Л. М. Лутай

ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ ДЛЯ ВИРОБНИЧИХ ПРОЦЕСІВ

2021

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»

Л. М. Лутай

ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ ДЛЯ ВИРОБНИЧИХ ПРОЦЕСІВ

Конспект лекцій

Харків «ХАІ» 2021

УДК 004.896:004.43 (075.8)

Л86

Рецензенти: канд. техн. наук, доц. О. Ю. Лиходєєв,
канд. техн. наук, доц. Д. О. Пшеничников

Лутай, Л. М.

Л86 Проектування інформаційних систем для виробничих процесів [Електронний ресурс] : консп. лекцій / Л. М. Лутай. – Харків : Нац. аерокосм. ун-т ім. М. Є. Жуковського «Харків. авіац. ін-т», 2021. – 124 с.

Описано процес проектування інформаційних систем за допомогою об’єктно-орієнтованого підходу.

Для студентів третього курсу, які навчаються за спеціальністю «Автоматизація та комп’ютерно-інтегровані технології». Може бути корисним усім тим, хто хоче навчитися швидко створювати інформаційні системи, до складу яких входять бази даних.

Іл. 25. Табл. 1. Бібліогр.: 7 назв

УДК 004.896:004.43 (075.8)

Лекція 1. ПОНЯТТЯ КЛАСУ. СТРУКТУРНІ ЕЛЕМЕНТИ КЛАСУ

Клас – це абстрактний тип даних. Тобто, клас – це деякий шаблон, на основі якого будуть створюватися його екземпляри – об'єкти. Описом об'єкта є клас, а об'єкт являє собою екземпляр цього класу. Можна ще провести таку аналогію. У нас у всіх є деяке уявлення про людину – наявність двох рук, двох ніг, голови, травної, нервової системи, головного мозку і т. д. Є деякий шаблон – цей шаблон можна назвати класом. Реально ж існуюча людина (фактично екземпляр класу) є об'єктом цього класу.

Загальний опис класу:

[модифікатор доступу] class [ім'я_класу]

{// тіло класу}

Модифікаторів доступу до класів є два:

- public – доступ до класу можливий з будь-якого місця одного збирання або з іншого збирання, на яке є посилання;
- internal – доступ до класу можливий тільки зі збирання, в якому він оголошений.

Збирання (assembly) – це готовий функціональний модуль у вигляді exe або dll-файлу (файлів), який містить скомпільований код для .NET. Збирання надає можливість повторного використання коду.

При оголошенні класу модифікатор доступу можна не вказувати, при цьому буде застосовуватися режим за замовчуванням internal.

Складові елементи класу подано на рис. 1.

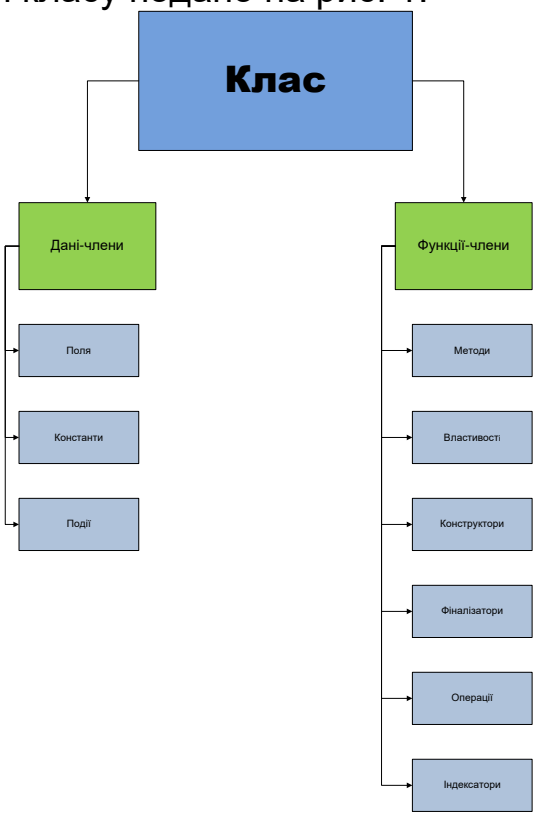

Рис. 1. Складові елементи класу

Дані-члени – це ті члени, які містять дані класу – поля, константи, події. Дані-члени можуть бути статичними (static). Член класу є членом екземпляру, якщо тільки він не оголошений явно як static. Давайте розглянемо види цих даних.

Поля (field) – це змінна, оголошена всередині класу. Як правило, поля оголошуються з модифікаторами доступу private або protected, щоб заборонити прямий доступ до них. Для отримання доступу до полів слід використовувати властивості або методи.

Приклад оголошення полів у класі:

```
class Student
{ private string firstName;
private string lastName;
private int age;
public string group; // не рекомендується використовувати public
для поля
}
```

Константи можуть бути асоційовані з класом тим же способом, що і змінні. Константа оголошується за допомогою ключового слова const. Якщо вона оголошена як public, то в цьому випадку стає доступною ззовні класу.

Події – це члени класу, що дозволяють об'єкту повідомляти про події або про деяку взаємодію з користувачем. Клієнт може мати код, відомий як обробник подій, що реагує на них.

Функції-члени – це члени, які забезпечують деяку функціональність для маніпулювання даними класу. Вони містять методи, властивості, конструктори, фіналізатор, операції й індексатори.

Методи (method) – це функції, асоційовані з певним класом. Як і дані-члени, за замовчуванням вони є членами примірника. Вони можуть бути оголошені статичними за допомогою модифікатора static.

Властивості (property) – це набори функцій, які можуть бути доступні клієнту таким же способом, як і загальнодоступні поля класу. У C# передбачений спеціальний синтаксис для реалізації читання і запису властивостей для класів, тому писати власні методи, що починаються на set і get, не знадобиться. Оскільки не існує якогось окремого синтаксису для властивостей, який відрізняв би їх від нормальних функцій, створюється ілюзія об'єктів як реальних сутностей, що надаються клієнтському коду.

Конструктори (constructor) – це спеціальні функції, що викликаються автоматично при ініціалізації об'єкта. Їх імена збігаються з іменами класів, яким вони належать, і вони не мають типу повернення. Конструктори корисні для ініціалізації полів класу.

Фіналізатори (finalizer) викликаються, коли середовище CLR визначає, що об'єкт більше не потрібен. Вони мають те ж ім'я, що і клас, але з попереднім символом тильди. Передбачити точно, коли буде викликаний фіналізатор, неможливо.

Операції (operator) – це найпростіші дії на кшталт + або -. Коли ви складаєте два цілих числа, то, строго кажучи, застосовуєте операцію + до цілих. Однак C# дозволяє вказати, як існуючі операції будуть працювати з одними класами (так зване перевантаження операції).

Індексатори (indexer) – дозволяють індексувати об'єкти таким же способом, як масив або колекцію.

Створення класу

Клас створюється за допомогою ключового слова class. Нижче наведена загальна форма визначення простого класу, що містить тільки змінні екземпляра і методи:

```
class ім'я_класу {
// Оголошення змінних екземпляра.
доступ тип змінна1;
доступ тип змінна2;
// ...
доступ тип змінnaN;

// Оголошення методів.
доступ тип_повернення метод1 (параметри) {
// тіло методу
}
доступ тип_повернення метод 2 (параметри) {
// тіло методу
}
//. . .
доступ тип_повернення методN (параметри) {
// тіло методу
}
}
```

Зверніть увагу на те, що перед кожним оголошенням змінної і методу вказується доступ. Це специфікатор доступу, наприклад public, що визначає порядок доступу до конкретного члена класу. Члени класу можуть бути як закритими (private) в межах класу, так і відкритими (public), тобто доступнішими. Специфікатор доступу визначає тип дозволеного доступу. Вказувати специфікатор доступу не обов'язково, але якщо він відсутній, то оголошений член вважається закритим у межах класу. Члени з закритим доступом можуть використовуватися тільки іншими членами їх класу.

```
class Book
{
```

```
    public string name;
    public string author;
    public int year;

  public Book() // public Book() {} альтернативний варіант
    {
      name = "невідомий";
      author = "невідомий";
      year = 0;
    }

    public Book(string name, string author, int year)
    {
      this.name = name;
      this.author = author;
      this.year = year;
    }

    public void Info()
    {
      Console.WriteLine("Книга '{0}' (автор {1}) була видана у {2} році", name, author, year);
    }
  }
```

Конструктор

Одне з призначень конструктора – початкова ініціалізація членів класу. Оскільки імена параметрів і імена полів (name, author, year) у цьому випадку збігаються, то використовуємо ключове слово this. Це ключове слово являє собою посилання на поточний екземпляр класу. Тому у виразі this.name = name; перша частина this.name означає, що name – це поле поточного класу, а не назву параметра name. Якби параметри і поля називалися по-різному, то використовувати слово this було б необов'язково. Слово this можна використовувати для нестатичних методів [1].

```
  class Program // Program.cs
  {
    static void Main(string[] args)
    {
      Book b1 = new Book("Таємний посол", "В. Малик", 1969);
      b1. Info();

      Book b2 = new Book();
      b2. Info();
```

```
            Console.ReadLine();
        }
    }
```

Інші способи ініціалізації:

```
/*1*/ Book b1 = new Book();
b1.name = "Таємний посол";
b1.author = "В. Малик";
b1.year = 1969;

b1.Info();

/*2*/Book b2 = new Book();
b2 = new Book { name = "Отцы и дети", author = "И.  Тургенев", year = 1862 };
b2.Info();
```

Конструктор ініціалізує об'єкт при його створенні. У конструктора таке ж ім'я, як і у його класу, а з точки зору синтаксису він подібний до методу. Але у конструкторів немає типу повернення. Нижче наведена загальна форма конструктора:

```
доступ ім'я_класу (список_параметрів) {
// тіло конструктора
}
```

Як правило, конструктор використовується для задавання початкових значень змінних екземпляра, визначених у класі, або ж для виконання будь-яких інших настановних процедур, які потрібні для створення повністю сформованого об'єкта. Крім того, доступ зазвичай являє собою модифікатор доступу типу public, оскільки конструктори часто викликаються в класі. А список_параметрів може бути порожнім або складатися з одного або більше згаданих параметрів.

Кожен клас C# забезпечується конструктором за замовчуванням, який при необхідності може бути перевизначений. За визначенням такий конструктор ніколи не приймає аргументів. Після розміщення нового об'єкта в пам'яті конструктор за замовчуванням гарантує заповнення всіх полів стандартними значеннями. Якщо програміст не задоволений такими присвоюваннями за замовчуванням, він може перевизначити конструктор за замовчуванням відповідно до своїх потреб.

Конструктор також може приймати один або кілька параметрів. У конструктор параметри вводяться таким же чином, як і в метод. Для цього достатньо оголосити їх у дужках після імені конструктора.

Давайте розглянемо застосування конструкторів на прикладі:

```
using System;
using System.Collections.Generic;
```

```
using System.Linq;
using System.Text;

namespace ConsoleApplication1
{
   class MyClass
   {
      public string Name;
      public byte Age;

      // Створюємо конструктор з параметрами
      public MyClass(string s, byte b)
      {
         Name = s;
         Age = b;
      }

      public void reWrite()
      {
         Console.WriteLine("Ім’я : {0}\nВік: {1}", Name, Age);
      }
   }

   class Program
   {
      static void Main(string[] args)
      {
         MyClass ex = new MyClass("Liudmyla", 26);
         ex.reWrite();

         Console.ReadLine();
      }
   }
}
```

Полями класу називаються звичайні змінні рівня класу. Раніше було розглянуто змінні – їх оголошення та ініціалізацію. Однак деякі моменти ще не розглядали, наприклад, константи і поля для читання.

Лекція 2. КЛАСИ. ЇХ СКЛАДОВІ. ОСОБЛИВОСТІ ПОСИЛАЛЬНИХ ТИПІВ ДАНИХ

Часткові класи

Часткові класи (partial class) надають можливість розділити функціонал одного класу на кілька файлів. Наприклад, зараз у нас код класу Book знаходиться в одному файлі Book.cs. Але можемо розділити його на кілька різних файлів. Для цього слід поставити перед визначенням класу ключове слово partial. Припустимо, в одному файлі буде:

```
partial class Book
{
    public string name;
    public string author;
    public int year;
}
```

А в іншому файлі:

```
partial class Book
{
    public Book(string name, string author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }

    public void GetInformation()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) була видана у {2} році", name, author, year);
    }
}
```

Усі члени класу – поля, методи, властивості – мають модифікатори доступу, що дозволяють задати допустиму область видимості для членів класу. Тобто контекст, у якому можна використовувати змінну або метод.

У C# застосовуються такі модифікатори доступу:

• public: публічний, загальнодоступний клас або член класу. Такий член класу доступний з будь-якого місця в коді, а також з інших програм і збирань;

• private: закритий клас або член класу. Являє собою повну протилежність модифікатору public. Такий закритий клас або член класу доступний тільки з коду в тому ж класі або контексті;

• protected: такий член класу доступний з будь-якого місця в поточному класі або в похідних класах;

• internal: клас і члени класу з подібним модифікатором доступні з будь-якого місця коду в тому ж збиранні, проте недоступні для інших програм і збирань (як у випадку з модифікатором public);

• protected internal: поєднує функціонал двох модифікаторів. Класи і члени класу з таким модифікатором доступні з поточного збирання і з похідних класів.

Оголошення полів класу без модифікатора доступу рівнозначно їх оголошенню з модифікатором private. Класи, оголошені без модифікатора, за замовчуванням мають доступ internal.

Розглянемо приклад класу

```
public class State
{
    int a; // все одно, що private int a;
    private int b; // поле доступно тільки з поточного класу
    protected int c; // доступно з поточного класу і похідних класів
    internal int d; // доступно в будь-якому місці програми
    protected internal int e; // доступно в будь-якому місці програми і з класів-спадкоємців
    public int f; // доступно в будь-якому місці програми, а також для інших програм і збірок.
    private void Display_f()
    {
        Console.WriteLine("Змінна f = {0}", f);
    }
    public void Display_a()
    {
        Console.WriteLine("Змінна a = {0}", a);
    }
    internal void Display_b()
    {
        Console.WriteLine("Змінна b = {0}", b);
    }
    protected void Display_e()
    {
        Console.WriteLine("Змінна e = {0}", e);
    }
}
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State();
        state1.a = 4; // Помилка, отримати доступ не можна
        // те ж саме відноситься і до змінної b
```

```
            state1.b = 3; // Помилка, отримати доступ не можна
            // присвоїти значення змінної «c» також не вийде,
            // оскільки клас Program не є класом-спадкоємцем класу State
            state1.c = 1; // Помилка, отримати доступ не можна
            // змінна d з модифікатором internal доступна з будь-якого місця програми
            // тому спокійно присвоюємо їй значення
            state1.d = 5;
            // змінна e так само доступна з будь-якого місця програми
            state1.e = 8;
            // змінна f загальнодоступна
            state1.f = 8;
            // Спробуємо вивести значення змінних
            // Цей метод оголошений як private, тому ми можемо використовувати його тільки всередині класу State
            state1.Display_f (); // Помилка, отримати доступ не можна
            // тому що цей метод оголошений як protected, а клас Program не є спадкоємцем класу State
            state1.Display_e (); // Помилка, отримати доступ не можна
            // Загальнодоступний метод
            state1.Display_a ();
            // Метод доступний з будь-якого місця програми
            state1.Display_b ();
            Console.ReadLine ();
        }
    }
```

Модифікатори public і internal схожі за своєю дією, але мають велику відмінність. Класи і члени класу з модифікатором public також будуть доступні й іншим програмам, якщо помістити їх у динамічну бібліотеку dll і потім її використовувати в цих програмах.

Завдяки такій системі модифікаторів доступу можна приховувати деякі моменти реалізації класу від інших частин програми. Таке приховування називається інкапсуляцією [1, 2].

Для оголошення об'єкта довільного типу використовується така конструкція:

```
<Тип класу> ім'я змінної = new <тип класу> ();
```

Доступ до членів об'єкта здійснюється за допомогою оператора точка «.».

```
InfoUser myinfo = new InfoUser();
namespace ConsoleApplication1
{
    class autoCar
    {
        public string marka;
```

```
        public short year;
    }
    class Program
    {
        static void Main(string[] args)
        {
            // використовуємо ініціалізатори
            autoCar myCar = new autoCar { marka = "Renault", year = 2004 };
            Console.ReadLine();
        }
    }
}
```

Властивості посилальних типів даних

Коли ж одній змінній посилання на об'єкт присвоюється інший, то ситуація дещо ускладнюється, оскільки таке присвоєння призводить до того, що змінна в лівій частині оператора присвоювання посилається на той же самий об'єкт, на який посилається змінна, що знаходиться в правій частині цього оператора. Сам же об'єкт не копіюється. Через цю відмінність присвоювання змінних посилального типу може привести до дещо несподіваних результатів (рис. 2).

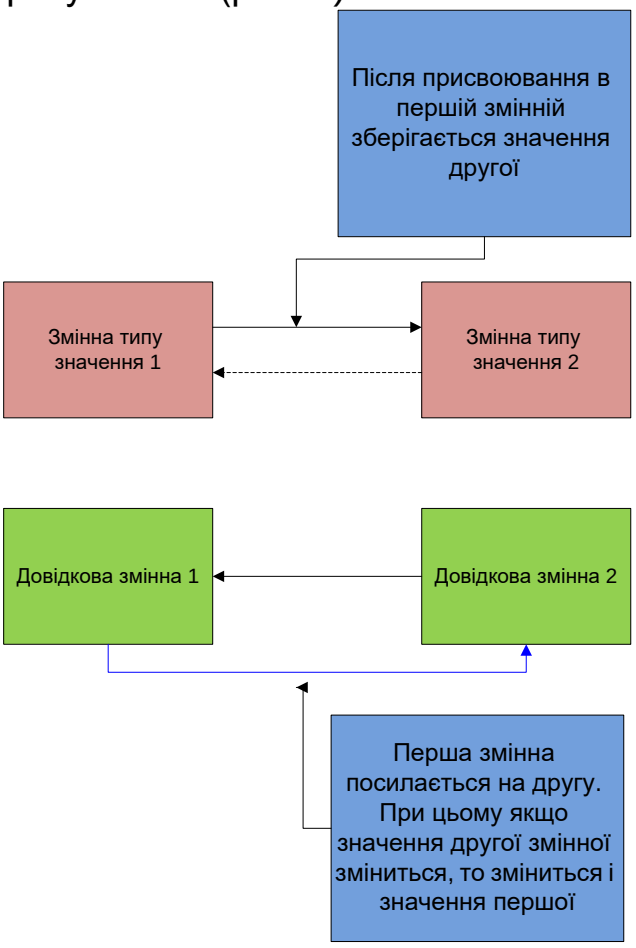

Рис. 2. Механізм роботи посилальних типів даних

Приклад 1:

```
autoCar Car1 = new autoCar();
        autoCar Car2 = Car1;
        Car1.marka = "Renault";
Console.WriteLine(Car1.marka); // виводить одне і те ж
Console.WriteLine(Car2.marka);//
…..
```

Коли змінна Car1 присвоюється змінній Car2, то в кінцевому підсумку змінна Car2 просто посилається на той же самий об'єкт, що і змінна Car1. Отже, цим об'єктом можна оперувати за допомогою змінної Car1 або Car2. Незважаючи на те що обидві змінні, Car1 і Car2, посилаються на один і той же об'єкт, вони ніяк інакше не пов'язані між собою.

Приклад 2:

```
namespace class3_c_sharp
{
   class Person
   {
      public string Name { get; set; }
      public int Age { get; set; }
      public Person(string name, int age)
      {
         Name = name;
         Age = age;
      }
   }
}
//form1.cs
namespace class3_c_sharp
{
public partial class Form1 : Form
{ public Form1()
{ InitializeComponent();}
private void button1_Click(object sender, EventArgs e)
      {
         Person p1 = new Person("xdfcgdf", 12);
         p1.Age = 24;
         p1.Name = "fghgfh";
         textBox1.Text = p1.Name;
         Person p2 = new Person("aaaaa", 20);
         p1 = p2;
textBox2.Text = p1.Age.ToString(); //20
         p1.Age = 15;
textBox3.Text = p1.Age.ToString(); //15
textBox4.Text = p2.Age.ToString(); //15
```

```
            p2.Age = 27;
        textBox5.Text = p1.Age.ToString(); //27
        textBox6.Text = p2.Age.ToString(); //27
            }
        }
    }
```

Методи

Статичний метод – це метод, який не має доступу до полів об'єкта, і для його виклику не потрібно створювати екземпляр (об'єкт) класу, в якому він оголошений.

Простий метод – це метод, який має доступ до даних об'єкта, і його виклик виконується через об'єкт. Прості методи служать для обробки внутрішніх даних об'єкта.

Слід зазначити, що офіційна термінологія C# робить відмінність між функціями і методами. Відповідно до цієї термінології, поняття "функція-член" включає не тільки методи, але також інші члени, які не є даними класу або структури. Сюди входять індексатори, операції, конструктори, деструктори, а також – можливо, дещо несподівано – властивості. Вони контрастують з даними-членами: полями, константами і подіями [3].

Оголошення методів

У C# визначення методу складається з будь-яких модифікаторів (таких як специфікація доступності), типу значення, що повертається, за яким слідує ім'я методу, потім список аргументів у круглих дужках і далі – тіло методу в фігурних дужках:

```
[Модифікатори доступу] тип_повернення ім’я_методу ([параметри])
{
// тіло методу
}
```

Кожен параметр складається з імені, типу параметра та імені, за яким до нього можна звернутися в тілі методу. До того ж, якщо метод повертає значення, то для зазначення точки виходу повинен використовуватися оператор повернення return разом зі значенням, що повертається.

Якщо метод не повертає нічого, то як тип повернення вказується void, оскільки взагалі опустити тип повернення неможливо. Якщо ж він не сприймає аргументів, то все одно після імені методу повинні бути присутніми порожні круглі дужки. При цьому включати в тіло методу оператор повернення не обов'язково – метод повертає керування автоматично після досягнення фігурної дужки.

У цілому, повернення з методу може статися за двох умов. По-перше, коли зустрічається фігурна дужка, що закриває тіло методу. І по-друге, коли виконується оператор return. Є дві форми оператора return: одна – для методів типу void (повернення з методу), тобто тих методів, які не

повертають значення, а інша – для методів, які повертають конкретні значення (повернення значення).

Приклад:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
   class MyMathOperation
   {
      public double r;
      public string s;
      // Повертає площу кола
      public double sqrCircle()
      {
         return Math.PI * r * r;
      }

      // Повертає довжину кола
      public double longCircle()
      {
         return 2 * Math.PI * r;
      }
      public void writeResult()
      {
     Console.WriteLine ("Обчислити площу або довжину? S / l:");
            s = Console.ReadLine ();
            s = s.ToLower ();
            if (s == "s")
            {
               Console.WriteLine ("Площа кола дорівнює {0: #. ###}",
sqrCircle ());
               return;
            }
            else if (s == "l")
            {
               Console.WriteLine ("Довжина кола дорівнює {0: #. ##}",
longCircle ());
               return;
            }
            else
            {
               Console.WriteLine ("Ви ввели не той символ");
```

```
            }
        }
    }
    class Program
    {
        static void Main (string [] args)
        {
            Console.WriteLine ( "Введіть радіус:");
            string radius = Console.ReadLine ();
            MyMathOperation newOperation = new MyMathOperation {r
= double.Parse (radius)};
            newOperation.writeResult ();
            Console.ReadLine ();
        }  }}
```

Використання параметрів

При виклику методу йому можна передати одне або кілька значень. Значення, що передається методу, називається аргументом. А змінна, яка отримує аргумент, називається формальним параметром, або просто параметром. Параметри оголошуються в дужках після імені методу. Синтаксис оголошення параметрів такий же, як і у змінних. А областю дії параметрів є тіло методу.

У загальному випадку метод може передавати параметри або за значенням, або за посиланням. Коли змінна передається за посиланням, метод, що викликається, отримує саму змінну, тому будь-які зміни, яким вона піддається всередині методу, залишаться в силі після його завершення. Але якщо змінна передається за значенням, метод, що викликається, отримує копію цієї змінної, а це значить, що всі зміни в ній по завершенні методу будуть загублені. Для складних типів даних передача за посиланням ефективніша через великий обсяг даних, який доводиться копіювати при передачі за значенням.

Приклад:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace ConsoleApplication1
{
    class myClass
    {
        public void someMethod (double [] myArr, int i)
        {
            myArr [0] = 12.0;
            i = 12;
```

```
        }
    }
    class Program
    {
        static void Main (string [] args)
        {
            double [] arr1 = {0, 1.5, 3.9, 5.1};
            int i = 0;
            Console.WriteLine ("Масив arr1 до виклику методу:");
            foreach (double d in arr1)
                Console.Write ("{0} \ t", d);
            Console.WriteLine ("\ nЗмінна i = {0} \ n", i);
            Console.WriteLine ("Виклик методу someMethod ...");
            myClass ss = new myClass ();
            ss.someMethod (arr1, i);
            Console.WriteLine ("Масив arr1 після виклику методу:");
            foreach (double d in arr1)
                Console.Write ("{0} \ t", d);
            Console.WriteLine ("\ nЗмінна i = {0} \ n", i);
            Console.ReadLine ();
        }
    }
}
```

Зверніть увагу, що значення «i» залишилося незмінним, але змінені значення в «myArr» також змінилися у вихідному масиві «arr1», тому що масиви є посилальними типами.

Поведінка рядків також відрізняється. Справа в тому, що рядки є незмінними (зміна значення рядка приводить до створення абсолютно нового рядка), тому рядки не демонструють поведінку, характерну для посилальних типів. Будь-які зміни, проведені в рядку всередині методу, не впливають на вихідний рядок.

Константи

Особливістю констант є те, що їх значення можна встановити тільки один раз. Наприклад, якщо у програмі є деякі змінні, які не повинні змінювати значення (наприклад, число PI, число E і т. д.), можемо оголосити їх константами. Для цього використовується ключове слово const:

```
const double PI = 3.14;
const double E = 2.71;
```

При використанні констант треба пам'ятати, що оголосити їх можемо тільки один раз і що до моменту компіляції вони повинні бути визначені.

```
class MathLib
{
```

```
    public const double PI = 3.141;
    public const double E = 2.81;
    public const double K;
    public MathLib ()   {
       K = 2.5; // помилка – константа повинна бути визначена до компіляції
     }}
  class Program
  {
    static void Main (string [] args)
    {
       MathLib.E = 3.8; // константу можна встановити кілька разів
    }
   }
```

Також зверніть увагу на синтаксис звернення до константи. Це статичне поле, тому нам необов'язково створювати об'єкт класу за допомогою конструктора. Можемо звернутися до неї, використовуючи ім'я класу.

Поля для читання

Поля для читання схожі на константи: їх також не можна встановити двічі, проте їх можна встановлювати під час виконання програми, наприклад, у конструкторі – неприпустимо.

Поле для читання оголошується з ключовим словом readonly:

```
class MathLib
{
    public readonly double K;

    public MathLib (double _k)
    {
       K = _k; // поле для читання може бути визначено після компіляції
    }
}
class Program
{
    static void Main (string [] args)
    {
       MathLib mathLib = new MathLib (3.8);
       Console.WriteLine (mathLib.K); // 3.8
       //mathLib.K = 7.6; // поле для читання можна встановити ізольованим від свого класу
       Console.ReadLine ();
    }
}
```

Лекція 3. ВЛАСТИВОСТІ ТА АВТОВЛАСТИВОСТІ

Властивість у C# – це член класу, який надає зручний механізм доступу до поля класу (читання поля і запис).

