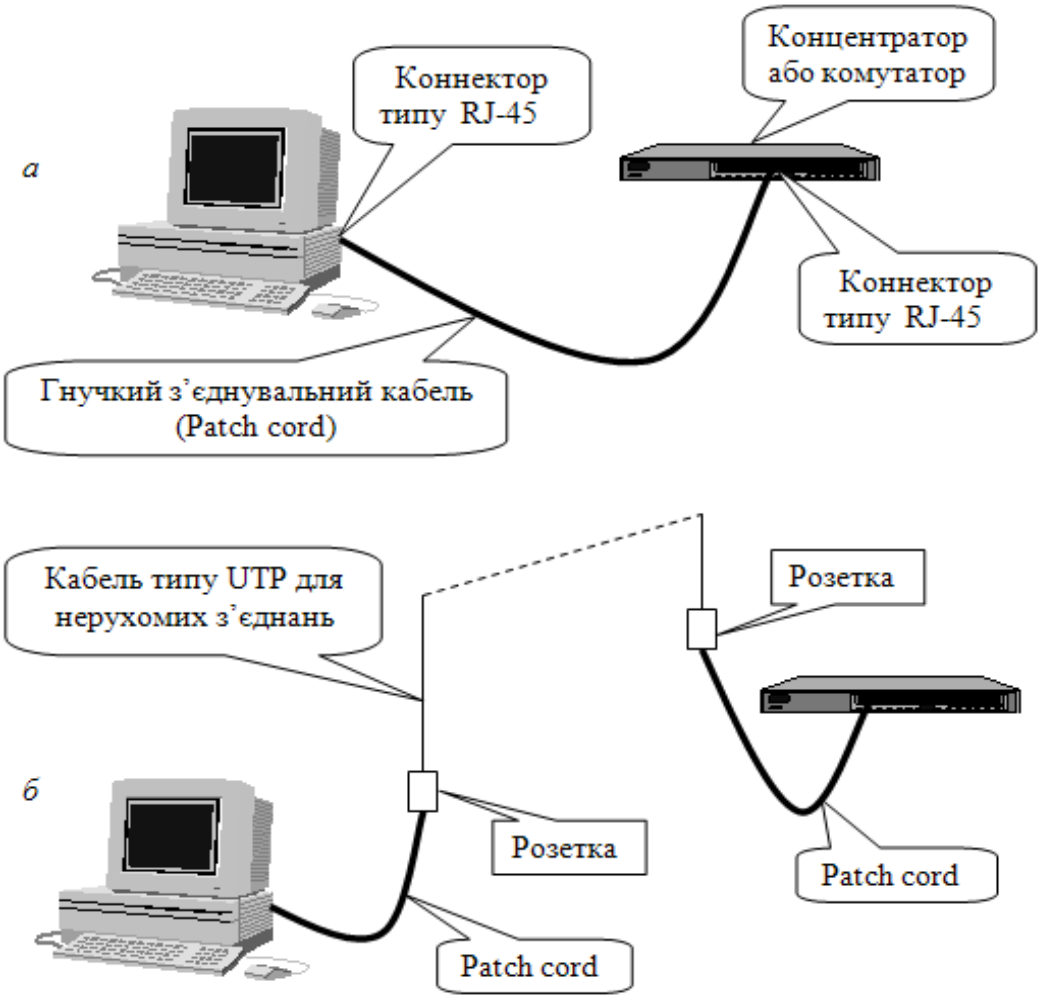


Рисунок 14.1 - Під'єднання комп'ютерів до мережі

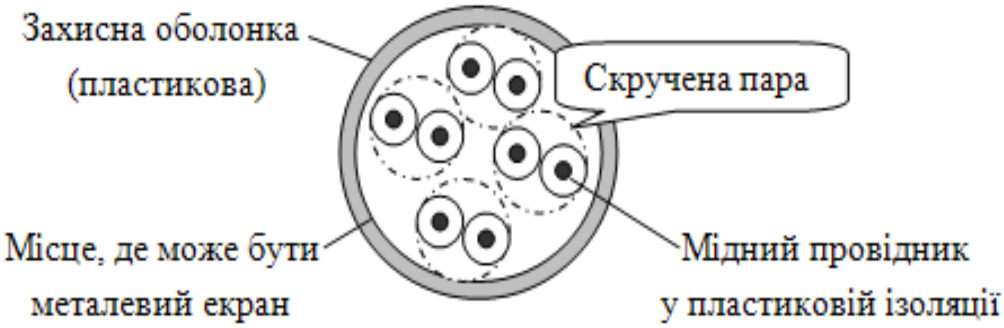


Рисунок 14.2 – Структура мережного кабелю

Конкретна реалізація моделі OSI знаходить своє відображення у конкретному стеку протоколів. В сучасних умовах найбільшого поширення

набув стек протоколів TCP/IP, на основі якого побудована глобальна мережа Internet та більшість локальних мереж у світі (рис. 14.3):

Рисунок 14.3 – Вигляд стеку протоколів

Стек протоколів TCP/IP був розроблений у 1969 році. Він містить лише 4 рівні і відрізняється від моделі OSI. Його рівні укрупнені і реалізують більшу функціональність порівняно з рівнями моделі (мова йде про прикладний та канальний рівні). Детально побачити різницю можна на рисунку 14.4. Прикладний рівень стеку об'єднав у собі останні 3 рівні в ієрархії моделі. Канальний об'єднав функціональність канального та фізичного рівнів моделі. В інших випадках зміни не відбулося. Назви протоколів, які відповідають кожному рівню стеку, показано на рисунку 14.5.

Рівні моделі OSI	Рівні стеку TCP/IP
Прикладний	Прикладний
Відображення	
Сеансовий	
Транспортний	Транспортний
Мережний	Міжмережний
Канальний	Мережного інтерфейсу (Канальний)
Фізичний	

Рисунок 14.4 - Стек протоколів TCP/IP

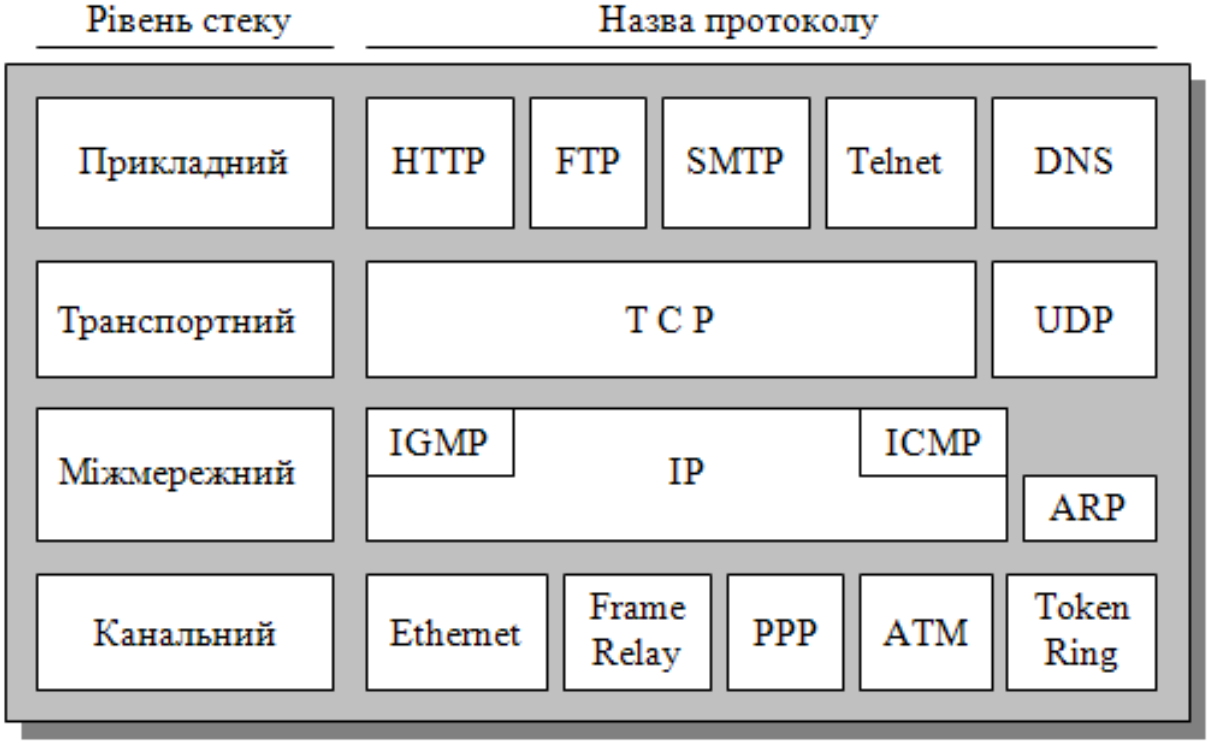


Рисунок 14.5 - Протоколи стеку TCP/IP

Для під'єднання комп'ютерів до мережі використовується IP-адресація. Властивості під'єднання, де можна подивитися адресу кожного комп'ютера, та структуру мережі показано на рисунках 14.6, 14.7. IP-адреси можуть відноситися до різних класів. Це залежить від адреси мережі чи вузла і демонструється на рисунку 14.8.

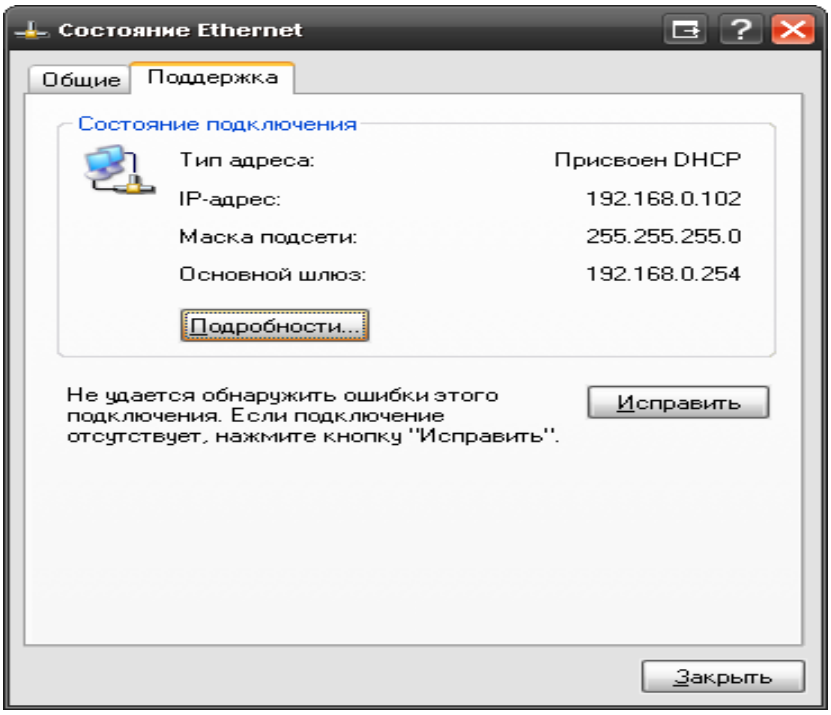


Рисунок 14.6 – Приклад властивості під’єднання

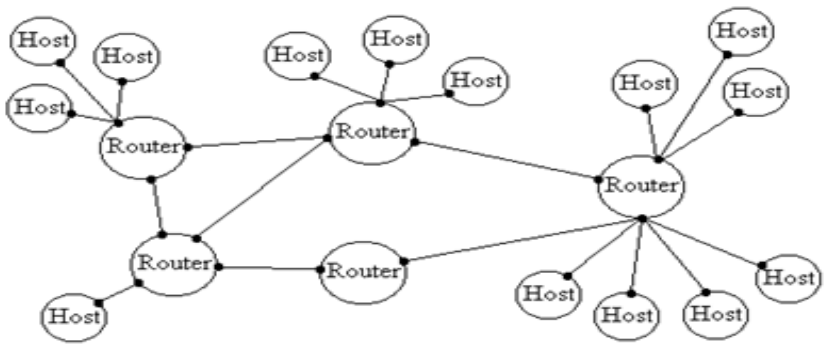


Рисунок 14.7 - Структура мережі

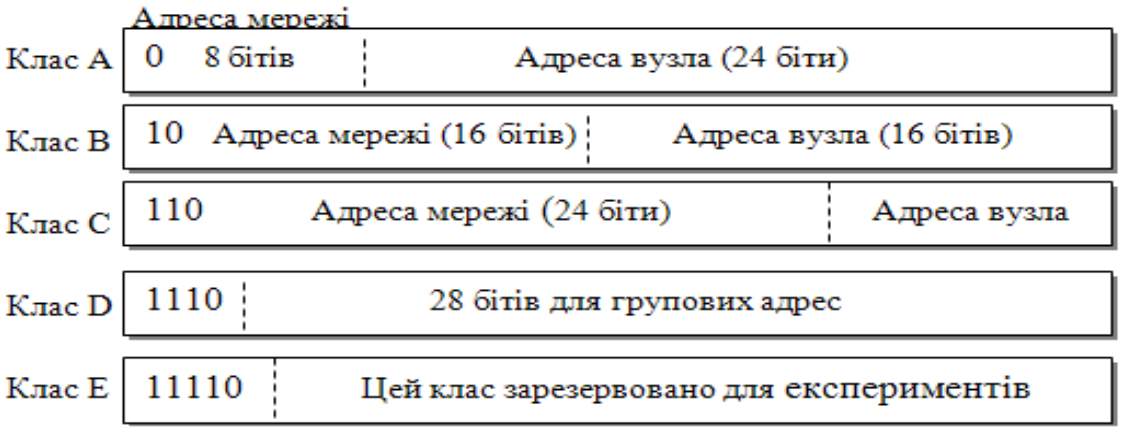


Рисунок 14.8 - Класи IP-адрес

Перші 3 класи з власними IP-адресами відносять до внутрішніх мереж. Рисунок 14.9 демонструє їх, а також діапазон адрес і їх кількість:

IP-адреси, що зарезервовані ICANN, для внутрішніх мереж

Клас	Діапазон IP-адрес	Кількість адрес
A	10.0.0.0 – 10.255.255.255	16 772 216
B	172.16.0.0 – 172.31.255.255	1 048 576
C	192.168.0.0 – 192.168.255.255	65 536

Рисунок 14.9 - Внутрішні мережі

Шлюз та клієнти повинні мати у мережі власні IP-адреси для їх розпізнавання, безпосередньої взаємодії та звернення. Клієнти підключені до мережі через комутатор, який у свою чергу зв’язаний із шлюзом. Наочний приклад простої мережі з адресами показано на рисунку 14.10:

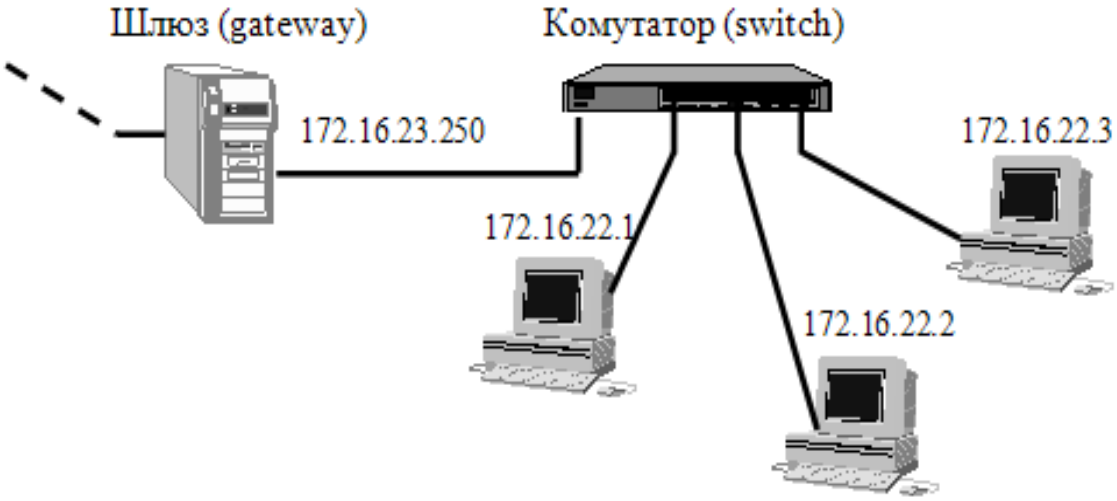


Рисунок 14.10 - Приклад маршрутизації

У світовій мережі Internet використовуються доменні імена для взаємодії та посилання на певний елемент мережі. Структуру простору доменних імен можна представити так (рис. 14.11):

