Лабораторна робота №12. Робота з файлами

Мета роботи: Навчитися працювати з файлами та директоріями у C#.

Теоретичні відомості

У середовищі .NET робота з файлами здійснюється за допомогою класів із простору імен System.IO. Вони дозволяють створювати, зчитувати, змінювати та видаляти файли. Для роботи з простими текстовими файлами найчастіше використовують методи класу File. Наприклад, для збереження одного рядка в окремий файл достатньо одного рядка коду:

```
File.WriteAllText("output.txt", "Це приклад тексту.");
```

Щоб прочитати вміст цього файлу назад у програму, можна скористатися методом ReadAllText:

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

У випадку, коли потрібно зберегти кілька рядків — наприклад, колекцію значень — доцільно використовувати метод WriteAllLines, який записує одразу весь масив рядків:

```
string[] lines = { "рядок 1", "рядок 2", "рядок 3" };
File.WriteAllLines("list.txt", lines);
```

Якщо потрібен контроль над процесом зчитування або запису (рядок за рядком), використовуються класи StreamReader та StreamWriter. У такому випадку рекомендується застосовувати оператор using, який гарантує автоматичне закриття файлу після завершення роботи з ним, навіть якщо виникла помилка.

```
using (StreamWriter writer = new StreamWriter("log.txt", append: true))
{
    writer.WriteLine("Запис з логуванням: " + DateTime.Now);
}
```

using створює блок, в межах якого ресурс є доступним, і після виходу з нього автоматично викликається метод Dispose(). Це запобігає витоку ресурсів, таких як незакриті файлові потоки. Починаючи з C# 8.0, також можна використовувати скорочений синтаксис:

```
using var reader = new StreamReader("data.txt");
string content = reader.ReadToEnd();
```

Часто в практиці використовують CSV-файли — формат зберігання табличних даних, де кожен рядок — це запис, а значення відокремлюються символом-комою, крапкою з комою або іншим роздільником. Наприклад, список студентів можна зберегти так:

```
var students = new List<string> {
    "Ім’я,Прізвище,Група",
    "Марія,Коваль,КБ-21-1",
    "Іван,Діденко,КБ-21-2"
};
```

```
File.WriteAllLines("students.csv", students);
```

Щоб опрацювати такі дані після зчитування, рядки розділяють методом Split:

```
var lines = File.ReadAllLines("students.csv");
foreach (var line in lines.Skip(1)) // пропускаємо заголовок
{
    var fields = line.Split(',');
    Console.WriteLine($"Ім’я: {fields[0]}, Група: {fields[2]}");
}
```

У складніших випадках, коли потрібно зберігати об’єкти зі структурою (наприклад, записи з кількома властивостями), краще застосовувати формат JSON. Його зручно використовувати разом із класом JsonSerializer, який доступний у просторі імен System.Text.Json. Наприклад, серіалізація об’єкта:

```
var person = new { Name = "Сергій", Age = 20 };
string json = JsonSerializer.Serialize(person);
File.WriteAllText("person.json", json);
```

Повернути об’єкт назад із файлу можна за допомогою десеріалізації:

```
string json = File.ReadAllText("person.json");
var result = JsonSerializer.Deserialize<Dictionary<string, object>>(json);
Console.WriteLine($"Ім’я: {result["Name"]}, Вік: {result["Age"]}");
```

У C# під час роботи з файлами можливі різноманітні помилки — файл може бути заблокований іншою програмою, не існувати або мати неправильний формат. Щоб уникнути аварійного завершення програми, всі файлові операції потрібно виконувати з використанням конструкції try-catch. Наприклад, спроба зчитати неіснуючий файл:

```
try
{
    string content = File.ReadAllText("info.txt");
    Console.WriteLine(content);
}
catch (FileNotFoundException ex)
{
    Console.WriteLine("Файл не знайдено.");
}
catch (IOException ex)
{
    Console.WriteLine("Виникла помилка при доступі до файлу: " + ex.Message);
}
```

У цьому прикладі використовується декілька гілок catch для різних типів винятків. FileNotFoundException ловить ситуації, коли файл відсутній, а IOException — всі інші помилки доступу до файлів (наприклад, порушення прав доступу чи відсутність диска). Так само обробляються помилки при записі у файл:

```
try
{
    File.WriteAllText("output.txt", "Збережені дані");
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Немає прав на запис у файл.");
}
catch (Exception ex)
```

```
{
    Console.WriteLine("Помилка: " + ex.Message);
}
```

Крім обробки винятків, варто перевіряти наявність файлу або директорії перед операціями. Це допоможе уникнути зайвих викликів catch:

```
if (File.Exists("data.txt"))
{
    string text = File.ReadAllText("data.txt");
}
else
{
    Console.WriteLine("Файл не знайдено.");
}
```

Обробка помилок — обов'язковий етап розробки, який забезпечує стабільну роботу програми та дозволяє уникати непередбачуваних збоїв.