При використанні властивості звертаємося до неї як до поля класу, але насправді компілятор перетворює це звернення до виклику відповідного неявного методу. Такий метод називається аксесор (accessor). Існує два таких методи: get (для отримання даних) і set (для запису). Оголошення простої властивості має таку структуру:

```
[Модифікатор доступу] [тип] [ім’я_властивості]
{
  get
  {
    // тіло аксесора для читання з поля
  }
  set
  {
    // тіло аксесора для запису в поле
  }
}
```

Наведемо приклад використання властивостей. Є клас Студент, і в ньому є закрите поле “курс”, яке не може бути нижче одиниці і більше п'яти. Для керування доступом до цього поля буде використано властивість Year:

```
class Student
{
  private int year; // оголошення закритого поля

  public int Year // оголошення властивості
  {
    get // аксесор читання поля
    {
      return year;
    }
    set // аксесор запису в поле
    {
      if (value <1)
        year = 1;
      else if (value> 5)
        year = 5;
      else year = value;
    }
  }
}
```

```
class Program
{
  static void Main (string [] args)
  {
    Student st1 = new Student ();
    st1.Year = 0; // записуємо в поле, використовуючи аксесор set
Console.WriteLine (st1.Year); // читаємо поле, використовуючи аксесор get,
//виведе 1
    Console.ReadKey ();
  }
}
```

Простіше кажучи, у властивості реалізуються два методи. У тілі аксесора get може бути більш складна логіка доступу, але в підсумку повинно повертатися значення поля або інше значення за допомогою оператора return. У аксесора set присутній неявний параметр value, який містить значення, що присвоюється властивості.

Якщо, наприклад, просто зробити поле year відкритим і не використовувати ні методи, ні властивість для доступу, можна було б записати в це поле будь-яке значення, навіть і некоректне, а так можемо контролювати читання і запис.

Властивість також може надавати доступ тільки до читання поля або тільки до запису. Якщо, наприклад, необхідно закрити доступ до запису, тоді просто не вказуємо аксесор set.

Властивості дозволяють вкласти додаткову логіку, яка може бути необхідна [4].

Блоки set і get не обов'язково одночасно повинні бути присутніми у властивості. Наприклад, можемо закрити властивість від установки, щоб можна було тільки отримувати значення. Для цього опускаємо блок set. І, навпаки, можна видалити блок get, тоді можна буде тільки встановити значення, але не можна отримати:

```
class Person
{
   private string name;
   // властивість тільки для читання
   public string Name
   {
      get
      {
         return name;
      }
   }

   private int age;
   // властивість тільки для запису
```

```
    public int Age
    {
        set
        {
            age = value;
        }
    }
}
```

Можемо застосовувати модифікатори доступу не тільки до всієї властивості, але і до окремих блоків – або get, або set. При цьому якщо застосовуємо модифікатор до одного з блоків, то до іншого вже не можемо застосувати модифікатор:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }
        private set
        {
            name = value;
        }
    }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Тепер закритий блок set зможемо використовувати тільки в певному класі – в його методах, властивості, конструкторі, але ніяк не в іншому класі:

```
Person p = new Person ("Tom", 24);

// Помилка – set оголошений з модифікатором private

//p.Name = "John";
```

```
Console.WriteLine (p.Name);
```

Інкапсуляція

Вище було розглянуто, що через властивості встановлюється доступ до приватних змінних класу. Подібне приховування стану класу від втручання ззовні являє собою механізм інкапсуляції, який є однією із ключових концепцій об'єктно-орієнтованого програмування. Застосування модифікаторів доступу типу private захищає змінну від зовнішнього доступу. З ідеєю інкапсуляції програмної логіки тісно пов'язана ідея захисту даних. Основною одиницею інкапсуляції в C# є клас, що визначає форму об'єкта. Він описує дані, а також код, який буде ними оперувати. У C# опис класу служить для побудови об'єктів, які є екземплярами класу. Отже, клас, по суті, являє собою ряд схематичних описів способу побудови об'єкта.

Автоматичні властивості

Автоматична властивість – це дуже проста властивість, яка, на відміну від звичайної властивості, вже визначає місце в пам'яті (створює неявне поле), але при цьому не дозволяє створювати логіку доступу. Структура оголошення автоматичної властивості:

[Модифікатор доступу] [тип] [ім’я_властивості] {get; set; }

У таких властивостей, у їх аксесорів відсутнє тіло. Приклад використання:

```
class Student {public int Year {get; set; }}
class Program
{Static void Main (string [] args)
  {Student st1 = new Student ();
    st1.Year = 0;
    Console.WriteLine (st1.Year);
    Console.ReadKey ();
  }
}
```

Автоматично реалізовані властивості є сенс використовувати тоді, коли немає необхідності накладати будь-які обмеження на можливі значення неявного поля властивості [3].

І тут може виникнути питання, а в чому тоді різниця між простими відкритими полями і автоматичними властивостями. У таких властивостей залишається можливість робити їх тільки для читання або тільки для запису. Для цього вже використовується модифікатор доступу private перед ім'ям аксесора [4]:

```
public int Year {private get; set; } // властивість тільки для запису
public int Year {get; private set; } // властивість тільки для читання
```

Насправді у випадку автовластивостей також створюються поля для властивостей, тільки їх створює не програміст у коді, а компілятор – автоматично генерує при компіляції.

З одного боку, автоматичні властивості досить зручні. З іншого боку, стандартні властивості мають низку переваг: наприклад, вони можуть інкапсулювати додаткову логіку перевірки значення; не можна створити автоматичну властивість тільки для запису, як у випадку зі стандартними властивостями.

У C# 6.0 була додана така функціональність, як ініціалізація автовластивостей:

```
class Person
{
    public string Name { get; set; } = "Tom";
    public int Age { get; set; } = 23;
}
class Program
{
    static void Main(string[] args)
    {
        Person person = new Person();
        Console.WriteLine(person.Name); // Tom
        Console.WriteLine(person.Age);  // 23

        Console.Read();
    }
}
```

І якщо не вказуємо для об'єкта Person значення властивостей Name і Age, то будуть діяти значення за замовчуванням.

Ще одна зміна торкнулася визначення автовластивостей. Наприклад, якщо в C# 5.0 була необхідність зробити автовластивість доступною для установки тільки з класу, то треба було вказати private set:

```
class Person
{
    public string Name { get; private set; }
    public Person(string n)
    {
        Name = n;
    }
}
```

Цю властивість можна встановити тільки з класу Person. В C# 6.0 нам необов'язково писати private set:

```
class Person
```

```
{
    public string Name { get;}
    public Person(string n)
    {
        Name = n;
    }
}
```

Розглянемо приклад по звірці класу. Завдання:

/ *Описати клас «товар», що містить такі закриті поля:

• назва товару;

• назва магазину, в якому продається товар;

• вартість товару в гривнях.

Передбачити властивості для отримання стану об'єкта.

Описати клас «склад», що містить закритий масив товарів. Забезпечити такі можливості:

• вивести інформацію про товар по номеру за допомогою індексу;

• вивести на екран інформацію про товар, назву якого введено з клавіатури;

• якщо таких товарів немає, видати відповідне повідомлення;

• сортувати товари за назвою магазину, за найменуванням і за ціною;*/

Файли проекту наведені нижче.

```
//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;
namespace class_example1
{
    public partial class Form1 : Form
    {   int i;
    Product [] masProd;
        public Form1()
        {
            InitializeComponent();
        }

        private void toolStripMenuItem2_Click(object sender, EventArgs e)
        {
            i=0;
            while (dataGridView1.Rows[i].Cells[0].Value != null)
```

```
                i++;
              masProd = new Product[i];
              //  Product masProd = new Product();
                for (int j = 0; j < i ; j++)
                {
        masProd[j] = new Product();
        masProd[j].Name = dataGridView1.Rows[j].Cells[0].Value.ToString();
        masProd[j].Shop = dataGridView1.Rows[j].Cells[1].Value.ToString();
       masProd[j].COST=
Convert.ToSingle(dataGridView1.Rows[j].Cells[2].Value);
                }
                toolStripMenuItem3.Enabled = true;
                toolStripMenuItem4.Enabled = true;
            }

            private void Form1_Load(object sender, EventArgs e)
            {
                dataGridView1.ColumnCount = 3;
                dataGridView1.RowCount = 10;
                dataGridView1.Columns[0].Name = "Найменування товару";
                dataGridView1.Columns[1].Name = "Магазин";
                dataGridView1.Columns[2].Name = "Вартість";

            }

            private void button1_Click(object sender, EventArgs e)
            {
                textBox1.Text = i.ToString();
            }

            private void toolStripMenuItem3_Click(object sender, EventArgs e)
            {
                int n = int.Parse(textBox2.Text);
                textBox3.Text = masProd[n - 1].Name + Environment.NewLine;
                textBox3.Text += masProd[n - 1].Shop + Environment.NewLine;
                textBox3.Text += masProd[n - 1].COST.ToString();

            }

            private void toolStripMenuItem4_Click(object sender, EventArgs e)
            {
                String Naz = textBox2.Text;
                int count=0;
                foreach (Product pr in masProd)
```

```
            {
                if (pr.metodName(Naz))
                {
                    textBox3.Text += pr.Shop + Environment.NewLine;
                    textBox3.Text += pr.COST + Environment.NewLine;
                    textBox3.Text += Environment.NewLine;
                    count++;
                }
            }
                if (count==0) MessageBox.Show("Таких товарів немає!!!");
        }

        private void toolStripMenuItem5_Click(object sender, EventArgs e)
        {
            dataGridView1.Columns.Clear();
            dataGridView1.ColumnCount = 3;
            dataGridView1.RowCount = 10;
            dataGridView1.Columns[0].Name = "Найменування товару";
            dataGridView1.Columns[1].Name = "Магазин";
            dataGridView1.Columns[2].Name = "Вартість";
            toolStripMenuItem3.Enabled = false;
            toolStripMenuItem4.Enabled = false;
            textBox2.Text = "";
            textBox3.Text = "";
            listBox1.Items.Clear();
        }

        private void toolStripMenuItem6_Click(object sender, EventArgs e)
        {
            listBox1.Items.Clear();
            foreach (Product pr in masProd)
            {
                listBox1.Items.Add(pr.Name + "      " + pr.Shop + "      " +
pr.COST.ToString());
            }

        }

        private void toolStripMenuItem7_Click(object sender, EventArgs e)
        {
          /*  Array.Sort<Product>(masProd);
            listBox1.Items.Add("---------------------------------");
            foreach (Product pr in masProd)
            {
```

```
                listBox1.Items.Add(pr.Name  +  "      "  +  pr.Shop  +  "      "  +
pr.COST.ToString());
              }*/
          }
        }
      }
      ////////////////////////////
      Product.cs
      ////////////////////////////
      using System;
      using System.Collections.Generic;
      using System.Linq;
      using System.Text;
      namespace class_example1
      {
        class Product
        {
          private String nameProdact;
          private String nameShop;
          private float cost;

          public Product()
          {
            nameProdact = "невідомо";
            nameShop = "невідомо";
            cost = 0;
          }

          public Product(String prod, String shop, float cost1)
          {
            nameProdact = prod;
            nameShop = shop;
            cost = cost1;
          }

          public String Name
          {
            get
            {
              return nameProdact;
            }
            set
            {
              nameProdact = value;
```

```
        }
    }

    public String Shop
    {
        set { nameShop = value; }
        get { return nameShop; }
    }

    public float COST
    {
        get { return cost; }
        set { cost = value; }
    }

    public bool metodName(String name1)
    {
        if (name1 == nameProdact) return true;
        else return false;
    }

  }
}
```

Інтерфейс проекту подано на рис. 3.

Рис. 3. Інтерфейс проекту

Лекція 4. ІНДЕКСАТОРИ

Індексатор є різновидом властивості. Якщо у класу є приховане поле, що являє собою масив, то за допомогою індексатора можна звернутися до елемента цієї властивості, використовуючи ім'я об'єкта і номер елемента масиву в квадратних дужках.

Синтаксис опису:

специфікатор_доступу тип this [опис_індексів]
{get код_доступу
set код_доступу}

Розглянемо приклад. Припустимо, у нас є клас, у якому сховище визначено у вигляді двовимірного масиву або матриці:

```
class Matrix {
private int [,] numbers = new int [,] {{1, 2, 4}, {2, 3, 6}, {3, 4, 8}};
public int this [int i, int j]
{
get {return numbers [i, j]; }
set {numbers [i, j] = value; }
}}
```

Тепер для визначення індексатора використовуються два індекси – «i» та «j». І в програмі вже можемо звертатися до об'єкта, використовуючи два індекси:

```
 Matrix matrix = new Matrix ();
Console.WriteLine (matrix [0, 0]);
matrix [0, 0] = 111;
Console.WriteLine (matrix [0, 0]);
```

Приклад.

Завдання:

Описати клас “поїзд”, що містить такі закриті поля:

• назва пункту призначення;

• номер поїзда (може містити букви і цифри);

• час відправлення.

Передбачити властивості для отримання стану об'єкта.

Описати клас “вокзал”, що містить закритий масив поїздів. Забезпечити такі можливості:

• отримання інформації про поїзд за номером за допомогою індексу;

• отримання інформації про поїзди, що відправляються після введеного з клавіатури часу;

• отримання інформації про поїзди, що відправляються в заданий пункт призначення.

Інформація повинна бути відсортована за часом відправлення.

Написати програму, яка демонструвала б усі розроблені елементи класів.

```
// train.cs
namespace class2_c_sharp
{
 class train
 {
 private String name;
 private String nom;
 private int time_h;
 private int time_m;
 public train ()
 {
 name = "Unknown";
 nom = "Unknown";
 time_h = 0;
 time_m = 0;
 }
public train (String name, String nom, int time_h, int time_m)
 {
 this.name = name;
 this.nom = nom;
 this.time_h = time_h;
 this.time_m = time_m;
 }
 public String Name
 {
 get { returnname; }
 set {Name =value;}
 }
 public String Nom
 {
 get { returnnom; }
 set {Nom = value; }
 }
 public int Time_h
 {
 get { returntime_h; }
 set {Time_h = value; }
 }
 public int Time_m
 {
 get { returntime_m; }
```

```
 set {Time_m = value; }
 }
 public bool method1 (String Name1)
 {
 if (Name1 == name) return true;
 else return false;
 }
 }
}
```
// train_station.cs
```
namespace class2_c_sharp
{
 class train_station
 {
 private train[] Mas;
 private int n;
 public train_station ()
 {
 n = 5;
 mas = new train [N];
 for (inti = 0; i <n; i ++)
 mas [i] = new train();
 }
 public train_station (int kol)
 {
 n = kol;
 mas = new train[N];
 for (inti = 0; i <n; i ++)
 mas [i] = new train("Харків", "12343", 12, 34);
 }
 public train this[int index]
 {
 get
 {
 return mas [index];
 }
 set { if (Index> -1 && index <n) mas [index] = value; }
 }
  }
}
```
//form1.cs
```
namespace class2_c_sharp
{
 public partial class Form1 : Form
```

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

}
train_station ob1;
private void button1_Click (object sender, EventArgs e)
{
listBox1.Items.Clear ();
for (inti = 0; i <3; i ++)
if (Ob1 [i] .method1 ("Київ"))
listBox1.Items.Add (ob1 [i] .Nom + "" + Ob1 [i] .Time_h + "" + Ob1 [i] .Time_m);
}


private void Form1_Load (object sender, EventArgs e)
{
ob1 = new train_station(3);
ob1 [0] .Name = "Київ";
ob1 [0] .Nom = "12345";
ob1 [0] .Time_h = 12;
ob1 [0] .Time_m = 30;

ob1 [1] .Name = "Дніпро";
ob1 [1] .Nom = "12000";
ob1 [1] .Time_h = 16;
ob1 [1] .Time_m = 45;

ob1 [2] .Name = "Чернігів";
ob1 [2] .Nom = "5689";
ob1 [2] .Time_h = 14;
ob1 [2] .Time_m = 15;
}
}
```

Перевантаження операторів

У C# існує ряд операторів для роботи з вбудованими типами даних. Це оператори «+», «-», «!», «==», «! =» і т. д. Наприклад, бінарний оператор «+» виконує операцію додавання над чисельними типами даних.

Перевантаження оператора – це реалізація власного функціоналу цього оператора для конкретного класу**.**

Перевантаження унарного оператора:

public static [тип_повернення] operator [оператор] ([тип_операнда] [операнд]) {// функціонал оператора

}

Перевантаження бінарного оператора:

public static [тип_повернення] operator [оператор] ([тип_операнда1] [операнд1], [тип_операнда2] [операнд2])

{// функціонал оператора

}

Бінарні оператори приймають два параметри, унарні – один параметр.

Можна перевантажувати

Унарні оператори: +, -,!, ++, -, true, false;
Бінарні оператори: +, -, *, /,%, &, |, ^, <<, >>, ==,! =, <,>, <=,> =.

Не можна перевантажувати

[] – функціонал цього оператора надають індексатори;
() – функціонал цього оператора надають методи перетворення типів;
+ =, - =, * =, / =,% =, & =, | =, ^ =, << =, >> = короткі форми оператора присвоювання будуть автоматично доступні при перевантаженні відповідних операторів (+, -, * ...) .

Слід враховувати, що при перевантаженні не повинні змінюватися ті об'єкти, які передаються в оператор через параметри. Тобто повертається новий об'єкт.

```
class Team {
private string name;
private int account;
public string Name {
 get { returnname; }
 set {Name = value; }}
public int Account{
 get { return account; }
set { if (value > = 0) account = value; else account = 0; }}
 public Team (){
 name = "Team1";
 account = 1;}
...
public static Team operator ++ (Team t1) {
 return new Team {Account = t1.Account + 1};}
 / * Public static Team operator ++ (Team t1)
// оператор не повинен змінювати значення своїх параметрів
{T1.Account ++;
```

return t1;} * /}

При цьому не треба визначати окремо оператори для префіксного і для постфіксного інкремента (а також декремента), тому що одна реалізація буде працювати в обох випадках.

```
class Team {
...
public static Team operator + (Team t1, Team t2){
return new Team {Account = t1.Account + t2.Account};}
public static bool operator <(Team t1, Team t2){
if (T1.Account <t2.Account) return true;
 else return false;}
public static bool operator > (Team t1, Team t2){
if (T1.Account> t2.Account) return true;
 else return false;}
public static int operator + (Team t1, int val){
 return t1.Account + val; }
...}
```

Також існує можливість перевантаження самого операторного методу. Це означає, що в класі може бути кілька перевантажень одного оператора за умови, що вхідні параметри будуть відрізнятися типом даних (наприклад, коли необхідно скласти об'єкт класу і рядок).

```
class Program{ static void Main (string[] Args) {
Team t11 = new Team {Account = 20};
Team t12 = new Team {Name ="Team12", Account = 30};
bool res = t11> t12;
Console.WriteLine ("T11> t12 {0}", Res);
Team t13 = t11 + t12;
Console.WriteLine ("T13.Account = {0}", T13.Account);
t11 ++;
++ t11;
Team t14 = ++ t11;
Console.WriteLine ("T14.Account = {0}", T14.Account);
t14 + = t12;
Console.WriteLine ("T14 + = t12; t14.Account = {0}", T14.Account);
int k = t12 + 5;
Console.WriteLine ("K = {0}", K);
 Console.ReadKey ();}}
```

Лекція 5. УСПАДКОВУВАННЯ

Принцип ООП – **успадковування** – стосується здатності мови дозволяти будувати нові визначення класів на основі визначень існуючих класів. По суті, успадковування дозволяє розширювати поведінку базового (або батьківського) класу, наслідуючи основну функціональність у похідному підкласі (що також іменується дочірнім класом).

Тобто успадкування є процесом, під час якого один об'єкт набуває властивостей іншого. Це дуже важливий процес, оскільки він забезпечує принцип ієрархічної класифікації. Якщо вдуматися, то велика частина знань піддається систематизації завдяки ієрархічній класифікації по низхідній.

Якщо не користуватися ієрархіями, то для кожного об'єкта довелося б явно визначати всі його властивості. А якщо скористатися спадкуванням, то досить визначити лише ті властивості, які роблять об'єкт особливим у його класі. Він може також успадковувати загальні властивості свого батька. Отже, завдяки механізму успадковування один об'єкт стає окремим екземпляром більш загального класу.

Завдяки спадкуванню один клас може успадкувати функціональність іншого класу. Після двокрапки вказуємо базовий клас для конкретного класу.

```
class Buy: Product
{}
```

Усі класи за замовчуванням можуть успадковуватися. Однак тут є кілька обмежень:

- не підтримується множинне успадковування, клас може успадковуватися тільки від одного класу;
- при створенні похідного класу треба враховувати тип доступу до базового класу – тип доступу до похідного класу повинен бути таким самим, як і до базового, або більш суворим. Тобто, якщо базовий клас має тип доступу **internal**, то похідний клас може мати тип доступу **internal** або **private**, але не **public**;
- якщо клас оголошений з модифікатором **sealed**, то від цього класу не можна успадковувати класи. Наприклад, цей клас не допускає створення спадкоємців:

```
sealed class Admin {}
```

- у межах похідного класу можна звертатися до елементів базового класу зі специфікатором доступу **private**. Таким чином, похідний клас може мати доступ тільки до тих членів базового класу, які визначені з модифікаторами **public, internal, protected** і **protected internal**;
- конструктори не передаються похідному класу при успадкуванні.

За допомогою ключового слова base можемо звернутися до базового класу.

Віртуальні методи

При спадкуванні нерідко виникає необхідність змінити в класі-спадкоємці функціонал методу, який був успадкований від базового класу. В цьому випадку клас-спадкоємець може перевизначати методи і властивості базового класу.

Методи і властивості, які необхідно зробити доступними для перевизначення, в базовому класі позначаються модифікатором **virtual**. Такі методи і властивості називають віртуальними. А щоб перевизначити метод у класі-спадкоємці, цей метод визначається з модифікатором **override**. Перевизначення методу в класі-спадкоємці повинно мати той же набір параметрів, що і віртуальний метод у базовому класі. У будь-якому похідному класі, який не є прямим спадкоємцем базового класу, можна перевизначити віртуальні методи.

Також можна заборонити перевизначення методів і властивостей. В цьому випадку їх треба оголошувати з модифікатором **sealed**.

При створенні методів з модифікатором sealed треба враховувати, що sealed застосовується в парі з override, тобто тільки в перевантажених методах.

Іншим способом змінити функціональність методу, успадкованого від базового класу, є приховування (shadowing / method hiding).

Фактично приховування являє собою визначення в класі-спадкоємці методу, який відповідає імені і набору параметрів методу базового класу. Для приховування методу застосовується ключове слово new.

Метод з ключовим словом new приховує реалізацію цього методу з базового класу.

При присвоєнні змінній базового класу об'єкта похідного класу за замовчуванням через цю змінну можна викликати тільки ті методи і властивості, які є в базовому класі. Компілятор до виконання програми на етапі компіляції повинен визначити адресу методу, який буде викликатися. Цей процес називається **раннім зв'язуванням** (early binding). При цьому при виборі реалізації методу компілятор керується типом змінної, а не типом об'єкта, посилання на який зберігається в цій змінній. Але також є механізм **пізнього зв'язування**, що являє собою вибір реалізації методу на етапі виконання. Пізнє зв'язування реалізується за допомогою віртуальних методів.

Завдання

Розробити три класи, які слід пов'язати між собою, використовуючи успадкування:

1. Клас Product, який має три елемент-даних – ім'я, ціна і вага товару (базовий клас для всіх класів).
2. Клас Buy, який містить дані про кількість товару, що купується в штуках, про ціну всього купленого товару і про вагу товару (похідний клас для класу Product і базовий клас для класу Check).

3. Клас Check, який не містить жодних елемент-даних. Цей клас повинен виводити на екран інформацію про товар і про покупку (похідний клас для класу Buy).

Для взаємодії з даними класів розробити set- і get-методи. Всі елемент-дані класів оголошувати як private.

//Product.cs

```
...
 class Product {
 private string name;
 private double price;
 private double weight;
 public string Name {
get {return name; }
set {name = value; }}
 public double Price {
get {return price; }
set {if (price <= 0) price = 0; else price = value; }}
 public double Weigth{
get {return weight; }
set {if (weight <= 0) weight = 0; else weight = value;}}
public Product(String name1, double price1, double weight1) {
 name = name1;
 price = price1;
 weight = weight1;}
 public Product (){
 name = "carrot";
 price = 7;
 weight = 1;}
 public double getCost () {
 return price * weight;}
 public virtual void Show (){
Console.Write ($ "name: {name}, price: {price}, weight: {weight}"); }
 public void ShowProduct () {
Console.Write ($ "name: {name}, cost (price * Weigth): {getCost ()}");}
 public virtual void ShowMessage() {Console.WriteLine ( "Товар: {0}, куплений.", Name);}}}
```

// class Buy

```
...
 class Buy: Product {
 private string market;
 public Buy(String name2, double price2, double weight2, string market2)
 : base (name2, price2, weight2) {
 market = market2;}
public Buy (string market2) : base () {
```

```
market = market2; }
 public Buy (): base (){
market = "Metro"; }
 public string Market{
get {return market; }
set {market = value; }}
public override void Show (){
 base.Show ();
Console.WriteLine ("Market: {0}", market);}
 public override sealed void ShowMessage (){
Console.WriteLine ("Супермаркет {0}, дякує за покупку!", Market);}}}
```

//Check.cs

```
...
 class Check: Buy {
 private DateTime date;
 public virtual DateTime Date
{Get {return date; }}
 public Check (): base () {
date = DateTime.Now; }
public Check (string n, double p, double w, string m): base (n, p, w,
m){
 date = DateTime.Now;}
 public override void Show (){
 base.Show ();
Console.WriteLine ("Дата в чеку: {0}", Date); }
public new void ShowProduct () {
 Console.WriteLine (Name);
Console.WriteLine ("{0} X {1} {2}", Weigth, Price, getCost ()); }}}
```

//Program.cs

```
...
 class Program {
 static void Main (string [] args) {
Product p = new Product {Name = "cheese", Price = 120, Weigth = 1.5};
 p.Show ();
 Console.WriteLine ("Cost: {0}", p.getCost ());
 Buy b1 = new Buy ();
 b1.Show ();
 Buy b2 = new Buy ("Клас");
 b2.Show ();
 Buy b = new Buy ("apple", 15, 2.5, "Ашан");
 b.Show ();
 Console.WriteLine ();
 Check c = new Check ();
 c.Show ();
```

```
 Product product1 = new Buy ("melon", 12, 3, "ATB");
product1.ShowProduct (); // name: melon, cost // (price * Weigth): 36
 Console.WriteLine ();
 Product product2 = new Buy ("melon", 12, 3, "ATB");
product1.Show (); // name: melon, price: 12, // weight: 3 Market: ATB
 Console.WriteLine ();
 Console.WriteLine ();
 List <Check> Sale = new List <Check> (2);
 Sale.Add (new Check ("tomato", 10, 2, "Metro"));
 Sale.Add (new Check ("milk", 14, 1.5, "Metro"));
 Sale.Add (new Check ());
Sale.Add (new Check () {Name = "strawberry", Price = 30, Weigth = 1});
 foreach (Check ch in Sale)
 ch.ShowProduct ();
 Sale [Sale.Count - 1] .ShowMessage ();
 Console.WriteLine (Sale [Sale.Count - 1] .Date);
Console.ReadKey (); }}
```

На рис. 4 наведено діаграму класів проекту, отриману автоматично в середовищі Visual Studio.

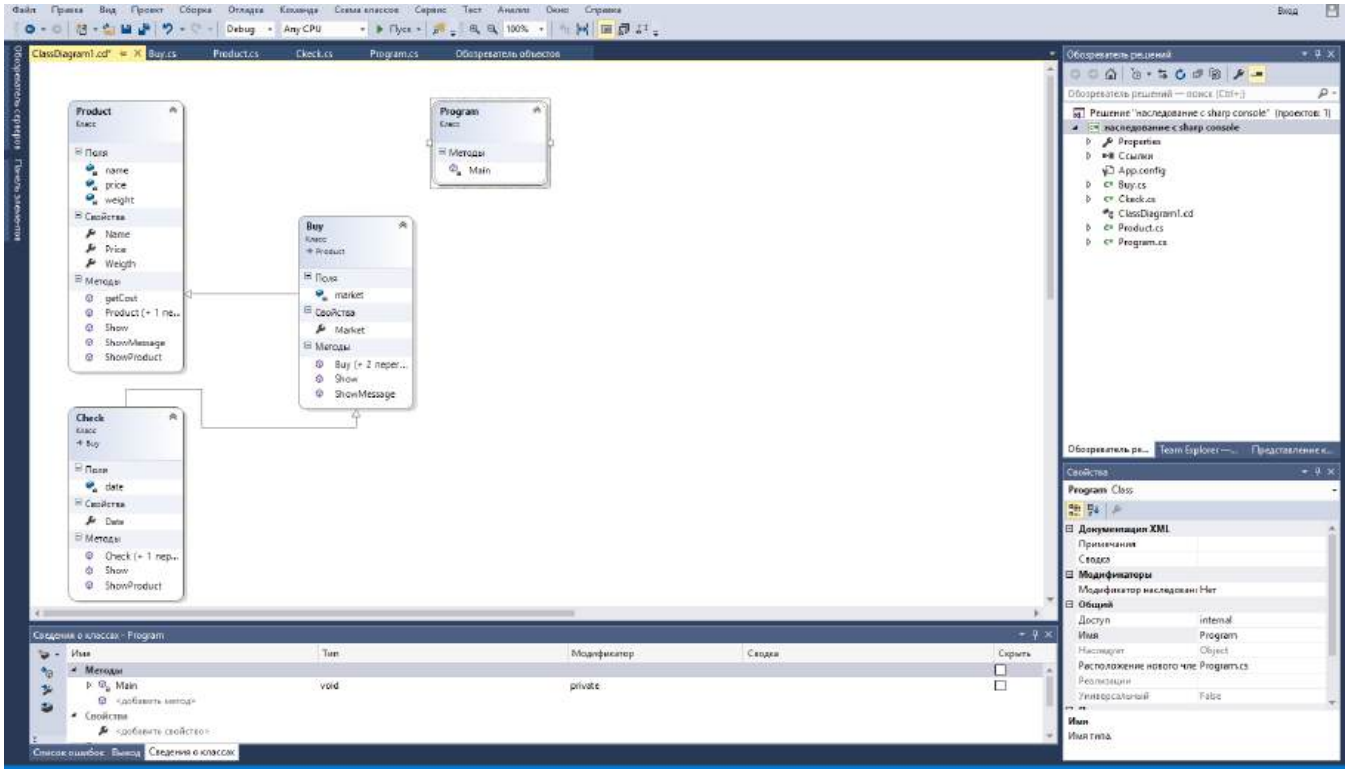

Рис. 4. Діаграма класів

Лекція 6. ПОЛІМОРФІЗМ

Останній принцип ООП – **поліморфізм**. Він позначає здатність мови трактувати пов'язані об'єкти в схожій манері. Зокрема, цей принцип ООП дає змогу базовому класу визначати набір членів (формально поліморфним інтерфейсом), які доступні всім спадкоємцям. Поліморфний інтерфейс класу конструюється з використанням будь-якої кількості віртуальних або абстрактних членів.

Абстрактні класи

Абстрактні класи також можуть мати змінні, методи, конструктори, властивості. При визначенні абстрактних класів використовується ключове слово abstract.

