Рекомендована література:

Базова

1. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений, 3-е изд.: Пер. с англ. / Г. Буч, Р. А. Максимчук, М. У. Энг и др. – М.: Вильямс, 2008. – 720 с.
2. Брила А. Ю. Основи об’єктно-орієнтованого програмування у C#. Методичні вказівки до лабораторних робіт для студентів І-го курсу математичного факультету спеціальності "Прикладна математика" / [А. Ю. Брила, П. П. Антосяк, М. І. Глебена та ін.]. – Ужгород, 2014. – 73 с.
3. Васильев А. C#. Объектно-ориентированное программирование. Учебный курс / Алексей Васильев. – СПб.: Питер, 2012. – 320 с.
4. Стиллмен Э. Изучаем C#. 3-е изд / Эндрю Стиллмен, Дженнифер Грин. – СПб.: Питер, 2014. – 816 с.
5. Троелсен Э. Язык программирования C# 6.0 и платформа .NET 4.6. 7-е изд. / Эндрю Троелсен, Филипп Джепикс. – СПб.: Диалектика-Вильямс, 2018. – 1440 с.
6. Албахари Д. C# 6.0. Справочник. Полное описание языка: Пер. с англ. / Джозеф Албахари, Бен Албахари. – М.: Вильямс, 2018. – 1040 с.
7. C#. Спецификация языка. Версия 5.0 / Microsoft Corporation, 2012. – 577 с.
8. Рихтер Дж. CLR via C#. Программирование на платформе Microsoft .NET Framework 2.0 на языке C#. Мастер-класс: Пер. с англ. / Дж. Рихтер. – СПб.: Питер, 2007. – 656 с.

Допоміжна

9. Буч Г. Объектно-ориентированный анализ и проектирование с примерами приложений на C++, 2-е изд.: Пер. с англ. / Г. Буч. – М.: "Издательство Бином" 1998. – 560 с.
10. Кнут Д. Е. Искусство программирования для ЭВМ. Т. 3: Сортировка и поиск / Д. Е. Кнут. – 2-е изд. – М.: Вильямс, 2008. – 824 с.
11. Жарков В. А. Самоучитель Жаркова по Visual Studio .Net: Visul Basic .NET, Visual C# .NET, Visual C++ .NET, Visual J# .NET / В. А. Жарков. – М.: Жарков Пресс, 2002. – 592 с.
12. Климов Л. П. C#. Советы программистам / Л. П. Климов. – СПб.: БХВ-Петербург, 2008. – 544 с.
13. Задачи по программированию / С. А. Абрамов,, Г. Г. Гнездилова, Е. Н. Капустина и др.. – М.: Наука, 2000. – 596 с.

Тема 1. Основні поняття і принципи об’єктно-орієнтованого аналізу та проектування

1. **Абстрагування.**
2. **Наслідування.**
3. **Інкапсуляція.**
4. **Поліморфізм.**

1. В своєму розвитку мови програмування поступово еволюціонують від алгоритмічних до об’єктно-орієнтованих. Застосування останнього підходу прискорює розробку програм, зменшує кількість потенційних помилок, хоча й може збільшити кількість використаної пам’яті. Переваги об‘єктно-орієнтованого підходу базуються на зменшені кількості повторюваних описів.

Абстрагування – це здатність описувати об’єкти оточуючого світу засобами мови програмування в рамках поставленої задачі.

В процесі програмування описують тип об’єкта, а під час завантаження програми створюються об’єкти обраного типу (наприклад, в wordi описаний тип об’єкта – документ, а в процесі в експлуатації користувач може обробляти декілька файлів-документів).

В C# типи об’єктів називають класами, а самі об’єкти екземплярами класу. Кожен об’єкт оточуючого світу здатний породжувати різні об’єкти в процесі програмування в рамках поставленої задачі. Наприклад, в банку для клієнта не зберігають дані про те, чи любить він оперу. Тобто в кожному проекті зберігаються лише ті дані, які необхідні для розв’язку задачі. Тому описувати об’єкти певного типу означає вказати, які дані про нього необхідно зберігати і які дії можливі над цими даними . Опис таких дій називається методом об’єкта. Тому фактично об’єкт – це сукупність даних та методів їх обробки. Для кожного об’єкта зберігають лише його дані, а методи зберігаються для типу об’єкта.

2. Наслідування – це здатність одного типу об’єкта бути породженим від іншого типу об’єкта.

В C# всі об’єкти породженні від загального класу **Tobject**. Ми будемо створювати власні форми, породженні від вже розробленого класу **Tform**. В кожній мові програмування використовується своя ієрархія об’єктів. Піднімаючись по ієрархії збільшується рівень абстрагування, а спускаючись рівень деталізації.

Наслідування прискорює розробку програм за рахунок зменшення кількості повторюваних описів та виправлення виявлених помилок.

Терміни:

– клас від якого породжений клас називається *батьківський клас (базовий, предок)*;
– породжений клас називається *нащадок (потомок)*;

В процесі наслідування спостерігається нарощування полів і методів.

Типи наслідування:

Найчастіше використовуються *одиночне наслідування*, тобто клас може породжуватися лише від одного класу. При *множинному наслідуванні* клас може наслідуватися від кількох класів, що прискорює розробку, але вимагає відслідковування сумісності (якщо два батьківські класи мають два методи з однаковою назвою, то для нащадка необхідно зазначити, який з уих двох класів буде використаний).

3. **Інкапсуляція** об’єктів передбачає:
   – Здатність поєднання полів, властивостей і методів їх обробки в описі самого класу;

- Незалежність об’єкта (всі потрібні поля і методи мають бути в об’єкті описані, а всі зайняті ресурси мають бути звільнені при знищенні об’єкта);
- Здатність класу приховувати від зовнішнього світу внутрішні деталі реалізації та відображати поля і методи, які можуть бути використані для подальшої розробки;

4. Поліморфізм (poli – багато; morfa – форма) – здатність різних об’єктів мати методи з однаковою назвою і по-різному реагувати в процесі їх виклику. Наприклад, різні компоненти форми (мітки, поля) мають процедури для перемальовування з однаковою назвою, але відтворюють потрібні елементи керування.

Поліморфізм дозволяє викликати методи об’єктів циклічно. В об’єктно-орієнтованому програмуванні одиночне зберігання методів для кожного класу об’єктів і множинне зберігання даних для кожного об’єкта.

Питання для самоконтролю

1. Базові принципи ООП.
2. Абстрагування в ООП.
3. Опис об'єктів в ООП.
4. Класи та об'єкти в ООП.
5. Наслідування в ООП.
6. Ієрархія об'єктів в ООП.
7. Інкапсуляція, як один з базових принципів ООП.
8. Використання поліморфізму в ООП.

Тема 2. Класи та їх складові. Конструктори класів

1. **Загальний синтаксис опису класів. Синтаксис опису складових класів.**
2. **Призначення та синтаксис опису методів.**
3. **Локальні змінні та константи.**
4. **Формальні та фактичні параметри методів.**
5. **Конструктори та деструктори.**

1. В C# класи використовуються в двох аспектах: клас як модуль і клас як тип даних.

При використанні **класу як модуля** C# дозволяє звертатися до поля чи методу класу без створення екземпляра. Для цього сам клас, його поля і методи мають бути статичними і описуватися з модифікатором **static**.

Класи-типи даних належать до посилкових типів і тому для кожного об'єкта такого класу необхідно виділяти місце під його дані з допомогою оператора **new**.

Загальний синтаксис опису класів:

[<атрибут класу>] [<модифікатор (рівень) доступу>] class **<назва класу>**
[:<назва батьківського класу>]
{<дані>
<методи>} .

Якщо батьківський клас не вказано, то наслідування автоматично відбувається від базового класу *Object*. Перед класом можуть використовуватися наступні **атрибути класу**:

- **partial** – розділюваний клас, що може описуватися в декількох місцях, при чому в одному місці може зазначатися заголовок методу, а в іншому – його реалізація;
- **abstract** – клас є базовим і використовується для породження інших класів. Створити екземпляр такого класу неможливо;
- **sealed** – заборона наслідування від даного класу.
- **unsafe** – дозволяє небезпечний код з використанням вказівників.

Крім цих атрибутів можуть застосовуватися такі **модифікатори доступу**:

- **public** – клас має необмежений доступ і може використовуватися у всьому проекті.
- **internal** – доступ обмежений поточною збіркою (по замовчуванню).

Складовими (**членами**) **класу можуть бути поля, властивості, методи, події та ін.** (наприклад, **константи, індексатори, вкладені типи**). **Для членів** класу визначені такі **модифікатори доступу**:

- **public** – член має необмежений доступ і може використовуватися у всьому рішенні;
- **private** – доступ обмежений лише своїм класом (по замовчуванню);
- **internal** – доступ обмежений поточною збіркою;
- **protected** – доступ обмежений активним класом і породженими класами.

2. Методи використання для виконання повторюваних операцій та для опису дій об’єктів. Типовий приклад методів `WriteLine`, який виводить дані користувача у вікно консольного додатку. Кожен метод має своє унікальне ім’я в сукупності з аргументами. Виклик методу виконується вказуванням його назви, після чого зазначаються круглі дужки, де перераховуються аргументи.

В C# використовуються статистичні методи та методи класу. Статистичні методи розміщуються в пам’яті один раз як і їхні змінні, перед назвою статистичного методу ставиться слово `static`. Методи класу створюються вказаного класу і їх змінні створюються для кожного екземпляра.

Синтексис опису статистичних методів:

```
static <тип результату> <назва методу>(<список аргументів>);
```

Для виходу з методу використовується оператор `return`. Якщо метод не повертає жодне значення в місце виклику, то замість типу його результату вказується слово `void`. Тип результату після `return` обов'язково має співпадати з типом результату підпрограми.

3. Всередині кожної підпрограми можуть використовуватися свої змінні і константи, які називають локальними, вони створюються під час виклику методу і зникають після завершення його роботи.кожній локальній змінній може відразу присвоюватися початкове значення. Наприклад: `int i = 0;` . C# забороняє використовувати змінні, доки їм не присвоєно значення.

4. В кожен метод можуть передаватися аргументи, які між собою відмежовані комами. Ім'я аргумента у списку аргументів і при виклику методу можуть не співпадати, але обов'язково має співпадати тип даних. Передача аргумента можлива по значенню і по змінній. Крім цього в C# ще й використовують аргументи результату.

При передачі по значенню створюються формальні параметри фактично формується нова змінна, в яку з основної програми заноситься початкове значення і надалі змінна програми і змінна підпрограми між собою незалежні.

При передачі за адресою дії відбуваються над змінними основної програми. Перед такою змінною в списку аргументів вказується службове слово ref, при цьому використовується фактичний параметр.

Для формування параметра виводу в списку аргументів і при виклику перед ним вказується слово out. В такий аргумент не заноситься початкове значення, а лише повертається результат.

Приклад:

Скласти підпрограму для коректного введення дійсного числа в діалоговому режимі.