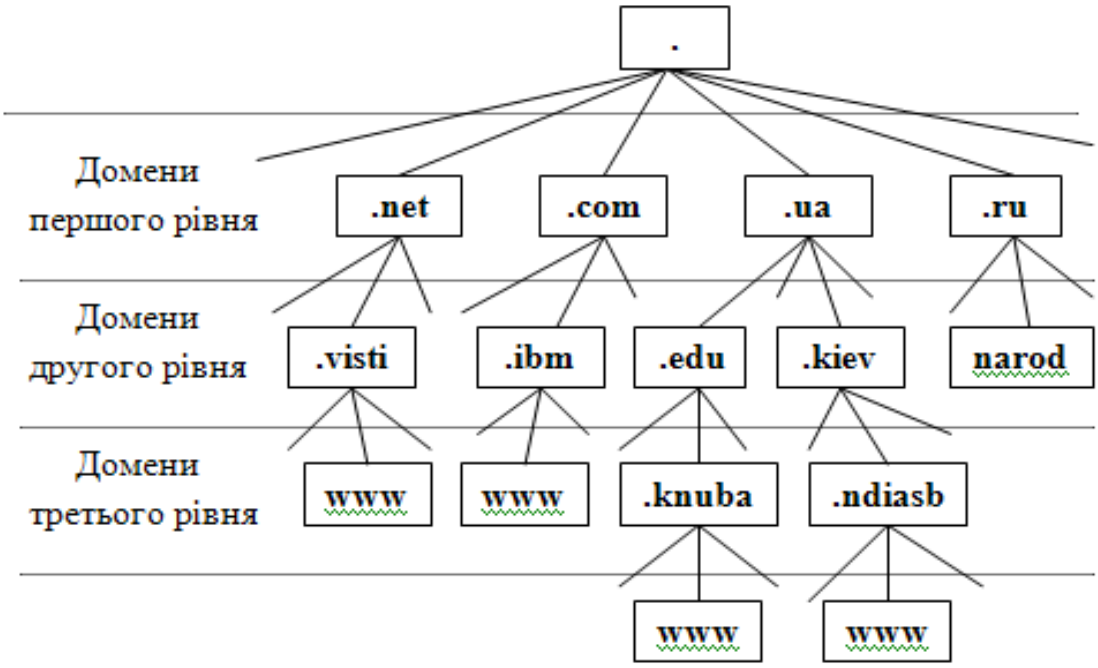


Рисунок 14.11 - DNS (доменні імена)

Сокетом називається конкретний порт і адреса протоколу (рис. 14.12):

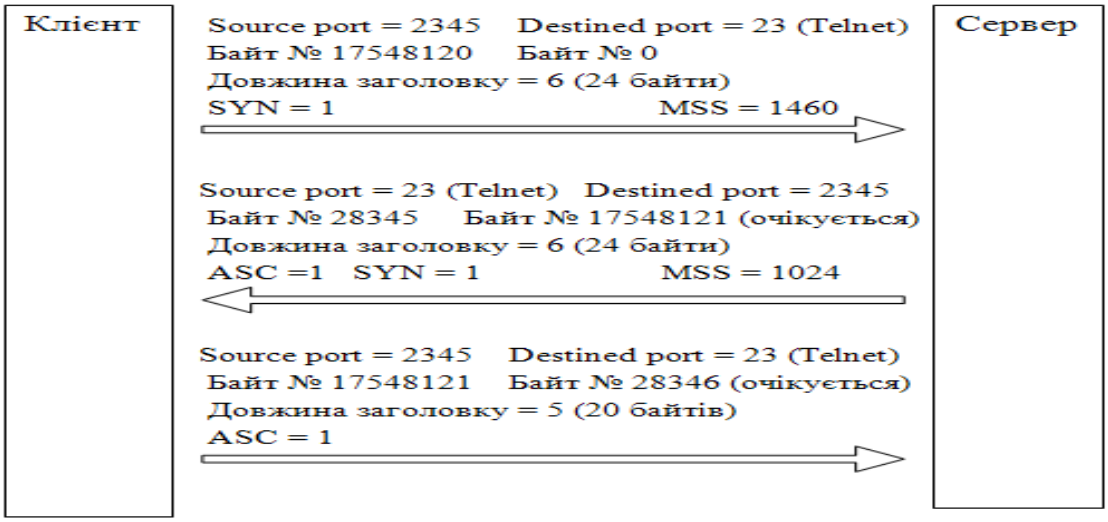


Рисунок 14.12 - Приклад комунікацій у протоколі TCP

2. Основи програмування мереж в .NET Framework

У .NET Framework існує декілька механізмів програмування мереж, які дають різний рівень контролю і мають різні сфери застосування:

– програмування сокетів – відносно низькорівневе програмування TCP/IP;
– технологія .NET Remoting – дозволяє звертатися до об’єктів у просторі імен іншої програми, навіть якщо вона функціонує на іншому комп’ютері;
– Web-сервіси – розробка служб, які використовують HTTP та XML для обміну інформацією;
– технологія WCF (Windows Communication Foundation) – найбільш сучасний і гнучкий підхід для високорівневого програмування комунікацій на основі технологій Web-сервісів у .NET Framework.

3. Простір імен System.NET

Простір імен System.NET реалізує програмні інтерфейси для програмування мереж. Для прикладу, складовою даного простору може бути System.Net.Sockets, що відповідає за програмування сокетів.

4. Робота з сокетами

Сокет – це комбінація IP-адреси та порту. Він фактично є точкою входу, за допомогою якої можуть зв’язуватися між собою програми на основі протоколу TCP/IP. У .NET Framework для програмування сокетів використовується простір імен System.Net.Sockets. Клас Socket надає гнучкі можливості для програмування сокетів, зокрема, дозволяє здійснювати комунікації як у синхронному, так і у асинхронному режимах.

Розглянемо програму взаємодії з сокетами у консольному режимі (лістинги 14.1, 14.2, 14.3).

Лістинг 14.1 - Структура програми для завантаження HTML-документів у консольному режимі:

```
using System;
using System.IO;
using System.Net;
using System.Text;
using System.Net.Sockets;
namespace SocketProgramming
{
    class Program
    {
        [STAThread]
        static void Main(string[] args)
        {
            // Тіло метода
        }
        public static void MakeRequest(Socket client, string path, string
server)
        {
            // Тіло метода
        }
    }
}
```

Лістинг 14.2 – Продовження програми:

```
 static void Main(string[] args)
        {
            // Перевірка параметрів
            if (args.Length < 2)
            {
                Console.WriteLine("Usage: Executable_file_name serverName
path");
                Console.WriteLine("Example: Executable_file_name contoso.com
/");
                return;
            }
            string server = args[0];
            string path = args[1];
            int port = 80;
            IPHostEntry host = null;
            IPEndPoint remoteEndPoint = null;
            Socket client = null;
            // Визначення адреси серверу за його іменем
            try
            {
                Console.WriteLine("Resolving the server name...");
                host = Dns.GetHostEntry(server);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.ToString());
                return;
            }
            // Attempt to connect on each address returned from DNS. Break
out once successfully connected
            foreach (IPAddress address in host.AddressList)
            {
                try
                {
```

```
                    client = new Socket(address.AddressFamily,
SocketType.Stream, ProtocolType.Tcp);
                    Console.WriteLine("Socket() is OK...");
                    remoteEndPoint = new IPEndPoint(address, port);
                    client.Connect(remoteEndPoint);
                    Console.WriteLine("Connect() is OK, address: " + address
+ ", port: " + port);
                      break;
                }
                catch (SocketException ex)
                {
                    Console.WriteLine(ex.ToString());
                }
            }
            // MakeRequest will issue the HTTP download request and write the
server response to the console
            MakeRequest(client, path, server);
            Console.WriteLine("MakeRequest() is OK...");
            client.Close();
            Console.WriteLine("Closing client...");
        }
```

Лістинг 14.3 - Код методу MakeRequest()

```
public static void MakeRequest(Socket client, string path, string server)
        {
            // Format the HTTP GET request string
            string Get = "GET " + path + " HTTP/1.0\r\nHost: " + server
               + "\r\nConnection: Close\r\n\r\n";
            Byte[] ByteGet = Encoding.ASCII.GetBytes(Get);

            // Send the GET request to the connected server
            client.Send(ByteGet);
            Console.WriteLine("Send() is OK...");

            // Create a buffer that is used to read the response
            byte[] responseData = new byte[1024];

            // Read the response and save the ASCII data in a string
            int bytesRead = client.Receive(responseData);
            Console.WriteLine("Receive() is OK...");
            StringBuilder responseString = new StringBuilder();
            while (bytesRead != 0)
            {
                responseString.Append(Encoding.ASCII.GetChars(responseData),
0, bytesRead);
                bytesRead = client.Receive(responseData);
            }
            // Display the response to the console
            Console.WriteLine(responseString.ToString());
        }
    }
}
```

Результат виконання даної програми показано в лістингу 14.4. Через консольний режим відбувається аналіз під’єднання до певної сторінки та

перевірка сокетів. Проводиться аналіз методів та виводиться вміст сторінки на екран.

Лістинг 14.4 – Результат виконання програми

```
C:\>SocketProgramming.exe www.google.com /
Resolving the server name...
Socket() is OK...
Connect() is OK, address: 74.125.39.99, port: 80
Send() is OK...
Receive() is OK...
HTTP/1.0 302 Found
Location: http://www.google.com.ua/
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Set-Cookie:
PREF=ID=a6943867f2b8e048:TM=1228769879:LM=1228769879:S=lmek7iDp1Dh_2Pmq;
expires=Wed, 08-Dec-2010 20:57:59 GMT; path=/; domain=.google.com
Date: Mon, 08 Dec 2008 20:57:59 GMT
Server: gws
Content-Length: 222
Connection: Close

<HTML><HEAD><meta http-equiv="content-type" content="text/html;charset=utf-
8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.com.ua/">here</A>.
</BODY></HTML>

MakeRequest() is OK...
Closing client...
```

5. Програмування протоколу TCP

Для більш високорівневого програмування мереж слід використовувати класи простору імен System.Net, які відповідають конкретним протоколам стеку. Код програми для завантаження Web-сторінки з використанням класу WebClient показано в лістингу 14.5:

Лістинг 14.5 - Програмування протоколу TCP

```
using System;
using System.Net;
using System.IO;
namespace ReadingWeb
{
    class Program
    {
        static void Main(string[] args)
```

```
        {
            try
            {
                WebClient fileReader = new WebClient();
                byte[] byteArray = null;
                Uri addr = new Uri("http://www.google.com/");
                byteArray = fileReader.DownloadData(addr);
                string content =
System.Text.Encoding.ASCII.GetString(byteArray);
     Console.WriteLine(content);
            }
            catch (Exception ex)
            {
                Console.WriteLine(ex.Message);
            }
        }
    }
}
```

6. Створення програми для обміну миттєвими повідомленнями

Приклад такої програми є у електронних додатках до лекції з виглядом на рисунку 14.13. Також інформацію можна знайти на сайтах:

– http://www.geekpedia.com/tutorial239_Csharp-Chat-Part-1---Building-the-Chat-Client.html;

– http://www.geekpedia.com/tutorial240_Csharp-Chat-Part-2---Building-the-Chat-Server.html.

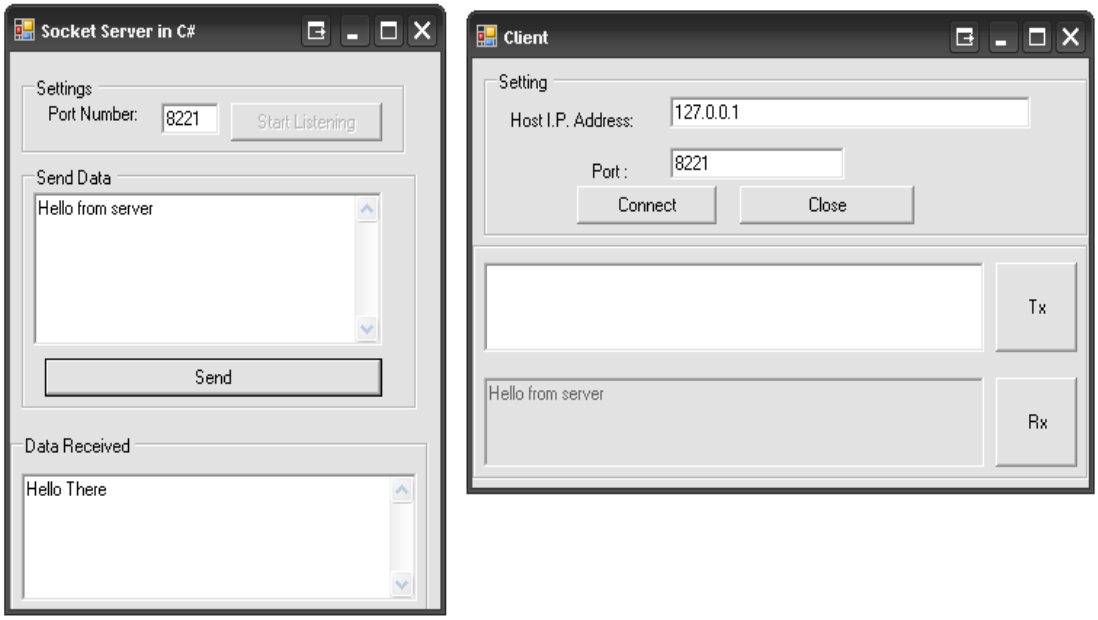


Рисунок 14.13 – Приклад програми – чату

ЛК.15 – СЕРВІС-ОРІЄНТОВАНА АРХІТЕКТУРА

Перелік питань

1. Архітектура та призначення Web-сервісів.
2. Простори імен .NET Framework для Web-сервісів.
3. Послідовність створення Web-сервісу вручну.
4. Тестування Web-сервісу.
5. Генерація Web-сервісу в Visual Studio.
6. Базовий клас WebService.
7. Атрибут [WebService].
8. Атрибут [WebServiceBinding].
9. Атрибут [WebMethod].
10. Мова опису Web-сервісів (WSDL).
11. Протоколи зв’язку у сервіс-орієнтованій архітектурі.
12. Побудова клієнтів Web-сервісів.
13. Використання WCF для розробки Web-сервісів.

На самостійне вивчення:

1. Стандарт пошуку та взаємодії (протокол UDDI) [1, С.1158-1159].

1. Архітектура та призначення Web-сервісів

Web-сервіс – це одиниця програмного коду, яка доступна для виклику за допомогою HTML-запитів. Він надає функціональні можливості, аналогічні стандартній бібліотеці програмного коду .NET (наприклад, спеціальні обчислення, вибірка даних із таблиці, читання цін на активи тощо). В додаток до цієї бібліотеки додатково визначена інфраструктура підтримки. На рисунку 15.1 показана базова схема взаємодії Web-сервісів.

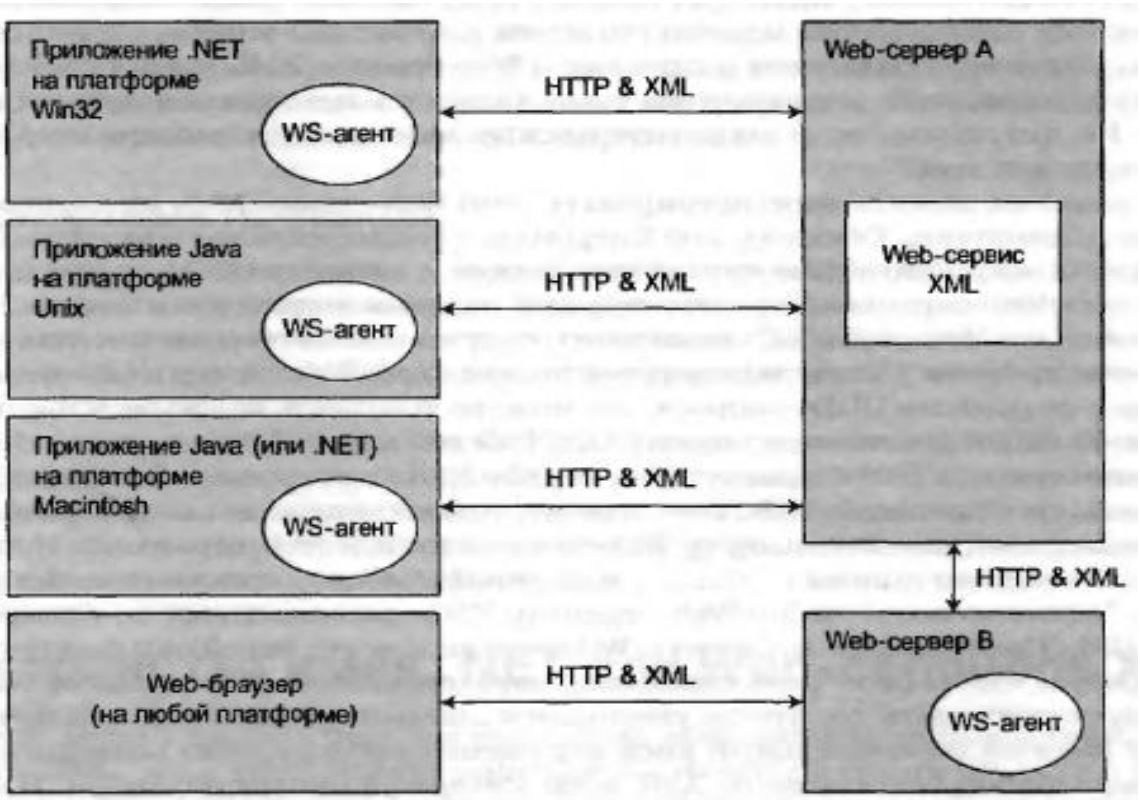


Рисунок 15.1 - Архітектура та призначення Web-сервісів

Web-сервіси XML використовують наступні базові технології:

– служба пошуку (дозволяє виявляти розміщення Web-сервісу).
– служба опису (дозволяє визначати параметри і результат, який повертається сервісом).
– транспортний протокол (дозволяє здійснювати обмін інформацією).

