

Лекція 8. Інтерфейси. Структури, перерахування, винятки, робота з файлами

1. Інтерфейси

1.1 Поняття інтерфейсу

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**.

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

Ім'я_інтерфейсу – зазвичай починається з великої літери **I** (використання великої літери **I** на початку назви інтерфейсу не є обов'язковим, проте – бажане, для полегшення розуміння коду).

Основне застосування інтерфейсів – полегшити синхронізацію програмного коду написаного різними розробниками. Знову розглянемо приклад з графічними об'єктами. 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);
    }

```

Тепер успадкуємо від цього 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();
    }
}

```

Перевага інтерфейсі над класами, це можливість успадковувати один клас від кількох інтерфейсів. Наприклад опишемо інтерфейс для обертання `IRotated`:

```
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();
    }
    //
    public void Rotate(int x0, int y0, double angle) {
        // метод що описує обертання
        // навколо заданого центру
        // ...
    }
}
```

Зауваження. Слід зазначити, що в C# клас в списку предків може мати лише один клас та декілька інтерфейсів перерахованих через кому.

1.2 Інтерфейс IComparable

Ще одна перевага інтерфейсів, це можливість за допомогою підтримки стандартних інтерфейсів середовища .Net користатися різноманітними методами стандартних об'єктів. Наприклад, щоб сортувати ваші 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, нехай:

- 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**.

Розглянемо відповідну програму:

```
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();
    }
}

```

Вона виведе на екран, точки у відсортованому вигляді:

```

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

```

1.3 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**.

Наприклад:

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

Дасть ствердну відповідь.

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

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

Виведе:

Не підтримує IComparable

2. Структури

Поряд з класами структури представляють ще один спосіб створення власних типів даних в C#. Більш того багато примітивних типи, наприклад, **int**, **double** і т.д., по суті є структурами.

Наприклад, визначимо структуру, яка представляє людину:

```
struct User  
{  
    public string name;  
    public int age;  
  
    public void DisplayInfo()  
    {  
        Console.WriteLine($"Name: {name} Age: {age}");  
    }  
}
```

```
}
```

Як і класи, структури можуть зберігати стан у вигляді змінних і визначати поведінку у вигляді методів. Так, в даному випадку визначено дві змінні - **name** і **age** для зберігання відповідно імені і віку людини і метод **DisplayInfo** для виведення інформації про людину.

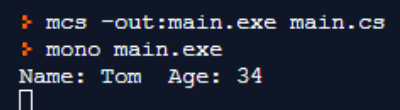
Використовуємо цю структуру в програмі:

```
class Program
{
    static void Main(string[] args)
    {
        User tom;
        tom.name = "Tom";
        tom.age = 34;
        tom.DisplayInfo();

        Console.ReadKey();
    }
}
```

В даному випадку створюється об'єкт **tom**. У нього встановлюються значення глобальних змінних, і потім виводиться інформація про нього.

```
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
█
```

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

```
User tom;
```

Однак якщо таким чином створюємо об'єкт структури, то обов'язково треба проініціалізувати усі поля (глобальні змінні) структури перед отриманням їх значень або перед викликом методів структури. Тобто, 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
```

Також 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 (наприклад, для числових типів це число 0):

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

При використанні конструктора за замовчуванням нам не треба явно ініціалізувати поля структури. Також можна визначити свої конструктори. Наприклад, змінимо структуру User:

```
using System;
```

```
namespace HelloApp
{
    struct User
    {
        public string name;
        public int age;
        public User(string name, int age)
        {
            this.name = name;
            this.age = age;
        }
        public void DisplayInfo()
        {
            Console.WriteLine($"Name: {name} Age: {age}");
        }
    }

    class Program
    {
        static void Main(string[] args)
        {
            User tom = new User("Tom", 34);
            tom.DisplayInfo();
        }
    }
}
```



```

        User bob = new User();
        bob.DisplayInfo();

        Console.ReadKey();
    }
}

```

```

Name: Tom Age: 34
Name:    Age: 0

```

На відміну від класу не можна ініціалізувати поля структури безпосередньо при їх оголошенні, наприклад, так:

```

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

```

Структури не підтримують успадкування і не можуть містити віртуальні методи.

3. 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 - він обов'язково повинен 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
}

```

У цих прикладах кожному елементу перерахування присвоюється цілочисельне значення, причому перший елемент буде мати значення 0, другий - 1 і так далі. Можемо явно вказати значення елементів, або вказати значення першого елемента:

