

ІНТЕРФЕЙСИ

Інтерфейси – це спеціальні класи, що містять тільки назви методів (без реалізації методів та без полів). Реалізацію ж, цих методів покладено на класи-нащадки інтерфейсів. Оголошують інтерфейси за допомогою ключового слова **interface**. **Ім'я_інтерфейсу** – зазвичай починається з великої літери І. Основне застосування інтерфейсів – полегшить синхронізацію програмного коду написаного різними розробниками.

```
[модифікатор доступу] interface <ім'я інтерфейсу> {  
    // оголошення методів  
    [мод._доступу] <тип_значення1> <ім'я_методу1>  
        ([параметри1]);  
    [мод._доступу] <тип_значення2> <ім'я_методу2>  
        ([параметри2]);  
}  
//...
```

Нехай нам потрібно, щоб всі графічні об'єкти мали дві координати, відображалися та рухалися. Можемо створити інтерфейс:

```
using System;  
public interface IGraphObject {  
    // властивості для координат  
    int X {  
        get;  
        set;  
    }  
    int Y {  
        get;  
        set;  
    }  
    //метод для відображення  
    void Draw();  
  
    //метод для переміщення  
    void Move(int dx, int dy);  
}
```

ІНТЕРФЕЙСИ

Успадкуємо від інтерфейсу клас `Object2D`:

```
public class Object2D : IGraphObject {
    protected int x, y; //поля-координати

    //конструктор
    public Object2D(int x, int y) {
        this.x = x;
        this.y = y;
    }

    //реалізація успадкованих властивостей
    public int X {
        get { return x; }
        set { x = value; }
    }
    public int Y {
        get { return y; }
        set { y = value; }
    }

    //реалізація успадкованих методів
    //метод для відображення
    public virtual void Draw() {
        Console.WriteLine(
            "Об'єкт з координатами {0},{1}", x, y);
    }

    //метод для переміщення
    public void Move(int dx, int dy) {
        x += dx;
        y += dy;
        Draw();
    }
}
```

ІНТЕРФЕЙСИ

Перевага інтерфейсів над класами, це можливість успадковувати один клас від кількох інтерфейсів:

```
public interface IRotated {  
    //метод для обертання  
    void Rotate(int x0, int y0, double angle);  
}
```

Додаємо його в список предків для `Object2D` і нам доведеться додатково описати метод `Rotate`

```
public class Object2D : IGraphObject, IRotated {  
    protected int x, y; //поля-координати  
  
    //конструктор  
    public Object2D(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    //реалізація успадкованих властивостей  
    public int X {  
        get { return x; }  
        set { x = value; }  
    }  
  
    public int Y {  
        get { return y; }  
        set { y = value; }  
    }  
  
    //реалізація успадкованих методів  
    //метод для "відображення"  
    public virtual void Draw() {  
        Console.WriteLine(  
            "Об'єкт з координатами {0},{1}", x, y);  
    }  
  
    //метод для "переміщення"  
    public void Move(int dx, int dy) {  
        x += dx;  
        y += dy;  
        Draw();  
    }  
  
    //метод для обертання  
    public void Rotate(int x0, int y0, double angle) {  
        // метод що описує обертання  
        // навколо заданого центру  
        // ...  
    }  
}
```

ІНТЕРФЕЙС IComparable

Ще одна перевага інтерфейсів, це можливість за допомогою підтримки стандартних інтерфейсів середовища .Net користатися різноманітними методами стандартних об'єктів. Наприклад, щоб сортувати об'єкти за допомогою класу **System.Array** – достатньо успадкувати їх від стандартного інтерфейсу **IComparable** і описати метод **int CompareTo(object obj)**. Цей метод повинен повертати:

- нуль – у випадку коли об'єкти, що порівнюються однакові;
- від'ємне ціле число – коли поточний об'єкт менше об'єкту- аргументу;
- додатне ціле число – коли поточний об'єкт більше об'єкту- аргументу.

Наприклад, припустимо:

- одна точка рівна іншій, коли обидві їх координати рівні;
- точка **A(x1, y1)** більша точки **B(x2, y2)**, коли перша координата **A** більша за першу координату **B** (**x1 > x2**), або коли **x1 = x2** і **y1 > y2**.