```
abstract class Human
{
public int Length {get; set; }
public double Weight {get; set; }
}
```

Але головна відмінність полягає в тому, що **не можна** використовувати конструктор абстрактного класу для створення його об'єкта. Наприклад, таким чином: Human h = new Human ();

Крім звичайних властивостей і методів абстрактний клас може мати **абстрактні методи і властивості**. Подібні методи і властивості визначаються за допомогою ключового слова abstract та не мають функціоналу. Крім того, вони не повинні мати модифікатор private.

При цьому похідній клас зобов'язаний перевизначити і реалізувати всі абстрактні методи і властивості, які є в базовому абстрактному класі. При перевизначенні в похідному класі такий метод або властивість також оголошується з модифікатором **override** (як і при звичайному перевизначенні віртуальних методів і властивостей). Також слід врахувати, що якщо клас має хоча б одну абстрактну властивість або метод, то він повинен бути визначений як **абстрактний**.

Абстрактні методи, як і віртуальні, є частиною поліморфного інтерфейсу. Але, якщо у випадку з віртуальними методами говоримо, що клас-спадкоємець успадковує реалізацію, то у випадку з абстрактними методами успадковується інтерфейс, представлений цими абстрактними методами.

Іншим хрестоматійним прикладом є система геометричності фігур. В реальності не існує геометричної фігури як такої. Є коло, прямокутник, квадрат, але просто фігури немає. Однак і коло, і прямокутник мають щось спільне і є фігурами.

Завдання

Розробити абстрактний базовий клас із двома віртуальними методами, що обчислюють площу і периметр плоскої фігурі. Розробити похідні класи: прямокутник, коло зі своїми методами обчислення площі периметра.

```
// class Figure
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace абстрактні_класи
{
 abstract class Figure
 {
 public abstract float Perimeter ();
 public abstract float Area ();
 }
}
// class Rectangle
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace абстрактні_класи
{
 class Rectangle: Figure
 {
 public float Width { get; set; }
 public float Height { get; set; }
 public Rectangle ()
 {
 Width = 1;
 Height = 1;
 }
 public Rectangle (float w, float h)
 {
 Width = w;
 Height = h;
 }
 public override float Perimeter ()
```

```
        {
        return (Width + Height) * 2;
        }
        public override float Area ()
        {
        return Width * Height;
        }
        }
       }
       // class Circle
       namespace абстрактні_класи
       {
        class Circle: Figure
        {
        private float R;
        public float r
        {
        get {return R;}
        set { if (value > 0) R = value; else Console.WriteLine ("Значення має бути більше 0!!!"); }
        }
        public Circle ()
        {
        R = 1;
        }
        public Circle (float radius)
        {
        R = radius;
        }
        public override float Perimeter ()
        {
        return 2 * 3.14f * R;
        }
        public override float Area ()
        {
        return 3.14f * R * R;
        }
        }
       }
       // class Program
       class Program
        {
        static void Main (string[] Args)
        {
```

```
Console.Write ("Введіть радіус:");
float r1 = float.Parse (Console.ReadLine ());
Circle c1 = new Circle(r1);
float p1 = c1.Perimeter ();
Console.WriteLine ("Периметр кола дорівнює {0}", p1);
Console.WriteLine ($ "Периметр кола дорівнює { p1}");
Console.WriteLine ("Площа кола дорівнює {0}", c1.Area ());
Console.WriteLine ();
Rectangle rect1 = new Rectangle();
Rectangle rect2 = new Rectangle(3,4);
Console.WriteLine ("Периметр rect1 дорівнює {0}", rect1.Perimeter ());
Console.WriteLine ("Площа rect1 дорівнює {0}", rect1.Area ());
Console.WriteLine ();
Console.WriteLine ("Периметр rect2 дорівнює {0}", rect2.Perimeter ());
Console.WriteLine ("Площа rect2 дорівнює {0}", rect2.Area ());
// Figure f = new Figure ();
Figure f = new Rectangle();
Console.WriteLine ("Периметр f дорівнює {0}", f.Perimeter ());
Console.WriteLine ("Площа f дорівнює {0}", f.Area ());
Console.ReadKey ();
}}
```

Лекція 7. СТАТИЧНІ ЧЛЕНИ КЛАСУ. СТАТИЧНІ КЛАСИ

Метод – це невелика підпрограма, яка виконує, в ідеалі, тільки одну функцію. Методи дозволяють скоротити обсяги коду.

Методи разом з полями є основними членами класу.

Статичний метод – це метод, який не має доступу до полів об'єкта, і для його виклику не потрібно створювати екземпляр (об'єкт) класу, в якому він оголошений. **Простий метод** – це метод, який має доступ до даних об'єкта, і його виклик виконується через об'єкт. Прості методи служать для обробки внутрішніх даних об'єкта.

Приклад використання простого методу.

Клас *Телевізор*, у нього є поле switchedOn, яке відображає стан увімкнений/вимкнений, і два методи – увімкнення і вимкнення:

```
class TVSet {
 private bool switchedOn;
public void SwitchOn () {switchedOn = true; }
public void SwitchOff () {switchedOn = false;}
class Program {static void Main (string [] args) {
TVSet myTV = new TVSet ();
myTV.SwitchOn (); // вмикаємо телевізор,
switchedOn = true;
myTV.SwitchOff (); // вимикаємо телевізор,
```

```
switchedOn = false; }
```

Щоб викликати простий метод, перед його ім'ям вказується ім'я об'єкта. Для виклику статичного методу необхідно вказувати ім'я класу. Статичні методи зазвичай виконують якусь глобальну, загальну функцію, обробляють «зовнішні дані». Наприклад, сортування масиву, обробка рядка, зведення числа в ступінь та ін. Приклад статичного методу, який обрізає рядок до зазначеної довжини і додає три крапки:

```
class StringHelper
 {
public static string TrimIt (string s, int max)
{
if (s == null)
return string.Empty;
if (s.Length <= max)
 return s; return s.Substring (0, max) + "...";
 }
}
Class Program
 {
static void Main (string [] args)
 {
string s = "Дуже довгий рядок, який необхідно обрізати до зазначеної довжини і додати три крапки";
Console.WriteLine (StringHelper.TrimIt (s, 20)); // "Дуже довгий рядок ..."
Console.ReadLine ();
 }
}
```

Статичний метод не має доступу до нестатичних полів класу (тільки до статичних).

```
class SomeClass {
private int a;
private static int b;
public static void SomeMethod ()
{
a = 5; // помилка
b = 10; // допустимо
}
}
```

Статичні члени класу

Раніше, щоб використовувати який-небудь клас, встановлювати і отримувати його поля, використовувати його методи, необхідно було створювати його об'єкт. Однак якщо певний клас має статичні методи, то, щоб отримати до них доступ, необов'язково створювати об'єкт цього класу.

Наприклад, створимо новий клас Account, що являтиме собою рахунок у банку:

```
class Account
{
public Account (decimal sum, decimal rate)
{
if (sum <MinSum) throw new Exception ("Неприпустима сума!");
Sum = sum; Rate = rate;
}
private static decimal minSum = 100; // мінімальна допустима сума для всіх рахунків
public static decimal MinSum
{
get {return minSum; }
set {if (value> 0) minSum = value; }
}
public decimal Sum {get; private set; } // сума на рахунку
public decimal Rate {get; private set; } // процентна ставка
// підрахунок суми на рахунку через певний період за певною ставкою
public static decimal GetSum (decimal sum, decimal rate, int period)
{
decimal result = sum;
for (int i = 1; i <= period; i ++)
result = result + result * rate / 100;
return result;
}
}
```

Змінна minSum, властивість MinSum, а також метод GetSum тут визначені з ключовим словом **static**, тобто вони є статичними.

Змінна minSum і властивість MinSum являють собою мінімальну суму, допустиму для створення рахунку. Ці показники належать не до якогось конкретного рахунку, а до всіх рахунків у цілому. Якщо змінимо цей показник для одного рахунку, то він повинен змінитися і для іншого. Тобто на відміну від властивостей Sum і Rate, які зберігають стан об'єкта, змінна minSum зберігає стан для всіх об'єктів певного класу.

Те ж саме з методом GetSum – він обчислює суму на рахунку через певний період за певною відсотковою ставкою для певної початкової суми. Виклик і результат цього методу не залежить від конкретного об'єкта або його стану.

Таким чином, змінні і властивості, які зберігають стан, загальний для всіх об'єктів класу, слід визначати як статичні. І методи, які визначають загальну для всіх об'єктів поведінку, також слід оголошувати як статичні.

Статичні члени класу є загальними для всіх об'єктів цього класу, тому до них треба звертатися по імені класу:

```
Account.MinSum = 560;
decimal result = Account.GetSum (1000, 10, 5);
```

При використанні статичних членів класу необов'язково створювати екземпляр класу, можна звернутися до них безпосередньо.

На рівні пам'яті для статичних полів буде створюватися ділянка в пам'яті, яка буде спільною для всіх об'єктів класу.

Нерідко статичні поля застосовуються для зберігання лічильників. Наприклад, нехай наявний клас User, і потрібно отримати лічильник, який дав би можливість дізнатися, скільки об'єктів User створено:

```
class User
{
private static int counter = 0;
public User ()
{
counter ++;
}

public static void DisplayCounter ()
{
Console.WriteLine ($ "Створено {counter} об'єктів User");
}
}
class Program
{
static void Main (string [] args)
{
User user1 = new User ();
User user2 = new User ();
User user3 = new User ();
User user4 = new User ();
User user5 = new User ();
User.DisplayCounter (); // 5
Console.Read ();
}
}
```

Статичний конструктор

Крім звичайних конструкторів у класу також можуть бути статичні конструктори. Статичні конструктори виконуються при першому створенні об'єкта певного класу або першому зверненні до його статичних членів (якщо такі є):

```
class User
{
static User ()
{
```

```
Console.WriteLine ("Створено першого користувача");
}
}
class Program
{
static void Main (string [] args)
{
User user1 = new User (); // тут спрацює статичний конструктор
User user2 = new User ();

Console.Read ();
}
}
```

Статичні класи

Статичні класи оголошуються з модифікатором static і можуть містити тільки статичні поля, властивості і методи. Наприклад, якби клас Account мав тільки статичні змінні, властивості і методи, то його можна було б оголосити статичним:

```
static class Account
{
private static decimal minSum = 100; // мінімальна допустима сума для всіх рахунків
public static decimal MinSum
{
get {return minSum; }
set {if (value> 0) minSum = value; }
}

// підрахунок суми на рахунку через певний період за певною ставкою
public static decimal GetSum (decimal sum, decimal rate, int period)
{
decimal result = sum;
for (int i = 1; i <= period; i ++)
result = result + result * rate / 100;
return result;}}
```

У C# показовим прикладом статичного класу є клас Math, який застосовується для різних математичних операцій.

Код:

```
// Conversion.cs
class Conversion {
 private static double rate = 27.5;
 private double money;
public double Money {
get {return money; }
```

```
set {if (value> 0) money = value; else money = 0; }}
public static double Rate{
get {return rate; } Set {if (value> 0) rate = value;}}
public static void RateGet (){
Console.WriteLine ("Курс: {0}", rate); }

public double resultMoney () {
return Money / rate;}}


// User.cs
class User {
private static int count = 0;
public User ()
{Count ++;
Console.WriteLine ("Object № {0} created!", Count);}

static User (){
Console.WriteLine ("First object created!", Count);}
public static void ShowCount ()
{Console.WriteLine ("Створено {0} об'єктів !!!", count); }}

// Program.cs
class Program {
static void Main (string [] args) {
Conversion ob1 = new Conversion ();
ob1.Money = 150;
Conversion.Rate = 22;
double res = ob1.resultMoney ();
Console.WriteLine ("res = {0}", res);

User u1 = new User ();
User u2 = new User (); User u3 = new User ();
User.ShowCount (); Console.ReadKey ();}}
```

Лекція 8. ПОКАЖЧИКИ НА БАЗОВИЙ КЛАС. ОПЕРАТОРИ IS ТА AS. УЗАГАЛЬНЕННЯ

У C# є можливість створення **масиву (або списку) покажчиків на базовий клас,** в якому елементами можуть бути об'єкти класу-спадкоємця. Наприклад, можемо створити масив об'єктів Риба, і елементами такого масиву будуть об'єкти класів Дельфін і Акула.

Хоча як елементи до цього списку додаємо об'єкти класів-спадкоємців Дельфін і Акула, будучи елементами списку покажчиків на базовий клас, ці об'єкти перетворюються на об'єкти базового класу, і маємо доступ тільки до тієї частини об'єктів, яка описана в базовому класі. Не можна створити масив об'єктів класу Акула і записати в нього об'єкти класу Риба.

Оператор is

Оператор is працює дуже просто – він перевіряє сумісність об'єкта з указаним типом (чи належить об'єкт певному класу). Оператор is повертає істину (true), якщо об'єкт належить класу. Істина буде також при перевірці сумісності об'єкта класу-спадкоємця і базового класу.

Оператор as

Замість явного приведення типів можна було використовувати оператор «as.(Dolphin) f» еквівалентно виразу «f as Dolphin». Різниця між оператором as і явним приведенням лише в тому, що в разі неможливості перетворення оператор as повертає null, тоді як явне приведення викидає виключення.

Приклад програми:

```
// Fish.cs
abstract class Fish{
 public string Name { get; set; }
 public Fish () {
 Name = "Fish";}
 public Fish (string Name) {
this.Name = Name;}
public abstract voidShowInfo (); }}
// Dolphin.cs
class Dolphin: Fish{
 public Dolphin (string name1): base(Name1) {}
 public override void ShowInfo () {
Console.WriteLine ("Ніжний ссавець!"); }
 public void PrintNameDolphin () {
Console.WriteLine ("Name dolphin: {0}", Name);}}
// Shark.cs
class Shark: Fish{
 public Shark (string name1): base(Name1) {}
 public override void ShowInfo () {
Console.WriteLine ("Небезпечний хижак!!!");}
public void PrintNameShark () {
Console.WriteLine ("Name shark: {0}", Name);}}

// Program.cs
class Program{
 static void Main (string[] Args) {
 Dolphin d1 = new Dolphin("Сніжка");
```

```
 Console.WriteLine (d1 is Dolphin);
 Console.WriteLine (d1 is Shark);
 Console.WriteLine (d1 is Fish);
List<Fish> List1 = new List<Fish> ();
 list1.Add (new Dolphin("Сніжка"));
 list1.Add (new Dolphin("Пломбір"));
 list1.Add (new Shark("Гострозуб"));

 foreach (Fish f in list1) {
if (f is Dolphin)
 / * ((Dolphin) f) .PrintNameDolphin ();
 else ((Shark) f) .PrintNameShark (); * /
 (f as Dolphin) .PrintNameDolphin ();
 else (f as Shark) .PrintNameShark ();
 f.ShowInfo ();}
 Console.ReadKey ();}}
```

UML-діаграма класів (рис. 5):

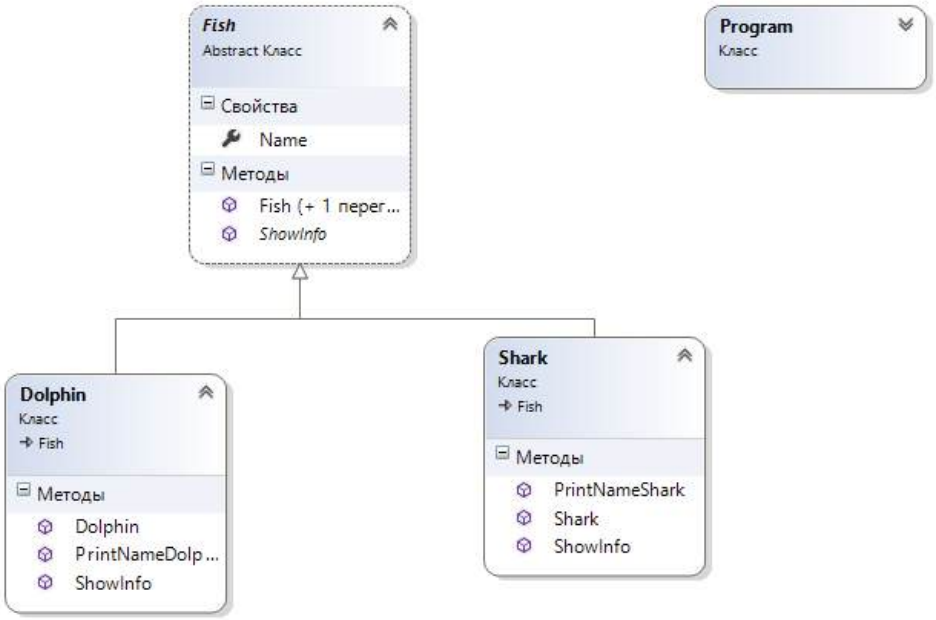

Рис. 5. UML-діаграма класів

Узагальнення

Узагальнені типи дозволяють вказати конкретний тип, який буде узятий. Тому визначимо клас Account як узагальнений. Кутові дужки в описі class Account <T> вказують, що клас є узагальненим, а тип T, узятий у кутові дужки, буде використовуватися цим класом. Необов'язково використовувати саме букву T, це може бути і будь-яка інша буква або набір символів. Причому зараз нам невідомо, що це буде за тип, це може бути будь-який тип. Параметр T у кутових дужках ще називається

універсальним параметром, тому що замість нього можна підставити будь-який тип.

Наприклад, замість параметра T можна використовувати об'єкт int, тобто число, що являє собою номер рахунку. Це також може бути об'єкт string або будь-який інший клас або структура. Оскільки клас Account є узагальненим, то при визначенні змінної після назви типу в кутових дужках необхідно вказати той тип, який буде використовуватися замість універсального параметра T. При спробі присвоїти значення властивості Id змінній іншого типу отримаємо помилку компіляції: Account <string> account2 = new Account <string> {Sum = 4000};

account2.Id = "4356";

int id1 = account2.Id; // помилка компіляції

Тим самим уникнемо проблем з еквідповідністю типів. Таким чином, використовуючи узагальнений варіант класу, знижуємо час на виконання і кількість потенційних помилок.

Іноді виникає необхідність присвоїти змінним універсальних параметрів деяке початкове значення. У цьому випадку нам треба використовувати оператор default (T). Він присвоює посилальним типам значення null, а значимим типам – значення 0.

Статичні поля узагальнених класів:

```
class Account <T>
{
public static T session;

public T Id {get; set; }
public int Sum {get; set; }
}
```

При типізації узагальненого класу певний тип буде створювати свій набір статичних членів. Тепер типізуємо клас двома типами int і string. Завдяки цьому для Account <string> і для Account <int> буде створена своя змінна session.

Узагальнення можуть використовувати кілька універсальних параметрів одночасно, які можуть представляти різні типи (використання декількох універсальних параметрів).

Узагальнені методи

Крім узагальнених класів можна також створювати узагальнені методи, які так само будуть використовувати універсальні параметри.

Наприклад:

```
class Program
{
```

```
private static void Main (string [] args)
{
int x = 7;
int y = 25;
Swap <int> (ref x, ref y);
Console.WriteLine ($ "x = {x} y = {y}"); // x = 25 y = 7

string s1 = "hello";
string s2 = "bye";
Swap <string> (ref s1, ref s2);
Console.WriteLine ($ "s1 = {s1} s2 = {s2}"); // s1 = bye s2 = hello

Console.Read ();
}
public static void Swap <T> (ref T x, ref T y)
{
T temp = x;
x = y;
y = temp;
}
}
```

Тут визначено узагальнений метод Swap, який приймає параметри за посиланням і змінює їх значення. При цьому в даному випадку не важливо, який тип представляють ці параметри.

У методі Main викликаємо метод Swap, типізуємо його певним типом і передаємо йому деякі значення.

Код програми:

```
class Account <T> {// T універсальний параметр
 public static T percent;
public T Id {get; set; }
public int Sum {get; set; }
public Account () {
 Id = default (T);
 Sum = 10;
 percent = default (T);}
public void Show () {
 Console.WriteLine ( "Id = {0}, Sum = {1}", Id, Sum);}}

class AccountNew <T, U>{
public T Id {get; set; }
public U Sum {get; set; }
public AccountNew () {
 Id = default (T);
 Sum = default (U);}
```

```
public void Show () {
 Console.WriteLine ( "Id = {0}, Sum = {1}", Id, Sum);}}

class Program {
 static void Main (string [] args) {
 Account <int> A1 = new Account <int> ();
 A1.Id = 18;
 A1.Show ();
 Account <double> .percent = 12.4;
 Account <string> A2 = new Account <string> ();
 A2.Id = "001";
 A2.Sum = 1000;
 A2.Show ();

AccountNew <string, double> AN = new AccountNew <string, double> ();
 AN.Id = "002";
 AN.Sum = 1200.5;
 AN.Show ();
AccountNew <int, string> AN2 = new AccountNew <int, string> ();
 AN2.Id = 8;
 AN2.Sum = "0020";
 AN2.Show ();
Console.ReadKey (); }}
```

Лекція 9. ІНТЕРФЕЙСИ

Інтерфейс – іменований набір сигнатур методів (string method (int a)) (іменований набір абстрактних членів [2]).

Це колекція загальнодоступних (а значить, не статичних) методів і властивостей, згрупованих для інкапсуляції конкретної функціональності.

Інтерфейс – це іменований набір сигнатур методів, властивостей, подій або індексаторів. Інтерфейси дозволяють визначити деякий функціонал, що не має конкретної реалізації. Потім цей функціонал реалізують класи, які застосовують ці інтерфейси. Клас або структура, які реалізують інтерфейс, повинні реалізувати члени цього інтерфейсу, зазначені в його визначенні.

Інтерфейс має важливу роль у системі ООП. Для його визначення використовується ключове слово interface. Як правило, назви інтерфейсів в C# починаються з великої літери «I», наприклад, IComparable, IEnumerable (так звана угорська нотація), однак це не обов'язкова вимога, а більше стиль програмування. Інтерфейси, як і класи, можуть містити властивості, методи і події, тільки без конкретної реалізації. У бібліотеках базових класів .NET надаються сотні готових типів інтерфейсів.

Завдяки підтримці інтерфейсів у C# може бути повною мірою реалізовано головний принцип поліморфізму: *один інтерфейс – безліч методів.*

Нижче наведена спрощена форма оголошення інтерфейсу:

```
interface ім'я {
 тип_повернення ім’я_методу_1 (список_параметрів);
тип_повернення ім’я_методу_2 (список_параметрів);
 // ...
 тип_повернення ім’я_методу_N (список_параметрів);
}
```

де ім'я – це конкретне ім'я інтерфейсу. В оголошенні методів інтерфейсу використовуються тільки їх тип_повернення і сигнатура. Вони, по суті, є абстрактними методами. Як пояснювалося вище, в інтерфейсі не може бути ніякої реалізації. Тому всі методи інтерфейсу повинні бути реалізовані в кожному класі, що включає в себе цей інтерфейс. У самому ж інтерфейсі методи неявно вважаються відкритими, тому доступ до них не потрібно вказувати явно.

Крім методів, в інтерфейсах можна також вказувати властивості, індексатори і події. Інтерфейси не можуть містити члени даних. У них не можна також визначити конструктори, деструктори або операторні методи. Крім того, жоден з членів інтерфейсу не може бути оголошений як static.

Як тільки інтерфейс буде визначено, він може бути реалізований в одному або декількох класах. Для реалізації інтерфейсу досить вказати його ім'я після імені класу, аналогічно базовому класу.

Нижче наведена загальна форма реалізації інтерфейсу в класі:

```
class ім'я_класу: ім’я_інтерфейсу {
 // тіло класу
}
```

де ім’я_інтерфейсу – це конкретне ім'я реалізованого інтерфейсу. Якщо вже інтерфейс реалізується в класі, то це повинно бути зроблено повністю. Реалізувати інтерфейс вибірково і тільки по частинах не можна.

Методи, що реалізують інтерфейс у класі, повинні бути оголошені як public. Справа в тому, що в самому інтерфейсі ці методи неявно сприймаються як відкриті, тому їх реалізація також повинна бути відкритою. Крім того, тип і сигнатура реалізованого методу повинні точно відповідати типу і сигнатурі, зазначеним у визначенні інтерфейсу.

Приклад реалізації інтерфейсу:

```
namespace Interface_4 {
 interface IShape
 {
 double square ();
 }
}
class Circle: IShape {
```

```
 public double r {get; set;}
 public Circle (double r1) {r = r1;}
 public double square ()
 {
 return 3.14 * r;
 }}

class Triangle: IShape
 {
 public double a {get; set;}
 public double b { get; set;}
 public double c { get; set;}
 public Triangle (double a, double b, double c)
 {
 this.a = a;
 this.b = b;
 this.c = c;
 }
 public double square ()
 { double p = (a + b + c) / 2;
 return Math.Sqrt (p * (p - a) * (p - b) * (p - c));
 }
 public void Display ()
 {
 Console.WriteLine ("A = {0}, b = {1}, c = {2}", A, b, c);
 }
 }

class Trapezium: IShape {
 public double a {get; set;}
 public double b { get; set;}
 public double h { get; set;}
 public Trapezium (double a, double b, double h)
 {
 this.a = a;
 this.b = b;
 this.h = h;
 }
 public double square () {
 return (A + b) / 2 * h;
 }}


class geometric_figures {
```

```
 public void view_square (IShape[] Arr) {
 for (inti = 0; i <arr.Count (); i ++)
Console.WriteLine ("Square is equal to {0: F3}", Arr [i] .square ()); }}
---------------------------------------------- ------------------------------
class Program {
 static void Main (string[] Args) {
 geometric_figures geometric = new geometric_figures();
 IShape[] Shapes = new IShape [] {new Circle(13), new Trapezium(10.5, 5.5, 8), new Triangle(3, 5, 7)};
 geometric.view_square (shapes);}
```

Для похідного класу ім'я базового класу пишеться першим, а потім, якщо необхідно, інтерфейси. У класі повинні бути реалізовані всі члени інтерфейсу.

У C# допускається оголошувати змінні посилального інтерфейсного типу, тобто змінні посилання на інтерфейс. Така змінна може посилатися на будь-який об'єкт, який реалізує її інтерфейс. При виклику методу для об'єкта за допомогою інтерфейсного посилання виконується його варіант, реалізований у класі об'єкта. Цей процес аналогічний застосуванню посилання на базовий клас для доступу до об'єкта похідного класу.

Змінній посилання на інтерфейс доступні тільки методи, оголошені в її інтерфейсі.