2. Простори імен .NET Framework для Web-сервісів

Бібліотеки базових класів визначають цілий ряд просторів імен, що забезпечують взаємодію з будь-якою з доступних технологій використання Web-сервісів (рис. 15.2). Вони забезпечують безпосередню їх роботу та необхідну взаємодію. Через ці простори реалізується їх опис у мові програмування.

Пространство имен	Описание
System.Web.Services	Содержит базовые типы (включая очень важный атрибут [WebMethod]), необходимые для построения любого Web-сервиса XML
System.Web.Services.Configuration	Содержит типы, позволяющие настроить поведение Web-сервиса XML в среде выполнения ASP.NET
System.Web.Services.Description	Содержит типы, обеспечивающие программное взаимодействие с WSDL-документом, предлагающим описание данного Web-сервиса
System.Web.Services.Discovery	Содержит типы, позволяющие потребителям Web-сервисов выполнять программный поиск Web-сервисов на соответствующей машине
System.Web.Services.Protocols	Определяет ряд типов, представляющих "атомы" различных протоколов связи Web-сервисов XML (HTTP-методы GET и POST, а также SOAP)

Рисунок 15.2 - Простори імен .NET Framework для Web-сервісів

3. Послідовність створення Web-сервісу вручну

Лістинг 15.1 - Код найпростішого Web-сервісу (файл HelloWorldWebService.asmx):

```
<%@ WebService Language="C#" Class="HelloWebService.HelloService" %>
using System;
using System.Web.Services;
namespace HelloWebService
{
    public class HelloService
    {
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello!";
        }
    }
}
```

4. Тестування Web-сервісу

Для тестування Web-сервісу можна використати відладочний Web-сервер, який входить в поставку Visual Studio. Запуск Web-серверу здійснемо наступною командою (за умови, що файл HelloWorldWebService.asmx збережений до папки C:\HelloWorldWebService):

C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\webdev.webserver.exe /port:4955 /path:C:\TestWService.

Внаслідок цього результат після відкриття браузера буде мати вигляд, що показано на рисунку 15.3:

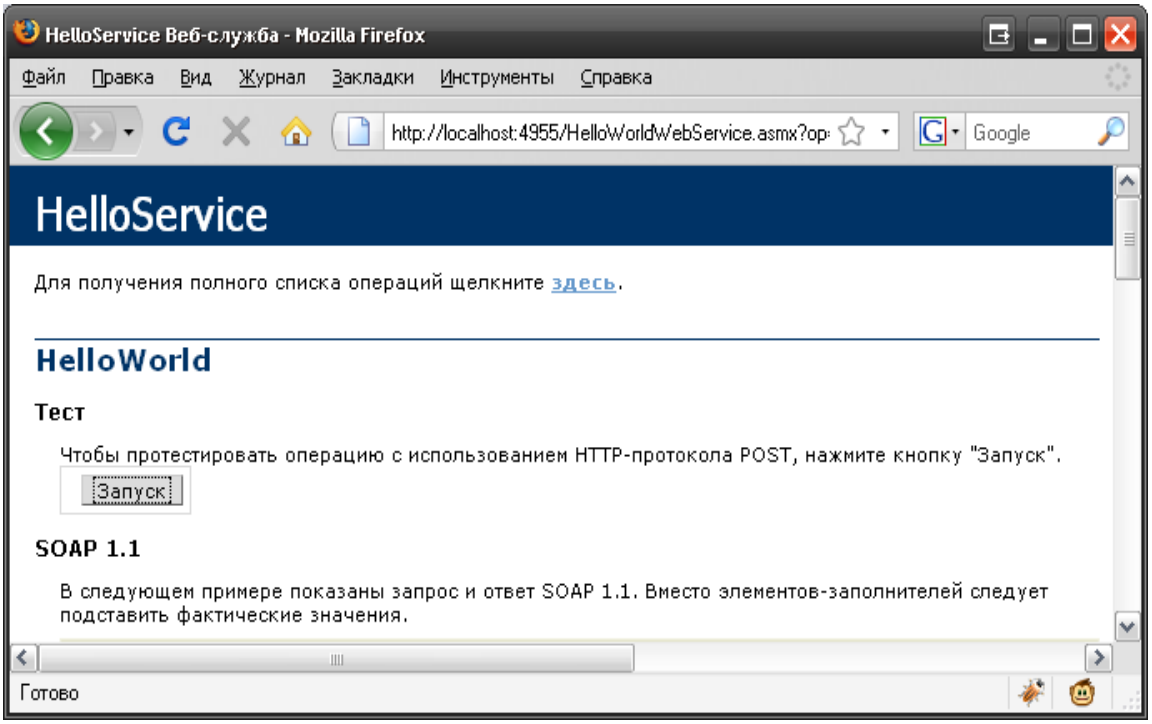


Рисунок 15.3 – Тестування Web-сервісу

5. Генерація Web-сервісу в Visual Studio

У середовищі Visual Studio для автоматичної генерації Web-сервісу існує окремий тип проекту (рис. 15.4):

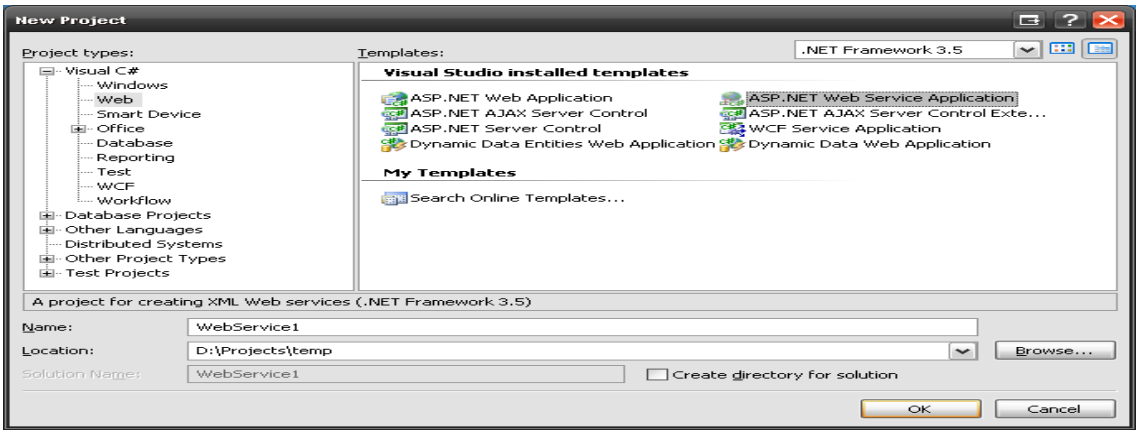


Рисунок 15.4 – Проект Web-сервісу в Visual Studio

Автоматично буде згенеровано після запуску проекту файл Service.asmx. За замовчуванням він визначається так (лістинг 15.2).

Лістинг 15.2 - Код заготовки Web-сервісу у Visual Studio

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Web;
using System.Web.Services;
namespace WebService1
{
    /// <summary>
    /// Summary description for Service1
    /// </summary>
    [WebService(Namespace = "http://tempuri.org/")]
    [WebServiceBinding(ConformsTo = WsiProfiles.BasicProfile1_1)]
    [System.ComponentModel.ToolboxItem(false)]
    // To allow this Web Service to be called from script, using ASP.NET
AJAX,
    // uncomment the following line.
    // [System.Web.Script.Services.ScriptService]
    public class Service1 : System.Web.Services.WebService
    {
        [WebMethod]
        public string HelloWorld()
        {
            return "Hello World";
        }
    }
}
```

6. Базовий клас WebService

Базовий клас WebService наслідується при створенні Web-сервісів, він містить важливі властивості і методи. Опис основних членів цього типу представлений на рисунку 15.5.

Свойство	Описание
Application	Обеспечивает доступ к объекту HttpApplicationState для текущего HTTP-запроса
Context	Обеспечивает доступ к типу HttpContext, инкапсулирующему все HTTP-содержимое, используемое Web-сервером для HTTP-запросов
Server	Обеспечивает доступ к объекту HttpServerUtility для текущего запроса
Session	Обеспечивает доступ к типу HttpSessionState для текущего запроса
SoapVersion	Читает версию протокола SOAP, используемую для SOAP-запросов к Web-сервису XML; это свойство появилось в .NET 2.0

Рисунок 15.5 – Основні члени типу WebService

7. Атрибут [WebService]

Клас Web-сервісу має бути помічений спеціальним атрибутом [WebService], у такому разі він може бути використаний для сторонніх викликів. Атрибут [WebService] має декілька властивостей:

– Namespace – простір імен для Web-сервісу (URL).
– Description – опис.
– Name – назва.

8. Атрибут [WebServiceBinding]

Специфікації, які визначають стандарти створення Web-сервісів (WSDL) змінювалися в процесі розвитку. Атрибут [WebServiceBinding] призначений для задання режиму сумісності, у якому працює Web-сервіс.

9. Атрибут [WebMethod]

Атрибутом [WebMethod] має позначатися кожен метод, який використовується у якості методу Web-сервісу. Властивості атрибуту:

– Description – опис Web-сервісу.
– MessageName – назва методу (може використовуватися для усунення конфліктів імен у перевантажених методах).

10.Мова опису Web-сервісів (WSDL)

Мова опису Web-сервісів (WSDL) регламентує характеристики методів:

- Назва методу.
- Кількість і тип параметрів (якщо передбачені).
- Тип значення, яке повертається.
- Умови виклику.

11.Протоколи зв’язку у сервіс-орієнтованій архітектурі.

Теоретично Web-сервіси можуть використовувати будь-які протоколи, які дозволяють здійснювати віддалений виклик процедур (RPC), наприклад DCOM чи CORBA. Однак у більшості випадків використовуються протоколи HTTP, які забезпечують поширеність і простоту відладки. Режими зв’язку сервісів показано на рисунку 15.6, а типи даних, які підтримуються методами GET та POST – на рисунку 15.7.

Режим связи	Описание
HTTP-метод GET	В режиме обмена GET параметры добавляются к строке запроса данного URL
HTTP-метод POST	В режиме обмена POST данные встраиваются в заголовок HTTP-сообщения, а не добавляются к строке запроса
SOAP	SOAP является протоколом связи, определяющим правила передачи данных и вызова методов в сети с помощью XML

Рисунку 15.6 – Режими зв’язку Web-сервісів

Типы данных	Описание
Перечни	GET и POST поддерживают передачу типов System.Enum .NET, поскольку эти типы представляются в виде статических строковых констант
Простые массивы	Вы можете использовать массивы любых примитивных типов
Строки	GET и POST осуществляют передачу любых числовых данных в виде строковых маркеров. *Строка* здесь на самом деле обозначает строковое представление среды CLR для таких примитивов, как Int16, Int32, Int64, Boolean, Single, Double, Decimal и т.д.

Рисунок 15.7 – Типи даних Web-сервісів

12.Побудова клієнтів Web-сервісів

Клієнтом Web-сервісу може бути будь-який тип проекту у Visual Studio. Для того, щоб була можливість звернутися до Web-сервісу, необхідно у Visual Studio додати посилання на Web-сервіс, що здійснюється за допомогою команди “Add Web Reference…”. Дане діалогове вікно показано на рисунку 15.8:

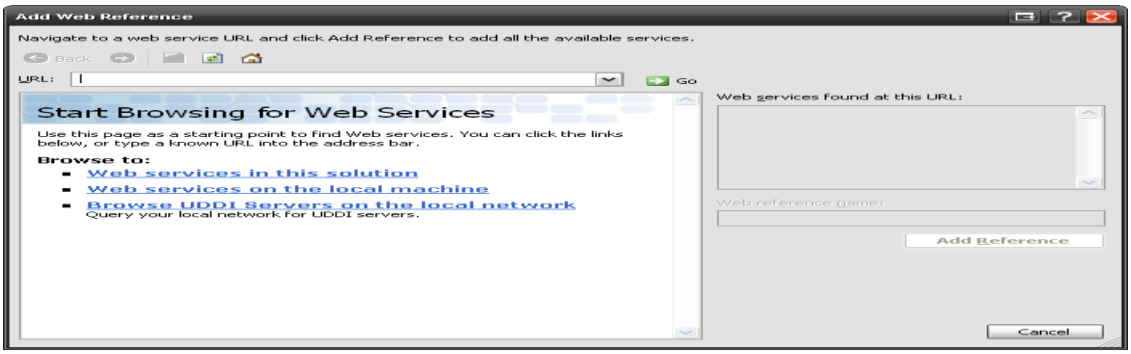


Рисунок 15.8 – Діалогове вікно Add Web Reference

13.Використання WCF для розробки Web-сервісів

WCF (Windows Communication Foundation) – нова платформа від Microsoft, призначена для побудови універсальних комунікаційних рішень (доступна з .NET 3.0). В основі WCF лежить сервіс-орієнтована архітектура і ця платформа є зручною для розробки Web-сервісів (рис. 15.9):

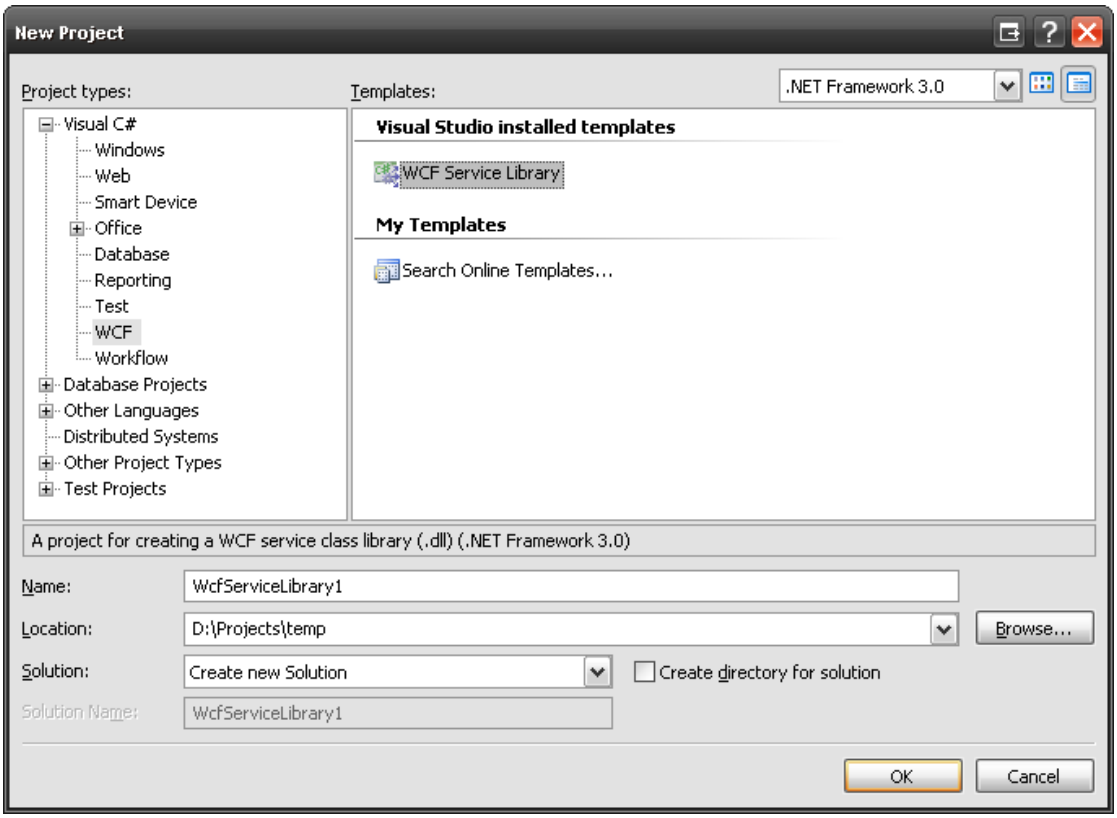


Рисунок 15.9 – Новий проект WCF для розробки Web-сервісів

ЛК.16 – УЗАГАЛЬНЕННЯ

Перелік питань

1. Поняття узагальнення в C#.
2. Простір імен System.Collections.Generic.
3. Тип List<T>.
4. Тип Collection<T>.
5. Тип Comparer<T>.
6. Тип Dictionary <TKey, TValue>.
7. Тип EqualityComparer<T>.
8. Тип LinkedListNode<T>.
9. Тип LinkedList<T>.
10. KeyedCollection<TKey, TItem>.
11. Тип Queue<T>.
12. Тип ReadOnlyCollection<T>.
13. Тип SortedDictionary<TKey, TValue>.
14. Тип SortedList<TKey, TValue>.
15. Тип Stack<T>.
16. Тип BindingList<T>.
17. Створення узагальнених методів.
18. Створення узагальнених структур і класів.
19. Створення узагальнених колекцій.
20. Створення узагальнених базових класів.
21. Створення узагальнених інтерфейсів.

На самостійне вивчення:

1. Створення узагальнених делегатів [1, С.437-439].

1. Поняття узагальнення в C#

Сьогодні існує проблема універсальності класів і строгої типізації даних. Узагальнені класи – це класи, для яких в момент їх декларації є невизначеним тип певного параметру. Лістинг 16.1 демонструє приклад використання узагальненого класу.

Лістинг 16.1 - Узагальнений клас в C#

