ТЕМА 2: ВИРАЗИ. ОПЕРАТОРИ. ЛОГІЧНІ ОПЕРАЦІЇ. РЯДКИ. МАСИВИ.

Теоретичні відомості.

Булевий тип. Представляє логічне значення, що виражається літералами `true` або `false`.

Операції порівняння == і != порівнюють значення будь-якого типу на еквівалентність.

```
int x = 1;
int y = 2, z = 1;
bool res = x == y; //false
     res = x == z; //true
```

Логічні операції. Операції `&&`, `||` та `&`, `|` реалізують швидкі та повні логічні операції І та АБО. Вони часто використовуються із оператором заперечення `!`, що виражає логічну операцію НЕ. Наприклад:

```
bool windy, rainy, sunny, useUmbrella;
windy = true;
rainy = true;
sunny = false;
useUmbrella = !windy && (rainy || sunny); //false
```

Умовна операція або тернарна (єдина, яка має три аргументи) записується у вигляді `q?t:f;`, де у випадку, якщо `q` дорівнює `true`, то результатом операції є `t`, в іншому разі – `f`. Наприклад:

```
double a = 2, b = 3, max;
max= (a >= b) ? a : b; //3
```

Символьний тип `char` в C# представляє символ Unicode. Літерал `char` записується як символ в одинарних лапках:

```
char c='A';
```

Рядковий тип `string` в C# представляє незмінну послідовність символів Unicode. Рядковий літерал вказується в подвійних лапках:

```
string s="Приклад рядкового літерала";
```

Для символьних та рядкових літералів визначені **керуючі символи**.

Керуючі символи	Що означають	Керуючі символи	Що означають
\'	Одинарні лапки	\f	Переведення сторінки
\"	Подвійні лапки	\n	Новий рядок
\\	Зворотна коса риска	\r	Повернення каретки
\0	Пусто	\t	Горизонтальна табуляція
\a	Звуковий сигнал	\v	Вертикальна табуляція
\b	Забій		

Префіксом **@** оснащуються **дослівні** рядкові літерали, які не підтримують керуючі символи.

```
string a1 = "c:\\tmp\\t1.txt"; //c:\tmp\t1.txt
            radioButton3.Text = @"c:\tmp\
                                t2.txt"; // c:\tmp\
                                         //          t2.txt
```

Для рядків визначена **операція конкатенції +.** Причому операнд може бути не рядковим значенням:

```
string s="Abc"+10; //Abc10
```

Масиви представляють фіксовану кількість змінних (елементів масиву) заданого типу. Масив визначається квадратними дужками після типу елементів:

```
char[] masyv=new char[10]; //масив з 10 елементів
```

Вираз ініціалізації масиву дозволяє оголошувати і заповнювати масив одним оператором:

```
char[] masyv=new char[]
             {'A','B','C','D','E','F','G','H','I','J'};
```

або

```
char[] masyv={'A','B','C','D','E','F','G','H','I','J'};
```

При створенні масиву без заповнення значеннями елементи масиву завжди ініціалізуються стандартними значеннями відповідного типу, які є результатом побітового обнулення пам'яті.

```
int[] M=new int[100];
Console.Write(M[55]); //0
```

За допомогою квадратних дужок також вказується індекс елемента в масиві, щоб забезпечити доступ до цього елемента за його позицією. Масив індексується, починаючи з 0:

```
Console.Write(masyv[1]);//B
masyv[10]='K';//помилка індексації масиву
```

Властивість `Length` масиву повертає **кількість елементів** масиву:

```
Console.Write(masyv.Length);//10
```

Багатовимірні масиви можуть бути прямокутними і зубчатими. Прямокутні масиви створюються з використанням ком для відокремлення кожного виміру. Наприклад, для прямокутного масиву розміру 5 на 10:

```
int [,]M=new int [5,10];
```

Метод `GetLength` масиву повертає довжину для заданого виміру, починаючи з 0:

```
Console.WriteLine(M.GetLength(0));//5
Console.WriteLine(M.GetLength(1));//10
Console.WriteLine(M.GetLength(2));//помилка індексації
```

Зубчаті масиви являють собою масиви масивів та записуються із використанням послідовних пар квадратних дужок. Наприклад, двовимірний зубчатий масив, що являє собою масив із трьох масивів (не обов'язково однакової довжини), визначається:

```
int [][] N=new int[3][];
```

