C# Developer.

Работа с файлами

otus.ru

Проверить, идет ли запись

Меня хорошо видно && слышно?

Ставим “+”, если все хорошо
“-”, если есть проблемы

Тема вебинара

Работа с файлами

Елена Сычева

Team Lead Full-Stack Developer

Об опыте:

Более 15 лет опыта работы разработчиком (C#, Angular, .Net, React, NodeJs)

Телефон / эл. почта / соц. сети:
https://t.me/lentsych

Правила вебинара

Активно
участвуем

Задаем вопрос
в чат или голосом

Вопросы вижу в чате,
могу ответить не сразу

Условные обозначения

Индивидуально

Время, необходимое
на активность

Пишем в чат

Говорим голосом

Документ

Ответьте себе или
задайте вопрос

Маршрут вебинара

- Что это? Зачем это?
- Базовые операции
- Работа с потоками
- Асинхронные операции
- Практика
- Рефлексия

Цели вебинара

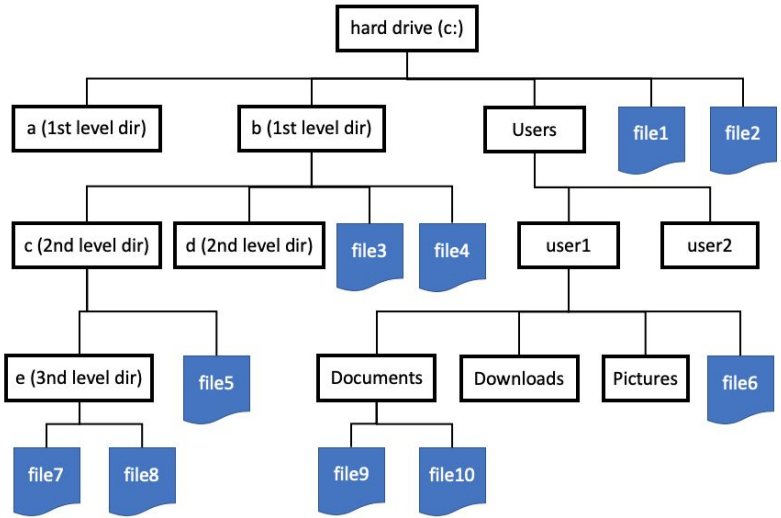
К концу занятия вы сможете

1. Научиться работать с файлами и каталогами в своих приложениях
3. Понимать как работают асинхронные операции с файлами

Смысл

Зачем вам это уметь

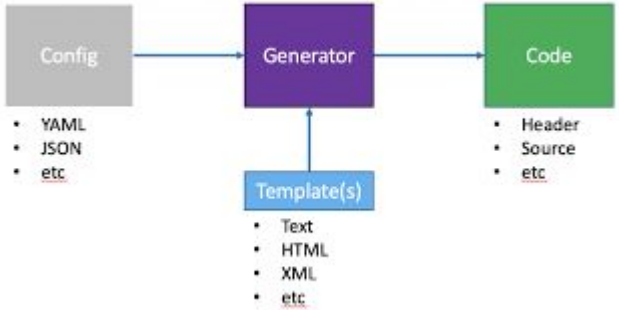
1. Освоить инструмент для работы с файлами и директориями
2. Понимать что делает встречающийся код



Работа с файлами. Зачем?

Что такое обработка файлов?

Определение: Обработка файлов относится к чтению и записи данных в файлы в файловой системе.



Важность:

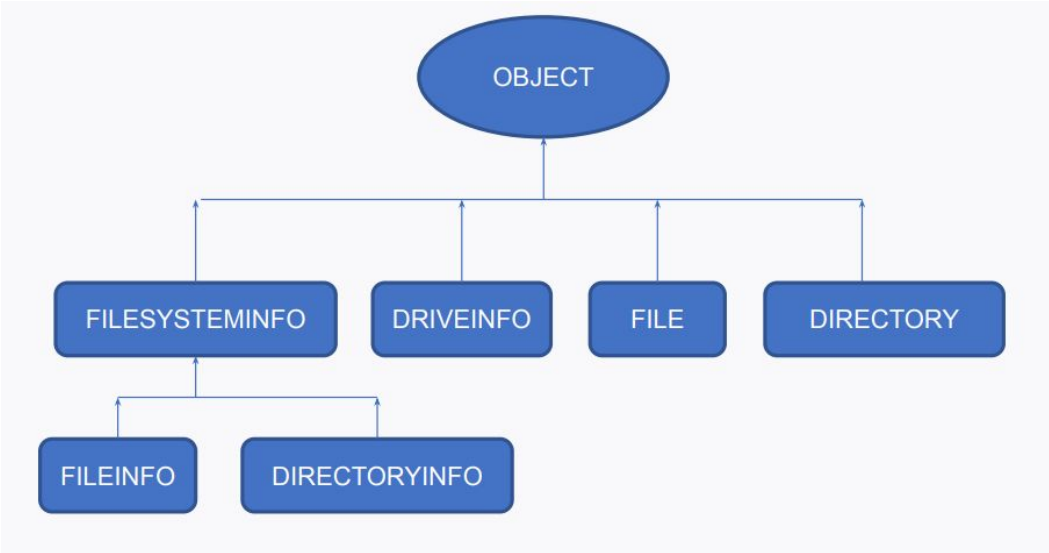
- Постоянное хранение данных между сеансами.
- Чтение файлов конфигурации или журналов.
- Запись выходных данных для последующей обработки или анализа.
- Генерация других файлов
- Межсистемная интеграция
- Формирование отчетов

Обработка файлов в C#

C# использует пространство имен System.IO для файловых операций.

Обычные файловые операции включают:

- Создание файлов
- Чтение файлов
- Запись в файлы
- Удаление файлов



Пространство имен System.IO

- Класс File: Предоставляет статические методы для операций с файлами (создание, чтение, запись).
- Класс Directory: Предоставляет статические методы для операций с каталогами (создание, удаление, перемещение).

Другие классы:

- FileInfo, DirectoryInfo (предоставляют методы экземпляра для расширенных операций).
- StreamReader и StreamWriter для работы с потоками файлов.

Основные операции с файлами и каталогами в C#

Создание и запись файлов

```
// Create a file
string filePath = "example.txt";
File.Create(filePath).Dispose(); // Dispose to release file handle

// Write to the file
File.WriteAllText(filePath, "Hello, C#!");
```

Проверка существования файла

```
if (File.Exists("example.txt"))
{
    Console.WriteLine("File exists!");
}
```

Чтение из файлов

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

Перемещение и копирование файлов

```
File.Move("example.txt", "newDirectory/example.txt");
```

```
File.Copy("example.txt", "example_copy.txt");
```