```
List<int> ints = new List<int>();
ints.Add(1);
ints.Add(2);
```

2. Простір імен System.Collections.Generic

Простір імен System.Collections.Generic містить набір заздалегідь визначених класів і інтерфейсів для реалізації узагальнень. Інтерфейси:

– ICollection<T>;
– IComparer<T>;
– IDictionary<K, V>;
– IEnumerable<T>;
– IEnumerator<T>;
– IList<T>.

За прийнятою домовленістю літеру T використовують для позначення типів, літеру K – ключів, V – значень. Основні класи з їх описом даного простору імен показано в таблиці 16.1.

Таблиця 16.1 – Класи простору імен System.Collections.Generic

Обобщенный класс	Необобщенный аналог в System.Collections	Описание
Collection<T>	CollectionBase	База для обобщенной коллекции
Comparer<T>	Comparer	Выполняет сравнение двух обобщенных объектов
Dictionary<K, V>	Hashtable	Обобщенная коллекция пар имен и значений
List<T>	ArrayList	Список элементов с динамически изменяемыми размерами
Queue<T>	Queue	Обобщенная реализация списка FIFO (дисциплина обслуживания типа “очередь”)
SortedDictionary<K, V>	SortedList	Обобщенная реализация сортированного набора пар имен и значений
Stack<T>	Stack	Обобщенная реализация списка LIFO (дисциплина обслуживания типа “стек”)
LinkedList<T>	—	Обобщенная реализация двусвязного списка
ReadOnlyCollection<T>	ReadOnlyCollectionBase	Обобщенная реализация набора элементов только для чтения

3. Тип List<T>

Тип List<T> реалізує узагальнений список елементів, розміри якого можуть змінюватися. Важливі методи і властивості:

– Capacity – отримує і задає кількість елементів, які може вмістити внутрішня структура без зміни розміру;
– Count – кількість елементів у списку;
– Item – повертає чи задає елемент за певним індексом;
– Add(T item) – додати елемент;
– Insert(int index, T item) – вставити елемент у певну позицію;
– Contains(T item) – перевіряє наявність елементу у списку;
– TrimExcess() – обмежити розмір внутрішньої структури кількістю елементів у списку;
– Clear() – видалити всі елементи із списку;
– BinarySearch(T) – виконує бінарний пошук у впорядкованому списку.

Приклад використання List<T> показано в лістингу 16.2.

Лістинг 16.2 – Використання List<T>

```
ususing System.Collections.Generic;
public class Example
{
public static void Main()
    {
        List<string> dinosaurs = new List<string>();
        dinosaurs.Add("Tyrannosaurus");
        dinosaurs.Add("Amargasaurus");
        dinosaurs.Add("Mamenchisaurus");

        Console.WriteLine("\nCapacity: {0}", dinosaurs.Capacity);
        Console.WriteLine("Count: {0}", dinosaurs.Count);
        Console.WriteLine("\nContains(\"Amargasaurus\"): {0}",
            dinosaurs.Contains("Amargasaurus"));
        Console.WriteLine("\nInsert(2, \"Compsognathus\")");
        dinosaurs.Insert(2, "Compsognathus");
        Console.WriteLine();
        foreach (string dinosaur in dinosaurs)
        {
            Console.WriteLine(dinosaur);
        }
        Console.WriteLine("\ndinosaurs[3]: {0}", dinosaurs[3]);
        Console.WriteLine("\nRemove(\"Compsognathus\")");
        dinosaurs.Remove("Compsognathus");
    }
}
```

4. Тип Collection<T>

Тип Collection<T> задекларований у просторі імен System.Collections.ObjectModel і призначений для реалізації базової колекції, що наслідується для виконання певних специфічних задач.

Тип Collection<T> дуже схожий до типу List<T>, однак головне його призначення – виступати у якості основи для наслідування, в той час як List<T> оптимізований по швидкості, призначений для створення кінцевих списків , і не рекомендований для використання у якості основи для наслідування.

5. Тип Comparer<T>

Тип Comparer<T> призначений для реалізації порівняння екземплярів типу T, що необхідно при використанні класів, що передбачають використання впорядкованого списку елементів, наприклад, SortedList<K,V> чи SortedDictionary<K,V>. Може використовуватися для впорядкування елементів у будь-якому іншому класі, який передбачає таку можливість.

Властивість Default дозволяє створювати екземпляри класу, у яких процедура порівняння визначається за замовченням відповідно до типу даних. Роботу з Comparer<T> показано в лістингу 16.3.

Лістинг 16.3 – Приклад використання Comparer<T>

```
public class ComparerTest
{
    public void NameSortTest()
    {
        List<Customer> collCustList = new List<Customer>();
        collCustList.Add(new Customer(99, "Happy Gillmore", "Platinum"));
        collCustList.Add(new Customer(77, "Billy Madison", "Gold"));
        collCustList.Add(new Customer(55, "Bobby Boucher", "Gold"));
        collCustList.Add(new Customer(88, "Barry Egan", "Platinum"));
        collCustList.Add(new Customer(11, "Longfellow Deeds", "Other"));

        Console.Out.WriteLine("Before Sort:");
        foreach (Customer cust in collCustList)
            Console.Out.WriteLine(cust);

        Customer.Order = Customer.SortOrder.Ascending;
        collCustList.Sort(delegate(Customer cust1, Customer cust2)
        {
            return Comparer<Customer>.Default.Compare(cust1, cust2);
        });
```

Продовження лістингу 16.3:

```
        Console.Out.WriteLine("After Ascending Sort:");
        foreach (Customer cust in collCustList)
            Console.Out.WriteLine(cust);

        Customer.Order = Customer.SortOrder.Descending;
        collCustList.Sort(delegate(Customer cust1, Customer cust2)
        {
            return Comparer<Customer>.Default.Compare(cust1, cust2);
        });

        Console.Out.WriteLine("After Descending Sort:");
        foreach (Customer cust in collCustList)
            Console.Out.WriteLine(cust);
    }
}
```

6. Тип Dictionary <TKey, TValue>

Тип Dictionary <TKey, TValue> представляє колекцію ключів і значень (словник). Витягнення значення, якщо відомий його ключ, відбувається дуже швидко (внутрішня структура збереження даних працює за принципом хешування). Кожен ключ у словнику має бути унікальним і не може бути пустим. Робота з Dictionary <TKey, TValue> показано в лістингах 16.4, 16.5, 16.6.

Лістинг 16.4 - Приклад створення Dictionary <TKey, TValue>

```
public void ConstuctorTest() {
Dictionary<int, Customer> custDict = new Dictionary<int, Customer>
    (Comparer<int>.Default);
custDict[99] = new Customer(99, "Happy Gillmore", "Platinum");
custDict[77] = new Customer(77, "Billy Madison", "Gold");
custDict[55] = new Customer(55, "Bobby Boucher", "Silver");
custDict[88] = new Customer(88, "Barry Egan", "Platinum");

Dictionary<int, Customer> copyDict = new Dictionary<int, Customer>(custDict);
copyDict[55] = new Customer(55, "Longfellow Deeds", "Other");


Dictionary<int, Customer>.ValueCollection custValues = custDict.Values;
foreach (Customer cust in custValues)
    Console.Out.WriteLine(cust);
   custValues = copyDict.Values;

foreach (Customer cust in custValues)
    Console.Out.WriteLine(cust);

Dictionary<int, Customer> capacityDict =
  new Dictionary<int, Customer>(100);
```

Лістинг 16.5 - Приклад заповнення Dictionary<TKey, TValue>

```
public void AddUpdateRetrieveItems() {
    Dictionary<int, Customer> custDict = new Dictionary<int,Customer>();
    custDict[99] = new Customer(99, "Happy Gillmore", "Platinum");
    custDict[77] = new Customer(77, "Billy Madison", "Gold");
    custDict[55] = new Customer(55, "Bobby Boucher", "Silver");
    custDict[88] = new Customer(88, "Barry Egan", "Platinum");

    custDict.Add(11, new Customer(11, "Longfellow Deeds", "Other"));
    custDict.Add(77, new Customer(77, "Test Person", "Gold"));

    custDict[55] = new Customer(55, "Bobby Boucher", "Gold");

    Customer tmpCust = custDict[88];
    tmpCust.Rating = "Other";
}
```

Лістинг 16.6 - Приклад пошуку дублюючих записів у Dictionary<TKey, TValue>

```
public Collection<Customer> FindDupeCustomers
      (Collection<Customer> customers) {
    Collection<Customer> retVal =
        new Collection<Customer>();
    foreach (Customer cust in customers) {
        Customer tmpCust = new Customer(0, "New");
        if (myDictionary.TryGetValue(cust.Id, out tmpCust)
              == true) {
            retVal.Add(tmpCust);
            Console.Out.WriteLine("Dupe Customer Found: {0}",
              cust.Id);
        }

    }
    return retVal;
}
```

7. Тип EqualityComparer<T>

Він забезпечує перевірку рівності значень (лістинг 16.7):

Лістинг 16.7 - Тип EqualityComparer<T>

```
public class EqualityTest<T>
{
    private T _value;
    public EqualityTest(T value)
    {
        if (EqualityComparer<T>.Equals(value, this._value) == false)
            this._value = value;
    }
}
```

8. Тип LinkedListNode<T>

Реалізує вузол для зв’язаного списку (LinkedList<T>). Основними властивостями є такі:

- List;
- Next;
- Previous;
- Value.

9. Тип LinkedList<T>

Реалізує зв’язний список. Елементи представляються за допомогою класу LinkedListNode<T>. Деякі його основні методи і властивості:

- AddAfter();
- AddBefore();
- AddFirst();
- AddHead();
- AddLast();
- AddTrail();
- Count;
- Find();
- First;
- Head;
- Last;
- Remove().

10.KeyedCollection<TKey, TItem>

Цей клас за своєю поведінкою дуже схожий до Dictionary<TKey, TItem>, однак призначений в першу чергу для створення спеціалізованих словників і має виступати у якості основи для наслідування.

11.Тип Queue<T>

Цей клас схожий до List<T>, однак реалізує чергу. Приклад використання показано в лістингу 16.8.

Лістинг 16.8 – Використання Queue<T>

```
Queue<Customer> custQueue = new Queue<Customer>(100);
custQueue.Enqueue(new Customer(99, "Happy Gillmore", "Platinum"));
custQueue.Enqueue(new Customer(77, "Billy Madison", "Gold"));
custQueue.Enqueue(new Customer(55, "Bobby Boucher", "Silver"));
custQueue.Enqueue(new Customer(88, "Barry Egan", "Platinum"));

Customer tmpCust = custQueue.Dequeue();
Console.Out.WriteLine("Dequeued: {0}", tmpCust);

foreach (Customer cust in custQueue)
    Console.Out.WriteLine("Queue Item: {0}", cust);

tmpCust = custQueue.Peek();
Console.Out.WriteLine("Queue Peek: {0}", tmpCust);

custQueue.Enqueue(new Customer(88, "Barry Egan", "Platinum"));

foreach (Customer cust in custQueue)
    Console.Out.WriteLine("Queue Item: {0}", cust);
```

12.Тип ReadOnlyCollection<T>

Цей клас реалізує колекцію, елементи якої не можна видаляти, додавати чи заміняти. Водночас самі елементи можуть бути змінені. Приклад використання показано в лістингу 16.9.

Лістинг 16.9 – Використання ReadOnlyCollection<T>

```
static public void BuildReadOnlyCollection()
{
    List<Customer> collCustList = new List<Customer>();
    collCustList.Add(new Customer(99, "Happy Gillmore", "Platinum"));
    collCustList.Add(new Customer(77, "Billy Madison", "Gold"));
    collCustList.Add(new Customer(55, "Bobby Boucher", "Gold"));
    collCustList.Add(new Customer(88, "Barry Egan", "Platinum"));
    collCustList.Add(new Customer(11, "Longfellow Deeds", "Other"));
    Collection<Customer> roCustColl =
        new Collection<Customer>(collCustList);
}
```

13.Тип SortedDictionary<TKey, TValue>

Тип SortedDictionary<TKey, TValue> схожий за своєю поведінкою до типу Dictionary<TKey, TValue> за винятком того, що його елементи впорядковуються за значеннями ключа, що може бути зручно у тому випадку, коли необхідно маніпулювати словником подібно до впорядкованої колекції.

14.Тип SortedList<TKey, TValue>

Реалізує впорядкований список подібно до SortedDictionary<TKey, TValue>, принципова відмінність полягає у способі збереженні інформації і швидкості роботи:

- SortedList<TKey, TValue> вимагає менше пам’яті;
- Операції вставки і видалення при роботі із невпорядкованими даними у SortedList<TKey, TValue> є повільнішими;
- При обробці списку на основі впорядкованих даних SortedList<TKey, TValue> функціонує швидше.

15. Тип Stack<T>

Належить до зв’язаних списків, реалізує стек.

16.Тип BindingList<T>

Даний тип використовується для створення списків, що необхідні для зв’язування різних елементів управління (DataGrid, DataGridView), як правило не використовується напряму.

17.Створення узагальнених методів.

Для створення узагальненого методу необхідно після зазначення назви методу у кутових дужках указати замінник типу, який може бути

використаний для опису параметрів чи значення, яке повертається, наприклад: static void SomeMethod<T>(a T, b T).

Приклад узагальненого методу (лістинг 16.10):

Лістинг 16.10 – Узагальнений метод

```
static void Swap<T>(ref T a, ref T b)
{
    Console.WriteLine("Методу Swap() передано тип {0}",
      typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Під час виклику узагальнених методів (лістинг 16.11) бажано зазначати узагальнений тип, наприклад: Swap<int>(ref a, ref b). Однак ця вимога не є обов’язковою у тому випадку, коли компілятор може визначити цей тип (зокрема, за типом змінних, які передаються у якості параметру) в такому разі узагальнений тип можна не позначати: Swap(ref a, ref b).

Проте у тих випадках, коли узагальнений метод не приймає узагальнених параметрів (наприклад, коли він повертає значення узагальненого типу, чи лише звертається до інформації типу), то узагальнений тип необхідно зазначати обов’язково.