```
class Program {
 static void Main (string[] Args) {
 ////////////////////////////////////////////////// ///////////
 IShape shape1 = new Triangle(10, 12, 13);
 Console.WriteLine ("Square triangle is equal to {0: f3}", Shape1.square ());
 ((Triangle) Shape1) .Display ();
 ////////////////////////////////////////////////// //////////
```

Тому інтерфейсне посилання не можна використовувати для доступу до будь-яких інших змінних і методів, які не підтримуються об'єктом класу, що реалізує цей інтерфейс.

Ключове слово as

Визначити, чи підтримує певний тип той чи інший інтерфейс, можна за допомогою ключового слова as. Якщо об'єкт вдається інтерпретувати як зазначений інтерфейс, то повертається посилання на інтерфейс, а якщо не вдається, то посилання – null. Отже, перш ніж продовжити код, необхідно передбачити перевірку на null.

У попередньому прикладі, в методі Main () можна додати таку перевірку:

```
class Program {
 static void Main (string[] Args) {
 Trapezium tr1 = new Trapezium(6, 11, 19);
 IShape obj1 = tr1 as IShape;
```

```
if (Obj1! = null)
{
Console.WriteLine ("Тип Trapezium підтримує інтерфейс IShape");
Console.WriteLine ("Square Trapezium is equal to {0: f3}", Obj1.square
());
}
else Console.WriteLine ("No!");}
```

Зверніть увагу, що в разі застосування ключового слова as використовувати логіку try/catch немає необхідності, оскільки повернення посилання, відмінного від null, означає, що виклик здійснюється з використанням дійсного посилання на інтерфейс.

Ключове слово is

Перевірити, чи був реалізований потрібний інтерфейс, можна також за допомогою ключового слова is. Якщо об'єкт, що перевіряється, не сумісний із зазначеним інтерфейсом, повертається значення false, а якщо сумісний, то можна спокійно викликати члени цього інтерфейсу без застосування логіки try/catch.

```
class Program {
 static void Main (string[] Args) {

 ////////////////////////////////////////////////// ////////
 Circle c1 = new Circle(8);
 if (c1 is IShape)
 {
 Console.WriteLine ("Тип Circle підтримує інтерфейс IShape");
 Console.WriteLine ("Square Circle is equal to {0: f3}", C1.square ());
 }
 else Console.WriteLine ("No!");}
```

Інтерфейсні властивості

Аналогічно методам, властивості вказуються в інтерфейсі взагалі без тіла. Нижче наведена загальна форма оголошення інтерфейсної властивості:

```
// Інтерфейсна властивість
тип ім'я {get; set;}
```

Очевидно, що у визначенні інтерфейсних властивостей, доступних тільки для читання або тільки для запису, повинен бути присутній єдиний аксесор: get або set відповідно.

Оголошення властивості в інтерфейсі дуже схоже на оголошення автоматично реалізованих властивостей класів, але між ними все ж є відмінність. При оголошенні в інтерфейсі властивість не стає автоматично реалізованою. У цьому випадку вказується тільки ім'я і тип властивості, а його реалізація надається в кожному класі. Крім того, при оголошенні властивості в інтерфейсі забороняється вказувати модифікатори доступу

для аксесора. Наприклад, аксесор set не може бути вказаний в інтерфейсі, як private.

Інтерфейсні індексатори

В інтерфейсі можна також вказувати індексатори. Нижче наведена загальна форма оголошення інтерфейсного індексатора:

```
// Інтерфейсний індексатор
тип_елемента this [int індекс] {
get;
set;
}
```

Як і раніше, в оголошенні інтерфейсних індексаторів, доступних тільки для читання або тільки для запису, повинен бути присутній єдиний аксесор: get або set відповідно.

```
using System;
namespace ConsoleApplication1
{
 interface IUserInfo
 {
 string Name
 {
 get;
 set;
 }

 string this[int index]
 {
 get;
 set;
 }
 }

 class UI : IUserInfo
 {
 string myName;

 public string Name
 {
 set
 {
 myName = value;
 }

 get
 {
```

```
return myName;
}
}

public string this[int index]
{
set {MyName = value; }
get { returnmyName; }
}
}

class Program
{
static void Main()
{
UI user1 = new UI ();
user1.Name = "Alexandr";
user1 [5] ="Dmitryi";
user1 [10] ="Alexey";

Console.ReadLine ();
}
}
}
```

Успадкування інтерфейсів

Один інтерфейс може успадковувати інший. Синтаксис успадкування інтерфейсів такий же, як і у класів. Коли в класі реалізується один інтерфейс, що успадковує інший, в ньому повинні бути реалізовані всі члени, визначені в ланцюжку успадкування інтерфейсів.

Таким чином, інтерфейси можуть бути організовані в ієрархії. Як і в ієрархії класів, в ієрархії інтерфейсів, коли якийсь інтерфейс розширює існуючий, він успадковує всі абстрактні члени свого батька (або батьків). Звичайно, на відміну від класів, похідні інтерфейси ніколи не успадковують саму реалізацію. Замість цього вони просто розширюють власне визначення за рахунок додавання додаткових абстрактних членів.

Використовувати ієрархію інтерфейсів може бути зручно, коли потрібно розширити функціональність певного інтерфейсу без порушення вже існуючих кодових баз.

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

namespace inrerfase_2
```

```
{
 interface IProductObject
 {
 float GetProductionCost ();
 }
}
////////////////////////////////////////////// ////////////////////////////
 interface ISpeedObject
 {
 float GetSpeed ();
 }
////////////////////////////////////////////// ////////////////////////////
interface IShip: IProductObject
 {
 double Move (double distance);
 }
////////////////////////////////////////////// ////////////////////////////
class TransportShip: IShip, ISpeedObject
 {
public double massaShip {get; set; }
public double maxMassaGr {get; set; }
public float Percentage {get; set; }
public float cost {get; set; }
public float avgSpeed {get; set; }

 public double Move (double distance)
 {
 return (maxMassaGr + massaShip) / 2 * distance;
 }
 public float GetProductionCost ()
 {
 return cost * Percentage;
 }

 public float GetSpeed ()
 {
 return avgSpeed * (float) maxMassaGr;
 }
 }
////////////////////////////////////////////// ////////////////////////////
class House: IProductObject
 {
public float cost {get; set; }
 public float GetProductionCost ()
```

```
{
return cost * 0.3f;
}
}
////////////////////////////////////////////// ////////////////////////////
class Program
{
static void Main (string [] args)
{
TransportShip ts = new TransportShip ();
ts.massaShip = 1500;
ts.maxMassaGr = 568;
ts.Percentage = 12.5f;
ts.cost = 20000f;
ts.avgSpeed = 70f;
Console.WriteLine ("ts.GetProductionCost () {0}", ts.GetProductionCost
());
Console.WriteLine ("ts.GetSpeed () {0}", ts.GetSpeed ());
IProductObject h1 = new House ();
((House) h1) .cost = 345;
Console.WriteLine ("h1.GetProductionCost () {0}",
h1.GetProductionCost());
Console.ReadKey ();
}
}
```

На відміну від класів, один інтерфейс може розширювати відразу кілька базових інтерфейсів, що дозволяє проектувати дуже потужні й гнучкі абстракції.

Явна реалізація інтерфейсу

Один клас або структура може реалізувати будь-яку кількість інтерфейсів. Через це завжди існує ймовірність реалізації інтерфейсів з членами, що мають ідентичні імена, і, отже, виникає необхідність в усуненні конфліктів на рівні імен. При реалізації члена інтерфейсу є можливість вказати його ім'я повністю разом з ім'ям самого інтерфейсу. В цьому випадку виходить явна реалізація члена інтерфейсу, або просто **явна реалізація.**

Коли один інтерфейс успадковує інший, то в похідному інтерфейсі може бути оголошений член, що приховує член з аналогічним ім'ям в базовому інтерфейсі. Таке приховування імен відбувається в тому випадку, якщо член у похідному інтерфейсі оголошується таким же чином, як і в базовому інтерфейсі. Але якщо не вказати в оголошенні члена похідного інтерфейсу ключове слово new, то компілятор видасть відповідне попередження.

Для явної реалізації інтерфейсного методу можуть бути дві причини. По-перше, коли інтерфейсний метод реалізується із зазначенням його повного імені, то такий метод виявляється доступним не за допомогою об'єктів класу, що реалізує цей інтерфейс, а за інтерфейсним посиланням. Отже, явна реалізація дозволяє реалізувати інтерфейсний метод таким чином, щоб він не став відкритим членом класу, який надає його реалізацію. І по-друге, в одному класі можуть бути реалізовані два інтерфейси з методами, оголошеними з однаковими іменами та сигнатурами. Але неоднозначність у такому випадку усувається завдяки зазначенню в іменах цих методів їх відповідних інтерфейсів. Розглянемо кожну з цих двох можливостей явної реалізації на прикладі:

```
using System;
namespace ConsoleApplication1
{
 public interface IName
 {
 void WriteName ();
 }
 public interface INameFamily
 {
 // Оголошуємо в цьому інтерфейсі такий же метод
 void WriteName ();
 void WriteFamily ();
 }
 public interface IUserInfo: INameFamily
 {
 // Обов'язково потрібно вказати ключове слово new
 // щоб не приховувалися методи базового інтерфейсу
 new void WriteName ();
 void WriteUserInfo ();
 }

 // Клас, який реалізує два інтерфейси
 class UserInfo: IUserInfo, IName
 {
 string ShortName, Family, Name;

 public UserInfo (string Name, string Family, string ShortName)
 {
 this.Name = Name;
 this.Family = Family;
 this.ShortName = ShortName;
 }
```

```
// Використовуємо явну реалізацію інтерфейсів
// для виключення неоднозначності
void IName.WriteName ()
{
Console.WriteLine ("Коротке ім'я: " + ShortName);
}
void INameFamily.WriteFamily ()
{
Console.WriteLine ("Прізвище:" + Family);
}
void INameFamily.WriteName ()
{
Console.WriteLine ("Повне ім'я: " + Name);
}
void IUserInfo.WriteName () {}
public void WriteUserInfo ()
{
UserInfo obj = new UserInfo (Name, Family, ShortName);
// Для використання закритих методів необхідно
// створити інтерфейсне посилання
IName link1 = (IName) obj;
link1.WriteName ();
INameFamily link2 = (INameFamily) obj;
link2.WriteName ();
link2.WriteFamily ();
IUserInfo link3 = (IUserInfo) obj;
link3.WriteName ();
}
}
class Program
{
static void Main ()
{
UserInfo obj = new UserInfo (Name: "Alexandr", ShortName: "Alex",
Family: "KYRICH");
obj.WriteUserInfo ();
Console.ReadLine ();
}
}
}
```

Важливо усвідомити, що інтерфейси є фундаментальним компонентом .NET Framework. Якого б типу програма не розроблялася (веб-додаток, додаток з настільним графічним інтерфейсом, бібліотека

доступу до даних і т. ін.), робота з інтерфейсами буде обов'язковою частиною цього процесу.

Підводячи підсумок, відзначимо, що інтерфейси можуть приносити надзвичайну користь у таких випадках:

- за наявності єдиної ієрархії, в якій тільки якийсь набір похідних типів підтримує загальну поведінку;
- при необхідності моделювати загальну поведінку, яка повинна зустрічатися в декількох ієрархіях, що не мають загального батьківського класу крім System.Object.

Сортування об'єктів. Інтерфейс IComparable () з параметром

Більшість вбудованих у .NET класів колекцій і масиви підтримують сортування. За допомогою одного методу, що, як правило, називається Sort (), можна відразу впорядкувати за зростанням весь набір даних. Наприклад:

```
int [] numbers = new int [] {97, 45, 32, 65, 83, 23, 15};
Array.Sort (numbers);
foreach (int n in numbers)
Console.WriteLine (n);
```

Однак метод Sort за замовчуванням працює тільки для наборів примітивних типів, таких як int або string. Для сортування наборів складних об'єктів застосовується інтерфейс IComparable. Він має всього один метод:

```
public interface IComparable
{int CompareTo (object o);}
```

Метод CompareTo призначений для порівняння поточного об'єкта з об'єктом, який передається як параметр object o. На виході він повертає ціле число, яке може мати одне з трьох значень:

- менше нуля (значить, поточний об'єкт повинен знаходитися перед об'єктом, який передається як параметр);
- дорівнює нулю (значить, обидва об'єкти рівні);
- більше нуля (значить, поточний об'єкт повинен знаходитися після об'єкта, переданого як параметр).

Інтерфейс IComparable доступний у двох формах: узагальненій і неузагальненій. Незважаючи на схожість застосування обох форм, між ними є невеликі відмінності.

Приклад програмного коду:

```
class Person: IComparable<Person> {
 public string Name { get; set;}
 public int Age { get; set;}
 public Person (string Name, int Age) {
 this.Name = Name;
 this.Age = Age; }
 / * Public int CompareTo (Person p) // (object o) {
```

```
return this.Name.CompareTo (p.Name); } * /
 / * Public int CompareTo (Person p) // (object o) {
return this.Age.CompareTo (p.Age); } * /
 public int CompareTo (Person p) // (object o)
 {
 if (Age> p.Age) return 1;
 if (Age <p.Age) return -1;
 else return0; }}
---------------------------------------------- -------------------------
 class Program
 {
 static void Main ()
 {

 List<Person> People = new List<Person> ();
 people.Add (new Person("David", 26));
 people.Add (new Person("Oleg", 20));
 people.Add (new Person("Ivan", 23));
 people.Sort ();
 foreach (Person person1 in people)
 Console.WriteLine ("Name: {0}, Age: {1}", Person1.Name, person1.Age);
 Console.ReadKey (); }}
```

Лекція 10. ДЕЛЕГАТИ

Делегат – це тип об'єкта, що дозволяє зберігати посилання на функції. Делегат дозволяє викликати функції, на які він посилається. Оголошуються делегати з ключовим словом delegate.

Приклад:

```
namespace delegate1
{
 public partial class Form1 : Form
 {
 public Form1 ()
 {
 InitializeComponent ();
 }
 delegate double Operations(double param1, double param2);
 delegate void GetMessage();
 delegate double Oper(char ch, double params1, double params2);
 delegate bool IsEqual(int x);

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

```
GetMessage del1;
if (DateTime.Now.Hour <12)
del1 = GoodMorning; // Надаємо цій змінній адресу методу
else if (DateTime.Now.Hour> 12 && DateTime.Now.Hour <18)
del1 = GoodDay;
else del1 = GoodEvening;
del1 ();
Operations op1 = new Operations(Sum);
textBox1.Text = op1 (13, 2) .ToString ();
op1 = new Operations(Del);
textBox2.Text = op1 (19, 7) .ToString ();
}
public void GoodMorning ()
{
MessageBox.Show ("Good Morning");
}
public void GoodDay ()
{
MessageBox.Show ("Good Day");
}

public void GoodEvening ()
{
MessageBox.Show ("Good Evening");
}

double Sum (double params1, double params2)
{
return params1 + params2;
}

double Prz (double params1, double params2)
{
return params1 * params2;
}

double Del (double params1, double params2)
{
return params1 / params2;
}
double Minus (double params1, double params2)
{
return params1 - params2;
}
```

```
 }
}
```

Для того щоб скористатися делегатом, необхідно створити його екземпляр і задати імена методів, на які він буде посилатися. При виклику екземпляра делегата викликаються всі задані в ньому методи.

Делегати застосовуються в основному для таких цілей:

• отримання можливості визначати метод, що викликається, не при компіляції, а динамічно під час виконання програми;

• забезпечення зв'язку між об'єктами по типу «джерело – спостерігач»;

• створення універсальних методів, в які можна передавати інші методи (підтримка механізму зворотних викликів).

Делегат може зберігати посилання на кілька методів і викликати їх по черзі за умови, що сигнатури всіх методів повинні збігатися.

Передача делегатів у методи

Делегат можна передавати в методи як параметр. Таким чином, забезпечується *функціональна параметризація:* в метод можна передавати не тільки різні дані, але і різні функції їх обробки. Функціональна параметризація застосовується для створення універсальних методів і забезпечення можливості зворотного виклику. Найпростішим прикладом *універсального методу* може бути метод виведення таблиці значень функції, в який передається діапазон значень аргументу, крок його зміни і вид функції обчислення. Цей приклад наводиться далі.

Зворотний виклик **(Callback)** являє собою виклик функції, що передається в іншу функцію як параметр. Механізм зворотного виклику широко використовується в програмуванні. Наприклад, він реалізується в багатьох стандартних функціях Windows.

Приклад 1:

```
{ public delegate double func(double arg);
 class newClass
 {
 public void Run1 (func f, double xn, double xk, double h)
 {
 Console.WriteLine ("X Y");
 while (Xn <= xk)
 {
 Console.WriteLine ("{0} {1}", Xn, f (xn));
 xn + = h;
 }
 }
 public double Simple1 (double x)
 {
 return x * x - 1;
 }}}
```

```
class Program
{
static void Main (string[] Args)
{
newClass class1 = new newClass();
Console.WriteLine ("Y = x ^ 2-1");
class1.Run1 (class1.Simple1, -11, 19, 1);
Console.WriteLine ("-----------------");
Console.WriteLine ("Y = sin (x)");
class1.Run1 (Math.Sin, 0, 10, 2);
// class1.Run1 (new func (Math.Sin), 0, 10, 2); //альтернативний варіант
}
```

Приклад 2:

```
class myPrinter
{
public delegate void StringPrinter (string str);

public void printString (string str,StringPrinter printer)
{Printer (str);}

public static void priner1 (string str)
{Console.WriteLine ("{0}", Str);}

public static void priner2 (string str)
{Console.WriteLine ("{0}!", Str);}}

class Program
{
static void Main (string[] Args)
{
string[] Lines = {"Array", "Of", "String"};
myPrinter p = new myPrinter();
myPrinter.StringPrinter sp = new myPrinter.StringPrinter(myPrinter.priner1);
foreach (string s in Lines)
p.printString (s, sp);
Console.WriteLine ("-----------------");
sp = myPrinter.priner2;
foreach (string s in Lines)
p.printString (s, sp);
}
```

Події

Подія – це елемент класу, що дозволяє йому посилати іншим об'єктам повідомлення про зміну свого стану. При цьому для об'єктів, які є спостерігачами події, активізуються методи-обробники цієї події. Обробники повинні бути зареєстровані в об'єкті-джерелі події. Механізм подій можна також описати за допомогою моделі публікація – підписка: один клас, який є *відправником* (Sender) повідомлення, публікує події, які він може ініціювати, а інші класи, які є *одержувачами* (Receivers) повідомлення, підписуються на отримання цих подій.

Події побудовані на основі делегатів: за допомогою делегатів викликаються методи-обробники подій. Тому *створення події* в класі складається з:

• опису делегата, що задає сигнатуру обробників подій;

• опису події;

• опису методу (методів), що ініціюють подію.

Синтаксис події схожий на синтаксис делегата:

[Атрибути] [специфікатор] event тип ім’я_події

Для подій застосовуються *специфікатори* new, public, protected, internal, private, static, virtual, sealed, override, abstract і extern, які вивчалися під час розгляду методів класів. Наприклад, так само як і методи, подія може бути статичною (static), тоді вона пов'язана з класом у цілому, або звичайною – в цьому випадку вона пов'язана з екземпляром класу. *Тип* події – це тип делегата, на якому застосовується подія.

Зв'язок з делегатом означає, що метод, який обробляє ці події, повинен приймати ті ж параметри, що і делегат, і повертати той самий тип.

Приклад опису делегата і відповідної йому події:

```
public delegate void Del (object o); // оголошення делегата
class A
{
public event Del Oops; // оголошення події
}
```

Події обробляються в класах-одержувачах повідомлення. Для цього в них описуються методи – обробники подій, сигнатура яких відповідає типу делегата.

Коли ж подія відбувається, викликаються всі зареєстровані обробники цієї події. Обробники подій зазвичай представлені делегатами.

Зовнішній код може працювати з подіями єдиним чином: додавати обробники в список або видаляти їх, оскільки поза класом можуть використовуватися тільки операції + = і - =. Тип результату цих операцій – void, на відміну від операцій складного присвоювання для арифметичних типів. Іншого способу доступу до списку обробників немає. Усередині класу, в якому описано подію, до нього можна звертатися, як до

звичайного поля, що має тип делегата: використовувати операції відношення, привласнення і т. д. Значення події за замовчуванням – null.

У бібліотеці .NET описана величезна кількість стандартних делегатів, призначених для реалізації механізму обробки подій. Більшість цих класів оформлено по одним і тим же правилам:

• ім'я делегата закінчується суфіксом EventHandler;

• делегат отримує два параметри:

1) перший параметр задає джерело події і має тип object;

2) другий параметр задає аргументи події і має тип EventArgs або похідний від нього.

Якщо обробникам події потрібна специфічна інформація про подію, то для цього створюють клас, похідний від стандартного класу EventArgs, і додають у нього необхідну інформацію. Якщо делегат не використовує таку інформацію, можна не описувати делегата і власний тип аргументів, а обійтися стандартним класом делегата System.EventHandler. Ім'я обробника події прийнято складати з префікса On та імені події.

Приклад 1

```
class UserInfo
{
public string login { get; set; }
public string password { get; set; }
public int count;
public UserInfo ()
{
login = "Name";
password = "******";
count = 0;
}
// Обробник події
public void UserInfoHandler ()
{
count ++;
Console.WriteLine ("Обробка події {0} -я \ n", Count);
Console.WriteLine ("Login: {0} \ nPassword: {1} \ n", Login, password);}}}
////////////////////////////////////////////////// //////////////////////////////////////////
public delegate void del1();
class Metod_Event
{
public event del1 Event1;
// Використовуємо метод для запуску події
public void OnEvent (string str)
{
Console.WriteLine (str);
Event1 ();
```

```
}}}
////////////////////////////////////////////////// ////////////////////////////////
class Program
{
static void Main (string[] Args)
{

Metod_Event ev = new Metod_Event();
UserInfo UI = new UserInfo();
// Додаємо обробник події
ev.Event1 + = UI.UserInfoHandler;
// Запустимо подію
ev.OnEvent ("Подія сталася!");
Console.ReadKey ();
}}
```

Приклад 2

```
public delegate void notification();

class source //// клас-джерело
{
public event notification Event1;
public void initiation () // метод, який ініціює подію
{
Console.WriteLine ("Error");
if (Event1! = null) Event1 ();
}
}
class receiver1 // клас-спостерігач
{
public void response1 ()// реакція на подію джерела
{ Console.WriteLine ("Identified error!"); }
}
class receiver2 // клас-спостерігач
{
public static void response2 ()// реакція на подію джерела
{ Console.WriteLine ("The error is eliminated!"); }
}
class Program
{
static void Main (string[] Args)
{
source source1 = new source();
receiver1 r1 = new receiver1();
```

```
 // додавання обробників до події
source1.Event1 + = new notification(R1.response1);
source1.Event1 + = new notification(receiver2.response2);
 source1.Event1 + = delegate
 {
 Console.WriteLine ("Data updated !!!");
 };
 source1.initiation (); // ініціювання події
 ////////////////////////////////////////////////// ////////////////////
 Console.ReadKey ();
 }}}
```

На рис. 6 наведено результат виконання прикладів 1 та 2

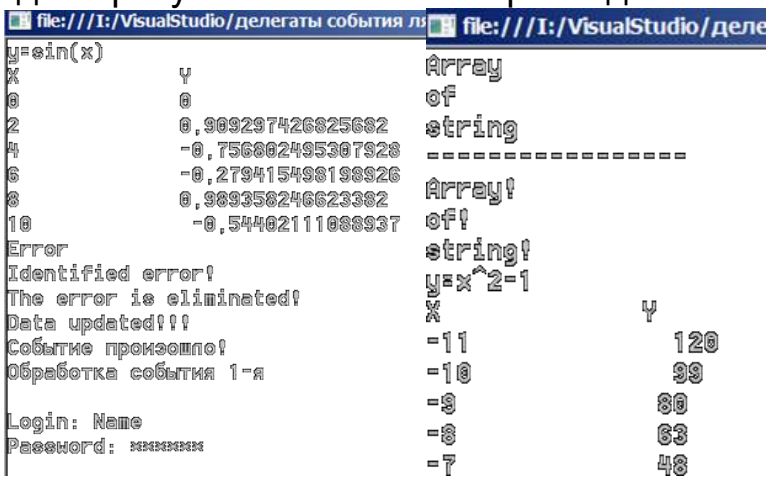

Рис. 6. Результат виконання прикладів 1 та 2

Анонімні методи

З делегатами тісно пов'язане поняття анонімних методів. Анонімні методи подають скорочений запис методів.

Приклад:

```
 private void button1_Click (object sender, EventArgs e)
 Oper op_new =delegate(char ch, double params1, double params2)
 {
 double result = 0;
 switch (Ch)
 {
 case '+': Result = params1 + params2; break;
 case '-': Result = params1 - params2; break;
 case '/': Result = params1 / params2; break;
 case '*': Result = params1 * params2; break;
 }
 return result;
 };
 textBox3.Text = op_new ('*', 4, 10) .ToString ();
}
```

Лямбди

Лямбда-вирази подають спрощений запис анонімних методів. Лямбда-вирази дозволяють створити ємкі лаконічні методи, які можуть повертати деяке значення і які можна передати як параметри в інші методи.

Лямбда-вирази мають такий синтаксис: зліва від лямбда-оператора => визначається список параметрів, а праворуч – блок виразів, що використовує ці параметри:

(Список_параметрів) => вираз.

Приклад 1:

```
class Program
{
delegate int Square (int x); // оголошуємо делегат, що приймає int і повертає int
static void Main (string [] args)
{
Square squareInt = i => i * i; // об'єкту делегата присвоюється лямбда-вираз
int z = squareInt (6); // використовуємо делегат
Console.WriteLine (z); // виводить число 36
Console.Read ();
}
}
```

Тут i => i * i являє собою лямбда-вираз, де i – це параметр, а i * i – вираз.

При використанні треба враховувати, що кожен параметр у лямбда-виразі неявно перетворюється у відповідний параметр делегата, тому типи параметрів повинні бути однаковими. Крім того, кількість параметрів повинна бути такою ж, як і у делегата. І значення лямбда-виразів має бути таким же, що і у делегата.

Як і делегати, лямбда-вирази можна передавати як параметри методу. Нерідко лямбда як параметри і використовуються, що досить зручно:

```
class Program
{
delegate bool IsEqual (int x);
static void Main (string [] args)
{
int [] integers = {1, 2, 3, 4, 5, 6, 7, 8, 9};
// знайдемо суму чисел більше 5
int result1 = Sum (integers, x => x> 5);
Console.WriteLine (result1); // 30
// знайдемо суму парних чисел
int result2 = Sum (integers, x => x% 2 == 0);
Console.WriteLine (result2); // 20
```

```
Console.Read ();
}
private static int Sum (int [] numbers, IsEqual func)
{
int result = 0;
foreach (int i in numbers)
{
if (func (i))
result + = i;
}
return result;
}
}
```

Метод Sum приймає як параметри масив чисел і делегат IsEqual і повертає суму чисел масиву у вигляді об'єкта int. У циклі проходимо по всім числам і знаходимо їх суму. Причому просумовуємо тільки ті числа, для яких делегат IsEqual func повертає true. Тобто делегат IsEqual тут фактично задає умову, якій повинні відповідати значення масиву. Але на момент написання методу Sum нам невідомо, що це за умова.

При виклику методу Sum йому передається масив і лямбда-вираз:

```
int result1 = Sum (integers, x => x> 5);
```

Тобто параметр «x» тут являтиме собою число, яке передається в делегат:

```
if (func (i)).
```

Вираз x>5 є умовою, якій має відповідати число. Якщо число відповідає цій умові, то лямбда-вираз повертає true, а передане число просумовується з іншими числами.

Подібним чином працює другий виклик методу Sum, тільки тут уже йде перевірка числа на парність, тобто якщо залишок від ділення на 2 дорівнює нулю: int result2 = Sum (integers, x => x% 2 == 0).

Приклад 2:

```
delegate void GetMessage();

private void button2_Click (object sender, EventArgs e)
{
GetMessage del2;
del2 = () => MessageBox.Show ("OK!!!");
del2 ();
}
```

Приклад 3:

```
delegate double Oper(char ch, double params1, double params2);

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

```
    {
Oper op_new = (ch, params1, params2) => // лямбда-вираз
    {
    double result = 0;
    switch (Ch)
    {
    case '+': Result = params1 + params2; break;
    case '-': Result = params1 - params2; break;
    case '/': Result = params1 / params2; break;
    case '*': Result = params1 * params2; break;
    }
    return result;
    };
    textBox3.Text = op_new ('*', 4, 10) .ToString ();
```

Лекція 11. ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ ДЛЯ ВИРОБНИЧИХ ПРОЦЕСІВ

Інформаційна технологія – це процес, що використовує сукупність засобів і методів збору, обробки і передачі даних для отримання інформації нової якості про стан об'єкта, процесу або явища. Мета інформаційної технології – виробництво інформації для її аналізу людиною і прийняття на його основі рішення щодо виконання якої-небудь дії.