Якщо не використовувати підпрограми, то фрагмент для коректного введення необхідно буде повторити при кожному введенні і використовувати різні мітки. Такі повтори необхідно буде використовувати і в інших програмах для введення дійсних чисел. Тому створимо власну підпрограму для введення дійсного числа і будемо її викликати потрібну кількість разів. Підпрограма може ввести і не ввести дані від користувача, тому буде повертати логічний результат.

```
static bool InputDouble(ref Double x, String Povidom)
 {String S;
  S = x.ToString();
 Povtor:
   S = Interaction.InputBox(Povidom, "Введення", S);
   try
   {x = Convert.ToDouble(S);
   }
   catch (System.FormatException)
   {if (MessageBox.Show("Ви ввели не число" + Strings.Chr(13) +
         Strings.Chr(13) + "Бажаєте повторити?", "Увага",
         MessageBoxButtons.YesNo, MessageBoxIcon.Warning) ==
        DialogResult.Yes)
     goto Povtor;
    else
      return false;
   }
  return true; }
```

5. Приклад опису класу. Створити клас *Паралелограм* з доступними процедурами визначення площі, периметра та друку його даних.

```
class Parallelogram
 {private double a, b, alfa;

  public double area()
   {return a * b * Math.Sin(alfa / 180 * Math.PI);
   }

  public double perimeter()
   {return 2 * (a + b);
   }

  public void Info()
   {MessageBox.Show("Дані паралелограма:\nдві сторони по "+a.ToString()+
        " та дві по "+b.ToString()+" од.;\nплоща: " + area().ToString() +
        " кв. од.;\nпериметр: " + perimeter().ToString() +
        " од.;\nдва кути по " + alfa.ToString() + " і два кути по " +
        (180 - alfa).ToString() + " градусів","Інформація",
        MessageBoxButtons.OK,MessageBoxIcon.Information);
   }
 }
```

В класах для задання початкових значень використовують спеціальні методи – **конструктори**. Вони не повертають жодного значення і мають назву, яка співпадає з назвою класу. Клас може мати декілька конструкторів з різною кількістю чи типами даних параметрів. Як правило, конструктори описують після полів класу, але перед його методами. Кожен клас може мати **конструктор без параметрів**, **конструктори з параметрами** та **конструктори копіювання** [2, с. 6-8]. Крім цього, у класі може бути **статичний конструктор** без параметрів, який виконується один раз перед першим використанням класу, та **закритий (private) конструктор** для недопущення створення об’єкта класу.

Приклад. Створити конструктор класу *Паралелограм*, в який будуть передаватися дві сторони і кут між ними та буде виводитися інформація про паралелограм.

```
public Parallelogram(double a, double b, double alfa)
 {this.a = s; this.b = b; this.alfa = alfa;
  Info();
 }
```

Після цього, якщо в програмі записати

```
Parallelogram p1 = new Parallelogram(5, 7, 60);   ,
```

то буде створений об'єкт-паралелограм.

Питання для самоконтролю

1. Синтаксис опису класів в C#
2. Поля та методи класів.
3. Області видимості в класах.
4. Підпрограми та їх використання в ООП.
5. Різновиди підпрограм.
6. Формальні та фактичні параметри підпрограм.
7. Локальні та глобальні змінні додатків.
8. Рекурсії та їх використання.

Тема 3. Спадкоємство класів як спадкоємство реалізацій

1. **Реалізація наслідування класів.**
2. **Організація ієрархії класів.**
3. **Використання в похідному класі конструктора базового класу з параметрами.**

1. На початку вивчення дисципліни нами бук розроблений клас *Паралелограм* з конструктором, доступними процедурами визначення площі, периметра та друку його даних. Створимо в цьому класі ще один **конструктор без параметрів**, який буде ініціалізувати поля об'єкта значеннями по замовчуванню:

```
class Parallelogram
 {private double a, b, alfa;

  public Parallelogram()
   {a = 5; b = 4; alfa = 60;
    Info();
   }

  public Parallelogram(double a, double b, double alfa)
   {this.a = a; this.b = b; this.alfa = alfa;
    Info();
   }

  public double area()
   {return a * b * Math.Sin(alfa / 180 * Math.PI);
   }

  public double perimeter()
   {return 2 * (a + b);
   }

  public void Info()
   {MessageBox.Show("Дані паралелограма:\nдві сторони по "+a.ToString()+
       " та дві по "+b.ToString()+" од.;\nплоща: " + area().ToString() +
       " кв. од.;\nпериметр: " + perimeter().ToString() +
       " од.;\nдва кути по " + alfa.ToString() + " і два кути по " +
       (180 - alfa).ToString() + " градусів","Інформація",
       MessageBoxButtons.OK,MessageBoxIcon.Information);
   }
 }
```

Після цього, якщо в програмі записати

```
Parallelogram p1 = new Parallelogram();
```

то буде створений об'єкт-паралелограм, ініціалізований значеннями по замовчуванню.

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

```
class Square : Parallelogram
 {public Square(double sa) : base(sa, sa, 90)
   { }

  public void Info()
   {MessageBox.Show("Дані квадрата:\nчотири сторони по "+
       Math.Sqrt(area()).ToString()+" од.;\nплоща " + area().ToString() +
       " кв. од.;\nпериметр " + perimeter().ToString() +
       " од.;\nчотири кути по 90 градусів");
   }
 }
```

При наслідуванні породжений клас автоматично отримує всі члени батька і може ще й бути розширений власними членами.

2. Спадкоємство є основною концепцією об'єктно-орієнтовного програмування (ООП). З його допомогою створюється ієрархія класів. Спадкоємство використовується

для розширення функціональних можливостей класу. При цьому, як було показано вище, похідний клас успадковує усі методи і властивості базового класу.

На відміну від C++, у C# заборонено *множинне спадкоємство*, тобто клас може успадковувати властивості і методи тільки від одного базового класу (предка).

Множинне спадкоємство можна реалізувати за допомогою *інтерфейсів*.

Таким чином, в C# є два типи спадкоємства: *спадкоємство реалізації* і *спадкоємство інтерфейсів*.

Розглянемо спадкоємство реалізації.

Синтаксис спадкоємства:

class ім'я_класу : им'я_батьківського класу

{тіло_класу}

Приклад 1

Розглянемо спадкоємство класів на прикладі. Створимо клас Person та похідний клас Student.

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

namespace Person
{
    class Person
    {
        public string Name;        //ім'я
        public int Age;            // вік
        public string Role;        // роль
        public string GetName() { return Name; }
    }
    class Student : Person
    {
        public string Facultet;
        public string Group;
        public int Course;
        public double Rating;

        public Student(string N, int A, string R, string F, string G, int C)
        {
          //конструктор з параметрами
            Name = N;
            Age = A;
            Role = R;
            Facultet = F;
            Group = G;
            Course = C;

        }
        public string GetRole(int Course)
        {
            if (Course <= 4)
                Role = "бакалавр";
            else
                Role = "магістр";
            return Role;
        }
        public void Student_Rating(double Rating)
        {
```

```
            if (Rating >= 82)
                Console.WriteLine("Привіт відмінникам");
            else
                if (Rating <= 45)
                    Console.WriteLine("Перездача! Треба краще вчитися!");
                else
                    Console.WriteLine("Можна  вчитися ще краще!");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            string r;
            string newRole;
            //дані рейтингу
            Student newSt = new Student("Іванов", 20, "студент", "КННІ", "К-
81", 4);
            Console.WriteLine("Ваш рейтинг?");
            r = Console.ReadLine();
            newSt.Rating = Convert.ToDouble(r);
            newSt.Student_Rating(newSt.Rating);
            newRole = newSt.GetRole(newSt.Course);
            Console.WriteLine("Прізвище = " + newSt.Name);
            Console.WriteLine("Вік= " + newSt.Age);
            Console.WriteLine("Роль= " + newSt.Role);
            Console.WriteLine("Факультет = " + newSt.Facultet);
            Console.WriteLine("група= " + newSt.Group);
            Console.WriteLine("курс= " + newSt.Course);
            Console.ReadLine();
        }
    }

      }
```

Увага!

При спадкоємстві конструктори не успадковуються, тому похідний клас повинен мати власні конструктори.

В цьому прикладі в класі Student успадковуються поля базового класу і визначається конструктор з параметрами. В базовому класі конструктора не має (за замовчанням створюється конструктор без параметрів).

Якщо в базовому класі конструктор **без параметрів**, то все буде добре.

Проблеми з успадкуванням конструкторів стосуються визначення і ініціалізації конструктора в похідному класі.

Порядок виклику конструкторів визначається наведеними нижче правилами.

- Якщо в конструкторі похідного класу явний виклик конструктора базового класу відсутній, автоматично викликається конструктор базового класу без параметрів.
- Для ієрархії, що складається з декількох рівнів, конструктори базових класів викликаються, починаючи з самого верхнього рівня. Після цього виконуються конструктори тих елементів класу, які є об'єктами, в порядку їх оголошення в класі, а потім виконується конструктор класу. Таким чином, кожен конструктор ініціалізує свою частину об'єкту.

- Якщо конструктор базового класу вимагає вказівки параметрів, він має бути явним чином викликаний в конструкторі похідного класу в списку ініціалізації. Виклик виконується за допомогою ключового слова **base**. Викликається та версія конструктора, список параметрів якої відповідає списку аргументів, вказаних після слова **base.**

3. Передача управління конструктору базового класу здійснюється за допомогою конструкції

...(...):base(...){...},

яка розташовується в оголошенні конструктора похідного класу між заголовком конструктора і тілом. Після ключового слова base в дужках розташовується список значень параметрів конструктора базового класу. Очевидно, що вибір відповідного конструктора визначається типом значень в списку.

Для створення об'єктів можна застосовувати конструктори трьох ступенів захисту:

public – при створенні об'єктів в рамках даного простору імен, в методах будь-якого класу — члена даного простору імен;

protected – при створенні об'єктів в рамках похідного класу, у тому числі при побудові об'єктів похідного класу, а також для внутрішнього використовування класом — власником даного конструктора;

private – застосовується виключно для внутрішнього використовування класом-власником даного конструктора. – не успадковується.

Приклад 2. Спадкоємство конструктора з параметрами.

Спадкоємство конструктора базового класу.

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

namespace Person
{
    class Person
    {
        public string Name; //ім'я
        public int Age;  // вік
        public string Role; // роль
        public Person(string N, string R, int A)
        {
            Name = N;
            Age = A;
            Role = R;
        }
        public string GetName() { return Name; }
        public string GetRole() { return Role; }
    }


    class Student : Person
    {
        public string Facultet;
        public string Group;
        public int Course;
        public double Rating;
```

```
        public Student(string N, int A, string R, string F, string G, int C):
base (N, R, A)

        {
            //конструктор з параметрами
            Name = N;
            Age = A;
            Role = R;
            Facultet = F;
            Group = G;
            Course = C;

        }
        public void Student_Rating(double Rating)
        {
            if (Rating >= 82)
                Console.WriteLine("Привіт відмінникам");
            else
                if (Rating <= 45)
                    Console.WriteLine("Перездача! Треба краще вчитися!");
                else
                    Console.WriteLine("Можна  вчитися ще краще!");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            //дані рейтингу
            Student newSt = new Student("Іванов", 20, "студент", "КННІ", "К-
61", 4);
            Console.WriteLine("Ваш рейтинг?");
            string r = Console.ReadLine();
            newSt.Rating = Convert.ToDouble(r);
            newSt.Student_Rating(newSt.Rating);
            Console.WriteLine("Прізвище = " + newSt.Name);
            Console.WriteLine("Вік= " + newSt.Age);
            Console.WriteLine("Роль= " + newSt.Role);
            Console.WriteLine("Факультет = " + newSt.Facultet);
            Console.WriteLine("група= " + newSt.Group);
            Console.WriteLine("курс= " + newSt.Course);
            Console.ReadLine();
        }
    }
}
```

Елементи базового класу, визначені як private, в похідному класі недоступні. Тому для доступу до полів класу Person вони повинні визначатися як public чи protected.