Лістинг 16.11 - Приклад програми, яка використовує узагальнений метод

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

namespace ConsoleApplication15
{
    class Program
    {

        static void Swap<T>(ref T a, ref T b)
        {
            Console.WriteLine("Методу Swap() передано тип {0}", typeof(T));
            T temp;
            temp = a;
            a = b;
```

```
            b = temp;
        }

        static void Main(string[] args)
        {
            int a = 10, b = 30;
            Swap<int>(ref a, ref b);
            Console.WriteLine("a = {0}, b = {1}", a, b);

            string s1 = "Програмувати", s2 = "Цікаво";
            Swap(ref s1, ref s2);
            Console.WriteLine("s1 = {0}, s2 = {1}", s1, s2);
        }
    }
}
```

18.Створення узагальнених структур і класів

Декларація узагальнених структур і класів здійснюється наступним чином (лістинг 16.12).

Лістинг 16.12 - Декларація узагальнених структур і класів

```
public struct Point<T>
{
    // Узагальнені дані
    private T xPos;
    private T yPos;
    // Узагальнений конструктор
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }
    // Узагальнені властивості
    public T X
    {
        get { return xPos; }
        set { xPos = value; }
    }
    public T Y
    {
        get { return yPos; }
        set { yPos = value; }
    }
}
```

В узагальнених структурах і класах можна використовувати ключове слово default разом із параметром узагальнюючого типу для отримання значення за замовчуванням:

– 0 для числових значень;

– null для значень за посиланням.

19.Створення узагальнених колекцій

Колекції – класи, призначені для збереження і маніпуляцій над списками певного типу (лістинг 16.13).

Лістинг 16.13 - Приклад створення узагальненої колекції

```
public class CarCollection<T> : IEnumerable<T> where T : Car
{
    private List<T> arCars = new List<T>();
    public T GetCar(int pos)
    { return arCars[pos]; }
    public void AddCar(T c)
    { arCars.Add(c); }
    public void ClearCars()
    { arCars.Clear(); }
    public int Count
    { get { return arCars.Count; } }
    IEnumerator<T> IEnumerable<T>.GetEnumerator()
    { return arCars.GetEnumerator(); }
    System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator()
{ return arCars.GetEnumerator(); }
}
```

20.Створення узагальнених базових класів.

Узагальнені класи можуть виступати у якості базових класів як для конкретних, так і інших узагальнених класів (лістинг 16.14).

Лістинг 16.14 - Створення узагальнених базових класів

```
// Базовий узагальнений клас
public class MyList<T>
{
    private List<T> ListOfData = new List<T>();
    public virtual void PrintList(T data) { }
}
// Наслідуваний конкретний клас
public class MyStringList : MyList<string>
{
    public override void PrintList(string data)
    { }
}
// Наслідуваний узагальнений клас
public class MyReadOnlyList<T> : MyList<T>
{
    public override void PrintList(T data)
    {
    }
}
```

21.Створення узагальнених інтерфейсів

За тими ж правилами, що й для структур і класів, можуть створюватися узагальнені інтерфейси (лістинг 16.15).

Лістинг 16.15 - Створення узагальнених інтерфейсів

```
public interface IBinaryOperations<T> where T : struct
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}
```

ЛК.17 – МОВА LINQ

Перелік питань

1. Введення до мови LINQ.
2. Розширення мови C# для підтримки LINQ.
3. Синтаксис LINQ.
4. Доступ до об’єктів в оперативній пам’яті за допомогою LINQ.
5. Доступ до реляційних даних за допомогою LINQ.
6. Доступ до XML за допомогою LINQ.

На самостійне вивчення:

1. Серіалізація об'єктів [1, С.677-699].

1. Введення до мови LINQ

Історично склалося, що між мовами для реалізації алгоритмів у програмах і мовами обробки даних у СУБД існує великий, навіть принциповий розрив: перші є у переважній більшості імперативними (алгоритмічними), другі – декларативними (SQL). У версії C# 3.0, яка вийшла разом із Visual Studio 2008, з’явилася вбудована декларативна мова LINQ, яка є дуже зручною для роботи з даними.

LINQ – Language INtegrated Query – дуже схожа до SQL мова формування запитів як до зовнішніх джерел, так і до внутрішніх структур даних у C#.

2. Розширення мови C# для підтримки LINQ

Для того, щоб була можливість реалізувати LINQ, мова C# була розширена наступними можливостями:

- виведення типу (type inference);
- анонімні типи (anonymous types);
- розширяючі методи (extension methods);

– лямбда-вирази (lambda expressions);
– дерево виразів (expression tree);
– відкладені обчислення (lazy evaluation).

Виведення типу дозволяє не зазначати тип при декларації змінної, він визначається автоматично в момент присвоєння значення (лістинг 17.1).

Лістинг 17.1 - Виведення типу (type inference)

```
// Замість того, щоб оголошувати змінну таким чином:
MyLongFooClassWithTemplate<MyLongTypeParameter> local
= new MyLongFooClassWithTemplate<MyLongTypeParameter>();

// Можна оголосити змінну так:
var localVar = new MyLongFooClassWithTemplate<MyLongTypeParameter>();

// Чи так:
var i = 1;
var s = "SomeString";
var z = i + i * i;
var c = 'c';
Console.WriteLine(i.GetType()); // System.Int32
Console.WriteLine(s.GetType()); // System.String
Console.WriteLine(c.GetType()); // System.Char
Console.WriteLine(z.GetType()); // System.Int32
```

Анонімні типи – можливість створювати змінні, тип яких не є визначеним на момент створення змінної, а генерується автоматично. Таким чином відпадає потреба у декларації типів, які потрібні лише для епізодичного використання і лише для отримання даних (лістинг 17.2).

Лістинг 17.2 - Анонімні типи (anonymous types)

```
// Оголосимо новий тип:
var at = new { a = 5, b = 3.14, c = "рядок" };

// Далі його можна використовувати
Console.WriteLine("a = " + at.a + "\tb = " + at.b + "\tc = " + at.c);
```

Розширяючі методи дозволяють розширювати існуючі типи новими методами більш спрощено, в обхід наслідування. Для реалізації необхідно створити метод, що приймає тип, для якого здійснюється розширення, з ключовим словом “this” (лістинг 17.3).

Лістинг 17.3 – Розширяючі методи (extension methods)

```
using System;
namespace DemoExtensioMethods
{
    public static class MyExtensions
    {
        public static int WordCount(this String str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
                StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            string s = "Тестовий рядок з декількома словами";
            Console.WriteLine(s.WordCount());
        }
    }
}
```

Лямбда-вирази – термін, запозичений із функціонального програмування, використовується для створення анонімних функцій (“функція” означає, що повертається результат). Декларація лямбда-виразів здійснюється наступним чином (лістинг 17.4, 17.5).

Лістинг 17.4 - Лямбда-вирази (lambda expressions)

```
public delegate int MultiplyInts(int arg, int arg2);
MultiplyInts myDelegate = (a, b) => a * b;
Console.WriteLine("{0}", myDelegate(5, 2));
```

Лістинг 17.5 - Приклад декларації лямбда-виразу всередині виразу без створення делегату

```
int[] source = new[] { 3, 8, 4, 6, 1, 7, 9, 2, 4, 8 };

foreach (int i in source.Where(
        x =>
        {
            if (x <= 3)
                return true;
            else if (x >= 7)
                return true;
            return false;
        }
    ))
    Console.WriteLine(i);
```

Дерево виразів – технологія, яка дозволяє динамічно формувати вирази у коді програми за тими ж підходами, які використовуються компілятором при аналізі програмного коду, що дозволяє виконувати програмний код, який міг бути не визначеним на момент написання програми. Таке дерево демонструється на рисунку 17.1.

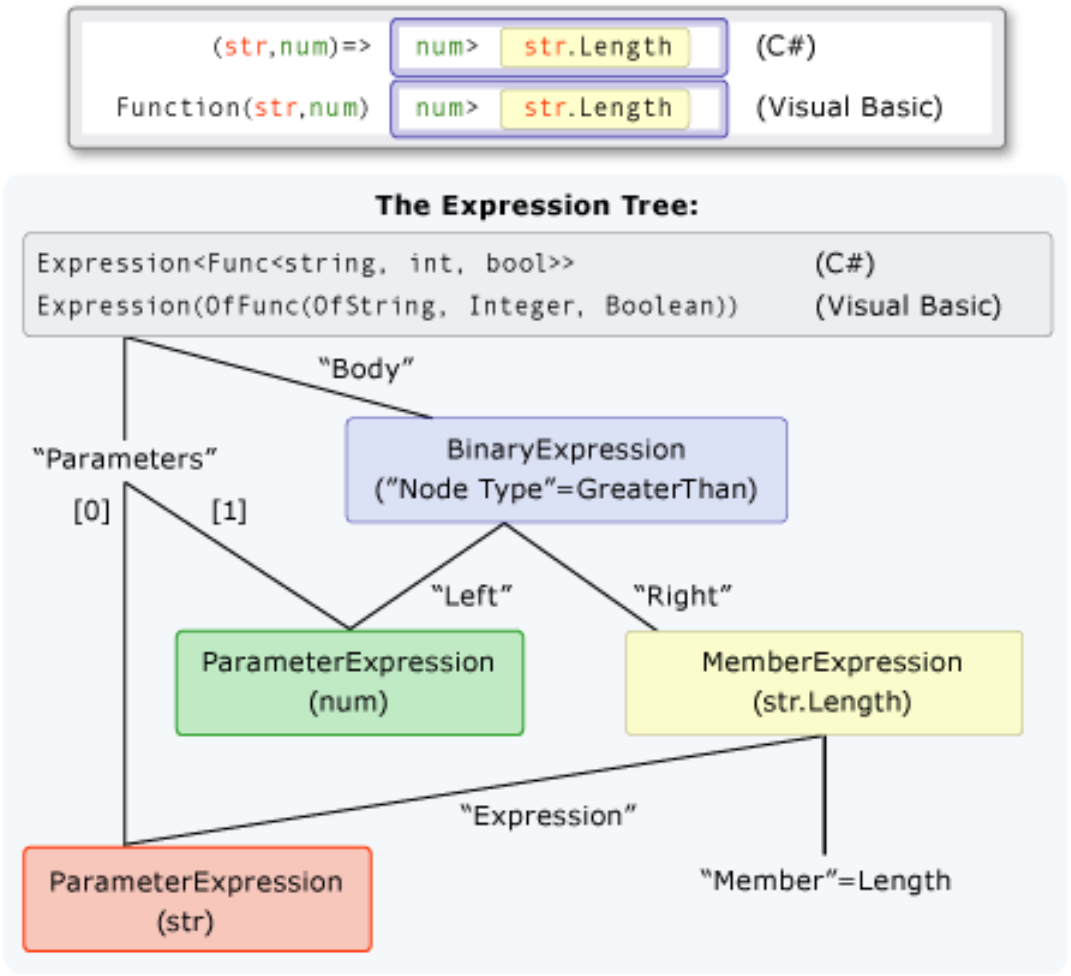


Рисунок 17.1 – Приклад дерева виразів

Кодова функціональність реалізована в лістингу 17.6.

Лістинг 17.6 - Дерево виразів

```
// Для використання дерева виразів необхідно використати простір імен:
// using System.Linq.Expressions;

// Побудова дерева виразів "вручну"
// для лямбда-функції num => num < 5.
ParameterExpression numParam = Expression.Parameter(typeof(int), "num");
ConstantExpression five = Expression.Constant(5, typeof(int));
```

Продовження лістингу 17.6:

```
BinaryExpression numLessThanFive = Expression.LessThan(numParam, five);
Expression<Func<int, bool>> lambda1 =
    Expression.Lambda<Func<int, bool>>(
        numLessThanFive,
        new ParameterExpression[] { numParam });

// Автоматична генерація дерева виразів
// для лямбда-функції num => num < 5.
Expression<Func<int, bool>> lambda2 = num => num < 5;
```

3. Синтаксис LINQ (рис. 17.2)

Синтаксис LINQ представлено на рис. 17.2.

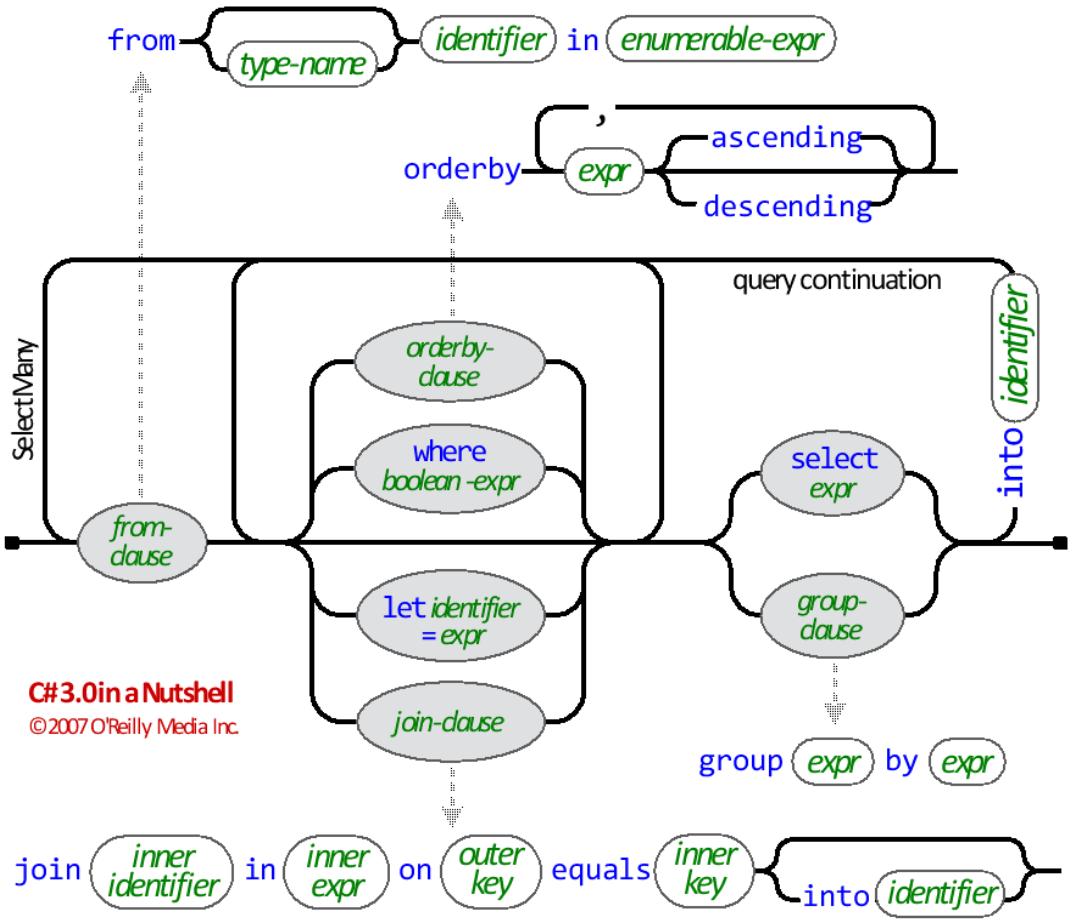


Рисунок 17.2 – Графічне представлення синтаксису мови LINQ

4. Доступ до об’єктів в оперативній пам’яті за допомогою LINQ

За допомогою LINQ можна звертатися до об’єктів в оперативній пам’яті, які підтримують інтерфейс IEnumerable (дозволяють використовувати ітератор foreach), а також до масивів.

Для прикладу розглянемо код програми, яка обирає парні елементи із масиву цілих чисел, сортує їх і виводить результат на екран у двох варіантах – спочатку без використання LINQ, потім з використанням LINQ (лістинг 17.7, 17.8). Також лістинг 17.9 показує процес групування слів у масиві з використанням LINQ.

Лістинг 17.7 - Програма, яка обирає із масиву парні числа, сортує результат і виводить на екран (без використання LINQ):

```
static void Main(string[] args)
{
    int[] arr = { 10, 5, 13, 18, 4, 24, 65, 41, 30 };
    List<int> evens = new List<int>();
    foreach (var number in arr)
    {
        if (number % 2 == 0)
            evens.Add(number);
    }
    evens.Sort();
    foreach (int number in evens)
        Console.WriteLine(number);
}
```

Лістинг 17.8 - Програма, яка обирає із масиву парні числа, сортує результат і виводить на екран (з використанням LINQ):

```
static void Main(string[] args)
{
    int[] arr = { 10, 5, 13, 18, 4, 24, 65, 41, 30 };

    var evens =
        from number in arr
        where number % 2 == 0
        orderby number
        select number;

    foreach (int number in evens)
        Console.WriteLine(number);