Кожен внутрішній масив неявно ініціюється `null`, та повинен бути створений вручну:

```
N[0]=new int[3];
N[1]=new int[5];
N[2]=new int[8];
```

Для визначення довжини внутрішніх масивів потрібно використовувати властивість `Length`:

```
Console.WriteLine(N[0].Length);//3
Console.WriteLine(N[1].Length);//5
Console.WriteLine(N[2].Length);//8
```

Неявна типізація. Якщо компілятор може вивести тип змінної під час оголошення і ініціалізації, та замість вказування типу можна використати ключове слово `var`.

```
var a1 = "c:\\tmp\\t1.txt";
```

Пріоритети і асоціативність операцій. Якщо вираз містить декілька операцій, порядок їх обчислення визначається пріоритетами та асоціативністю. Операції з більш

високим пріоритетом виконуються у першу чергу. Якщо операції мають однакові пріоритети, то порядок виконання визначається асоціативністю.

Бінарні операції (крім присвоєння, лямбда операції та операції об'єднання з `null`) є лівоасоціативні, тобто обчислюються зліва направо. Наприклад,

```
8/4/2       //1
8/(4/2)     //4
```

Оператори. Функції складаються з операторів, що виконуються один за одним. Блок операторів – це послідовність операторів, що знаходяться в фігурних дужках {}.

Оператори оголошення.

```
char a = '5';
const int b= 5;
b++; //помилка на етапі компіляції
```

Локальні змінні. Областю видимості локальної змінної або локальної константи є поточний блок {} та всі вкладені блоки.

```
static void Main()
{
     int x;
     {
          int y;
          int x; //помилка
     }
     {
          int y; //помилки немає – y не знаходиться в області видимості
     }
     Console.Write(y); //помилка - y знаходиться поза областю видимості
}
```

Оператори-вирази. Являють собою вирази. Вони мають або змінити стан, або викликати щось, що може змінити стан. Наприклад:

- вирази присвоєння;
- вирази виклику методів;
- вирази створення екземплярів об'єктів.

Оператори вибору. Для умовного керування потоком виконання програми. В C# існує декілька операторів, що дозволяють змінювати потік команд:

- оператори розгалуження і вибору (`if`, `switch`);
- умовна операція (`? :`);
- оператори циклу (`while`, `do..while`, `for`, `foreach`).

Оператор розгалуження:

```
if (x>y) maxx=x; else maxx=y;
```

Оператор вибору:

```
int x, y;
x=5;y=2;
     switch (x)
     {
          case 1:
          case 2:
          case 3:
                    Console.WriteLine("від 1 до 3");
                    break;
          case 5: Console.WriteLine(555);
                    break;
```

```
        default: Console.WriteLine(123);
                 break;
    }
```

В операторі `switch` можна використовувати лише такі значення, які можуть бути статично обчисленими (цілочисельний, `bool`, `enum`, `string`). Якщо один і той самий код має виконуватися для декількох значень, застосовується послідовний запис конструкції `case`.

Оператори ітерацій.

Цикл `while` виконується до тих пір, поки умова істина.

```
int x=0;
    while (x<3)
    {
        Console.WriteLine(++x);
    }   //123
```

Цикл `do..while` відрізняється від циклу `while` лише тим, що умова перевіряється лише після виконання блоку операторів.

```
int x=0;
do
    {
        Console.Write(x);
        x++;
    }
while (x<3);  //012
```

Цикл `for` включає спеціальні конструкції (не обов'язкові) для ініціалізації та ітерації змінної циклу:

`for` (*конструкція ініціалізації*; *конструкція умови*; *конструкція ітерації*) *оператор або блок операторів*;

Наприклад:

```
int x,y;
for (x=0,y=0;x<3;x++,y++)
    Console.Write(x+y); //024
```

Оператор `foreach` забезпечує проходження за всіма елементами перечисляємого аргумента:

```
foreach (char c in "example")
        Console.Write(c+" "); //e x a m p l e
```

Оператори переходу в C# є `break`, `continue`, `goto`, `return`, `throw`.

Оператор `break` припиняє виконання тіла ітерації або оператора `switch`.

Оператор `continue` пропускає оператори, що залишились після цього оператора в тілі циклу, і розпочинає наступну ітерацію.