Інформаційна система (ІС)

Інформаційною системою (ІС) називають сукупність взаємопов'язаних апаратно-програмних засобів для автоматизації накопичення і обробки інформації. В інформаційну систему дані надходять від джерела інформації. Ці дані відправляються на зберігання або підлягають у системі деякій обробці і потім передаються споживачеві.

Однією зі складових ІС або інформаційної технології є БД.

База даних (англ. Database), (БД) – сукупність описів об'єктів реального світу, певним чином структурованих і пов'язаних між собою, актуальних для конкретної прикладної області і представлених на машинозчитуваних носіях у формі, придатній для застосування ІТ. Організація даних у БД вимагає попереднього вибору і побудови моделі даних, призначення якої – систематизація інформації та відображення її властивостей (за змістом, структурою, обсягом зв'язків, динамікою, джерелами і т. д.).

Банк даних – автоматизована інформаційна система централізованого зберігання і колективного використання даних. До складу банку даних входять одна або кілька баз даних, довідник баз даних, СКБД, а також бібліотеки запитів і прикладних програм.

Модель даних

У класичній теорії баз даних модель даних є формальною теорією подання і обробки даних у системі керування базами даних (СКБД), яка включає, щонайменше, три аспекти:

1) аспект структури: методи опису типів логічних структур даних у базі даних;

2) аспект маніпуляції: методи маніпулювання даними;

3) аспект цілісності: методи опису і підтримки цілісності бази даних.

Аспект структури визначає, що логічно являє собою база даних, аспект маніпуляції визначає способи переходу між станами бази даних (тобто способи модифікації даних) і способи отримання даних із бази даних, аспект цілісності визначає засоби описів коректних станів бази даних.

Кожна БД і СКБД будується на основі певної явної або неявної моделі даних. Усі СКБД, побудовані на одній і тій же моделі даних, відносять до одного типу. Наприклад, основою реляційних СКБД є реляційна модель даних, мережевих СКБД – мережева модель даних, ієрархічних СКБД – ієрархічна модель даних і т. д.

Модель даних – це сукупність взаємопов'язаних структур даних і операцій над цими структурами. Для розміщення однієї і тієї ж інформації у внутрішній сфері можуть бути використані різні структури і моделі даних. Це залежить від користувача, від технічного і програмного забезпечення, визначається складністю автоматизованих завдань і обсягом інформації.

За структурою організації інформації в БД розрізняють такі моделі баз даних: ієрархічна, мережева і реляційна.

Ієрархічна модель бази даних. Ця модель являє собою структуру даних, які впорядковані від загального до конкретного; нагадує «дерево» (граф), тому має такі ж параметри: рівень, вузол, зв'язок. Модель працює за таким принципом: кілька вузлів нижчого рівня з'єднуються за допомогою зв'язку з одним вузлом вищого рівня. Ієрархічна модель бази даних має такі властивості: кілька вузлів нижчого рівня пов'язані тільки з одним вузлом вищого рівня; дерево ієрархії має тільки одну вершину, що не підлягає іншій; кожен вузол має власну назву, є тільки один маршрут від вершини дерева (кореневого вузла) до будь-якого вузла структури.

Мережева модель бази даних. Загальним виглядом вона схожа на ієрархічну. Має такі самі складові структури, відрізняється характером відносин між ними. Між елементами структури довільна, необмежена кількість елементів – зв'язків.

Реляційна модель бази даних (назва походить від латинського слова relatio – відношення). Модель побудована на взаємовідносинах між складовими структури. Являє собою одну таблицю або сукупність взаємопов'язаних двовимірних таблиць. Реляційна модель створена на основі двовимірної таблиці.

Класифікація баз даних

Найпростіші типи баз даних

1. Прості структури даних.

Перший і найпростіший спосіб зберігання даних – текстові файли. Метод застосовується і сьогодні для роботи з невеликими обсягами інформації. Для поділу полів використовується спеціальний символ: кома або крапка з комою в csv-файлах набору даних, двокрапка або пробіл в nix-подібних системах.

Особливості:

- обмежений тип і рівень складності інформації, що зберігається;
- важко встановити зв'язок між компонентами даних;
- відсутність функцій паралелізму;
- практичні тільки для систем з незначними вимогами до читання і запису;
- використовуються для зберігання конфігураційних даних;
- немає необхідності в сторонньому програмному забезпеченні.

2. Ієрархічні бази даних (рис. 7).

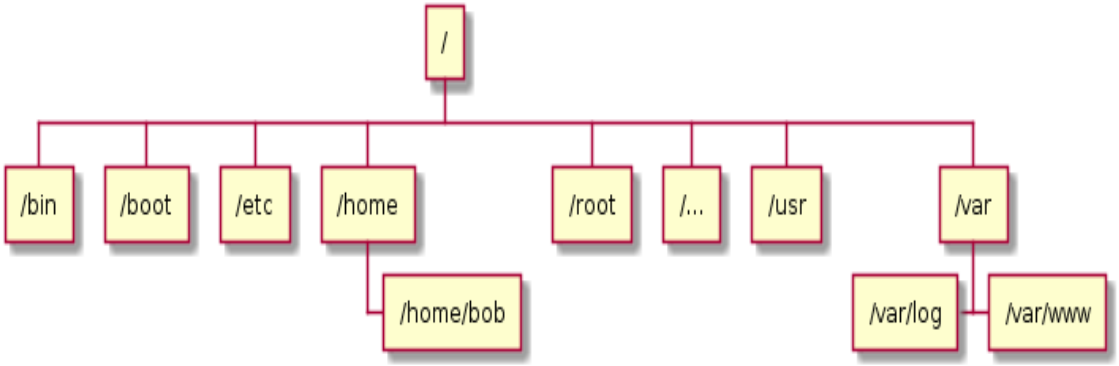

Рис. 7. Приклад ієрархічної бази даних

Особливості:

- інформація організована у вигляді дерева з відносинами «предок – нащадок»;
- кожен запис може мати не більше одного з батьків;
- зв'язки між записами виконані у вигляді фізичних покажчиків;
- неможливо реалізувати відносини «багато до багатьох».

Приклади:

- файлові системи;
- DNS;
- LDAP.

3. Мережеві бази даних (рис. 8).

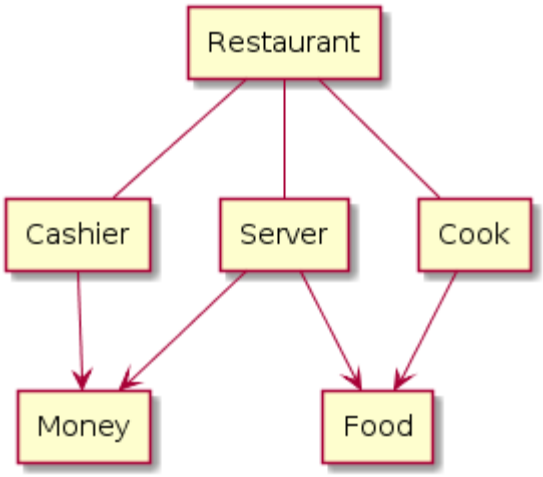

Рис. 8. Приклад мережевої бази даних

Особливості:

- мережеві бази даних подаються не деревом, а загальним графом;
- обмежені тими ж шаблонами доступу.

Приклади:

- IDMS.

I. Реляційні БД

4. SQL бази даних.

Реляційні бази даних – найстаріший тип досі широко використовуваних БД загального призначення. Дані та зв'язки між даними організовані за допомогою таблиць. Кожен стовпчик у таблиці має ім'я і тип. Кожен рядок являє собою окремий запис або елемент даних у таблиці, який містить значення для кожного із полів (рис. 9).

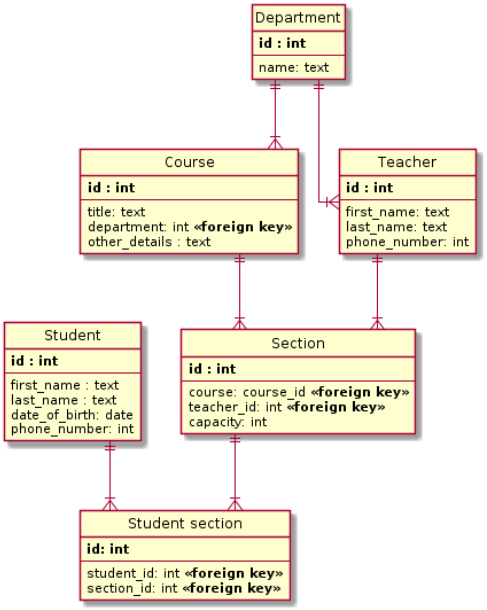

Рис. 9. Приклад реляційної бази даних

Особливості:

- поле в таблиці, що називається зовнішнім ключем, може містити посилання на поля в інших таблицях, що дозволяє їх з'єднувати;
- високоорганізована структура і гнучкість робить реляційні БД потужними і такими, що адаптуються до різних типів даних;
- для доступу до даних використовується мова структурованих запитів (SQL);
- надійний вибір для багатьох додатків.

Приклади:

- MySQL;
- MariaDB;
- PostgreSQL;
- SQLite.

II. NoSQL бази даних

Це група типів БД, що пропонують підходи, відмінні від стандартного реляційного шаблону. Говорячи NoSQL, мають на увазі або "не-SQL", або "не тільки SQL". Іноді допускається SQL-подібний запит.

5. Бази даних «ключ-значення» (рис. 10). У базах даних «ключ-значення» для зберігання інформації необхідно представити ключ і об'єкт даних, який потрібно зберегти. Наприклад, JSON-об'єкт, зображення або текст. Щоб зробити запит, відправляєте ключ і отримуєте об'єкт.

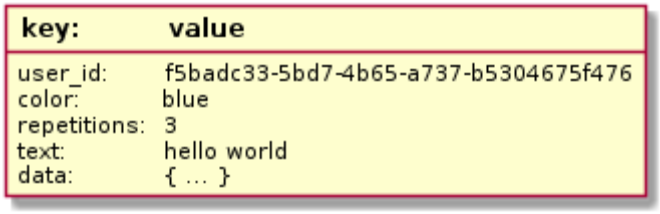

Рис. 10. Приклад бази даних «ключ-значення»

Особливості:

- сховища забезпечують швидкий і незначними зусиллями доступ;
- часто зберігають дані конфігурацій та інформацію про стан даних, представлених словниками або хешем;
- немає жорсткої схеми відносин між даними, тому в таких БД часто зберігають одночасно різні типи даних;
- розробник відповідає за визначення схеми іменування ключів і за те, щоб значення мало відповідний тип/формат.

Приклади:

- Redis;
- memcached;
- etcd.

6. Документні бази даних (рис. 11) (також документоорієнтовані БД або сховища документів) спільно використовують базову семантику доступу і пошуку сховищ ключів і значень. Такі БД також використовують ключ для унікальної ідентифікації даних. Різниця між сховищами «ключ-значення» і документної БД полягає в тому, що замість зберігання blob-об'єктів документоорієнтовані бази зберігають дані в структурованих форматах – JSON, BSON або XML.

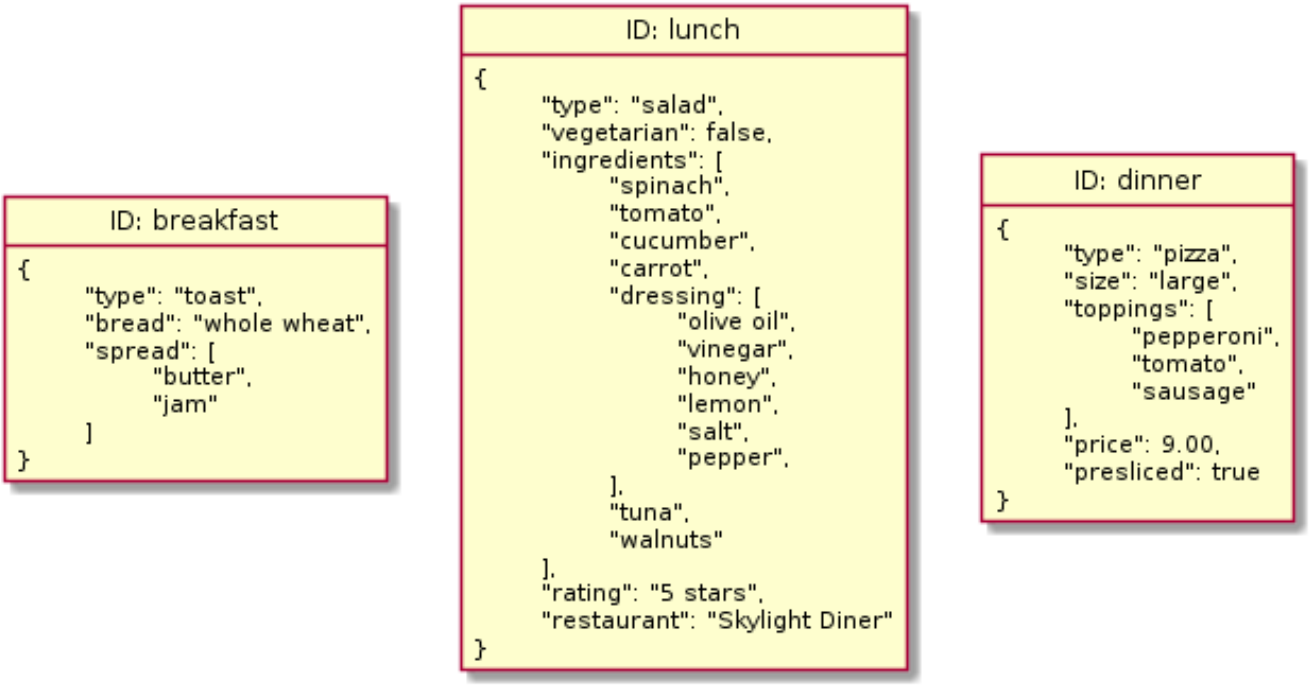

Рис. 11. Приклад документної бази даних

Особливості:

- база даних не вимагає конкретного формату або схеми;
- кожен документ може мати свою внутрішню структуру;
- документні БД є хорошим вибором для швидкої розробки;
- у будь-який момент можна змінювати властивості даних, не змінюючи структуру або самі дані.

Приклади:

- MongoDB;
- RethinkDB.

7. Графові бази даних — такі, що замість зіставлення зв'язків з таблицями і зовнішніми ключами встановлюють зв'язок, використовуючи вузли, ребра і властивості (рис. 12).

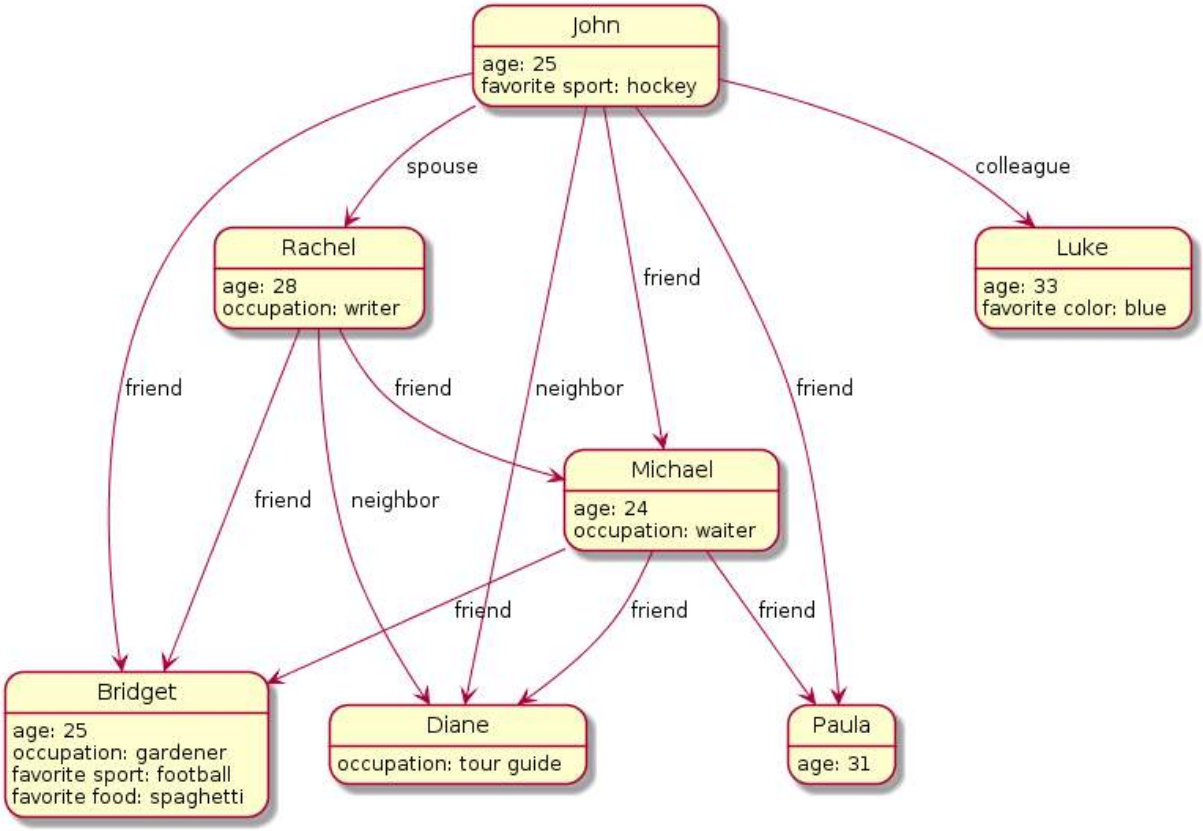

Рис.12. Приклад графової бази даних

Графові бази подають дані у вигляді окремих вузлів, які можуть мати будь-яку кількість пов'язаних з ними властивостей.

Особливості:

- мають такий самий вигляд, як мережі;
- фокусуються на зв'язках між елементами;
- явно відображають зв'язки між типами даних;
- не потребують покрокового обходу для переміщення між елементами;
- немає обмежень у типах подачі зв'язків.

Приклади:

- Neo4j;
- JanusGraph;
- Dgraph.

8. Колоночні бази даних (також нереляційні колоночні сховища або бази даних з широкими стовпцями) належать до сімейства NoSQL БД, але зовні схожі на реляційні БД. Як і у реляційних, стовпчики БД зберігають дані, використовуючи рядки і стовпці, але з іншим зв'язком між елементами.

У реляційних БД усі рядки повинні відповідати фіксованій схемі. Схема визначає, які стовпчики будуть у таблиці, типи даних та інші критерії. В основі баз даних замість таблиць є структури – «стовпчики сімейства». Сімейства містять рядки, кожен з яких визначає власний формат. Рядок

складається з унікального ідентифікатора, що використовується для пошуку, за яким слідують набори імен та значень стовпців (рис. 13).

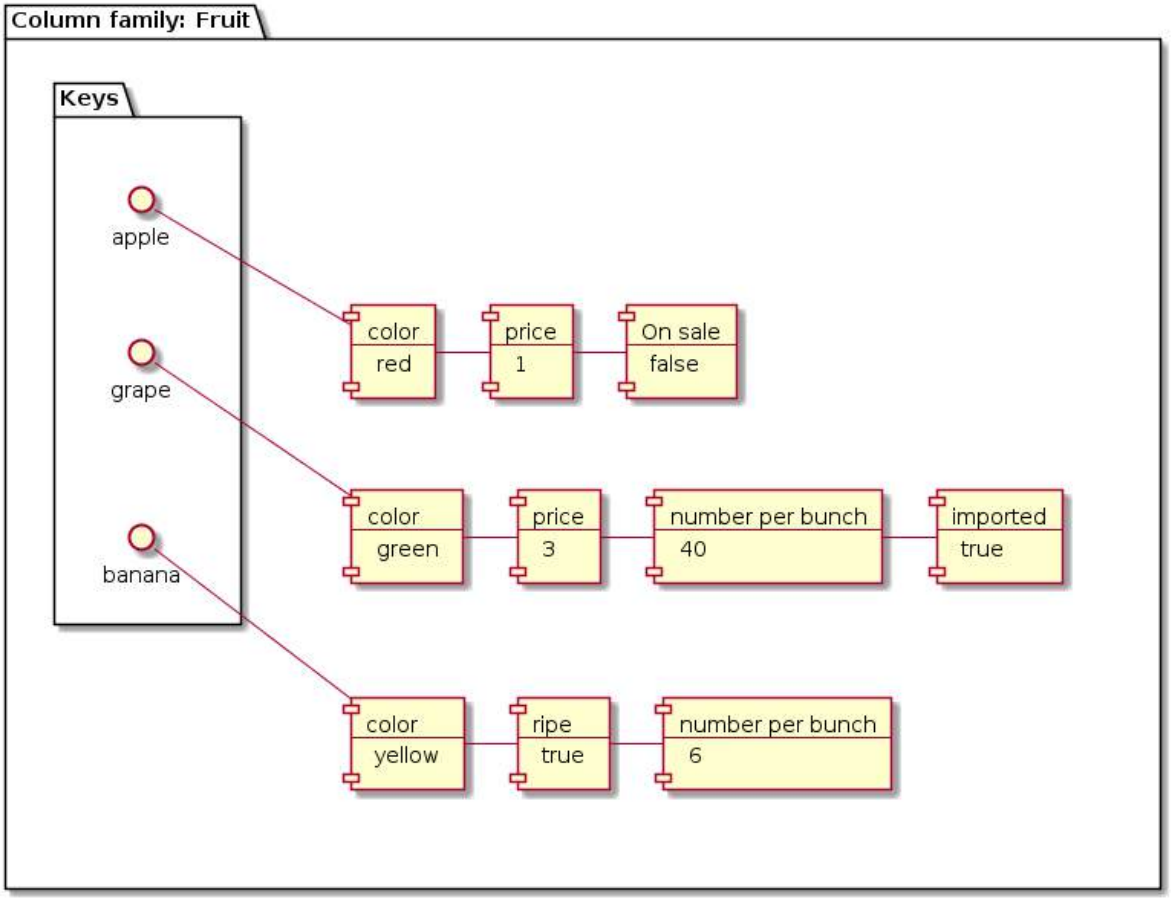

Рис. 13. Приклад колоночної бази даних

Особливості:

- БД зручні при роботі з додатками, що вимагають високої продуктивності;
- дані та метадані записи доступні по одному ідентифікатору;
- гарантовано розміщення всіх даних з рядка в одному кластері, що спрощує сегментацію і масштабування даних.

Приклади:

- Cassandra;
- HBase.

9. Бази даних часових рядів (рис. 14) створені для збору і керування елементами, що змінюються з плином часу. Більшість таких БД організовані в структури, які записують значення для одного елемента. Наприклад, можна створити таблицю для відстеження температури процесора. Усередині кожне значення буде складатися з тимчасової мітки і показника температури. У таблиці може бути декілька метрик.

Time	CPU Temp	System Load	Memory Usage %
2019-10-31T03:48:05+00:00	37	0.85	92
2019-10-31T03:48:10+00:00	42	0.87	90
2019-10-31T03:48:15+00:00	33	0.74	87
2019-10-31T03:48:20+00:00	34	0.72	77
2019-10-31T03:48:25+00:00	40	0.88	81
2019-10-31T03:48:30+00:00	42	0.89	82
2019-10-31T03:48:35+00:00	41	0.88	82

Рис. 14. Приклад бази даних часових рядів

Особливості:

- орієнтовані на запис;
- призначені для обробки постійного потоку вхідних даних;
- продуктивність залежить від кількості елементів, що відслідковуються, інтервалу опитування між записом нових значень і фактичного корисного навантаження даних.

Приклади:

- OpenTSDB;
- Prometheus;
- InfluxDB;
- TimescaleDB.

III. Комбіновані типи

NewSQL і багатомодельна БД є різними типами баз даних, але вирішують одну групу проблем, викликаних полярними підходами SQL або NoSQL-стратегіями. Тому надається можливість об'єднати переваги обох груп.

10. NewSQL бази даних успадковують реляційну структуру і семантику, але побудовані з використанням більш сучасних, масштабованих конструкцій. Мета – забезпечити більшу масштабованість, ніж реляційні БД, і більш високі гарантії узгодженості, ніж у NoSQL. Компроміс між узгодженістю і доступністю є фундаментальною проблемою розподілених баз даних.

Особливості:

- можливість горизонтального масштабування;
- висока доступність;
- велика продуктивність і реплікація;
- невеликий функціонал і гнучкість;
- чимале споживання ресурсів і необхідність спеціалізованих знань для роботи з базою даних.

Приклади:

- MemSQL;
- VoltDB;
- Spanner;

- Calvin;
- CockroachDB;
- FaunaDB;
- yugabyteDB.

11. Багатомодельні бази даних – бази, які поєднують функціональні можливості кількох видів БД. Переваги такого підходу очевидні – одна і та ж система може використовувати різні типи подання для різних типів даних.