}
```

Лістинг17.9 - Групування слів у масиві з використанням LINQ

```
static void Main()
{
   string[] words = { "LINQ", "позволяет", "сделать", "многое", "гораздо",
"проще" };

   var groups =
        from word in words
        orderby word ascending
        group word by word.Length into lengthGroups
   orderby lengthGroups.Key descending
    select new { Length = lengthGroups.Key, Words = lengthGroups };

    foreach (var group in groups)
    {
        Console.WriteLine("Слова длиной {0} символов", group.Length);
        foreach (string word in group.Words)
            Console.WriteLine(" " + word);
    }
}
```

5. Доступ до реляційних даних за допомогою LINQ

За допомогою технології LINQ to SQL можна виконувати запити до реляційної БД безпосередньо із програмного коду у C# без використання об’єктів доступу до БД та SQL.

Для того, щоб була можливість звернутися до БД SQL Server за допомогою LINQ, необхідно спочатку створити проміжну ланку – LINQ to SQL Classes, за допомогою якої слід підключитися до серверу та обрати таблиці, які будуть доступні у LINQ-запиті, далі, використовуючи звичайний синтаксис LINQ, можна звертатися до об’єктів у БД. Даний доступ у мові C# можна отримати так (рис. 17.3).

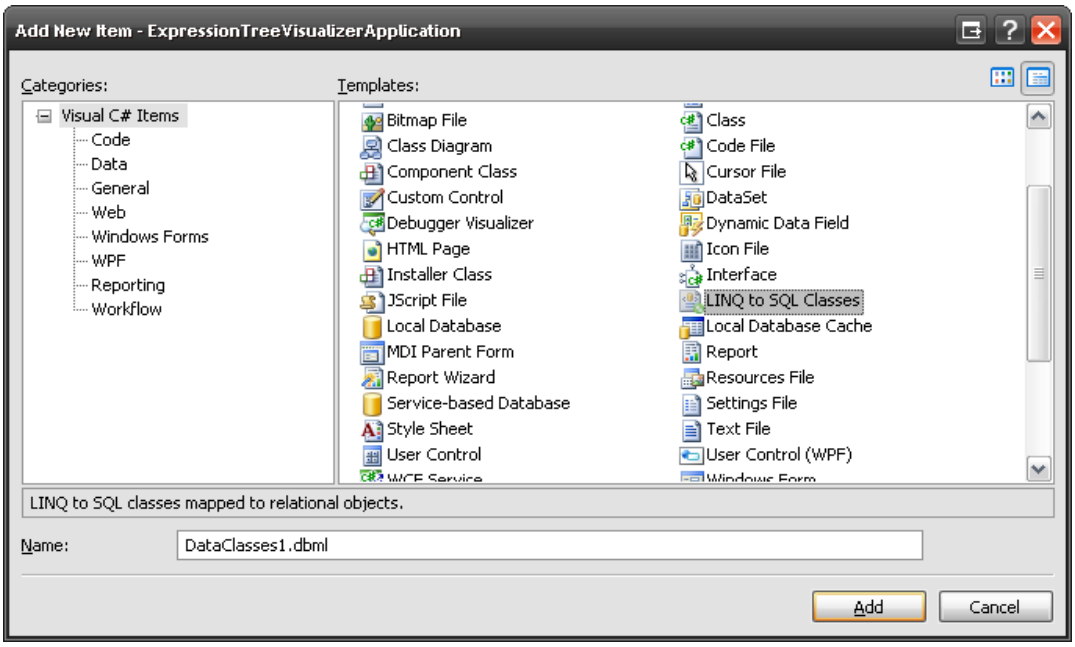


Рисунок 17.3 – Додання проекту для доступу до реляційних даних

6. Доступ до XML за допомогою LINQ

За допомогою LINQ to XML можна здійснювати LINQ-запити до XML-файлів (лістинг 17.10, 17.11).

Лістинг 17.10 – Доступ на мові HTML

```
<?xml version="1.0"?>
<catalog>
  <book id="bk101">
    <author>Gambardella, Matthew</author>
    <title>XML Developer's Guide</title>
    <genre>Computer</genre>
    <price>44.95</price>
    <publish_date>2000-10-01</publish_date>
    <description>
      An in-depth look at creating applications
      with XML.
    </description>
  </book>
  <book id="bk102">
      …
  </book>
</catalog>
```

Лістинг 17.11 – Доступ до XML мовою LINQ

```
XElement xElement = XElement.Load("books.xml");
var query = from c in xElement.Descendants("book")
            where c.Element("description").Value.Contains("XML")
            select c;
foreach (var q in query)
{
    Console.WriteLine(q.Element("author").Value + "-" +
q.Element("title").Value);
}
```

ЛК.18 – ЗАБЕЗПЕЧЕННЯ ЯКОСТІ ПРОГРАМНОГО ПРОДУКТУ

Перелік питань

1. Введення до тестування та розробки керованої тестуванням.
2. Поняття та види модульних тестів.
3. Створення модульних тестів у Visual Studio.
4. Створення модульного тесту для перевірки списків.
5. Створення модульного тесту для перевірки алгоритму сортування.
6. Введення до методів відладки комп’ютерних програм.
7. Процедура здійснення пошуку та виправлення помилок.
8. Виявлення та усунення типових помилок.
9. Введення до захищеного програмування.
10. Основи профілювання програм.
11. Виявлення вузьких місць та оптимізація програмного коду деяких типових алгоритмів.

На самостійне вивчення:

1. Додаткові засоби розробки .NET-програм [1, С.111-112]

1. Введення до тестування та розробки керованої тестуванням

Тестування – обов’язкова складова сучасної розробки програмного забезпечення. Тестуванню і перевірці підлягають усі складові проекту – починаючи з вимог, включаючи специфікації і проект, закінчуючи програмним кодом і готовим продуктом.

Розробка, керована тестуванням – підхід, у якому спочатку створюються тести, а потім розробляється програмний код, який їх задовольняє.

2. Поняття та види модульних тестів

Модульний тест (unit test) – це тест для окремої складової програми (модуля), яким може бути, наприклад, клас. Ціль модульного тестування – виявлення локалізованих у модулі помилок, здійснюється за принципом “білої скриньки”, коли розробнику тесту відома внутрішня будова модуля, для якого здійснюється тестування.

В залежності від того, що саме ставиться за основу для побудови модульних тестів, їх розподіляють на:

– тести, побудовані на основі аналізу потоку керування;
– тести, побудовані на основі аналізу потоку даних.

3. Створення модульних тестів у Visual Studio

Середовище Visual Studio містить вбудовані можливості для створення модульних тестів (рис. 18.1).

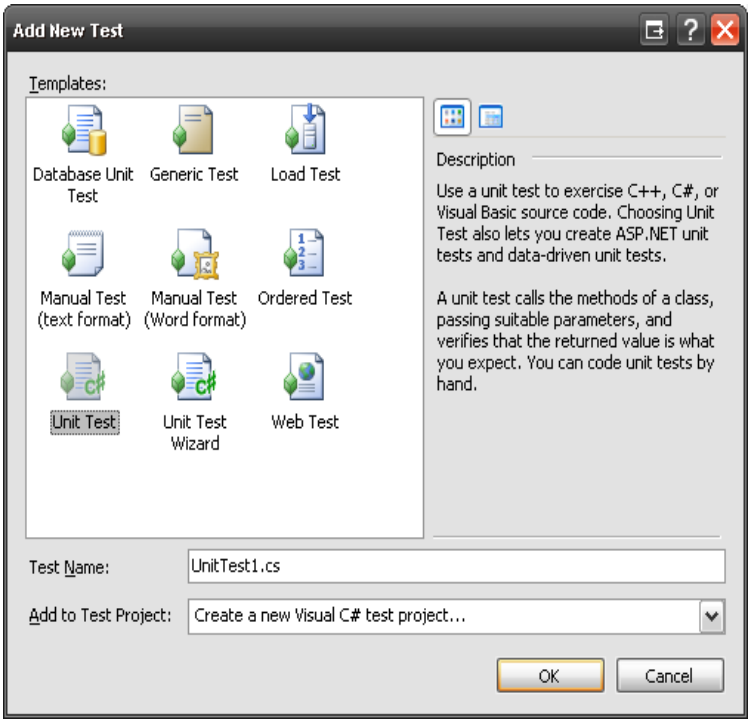


Рисунок 18.1 – Створення модульного тесту

Створимо модульний тест для наступного класу, який реалізує дві математичні операції – складання і множення (друга операція для прикладу реалізована з помилкою), як показано в лістингу 18.1.

Лістинг 18.1 - Приклад створення модульного тесту

```
class MathOps
{
    public int DoSum(int A, int B)
    {
        return A + B;
    }