Питання для самоконтролю

1. Для чого використовується спадкоємство?
2. Який синтаксис опису похідного класу?
3. Які специфікатори доступу застосовуються в ієрархіях?
4. Як викликати метод базового класу з похідного?
5. Опишіть порядок виклику конструкторів базових класів при роботі конструктора похідного класу.

Тема 4. Пізнє і раннє зв'язування методів класів. Абстрактні класи

1. **Віртуальні методи. Пізнє і раннє зв'язування об'єктів класу**
2. **Абстрактні класи і методи**
3. **Приховані класи.**

1. Віртуальні методи. Пізнє і раннє зв'язування об'єктів класу

При **ранньому зв'язуванні** (на етапі проектування програми) програма є структурою, логіка виконання якої жорстко визначена. Якщо ж потрібно щоб рішення про те, який з однойменних методів різних об'єктів ієрархії використовувати, приймалося залежно від конкретного об'єкту, для якого виконується виклик, то заздалегідь жорстко пов'язувати ці методи з останньою частиною коду не можна.

Отже, треба якимсь чином дати знати компілятору, що ці методи оброблятимуться по-іншому. Для цього в C# існує ключове слово *virtual*. Воно записується в заголовку методу *базового класу*, наприклад:

virtual public void Passport() ...

Оголошення методу віртуальним означає, що всі посилання на цей метод будуть визначатися у момент його виклику під час виконання програми. Цей механізм називається **пізнім зв'язуванням**.

При визначенні віртуального методу у складі базового класу перед типом значення, що повертається, указується ключове слово **virtual** а при перевизначенні віртуального методу в похідному класі використовується модифікатор **override**. Віртуальний метод не може бути визначений з модифікатором static або abstract.

Перевизначений віртуальний метод повинен мати такий самий набір параметрів, як і однойменний метод базового класу.

Приклад 1

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

namespace Person
{
    class Person
    {
        public string Name; //ім'я
        public int Age;     // вік
        public string Role;       // роль
        public string GetName() { return Name; }
        public virtual string GetRole(int Age) { return Role; }
    }
    class Student : Person
    {
        public string Facultet;
        public string Group;
        public int Course;
        public double Rating;
        public Student(string N, int A, string R, string F, string G, int C)
```

```
        {
            //конструктор з параметрами
            Name = N;
            Age = A;
            Role = R;
            Facultet = F;
            Group = G;
            Course = C;
        }
        public override string GetRole(int Course)
        {
            if (Course <= 4)
                Role = "бакалавр";
            else
                Role = "магістр";
            return Role;
        }
        public void Student_Rating(double Rating)
        {
            if (Rating >= 82)
                Console.WriteLine("Привіт відмінникам");
            else
                if (Rating <= 45)
                    Console.WriteLine("Перездача! Треба краще вчитися!");
                else
                    Console.WriteLine("Можна  вчитися ще краще!");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            string r;
            string newRole;
            //дані рейтингу
            Student newSt = new Student("Іванов", 20, "студент", "КННІ", "К-
61", 4);
            Console.WriteLine("Ваш рейтинг?");
            r = Console.ReadLine();
            newSt.Rating = Convert.ToDouble(r);
            newSt.Student_Rating(newSt.Rating);
            newRole = newSt.GetRole(newSt.Course);
            Console.WriteLine("Прізвище = " + newSt.Name);
            Console.WriteLine("Вік= " + newSt.Age);
            Console.WriteLine("Роль= " + newSt.Role);
            Console.WriteLine("Факультет = " + newSt.Facultet);
            Console.WriteLine("група= " + newSt.Group);
            Console.WriteLine("курс= " + newSt.Course);
            Console.ReadLine();
        }
    }
}
```

Примітка

Використання віртуальних методів ускладнює тестування класу.

Віртуальні методи базового класу визначають інтерфейс всієї ієрархії. Цей інтерфейс може розширюватися в нащадках за рахунок додавання нових віртуальних методів.

Перевизначати віртуальний метод в кожному з нащадків не обов'язково: якщо він виконує дії, що влаштовують нащадка, метод успадковується.

За допомогою віртуальних методів реалізується один з основних принципів об'єктно-орієнтованого програмування — поліморфізм. Це слово в перекладі з грецького означає "багато форм", що в даному випадку означає "один виклик — багато методів".

Віртуальні методи незамінні і при передачі об'єктів в методи як параметри. У параметрах методу описується об'єкт базового типа, а при виклику в нього передається об'єкт похідного класу. Віртуальні методи, що викликаються для об'єкту з методу, відповідатимуть типу аргументу, а не параметра.

2. Абстрактні класи і методи

При створенні ієрархії об'єктів для виключення коду, що повторюється, часто буває логічно виділити їх загальні властивості в один базовий клас. При цьому може виявитися, що створювати екземпляри такого класу не має сенсу, тому що жодні реальні об'єкти їм не відповідають.

Такі класи називають абстрактними.

В абстрактному класі визначаються лише загальні призначення методів, які повинні бути реалізовані в похідних класах, але сам по собі цей клас не містить реалізації методів, а тільки їх сигнатуру (тип значення, що повертається, ім'я методу і список параметрів).

При оголошенні абстрактного методу використовується модифікатор *abstract*. Абстрактний метод автоматично стає віртуальним, так що модифікатор virtual при оголошенні методу не використовується.

Абстрактний клас призначений тільки для створення ієрархії класів, не можна створити *об'єкт абстрактного класу.*

Якщо в класі є хоч би один абстрактний метод, весь клас також має бути описаний як абстрактний, наприклад:

Приклад 2

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

namespace Person
{
    abstract class Person
    {
        public string Name; //имя
        public int Age;  // возраст
        public string Role;        // роль
        public string GetName() { return Name; }
        abstract public string GetRole(int Course);
    }
    class Student : Person
    {
        public string Facultet;
        public string Group;
        public int Course;
        public double Rating;
```

```
        public Student(string N, int A, string R, string F, string G, int C)
        {
            //конструктор з параметрами
            Name = N;
            Age = A;
            Role = R;
            Facultet = F;
            Group = G;
            Course = C;
        }
        public override string GetRole(int Course)
        {
            if (Course <= 4)
                Role = "бакалавр";
            else
                Role = "магістр";
            return Role;
        }
        public void Student_Rating(double Rating)
        {
            if (Rating >= 82)
                Console.WriteLine("Привіт відмінникам");
            else
                if (Rating <= 45)
                    Console.WriteLine("Перездача! Треба краще вчитися!");
                else
                    Console.WriteLine("Можна  вчитися ще краще!");
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            string r;
            string newRole;
            //дані рейтингу
            Student newSt = new Student("Іванов", 20, "студент", "КННІ", "К-
81", 4);
            Console.WriteLine("Ваш рейтинг?");
            r = Console.ReadLine();
            newSt.Rating = Convert.ToDouble(r);
            newSt.Student_Rating(newSt.Rating);
            newRole = newSt.GetRole(newSt.Course);
            Console.WriteLine("Прізвище = " + newSt.Name);
            Console.WriteLine("Вік= " + newSt.Age);
            Console.WriteLine("Роль= " + newSt.Role);
            Console.WriteLine("Факультет = " + newSt.Facultet);
            Console.WriteLine("група= " + newSt.Group);
            Console.WriteLine("курс= " + newSt.Course);
            Console.ReadLine();
        }
    }
}
```

3. Приховані (закриті) класи

В C# можна визначати класи і методи як sealed (закриті). У випадку класу це означає, що ви не можете успадковувати від нього. У випадку методу це означає, що його не можна перевизначити.

```
sealed class FinalClass
{
```

```
  //тіло класу
}
class DerivedClass : FinalClass  //неправильно. Помилка компіляції
{
  //тіло класу
}
```

Більшість вбудованих типів даних описана як sealed. Якщо необхідно використовувати функціональність такого класу, застосовується не спадкоємство, а включення: у класі описується поле відповідного типу.

Включення класів, коли один клас включає поля, що є класами, є альтернативою спадкоємству при проектуванні. Наприклад, якщо є об'єкт "двигун", а потрібно описати об'єкт "літак", логічно зробити двигун полем цього об'єкту, а не його предком.

4. Види відношень між класами

Механізм спадкоємства класів надає великі можливості організації коду і його багатократного використання. Вибір найбільш відповідних засобів для цілей конкретного проекту ґрунтується на знанні механізму їх роботи і взаємодії.

Коли потрібно використовувати спадкоємство? Це залежить від різних причин, обумовлених вирішуваною задачею, наприклад:

- Спеціалізація. Клас-нащадок є спеціалізованою формою базового класу — в нащадку просто перевизначаються методи.
- Специфікація. Клас-нащадок реалізує поведінку, описану в батьківському класі. У C# ця форма реалізується спадкоємством від абстрактного класу.
- Конструювання. Клас-нащадок використовує методи базового класу, але не є його підтипом.
- Розширення. У клас-нащадок додають нові методи, розширюючи поведінку батьківського класу.
- Узагальнення. Клас-нащадок узагальнює поведінку базового класу. Звичайно таке спадкоємство використовується в тих випадках, коли змінити поведінку базового класу неможливо (наприклад, базовий клас є бібліотечним класом).
- Обмеження. Клас-нащадок обмежує поведінку батьківського класу.
- Комбінування. Клас-нащадок успадковує риси декількох класів — це множинне спадкоємство (у C# не використовується, оскільки множинне спадкоємство заборонене, а спадкоємство від декількох інтерфейсів має інший сенс).

Альтернативою спадкоємству при проектуванні ієрархії класів є **вкладення**, коли один клас включає поля, які самі є класами. Наприклад, якщо є клас "**двигун**", а потрібно описати клас "літак", логічно зробити двигун полем цього класу, а не його предком. Вкладення представляє відношення класів "Y містить X" або "Y реалізується за допомогою X" і зазвичай реалізується за допомогою моделі "включення-делегування". Приклад 9.5. ілюструє цю можливість.

Приклад 3

```
using System;
namespace ConsoleApplication1
{
```

```
class Двигун
{   public void Запуск()
    {
       Console.WriteLine( "вжжжж!!" );
    }
}
class Літак
{  public Літак ()
   {
       лівий  = new Двигун();
       правий = new Двигун();
    }
    public void Запустити_двигуни()
    {
       лівий.Запуск();
       правий.Запуск();
    }
    Двигун лівий, правий;
}

class Class1
{   static void Main()
    {
    Літак АН24_1 = new Літак();
    АН24_1. Запустити_двигуни ();
    }
}
}
```

5. Клас object – базовий клас ієрархії класів C#

Кореневий клас System.Object всієї ієрархії об'єктів .NET, який має в C# назву **object**, забезпечує всіх нащадків кількома важливими методами. Похідні класи можуть використовувати ці методи безпосередньо або перевизначати їх.

Клас object часто використовується і безпосередньо при описі типу параметрів методів для додавання їм універсальності, а також для зберігання посилань на об'єкти різних типів — таким чином реалізується **поліморфізм**.

Відкриті методи класу System.Object перелічені нижче.