Удаление файлов

```
File.Delete("example.txt");
```

Directories

Создание и проверка

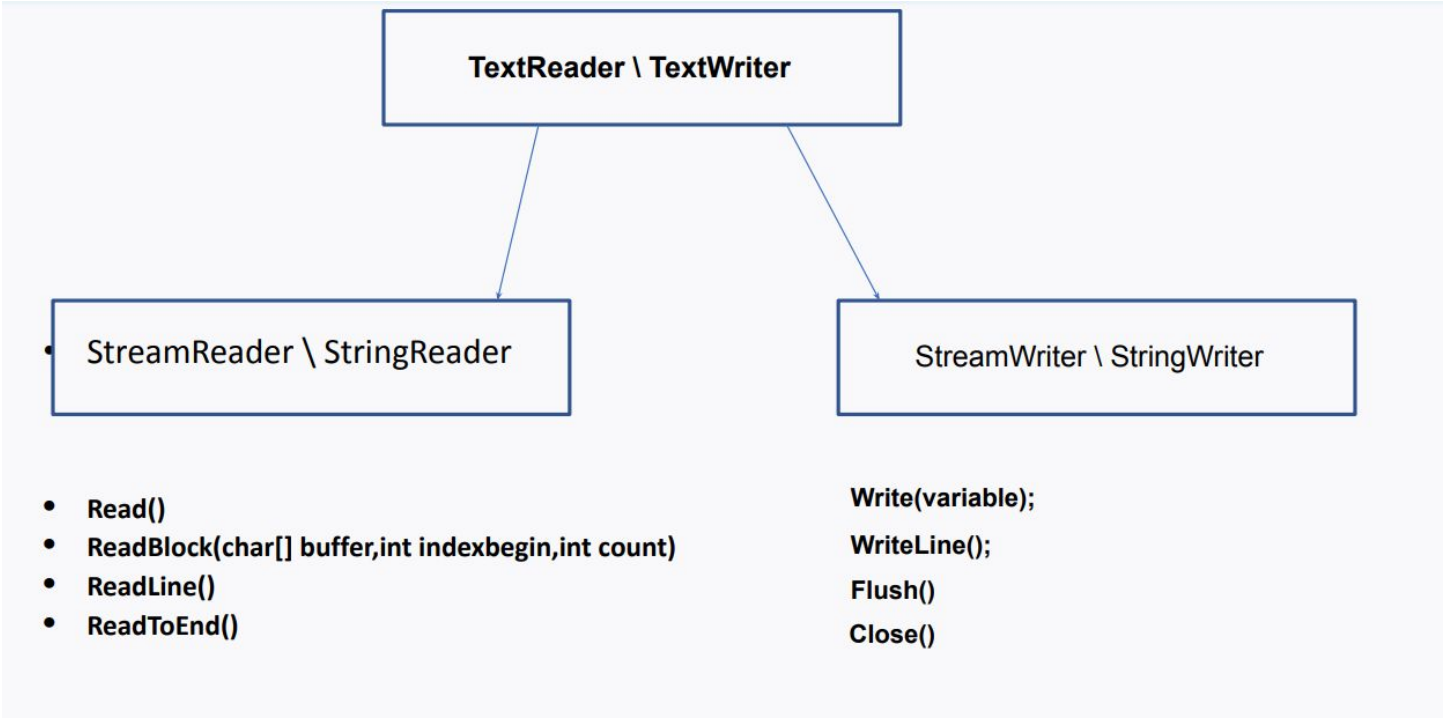
```
Directory.CreateDirectory("NewFolder");
```

```
if (Directory.Exists("NewFolder"))
{
    Console.WriteLine("Directory exists!");
}
```

Список файлов в каталоге

```
string[] files = Directory.GetFiles("NewFolder");
foreach (string file in files)
{
    Console.WriteLine(file);
}
```

TEXTREADER /TEXTWRITER и его наследники



- Read()
- ReadBlock(char[] buffer,int indexbegin,int count)
- ReadLine()
- ReadToEnd()

Write(variable);

WriteLine();

Flush()

Close()

Stream

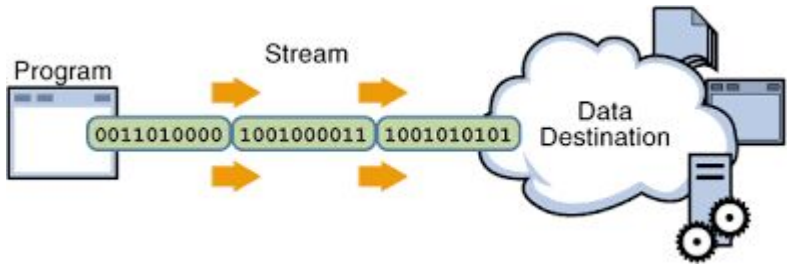
Что такое поток (Stream)?

Поток — это последовательность данных.

Потоки позволяют выполнять чтение и запись в источник данных (например, файл) небольшими порциями.

Два типа потоков:

- Входные потоки (для чтения).
- Выходные потоки (для записи).

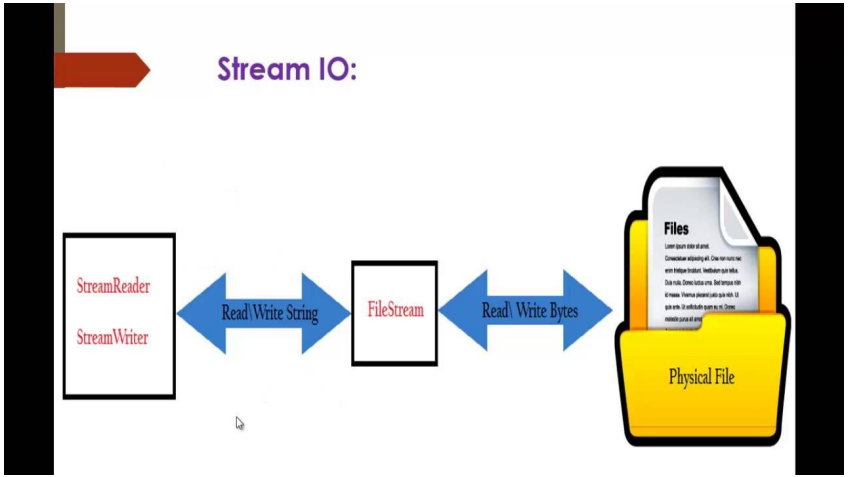


Класс System.IO.Stream

Базовым классом для всех типов потоков в C# является Stream.