    public int DoMult(int A, int B)
    {
        return A - B;
    }
}
```

Переводимо курсор до класу, обираємо “Create Unit Tests…” в контекстному меню, у вікні, що з’явилося необхідно обрати методи для створення тестів (рис. 18.2).

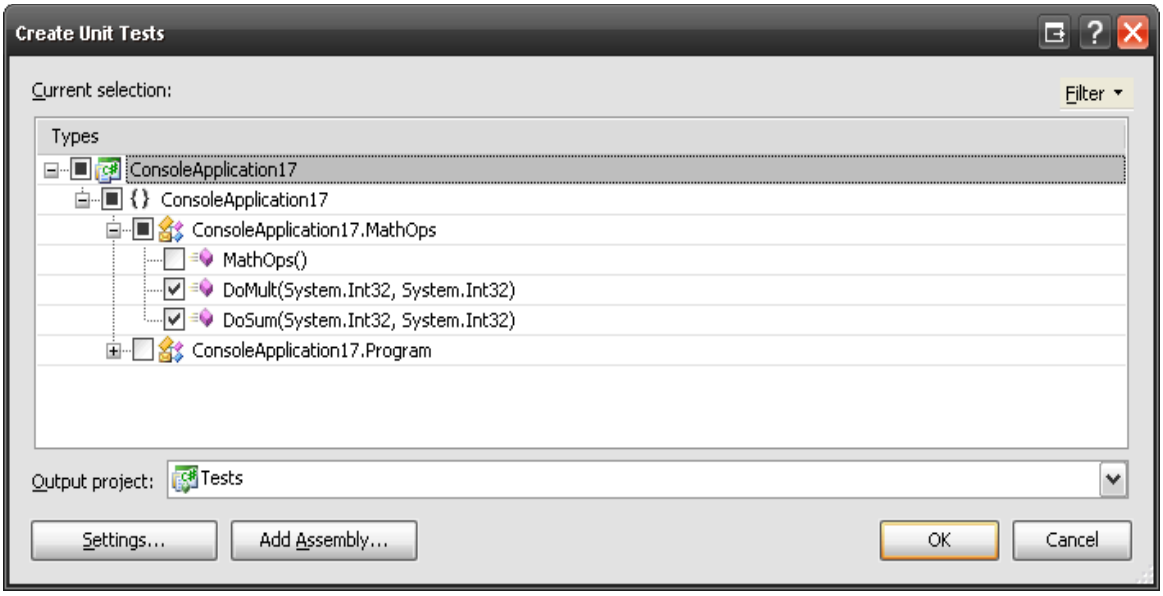


Рисунок 18.2 - Створення модульних тестів для класу

Після створення нового тесту у Visual Studio з’являється дана заготовка, яку в подальшому потрібно змінювати для коректної роботи модульного тесту (рис. 18.3).

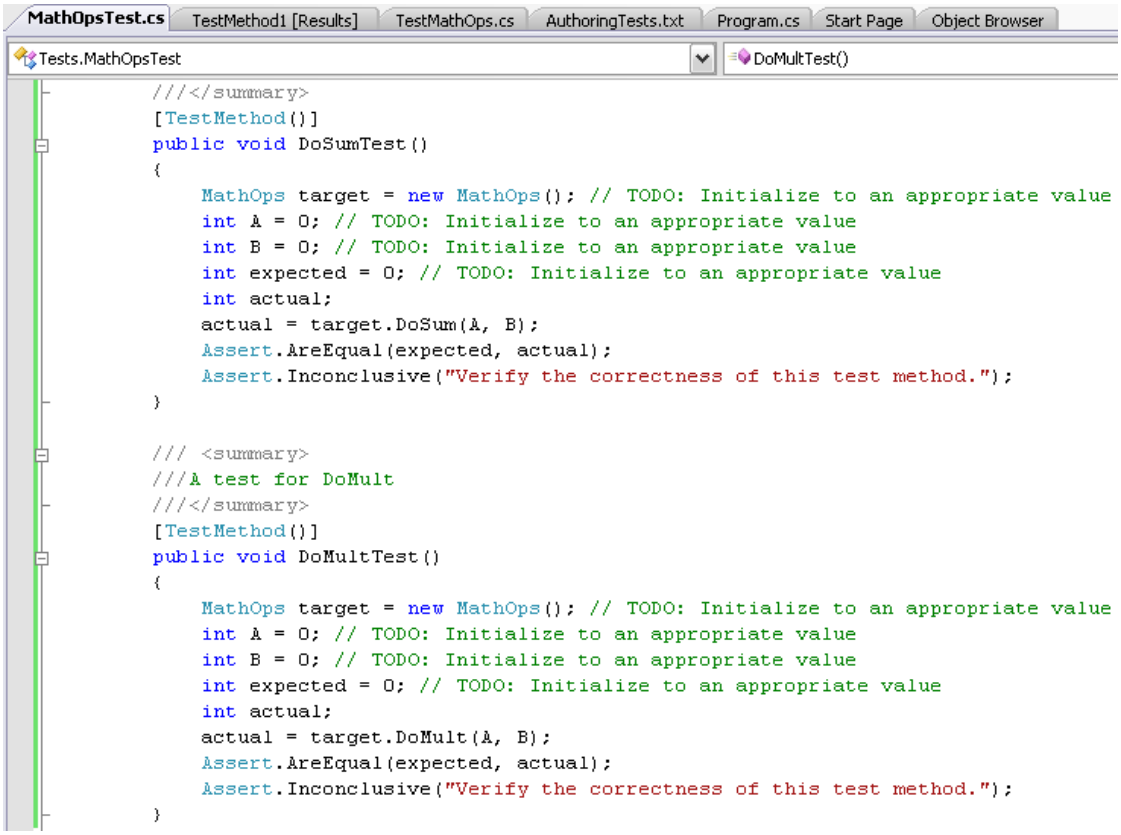


Рисунок 18.3 – Тіло модульного тесту

Тепер необхідно реалізувати методи у модульному тесті для його коректної роботи та перевірки правильності коду програми. Для заданого прикладу реалізація показана на рисунку 18.4.

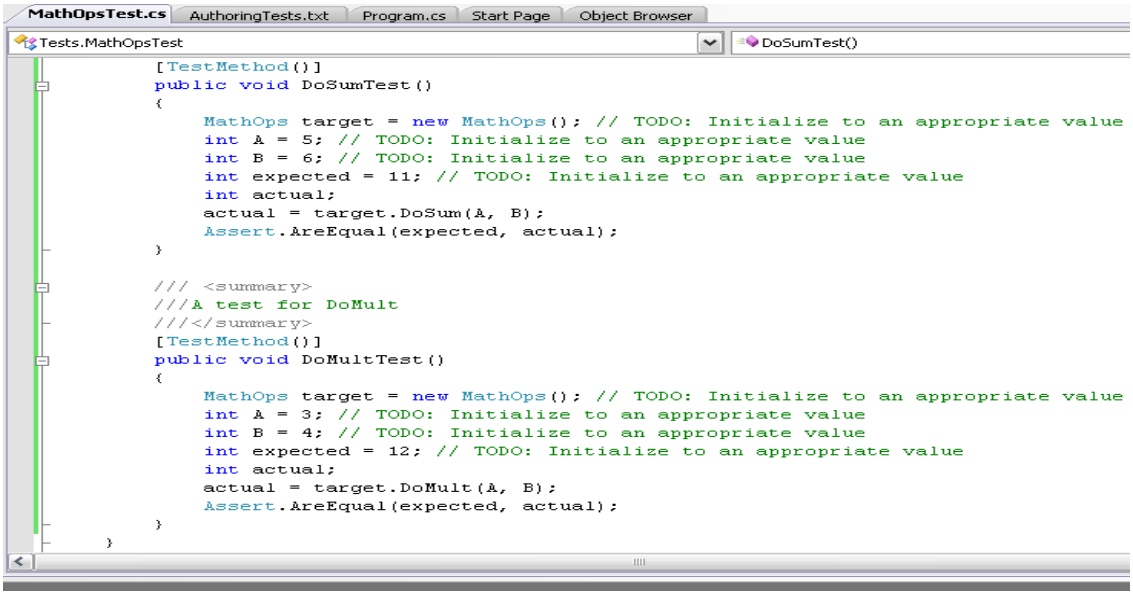


Рисунок 18.4 – Реалізація методів у тестуванні

Запустимо тест на виконання за допомогою головного меню та команди Test -> Run -> All Tests in Solution. Отримаємо такі результати внаслідок перевірки коду програми за допомогою модульного тестування (рис. 18.5).

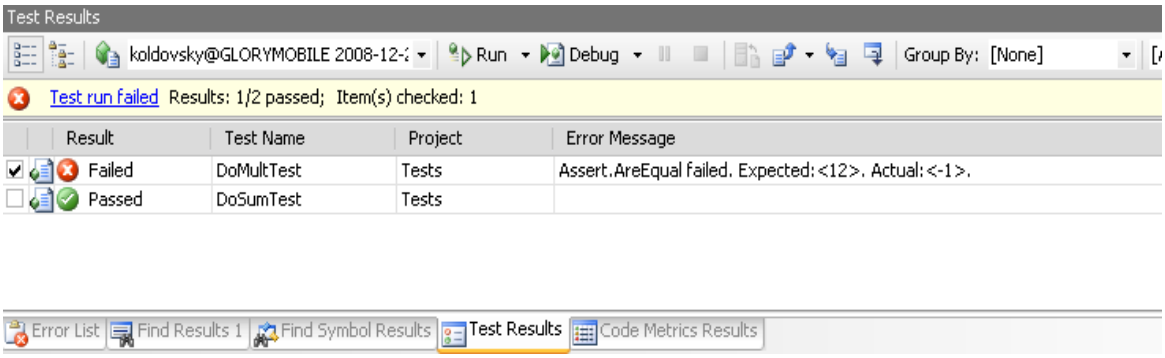


Рисунок 18.5 – Результати виконання тесту

4. Створення модульного тесту для перевірки списків

При створенні модульних тестів для перевірки списків слід перевіряти наступні операції:

- вставка;
- видалення;
- отримання елементу зі списку;
- розмір списку.

Після виконання кожної операції, яка призводить до зміни елементів у списку, необхідно перевіряти правильність елементів у списку. В залежності від конкретного виду списку (стек, черга і т.д.) необхідно перевіряти правильність виконання відповідних операцій.

5. Створення модульного тесту для перевірки алгоритму сортування

При перевірці алгоритмів сортування необхідно перевіряти як кінцевий результат роботи алгоритму, так і поетапне його виконання (клас сортування

слід спроектувати таким чином, щоб була можливість перевірити його покроково).

6. Введення до методів відладки комп’ютерних програм

Відладка – це процес пошуку і усунення помилок. Здійснюється за допомогою спеціального інструментарію (відладчика). Програміст має дуже добре знатися на методах відладки, оскільки за статистикою у середньому до 50% його робочого часу може йти на відладку програм. Типові можливості відладчика:

- можливість встановлювати точки переривання;
- трасування (покрокове виконання) програми;
- можливість переглядати та змінювати значення змінних;
- можливість виконувати фрагменти коду в контексті програми;
- віддалена відладка.

Але як зауваження, варто зазначити, що відладчик не може допомогти знайти помилку, яка викликана неправильним розумінням вимог до програми. Він призначений лише для виявлення і усунення таких помилок, які є результатом невірно написаного вихідного коду внаслідок допущених програмістом помилок при розумінні його поведінки. Лише в тому разі, якщо код написаний якісно, однак містить помилки, необхідно здійснити пошук і виправлення помилок за допомогою відгадчика.

7. Процедура здійснення пошуку та виправлення помилок

В процесі локалізації помилки необхідно на основі симптомів, у яких вона проявляється, сформувати гіпотезу про характер помилки (помилка в обчисленнях, помилка при роботі із класами, помилка при роботі з ресурсами і т.д.). Відповідно до сформульованої гіпотези слід визначити найбільш ймовірні джерела виникнення помилки (конкретні класи, методи), в яких слід поставити точки преривання та за необхідності здійснити преривання, перевірити значення змінних.

Спрощенню процедури відладки сприяють:

– модульні тести, які допомагають локалізувати джерело помилки конкретними класами і методами;
– дотримання правил іменування елементів програмного коду, що дозволяє спростити його розумінні;
– “захищене програмування” – набір практик, які дозволяють знизити ймовірність виникнення типових помилок.

8. Виявлення та усунення типових помилок

До основних найпоширеніших помилок у програмуванні можна віднести такі:

8.1. Значна втрата точності під час цілочисленого ділення.

Мова програмування C# під час ділення двох цілих чисел у результаті повертає також ціле число. Слід ставитися уважно до виконання арифметичних операцій, уникати використання літералів, заміняти їх константами дійсного типу.

8.2. Зплутані одиниці розмірності.

У обчисленнях, в яких приймають участь значення, що можуть вимірюватися у різних одиницях вимірювання, можна помилково використовувати одиниці розмірності без перетворення. До подібних обчислень слід ставитися з уважністю, одиниці розмірності коментувати та навіть вносити до назви змінної (sizeInMeters).

Головне правило при виявленні будь-яких помилок: для того, щоб мати можливість виявлення помилки важливо забезпечити її відтворення, тобто знайти чітку послідовність дій, яка приводить до їх появи. Інакше можна безрезультатно шукати помилку тривалий час тоді, коли вона не повторюється.

8.3. Залежні від регіональних параметрів значення дати та чисел.

В ОС Windows є можливість змінювати регіональні параметри системи, що впливає на формат чисел (зокрема, роздільник, який використовується

для відокремлення цілої та дрібної частини), формат дат (зокрема, порядок, в якому записується число, місяць, рік) та ін.

Програма повинна використовувати лише спеціальні типи даних для збереження відповідних даних, а також використовувати системні функції для перетворення між цими типами даних і рядком, а також навпаки.

Не можна зберігати числа, дати у рядкових змінних, полях БД, інакше це може слугувати джерелом для виникнення помилок, які складно виявляти і відлажувати (якщо програма виконується на комп'ютері з іншими регіональними параметрами).

8.4. Помилки, викликані непродуманими назвами класів, методів, змінних і інших програмних елементів.

Типовою складною для виявлення помилкою є ситуація, коли із-за неоднозначного трактування назви програмного елементу він використовується не за призначенням.

8.5. Не звільнення ресурсів.

Виникає в ситуації, коли використовується ресурс, який необхідно звільняти, однак це звільнення не відбувається. Кожне виділення ресурсу, який необхідно звільняти, слід заключати в блок try…finally (саме виділення йде безпосередньо перед try, а в блоці finally слід звільнити ресурс).

8.6. Приглушення виключної ситуації.

Відбувається коли обробник виключної ситуації "ловить" не тільки ті ситуації, для яких він створений, а і ті, які ним не передбачені. Слід уважно створювати обробники виключних ситуацій.

9. Введення до захищеного програмування

Пошук і усунення помилок – не найкращий спосіб витрачання робочого часу програміста, набагато краще взагалі їх не допускати.

Поряд із загальним підвищенням якості програмного коду тими методами, які розглядалися раніше, існує також спеціальний підхід, головна

мета якого полягає у тому, щоб скоротити кількість помилок з вини програміста до мінімуму – захищене програмування.

Існує значна кількість рекомендацій, які відносяться до захищеного програмування, дуже часто вони є специфічними для середовища розробки, яке використовується, однак існують також і загальні рекомендації. Серед них виділяють такі:

- необхідно перевіряти правильність вхідних даних, навіть якщо це не вимагається функціональними вимогами;
- усі елементи програмних конструкцій повинні мати назви, які відповідають їх змісту;
- будь-яке динамічне виділення ресурсу має закінчуватися обов’язковим його вивільненням у секції фіналізації, що має спрацювати навіть за умови появи виключних ситуацій після виділення ресурсу;
- кожна змінна має бути використана лише з тією метою, для якої вона була задекларованe, повторне використання змінних для цілей, не передбачених в момент їх декларації, не допускається;
- програмний код не має містити жодних числових чи строкових значень – вони мають бути представлені у вигляді констант;
- повідомлення про помилки мають бути осмисленими і чітко вказувати на причину їх появи;
- при написанні програм не можна покладатися на особливості платформи, для якої вони розробляються, зокрема, розрядність типів даних та ін.;
- модульні тести мають покривати максимум програмного коду, якщо код не підлягає тестуванню за допомогою модульних тестів, то його слід перепроектувати таким чином, щоб обсяг подібного коду був мінімальним;
- усі повторювальні блоки програмного коду мають виділятися у повторно використовувані функції, методи, класи, модулі, бібліотеки;

- складний для розуміння код має бути спрощений, розбитий на окремі, більш логічні складові та документований;
- великі за обсягом класи, методи, модулі та ін. мають бути розділені на більш дрібні складові;
- програмні конструкції які не використовуються, мають бути видалені;
- код не має містити ніяких припущень – усі подібні ситуації необхідно документувати;
- код повинен підтримувати можливість відновлювати роботу після виникнення некритичних помилок і припиняти свою роботу та чітко інформувати про появу критичних помилок;
- програмний код не має подавляти інформацію про помилки, які виникають, у тому числі заміняти конкретну інформацію більш загальною.

10.Основи профілювання програм

На відміну від перших комп’ютерів, сучасні ЕОМ мають надзвичайно великі обчислювальні можливості і обсяги пам’яті. Тому програміст при створенні програми повинен насамперед хвилюватися про її якість і відповідність поставленому завданню, а лише потім – про її оптимізацію з метою збільшення швидкості функціонування.

Звичайно, існує особлива група комп’ютерних програм, для яких швидкість виконання є головною вимого, але у більшості випадків оптимізація по швидкості виноситься на окремий етап розробки і здійснюється над уже працюючим кодом, коректність якого була перевірена відповідними тестами.

Важливо: сучасні компілятори і середовища виконання комп’ютерних програм є надзвичайно складними і володіють власними механізмами оптимізації. Якщо програміст не володіє детальною інформацію про особливості їх роботи, то його спроби «оптимізувати» код можуть призвести до зворотного результату: наприклад, якщо програміст напише частину коду

на асемблері, то оптимізуючий компілятор не буде мати можливості здійснити високорівневий аналіз коду і оптимізувати його.

При оптимізації програм необхідно приймати до уваги наступне емпіричне правило: оптимізація 10 відсотків програмного коду може призвести до 90 відсотків потенціального можливого поліпшення швидкості виконання програми. Дуже часто це співвідношення є ще більш диспропорційним.

Перш ніж переходити до спроб оптимізації коду необхідно здійснити процес профілювання програми – виявлення швидкості роботи її окремих ділянок. Здійснювати цей процес можна як і за рахунок самостійного вимірювання тривалості виконання ділянок програмного коду, розмістивши у коді виклики методів, що здійснюють заміри швидкості, так і за допомогою спеціального інструментарію – профайлерів. Важливо знати: те, що не можна вимірювати (навіть приблизно), не можна оптимізовувати.

Середовище Visual Studio 2008 містить вбудований профайлер, який можна використовувати для аналізу продуктивності (рис. 18.6):

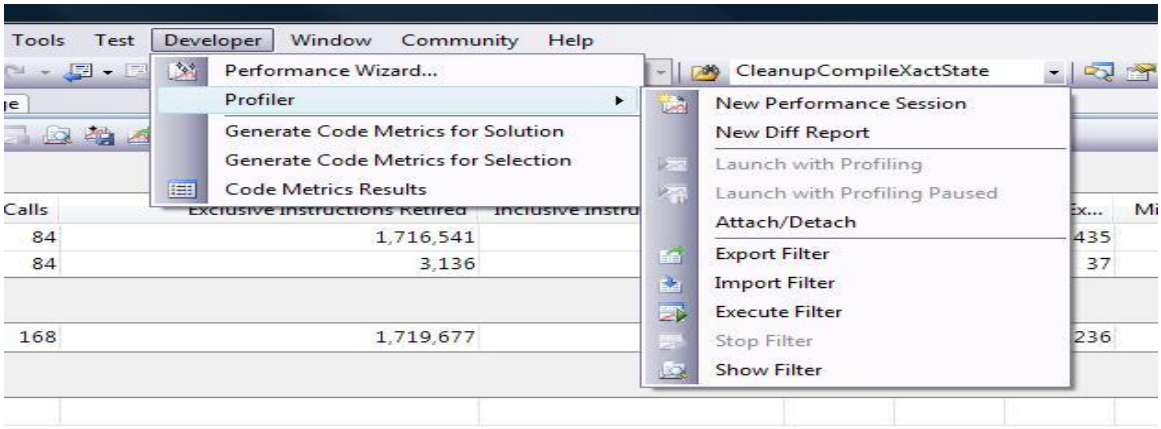


Рисунок 18.6 - Профайлер Visual Studio 2008

11. Виявлення вузьких місць та оптимізація програмного коду деяких типових алгоритмів.

При оптимізації алгоритмів слід звертати увагу на операції, які повторюються. Цикли – основний об’єкт оптимізації.
У процесі профілювання і оптимізації особливу увагу слід приділяти циклам, оскільки саме вони у абсолютній більшості випадків займають найбільше часу при виконанні, а також добре піддаються оптимізації.

ЛК.19 – СТВОРЕННЯ ДОКУМЕНТАЦІЇ ТА ДИСТРИБУТИВУ ПРОГРАМНОГО ПРОДУКТУ

Перелік питань

1. Основи створення закінченого програмного продукту.
2. Документація і довідкова система до програмного продукту.
3. Документування вихідного коду.
4. Побудова довідкової системи.
5. Інтегровані інструменти генерації документації і побудови довідкової системи.
6. Технології розповсюдження програмного продукту.
7. Створення дистрибутиву програмного продукту.
8. Створення документації до програмного продукту.

На самостійне вивчення:

1. Засоби візуального проектування та тестування класів, інтегрована довідкова система Visual Studio [1, С.106-110].

1. Основи створення закінченого програмного продукту

Закінчений програмний продукт – це значно більше, ніж просто скомпільований програмний код. Закінчений програмний продукт, крім власне комп’ютерної програми, повинен включати також, як мінімум, документацію та дистрибутив.

Необхідність доповнення комп’ютерної програми документацією і дистрибутивом пояснюється тим, що сучасні програмні продукти розглядаються як ринкові продукти, які пропонуються в умовах вільної конкуренції, а тому мають пропонувати максимальний рівень задоволення потреб потенційних користувачів. Навіть якщо програма володіє інтуїтивно зрозумілим інтерфейсом та не потребує інсталяції, наявність документації та

дистрибутиву є обов’язковою умовою для того, щоб програма розглядалася як серйозний продукт.

2. Документація і довідкова система до програмного продукту

Документація до програмного продукту представляється, як правило, у друкованому вигляді, або в електронному, однак з можливістю виведення на друк (у форматах .PDF, .DOC, .RTF). Документація являє собою посібник по освоєнню програмного продукту, з яким користувач має ознайомитись перш ніж приступити до його використання, а також може звертатися у тому разі, якщо у нього виникнуть складнощі при використанні програмного продукту.

Довідкова система – це електронний посібник, до якого користувач може звертатися під час роботи з програмним продуктом для того, щоб зрозуміти, як користуватися його окремими функціями. Довідкова система має підтримувати контекстний виклик – тобто викликатися відповідно до функціонального модуля, з яким працює користувач. Виклик здійснюється за рахунок натискання клавіші загальноприйнятої у Windows клавіші «F1». Довідкова система створюється у форматах, які підтримують контекстний виклик - .CHM, .HTML, .HLP.

3. Документування вихідного коду

Документування вихідного коду – обов’язкова складова процесу розробки програмного забезпечення. До документування вихідного коду висуваються наступні вимоги:

- програмний код має бути максимально зрозумілим без додаткових пояснень, тому, якщо це можливо, то документування краще замінити поліпшенням коду;
- документуванню підлягають найбільш складні ділянки коду, прості і зрозумілі фрагменти не слід документувати.

4. Побудова довідкової системи

Стандартним для застарілих операційних систем Windows, починаючи з версії Windows 3.0 і закінчуючи Windows ME, був формат .HLP. Створення довідкової системи для цього формату здійснювалося за допомогою програмного продукту Microsoft Help Workshop.

Однак для усіх сучасних версій ОС Windows, починаючи з Windows 2000 і закінчуючи Windows VISTA, стандартним форматом побудови довідкової системи є формат .CHM. Побудова довідкової системи у форматі .CHM здійснюється за допомогою програмного продукту Microsoft HTML Help Workshop. Програмний продукт Microsoft HTML Help Workshop вільно розповсюджується корпорацією Microsoft і може бути завантажений із сайту корпорації.

Послідовність побудови довідкової системи за допомогою Microsoft HTML Help Workshop має виглядати таким чином:

- створити файл проекту довідкової системи;
- задати параметри форматування та шрифти;
- задати заголовок;
- створити файл теми;
- створити файли вкладених тем;
- наповнити файли з темами інформацією;
- створити зміст;
- створити індекс;
- скомпілювати проект довідкової системи;
- пов’язати файл довідкової системи із комп’ютерною програмою.

5. Інтегровані інструменти генерації документації і побудови довідкової системи.

Зважаючи на те, що текст документації і довідкової системи у значній мірі може збігатися, крім того, їх підготовкою, як правило, займаються одні і

ті ж співробітники, для спрощення побудови довідкової системи та створення документації використовуються так звані інтегровані системи генерації документації та побудови довідкової системи. Наглядна схема до пояснення демонструється на рисунку 19.1.

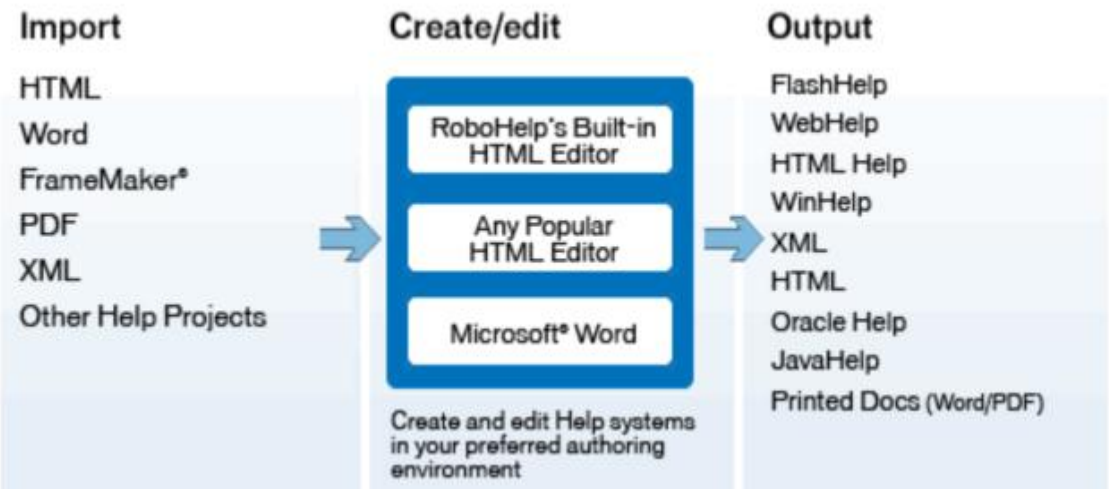


Рисунок 19.1 - Інструменти генерації документації

До інтегрованих системи генерації документації та побудови довідкової системи відносяться такі програмні продукти як RoboHelp Office (www.adobe.com/products/robohelp), HelpScrible (www.helpscribble.com), FastHelp (www.fast-help.com), приклад яких зображено на рисунку 19.2.

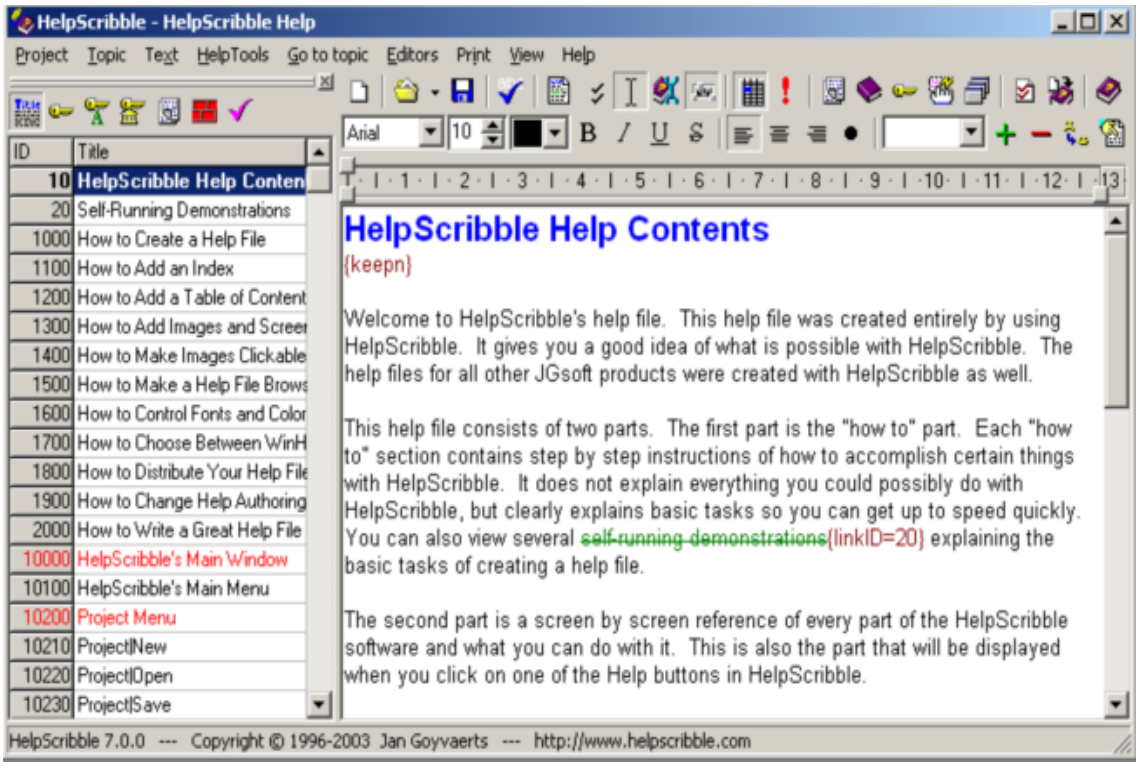


Рисунок 19.2 – Приклад інтегрованих системи генерації документації та побудови довідкової системи

6. Технології розповсюдження програмного продукту

Сучасне програмне забезпечення може бути достатньо складним, включати різні набори компонентів, призначені для вирішення різних задач, чи орієнтовані на використання під управлінням різних платформ. Дуже часто програмне забезпечення вимагає наявність пререквизитів – попередньо встановлених у системі компонентів. Крім того, програмне забезпечення корпоративного призначення може вимагати наявності заздалегідь визначеної і налагодженої певним чином інфраструктури.

Таким чином, для розповсюдження програмного продукту недостатньо просто скопіювати його файли – необхідно виконати ще значну кількість дій, які виконуються спеціально для цього призначеними програмами – інсталяторами, а підготовлений для розповсюдження таким чином програмний продукт має назву дистрибутиву.

Крім власне виконання необхідних дій по установці програмного продукту дистрибутиви необхідні для того, щоб забезпечити можливість встановлення програмного забезпечення за допомогою централізованих інструментів, таких як групові політики Active Directory.

7. Створення дистрибутиву програмного продукту

Visual Studio має вбудовані можливості для створення дистрибутиву програмного продукту (рис. 19.3).

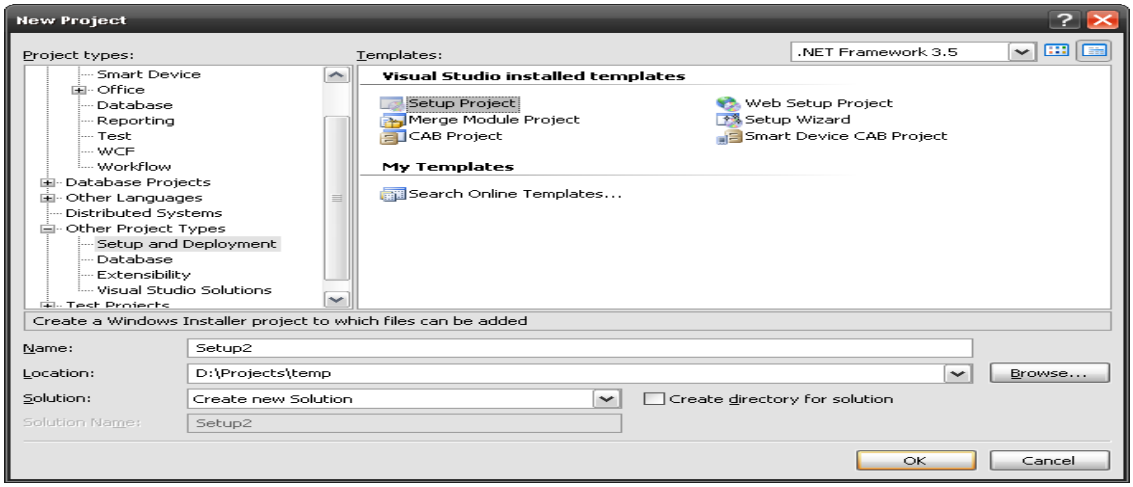


Рисунок 19.3 – Приклад створення дистрибутиву програмного продукту

Розглянемо процедуру створення дистрибутиву програмного продукту за допомогою вільно доступного інструментарію WIX (Windows Installer XML, wix.sourceforge.net).

Проект WIX був започаткований корпорацією Microsoft і є першим продуктом корпорації, опублікованим на умовах вільного коду. WIX дозволяє на основі вихідних XML-файлів згенерувати пакет установки для інсталятора Windows Installer (.MSI), який є стандартним для сучасних операційних систем Microsoft Windows і підтримує можливості для встановлення програмного забезпечення за допомогою групових політик Active Directory.

Для створення проекту дистрибутиву за допомогою WIX необхідно створити XML-файл, який буде містити інформацію про файли проекту, що

необхідно включити в дистрибутив, а також опис необхідних дій та пререквизитів, потрібних для установки.

Створення дистрибутиву програмного продукту з WIX демонструється на рисунку 19.4.