Окрім роботи з файлами, платформа .NET надає можливості для керування папками (директоріями) за допомогою класу Directory із простору імен System.IO. Наприклад, щоб перевірити, чи існує папка, можна скористатися методом Exists:

```
if (!Directory.Exists("Data"))
{
    Directory.CreateDirectory("Data");
}
```

Якщо потрібно створити тимчасову директорію або згенерувати унікальний шлях до файлу, зручно використовувати Path.Combine, Path.GetTempPath, Path.GetRandomFileName:

```
string fullPath = Path.Combine("Logs", "session_" + Path.GetRandomFileName() +
".txt");
```

Крім базових операцій зчитування та запису, у C# також є можливість відслідковувати зміни у файловій системі в реальному часі. Це здійснюється за допомогою класу FileSystemWatcher, який дозволяє реагувати на створення, зміну, перейменування чи видалення файлів і папок у заданому каталозі. Наприклад, створимо об'єкт, який відстежує всі зміни в папці data:

```
using System.IO;

var watcher = new FileSystemWatcher();
watcher.Path = @"C:\MyLab\data";
watcher.Filter = "*.json"; // відслідковуємо лише JSON-файли
watcher.EnableRaisingEvents = true;
```

Щоб відреагувати на певну подію (наприклад, створення нового файлу), додаємо обробник події:

```
watcher.Created += (sender, e) =>
{
    Console.WriteLine($"Новий файл: {e.FullPath}");
};
```

Методи/властивості для роботи з файлами та папками:

Клас	**Метод/Властивість**	**Призначення**
File	WriteAllText(path, text)	Записує рядок тексту у файл

File	ReadAllText(path)	Зчитує весь вміст файлу у рядок
File	WriteAllLines(path, lines)	Записує масив рядків у файл
File	ReadAllLines(path)	Зчитує файл у масив рядків
File	Exists(path)	Перевіряє наявність файлу
File	Delete(path)	Видаляє вказаний файл
Directory	CreateDirectory(path)	Створює папку (включно з вкладеними)
Directory	Exists(path)	Перевіряє наявність директорії
Directory	GetFiles(path)	Повертає масив шляхів до файлів у вказаній папці
Directory	GetDirectories(path)	Повертає масив підкаталогів
Directory	Delete(path, recursive)	Видаляє папку, якщо recursive=true — з усім вмістом
Path	Combine(parts…)	Об'єднує частини шляху
Path	GetFileName(path)	Повертає ім'я файлу з повного шляху
Path	GetExtension(path)	Повертає розширення файлу
JsonSerializer	Serialize(obj)	Перетворює об'єкт у JSON
JsonSerializer	Deserialize<T>(json)	Створює об'єкт з JSON
FileSystemWatcher	Created / Changed / Deleted	Події при зміні файлів у папці
FileSystemWatcher	EnableRaisingEvents	Увімкнення або вимкнення прослуховування подій

Зміст роботи

Згідно з обраним у лабораторній роботі 5 варіантом теми, необхідно виконати наступну дії. Всі дії відбуваються в репозиторії OOPWPFProject.

Завдання 1: Злити гілку feature/add- ovverride в основну гілку (master або main). Створити нові гілку

```
git checkout -b feature/add-file
```

Завдання 2: Реалізувати створення директорії Data під час запуску програми. Необхідно додати перевірку на те, чи існує директорія і створювати її тільки у випадку якщо директорії немає. Всі файли, що будуть зберігаються та зчитуються під час виконання програми, повинні бути розміщені в цій папці, а не в корені проєкту.

Завдання 3: Реалізувати збереження даних у файл перед закриттям програми. Дані повинні автоматично записуватись у відповідний файл (TXT, CSV або JSON) при закритті вікна. У разі виникнення помилки виводити повідомлення про помилку збереження.

Завдання 4: Реалізувати можливість зчитування даних із файлу при запуску програми. У разі виникнення помилки виводити повідомлення про помилку збереження.

Завдання 5: Реалізувати збереження окремого файлу, який міститиме лише частину даних, відібраних за логічною умовою відповідно до тематики проєкту. Умова фільтрації визначається змістом теми (наприклад, майбутні події, замовлення на велику суму, кіберспортивні турніри по певній грі, тощо). Збереження даних у цей файл необхідно реалізувати через інтерфейс

користувача (наприклад при натисканні на кнопку).

Завдання 6: Потрібно реалізувати запис дій користувача, пов'язаних із обробкою даних, у лог-файл. Логування повинно здійснюватися у текстовий файл (наприклад, log.txt) і містити запис про кожну з таких дій:

- додавання запису;
- редагування запису;
- видалення запису;
- збереження даних у файл;

Для роботи з логуванням потрібно створити окремий клас Logger. Формат рядку для логів:

```
[2025-01-23 14:16:00] Дія (Додано/Видалено/Змінено/Збережено): Опис (що було додано/змінено/видалено)
```

Контрольні запитання

1. Які методи класу File використовуються для зчитування та запису текстових файлів?
2. Як перевірити, чи існує файл або папка перед спробою доступу до них?
3. Для чого використовується метод Path.Combine()? Наведіть приклад.
4. Як реалізувати обробку помилок при роботі з файлами? Які типи винятків можуть виникати?
5. Що таке Directory.CreateDirectory() і чим він відрізняється від Directory.Exists()?
6. Для чого використовується клас FileSystemWatcher і які події він дозволяє відслідковувати?
7. Чим відрізняється File.AppendAllText() від File.WriteAllText()?
8. Як правильно обробити ситуацію, коли файл для зчитування не існує?
9. У чому різниця між StreamReader / StreamWriter та методами класу File?
10. Як за допомогою using забезпечити автоматичне закриття файлового потоку?