Windows Forms програма:

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

namespace WindowsFormsApplication1
{

    public partial class Form1 : Form
    {

        public Form1()
        {
            InitializeComponent();

        }

        public UnitConverter USD2UAH = new UnitConverter(26, "USD", "UAH");
        public UnitConverter EUR2UAH = new UnitConverter(29, "EUR", "UAH");
        public UnitConverter KM2M = new UnitConverter(1000, "Kilometer", "Meter");

        private void button1_Click(object sender, EventArgs e)
        {
            label1.Text = (textBox1.Text + " "+USD2UAH.nameFrom+" = "+
                USD2UAH.Convert(Convert.ToDouble(textBox1.Text))).ToString()+
                " "+USD2UAH.nameTo;
        }

        private void Form1_Load(object sender, EventArgs e)
        {
            radioButton1.Text = USD2UAH.nameFrom + " to " + USD2UAH.nameTo;
            radioButton1.Checked = true;
            radioButton2.Text = EUR2UAH.nameFrom + " to " + EUR2UAH.nameTo;
            radioButton3.Text = KM2M.nameFrom + " to " + KM2M.nameTo;
        }
    }
    public class UnitConverter
    {
        double Koef;
        string NameFrom, NameTo;
        static int UnitConverterCount;
        public UnitConverter(double koef, string nameFrom, string nameTo)
        {
            Koef = koef;
            NameFrom = nameFrom;
            NameTo = nameTo;
            UnitConverterCount++;
        }
        public double Convert(double vhid)
        {
            return Koef * vhid;
        }
        public string nameFrom { get {return NameFrom;} }
        public string nameTo { get { return NameTo; } }

    }
}
```

Так, як із простих функцій можна утворювати складні функції, так і з примітивних типів можна створювати складні типи. Наприклад:

```
public class UnitConverter
  {int Koef;   //Поле
   public UnitConverter (int UnitKoef)
    {Koef=UnitKoef;}     //Конструктор
   public int Convert (int Unit)
    {return Unit*Koef;}} //Метод
```

Проте екземпляр спеціального типу створюється за допомогою наступного оператора із використанням службового слова `new`:

```
UnitConverter Tona2Kilo = new UnitConverter (1000);
```

В цьому випадку безпосередньо після створення екземпляра об'єкта викликається конструктор об'єкта за допомогою команди `new` для виконання ініціалізації.

Ключове слово `public` відкриває для доступу відповідне поле чи метод для інших класів, тоді як інші поля і методи, що не позначені ключовим словом `public` відносяться до закритих деталей реалізації. Відповідно до об'єктно-орієнтованої термінології кажуть, що відкриті члени класу інкапсулюють закриті члени класу.

Поля і методи, які оперують не з екземплярами типів, а із самим типом, є статичними та повинні помічатися як `static`.

Об'єктно-орієнтоване програмування. В ООП використовуються класи (типи, що містять дані та функції для маніпулювання даними) для інкапсуляції (об'єднання) даних (атрибутів) і методів (поведінки). Класи можна розглядати як шаблони для створення об'єктів (екземплярів класу).

```
public class MyRadioButtonClass : System.Windows.Forms.RadioButton
     {
          public MyRadioButtonClass()
          {
               this.AutoSize=true;
               this.CheckedChanged+=ChkChnd;
          } //Конструктор
          public void ChkChnd(object sender, EventArgs e)
          {
               int j=0;
               foreach (System.Windows.Forms.RadioButton RB in
this.Parent.Controls)
               {
                    if (RB.Checked)
                    {
                         this.Parent.Tag=j;
                         break;
                    }
                    j++;
               }
          }//Метод
     }
```

Об'єкт є представленням чогось, що існує в реальному світі з відображенням того, чим він є, і того, що він робить. Об'єкти мають властивість приховування своєї реалізації від інших об'єктів, не зважаючи на те, що деякі об'єкти можуть взаємодіяти один з одним через чітко визначені інтерфейси.

В оголошенні класу можуть бути присутні такі компоненти:

Перед ключовим словом `class`	*Атрибути і модифікатори класу.* Модифікаторами не вкладених класів є: `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe`, `partial`.
Після імені класу	*Параметри узагальнених типів, базовий клас, інтерфейси*
У фігурних дужках	*Члени класу* (до них відносяться: *властивості*, *конструктори*, *методи*, *поля*, *події*, *індексатори*, *перевантажені оператори*, *вкладені типи* і *фіналізатор*)

Поле – це змінна, яка є членом класу або структури. Наприклад:

```
class price
     {
          public string Tovar;
          private decimal cina_opt;
          public decimal cina_rozdrib;
     }
```

З полями дозволено використовувати наступні модифікатори:

Статичний модифікатор	`static`
Модифікатори доступу	`public`, `internal`, `private`, `protected`
Модифікатор успадковування	`new`
Модифікатор не безпечного коду	`unsafe`
Модифікатор доступу лише для читання	`readonly`
Модифікатор багатопоточності	`volatile`

Ініціалізація полів є необов'язковою. Не ініційовані поля отримують свої стандартні значення (`0`, `\0`, `null`, `false`). Ініціалізатори полів виконуються перед конструкторами.

Методи виконують деякі дії у послідовності операторів. Метод може отримувати вхідні дані з коду, який його викликає, за рахунок вказування параметрів і повертати вихідні дані через вказування типу, що повертається методом, або через параметри `ref`/`out`. Якщо метод не повертає даних, то слід вказати тип `void`. Сигнатура методу (включає тип і кількість параметрів, спосіб передачі параметрів, але не включає імена параметрів і тип, що повертається методом) має бути унікальною в межах типу. З методами дозволено застосовувати наступні модифікатори:

Статичний модифікатор	`static`
Модифікатори доступу	`public`, `internal`, `private`, `protected`
Модифікатори успадковування	`new, virtual, abstract, override,sealed`
Модифікатор частинного методу	`partial`

Модифікатор некерованого коду	`unsafe, extern`

Тип може перевантажувати методи (мати декілька методів з одним і тим же іменем) за умови, що їх сигнатури будуть різними. Наприклад:

```
void MyMethod(int x) {}
void MyMethod(int y) {}//Помилка компіляції
void MyMethod(int x,char c) {}
void MyMethod(char d,int y) {}
double MyMethod(int x,char c) {}//Помилка компіляції
void MyMethod(int[] x) {}
void MyMethod(ref int x) {}
void MyMethod(out int x) {}//Помилка компіляції
```

Конструктори екземплярів виконують код ініціалізації класу або структури. Конструктор записується подібно до методу, лише в якості імені вказується ім'я типу:

```
class Panda
{
     string name;//Визначення поля
     public Panda(string n)//Визначення конструктора
     {
          name=n;//Код ініціалізації
     }
}
...
Panda P=new Panda("Мишко");//Виклик конструктора
```

Конструктори допускають застосування наступних модифікаторів:

Модифікатори доступу	`public, internal, private, protected`
Модифікатор некерованого коду	`unsafe, extern`

Клас або структура може перевантажувати конструктори. Для запобігання дублювання коду один конструктор може викликати інший, використовуючи ключове слово `this`:

```
class Panda
{
     string name;
     int BrYear;
     public Panda(string n){name=n;}
     public Panda(string n, int b): this (n) { BrYear=b;}
}
```

Викликати таким чином визначений конструктор можна з різною кількістю аргументів:

```
Panda P1=new Panda("Васько");
Panda P2=new Panda("Мишко", 2015);
```

Властивості класу виглядають як поля, але при зчитуванні або встановленні значень містять логіку. Реалізується ця логіка за допомогою відповідно блоків `get` і `set`:

```
class price
{
     private decimal cina;
     public decimal Cina
     {
          get {return cina;} //виконується при зчитуванні
          set {cina=value;} // виконується при присвоєнні
```

```
    }
}
```

Засіб доступу `get` виконується при зчитуванні властивості, а `set` – при присвоєнні. Засіб доступу `set` має неявний параметр `value` з типом властивості, що присвоюється. Хоча доступ до властивостей відбувається таким самим способом, як і до полів, проте властивості відрізняються тим, що надають програмісту повний контроль над отриманням і встановленням їх значень. Це дозволяє обирати будь-яке необхідне внутрішнє представлення, не показуючи внутрішні деталі користувачу.

Якщо вказати лише засіб доступу `get`, то властивість буде доступна лише для читання, а якщо лише `set` – тільки для запису.

З властивостями дозволено застосовувати наступні модифікатори:

Статичний модифікатор	`static`
Модифікатори доступу	`public, internal, private, protected`
Модифікатори успадковування	`new, virtual, abstract, override, sealed`
Модифікатор некерованого коду	`unsafe, extern`

Модифікатори доступу можна окремо застосовувати для засобів доступу `get` і `set`:

```
class price
{
     private decimal cina;
     public decimal Cina //властивість більш доступна
     {
          get {return cina;}
          private set {cina=value;} // засіб доступу менш доступний
     }
}
```

В C# елементом програмування є клас. Об’єкти створюються з цих класів, а функції інкапсулюються в класах як методи.

Об’єкти підтримують концепцію успадковування для розширення або налаштування класу, шляхом якої похідний клас успадковує поля і методи базового класу. Успадковування дозволяє повторно використовувати функціональність цього класу замість того, щоб розробляти її заново. Клас може успадковуватися лише від одного класу, але сам може бути успадкований багатьма класами, утворюючи, таким чином, ієрархію класів.