Спільне розміщення даних з декількох типів БД в одній системі дозволяє виконувати нові операції, які в іншому випадку були б ускладнені або неможливі. Наприклад, багатомодельні бази можуть дозволити користувачам отримати доступ до даних, що зберігаються в різних типах БД, і керувати ними в межах одного запиту, а також вони підтримують узгодженість даних при виконанні операцій, що змінюють інформацію відразу в декількох системах.

Особливості:

- допомагають зменшити навантаження на СКБД;
- дозволяють розширюватися до нових моделей у міру зміни потреб без внесення змін до базової інфраструктури;
- забезпечують безперервний доступ і простий розподіл даних;
- мають лінійну масштабованість і прості для розробки.

Приклади:

- ArangoDB;
- OrientDB;
- Couchbase.

Зміна типів даних, що зберігаються, вимоги до швидкості і продуктивності привели і до значного розширення типів баз даних. При цьому кожен з них продовжує бути потрібним у своїй ніші, де взаємозв'язки між даними асоціюються з певною схемою будови бази даних [5].

Лекція 12. ПРОЕКТУВАННЯ БАЗ ДАНИХ

Можна з легкістю спроектувати базу даних з єдиною таблицею, що містить усі дані. Основним недоліком такої бази даних буде високий рівень надмірності даних.

Наприклад, єдина таблиця бази даних, що містить усі дані про співробітників і проекти, в яких вони беруть участь (вважаючи, що кожен співробітник може одночасно працювати в одному або декількох проектах і що в кожному проекті може працювати один або кілька співробітників), буде складатися з великої кількості стовпців і рядків. Основним недоліком такої таблиці є труднощі збереження узгодженості даних через їх повторюваність.

Модель "сутність – відношення" (entity – relationship (ER)) використовується при проектуванні реляційних баз даних з метою

видалення будь-якої надмірності даних. Основним об'єктом моделі "сутність – відношення" є сутність, тобто реальний об'єкт. Кожна сутність має декілька атрибутів, які є властивостями сутності, що, таким чином, описують її. Атрибут може бути одного з таких типів:

- атомарним (або однозначним). Атомарний атрибут конкретної сутності завжди представляється одним значенням. Наприклад, статус стану особи в шлюбі завжди буде атомарним атрибутом. Більшість атрибутів є атомарними;
- багатозначним. Для конкретної сутності багатозначний атрибут може мати одне або кілька значень. Наприклад, атрибут Location суті Enterprise є багатозначним, тому що підприємство може розташовуватися в декількох місцях;
- складним. Складні атрибути не є атомарними, тому що вони складаються з декількох атомарних атрибутів. Типовим прикладом складного атрибута буде поштова адреса, яка складається з таких атомарних атрибутів, як місто, індекс, вулиця і т. ін.

Сутність Person у прикладі нижче має декілька атомарних атрибутів, один складний атрибут Address і багатозначний атрибут College_degree:

PERSON (Personal_number, F_name, L_name,
Address (City, Zip, Street),
{College_degree}
)

Кожна сутність має один або кілька ключових атрибутів, які є атрибутами (або комбінацією двох або більше атрибутів) з однозначними значеннями для кожної конкретної сутності. У прикладі ключовим атрибутом сутності PERSON є атрибут Personal_number.

Крім сутності й атрибута, ще одним базовим поняттям моделі "сутність–відношення" є відношення. Відношення існує, коли одна сутність посилається на іншу (або кілька інших). Кількість залучених сутностей визначає ступінь відношення. Наприклад, між сутностями Employee і Project існує відношення другого ступеня – Works_on.

Кожне існуюче відношення між двома сутностями повинно належати до одного з трьох типів: 1: 1, 1: N або M: N. Ця властивість відношення називається потужністю відношення (cardinality ratio) або кардинальним числом. Наприклад, між сутностями Department і Employee існує відношення типу 1:N, тому що кожен співробітник належить тільки одному відділу, який, у свою чергу, містить одного або декількох співробітників. А між сутностями Project і Employee існує відношення типу M:N, оскільки в кожному проекті бере участь один або більше співробітників і кожен співробітник одночасно бере участь в одному або більше проектів.

Процес проектування БД з використанням методу нормальних форм (НФ) є ітераційним і полягає в послідовному перетворенні відносини з 1НФ в НФ більш високого порядку за певними правилами. Кожна наступна НФ обмежується певним типом функціональних залежностей і усуненням

відповідних аномалій при виконанні операцій над відносинами БД, а також збереженням властивостей попередніх НФ.

При проектуванні БД використовуються такі визначення та поняття:

Атрибут – властивість деякої сутності. Часто називається полем таблиці.

Домен атрибута – безліч допустимих значень, яких може набувати атрибут.

Кортеж – запис у таблиці (рядок таблиці).

Сутність – таблиця.

Відношення – зв'язок між таблицями на основі первинних ключів.

Первинний ключ – це поле або комбінація полів таблиці, які забезпечують унікальність запису.

Проекція – відношення, отримане з заданим шляхом видалення і (або) перестановки деяких атрибутів.

Функціональна залежність між атрибутами (множинами атрибутів) X і Y означає, що для будь-якого допустимого набору кортежів у відношенні: якщо два кортежі збігаються за значенням X, то вони збігаються за значенням Y. Наприклад, якщо значення атрибута «Назва компанії» – Canonical Ltd, то значенням атрибута «Штаб-квартира» в такому кортежі завжди буде Millbank Tower, London, United Kingdom. Позначення: {X} -> {Y}.

Нормальна форма – вимога до структури таблиць у теорії реляційних баз даних для усунення з бази надлишкових функціональних залежностей між атрибутами (полями таблиць).

Метод нормальних форм (НФ) полягає в зборі інформації про об'єкти вирішення завдання в межах одних відносин і подальшої декомпозиції цих відносин на кілька взаємопов'язаних відносин на основі процедур нормалізації відносин.

Мета нормалізації: виключити надмірне дублювання даних, яке є причиною аномалій, що виникли при додаванні, редагуванні і видаленні кортежів (рядків таблиці).

Перша нормальна форма

1НФ означає, що таблиця не містить багатозначних або складних атрибутів (полів).

Наприклад, є таблиця «Автомобілі»:

Фірма	Моделі
BMW	M5, X5M, M1
Nissan	GT-R

Перетворимо таблицю до 1НФ:

Фірма	Моделі
BMW	M5
BMW	X5M
BMW	M1
Nissan	GT-R

Друга нормальна форма

Таблиця знаходиться у 2НФ, якщо вона знаходиться в 1НФ і не містить ключових стовпців, залежних від частини стовпців первинного ключа цієї таблиці. Це означає, що якщо (A, B) є комбінацією двох стовпців таблиці, що становлять первинний ключ, тоді таблиця не містить стовпців, що залежать тільки від A чи тільки від B.

Наприклад, є таблиця:

Модель	Фірма	Ціна	Знижка
M5	BMW	5500000	5 %
X5M	BMW	6000000	5 %
M1	BMW	2500000	5 %
GT-R	Nissan	5000000	10 %

Таблиця знаходиться в першій нормальній формі, але не в другій. Ціна машини залежить від моделі і фірми. Знижка залежить від фірми, тобто залежність від первинного ключа неповна. Виправляється це шляхом декомпозиції її на дві відносини, в яких не ключові атрибути залежать від первинного ключа.

Перетворимо таблицю:

Модель	Фірма	Ціна
M5	BMW	5500000
X5M	BMW	6000000
M1	BMW	2500000
GT-R	Nissan	5000000

Фірма	Знижка
BMW	5 %
Nissan	10 %

Третя нормальна форма

Таблиця знаходиться в 3НФ, якщо вона знаходиться у 2НФ і відсутні функціональні залежності між не ключовими стовпцями.

Розглянемо таблицю:

Модель	Магазин	Телефон
BMW	Ріал-авто	87-33-98
Audi	Ріал-авто	87-33-98
Nissan	Некст-Авто	94-54-12

Таблиця знаходиться у 2НФ, але не в 3НФ. У таблиці атрибут «Модель» є первинним ключем. Особистих телефонів у автомобілів немає, і телефон залежить виключно від магазину.

Таким чином, у таблиці існують такі функціональні залежності: Модель → Магазин, Магазин → Телефон, Модель → Телефон. Залежність Модель → Телефон є транзитивною, отже, відношення не перебуває у 3НФ.

Унаслідок поділу вихідної таблиці формуються дві таблиці, що знаходяться в 3НФ:

Магазин	Телефон
Ріал-авто	87-33-98
Некст-Авто	94-54-12

Модель	Магазин
BMW	Ріал-авто
Audi	Ріал-авто
Nissan	Некст-Авто

Існує, як мінімум, шість нормальних форм.

Нормалізацією даних (data normalization) називається процес, у якому таблиці бази даних перевіряються на наявність певних залежностей між стовпцями таблиці. Якщо таблиця містить такі залежності, вона розбивається на кілька таблиць (зазвичай дві), що дозволяє позбутися від залежностей між стовпцями. Якщо одна з цих таблиць все ще містить залежності, процес нормалізації повторюється доти, доки всі залежності не будуть дозволені.

Система керування базами даних MS SQL Server

Система керування базами даних (СКБД) – це загальний набір різних програмних компонентів баз даних і власне баз даних, що містить такі складові:

- прикладні програми баз даних;
- клієнтські компоненти;
- сервери баз даних;
- власне бази даних.

Прикладна програма баз даних являє собою програмне забезпечення спеціального призначення, розроблене і реалізоване користувачами чи третіми особами – компаніями-розробниками програмного забезпечення. На противагу, клієнтські компоненти – це програмне забезпечення баз даних загального призначення, розроблене і реалізоване компанією-розробником бази даних. За допомогою клієнтських компонентів користувачі можуть отримати доступ до даних, що зберігаються на локальному або віддаленому комп'ютері.

Сервер баз даних виконує завдання керування даними, що зберігаються в базі даних. Клієнти взаємодіють з сервером баз даних, відправляючи йому запити. Сервер обробляє кожен отриманий запит і відправляє результати відповідному клієнту.

Можливості СКБД

Загалом, базу даних можна розглядати з двох точок зору – користувача і системи бази даних. Користувачі бачать базу даних як набір логічно пов'язаних даних, а для системи баз даних це просто послідовність байтів, які зазвичай зберігаються на диску. Хоча це два повністю різні погляди, між ними є щось спільне: система баз даних повинна надавати не тільки інтерфейс, що дозволяє користувачам створювати бази даних і вибирати або модифікувати дані, але також системні компоненти для керування збереженими даними. Тому система баз даних повинна надавати такі можливості:

- різноманітні інтерфейси;
- фізичну незалежність даних;
- логічну незалежність даних;
- оптимізацію запитів;
- цілісність даних;
- керування паралелізмом;

- резервне копіювання та відновлення;
- безпеку баз даних.

Різноманітні інтерфейси

Більшість баз даних проектуються і реалізуються для роботи з ними різних типів користувачів, що мають різні рівні знань. З цієї причини система баз даних повинна надавати кілька окремих користувальницьких інтерфейсів. Інтерфейс може бути графічним або текстовим.

У графічних інтерфейсах введення здійснюється за допомогою клавіатури або миші, а відображення – в графічному вигляді на монітор. Різновидом текстового інтерфейсу, часто використовуваного в системах баз даних, є інтерфейс командного рядка, за допомогою якого користувач здійснює введення за допомогою набору команд на клавіатурі, а система відображає результат у текстовому форматі на моніторі.

Фізична незалежність даних

Фізична незалежність даних означає, що прикладні програми бази даних не залежать від фізичної структури даних, що зберігаються в БД. Ця важлива особливість дозволяє змінювати збережені дані без необхідності вносити будь-які зміни в прикладні програми баз даних.

Наприклад, якщо дані були спочатку впорядковані за одним критерієм, а потім цей порядок був змінений за іншим критерієм, зміна фізичних даних не повинна впливати на існуючі програми баз даних або її схему (опис бази даних, створений мовою визначення даних системи бази даних).

Логічна незалежність даних

При обробці файлів, використовуючи традиційні мови програмування, файли оголошуються прикладними програмами, тому будь-які зміни в структурі файлу зазвичай вимагають внесення відповідних змін до всіх програм, що їх використовують.

Системи баз даних надають логічну незалежність файлів, тобто, іншими словами, логічну структуру бази даних можна змінювати без необхідності внесення будь-яких змін у прикладні програми бази даних. Наприклад, додавання атрибута до вже існуючої в системі баз даних структури об'єкта з ім'ям Person (наприклад, адреса) викликає необхідність модифікувати тільки логічну структуру бази даних, а не існуючі прикладні програми. (Однак додатки потребують модифікування для використання нового стовпчика).

Оптимізація запитів

Більшість систем баз даних містять підкомпонент, який має назву оптимізатор, що розглядає кілька можливих стратегій виконання запиту даних і вибирає з них найбільш ефективну. Обрана стратегія називається планом виконання запиту. Оптимізатор приймає рішення, беручи до уваги такі фактори, як розмір таблиць, до яких направлено запит, існуючі індекси

і логічні оператори (AND, OR та NOT), які використовуються у виразі WHERE.

Цілісність даних

Одним із завдань, які виконуються системою баз даних, є ідентифікувати логічно суперечливі дані і не допустити їх переміщення в базу даних. (Прикладом таких даних буде дата "30 лютого" або час "5:77:00"). Крім цього, для більшості реальних завдань, які реалізуються за допомогою систем баз даних, існують обмеження для забезпечення цілісності (integrity constraints), які повинні виконуватися для даних. (Як приклад обмеження для забезпечення цілісності можна назвати вимогу, щоб табельний номер співробітника був п'ятизначним цілим числом).

Забезпечення цілісності даних може здійснюватися користувачем в прикладній програмі або ж системою керування базами даних. Це завдання слід виконувати за допомогою СКБД.

Керування паралелізмом

Система баз даних являє собою розраховану на багато користувачів систему програмного забезпечення, що означає одночасне звернення до бази даних множинних користувальницьких додатків. Тому кожна система баз даних повинна мати механізм, що забезпечує керування спробами модифікувати дані декількома додатками одночасно. Далі наводиться приклад проблеми, яка може виникнути, якщо система баз даних не оснащена таким механізмом:

1. На загальному банківському рахунку № 3811 в банку X є $ 1500.
2. Власники цього рахунку, пані А і пан Б, йдуть у різні відділення банку і одночасно знімають з рахунку по $ 750 кожен.
3. Сума, що залишилася на рахунку № 3811 після цих транзакцій, повинна бути $ 0, і в жодному разі не $ 750.

Усі системи баз даних повинні мати необхідні механізми для обробки подібних ситуацій, забезпечуючи керування паралелізмом.

Створення резервних копій та відновлення

Система баз даних повинна бути оснащена підсистемою для відновлення після помилок у програмному й апаратному забезпеченні. Наприклад, якщо в процесі оновлення 100 рядків таблиці бази даних відбувається збій, то підсистема відновлення повинна виконати відкат усіх виконаних оновлень, щоб забезпечити несуперечливість даних.

Безпека баз даних

Найбільш важливими поняттями безпеки баз даних є аутентифікація й авторизація. Аутентифікація – це процес перевірки автентичності облікових даних користувача, щоб не допустити використання системи несанкціонованими користувачами. Аутентифікація найбільш часто реалізується через введення імені користувача та пароля. Система перевіряє достовірність цієї інформації, щоб вирішити, має користувач

право входу в систему чи ні. Цей процес можна посилити застосуванням шифрування.

Авторизація – це процес, що застосовується до користувачів, які вже отримали доступ до системи, щоб визначити їх права на використання певних ресурсів. Наприклад, доступ до інформації про структуру бази даних і системного каталогу певної сутності можуть отримати тільки адміністратори.

Мова реляційної бази даних у системі SQL Server називається Transact-SQL. Це різновид найзначнішої на сьогодні мови бази даних – мови SQL (Structured Query Language – мова структурованих запитів). Походження мови SQL тісно пов'язане з проектом під назвою System R, розробленим і реалізованим компанією IBM ще на початку 80-х років минулого століття. За допомогою цього проекту було продемонстровано, що, використовуючи теоретичні основи роботи Едгара Ф. Кодда, можна створити систему реляційних баз даних.

На відміну від традиційних мов програмування, таких як C#, C++ і Java, мова SQL є множинно-орієнтованою (set-oriented). Розробники мови також називають її запис-орієнтованою (record-oriented). Це означає, що в мові SQL можна запитувати дані з декількох рядків однієї або декількох таблиць, використовуючи лише одну інструкцію. Це одна з найважливіших переваг мови SQL, що дозволяє використовувати цю мову на логічно більш високому рівні, ніж традиційні мови програмування.

Іншою важливою властивістю мови SQL є її непроцедурність. Будь-яка програма, написана процедурною мовою (C#, C++, Java), крок за кроком описує, як виконувати певне завдання. На противагу цьому, мова SQL, як і будь-яка інша непроцедурна мова, описує, чого хоче користувач. Таким чином, відповідальність за знаходження відповідного способу для задоволення запиту користувача лежить на системі.

SQL Server Management Studio

Система SQL Server надає різні інструменти для виконання різноманітних завдань, таких як установка, конфігурація, контрольна перевірка системи (аудит) та налаштування її продуктивності. Основним інструментом адміністратора для взаємодії з системою є IDE-середовище керування SQL Server Management Studio. Як адміністратори, так і кінцеві користувачі можуть використовувати цей інструмент для адміністрування множинних серверів, розробки баз даних і реплікації даних.

Середовище керування SQL Server Management Studio складається з декількох різних компонентів, які використовуються для адміністрування та керування всією системою. Основні з цих компонентів:

- Registered Servers (Зареєстровані сервери);
- Object Explorer (Оглядач об'єктів);
- Query Editor (Редактор запитів);
- Solution Explorer (Оглядач рішень).

Щоб відкрити головний інтерфейс середовища SQL Server Management Studio, потрібно спочатку підключитися до сервера.

Підключення до сервера

При запуску середовища SQL Server Management Studio відкривається діалогове вікно Connect to Server (З'єднання з сервером), в якому потрібно задати необхідні параметри для підключення до сервера:

Server type (Тип сервера)

Зі списку вибираємо опцію Database Engine (Компонент Database Engine). За допомогою середовища SQL Server Management Studio також можна керувати об'єктами компонента Database Engine і служб Analysis Services. Розглянемо використання середовища SQL Server Management Studio тільки для керування об'єктами компонента Database Engine.

Server name (Ім'я сервера)

Виберіть зі списку або введіть з клавіатури ім'я сервера, до якого потрібно підключитися. (Зазвичай, середовище SQL Server Management Studio можна підключити до будь-якого встановленого продукту на конкретному сервері).

Authentication (Перевірка справжності)

Виберіть один з двох типів перевірки автентичності:

- Windows Authentication (Перевірка справжності Windows). Підключитися за допомогою SQL Server до свого облікового запису Windows. Це найбільш легкий варіант підключення і рекомендується компанією Microsoft;
- SQL Server Authentication (Перевірка справжності SQL Server). Використовується перевірка справжності компонента Database Engine за іменем користувача і паролем (рис. 15).

Рис. 15. З'єднання з сервером

Вказавши всі необхідні параметри, натисніть кнопку Connect (З'єднати) – і Database Engine підключиться до вказаного серверу. Після підключення до сервера бази даних відкривається головне вікно середовища SQL Server Management Studio. Своїм зовнішнім виглядом воно схоже на головне вікно середовища розробки Visual Studio, тому користувачі можуть застосувати свій досвід роботи в Visual Studio в цьому середовищі. На рисунку нижче показано головне вікно середовища SQL Server Management Studio з декількома панелями (рис. 16).

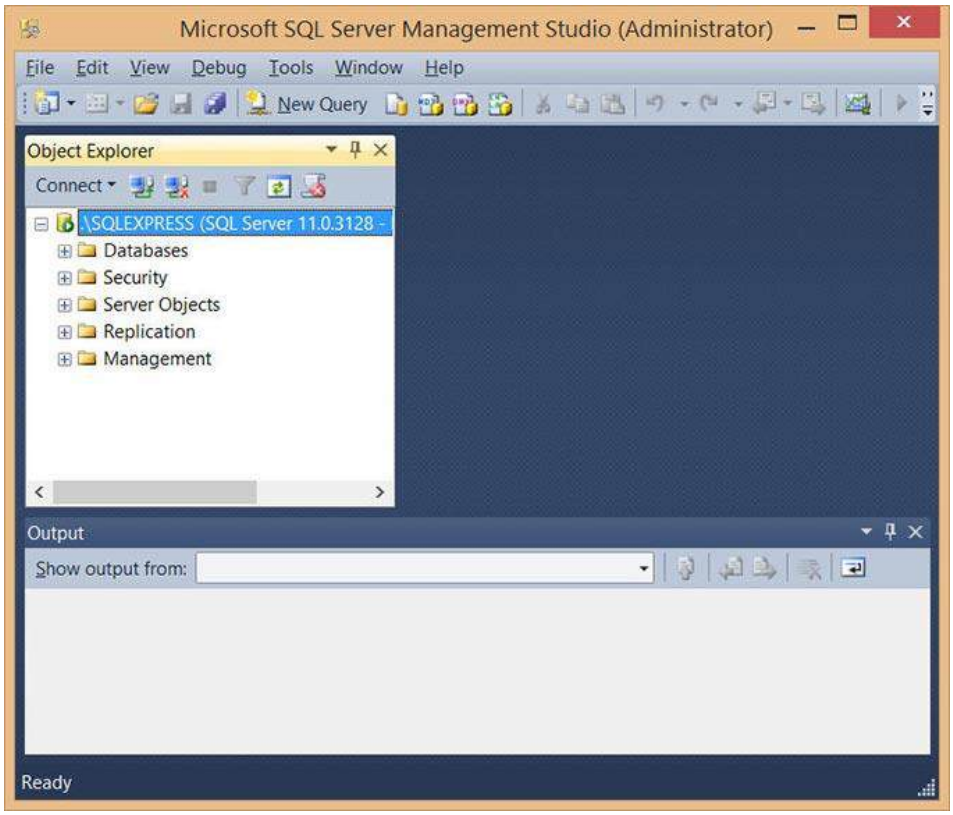

Рис. 16. Головне вікно середовища SQL Server Management Studio

Середовище SQL Server Management Studio надає єдиний інтерфейс для керування серверами і створення запитів для всіх компонентів SQL Server. Іншими словами, для компонентів Database Engine, служб Analysis Services, служб Integration Services і служб Reporting Services застосовується один і той же графічний інтерфейс.

Панель Object Explorer (Оглядач об'єктів) містить у вигляді дерева подання всіх об'єктів баз даних сервера. Якщо панель Object Explorer не відтворена, то її можна відкрити, вибравши послідовність команд з меню View -> Object Explorer. Це деревоподібне уявлення відображає ієрархію об'єктів на сервері. Таким чином, якщо її розгорнути, буде показано логічну структуру відповідного сервера.

Оглядач об'єктів дозволяє підключатися в одній панелі до одного сервера. Це можуть бути будь-які з наявних серверів для компонента Database Engine, служб Analysis Services, Reporting Services або Integration

Services. Така можливість полегшує роботу користувача, оскільки вона дозволяє керувати всіма серверами одного або різних типів з одного місця.

Лекція 13. МОВА ЗАПИТІВ SQL

Виділяють такі види SQL-запитів:

DDL (Data Definition Language) – мова визначення даних. Завданням DDL-запитів є створення БД і опис її структури. Запитами такого виду встановлюються правила того, в якому вигляді різні дані будуть розміщуватися в БД.

DML (Data Manipulation Language) – мова маніпулювання даними. До запитів цього типу належать різні команди, з використанням яких безпосередньо виробляються деякі маніпуляції з даними. DML-запити потрібні для додавання змін до вже внесених даних, для отримання даних з БД, для їх збереження, для поновлення різних записів і для їх видалення з БД. До елементів DML-звернень входить основна частина SQL-операторів.

DCL (Data Control Language) – мова керування даними. Включає в себе запити і команди, що стосуються дозволів, прав та інших налаштувань СКБД.

TCL (Transaction Control Language) – мова керування транзакціями. Конструкції такого типу застосовують, щоб керувати змінами, які виробляються з використанням DML-запитів. Конструкції TCL дозволяють об'єднувати DML-запити в набори транзакцій.

Основні типи SQL-запитів за їх видами показано на рис. 17.

Рис. 17. Команди SQL

Загальна структура запиту виглядає так:
SELECT ('стовпці або * для вибору всіх стовпців; обов'язково')
FROM ('таблиця; обов'язково')
WHERE ('умова / фільтрація; необов'язково')
GROUP BY ('стовпець, по якому хочемо згрупувати дані; необов'язково')
HAVING ('умова / фільтрація на рівні згрупованих даних; необов'язково')
ORDER BY ('стовпець, по якому хочемо впорядкувати висновок; необов'язково')

SELECT, FROM
SELECT, FROM – обов'язкові елементи запиту, які визначають вибрані стовпці, їх порядок і джерело даних.
Вибрати все (позначається як *)

Розглянемо базу даних, що складається з двох таблиць:
students (поля: Id, fam, nomgroup, kyrs);
data_student (поля: Id, fams, addr, year_b, tel, propuski).

Приклади запитів:

Select * from students

Вибрати конкретні стовпці
select Id, fam from students
select 23, SYSTEM_USER, propuski * 2 from data_student

Використання псевдонімів стовпців
Щоб привласнити стовпцю псевдонім, можна використовувати ключове слово AS.
select NomGroup As НОМЕР_групи, fam прізвище from students

Обчислювані стовпці
select fams прізвище, 2020 year_b as Age from data_student

ORDER BY
ORDER BY – необов'язковий елемент запиту, який відповідає за сортування таблиці.
Простий приклад сортування за одним стовпцем.
select fams from data_student order by 2020 year_b (сортування від меншого до більшого)
select * from data_student order by 6 (пропуски)

Приклад сортування по обчислювальному полю

select fams, propuski / 2 as kol_par from data_student order by kol_par

Здійснювати сортування можна і за кількома стовпцями, в цьому випадку сортування відбувається по порядковому номеру зазначених стовпців. За замовчуванням сортування відбувається по збільшенню значень для чисел і в алфавітному порядку для текстових значень. Якщо потрібне зворотне сортування, то в конструкції ORDER BY після назви стовпчика треба додати DESC.

Сортування за замовчуванням за одним стовпцем і зворотне сортування за іншим стовпцем:

select fams, propuski as P, tel, addr from data_student order by 1, P desc

Ключове слово DISTINCT дозволяє виводити записи, що не повторюються:

select DISTINCT kyrs from students

select DISTINCT nomgroup, kyrs from students

Ключове слово WHERE дозволяє фільтрувати дані за певними умовами.

select fam, nomgroup, kyrs from students where kyrs = 6 order by fam

Умови в WHERE можуть бути написані з використанням логічних операторів (AND / OR) і математичних операторів порівняння (=, <,>, <=,> =, <>).

select fams as surname, addr from data_student where year_b <= 1994 and propuski <10

Умови в WHERE можуть бути записані з використанням ще кількох команд, якими є:

IN – порівнює значення в стовпці з декількома можливими значеннями і повертає true, якщо поле збігається хоча б з одним значенням.

BETWEEN – перевіряє, чи знаходиться значення в якомусь проміжку.

LIKE – шукає за шаблоном.

Виводимо студентів, прізвище яких починається на букву «К». Знак % означає будь-яку послідовність символів (0 символів теж вважається за послідовність), а знак «_» означає один символ.

select fam as surname, nomgroup from students where fam like 'До%' and kyrs> 2 (важлива мовна розкладка).

select fam as surname, nomgroup from students where fam like '_н%' and kyrs> 2

select fams, addr from data_student where propuski between 10 and 20

select fams, addr from data_student where fams in ('Сірман Е.', 'Сафонов В.')

select distinct fams, addr, propuski from data_student order by fams

У SQL багато вбудованих функцій для виконання різних операцій. Наведемо тільки найбільш часто використовувані:

COUNT () – повертає кількість рядків.