ІНТЕРФЕЙС IComparable

```
using System;

public class Point : IComparable {
    double x, y;

    public Point(double x, double y) {
        this.x = x;
        this.y = y;
    }
    public override string ToString() {
        return String.Format(
            "Точка({0},{1})", this.x, this.y
        );
    }

    //метод від IComparable
    public int CompareTo(object obj) {
        Point p = (Point)obj;
        if(this.x > p.x) {
            return 1;
        } else if(this.x < p.x) {
            return -1;
        } else {
            if (this.y > p.y) {
                return 1;
            } else if(this.y < p.y) {
                return -1;
            } else {
                return 0;
            }
        }
    }
}
```

```
class Program {
    static void Main(string[] args) {
        Point p1 = new Point(5, 8);
        Point p2 = new Point(7, 12);
        Point[] points = new Point[5];
        points[0] = p1;
        points[1] = p2;
        points[2] = new Point(5, 10);
        points[3] = new Point(5, 1);
        points[4] = new Point(15, 10);
        // для сортування використовується
        // стандартний метод Sort
        Array.Sort(points);

        foreach(Point p in points) {
            Console.WriteLine(p);
        }
        Console.ReadKey();
    }
}
```

Точка (5,1)
Точка (5,8)
Точка (5,10)
Точка (7,12)
Точка (15,10)

ОПЕРАТОРИ **IS** ТА **AS**

Щоб перевірити чи підтримує об'єкт той чи інший інтерфейс, можна скористатися одним з операторів **is** або **as**.

Оператор is – використовується для перевірки сумісності змінної або об'єкта одного типу з іншим типом даних і повертає значення **true** або **false**.

```
int i;  
if(i is IComparable){  
    Console.WriteLine("Підтримує IComparable");  
}
```

Підтримує IComparable

Оператор as – використовується для перетворення змінної одного типу в інший сумісний тип. Якщо типи не сумісні поверне **null**

```
Point p0 = new Point(1, 2);  
IConvertible c = p0 as IConvertible;  
if(c == null) {  
    Console.WriteLine(  
        "Не підтримує IComparable"  
    );  
}
```

Не підтримує IComparable

СТРУКТУРИ

Структура є типом значення, яке використовується для інкапсуляції невеликих груп пов'язаних змінних та представляють ще один спосіб створення власних типів даних в C#. Як і класи, структури можуть зберігати стан у вигляді змінних і визначати поведінку у вигляді методів. Структури не підтримують успадкування і не можуть містити віртуальні методи.

```
1  using System;
2
3  struct User
4  {
5      public string name;
6      public int age;
7
8      public void DisplayInfo()
9      {
10         Console.WriteLine($"Name: {name} Age: {age}");
11     }
12 }
13
14 class Program {
15     public static void Main (string[] args) {
16         User tom;
17         tom.name = "Tom";
18         tom.age = 34;
19         tom.DisplayInfo();
20
21         Console.ReadKey();
22     }
23 }
24 }
```

```
> mcs -out:main.exe main.cs
> mono main.exe
Name: Tom Age: 34
█
```

СТРУКТУРИ

Як і клас, структура може визначати конструктори. Але на відміну від класу не обов'язково викликати конструктор для створення об'єкта структури:

```
User tom;
```

Обов'язково потрібно ініціалізацію усіх полів (глобальні змінні) структури перед отриманням їх значень або перед викликом методів структури. Наприклад, в наступному випадку отримаємо помилку, так як звернення до полів і методів відбувається до присвоєння їм початкових значень:

```
1 User tom;
2 int x = tom.age;    // Ошибка
3 tom.DisplayInfo(); // Ошибка
```

Можемо використовувати для створення структури конструктор без параметрів, який є в структурі за замовчуванням і при виклику якого полів структури буде присвоєно значення за замовчуванням

```
1 User tom = new User();
2 tom.DisplayInfo(); // Name:  Age: 0
```

При використанні конструктора за замовчуванням не потрібно явно ініціалізувати поля структури. Також можна визначити свої конструктори.