Метод **Equals** з одним параметром повертає значення true, якщо параметр і об'єкт, який його викликає посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public virtual bool Equals(object obj);
```

Метод **Equals** з двома параметрами повертає значення true, якщо обидва параметри посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public static bool Equals(object ob1, object ob2);
```

Метод **GetHashCod**e формує хеш-код об'єкту і повертає число, що однозначно ідентифікує об'єкт. Це число використовується в різних структурах і алгоритмах бібліотеки. Якщо перевизначається метод Equals, необхідно перенавантажувати і метод GetHashCode. Синтаксис:

```
public virtual int GetHashCode();
```

Метод **GetType** повертає поточний поліморфний тип об'єкту, тобто не тип посилання, а тип об'єкту, на який посилання вказує в даний момент. Значення, що повертається має тип Type. Це абстрактний базовий клас ієрархії, що використовується для отримання інформації про типи під час виконання. Синтаксис:

```
public Type GetType();
```

Метод **ReferenceEquals** повертає значення true, якщо обидва параметри посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public static bool ReferenceEquals(object ob1, object ob2);
```

Метод **ToString** за замовчанням повертає для посилкових типів повне ім'я класу у вигляді рядка, а для значимих — значення величини, перетворене в рядок. Цей метод перевизначають для того, щоб можна було виводити інформацію про стан об'єкту. Синтаксис:

```
public virtual string ToString()
```

У похідних об'єктах ці методи часто перевизначають. Наприклад, можна перевизначити метод Equals для того, щоб задати власні критерії порівняння об'єктів.

Висновки

Спадкоємство – важлива риса об'єктно-орієнтованого програмування. Воно призначене для відображення такої риси програмних систем як ієрархічність. Створення ієрархії класів дозволяє розширювати функціональні можливості класів в похідних класах, а також повторно використовувати код методів базових класів.

На відміну від C++, у C# заборонено множинне спадкоємство, тобто похідний клас (нащадок) може мати тільки один базовий клас (предок).

При спадкоємстві конструктори класів не успадковуються, тому похідний клас повинен мати власні конструктори.

Якщо конструктор базового класу має параметри, він має бути явним чином викликаний в конструкторі похідного класу за допомогою ключового слова **base.**

Віртуальні методи є ще однією рисою поліморфізму, реалізованому в C#. На відміну від перегружених методів, віртуальні методи повинні мати однакові списки параметрів. Віртуальні методи надають можливості пізнього зв'язування об'єктів (під час виконання).

Альтернативою спадкоємству класів при проектуванні ієрархії класів є **вкладення**, коли один клас включає поля, які самі є класами.

Таким чином в C# є два типи зв'язків між класами: спадкоємство і вкладення.

Питання для самоконтролю

1. Які ключові слова використовуються при перевизначенні методів базового класу в похідному?
2. Опишіть механізми раннього і пізнього зв'язування.
3. Опишіть процес виклику віртуального методу. Для чого застосовуються віртуальні методи?
4. Чи всі методи слід описувати як віртуальні?
5. Для чого використовуються абстрактні класи?

Тема 5. Інкапсуляція полів і методів їх коригування за допомогою властивостей

1. **Властивості.**
2. **Індексатори.**

1. Властивість – це ідентифікатор визначеного типу, при звертанні до якого викликаються пов'язані **методи доступу**. Фактично властивість – це природнє розширення поля класу. Синтаксис використанні властивості нічим не відрізняється від поля, але при звертанні до властивості **не опрацьовуються дані, а викликаються пов'язані методи**.

Синтаксис опису властивостей в C# [2, с. 24-27] … :

Загальний синтаксис опису властивості:

```
[специфікатор доступу] <ідентифікатор>
{ get { // тіло функції зчитування значення return ; }
  set { // тіло функції встановлення значення } }
```

Як правило, властивість оголошують відкритою (зі специфікатором доступу public). Якщо властивість використовується для доступу до деякого закритого поля, то тип властивості повинен співпадати із типом цього поля. Тип властивості не може бути void. При описі властивості може бути відсутньою або частина get або частина set, але не обидві одночасно. Якщо у описі властивості присутня тільки частина get, то така властивість називається властивістю **тільки для читання**. Якщо у описі властивості присутня тільки частина set, то така властивість називається властивістю **тільки для запису**.

Метод get повинен містити оператор return. В методі set для доступу до значення, яке встановлюється, використовується неявний параметр value.

```
class Parallelogram
 {private double a, b, alfa;
  public double A
     {get { return a; }
      set { if (value >= 0)
            a = value; }}
  public double B
     {get { return b; }
      set { if (value >= 0)
             b = value; }}
…}

class Parallelepiped : Parallelogram
 {private double h;
  public double H
     {get { return h; }
      set { if (value >= 0)
             h = value; }}
  public Parallelepiped(double a,double b,double h) : base(a, b, 90)
     { this.h = h; }
  new public double area()
     { return 2*(A * B + A * H + B * H); }
     public double volume()
     { return A * B * H; }
     public override void Info()
     {MessageBox.Show("Дані паралелепіпеда:\nсторони по " + A.ToString() +
  ", "+B.ToString()+", "+H.ToString()+" од.;\nплоща: " + area().ToString() +
  " кв. од.\nоб'єм: " + volume().ToString() + " куб. од.", "Інформація",
  MessageBoxButtons.OK, MessageBoxIcon.Information); }}
```

Характерною рисою властивостей є те, що із властивостями можна працювати як із звичайними полями об'єкта. В той же час, при наданні властивості деякого значення

насправді викликається функція встановлення значення (блок set), а при звертанні до значення властивості викликається функція зчитування значення (блок get).

2. Індексатори є різновидом властивостей і дозволяють звертатися до полів класу за деяким індексом. Загальний синтаксис опису індексатора:

```
[специфікатор доступу] this[<список індексів>]
 { get { //тіло функції одержання значень за індексами return ; }
   set { //тіло функції встановлення значень за індексами } }
```

Як і у випадку властивостей, як правило, індексатор оголошують відкритим (зі специфікатором доступу public). Тип індексатора не може бути void. При описі індексатора може бути відсутньою або частина get або частина set, але не обидві одночасно. Метод get повинен містити оператор return. В методі set для доступу до значення, яке встановлюється, використовується неявний параметр value. У списку індексів вказують через кому опис індексів (тип індексу та його ім'я). Тип індексу та кількість індексів у списку може бути довільною, але найчастіше використовують один індекс цілого типу. Індексатори в основному використовуються у класах для доступу до закритого поля-масиву (одновимірний або багатовимірний), хоча можуть надавати доступ і до інших членів.

Задача. Довжини сторін початкового прямокутного паралелепіпеда задані користувачем, а кожного наступного – вдвічі менші від попереднього. Визначити номер першого паралелепіпеда, об'єм якого менший від заданого числа.

```
class EmbeddedParallelepiped : Parallelepiped
 {public EmbeddedParallelepiped(double a, double b, double h) : base(a, b, h)
   { }

  public Parallelepiped this[int i]
     {get {if (i <= 0) return new Parallelepiped(A, B, H);
           else return new Parallelepiped(A / Math.Pow(2, i), B / Math.Pow(2, i),
                                          H / Math.Pow(2, i)); }
     }}
```

Тоді використання об'єкту цього класу при натисненні відповідної кнопки буде таким:

```
private void button4_Click(object sender, EventArgs e)
  {var vp=new EmbeddedParallelepiped(5,7,6);
   int i=1;
   double VMax = 17;
   if (!inputDouble(ref VMax, "Введіть обмеження об'єму")) return;
   while (vp[i].volume() > VMax) i++;
   MessageBox.Show("Вкладений паралелепіпед № " + i.ToString() + " має об'єм " +
    vp[i].volume()    +    "    куб.    од.",    "Інформація",    MessageBoxButtons.OK,
    MessageBoxIcon.Information); }
```

Приклад. Визначення вільних пар протягом тижня.

```
private void button1_Click(object sender, EventArgs e)
 {   CRoozkladWeek rozkladWeek = new CRoozkladWeek(4);
     string[] nameDay = new string[] { "понеділ.", "вівторок", "середа", "четвер",
      "п'ятниця", "субота", "неділя" };
     rozkladWeek[0, 1] = "Програмування";
     rozkladWeek[0, 2] = "Дискретна математика";
     string rez="";
     for (int day=0; day<=6; day++)
      {bool nayavno=false;
       for (int para=0; para<=5; para++)
           if (rozkladWeek[day,para]==null)
           {if (!nayavno)
             {rez+=nameDay[day]+":\t"+(para+1).ToString(); nayavno=true;}
            else rez+=", "+(para+1).ToString(); }
       if (nayavno) rez+="\n"; }
     if (rez == "") rez = "Вільні пари відсутні";
     else rez = "Вільні пари:\n" + rez;
     MessageBox.Show(rez); }}
```

```
class CRoozkladWeek
{private string[,] rozklad = new string[7,6];
 int nomerWeek;
 public CRoozkladWeek(int nomerWeek)
  {this.nomerWeek = nomerWeek; }
 public string this[int weekDay, int para]
  {get {if (weekDay >= 0 && weekDay <= 6 &&
            para >= 0 && para <= 5)
            return rozklad[weekDay, para];
        else return null; }
   set {if (weekDay >= 0 && weekDay <= 6 &&
            para >= 0 && para <= 5)
         rozklad[weekDay, para]=value;}}}
```

Питання для самоконтролю

1. Чим поля класу відрізняються від його методів?
2. Що таке властивості об'єкта? Чим властивості об'єкта відрізняються від полів? Чому для властивості зазначають тип даних?
3. Чому доступ до даних об'єкта рекомендується виконувати через його властивості?
4. Як заборонити зчитування/запис властивості об'єкта?
5. Що таке індексатори? Що спільного між властивостями та індексаторами?

Тема 6. Ролі класів та підкласів

1. **Види відношень між класами.**
2. **Клас object – базовий клас ієрархії класів C#.**

1. Види відношень між класами

Механізм спадкоємства класів надає великі можливості організації коду і його багатократного використання. Вибір найбільш відповідних засобів для цілей конкретного проекту ґрунтується на знанні механізму їх роботи і взаємодії.

Коли потрібно використовувати спадкоємство? Це залежить від різних причин, обумовлених вирішуваною задачею, наприклад:

- Спеціалізація. Клас-нащадок є спеціалізованою формою базового класу — в нащадку просто перевизначаються методи.
- Специфікація. Клас-нащадок реалізує поведінку, описану в батьківському класі. У C# ця форма реалізується спадкоємством від абстрактного класу.
- Конструювання. Клас-нащадок використовує методи базового класу, але не є його підтипом.
- Розширення. У клас-нащадок додають нові методи, розширюючи поведінку батьківського класу.
- Узагальнення. Клас-нащадок узагальнює поведінку базового класу. Звичайно таке спадкоємство використовується в тих випадках, коли змінити поведінку базового класу неможливо (наприклад, базовий клас є бібліотечним класом).
- Обмеження. Клас-нащадок обмежує поведінку батьківського класу.
- Комбінування. Клас-нащадок успадковує риси декількох класів — це множинне спадкоємство (у C# не використовується, оскільки множинне спадкоємство заборонене, а спадкоємство від декількох інтерфейсів має інший сенс).

Альтернативою спадкоємству при проектуванні ієрархії класів є **вкладення**, коли один клас включає поля, які самі є класами. Наприклад, якщо є клас "**двигун**", а потрібно описати клас "літак", логічно зробити двигун полем цього класу, а не його предком. Вкладення представляє відношення класів "Y містить X" або "Y реалізується за допомогою X" і зазвичай реалізується за допомогою моделі "включення-делегування". Приклад 9.5. ілюструє цю можливість.

Приклад 1

```
using System;
namespace ConsoleApplication1
{
   class Двигун
   {   public void Запуск()
       {
          Console.WriteLine( "вжжжж!!" );
       }
   }
   class Літак
   {  public Літак ()
      {
          лівий  = new Двигун();
```