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

Але можна і для всіх елементів явно вказати значення. Можна привласнювати однієї константі значення іншої константи:

```
enum Operation          enum Color
{
    Add = 2,             {
    Subtract = 4,         White = 1,
    Multiply = 8,         Black = 2,
    Divide = 16           Green = 2,
                        Blue = White // Blue = 1
    }
}
```

Кожне перерахування фактично визначає новий тип даних. Потім в 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();
    }
}
```

Незважаючи на те, що кожна константа зіставляється з певним числом, не можна присвоїти їй числове значення, наприклад, **Operation op = 1;** Якщо будемо виводити на консоль значення цієї змінної, то отримаємо ім'я константи, а не числове значення. Якщо ж необхідно отримати числове значення, то слід виконати приведення до числового типу:

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

Також варто відзначити, що перерахування необов'язково визначати всередині класу, можна і поза класом, але в межах простору імен. Найчастіше змінна перерахування виступає в якості сховища стану, в залежності від якого виконуються деякі дії. Так, розглянемо застосування перерахування на більш реальному прикладі:

```
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;  
        }  
  
        Console.WriteLine("Результат операції дорівнює {0}", result);  
    }  
}
```

```

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();
}
}

```

4 Обробка винятків. Конструкція try..catch..finally

Іноді при виконанні програми виникають помилки, які важко передбачити. Наприклад, при передачі файлу по мережі може несподівано обірватися підключення до мережі. Такі ситуації називаються винятками. Мова C# надає розробникам можливості для обробки таких ситуацій. Для цього в C# призначена конструкція try ... catch ... finally.

```

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**.

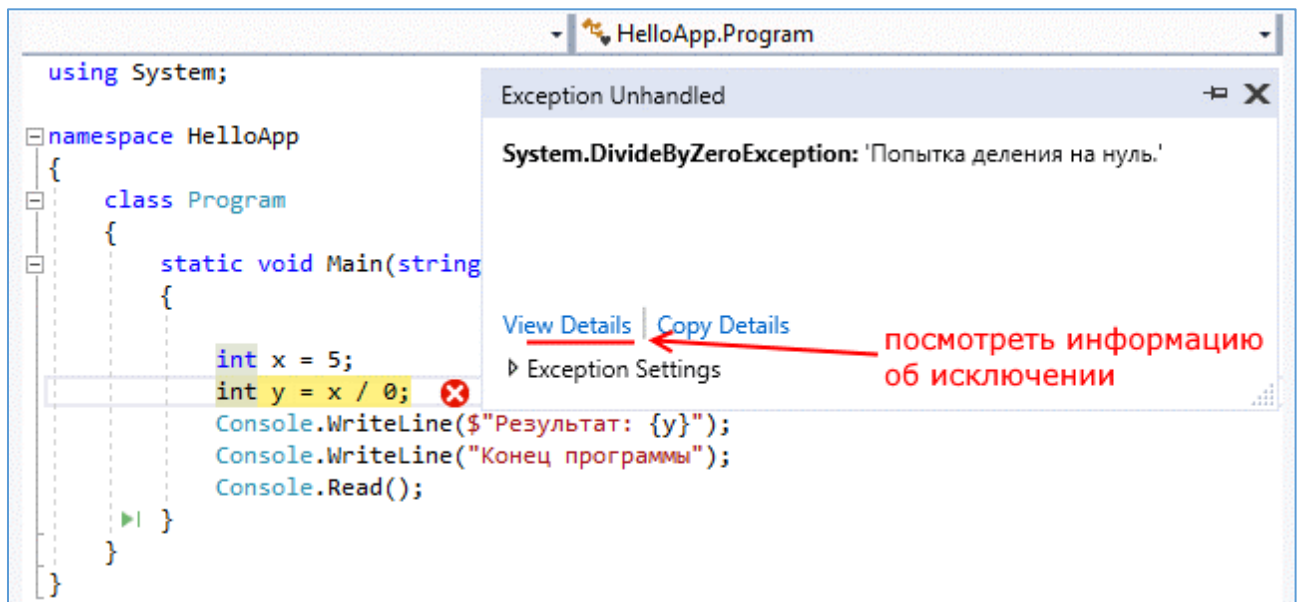
Якщо потрібний блок **catch** не знайдений, то при виникненні виключення програма аварійно завершує своє виконання. Розглянемо наступний приклад:

```

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