На відміну від класу не можна ініціалізувати поля структури безпосередньо при їх оголошенні

```
struct User
{
    public string name = "Sam";    // ! Ошибка
    public int age = 23;           // ! Ошибка
    public void DisplayInfo()
    {
        Console.WriteLine($"Name: {name} Age: {age}");
    }
}
```


СТРУКТУРИ

```
1  using System;
2
3  namespace HelloApp
4  {
5      struct User
6      {
7          public string name;
8          public int age;
9          public User(string name, int age)
10         {
11             this.name = name;
12             this.age = age;
13         }
14         public void DisplayInfo()
15         {
16             Console.WriteLine($"Name: {name} Age: {age}");
17         }
18     }
19
20     class Program
21     {
22         static void Main(string[] args)
23         {
24             User tom = new User("Tom", 34);
25             tom.DisplayInfo();
26
27             User bob = new User();
28             bob.DisplayInfo();
29
30             Console.ReadKey();
31         }
32     }
33 }
```

```
> mcs -out:main.exe main.cs
> mono main.exe
Name: Tom Age: 34
Name:   Age: 0
> □
```

ПЕРЕРАХУВАННЯ ENUM

*Перерахування enum представляють набір логічно пов'язаних констант. Оголошення перерахування відбувається за допомогою оператора enum. Далі йде назва перерахування, після якого вказується тип перерахування - обов'язково повинно представляти цілочисельний тип (**byte**, **int**, **short**, **long**). Якщо тип явно не вказано, то за замовчуванням використовується тип **int**. Потім йде список елементів перерахування через кому:*

```
enum Days
{
    Monday,
    Tuesday,
    Wednesday,
    Thursday,
    Friday,
    Saturday,
    Sunday
}

enum Time : byte
{
    Morning,
    Afternoon,
    Evening,
    Night
}
```

```
enum Operation
{
    Add = 2,
    Subtract = 4,
    Multiply = 8,
    Divide = 16
}
```

```
enum Operation
{
    Add = 1,    // Next element + 1
    Subtract,  // 2
    Multiply,  // 3
    Divide     // 4
}
```

```
enum Color
{
    White = 1,
    Black = 2,
    Green = 2,
    Blue = White // Blue = 1
}
```

ПЕРЕРАХУВАННЯ ENUM

Кожне перерахування фактично визначає новий тип даних. У програмі можемо визначити змінну цього типу і використовувати її

```
enum Operation
{
    Sum = 1,
    Subtract,
    Multiply,
    Divide
}

class Program
{
    static void Main(string[] args)
    {
        Operation op;
        op = Operation.Sum;
        Console.WriteLine(op); // Sum

        Console.ReadLine();
    }
}
```

```
Operation op;
op = Operation.Multiply;
Console.WriteLine((int)op); // 3
```

ПЕРЕРАХУВАННЯ ENUM

Найчастіше змінна перерахування виступає в якості сховища стану, в залежності від якого виконуються деякі дії

```
class Program
{
    enum Operation
    {
        Add = 1,
        Subtract,
        Multiply,
        Divide
    }

    static void MathOp(double x, double y, Operation op)
    {
        double result = 0.0;

        switch (op)
        {
            case Operation.Add:
                result = x + y;
                break;
            case Operation.Subtract:
                result = x - y;
                break;
            case Operation.Multiply:
                result = x * y;
                break;
            case Operation.Divide:
                result = x / y;
                break;
        }
    }
}
```

```
static void Main(string[] args)
{
    // Тип операції задаємо константою Operation.Add,
    // яка дорівнює 1
    MathOp(10, 5, Operation.Add);
    // Тип операції задається константою Operation.Multiply,
    // яка дорівнює 3
    MathOp(11, 5, Operation.Multiply);

    Console.ReadLine();
}
```