```
class price_rozdr : price
{
     private decimal nacenka=0.2M;//числовий суфікс «M»
     public decimal Cina_rozdr//розширення базового класу властивістю
     {
          get {return Cina*nacenka;}//похідний клас успадкував
                                    //властивість Cina базового класу
     }
}
```

Успадковування також дозволяє трактувати об’єкти похідного класу як об’єкти його базового класу.

Поліморфізм можна виразити як «один інтерфейс, безліч функцій». Поліморфізм може бути статичним і динамічним. Статичний поліморфізм реалізується шляхом

перевантажень методів. Наприклад, можна мати декілька описів методу в залежності від типу вхідного параметра:

```
class Printdata
{
     public void print(int i)
     {
          Console.WriteLine("Printing int: "+i);
     }
     public void print(double f)
     {
          Console.WriteLine("Printing float: "+f);
     }
     public void print(string s)
     {
          Console.WriteLine("Printing string: "+s);
     }
}
```

В залежності від типу вхідного параметра буде викликатись той чи інший варіант методу `print()`.

```
public static void Main(string[] args)
{
     Printdata p = new Printdata();
     p.print(5);// Printing int: 5
     p.print(3.14);// Printing float: 3,14
     p.print("Hello C#");// Printing string: Hello C#
}
```

Динамічний поліморфізм реалізовується за допомогою віртуальних функцій і абстрактних класів. Метод, позначений як віртуальний (`virtual`), можна перевизначати (за допомогою ключового слова `override`) у підкласах, в яких необхідно специфічним способом реалізувати даний метод. Сигнатури, доступність і типи значень, що повертаються, віртуального і перевизначеного методів повинні бути ідентичними.

```
class shape
{
   public virtual void draw() {Console.WriteLine("Draw shape");}
}
class rectangle :shape
{
   public override void draw() {Console.WriteLine("Draw rectangle");}
}
class circle :shape
{
   public override void draw() {Console.WriteLine("Draw circle");}
}
```

Універсальний метод для їх виклику може мати вигляд:

```
void display(shape f)
{
     f.draw();
}
```

Питання вибору конкретного методу в залежності від типу змінної вирішується на етапі виконання. Наприклад:

```
shape a = new shape();
rectangle b = new rectangle();
```

```
circle c = new circle();
shape d = new circle();
List<shape> L = new List<shape>();
L.Add(a);L.Add(b);L.Add(c);L.Add(d);
foreach (shape i in L) {new Program().display(i);}
// Draw shape
// Draw rectangle
// Draw circle
// Draw circle
```

Клас, який позначений як абстрактний (`abstract`), не дозволяє створення екземплярів. Замість цього можна створити лише екземпляри його підкласів.

В абстрактних класах дозволяється створювати абстрактні члени. Абстрактні члени схожі на віртуальні, за виключенням того, що вони не представляють стандартну реалізацію. Реалізація абстрактних членів може бути визначена в підкласах.

За допомогою ключового слова `sealed` перевизначена функція або клас може бути запакований для недопущення подальшого перевизначення в інших підкласах. Наприклад:

```
class circle :shape
{
   public sealed override void draw()
   {
       Console.WriteLine("Draw circle");
   }
}
```

Ключове слово `base` схоже до `this`. Воно служить для:

- доступу до методу базового класу при перевизначенні її в підкласі;
- виклику конструктора базового класу.

Приклад виклику базового методу при реалізації підкласу:

```
class circle :shape
{
     public override void draw()
     {
          Console.Write("Реалізація базового класу: ");
          base.draw();
          Console.WriteLine("Draw circle");
     }
}
// Реалізація базового класу: Draw shape
// Draw circle
```

Якщо у базовому класі відсутній конструктор без параметрів, то при успадкуванні у підкласі мають бути оголошені власні конструктори, тому що конструктори базового класу ніколи не успадковуються автоматично. Якщо ж у базовому класі оголошено конструктор без параметрів, то він викликається автоматично у підкласах, перевизначені конструктори яких не використовують виклик конструктора базового класу за допомогою ключового слова `base`. Наприклад, нехай визначено базовий клас з віртуальним методом і двома конструкторами:

```
class shape {
     public string s;
     public virtual void  draw() {
          Console.WriteLine("Метод draw базового класу з параметром
конструктора "+s);
```

```
     }
     public shape(string n) {
          Console.WriteLine("Конструктор базового класу з параметром
"+(s=n));
     }
     public shape() {
          Console.WriteLine("Конструктор базового класу без параметру");
     }
}
```

І нехай маємо підклас з перевизначеним методом і загальний метод `display`:

```
class rectangle :shape
{
     public override void draw()
     {
     Console.WriteLine("Перевизначений класом rectangle метод draw "+s);
     }
     public rectangle (string n)
     {
     Console.WriteLine("Конструктор перевизначеного класу rectangle з
параметром "+(s=n));
     }
}
void display(shape f)
{
     f.draw();
}
```

Тоді при створенні об’єкта підкласу неявно викликається спочатку конструктор базового класу без параметрів, а потім конструктор підкласу з параметром:

```
rectangle b = new rectangle("rectangle");
// Конструктор базового класу без параметру
// Конструктор перевизначеного класу rectangle з параметром rectangle
```

У підкласах конструктори та методи базового класу можуть бути викликані із підкласу за допомогою ключового слова `base`.

```
class circle :shape
{
     public override void draw()
     {
          Console.Write("Реалізація базового класу: ");
          base.draw();
          Console.WriteLine("Перевизначений класом circle метод draw з
параметром "+s);
     }
     public circle (string n) : base(n)
     {
          Console.WriteLine("Конструктор перевизначеного класу circle з
параметром "+(s=n));
     }
 }
```

Тоді при створенні об’єкта підкласу викликається спочатку конструктор базового класу з параметром, а потім конструктор підкласу з параметром:

```
circle c = new circle("Draw circle");
// Конструктор базового класу з параметром Draw circle
// Конструктор перевизначеного класу circle з параметром Draw circle
```

Структура схожа на клас, але володіє наступними ключовими відмінностями:

- структура є типом значень, в той час, коли клас – тип посилань;
- структура не підтримує успадковування.

Структура може мати ті ж самі члени, що і клас, за виключенням:
- конструктора без параметрів;
- фіналізатора;
- віртуальних членів.

Прикладами структур можуть слугувати числові типи, для яких більш природнім способом присвоювання є копіювання значень, а не посилань.

Семантика конструювання структури може бути описана наступним чином:
- конструктор без параметрів, який неможливо перевизначити, існує неявно; він виконує побітове обнулення полів структури;
- при визначенні конструктора структури кожному з полів обов'язково має бути присвоєне значення;
- ініціалізатори полів в структурах не існують.

Наприклад, описати структуру та викликати її конструктори можна так:
```
public struct point
{
    public int x,y;
    public point (int x, int y){this.x=x; this.y=y;}
}
class Program
{
    static void Main()
    {
        point P0=new point();
        point P1=new point(1,1);
        Console.WriteLine(P1.x);
    }
```
Наступний код спричинить генерацію трьох помилок:
```
public struct point
{
    public int x=0; //не допускається ініціалізація полів
    public int y;
    public point () {} //конструктор без параметрів не можна визначати
    public point (int x, int y) {this.x=x;} //потрібно визначити ще й
                                            //поле point.y
}
```
Інтерфейс схожий на клас, але надає лише специфікацію, а не реалізацію для своїх членів. Інтерфейс володіє наступними особливостями:
- всі члени інтерфейсу є неявно абстрактними; на противагу цьому, клас може мати як абстрактні члени, так і конкретні члени з реалізацією;
- клас (або структура) може реалізовувати декілька інтерфейсів; на противагу цьому клас може бути успадкований лише від одного класу, а структура взагалі не підтримує успадкування.

Опис інтерфейсу схожий на опис класу, але при цьому ніякої реалізації не надається, оскільки всі члени інтерфейсу неявно абстрактні. Всі ці члени обов'язково мають бути реалізовані класами і структурами, які включають даний інтерфейс. Інтерфейс може містити лише методи, властивості, події та індексатори, що відповідає членам класу, які можуть бути абстрактними.

Члени інтерфейсу завжди неявно `public`, і для них не можна оголошувати модифікатори доступу. Реалізація інтерфейсу означає надання реалізації `public` для всіх його членів. Жоден з членів інтерфейсу не може бути оголошений як `static`.