```
            правий = new Двигун();
        }
        public void Запустити_двигуни()
        {
            лівий.Запуск();
            правий.Запуск();
        }
        Двигун лівий, правий;
    }

    class Class1
    {   static void Main()
        {
        Літак АН24_1 = new Літак();
        АН24_1. Запустити_двигуни ();
        }
    }
}
```

2. Клас object – базовий клас ієрархії класів C#

Кореневий клас System.Object всієї ієрархії об'єктів .NET, який має в C# назву **object**, забезпечує всіх нащадків кількома важливими методами. Похідні класи можуть використовувати ці методи безпосередньо або перевизначати їх.

Клас object часто використовується і безпосередньо при описі типу параметрів методів для додавання їм універсальності, а також для зберігання посилань на об'єкти різних типів — таким чином реалізується **поліморфізм**.

Відкриті методи класу System.Object перелічені нижче.

Метод **Equals** з одним параметром повертає значення true, якщо параметр і об'єкт, який його викликає посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public virtual bool Equals(object obj);
```

Метод **Equals** з двома параметрами повертає значення true, якщо обидва параметри посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public static bool Equals(object ob1, object ob2);
```

Метод **GetHashCod**e формує хеш-код об'єкту і повертає число, що однозначно ідентифікує об'єкт. Це число використовується в різних структурах і алгоритмах бібліотеки. Якщо перевизначається метод Equals, необхідно перенавантажувати і метод GetHashCode. Синтаксис:

```
public virtual int GetHashCode();
```

Метод **GetType** повертає поточний поліморфний тип об'єкту, тобто не тип посилання, а тип об'єкту, на який посилання вказує в даний момент. Значення, що повертається має тип Type. Це абстрактний базовий клас ієрархії, що використовується для отримання інформації про типи під час виконання. Синтаксис:

```
public Type GetType();
```

Метод **ReferenceEquals** повертає значення true, якщо обидва параметри посилаються на одну і ту саму ділянку пам'яті. Синтаксис:

```
public static bool ReferenceEquals(object ob1, object ob2);
```

Метод **ToString** за замовчанням повертає для посилкових типів повне ім'я класу у вигляді рядка, а для значимих — значення величини, перетворене в рядок. Цей метод перевизначають для того, щоб можна було виводити інформацію про стан об'єкту. Синтаксис:

```
public virtual string ToString()
```

У похідних об'єктах ці методи часто перевизначають. Наприклад, можна перевизначити метод Equals для того, щоб задати власні критерії порівняння об'єктів.

Висновки

Спадкоємство – важлива риса об'єктно-орієнтованого програмування. Воно призначене для відображення такої риси програмних систем як ієрархічність. Створення ієрархії класів дозволяє розширювати функціональні можливості класів в похідних класах, а також повторно використовувати код методів базових класів.

На відміну від C++, у C# заборонено множинне спадкоємство, тобто похідний клас (нащадок) може мати тільки один базовий клас (предок).

При спадкоємстві к*онструктори класів не успадковуються, тому похідний клас повинен мати власні конструктори.*

Якщо конструктор базового класу має параметри, він має бути явним чином викликаний в конструкторі похідного класу за допомогою ключового слова **base.**

Віртуальні методи є ще однією рисою поліморфізму, реалізованому в C#. На відміну від перегружених методів, віртуальні методи повинні мати однакові списки параметрів. Віртуальні методи надають можливості пізнього зв'язування об'єктів (під час виконання).

Альтернативою спадкоємству класів при проектуванні ієрархії класів є **вкладення**, коли один клас включає поля, які самі є класами.

Таким чином в C# є два типи зв'язків між класами: спадкоємство і вкладення.

Питання для самоконтролю

1. Які види відношень використовуються між класами?
2. Які є альтернативи наслідуванню класів?
3. Що означає вкладення класів?
4. Який базовий клас ієрархії класів C#? Які він має методи?

Тема 7. Поліморфізм

1. **Поліморфізм. Перевантаження методів**
2. **Методи зі змінною кількістю аргументів**
3. **Перевантаження операцій**
4. **Приклад класу з перевантаженими методами і операціями**

1. Перевантаження методів

Поліморфізм – це концепція, що дозволяє мати різні реалізації для одного і того ж *методу*, які будуть вибиратися залежно від *типу об'єкту*, переданого до методу при його виклику.

Ця концепція в C# реалізується як перевантаження (методів, операцій).

Часто буває зручно, аби методи, що реалізують один і той самий алгоритм для різних типів даних, мали одне і те саме ім'я. Використання декількох методів з одним і тим самим іменем, але різними типами параметрів називається *перевантаженням методів.*

Компілятор визначає, який саме метод потрібно викликати за типом фактичних параметрів. Наприклад, нижче наведено декілька реалізацій метода max, який повертає найбільше значення для різних типів і кількості параметрів.

```
// Повертає найбільше з двох цілих:
int max( int a, int b )
// Повертає найбільше з трьох цілих:
int max( int a, int b, int з )
// Повертає найбільше першого параметра і довжини другого:
int max ( int a, string b )
// Повертає найбільше другого параметра і довжини першого:
int max ( string b, int a )
...
Console.WriteLine( max( 1, 2 ));
Console.WriteLine( max( 1, 2, 3 ));
Console.WriteLine( max( 1, "2" ));
Console.WriteLine( max( "1", 2 ));
```

При виклику методу max компілятор вибирає варіант методу, який відповідає типу аргументів методу.

Якщо точної відповідності не знайдено, виконуються неявні перетворення типів відповідно до загальних правил. Якщо перетворення неможливе, видається повідомлення про помилку. Якщо відповідність на одному і тому ж етапі може бути отримана більш ніж одним способом, вибирається варіант, що містить меншу кількість і довжину перетворень. Якщо існує декілька варіантів, з яких неможливо вибрати кращий, видається повідомлення про помилку.

Перевантажені методи мають одне **ім'я**, але повинні розрізнятися параметрами, точніше їх типами і кількістю.

Перевантаження широко використовується в класах бібліотеки .NET. Наприклад, в стандартному класі Console метод WriteLine перевантажений 19 разів для виведення величин різних типів.

Перевантажуватися можуть як конструктори, так і звичайні методи класу. Наприклад, клас Random має перевантажені конструктори:

Перший конструктор викликається без параметрів.

public Random();

Інший конструктор з параметром –

public Random (int);

Перевантажений метод public int Next(); при кожному виклику повертає позитивне ціле число, рівномірно розподілене в деякому діапазоні. Діапазон задається параметрами методу. Три реалізації методу відрізняються набором параметрів:

public int Next () - метод без параметрів видає цілі позитивні числа у всьому позитивному діапазоні типу int;

public int Next (int max) - видає цілі позитивні числа в діапазоні [0,max];

public int Next (int min, int max) - видає цілі позитивні числа в діапазоні [min,max].

Метод public double NextDouble () має одну реалізацію. При кожному виклику цього методу видається нове випадкове число, рівномірно розподілене в інтервалі [0,1).

2. Методи зі змінною кількістю аргументів

Інколи буває зручно створити метод, в який можна передавати різну кількість аргументів. Мова C# надає таку можливість за допомогою ключового слова *params*. Параметр, помічений цим ключовим словом, розміщується в списку параметрів останнім і позначає масив заданого типу невизначеної довжини, наприклад:

public int Calculate(int a, out int c, params int[] d).

У цей метод можна передати три і більше параметрів. Усередині методу до параметрів, починаючи з третього, звертаються як до звичайних елементів масиву. Кількість елементів масиву отримують за допомогою його властивості Length. Як приклад розглянемо метод обчислення середнього значення елементів масиву.

Приклад 1

```
using System;
namespace ConsoleApplication1
{    class Class1
   {
      public static double Average( params int[] a )
      {
         if ( a.Length == 0 )
            throw new Exception("Недостатньо аргументів в методі" );

         double av = 0;
         foreach ( int elem in a ) av += elem;
         return av / a.Length;
      }