ОБРОБКА ВИНЯТКІВ. КОНСТРУКЦІЯ **TRY..CATCH..FINALLY**

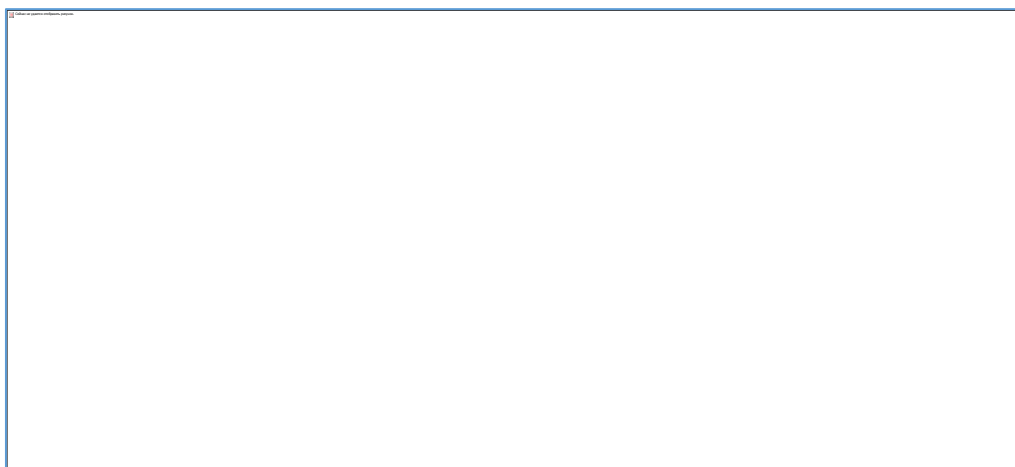
Іноді при виконанні програми виникають помилки, які важко передбачити. Такі ситуації називаються винятками. Мова C # надає розробникам можливість для обробки таких ситуацій. Для цього в C # призначена конструкція `try ... catch ... finally`

При використанні блоку **try..catch..finally** спочатку виконуються всі інструкції в блоці **try**. Якщо в цьому блоці не виникло винятків, то після його виконання починає виконуватися блок **finally**. І потім конструкція **try..catch..finally** завершує свою роботу.

Якщо ж в блоці **try** раптом виникає виняток, то звичайний порядок виконання зупиняється, і середовище CLR починає шукати блок **catch**, який може обробити цей виняток. Якщо потрібний блок **catch** знайдений, то він виконується, і після його завершення виконується блок **finally**.

Якщо потрібний блок **catch** не знайдений, то при виникненні виключення програма аварійно завершує своє виконання.

ОБРОБКА ВИНЯТКІВ. КОНСТРУКЦІЯ **TRY..CATCH..FINALLY**



```
using System;

namespace HelloApp
{
    class Program
    {
        static void Main(string[] args)
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
            Console.WriteLine("Конец программы");
            Console.Read();
        }
    }
}
```

Exception Unhandled

System.DivideByZeroException: 'Попытка деления на нуль.'

[View Details](#) [Copy Details](#)

Exception Settings

посмотреть информацию об исключении

ОБРОБКА ВИНЯТКІВ. КОНСТРУКЦІЯ **TRY..CATCH..FINALLY**

```
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
        catch
        {
            Console.WriteLine("Виникло виключення!");
        }
        finally
        {
            Console.WriteLine("Блок finally");
        }
        Console.WriteLine("Кінець програми");
        Console.Read();
    }
}
```

Виникло виключення!

Блок finally

Кінець програми

Обробка винятків і умовні конструкції

Ряд виняткових ситуацій може бути передбачений розробником. Якщо користувач введе не число, а рядок, якісь інші символи, то програма випаде в помилку. Однак набагато оптимальніше було б перевірити допустимість перетворення:

```
static void Main(string[] args)
{
    Console.WriteLine("Введіть число");
    int x;
    string input = Console.ReadLine();
    if (Int32.TryParse(input, out x))
    {
        x *= x;
        Console.WriteLine("Квадрат числа: " + x);
    }
    else
    {
        Console.WriteLine("Некоректний ввід");
    }
    Console.Read();
}
```

Блок catch і фільтри винятків

```
catch
{
    // інструкції
}
```

```
catch (ТИП_ВИКЛЮЧЕННЯ)
{
    // інструкції
}
```

```
int x, y;
try {
    x = Convert.ToInt32(Console.Read());
    y = Convert.ToInt32(Console.Read());
    Console.WriteLine(x / y);
}
catch (DivideByZeroException e) {
    Console.WriteLine("Cannot divide by 0");
}
catch (Exception e) {
    Console.WriteLine("An error occurred");
}
```