SUM () – повертає суму всіх полів з числовими значеннями в них.

AVG () – повертає середнє значення серед рядків.

MIN () / MAX () – повертає мінімальне / максимальне значення серед рядків.

select count (DISTINCT fams) AS nomb_students, max (propuski) Max_pr from data_student where year_b> = 1993

GROUP BY

GROUP BY – необов'язковий елемент запиту, за допомогою якого можна задати агрегацію за потрібною колонкою (наприклад, якщо потрібно дізнатися, скільки є студентів конкретного року народження). При використанні GROUP BY обов'язково, щоб:

- перелік стовпців, по яких робиться проекція, був однаковим усередині SELECT і всередині GROUP BY;
- агрегатні функції (SUM, AVG, COUNT, MAX, MIN) були також указані всередині SELECT із зазначенням стовпчика, до якого така функція застосовується.

select year_b, count (DISTINCT fams) AS nomb_students, max (propuski) Max_pr from data_student Group by year_b

HAVING

HAVING – необов'язковий елемент запиту, який відповідає за фільтрацію на рівні згрупованих даних (по суті, WHERE, але тільки на рівень вище).

select year_b, count (DISTINCT fams) AS nomb_students, max (propuski) Max_pr from data_student Group by year_b Having MAX (propuski)> 15

Об'єднання таблиць. Приклад внутрішнього сполучення

Повертає рядок тільки в тому випадку, якщо стовпець з'єднання містить значення, яке задовольняє умові з'єднання. Це означає, якщо рядок буде містити пусте значення в одному зі стовпців умови з'єднання, запис не буде повернуто.

select T1.fam, T1.nomGroup, T2.year_b, T2.propuski from students T1, data_student T2 where T1.fam = T2.fams

Зовнішні об'єднання

JOIN – необов'язковий елемент, використовується для об'єднання таблиць по ключу.

Може повертати рядок, навіть якщо один зі стовпців в умові з'єднання містить значення NULL.

Праве з'єднання

select * from students right outer join data_student on students.fam = data_student.fams

Результаты | Сообщения

	Id	fam	nomgro...	kyrs	Id	fams	addr	year...	tel	propuski
1	1001	Редько В.	359	6	1001	Редько В.	парпарпа 12	1993	0504576	17
2	1002	Хазай К.	369	6	1002	Хазай К.	орлорло 14	1993	05045643	6
3	NULL	NULL	NULL	NULL	1003	Сафонов В.	дллодлоо 18	1994	050464365	20
4	1004	Сирман Е.	339	3	1004	Сирман Е.	йцуцкукцу 10	1995	5676576665	25
5	1005	Цветковский И.	329	2	1005	Цветковский И.	рлорлр19	1996	6876867222	12

Рис. 18. Праве з’єднання

Ліве з'єднання

select T1.fam, T2.tel from students T1 left outer join data_student T2 on T1.fam = T2.fams

	fam	tel
1	Редько В.	0504576
2	Хазай К.	05045643
3	Софонов В.	NULL
4	Сирман Е.	5676576665
5	Цветковский И.	6876867222

Рис. 19. Ліве з’єднання

Повне з'єднання

select * from students full outer join data_student on students.fam = data_student.fams

	Id	fam	nomgro...	kyrs	Id	fams	addr	year...	tel	propuski
1	1001	Редько В.	359	6	1001	Редько В.	парпарпа 12	1993	0504576	17
2	1002	Хазай К.	369	6	1002	Хазай К.	орлорло 14	1993	05045643	6
3	1003	Софонов В.	359	5	NULL	NULL	NULL	NULL	NULL	NULL
4	1004	Сирман Е.	339	3	1004	Сирман Е.	йцуцкукцу 10	1995	5676576665	25
5	1005	Цветковский И.	329	2	1005	Цветковский И.	рлорлр19	1996	6876867222	12
6	NULL	NULL	NULL	NULL	1003	Сафонов В.	дллодлоо 18	1994	050464365	20

Рис. 20. Повне з’єднання

Вкладені запити

select * from data_student where year_b = (select MAX (year_b) from data_student)

// всі студенти максимального року народження

select * from students where fam in (select fams from data_student).

Лекція 14. ВВЕДЕННЯ В ENTITY FRAMEWORK

Entity Framework є спеціальною об'єктно-орієнтованою технологією на базі фреймворка .NET для роботи з даними. Якщо традиційні засоби ADO.NET дозволяють створювати підключення, команди та інші об'єкти для взаємодії з базами даних, то Entity Framework являє собою більш високий рівень абстракції, який дозволяє абстрагуватися від самої бази даних і працювати з даними незалежно від типу сховища. Якщо на фізичному рівні оперуємо таблицями, індексами, первинними і зовнішніми ключами, то на концептуальному рівні, який нам пропонує Entity Framework, вже працюємо з об'єктами [6].

Центральною концепцією Entity Framework є поняття сутності (entity). Сутність являю собою набір даних, асоційованих з певним об'єктом. Тому ця технологія передбачає роботу не з таблицями, а з об'єктами і їх наборами.

Будь-яка сутність, як і будь-який об'єкт з реального світу, має низку властивостей. Наприклад, якщо сутність описує людину, то можемо виділити такі властивості, як ім'я, прізвище, зріст, вік, вага. Властивості необов'язково є простими даними типу int, а можуть бути більш комплексними структурами даних. І кожна сутність може мати одну або кілька властивостей, що відрізнятимуть її від інших і унікально визначатимуть її. Подібні властивості називають ключами.

При цьому сутності можуть бути пов'язані асоціативним зв'язком одна-до-багатьох, одна-до-одної і багато-до-багатьох, подібно до того, як у реальній базі даних відбувається зв'язок через зовнішні ключі.

Відмінною рисою Entity Framework є використання запитів LINQ для вибірки даних з БД. За допомогою LINQ можемо не тільки отримувати певні рядки, що зберігають об'єкти, з БД, але й отримувати об'єкти, пов'язані різними асоціативними зв'язками.

Іншим ключовим поняттям є Entity Data Model. Ця модель зіставляє класи сутностей з реальними таблицями в БД.

Entity Data Model складається з трьох рівнів: концептуального, рівня сховища і рівня зіставлення (мапінгу).

На концептуальному рівні відбувається визначення класів сутностей, які використовуються в додатку.

Рівень сховища визначає таблиці, стовпці, відносини між таблицями і типи даних, з якими порівнюється використовувана база даних.

Рівень зіставлення (мапінгу) служить посередником між попередніми двома, визначаючи зіставлення між властивостями класу сутності і стовпцями таблиць.

Таким чином, можемо через класи, визначені у додатку, взаємодіяти з таблицями з бази даних.

Платформа ADO.NET Entity Framework (EF) – це програмна модель, яка намагається заповнити прогалину між конструкціями бази даних і

об'єктно-орієнтованими конструкціями. Використовуючи EF, можна взаємодіяти з реляційними базами даних, які мають працювати з кодом SQL (при бажанні). Виконуюче середовище EF генерує відповідні оператори SQL, коли застосовуються запити LINQ до суворо типізованих класів.

LINQ to Entities – це термін, що описує застосування запитів LINQ до сутнісних об'єктів ADO.NET [7].

*Роль файлу * .edmx*

Отже, сутності – це класи клієнтської сторони, що функціонують як модель сутнісних даних (Entity Data Model). Хоча сутності клієнтської сторони в кінцевому підсумку виливаються в таблицю бази даних, жорсткий зв'язок між іменами властивостей сутнісних класів та іменами стовпців таблиць з даними відсутній.

У контексті API-інтерфейсу Entity Framework, щоб дані сутнісних класів відображалися в дані таблиць коректно, потрібно правильне визначення логіки відображення. У будь-якій системі, керованій моделлю даних, рівень сутностей реальної бази даних і відображення розділені на окремі частини: концептуальна модель, логічна модель і фізична модель.

Концептуальна модель визначає сутності і відносини між ними (якщо є).

Логічна модель відображає сутності і відносини на таблиці з будь-якими необхідними обмеженнями зовнішніх ключів.

Фізична модель надає можливості конкретного механізму даних, вказуючи деталі сховища, такі як таблична схема, розбиття на розділи й індексація.

Після створення EDM може знадобитися визначення рядків з'єднання для Entity Framework. Зверніть увагу, що при використанні дизайнера вона буде автоматично внесена в конфігурацію програми. А ось у разі застосування підходу «код спочатку» її необхідно додати самостійно.

Використання файлу конфігурації (app.config або web.config)

name – ім'я рядка з'єднання;

connectionString – параметри з'єднання для Entity Framework;

metadata – де розташовані метадані EDM. Три зазначених файли відповідають за контекстну модель (csdl), модель сховища (ssdl) і модель співвідношення (msl);

provider – ім'я провайдера даних для Entity Framework. Це System.Data.SqlClient для роботи з SQL Server або SQL Server Express;

provider connection string – параметри з'єднання. Вміст цього параметра залежить від обраного провайдера даних;

providerName – ім'я провайдера даних для додатку. Воно завжди System.Data.EntityClient.

Розробники запропонували внести в програмні проекти "об'єктний шар бази даних", в якому таблиці баз даних було подано у вигляді класів, стовпці у вигляді властивостей, а вся робота з базою даних будувалася на

програмному коді без використання SQL. Такий підхід отримав назву об'єктно-реляційного відображення (object-relational mapping – ORM).

Різні програмні платформи пропонують безліч систем, що реалізують модель ORM. Entity Framework у поєднанні з LINQ (Language-Integrated Query) являє собою реалізацію ORM для платформи .NET Framework від компанії Microsoft. Entity Framework містить механізми створення і роботи з сутностями бази даних через об'єктно-орієнтований код мовою, сумісною з CLR (у нашому курсі використовуємо C#). LINQ являє собою бібліотеку, яка розширює можливості C# і полегшує створення запитів (завдяки LINQ можна створювати SQL-подібні запити в коді C#).

Entity Framework є продовженням іншого API-інтерфейсу для роботи з базами даних у .NET – ADO.NET, в якому для роботи з базами даних доводилося писати запити на SQL і вставляти їх у команди. Але Entity Framework значно полегшує життя програмістам C#.

Модель EDM

Entity Framework акцентує увагу на моделюванні – тут використовуються діаграми ER (entity-relationship, "сутність-відношення"), підхід з використанням логічного і фізичного проектування шарів та ін. Ядром Entity Framework є модель EDM (Entity Data Model), суть якої полягає в зберіганні сутностей (entity) у вигляді суворо типізованих класів, а не у вигляді об'єктів схеми бази даних. Модель EDM дозволяє забезпечити зв'язок між сутнісними класами в коді і таблицями бази даних.

Шари

Архітектура Entity Framework в абстрактному сенсі заснована на шарах (layers): робочому, віддаленому і сполучному.

Класи з кодом сутностей містяться в робочому шарі, в якому працюють програмісти. Залежно від того, який підхід використовується (Code-First або DB-First), робочий шар може бути змодельований або за допомогою графічного дизайнера Visual Studio, або за допомогою коду. Після цього у програмістів з'являється широкий інструментарій для роботи з Entity Framework. Синтаксис робочого шару описується за допомогою мови Conceptual Schema Definition Language (CSDL).

Віддалений шар визначає таблиці, стовпці, рядки, відносини між таблицями бази даних. Синтаксис віддаленого шару описується за допомогою мови Store Schema Definition Language (SSDL).

Сполучний шар визначає відповідність між робочим і віддаленим шарами, він пов'язує властивості сутнісного класу в робочому шарі за допомогою стовпців таблиці бази даних у віддаленому шарі. Керувати цим шаром (тобто деталями прив'язки) можна з вікна Mapping Deatails, що знаходиться в інструментах дизайнера Visual Studio, або за допомогою анотацій Fluent API, якщо використовується підхід Code-First. Мова Mapping Specification Language (MSL) визначає синтаксис сполучного шару.

Важливо відзначити, що мови CSDL, SSDL і MSL мають синтаксис XML, але при цьому використовують різну семантику.

Способи взаємодії з БД

Entity Framework передбачає три можливі способи взаємодії з базою даних:

Database first (БД спочатку): Entity Framework створює набір класів, які відображають модель конкретної бази даних.

Model first (модель спочатку): спочатку розробник створює модель бази даних, за якою потім Entity Framework створює реальну базу даних на сервері.

Code first (код спочатку): розробник створює клас моделі даних, які будуть зберігатися в БД, а потім Entity Framework за цією моделлю генерує базу даних і її таблиці.

У табл. 1 наведено різні підходи для роботи з Entity Framework.

Таблиця 1

Використання БД	Орієнтація підходу	Підхід	Опис
Нова БД	На графічну модель	Model-First	Якщо необхідно створити нову базу даних і є необхідність побачити дизайн бази в графічному вигляді, підхід Model-First працює найкраще. Model-First використовує робочий процес: Можете створити модель, використовуючи графічний конструктор EDMX. Генеруєте базу даних на основі цієї моделі. Entity Framework автоматично генерує класи, необхідні для взаємодії з базою даних.
Існуюча БД	На графічну модель	Database-First	Якщо вже є база даних і є необхідність маніпулювати нею, використовуючи графічний редактор, то підхід Database-First працює найкраще. Database-First використовує робочий процес: Генеруєте модель EDMX з існуючої

Закінчення табл. 1

Використання БД	Орієнтація підходу	Підхід	Опис
			бази даних. Entity Framework автоматично генерує класи, необхідні для взаємодії з базою даних.
Нова БД	На код	Code-First	Якщо необхідно створити нову базу даних і прийнято рішення працювати з кодом, то найкращим підходом буде Code-First, який використовує робочий процес: Вручну створюються класи, які визначають дані в кожній таблиці бази даних. Визначаєте відносини між таблицями за допомогою різних атрибутів і віртуальних методів. При необхідності, також налаштовуєте деталі підключення (наприклад, рядок підключення). Entity Framework автоматично формує базу даних під час виконання.
Існуюча БД	На код	Code-First (Code-Second)	Якщо для існуючої бази даних хочете вручну створити модель у коді, то потрібно використовувати підхід Code-First, який деякі розробники також називають Code-Second, тому що він застосовується до вже існуючої бази даних. Робочий процес виглядає таким чином: перепроектування класів, які визначають дані для кожної таблиці в базі даних за допомогою коду. Відображаєте відносини між таблицями в базі даних на код моделі.

База даних спочатку (Database first)

Такий підхід передбачає, що в першу чергу проектується і розробляється база даних. Це може бути зроблено за допомогою будь-яких доступних розробнику інструментів. Після цього на її основі Entity Framework створить опис EDM і класи концептуальної моделі.

При такому варіанті проектування архітектури додатку головна роль відводиться структурі бази даних. Класи бізнес-логіки змушені адаптуватися під неї. Однак це дозволяє максимально розкрити потенціал використовуваної системи керування базами даних.

Щоб використовувати розглянутий підхід у своєму проекті, необхідно вибрати пункт "Add Item" у відповідному меню проекту і додати опис Моделі даних Entity (ADO.NET Entity Data Model).

У діалозі "Entity Data Model Wizard" потрібно вибрати варіант "Generate from a database". Після цього потрібно буде вказати базу даних і параметри з'єднання з нею (вибрати або створити рядок з'єднання). Унаслідок цього в проект буде додано EDMX-файл, який містить опис EDM у форматі XML.

Кожен представлений клас у термінології Entity Framework називається сутністю. Їх властивості розділені на дві групи:

1. Пов'язані безпосередньо з полями бази даних: Id, Title і т. д.
2. Навігаційні: Language, Publisher, Tags.

Останні не мають прямих аналогів серед полів бази даних і створені виходячи з аналізу зв'язків таблиць. Вони дозволяють зручно і просто запитувати пов'язану з певною сутністю інформацію. Наприклад, список книг певною мовою можна отримати, використовуючи колекцію BookDetails у відповідного екземпляру Language.

Крім того, варто звернути увагу, що між сутностями вказані не тільки зв'язки, але і їх тип. При цьому символ "зірочка" позначає необмежену кількість елементів (many). Отже, можуть бути такі варіанти:

* -1 - тип One-to-many (одна до багатьох). Тобто будь-якому запису в таблиці BookDetails відповідає один запис в Language. І навпаки, одному запису в таблиці Language може відповідати будь-яке число записів в BookDetails.

* -0..1 - також тип One-to-many (одна до багатьох), але цей зв'язок не обов'язково повинний існувати для кожного запису в таблиці BookDetails.

* - * - цей варіант вказує на зв'язок типу Many-to-many (багато до багатьох). Зокрема, будь-якій книзі (BookDetails) відповідає будь-яке число ключових слів (Tag), як і навпаки.

Модель спочатку (Model first)

Цей підхід є протилежністю попереднього варіанта. При його використанні спочатку в дизайнері створюється опис EDM, керуючись вимогами бізнес-логіки. Після чого на його основі генерується база даних.

Необхідно акуратно використовувати цей підхід, тому що неправильні архітектурні рішення можуть завдати шкоди продуктивності бази даних, а значить, і додатку в цілому.

Додавання в проект опису EDM практично аналогічне. Відмінність тільки в діалозі "Entity Data Model Wizard", де необхідно вибрати пункт "Empty Model". Після завершення робіт зі створення Моделі залишається тільки згенерувати базу даних. Для цього потрібно вибрати пункт "Generate Database from Model" у відповідному меню дизайнера.

Код спочатку (Code First)

У розглянутих вище варіантах класи Моделі отримують шляхом їх автоматичної генерації. А якщо класи Моделі вже є? Або, наприклад, зручніше написати їх C# код, ніж малювати в дизайнері?

Починаючи з версії 4.1 в Entity Framework ще один підхід до розробки опису EDM – «Код спочатку». З його допомогою можна створити базу даних на основі класів C# або Visual Basic. Причому для цього достатньо їх найпростішого варіанта – POCO (Plain Old CLR Object).

Для впровадження цього підходу досить вказати Entity Framework використовувані типи. Велика частина роботи виконується на основі погоджень. Однак при необхідності можна використовувати атрибути для задання необхідних параметрів.

Такий підхід дозволяє дуже скоротити час розроблення на початковому етапі. Наприклад, при перевірці якоїсь ідеї розробник може повністю зосередитися на Моделі і бізнес-логіці, відклавши на деякий час питання про базу даних.

Database First

Database First був першим підходом, який з'явився в Entity Framework. Цей підхід багато в чому схожий на Model First і підходить для тих випадків, коли розробник вже має готову базу даних.

Щоб Entity Framework міг отримати доступ до бази даних, у системі повинен бути встановлений відповідний провайдер. Так, Visual Studio вже підтримує відповідну інфраструктуру для СКБД MS SQL Server. Для інших СКБД, наприклад, MySQL, Oracle треба встановлювати відповідні провайдери. Список провайдерів для найбільш поширених СКБД можна знайти на сторінці ADO.NET Data Providers.

Лекція 15. СТВОРЕННЯ ПРОЕКТУ ENTITY FRAMEWORK

Створимо новий проект по типу Console Application. Щоб задіяти базу даних, нам треба її мати в наявності. Створимо нову БД або візьмемо вже наявну.

У Visual Studio у вікні Solution Explorer натиснемо на проект правою кнопкою миші і виберемо в Add -> New Item. Далі у вікні додавання нового елемента виберемо ADO.NET Entity Data Model. Дамо новому компоненту назву.

Після цього відкриється вікно майстра створення моделі. У ньому потрібно вибрати опцію EF Designer from database.

Потім відкриється вікно наступного кроку зі створення моделі, на якому треба буде встановити підключення до бази даних:

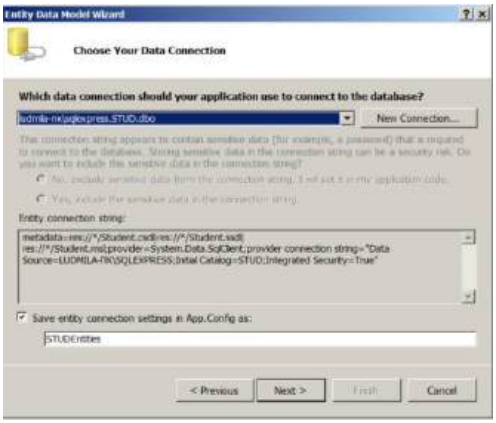

Рис. 21. Підключення до БД

У випадному списку виберемо одне з доступних підключень. Якщо в списку немає бажаних підключень, то можна натиснути на кнопку New Connection і встановити нове підключення.

Також унизу вказується назва контексту даних, який буде використовуватися для доступу до даних. За замовчуванням контекст має назву STUDEntities. Можна змінити, а можна і залишити.

Далі Visual Studio витягує всю інформацію про базу даних:

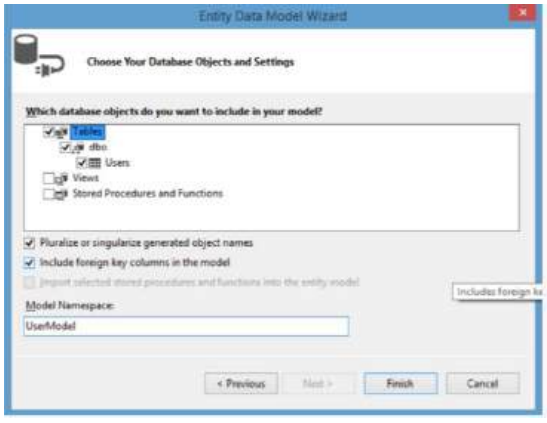

Рис. 22. Інформація про БД

Розкриємо вузол Tables. Він відображає всі таблиці, наявні в базі даних. У цьому випадку є тільки одна таблиця Students. Відзначимо всі підвузли в гілці Tables.

У полі Model Namespace встановимо ім'я моделі і натиснемо Finish. Після цього Entity Framework згенерує модель по базі даних і додасть її в проект.

Visiual Studio відобразить схему моделі. В цьому випадку в БД є тільки одна таблиця, тому на схемі відображається тільки одна сутність STUDENTS_ONE : (рис. 23).

Після виділення сутності в правому нижньому кутку Visual Studio побачимо властивості для цієї сутності. Властивість Name у вікні властивостей вказує на клас, яким буде представлена ця сутність. А властивість Entity Set Name вказує на ім'я набору об'єктів (тобто властивість DbSet контексту даних).

Використання оператора using. Не треба плутати цей оператор з директивою using, яка підключає простір імен на початку файлу коду. Оператор using дозволяє створювати об'єкт у блоці коду, по завершенню якого викликається метод Dispose у цього об'єкта і, таким чином, об'єкт знищується.

Крім того, в контексті using допускається оголошувати кілька об'єктів одного і того ж типу. Як не важко здогадатися, в такому випадку компілятор буде вставляти код з викликом Dispose () для кожного оголошеного об'єкта.

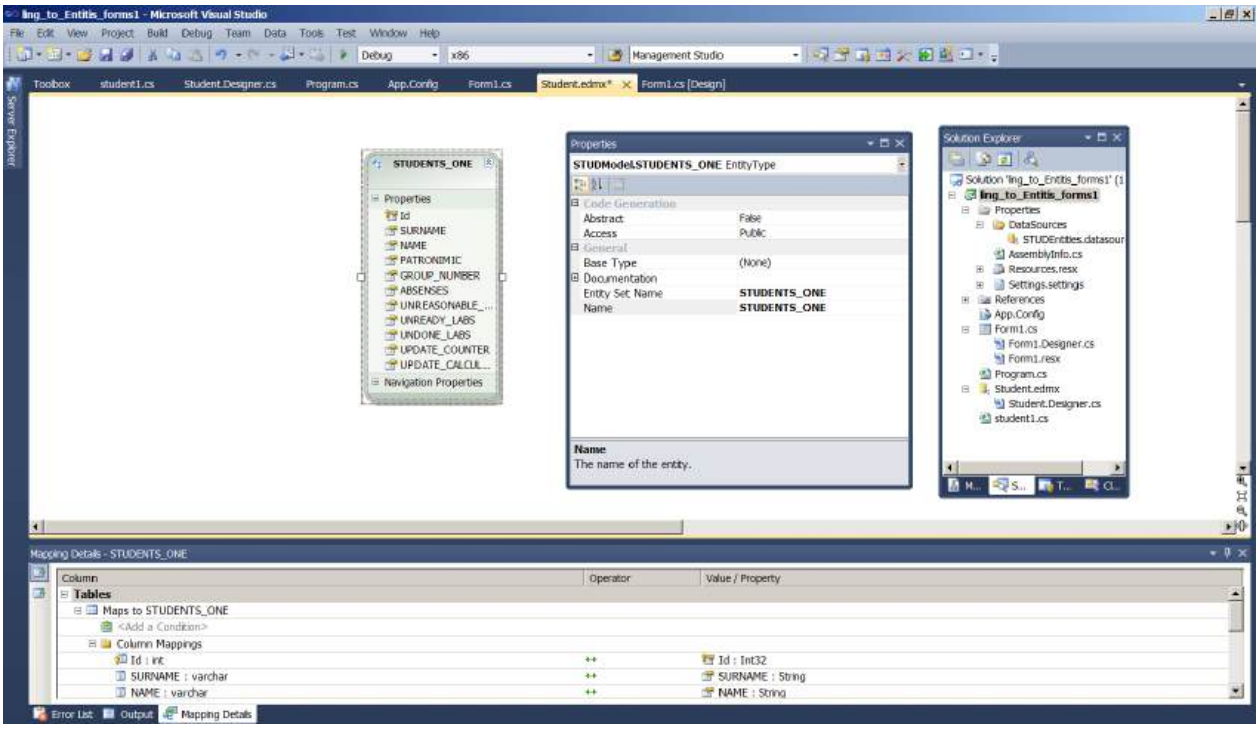

Рис. 23. Сутнісна модель даних

Вибираємо мову запитів

Як вже було зазначено, для створення запиту клієнт може використовувати одну з двох мов:

Entity SQL – це не залежна від сховища мова запитів. Зовні схожа зі стандартною SQL, але відрізняється від неї функціями. Необхідно відзначити, що тільки з використанням Entity SQL можна безпосередньо звертатися до клієнтських провайдерів даних Entity, минаючи шар Служби об'єктів. У деяких сценаріях це може дати великий приріст продуктивності.

LINQ to Entities є діалектом мови запитів LINQ для Entity Framework і використовує його синтаксис. Варто відзначити, що підтримуються не всі доступні методи. При цьому їх використання не є помилкою на етапі компіляції. Однак у процесі виконання програми буде викинуто виключення NotSupportedException.

LINQ to Entities

LINQ – технологія Microsoft, призначена для підтримки запитів до даних усіх типів на рівні мови. Ці типи включають масиви та колекції в пам'яті, бази даних, документи XML і багато іншого [7].

LINQ включає в себе близько 50 стандартних операцій запитів, поділених на дві великі групи – відкладені операції (виконуються не під час ініціалізації, а тільки при їх виклику) і не відкладені операції (виконуються відразу). На рис. 24 наочно показана "градація" операцій LINQ.