      static void Main()
      {
```

```
        try
        {
           int[] a = { 10, 20, 30 };
           Console.WriteLine( Average( a ) );           // 1
           int[] b = { -11, -4, 12, 14, 32, -1, 28 };
           Console.WriteLine( Average( b ) );           // 2
           short z = 1, e = 12;
           byte v = 100;
           Console.WriteLine( Average( z, e, v ) );     // 3
           Console.WriteLine( Average() );              // 4
        }
        catch( Exception e )
        {
           Console.WriteLine( e.Message );
           return;
        }
     }
  }
}
```

Результат роботи програми:

10

20

40

Недостатньо аргументів в методі

Параметр-масив може бути лише один і повинен розташовуватися останнім в списку. Відповідні йому аргументи повинні мати типи, для яких можливе неявне перетворення до типа масиву.

3. Перевантаження операцій

Клас — це тип даних, тому програміст може визначити для нього власні операції. Ви вже стикалися з перевантаженням (тобто перевизначенням) операцій: наприклад, знак "+" для арифметичних типів означає додавання, а для рядків — склеювання. Операції перевантажують в основному для класів, за допомогою яких задають які-небудь математичні поняття, тобто тоді, коли знаки операцій мають загальноприйняту семантику.

3.1.Унарні операції

Можна визначати в класі наступні унарні операції:

+ - ! ~ ++ -- true false

Синтаксис визначення унарної операції:

тип operator унарна_операція (параметр)

Приклади заголовків унарних операцій:

public static int operator +(MyObject m)

public static MyObject operator --(MyObject m)
public static bool operator true(MyObject m)

Операції не повинні змінювати значення операнду, який передається ним. Операція, що повертає величину типа класу, для якого вона визначається, повинна створити новий об'єкт цього класу, виконати з ним необхідні дії і передати його як результат.

3.2. Бінарні операції

Можна визначати в класі наступні бінарні операції:
+ - * / % & | ^ << >> == != > < >= <=
Синтаксис визначення бінарної операції:
тип operator бінарна_операція (параметр1, параметр2)

Приклади заголовків бінарних операцій:
public static MyObject operator + (MyObject m1, MyObject m2)
public static bool operator == (MyObject m1, MyObject m2)

Хоч б один параметр, який передається в операцію, повинен мати тип класу, для якого вона визначається. Операція може повертати величину будь-якого типу.

Операції == і !=, > і <, >= і <= визначаються лише парами і зазвичай повертають булеве значення. Найчастіше в класі визначають операції порівняння на рівність і нерівність для того, щоб забезпечити порівняння об'єктів, а не їх посилань, як визначено за замовчанням для посилкових типів.

3.3. Операції перетворення типу

Операції перетворення типу забезпечують можливість явного і неявного перетворення між типами даних користувача. Синтаксис визначення операції перетворення типу:

implicit operator тип (параметр) // неявне перетворення
explicit operator тип (параметр) // явне перетворення

Ці операції виконують перетворення з типу параметра до типу, вказаного в заголовку операції. Одним з цих типів має бути клас, для якого визначається операція. Таким чином, операції виконують перетворення або типу класу до іншого типу, або навпаки. Перетворювані типи не повинні бути зв'язані відношеннями спадкоємства.

- при використанні об'єкту у виразі, що містить змінні цільового типу;
- при передачі об'єкту в метод на місце параметра цільового типа;
- при явному приведенні типу.

Явне перетворення виконується при використанні операції приведення типу.

Всі операції класу повинні мати різні сигнатури. На відміну від інших видів методів, для операцій перетворення тип значення, яке повертається включається в сигнатуру, інакше не можна було б визначати варіанти перетворення цього типу в інші. Ключові слова implicit і explicit в сигнатуру не включаються, отже, для одного і того ж перетворення не можна визначити одночасно явну і неявну версію.

Неявне перетворення слід визначати так, щоб при його виконанні не виникала втрата точності і не генерувалися виключення. Якщо ці ситуації можливі, перетворення слід описати як явне.

4. Приклад класу з перевантаженими методами і операціями

Розглянемо рішення наступної задачі.

У текстовому файлі зберігається база відділу кадрів підприємства. На підприємстві 100 співробітників. Кожен рядок файлу містить запис про одного співробітника. Формат запису: прізвище і ініціали (30 поз., прізвище повинне починатися з першої позиції), рік народження (5 поз.), оклад (10 поз.). Написати програму, яка по заданому прізвищу виводить на екран відомості про співробітника, підраховуючи середній оклад всіх відібраних співробітників.

Для вирішення цієї задачі створимо клас **Person** і організуємо з екземплярів цього класу масив. При описі класу корисно задатися наступним питанням: які обов'язки мають бути покладені на цей клас? Вочевидь, що перший обов'язок — зберігання відомостей про співробітника. Аби скористатися цими відомостями, клієнт (тобто код, що використовує клас) повинен мати можливість отримати ці відомості, змінити їх і вивести на екран. Окрім цього, для пошуку співробітника бажано мати можливість порівнювати його ім'я із заданим.

Поля класу зробимо відповідно до принципу інкапсуляції закритими, а доступ до них організуємо за допомогою властивостей. Це дозволить контролювати процес занесення даних в поля. Для ідентифікації спроби введення невірних даних скористаємося механізмом виключень. Щоб зробити клас більш універсальним, введемо в нього можливість збільшувати і зменшувати оклад співробітника за допомогою перевантажених операцій класу.

Вхідні дані, результати і проміжні величини.

Вхідні дані. База співробітників знаходиться в текстовому файлі. Перш за все треба вирішити, чи зберігати в оперативній пам'яті одночасно всю інформацію з файлу чи можна обійтися

буфером на один рядок. Якби відомості про співробітника запитувалися одноразово, можна було б зупинитися на другому варіанті, але оскільки пошук по базі виконуватиметься більше одного разу, всю інформацію бажано зберігати в оперативній пам'яті, оскільки багатократне читання з файлу нераціонально.

Максимальна кількість рядків файлу за умовою завдання обмежена, тому можна виділити для їх зберігання масив з 100 елементів. Кожен елемент масиву міститиме відомості про одного співробітника, організовані у вигляді класу.

У програму потрібно також вводити прізвища потрібних співробітників. Чергове прізвище, що вводиться, зберігатимемо в стандартному рядку типу string.

Результати. В результаті роботи програми потрібно вивести на екран необхідні елементи результуючого масиву. Оскільки ці результати є вибіркою з вхідних даних, додаткова пам'ять для них не відводиться. Крім того, необхідно підрахувати середній оклад для знайдених співробітників. Для цього необхідна змінна дійсного типу.

Проміжні величини. Для пошуку середнього окладу необхідно підрахувати кількість співробітників, для яких виводилися відомості. Заведемо для цього змінну цілого типа.

Алгоритм рішення задачі:

1. Ввести з файлу в масив відомості про співробітників.
2. Організувати цикл виведення відомостей про співробітника:
   - ввести з клавіатури прізвище;
   - виконати пошук співробітника в масиві;
   - збільшити сумарний оклад і лічильник кількості співробітників;
   - вивести відомості про співробітника або повідомлення про їх відсутність;
3. Вивести середній оклад.

Необхідно вирішити, яким чином виконувати вихід з циклу виведення відомостей про співробітників. Умовимося, що для виходу з циклу замість прізвища слід просто натиснути клавішу Enter. Текст програми наведений в прикладі 8.2.

Приклад 2

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;
namespace Person1
{
   class Person
```

```
{                                   // 1
    const int l_name = 30;
    string name;
    int birth_year;
    double pay;
    public Person()           // конструктор без параметрів
    {
        name = "Anonimous";
        birth_year = 0;
        pay = 0;
    }
    public Person(string s)                    // 2 конструктор з параметром
    {
        name = s.Substring(0, l_name);
        birth_year = Convert.ToInt32(s.Substring(l_name, 4));
        pay = Convert.ToDouble(s.Substring(l_name + 4));
        if (birth_year < 0) throw new FormatException();
        if (pay < 0) throw new FormatException();
    }
    public override string ToString()        // 3 перевантажений метод
    {
        return string.Format("Name: {0,30} birth: {1} pay: {2:F2}",name, birth_year, pay);
    }
    public int Compare(string name) // порівняння прізвища
    {
        return (string.Compare(this.name, 0,name + " ", 0, name.Length +
1,StringComparison.OrdinalIgnoreCase));
    }
    // --------------------  властивості класу -------------------------
    public string Name
    {
        get { return name; }
        set { name = value; }
    }
    public int Birth_year
    {
        get { return birth_year; }
        set
        {
            if (value > 0) birth_year = value;
            else throw new FormatException();
        }
    }
    public double Pay
    {
        get { return pay; }
        set
        {
            if (value > 0) pay = value;
            else throw new FormatException();
        }
    }
```

```
    // ------------------  операції класу --------------------------
    public static double operator +(Person pers, double a)
    {
        pers.pay += a;
        return pers.pay;
    }
    public static double operator +(double a, Person pers)
    {
        pers.pay += a;
        return pers.pay;
    }
    public static double operator -(Person pers, double a)
    {
        pers.pay -= a;
        if (pers.pay < 0) throw new FormatException();
        return pers.pay;
    }
};


class Program
{
    static void Main(string[] args)
    {
        Person[] dbase = new Person[100];
        int n = 0;
        try
        {
            StreamReader f = new StreamReader("dbase.txt");    // 4
            string s;
            int i = 0;

            while ((s = f.ReadLine()) != null)                 // 5
            {
                dbase[i] = new Person(s);
                Console.WriteLine(dbase[i]);
                ++i;
            }
            n = i - 1;
            f.Close();
        }
        catch (FileNotFoundException e)
        {
            Console.WriteLine(e.Message);
            Console.WriteLine("Перевірте правильність імені і шляху до файлу!");
            return;
        }

        catch (IndexOutOfRangeException)
        {
            Console.WriteLine("Дуже великий файл!");
            return;
```

```
        }
        catch (FormatException)
        {
           Console.WriteLine("Недопустима дата народження або оклад");
           return;
        }

        catch (Exception e)
        {
           Console.WriteLine("Помилка: " + e.Message);
           return;
        }

        int n_pers = 0;
        double mean_pay = 0;
        Console.WriteLine("Введіть прізвище співробітника");
        string name;
        while ((name = Console.ReadLine()) != "")                    // 6
        {
           bool not_found = true;
           for ( int k = 0; k <= n; ++k )
           {
              Person pers = dbase[k];
              if (pers.Compare(name) == 0)
              {
                 Console.WriteLine(pers);
                 ++n_pers; mean_pay += pers.Pay;
                                          not_found = false;
              }
           }
           if (not_found) Console.WriteLine("Такого співробітника немає");
           Console.WriteLine(
        "Введіть прізвище співробітника або Enter для завершення");
        }
        if (n_pers > 0)
                        Console.WriteLine("Середній оклад: {0:F2}", mean_pay / n_pers);
        Console.ReadKey();
     }
  }
}
```

В операторі 1 описаний клас **Person** для зберігання інформації про одного співробітника. Окрім полів даних, в ньому заданий конструктор, що виконує заповнення полів (оператор 2), перевизначений метод перетворення в рядок (оператор 3), що дозволяє виводити екземпляр об'єкту на екран за допомогою методу WriteLine класу Console, а також описаний метод Compare, в якому прізвище співробітника порівнюється із заданим через параметр.

Порівняння прізвища виконується за допомогою одного з варіантів методу Compare класу string, який дозволяє порівнювати підрядки без

врахування регістра. Після заданого прізвища доданий пропуск, оскільки якщо пропуску немає, то потрібне прізвище є частиною іншого, і цей рядок нам не підходить.

Властивості і операції класу дозволяють виконувати введення і редагування окладу і року народження.

Вхідний файл слід створити до першого запуску програми відповідно до формату, заданого в умові завдання. Можна використовувати будь-який текстовий редактор, що підтримує кодування Unicode (наприклад, Блокнот).

У операторі 4 описаний екземпляр стандартного класу символьного потоку. Цикл 5 виконує порядкове прочитування з файлу в рядок s і заповнення чергового елементу масиву dbase. У операторі 6 організується цикл перегляду масиву. Є видимими лише заповнені при введенні елементи.

Висновки

Поліморфізм – одна з базових концепцій об'єктно-орієнтованого програмування, що дозволяє мати різні реалізації для одного і того ж *методу*, які будуть вибиратися залежно від *типу об'єкту*, переданого до методу при його виклику. Ця концепція в C# реалізується як перевантаження (методів, операцій).

Перевантаження методів означає, що можна мати декілька методів з одним іменем, але різними типами та (або) кількістю параметрів.

Перевантажуватися можуть як конструктори, так і звичайні методи класу.

Крім перевантаження методів в C# дозволяється передавати в метод різну кількість параметрів. Для цього призначене ключове слово *params.*

Новою можливістю в C# є індексатори, які дозволяють отримати доступ до закритих полів класу, які є масивами, за допомогою властивостей.

Перевантаження операцій в C# дозволяє визначати власні операціями над об'єктами створених класів. Крім перевантаження операцій в C# введено операції перетворення типу, які *забезпечують можливість явного і неявного перетворення між типами даних користувача (наприклад, класами).*

Питання для самоконтролю

6. Що таке поліморфізм? Як поліморфізм реалізований в C#?
7. Що таке індексатор, для яких класів він застосовується?
8. Чи можна в C# викликати метод зі змінною кількістю аргументів без перевантаження? Якщо так, то як це зробити?
9. Яка різниця між перевантаженням методів і викликом методів зі змінною кількістю аргументів?
10. Чи може статичний конструктор ініціалізувати поля екземпляра?
11. Навіщо потрібне перевантаження методів і операцій?
12. Опишіть способи перевантаження операцій.
13. Для яких класів має сенс використовувати перевантажені операції?
14. Який тип має параметр унарної операції класу?
15. Чи можна перенавантажувати операції простого і складного призначення?

Тема 8. Інтерфейси

1. **Інтерфейс як окремий випадок абстрактного класу**
2. **Стратегії реалізації інтерфейсів у C#**
3. **Інтерфейси і поля**
4. **Інтерфейси і спадкоємство**

Термін інтерфейс в програмуванні має багато різних значень, наприклад:

- Інтерфейс користувача – засіб взаємодії користувача з програмою. Приклади – консольний інтерфейс, Windows- інтерфейс, Web- інтерфейс.
- Інтерфейс між модулями програми – формальні і фактичні параметри методів (процедур і функцій).
- Інтерфейс як відкрита частина класу – поля і методи з модифікатором public. Таким чином реалізується принцип інкапсуляціїї в ООП.
- **Інтерфейс як окремий випадок абстрактного класу. Саме в цьому значенні ми будемо вживати в лекції цей термін.**

1. Інтерфейс як окремий випадок абстрактного класу

Інтерфейс це повністю абстрактний клас, всі методи якого абстрактні (не містять реалізації). Від абстрактного класу інтерфейс відрізняється деякими деталями в синтаксисі і поведінці.

Синтаксична відмінність полягає в тому, що методи інтерфейсу оголошуються без вказівки модифікатора доступу.

Відмінність в поведінці полягає в жорсткіших вимогах до похідних класів - в похідних класах повинні бути реалізовані всі методи інтерфейсного класу.

Це означає, що клас, який наслідує інтерфейс, зобов'язаний повністю реалізувати всі методи інтерфейсу. У цьому відмінність від класу, що наслідує абстрактний клас, де похідний клас може реалізувати лише деякі методи базового абстрактного класу.