```
<?xml version="1.0" encoding="UTF-8"?>
<Wix xmlns="http://schemas.microsoft.com/wix/2003/01/wi">
  <Product Id="PUT-GUID-HERE" Name="Your Product" Language="1033"
           Version="0.0.0.0" Manufacturer="Your Company">
    <Package Id="????????-????-????-????-????????????"
             Description="Description of your product"
             Comments="This will appear in the file summary stream."
             InstallerVersion="200" Compressed="yes" />

    <Media Id="1" Cabinet="Product.cab" EmbedCab="yes" />

    <Directory Id="TARGETDIR" Name="SourceDir">
      <Directory Id="ProgramFilesFolder">
        <Directory Id="INSTALLLOCATION" Name="MyAppDir"
                   LongName="My Application Directory">

          <Component Id="ProductComponent" Guid="PUT-GUID-HERE">
            <!-- TODO: Insert your files, registry keys, and other
            resources here. -->
          </Component>

        </Directory>
      </Directory>
    </Directory>

    <Feature Id="ProductFeature" Title="Feature Title" Level="1">
      <ComponentRef Id="ProductComponent" />
    </Feature>
  </Product>
</Wix>
```

Рисунок 19.4 - Шаблон XML-файлу з описом програмного продукту для створення дистрибутиву за допомогою WIX

Розглянемо приклад створення найпростішого дистрибутиву. Припустимо, що у нас є програмний продукт з назвою «TestProduct», який складається з одного файлу – «TestProduct.exe» і розміщений у папці «C:\TestProduct\».

Встановлений продукт має бути у стандартну папку для програмних продуктів операційної системи «Program Files» і вкладену папку із назвою продукту. Даний дистрибутив показано на рисунку 19.5, а опис представлений на рисунку 19.6.

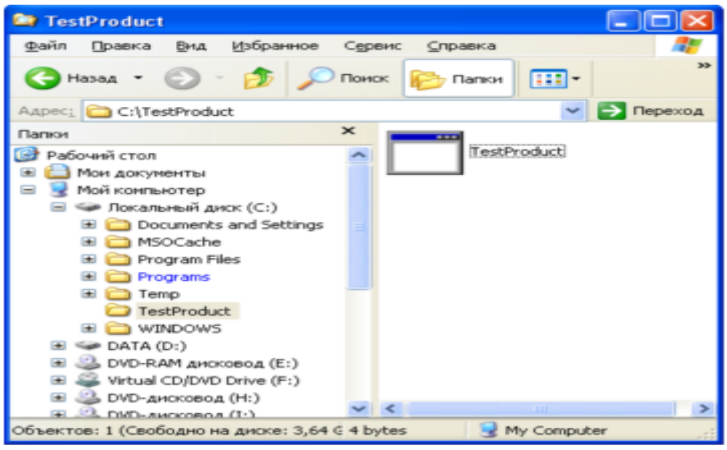


Рисунок 19.5 - Приклад створення найпростішого дистрибутиву за допомогою WIX

```
<?xml version='1.0'?>
<Wix xmlns='http://schemas.microsoft.com/wix/2003/01/wi'>
  <Product Id='925E5742-30D7-44C8-8C14-B8757B54BEC1'
           Name='TestProduct' Language='1033'
           Version='0.0.0.0' Manufacturer='Microsoft Corporation'>
    <Package Id='D78F5C66-8BB0-408f-91B4-CF9DBCC7E773'
             Description='Test Product'
             Comments='Test Product'
             InstallerVersion='200' Compressed='yes' />

    <Media Id='1' Cabinet='product.cab' EmbedCab='yes' />

    <Directory Id='TARGETDIR' Name='SourceDir'>
      <Directory Id='ProgramFilesFolder' Name='PFiles'>
        <Directory Id='TestProductDirectory' Name='testprod'
                   LongName='Test Assembly'>
          <Component Id='TestProductComponent'
                     Guid='8864F039-1F54-4A1F-83CD-3C67FD7448F6'>
            <File Id='TestAssemblyProductFile' Name='TestProd.exe'
                  KeyPath='yes' DiskId='1'
                  src='c:\TestProduct\TestProduct.exe'/>
          </Component>
        </Directory>
      </Directory>
    </Directory>

    <Feature Id='TestProductFeature' Title='Test Product Feature' Level='1'>
      <ComponentRef Id='TestProductComponent'/>
    </Feature>
  </Product>
</Wix>
```

Рисунок 19.6 - Опис дистрибутиву

Значення ідентифікаторів («Product Id», «Package Id» та «Guid») необхідно заповнити глобальними унікальними ідентифікаторами – GUID (у Delphi їх можна згенерувати автоматично за рахунок клавіатурного скорочення Ctrl-Shift-G). При зазначені атрибуту «Name» необхідно використовувати формат «8.3» (ім’я не більше восьми символів і розширення через крапку не більше трьох).

Тег «Feature» визначає набір компонентів, які необхідно встановити. Компоненти визначаються тегом «Component», він описується вкладеним у структуру папок, до яких має бути здійснена установка файлів компоненту.

Для того, щоб більш детально розібратися із структурою і синтаксисом XML-файлів опису дистрибутиву необхідно ознайомитися із документацією WIX. Для створення MSI-дистрибутиву необхідно скомпілювати вихідних XML-файл за допомогою програми candle.exe, а потім отриманий об’єктний файл передати у вигляді параметру до генератора MSI-файлів light.exe.

8. Створення документації до програмного продукту.

Для створення документації до програмного продукту у стилі MSDN можна використати проект SandCastle (http://www.codeplex.com/Sandcastle), як показано на рисунку 19.7.

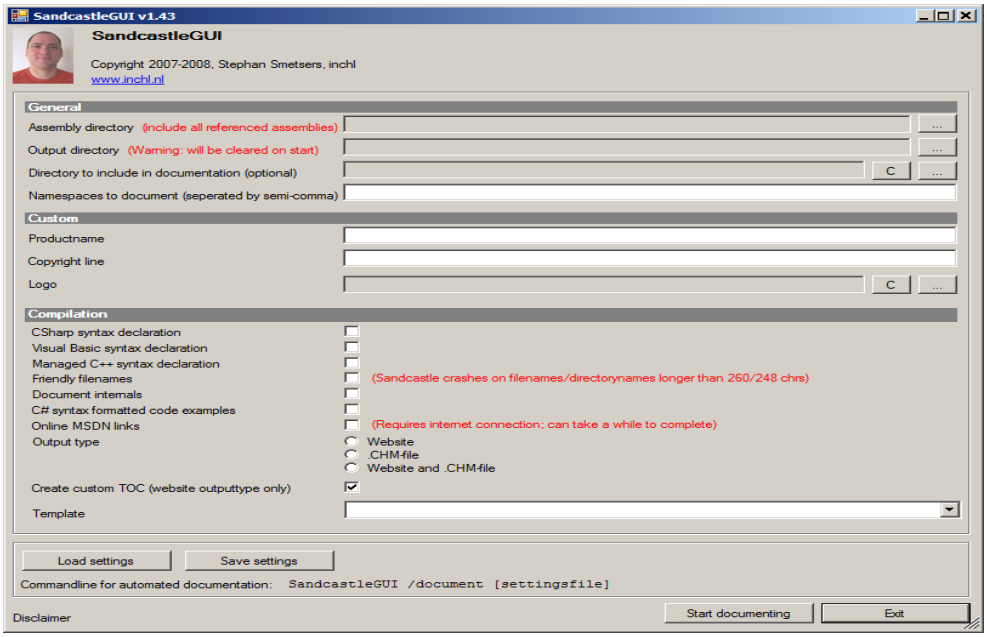


Рисунок 19.7 – Створення документації до програмного продукту

СПИСОК ЛІТЕРАТУРИ

1. Троелсен Э. Язык программирования C# 2005 и платформа .NET 2.0, 3-е изд.: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2007. – 1168 с.
2. Дейтел Х. C# в подлиннике: Пер. с англ. / Дейтел Х., Дейтел П., Листфилд. Дж., Нието Т., Йегер Ш., Златкина М. – СПб.: БХВ-Петербург, 2006. – 1056 с.
3. Шилдт Г. C#: учебный курс. – СПб.: Питер; К.: Издательская группа BHV, 2003. – 512 с.

Колдовський, В. В. Економічна кібернетика. Програмування [Текст] : конспект лекцій / В. В. Колдовський. - Суми : ДВНЗ "УАБС НБУ", 2008 - 308 с.