Типи виключень. Клас Exception

InnerException: зберігає інформацію про виключення, яке послужило причиною поточного виключення

Messenger: зберігає повідомлення про виключення, текст помилки

Source: зберігає ім'я об'єкта або збірки, яке викликало виключення

StackTrace: повертає строкове представлення стека викликаючи, які привели до виникнення виключення

TargetSite: повертає метод, в якому і було викликано виключення

```
static void Main(string[] args)
{
    try
    {
        int x = 5;
        int y = x / 0;
        Console.WriteLine($"Результат: {y}");
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Виключення: {ex.Message}");
        Console.WriteLine($"Метод: {ex.TargetSite}");
        Console.WriteLine($"Трасування стека: {ex.StackTrace}");
    }

    Console.Read();
}
```


Типи виключень. Клас Exception

Спеціалізовані типи винятків

```
static void Main(string[] args)
{
    try
    {
        int[] numbers = new int[4];
        numbers[7] = 9; // IndexOutOfRangeException

        int x = 5;
        int y = x / 0; // DivideByZeroException
        Console.WriteLine($"Результат: {y}");
    }
    catch (DivideByZeroException)
    {
        Console.WriteLine("Виникло виключення DivideByZeroException");
    }
    catch (IndexOutOfRangeException ex)
    {
        Console.WriteLine(ex.Message);
    }
    catch (Exception ex)
    {
        Console.WriteLine($"Виключення: {ex.Message}");
    }

    Console.Read();
}
```

DivideByZeroException : представляє виняток, який генерується при діленні на нуль

ArgumentOutOfRangeException: генерується, якщо значення аргументу виходить за допустимі допустимих значень

ArgumentException : генерується, якщо в метод для параметра передається некоректне значення

IndexOutOfRangeException: генерується, якщо індекс елемента масиву або колекції виходить за допустимі допустимих значень

InvalidCastException : генерується при спробі провести неприпустимі перетворення типів

NullReferenceException: генерується при спробі звернення до об'єкта, який дорівнює null (тобто по суті невизначений)

ГЕНЕРАЦІЯ ВИКЛЮЧЕННЯ І ОПЕРАТОР **THROW**

Зазвичай система сама генерує виключення за певних ситуацій, наприклад, при діленні числа на нуль. Але мова C # також дозволяє генерувати виключення вручну за допомогою оператора **throw**. За допомогою цього оператора можемо створити виняток і викликати його в процесі виконання.

Наприклад, в нашій програмі відбувається введення рядка, і якщо довжина рядка буде більше 6 символів, виникає виняток:

```
static void Main(string[] args)
{
    try
    {
        Console.Write("Введіть рядок: ");
        string message = Console.ReadLine();
        if (message.Length > 6)
        {
            throw new Exception("Довжина рядка більше 6 символів");
        }
    }
    catch (Exception e)
    {
        Console.WriteLine($"Помилка: {e.Message}");
    }
    Console.Read();
}
```

РОБОТА З ФАЙЛАМИ

У просторі імен **System.IO** є різні класи, які використовуються для виконання численних операцій з файлами, таких як створення та видалення файлів, читання або запис у файл, закриття файлу тощо. Клас **File** - один із них.

```
string str = "Some text";  
File.WriteAllText("test.txt", str);
```

WriteAllText () метод створює файл із зазначеним шляхом і записує в нього вміст. Якщо файл вже існує, він буде перезаписаний. Для читання вмісту файлу, використовується метод **ReadAllText ()**

```
string txt = File.ReadAllText("test.txt");  
Console.WriteLine(txt);
```

AppendAllText () - додає текст до кінця файлу.

Create () - створює файл у вказаному місці.

Delete () - видаляє вказаний файл.

Exists () - визначає, чи існує вказаний файл.

Copy () - копіює файл у нове місце.

Move () - переміщує вказаний файл на нове місце.

Усі методи автоматично закривають файл після виконання операції.