```

У даному випадку відбувається розподіл числа на 0, що призведе до генерації виключення. І під час запуску програми в режимі налагодження ми побачимо в Visual Studio вікно, яке інформує про виключення:



У цьому вікні побачимо, що виникло виключення, яке представляє тип `System.DivideByZeroException`, тобто спроба ділення на нуль. За допомогою пункту `View Details` можна подивитися більш детальну інформацію про виключення. Щоб уникнути подібного аварійного завершення програми, слід використовувати для обробки винятків конструкцію `try ... catch ... finally`. Так, перепишемо приклад наступним чином:

```

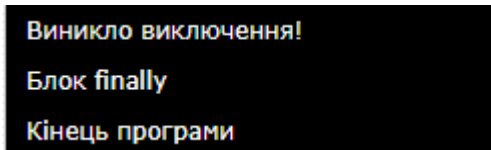
class Program
{
    static void Main(string[] args)
    {
        try
        {
            int x = 5;
            int y = x / 0;
            Console.WriteLine($"Результат: {y}");
        }
    }
}

```

```

        catch
        {
            Console.WriteLine("Виникло виключення!");
        }
        finally
        {
            Console.WriteLine("Блок finally");
        }
        Console.WriteLine("Кінець програми");
        Console.Read();
    }
}

```



```

Виникло виключення!
Блок finally
Кінець програми

```

4.1 Обробка винятків і умовні конструкції

Ряд виняткових ситуацій може бути передбачений розробником. Наприклад, нехай програма передбачає введення числа і виведення його квадрата:

```

static void Main(string[] args)
{
    Console.WriteLine("Введіть число");
    int x = Int32.Parse(Console.ReadLine());

    x *= x;
    Console.WriteLine("Квадрат числа: " + x);
    Console.Read();
}

```

Якщо користувач введе не число, а рядок, якісь інші символи, то програма випаде в помилку. З одного боку, тут якраз та ситуація, коли можна застосувати блок try..catch, щоб обробити можливу 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();
}

```

Метод `Int32.TryParse()` повертає `true`, якщо перетворення можна здійснити, і `false` - якщо не можна. При допустимості перетворення змінна `x` буде містити введене число. Так, не використовуючи `try...catch` можна обробити можливу виняткову ситуацію.

З точки зору продуктивності використання блоків `try..catch` більш накладно, ніж застосування умовних конструкцій. Тому по можливості замість `try..catch` краще використовувати умовні конструкції на перевірку виняткових ситуацій.

4.2 Блок `catch` і фільтри винятків

За обробку виключення відповідає блок `catch`, який може мати такі форми:

<pre> catch { // інструкції } </pre>	<pre> catch (тип_виключення) { // інструкції } </pre>
--	---

Обробляє будь-який виняток, яке виникло в блоці `try`. Обробляються тільки ті винятки, які відповідають типу, зазначеному в дужках після оператора `catch`.

```

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");
}

```

4.3 Типи виключень. Клас `Exception`

Базовим для всіх типів винятків є тип `Exception`. Цей тип визначає ряд властивостей, за допомогою яких можна отримати інформацію про виключення.

- **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
- Наприклад, опрацюємо виключення типу Exception:

```
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();
}
```

Однак так як тип **Exception** є базовим типом для всіх винятків, то вираз **catch (Exception ex)** буде обробляти всі винятки, які можуть виникнути.

Але також є більш спеціалізовані типи винятків, які призначені для обробки якихось певних видів винятків. Їх досить багато, лише деякі:

- **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)

І при необхідності ми можемо розмежувати обробку різних типів винятків, включивши додаткові блоки catch:

```
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();
}
```

В даному випадку блок catch (Exception ex){} буде обробляти всі винятки крім DivideByZeroException і IndexOutOfRangeException. При цьому блоки catch для більш загальних, найбільш фундаментальних винятків слід поміщати в кінці - після блоків catch для більш конкретних, спеціалізованих типів.

4.4 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**. Тобто 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();
}

```

5 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. Наприклад:

```

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. Код вище створює файл **test.txt** та записує в нього вміст рядка **str**. Щоб використовувати клас File, необхідно використовувати **namespace: using System.IO**.

Для читання вмісту файлу, використовується метод ReadAllText () класу File:

```

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

```

У класі File доступні такі методи:

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.