Здебільшого мова LINQ орієнтована на запити – це можуть бути запити, які повертають набір відповідних об'єктів, єдиний об'єкт або підмножину полів з об'єкта або набору об'єктів. У LINQ цей повернутий набір називається послідовністю (sequence). Більшість послідовностей LINQ мають тип IEnumerable <T>, де T – тип даних об'єктів, що знаходяться в послідовності. Наприклад, якщо є послідовність цілих чисел, вони повинні зберігатися у змінній типу IEnumerable <int>. IEnumerable <T> буквально панує в LINQ і дуже багато методів LINQ повертають саме тип IEnumerable <T>.

```
string [] numbers = { "40", "2012", "176", "5"};
// Перетворимо масив рядків в масив типу int, використовуючи LINQ
int [] nums = numbers.Select (s => Int32.Parse (s)). ToArray ();
foreach (int n in nums)
 Console.Write (n + "");
```

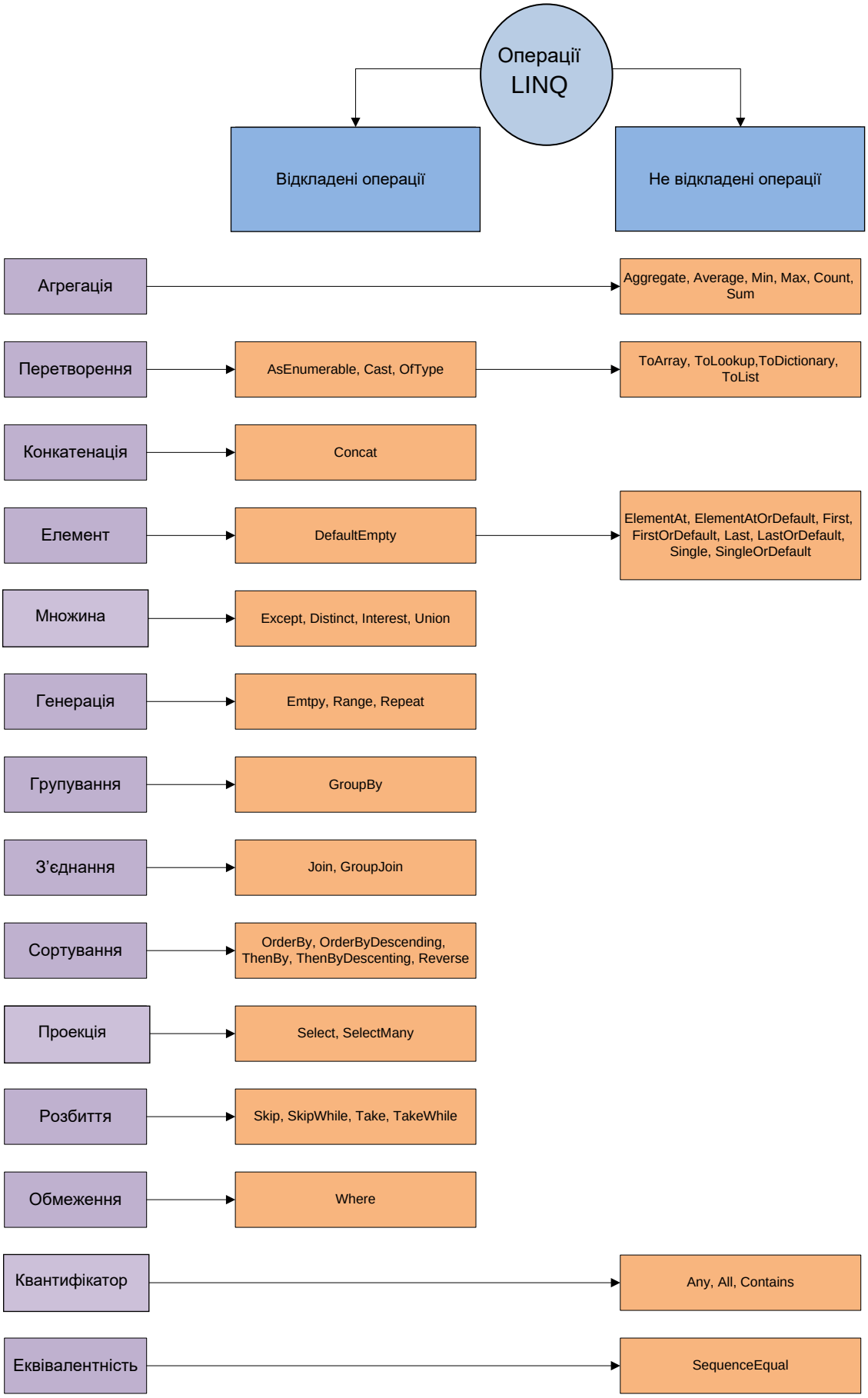

Рис. 24. Операції LINQ

Базовими одиницями даних у LINQ є послідовності й елементи. Послідовність – це будь-який об'єкт, що реалізує узагальнений інтерфейс IEnumerable, а елемент – це просто елемент послідовності.

У наступному прикладі масив рядків names – це послідовність, а Tom, Dick і Harry – елементи:

string [] names = {"Tom", "Dick", "Harry"};

Така послідовність називається локальною, тому що являє собою локальну колекцію об'єктів у пам'яті.

Оператор запиту – це метод, що перетворює послідовність. У типовому випадку оператор запиту приймає вхідні последовності і повертає результат перетворення – вихідну послідовність.

Запит – це вираз, який перетворює послідовності за допомогою операторів запиту. Найпростіший запит складається з однієї вхідної послідовності і одного оператора. Наприклад, можемо застосувати оператор Where до строкового масиву і витягти із нього елементи, довжина яких не менше чотирьох символів:

```
string [] names = { "Tom", "Dick", "Harry"};
IEnumerable <string> filteredNames
System.Linq.Enumerable.Where
(Names, n => n.Length> = 4);
foreach (string n in filteredNames)
Console.Write (n + "1");
```

Dick1 Harry1

Оскільки стандартні оператори запиту реалізовані у вигляді методів розширення, можемо викликати Where безпосередньо для масиву names так, немов це метод екземпляру:

```
 IEnwneraИe <string> filteredNames:
string [] names = { "Tom", "Dick", "Harry");
IEnumerable <string> filteredNames = names.Where (n => n.Length> = 4);
foreach (strinп name iп filteredNames)
Conѕole.Write (name + "1");
```

Можна було б ще більше скоротити запит за допомогою неявного приведення типу змінної filteredNames:

var filteredNames = пames.Where (n => n.Lепgth> = 4);

Більшість операторів запиту приймає лямбда-вираз як аргумент. Лямбда-вираз допомагає направити і сформувати запит. У нашому прикладі лямбда-вираз виглядає так:

n => n.Lепgth> = 4.

Вхідний аргумент відповідає вихідному елементу. У цьому випадку вхідний аргумент n представляє ім'я масиву і має тип string. Оператор Where вимагає, щоб лямбда-вираз повертав значення типу bool. Коли воно істинне, елемент повинен бути включений у вихідну послідовність. У межах цього курсу лекцій будемо називати такі запити лямбда-запитами. У мові C# є спеціальний синтаксис для написання запитів, і він називається синтаксисом, що полегшує сприйняття запиту. Перепишемо попередній приклад відповідно до цього синтаксису:

```
IEnumerable <string> filteredNarnes =
from n in narnes
where n.Contains ( "a")
select n;
```

Лямбда-синтаксис і синтаксис, який полегшує сприйняття, доповнюють один одного.

Лямбда-вираз (lambda expression) – це засіб C#, що є скороченим способом позначення анонімних методів.

Інтеграція LINQ з C#

Щоб забезпечити безперешкодну інтеграцію LINQ з C#, до мови C# знадобилося внести істотні вдосконалення. Хоча всі ці засоби цінні і самі по собі, вони є частинами загального вкладу в LINQ, який робить розширення C# достатньо зручним.

Щоб дійсно зрозуміти більшу частину синтаксису LINQ, необхідно спочатку розібратися в деяких інструментах мови C# і тільки потім почати роботу з компонентами LINQ.

Лямбда-вирази

Починаючи з версії 3, мова C# підтримує **лямбда-вирази (lambda expressions).** Лямбда-вирази використовувалися в мовах програмування на зразок LISP з давніх часів, а вперше їх концепція була сформульована 1936 року американським математиком Алонзо Черчем (Alonzo Church). Ці вирази – скорочений синтаксис для визначення алгоритму.

Але, перш ніж звернутися безпосередньо до лямбда-виразів, давайте поглянемо на еволюцію способів позначення алгоритму як аргументу методу, оскільки саме в цьому і полягає призначення лямбда-виразів.

Використання іменованих методів

Раніше, коли метод або змінна були типізовані так, що вимагали делегата (delegate), розробник повинен був створювати іменований метод і передавати його ім'я туди, де був потрібний делегат.

Як приклад розглянемо таку ситуацію. Припустимо, що є два розробника, один з яких займається кодом загального призначення, а інший – прикладним. Не обов'язково, щоб це були два різних розробника,

нам просто потрібно розмежувати дві різні ролі. Розробник загального коду хоче створювати код загального призначення, який може бути використаний багаторазово в усьому проекті. Прикладний розробник буде формувати код загального призначення, щоб створювати додаток.

У цьому випадку розробник загального коду бажає створити метод для фільтрації масивів цілих чисел, але з можливістю позначення алгоритму, що застосовується для фільтрації. Для початку має бути оголошений делегат. Цей делегат буде прототиповано для прийому параметра int і повернення значення true, якщо цей int потрібно включити у відфільтрований масив.

Отже, він створює службовий клас і додає делегат і метод фільтрації. Ось цей код загального призначення:

```
public class Common
 {
 public delegate bool IntFilter (int i);
 public static int [] FilterArrayOfInts (int [] ints, IntFilter filter)
 {
 ArrayList aList = new ArrayList ();
 foreach (int i in ints)
 {
 if (filter (i))
 {
 aList.Add (i);
 }
 }
 return ((int []) aList.ToArray (typeof (int)));
 }
 }
```

Розробник загального коду помістить і оголошення делегата, і FilterArrayOfInts в загальну бібліотечну збірку – бібліотеку, що динамічно підключається (DLL) – щоб його можна було використовувати в багатьох додатках.

Наведений вище метод FilterArrayOfInts дозволяє прикладному розробнику передавати масив цілих чисел і делегат його методу фільтрації, отримуючи назад відфільтрований масив.

Тепер припустимо, що прикладний розробник бажає відфільтрувати тільки непарні числа. Ось його метод фільтрації, який оголошено в прикладному коді:

```
public class Application
 {
 public static bool IsOdd (int i)
 {
 return ((i & 1) == 1);
 }
```

```
}
```

На основі коду методу FilterArrayOfInts цей метод буде викликаний для кожного значення int у масиві, переданому йому. Фільтр поверне true, якщо передане значення є непарним. Нижче показаний приклад використання методу FilterArrayOfInts, за яким представлений результат:

```
class Program
{
static void Main ()
{
int [] nums = {1,2,3,4,5,6,7,8,9,10};
int [] oddNums = Common.FilterArrayOfInts (nums, Application.IsOdd);
foreach (int i in oddNums)
Console.WriteLine (i);
}
}
```

Зверніть увагу, що для передачі делегата в другому параметрі FilterArrayOfInts прикладний розробник передає ім'я методу. Просто створивши інший фільтр, він може фільтрувати числа інакше. Він може мати фільтр для парних чисел, простих чисел або відібраних за будь-якими потрібними критеріями. Делегати дозволяють створювати код повторного використання.

Використання анонімних методів

Так чи інакше, але написання всіх цих методів фільтрації або будь-яких інших методів delegate може виявитися досить виснажливим. Багато з цих методів будуть використані лише одноразово, і нудно створювати іменовані методи для таких випадків. Починаючи з версії C#2.0, у розробників з'явилася можливість створювати екземпляр делегата за рахунок надання вбудованого коду як анонімного методу. Анонімні методи дозволяють розробнику вказувати код практично скрізь, де зазвичай повинен передаватися делегат. Замість створення методу IsOdd він може написати код фільтрації прямо в точці, де зазвичай передається делегат:

```
int [] nums = {1,2,3,4,5,6,7,8,9,10};
int [] oddNums = Common.FilterArrayOfInts (nums, delegate (int i)
{Return ((i & 1) == 1); });
foreach (int i in oddNums)
 Console.WriteLine (i);
```

Прикладний розробник більш не зобов'язаний десь оголошувати метод. Це чудово для коду логіки фільтрації, ймовірність багаторазового використання якого невисока. Як і очікувалося, висновок програми не відрізняється від попереднього.

Із застосуванням анонімних методів пов'язаний один недолік. Одержуваний за допомогою код досить громіздкий і важко читається. Повинен існувати більш зручний спосіб написання коду методу.

Використання лямбда-виразів (синтаксис запитів на основі методів)

Лямбда-вирази визначаються як розділений комами список параметрів, за яким слідує лямбда-операція, а за нею – вираз або блок операторів. Якщо параметрів більше одного, вхідні параметри поміщаються в дужки. У C# лямбда-операція записується як =>. Отже, лямбда-вираз у C# виглядає подібно до такого:

(Параметр1, параметр2, параметр3) => вираз

Або, коли потрібна більш складна логіка, може застосовуватися блок операторів:

```
(Параметр1, параметр2, параметр3) =>
{
 оператор1;
 оператор2;
 оператор3;
 return (тип_повернення_лямбда_виразу);
}
```

У цьому прикладі тип даних, що повертається в кінці блоку операторів, повинен відповідати типу повернення, вказаному делегатом. Ось приклад лямбда-виразу:

x => x – цей лямбда-вираз може бути прочитано, як "x йде до x" або, можливо, "введення x повертає x". Це означає, що при вхідній змінній x вираз поверне x. Цей вираз просто повертає те, що отримав. Оскільки тут єдиний параметр x, немає необхідності брати його в дужки. Важливо знати, що цей делегат диктує тип вхідного параметра x і тип повернення.

Наприклад, якщо делегат визначає на вході string, але повертає bool, тоді вираз x => x не може використовуватися, бо якщо вхідний x буде мати тип string, то повертається x, який також повинен відноситися до типу string, але делегат визначає тип повернення bool. Тому з delegate права частина лямбда-виразу повина обчислюватися для повернення bool, як показано в прикладі:

x => x.Length> 0 – цей лямбда-вираз може бути прочитаний як "x йде в x.Length> 0", або, можливо, "введення x повертає x.Length> 0". Оскільки права частина виразу обчислюється як bool, делегат повинен вказувати, що метод повертає bool, інакше компілятор повідомить про помилку.

Лямбда-вираз намагається повернути довжину вхідного аргументу. Значить, делегат повинен визначати int як тип повернення:

s => s.Length.

Якщо лямбда-виразу передається кілька параметрів, відокремлюйте їх комами і розміщуйте в дужки, як показано нижче:

(X, y) => x == y.

Складні лямбда-вирази можуть навіть включати блок операторів:

```
(x, y) =>
{
 if (x> y)
 return (x);
 else
 return (y);
}
```

Важливо пам'ятати, що делегат визначає, якими мають бути типи вхідних параметрів і яким – тип повернення. Тому переконайтеся, що лямбда-вираз відповідає визначенню делегата.

Переконайтеся, що лямбда-вираз розрахований на прийом вхідних типів, зазначених у визначенні делегата, і повертає необхідний тип делегата.

Нижче наведено оголошення делегата розробником загального коду:

delegate bool IntFilter (int i);.

Лямбда-вираз розробника прикладного коду має підтримувати int, переданий у параметрі, і повертати bool.

Попередній приклад, у якому на цей раз використовується лямбда-вираз, повинен виглядати так:

```
int [] nums = {1,2,3,4,5,6,7,8,9,10};
int [] oddNums = Common.FilterArrayOfInts (nums, i => ((i & 1) == 1));
foreach (int i in oddNums)
 Console.WriteLine (i);
```

Використовувати іменовані методи, анонімні методи або лямбда-вирази – вибір за розробником. Застосовуйте те, що має сенс у кожній конкретній ситуації.

Перевагами лямбда-виразів часто користуються при передачі їх як аргументів на виклики операцій запитів LINQ. Оскільки кожен запит LINQ, швидше за все, буде мати унікальний або дуже обмежений до використання лямбда-вираз, це забезпечує гнучкість вказівки логіки операції без необхідності постійного створення методів майже для кожного запиту.

Дерева виразів (синтаксис виразів запитів, синтаксис запитів на основі виразів)

Дерево виразів (expression tree) – ефективне подання в деревоподібній формі даних лямбда-виразів операції запиту. Ці подання дерев виразів можуть бути обчислені всі відразу, так що єдиний запит може бути побудований і виконаний на одному джерелі даних, такому як база даних.

У більшості прикладів, розглянутих досі, операції виразів виконувалися в лінійній манері. Розглянемо такий код:

```
int [] nums = new int [] {6, 2, 7, 1, 9, 3};
IEnumerable <int> numsLessThanFour = nums
 .Where (i => i <4)
 .OrderBy (i => i);
```

Цей запит містить дві операції – Where і OrderBy, які очікують делегати як свої аргументи. Унаслідок компіляції генерується код проміжною мовою .NET (Intermediate Language – IL), ідентичний IL-коду анонімного методу для кожного лямбда-виразу операції запиту.

Коли виконується цей запит, спочатку викликається операція Where, за нею – операція OrderBy. Таке лінійне виконання операцій здається виправданим для наведеного прикладу, але давайте подумаємо про запит до дуже великого джерела, такого як база даних. Чи має сенс для SQL-запиту спочатку звернутися до бази даних тільки з конструкцією Where, щоб змінити порядок подальших викликів? Зрозуміло, що це не реально для запитів до бази даних. Саме тут приходять на допомогу дерева виразів. Оскільки дерево виразів допускає паралельне обчислення і виконання всіх операцій у запиті, то можливо зробити єдиний загальний запит замість окремих запитів для кожної операції.

Отже, тепер є дві різні речі, які може генерувати компілятор для лямбда-виразу операції – IL-код і дерево виразу. Від чого залежить, буде лямбда-вираз операції компілюватися в IL-код або в дерево виразу? Яка з цих двох дій зробить компілятор визначальним прототипом операції? Якщо операція оголошена для прийому делегата методу, буде згенеровано IL-код. Якщо ж операція оголошена для прийому виразу делегата, буде створено дерево виразу.

Розглянемо дві різні реалізації операції Where. Перша – стандартна операція запиту, присутня в API-інтерфейсі LINQ to Objects і визначена в класі.

System.Linq.Enumerable:

public static IEnumerable <T> Where <T> (this IEnumerable <T> source, Func <T, bool> predicate);.

Друга реалізація операції Where знаходиться в API-інтерфейсі LINQ to SQL і належить класу System.Linq.Queryable:

public static IQueryable <T> Where <T> (this IQueryable <T> source, System.Linq.Expressions.Expression (Func <int, bool> predicate);.

Як бачимо, перша операція Where оголошена як приймаючий делегат, на що вказує делегат Func, і компілятор для цього лямбда-виразу операції згенерує IL-код. Потрібно мати на увазі, що делегат Func визначає сигнатуру делегата, переданого як аргумент предиката. Друга операція Where оголошена для прийому дерева виразу (Expression), тому тут компілятор згенерує деревоподібне подання лямбда-виразу.

Лекція 16. ЗАПИТИ LINQ

Одним із привабливих для розробників засобів LINQ є SQL-подібний синтаксис, доступний у LINQ-запитах. Цей синтаксис використовувався в декількох прикладах LINQ, наведених раніше. Синтаксис надано через розширення мови C#, яке називається **вирази запитів**. Вирази запитів дозволяють запитам LINQ приймати форму, подібну SQL, всього лише з рядом невеликих відмінностей.

Для виконання запиту LINQ вирази запитів не обов'язкові. Альтернативою є використання стандартної крапкової нотації C# з викликом методів на об'єктах і класах. У багатьох випадках застосування стандартної крапкової нотації виявляється кращим, оскільки вона більш наочно демонструє, що насправді відбувається і коли.

При записі запиту в стандартній крапковій нотації не відбувається ніякої трансформації при компіляції. Саме тому в багатьох прикладах не використовується синтаксис виразів запитів, а перевага віддається стандартному синтаксису крапкової нотації. Однак привабливість синтаксису виразів запитів безперечна.

Отримати уявлення про відмінності між цими двома синтаксисами найкраще на прикладі:

```
string [] names = { "Adams", "Arthur", "Buchanan", "Bush", "Carter", "Cleveland", "Taft", "Taylor", "Truman", "Tyler", "Van Buren" , "Washington", "Wilson"}; // Використання крапкової нотації
 IEnumerable <string> sequence = names
 .Where (n => n.Length <6)
 .Select (n => n);
 // Використання синтаксису виразу запиту
 IEnumerable <string> sequence = from n in names
 where n.Length <6
 select n;
 foreach (string name in sequence)
 {
```

```
Console.WriteLine ( "{0}", name);
}
```

Перше, що можна помітити в прикладі з виразом запиту – це те, що на відміну від SQL, операція from передує операції select. Однією з причин цього була необхідність звуження контексту для засобу IntelliSense. Без такої інверсії операцій введення в текстовому редакторі Visual Studio слова "select", за яким слідує пробіл, не дозволить засобу IntelliSense визначити, які змінні повинні відображатися в його списку. Контекст допустимих змінних у цій точці нічим не обмежений. Якщо ж спочатку вказати, звідки надходять дані, то IntelliSense отримує контекст і може надати список змінних для вибору.

Важливо відзначити, що синтаксис виразів запитів підтримується тільки для найбільш поширених операцій запитів: Where, Select, SelectMany, Join, GroupJoin, GroupBy, OrderBy, ThenBy, OrderByDescending і ThenByDescending.

Граматика виразів запитів

Вирази запитів повинні підкорятися перерахованим нижче правилам:

1. Вираз повинен починатися з конструкції from.
2. Інша частина виразу може містити нуль або більше конструкцій from, let або where. Конструкція from – це генератор, який оголошує одну або більше змінних діапазону, які перераховують послідовність або з'єднання декількох послідовностей. Конструкція let являє собою змінну діапазону і привласнює їй значення. Конструкція where фільтрує елементи з вхідної послідовності або з'єднання кількох вхідних послідовностей у вихідну послідовність.
3. Інша частина виразу запиту може потім включати конструкцію orderby, яка містить одне або більше полів сортування з необов'язковим напрямком упорядкування. Напрямок може бути ascending (за зростанням) або descending (за спадною).
4. Потім у виразі може йти конструкція select або group.
5. Нарешті, у виразі може слідувати необов'язкова конструкція продовження. Такою конструкцією може бути або into, нуль або більше конструкцій join, або ж інша актуальна послідовність перерахованих елементів, починаючи з конструкцій з п. 2. Конструкція into направляє результати запиту в уявну вихідну послідовність, яка служить конструкцією from для наступних виразів запитів, починаючи з конструкцій із п. 2.

На рис. 25 наведено приклад використання технології Entity Framework:

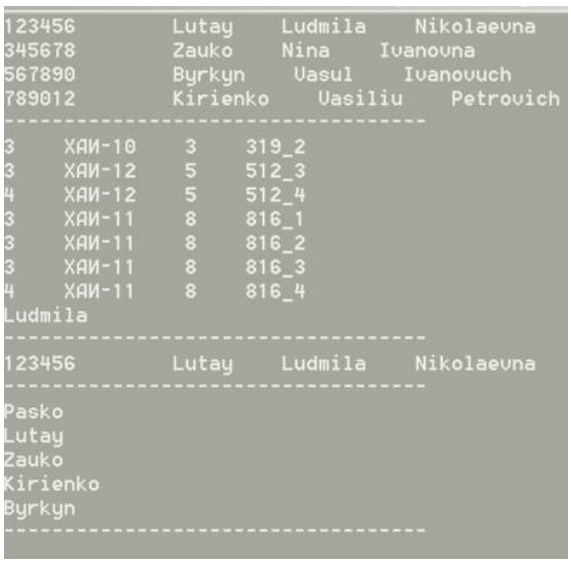

Рис. 25. Використання технології

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// using System.Data.Entity;
// using System.Data;
// using System.Data.SqlClient;

namespace ADO_NET_EF_example2
{
 class Program
 {
 static void Main (string [] args)
 {
 using (New_database1Entities db = new New_database1Entities ())
 {
 var st = db.students.Select (p => p);
 foreach (students s in st)
 Console.WriteLine ( "{0} {1} {2} {3}", s.nom_zach, s.fam, s.name, s.och);
 Console.WriteLine ( "----------------------------------");
 var rm = db.Room.Select (p => p);
 foreach (Room_com r in rm)
Console.WriteLine ( "{0} {1} {2} {3}", r.nomCap, r.nomGurt, r.nomLevel,
r.roomID);

 // var studs = from p in db.students
 // where p.nom_zach == "123456"
 // select p;
```

```
 ////////////////////////////////////////////////// ////////////
 var studs = db.students.Where (p => p.nom_zach == "123456");
 ////////////////////////////////////////////////// ////////////
 students myname = db.students.FirstOrDefault (p => p.fam == "Lutay");
 if (myname! = null)
 Console.WriteLine (myname.name);
 Console.WriteLine ( "---------------------------------");
 ////////////////////////////////////////////////// ////////////
 foreach (var s in studs)
Console.WriteLine ( "{0} {1} {2} {3}", s.nom_zach, s.fam, s.name, s.och);
 Console.WriteLine ( "---------------------------------");
 ////////////////////////////////////////////////// ////////////
 students st1 = new students (); // name = "name", och = "och");
 st1.nom_zach = "678955";
 st1.fam = "Kovalenko";
 st1.name = "Taras";
 st1.och = "Iosupovuch";
 db.students.AddObject (st1);
 db.SaveChanges ();
 ////////////////////////////////////////////////// ////////////
 var ode_name = db.students.OrderBy (p => p.name);
 foreach (students p in ode_name)
 Console.WriteLine (p.fam);
 Console.WriteLine ( "---------------------------------");
 }
 Console.ReadKey ();
 }
 }
}
```

БІБЛІОГРАФІЧНІЙ СПИСОК

1. Mark, J. Price. C # 7.1 and .NET Core 2.0 – Modern Cross-Platform Development – Third Edition / Price Mark J. – Packt, 2017. – 800 p.

2. Andrew, Troelsen. Pro C # 7: With .NET and .NET Core 8th Edition /Troelsen Andrew, Japikse Philip. – Apress : USA 2017. – 2077 p.

3. Бен, Албахарі. C # 7.0. Довідник. Повний опис мови / Албахарі Бен, Албахарі Джозеф. – СПб. : Альфа-книга, 2018. – 1026 с.

4. Властивості в C # [Електронний ресурс] // Матеріали сайту C #. Уроки програмування з нуля. – Режим доступу : http://mycsharp.ru/ post/27/ 2013_07_12_svojstva_v_si-sharp_ avtomaticheskie_ svojstva_aksessory _get _i_set.html

5. Одинадцять типів сучасних баз даних: Короткі описи, схеми і приклади БД [Електронний ресурс] // Матеріали сайту Proglib. – Режим доступу : https://proglib.io/index.php/p/11-tipov-sovremennyh-baz-dannyh-ratkie-opisaniya-shemy-i-primery-bd-2020-0107?action=answer&comment=f632b05-3110-4644-aa05-7316a495bf9c.

6. Керівництво по Entity Framework Entities [Електронний ресурс] // Матеріали сайту Metanit. – Режим доступу: https://metanit.com/sharp/entityframework/.

7. LINQ to Entities [Електронний ресурс] // Матеріали сайту Professor Web. – Режим доступу : https://professorweb.ru/my/LINQ/ inq_entities/level14/ linq_to_entities_index.php.

ЗМІСТ

Навчальне видання

Лутай Людмила Миколаївна

ПРОЕКТУВАННЯ ІНФОРМАЦІЙНИХ СИСТЕМ ДЛЯ ВИРОБНИЧИХ ПРОЦЕСІВ

Редактор Н. В. Мазепа

Зв. план, 2021
Підписано до видання 30.11.2021
Ум. друк. арк. 6,9. Обл.-вид. арк. 7,75. Електронний ресурс

Видавець і виготовлювач
Національний аерокосмічний університет ім. М. Є. Жуковського
«Харківський авіаційний інститут»
61070, Харків-70, вул. Чкалова, 17
http://www.khai.edu
Видавничий центр «ХАІ»
61070, Харків-70, вул. Чкалова, 17
izdat@khai.edu

Свідоцтво про внесення суб'єкта видавничої справи
до Державного реєстру видавців, виготовлювачів і розповсюджувачів
видавничої продукції сер. ДК № 391 від 30.03.2001