Головне призначення інтерфейсів – забезпечення множинного наслідування класів.

Є ще одне важливе призначення інтерфейсів, що відрізняє їх від абстрактних класів. Абстрактний клас є початковим етапом проектування класу, який в майбутньому отримає конкретну реалізацію. Інтерфейси задають додаткові властивості класу. Один і той самий інтерфейс дозволяє описувати властивості, які можуть мати різні класи.

Кожен клас може визначати елементи інтерфейсу по-своєму. Так досягається поліморфізм: об'єкти різних класів по-різному реагують на виклики одного і того ж методу.

Синтаксис інтерфейсу аналогічний синтаксису класу:

[атрибути] [специфікатори] interface ім'я_інтерфейсу [: предки]
тіло_інтерфейсу [;]

Для інтерфейсу можуть бути вказані специфікатори new, public, protected, internal і private. Специфікатор new застосовується для вкладених інтерфейсів і має такий самий сенс, як і відповідний модифікатор методу класу. Інші специфікатори управляють видимістю інтерфейсу. За замовчанням інтерфейс доступний лише із збірки, в якій він описаний (internal).

Інтерфейс може успадковувати методи декількох інтерфейсів, в цьому випадку предки перераховуються через кому.

Тіло інтерфейсу складають абстрактні методи, шаблони властивостей і індексаторів, а також події.

2. Стратегії реалізації інтерфейсу

Опишемо деякий інтерфейс, реалізація методів якого дозволить класу робити певні перетворення над об'єктами класу і повертати рядки, що представляють результат цих перетворень.

```
interface IStrings
 {
   /// <summary>
   /// Перетворення
   /// </summary>
   /// <returns> результат перетворення </returns>
   string Convert();

   /// <summary>
   /// Шифрування
   /// </summary>
   /// <param name="code"> код шифру </param>
   /// <returns> результат шифрування</returns>
   string Cipher(string[] code);
 }
```

В цьому інтерфейсі є два методи, які будуть реалізувати всі класи, які будуть наслідувати цей інтерфейс. Метод Convert повинен, слідуючи алгоритму, вибраному похідним класом, перетворити об'єкт, повертаючи рядок, а метод Chiper, що повертає рядок, розглядається як шифрування, в алгоритмі якого використовується масив рядків code, переданий методу.

Загалом існує дві стратегії реалізації методів інтерфейсу: як відкритих методів і як закритих. Ми розглянемо тільки першу стратегію, яка використовується найчастіше.

Клас, що успадковує інтерфейс і реалізує його методи, може оголосити відповідні методи класу **відкритими (public)**. У методів інтерфейсу не задані модифікатори доступу. Побудуємо приклад класу, що успадковує інтерфейс IStrings:

Приклад 1

```
 /// <summary>
 /// Успадковує інтерфейс IStrings
 /// реалізуючи його методи як відкриті (public)
 /// </summary>
 class SimpleText : IStrings
 {
//поля класу
string text;
static string[] codeTable =
{
  "абвгдеёжзийклмнопрстуфхцчшщыьъэюя ,.!?;:",
  "ъьыщшчцхфуэюя ,.!?;:тсрпонмлкйабвгдеёжзи"
};
//Конструктори
public SimpleText()
{
   text = "Простий текст!";
}
public SimpleText(string txt)
{
   text = txt;
}
public string Text
{
   get { return text; }
}
}
```

Побудований клас є звичайним класом, що містить текстове поле text, 2 конструктори, метод - властивість, що забезпечує доступ до закритого поля. Але оскільки клас оголосив себе спадкоємцем інтерфейсу IStrings, він зобов'язаний реалізувати методи інтерфейсу, запропонувавши деяку їх реалізацію. Ось приклад подібної реалізації:

```
/// Реалізація інтерфейсів
/// <summary>
/// Видалення пробілів в полі text
/// перетворення до нижнього регістра
/// </summary>
/// <returns>  перетворений рядок </returns>
public string Convert()
{
```

```
    string res = "";
     foreach (char sym in text)
    if (sym != ' ') res += sym.ToString();
     res = res.ToLower();
     return res;
  }
  /// <summary>
  /// шифрування поля text
  /// з використанням таблиці кодування символів
  /// </summary>
  /// <param name="code"> таблиця кодування </param>
  /// <returns> зашифрований текст </returns>
  public string Cipher(string[] code)
  {
     string res = "";
     foreach (char sym in text)
     {
    int k = code[0].IndexOf(sym);
    if (k >= 0) res += code[1][k];
    else res += sym.ToString();
     }
     return res;
  }
```

Клас реалізує методи інтерфейсу, роблячи їх відкритими для клієнтів класу і спадкоємців. Клас може побудувати власні методи, використовуючи реалізацію методів інтерфейсу.

```
/// <summary>
  /// Перевірка поля text,  чи є він паліндромом
  ///  після перетворення Convert
  /// </summary>
  /// <returns> true, якщо паліндром  </returns>
  public bool IsPalindrom()
  {
     string txt = Convert();
     for(int i=0, j = txt.Length-1; i<j; i++, j--)
    if(txt[i] != txt[j])  return false;
     return true;
  }
  /// <summary>
  /// Шифрування, задане власною таблицею кодування
  /// </summary>
  /// <returns> зашифрований текст</returns>
  public string Coding()
  {
     return Cipher(codeTable);
  }
```

Розглянемо клас Testing, методи якого дозволять виконати тестування об'єктів створюваних нами класів:

```
/// <summary>
   /// тестовий клас
   /// </summary>
   class Testing
   {
  //поля класу
  const string PAL =
     "А роза упала на лапу Азора";
  static string[] CODE =
  {
     "абвгдежзиклмнопрстуфхцчшыъьэюя",
     "abvgdejziklmnoprstyfhc461w'qux"
  };

  /// <summary>
  /// Тестування класу SimpleText
  /// </summary>
  public void TestText()
  {
     Console.WriteLine("Работа с объектом класу SimpleText! ");
     SimpleText simpleText = new SimpleText(PAL);
     Console.WriteLine("Початковий текст : " + PAL);
     string text;
     text = simpleText.Convert();
     Console.WriteLine("Перетворений текст : " + text);
     if(simpleText.IsPalindrom())
    Console.WriteLine("Це паліндром!");
     text = simpleText.Coding();
     Console.WriteLine("Шифрований текст : " + text);

     Console.WriteLine("Робота з об'єктом інтерфейсу IStrings! ");
     IStrings istrings;
     text = "Це простий текст!";
     Console.WriteLine("Початковий  текст : " + text);
     simpleText = new SimpleText(text);
     istrings = (IStrings)simpleText;
     text = istrings.Convert();
     Console.WriteLine("Перетворений текст : " + text);
     text = istrings.Cipher(CODE);
     Console.WriteLine("Шифрованний текст : " + text);
  }
   }
```

Зверніть увагу, що у класі Testing оголошений як об'єкт класу SimpleText, так і об'єкт istrings інтерфейсу IStrings. У методі TestText об'єкт simpleText створюється звичайним способом при виклику конструктора класу. Потім цей об'єкт викликає як відкритий метод Convert, успадкований від інтерфейсу, так і відкриті методи класу IsPalindrom, Coding, що використовують методи інтерфейсу.

Повний код програми:

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

namespace ConsoleInterface1
{
    interface IStrings
    {
        string Convert();
        string Cipher(string[] code);
    }
    /// <summary>
    /// Успадковує інтерфейс IStrings
    /// реалізуючи його методи як відкриті (public)
    /// </summary>
    class SimpleText : IStrings
    {
        //поля класу
        string text;
        static string[] codeTable =
   {
     "абвгдеёжзийклмнопрстуфхцчшщыьъэюя ,.!?;:",
     "ъьыщшчцхфуэюя ,.!?;:тсрпонмлкйабвгдеёжзи"
   };
        //Конструктори
        public SimpleText()
        {
            text = "Простий текст!";
        }
        public SimpleText(string txt)
        {
            text = txt;
        }
        public string Text
        {
            get { return text; }
        }
        /// Реалізація інтерфейсів
        /// <summary>
        /// Видалення пробілів в полі text
        /// перетворення до нижнього регістра
        /// </summary>
        /// <returns>  перетворений рядок </returns>
        public string Convert()
        {
            string res = "";
            foreach (char sym in text)
                if (sym != ' ') res += sym.ToString();
            res = res.ToLower();
            return res;
        }
        /// <summary>
        /// шифрування поля text
        /// з використанням таблиці кодування символів
        /// </summary>
        /// <param name="code"> таблиця кодування </param>
        /// <returns> зашифрований текст </returns>
        public string Cipher(string[] code)
        {
            string res = "";
```

```
                foreach (char sym in text)
                {
                    int k = code[0].IndexOf(sym);
                    if (k >= 0) res += code[1][k];
                    else res += sym.ToString();
                }
                return res;
            }
        /// <summary>
            /// Перевірка поля text,  чи є він паліндромом
            ///  після перетворення Convert
            /// </summary>
            /// <returns> true, якщо паліндром  </returns>
            public bool IsPalindrom()
            {
                string txt = Convert();
                for (int i = 0, j = txt.Length - 1; i < j; i++, j--)
                    if (txt[i] != txt[j]) return false;
                return true;
            }
            /// <summary>
            /// Шифрування, задане власною таблицею кодування
            /// </summary>
            /// <returns> зашифрований текст</returns>
            public string Coding()
            {
                return Cipher(codeTable);
            }

        }

        class Testing
        {
            //поля класу
            const string PAL =
                "А роза упала на лапу Азора";
            static string[] CODE =
        {
            "абвгдежзиклмнопрстуфхцчшыъьэюя",
            "abvgdejziklmnoprstyfhc461w'qux"
        };

            /// <summary>
            /// Тестування класу SimpleText
            /// </summary>
            public void TestText()
            {
                Console.WriteLine("Работа с объектом класса SimpleText! ");
                SimpleText simpleText = new SimpleText(PAL);
                Console.WriteLine("Початковий текст : " + PAL);
                string text;
                text = simpleText.Convert();
                Console.WriteLine("Перетворений текст : " + text);
                if (simpleText.IsPalindrom())
                    Console.WriteLine("Це паліндром!");
                text = simpleText.Coding();
                Console.WriteLine("Шифрований текст : " + text);

                Console.WriteLine("Робота  с  об'єктом  інтерфейсуа  IStrings!
");
                IStrings istrings;
                text = "Це простий текст!";
                Console.WriteLine("Початковий текст : " + text);
                simpleText = new SimpleText(text);
```

```
                istrings = (IStrings)simpleText;
                text = istrings.Convert();
                Console.WriteLine("Перетворений текст : " + text);
                text = istrings.Cipher(CODE);
                Console.WriteLine("Шифрований текст : " + text);
            }
        }
        class Program
        {

            static void Main(string[] args)
            {
                Testing tint = new Testing();
                tint.TestText();
                Console.ReadKey();

            }
        }
    }
```

Розглянемо ще один приклад. Класи Person і Student (модифікація лекції 9).

Приклад 2.

Створимо інтерфейсний клас з двома абстрактними методами:

```
interface IRating
    {
        void Rating(double R);
        string GetRole(int Course);
    }
```

В класі Person створимо метод GetAge()

```
class Person
    {
        public string Name;        //имя
        public int Age;            // возраст
        public string Role;        // роль
        public string GetName() { return Name; }
        public int GetAge() { return Age; }
    }
```

Модифікуємо клас Student. Додамо наслідування інтерфейсу IRating. В цьому класі є реалізація методів інтерфейсу IRating.

```
class Student : Person,IRating
    {
        public string Facultet;
        public string Group;
        public int Course;
        public Student(string N, int A, string R, string F, string G,
int C)
        {
            //конструктор з параметрами
            Name = N;
            Age = A;
            Role = R;
            Facultet = F;
            Group = G;
            Course = C;
        }
```

```
        public string GetRole(int Course)
        {
            if (Course <= 4)
                Role = "бакалавр";
            else
                Role = "магістр";
            return Role;
        }
        public void Rating(double Student_Rating)
        {
            if (Student_Rating >= 82)
                Console.WriteLine("Привіт відмінникам");
            else
                if (Student_Rating <= 45)
                    Console.WriteLine("Перездача!       Треба       краще
вчитися!");
                else
                    Console.WriteLine("Можна  вчитися ще краще!");
        }
    }
```

Створимо новий клас Prepod, похідний від класу Person з інтерфейсом IRating. В класі Prepod створимо власний конструктор і реалізацію методів інтерфейсу IRating.

```
class Prepod : Person
    {
        string kafedra;

        public Prepod(string N, string R)
        {
            Name = N;
            Role = R;
        }

        public string GetRole(int Course)
        {
            switch (Course)
            {
                case 1:
                return ("Не читаю");
                case 2:
                 return ("Matlab");
                case 3:
                 return ("C#");
                case 4:
                 return ("Технологія програмування");
                case 5:
                 return ("Semantic web and XML");
                case 6:
                 return ("Парадигми програмування");
                default:
                 return ("Неправильно заданий курс");
            }
        }

        public void Rating(double Prepod_Rating)
        {
            Console.WriteLine("Рейтинг викладача" + Prepod_Rating);
        }
        public  string Kafedra
```

```
        {
            get { return kafedra; }
            set
            {
                kafedra = value;
            }
        }
    }
```

В класі Program напишемо код для перевірки роботи класів.

```
class Program
    {
        static void Main(string[] args)
        {
            string r;
            string newRole;
            //дані про студента
            Student newSt = new Student("Іванов", 20, "студент", "КННІ",
"К-81", 4);
            Console.WriteLine("Дані про студента");
            string Name = newSt.GetName();
            int Age = newSt.GetAge();
            Console.WriteLine("Прізвище = " + Name);
            Console.WriteLine("Вік= " + Age);
            newRole = newSt.GetRole(newSt.Course);
            Console.WriteLine("Ви ще = " + newSt.Role);
            Console.WriteLine("Факультет = " + newSt.Facultet);
            Console.WriteLine("група= " + newSt.Group);
            Console.WriteLine("курс= " + newSt.Course);
            Console.WriteLine("Ваш рейтинг?");
            r = Console.ReadLine();
            double rating = Convert.ToDouble(r);
            newSt.Rating(rating);
            //дані про викладача
            Prepod pr = new Prepod("Коротун Т.М.", "зав.кафедрою");
            newRole = pr.Role;
            pr.Kafedra = "КНІС"; // ініціалізація через властивість
Kafedra
            Console.WriteLine("Дані про викладача:  " + pr.GetName());
            Console.WriteLine("Посада:  " + newRole);
            Console.WriteLine("Кафедра: " + pr.Kafedra);
            pr.Rating(100);      // інтерфейсний метод
            Console.WriteLine("Які дисципліни ви викладаєте?");
            Console.WriteLine("На якому курсі?");
            int Cource = int.Parse(Console.ReadLine());
            string disc = pr.GetRole(Cource);
            Console.WriteLine("на " + Cource + "курсі дисципліну  " +
disc);
            Console.ReadLine();
        }
    }
```

3. Інтерфейси і поля

Ми говорили, що в інтерфейсі можна оголошувати тільки методи, а поля не можна (в явному вигляді).

Але в інтерфейсі можна оголосити властивість (шаблон властивості) з методами get і set, що забезпечують доступ до поля.

Ось приклад такого інтерфейсу:

```
 /// <summary>
/// Доступ до полів Name і Age
/// </summary>
interface IFields
{
   string Name { get; set; }
   int Age {get;}
}
```

Створимо клас, що успадковує цей інтерфейс:

```
 /// <summary>
/// Клас, що успадковує інтерфейс IFields
/// Має поля Name і Age
/// доступ до поля Name відкритий клієнтам класу
/// доступ до поля Age закритий і відкритий з перейменуванням!
/// </summary>
class TwoFields:IFields
{
   string name;
   int age;
   public TwoFields()
   {
      name = "Nemo"; age = 37;
   }
   public TwoFields(string name, int age)
   {
      this.name = name; this.age = age;
   }
   public string Name
   {
      get { return name; }
      set { name = value; }
   }
   int IFields.Age
   {
      get { return age; }
   }
   public int WhatAge()
   {
      return age;
   }
}
```

Зверніть увагу: методи доступу до полів, успадковані від інтерфейсу, можна реалізувати як відкриті так і як закриті, відкриваючи їх потім під іншим іменем. У класу завжди має бути можливість перейменовувати імена методів і полів, успадкованих від інтерфейсу.

Додамо в клас Program в метод main код, що дозволяє протестувати роботу з успадкованими полями.

```
public void TestFields()
 {
    Console.WriteLine("Робота з об'єктом класу TwoFields!");
    TwoFields twofields = new TwoFields();
    Console.WriteLine("Ім'я: {0}, Вік: {1}",
       twofields.Name, twofields.WhatAge());
    twofields.Name = "Captain Nemo";
    Console.WriteLine("Робота з інтерфейсним об'єктом  IFields!");
    IFields ifields = (IFields)twofields;
    Console.WriteLine("Ім'я: {0}, Вік: {1}",
       ifields.Name,ifields.Age);
 }
```

Повний код програми.

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

namespace Console3_Lab10
{
    interface IFields
    {
        string Name { get; set; }
        int Age { get; }
    }

    /// <summary>
    /// Клас, що успадковує інтерфейс IFields
    /// Має поля Name і Age
    /// доступ до поля Name відкритий клієнтам класу
    /// доступ до поля Age закритий і відкритий з перейменуванням!
    /// </summary>
    class TwoFields : IFields
    {
        string name;
        int age;
        public TwoFields()
        {
            name = "Nemo"; age = 37;
        }
        public TwoFields(string name, int age)
        {
            this.name = name; this.age = age;
        }
        public string Name
        {
            get { return name; }
            set { name = value; }
        }
        int IFields.Age
        {
            get { return age; }
        }
        public int WhatAge()
        {
            return age;
        }
    }
```

```
        class Program
        {
            static void Main(string[] args)
            {
                Console.WriteLine("Робота з об'єктом класу TwoFields!");
                TwoFields twofields = new TwoFields();
                Console.WriteLine("Ім'я:  {0},  Вік:  {1}",twofields.Name,
twofields.WhatAge());
                twofields.Name = "Captain Nemo";
                Console.WriteLine("Робота  з  інтерфейсним  об'єктом
IFields!");
                IFields ifields = (IFields)twofields;
                Console.WriteLine("Ім'я:  {0},  Вік:  {1}",ifields.Name,
ifields.Age);
                Console.ReadLine();
            }
        }
    }
```

4. Інтерфейси і спадкоємство

Інтерфейс може не мати або мати скільки завгодно інтерфейсів-предків, в останньому випадку він успадковує всі елементи всіх своїх базових інтерфейсів, починаючи з самого верхнього рівня.

Як і в звичайній ієрархії класів, базові інтерфейси визначають загальну поведінку, а їх нащадки конкретизують і доповнюють його. У інтерфейсі-нащадку можна також вказати елементи, які перевизначають успадковані елементи з такою ж сигнатурою. В цьому випадку перед елементом вказується ключове слово **new**, як і в аналогічній ситуації в класах. За допомогою цього слова відповідний елемент базового інтерфейсу приховується. Клас, що реалізує інтерфейс, повинен визначати всі його елементи, у тому числі успадковані.

Інтерфейс, на власні або успадковані елементи якого є явне посилання, має бути вказаний в списку предків класу.

Клас успадковує всі методи свого предка, у тому числі ті, які реалізували інтерфейси. Він може перевизначити ці методи за допомогою специфікатора **new**, але звертатися до них можна буде лише через об'єкт класу. Якщо використовувати для звернення посилання на інтерфейс, викликається не перевизначена версія:

```
interface IBase
{
  void A();
}

class Base : IBase
{
  public void A() { ... }
}

class Derived: Base
```

```
{
  new public void A() { ... }
}

...
Derived d = new Derived ();
d.A();                  // викликається Derived.A();
IBase id = d;
id.A();                 // викликається Base.A();
```

Проте якщо інтерфейс реалізується за допомогою віртуального методу класу, після його перевизначення в нащадку будь-який варіант звернення (через клас або через інтерфейс) призведе до одного і того самого результату.

Метод інтерфейсу, реалізований явною вказівкою імені, оголошувати віртуальним забороняється.

Існує можливість повторно реалізувати інтерфейс, вказавши його ім'я в списку предків класу разом з класом-предком, що вже реалізував цей інтерфейс. При цьому реалізація перевизначених методів базового класу в увагу не береться:

```
interface IBase
{
  void A();
}

class Base : IBase
{
  void IBase.A() { ... }        // не використовується в Derived
}

class Derived : Base, IBase
{
  public void A() { ... }
}
```

Якщо клас успадковує від класу і інтерфейсу, які містять методи з однаковими сигнатурами, успадкований метод класу сприймається як реалізація інтерфейсу. Взагалі при реалізації інтерфейсу враховується наявність "відповідних" методів в класі незалежно від їх походження. Це можуть бути методи, описані в поточному або базовому класі, які реалізують інтерфейс явним або неявним чином.

Висновки:

Головне призначення інтерфейсів – забезпечення множинного наслідування класів. Клас може успадковувати (реалізовувати) методи декількох інтерфейсів. Через те, що інтерфейсний клас не містить реалізації методів, то кожний клас, який успадковує інтерфейс, повинен реалізувати всі його методи. Кожен клас може визначати методи інтерфейсу по-своєму. Так досягається поліморфізм: об'єкти різних класів по-різному реагують на виклики одного і того ж методу.

Клас, що успадковує інтерфейс і реалізує його методи, може оголосити відповідні методи класу відкритими (public) чи закритими (цю стратегію реалізації ми не розглядали).

В інтерфейсі не можна оголошувати в явному вигляді поля, але можна оголосити властивість (шаблон властивості) з методами get і set, що забезпечують доступ до поля.

Інтерфейс може не мати або мати скільки завгодно інтерфейсів-предків, в останньому випадку він успадковує всі елементи всіх своїх базових інтерфейсів, починаючи з самого верхнього рівня. Як і в звичайній ієрархії класів, базові інтерфейси визначають загальну поведінку, а їх нащадки конкретизують і доповнюють його.

Питання для самоконтролю

1. Що означає поняття "інтерфейс" як відкрита частина класу?
2. Що означає поняття "інтерфейс" як окремий випадок класу?
3. Для чого використовуються інтерфейси?
4. Опишіть синтаксис інтерфейсу. Які специфікатори доступу застосовуються в інтерфейсі?
5. У чому відміна між абстрактними класами та інтерфейсами?
6. Чи повинен клас реалізовувати всі методи всіх своїх інтерфейсів-предків?
7. Які є стратегії реалізації методів інтерфейсу? Яка стратегія розглядалася в лекції?
8. Чи може інтерфейс включати методи з реалізацією?
9. Чи може інтерфейс містити ініціалізацію полів класу?
10. Як створити новий клас, який наслідує методи і поля двох класів одночасно?