Обычно используемые потоки:

- FileStream: для чтения/записи байтов в/из файлов.
- MemoryStream: для работы с данными в памяти.
- NetworkStream: для сетевого взаимодействия.



Запись в файлы с использованием FileStream

```
using (FileStream fs = new FileStream("data.txt", FileMode.Create))
{
    byte[] data = new UTF8Encoding(true).GetBytes("Hello, Stream!");
    fs.Write(data, 0, data.Length);
}
```

Чтение из файлов с использованием FileStream

```
using (FileStream fs = new FileStream("data.txt", FileMode.Open))
{
    byte[] data = new byte[fs.Length];
    fs.Read(data, 0, data.Length);
    string text = new UTF8Encoding(true).GetString(data);
    Console.WriteLine(text);
}
```

Использование StreamWriter для текстовых файлов

```
using (StreamWriter writer = new StreamWriter("example.txt"))
{
    writer.WriteLine("This is a text file.");
}
```

Чтение текстовых файлов с помощью StreamReader

```
using (StreamReader reader = new StreamReader("example.txt"))
{
    string content = reader.ReadToEnd();
    Console.WriteLine(content);
}
```

Обработка исключений при файловом вводе-выводе

Зачем обрабатывать исключения в файловых операциях?

- Файлы могут не существовать.
- Проблемы с разрешениями.
- Ошибки диска.

Ввод-вывод двоичных файлов

Зачем использовать двоичные файлы?

- Для нетекстовых данных, таких как изображения, видео или сериализованные объекты.

```
using (FileStream fs = new FileStream("binarydata.dat", FileMode.Create))
{
    BinaryWriter writer = new BinaryWriter(fs);
    writer.Write(1234);  // Writing an integer
    writer.Write(99.99); // Writing a double
    writer.Close();
}
```

Чтение двоичных файлов

```
using (FileStream fs = new FileStream("binarydata.dat", FileMode.Open))
{
    BinaryReader reader = new BinaryReader(fs);
    int intValue = reader.ReadInt32();
    double doubleValue = reader.ReadDouble();
    Console.WriteLine($"Integer: {intValue}, Double: {doubleValue}");
}
```

Пример

Итого

- Потоки обеспечивают гибкий доступ к файлам.
- FileStream позволяет выполнять операции с файлами на уровне байтов.
- StreamReader и StreamWriter упрощают обработку текстовых файлов.
- Обработка исключений обеспечивает стабильные операции с файлами.
- Используйте BinaryWriter/BinaryReader для нетекстовых данных.

Асинхронная обработка файлов в C#

Зачем использовать асинхронную обработку файлов?

синхронные файловые операции: программа ожидает завершения каждой файловой операции перед продолжением.

Асинхронные файловые операции: программа может выполнять другие задачи, ожидая завершения файловых операций.

Преимущества:

- Улучшенная отзывчивость: идеально подходит для приложений пользовательского интерфейса.
- Эффективное использование ресурсов: сокращает время простоя при ожидании ввода-вывода.

Ключевые слова async и await

- **async**: Помечает метод как асинхронный.
- **await**: Приостанавливает выполнение метода до тех пор, пока ожидаемая задача не завершится.

```
public async Task ReadFileAsync()
{
    string content = await File.ReadAllTextAsync("example.txt");
    Console.WriteLine(content);
}
```

Асинхронные методы в пространстве имен System.IO

Асинхронное чтение:

- File.ReadAllTextAsync()
- File.ReadAllLinesAsync()
- File.ReadAllBytesAsync()

Асинхронная запись:

- File.WriteAllTextAsync()
- File.WriteAllLinesAsync()
- File.WriteAllBytesAsync()

Асинхронная запись файлов

```
public async Task WriteToFileAsync()
{
    string content = "Hello, async file handling!";
    await File.WriteAllTextAsync("example.txt", content);
    Console.WriteLine("File written successfully.");
}
```

Асинхронная обработка больших файлов

Для больших файлов использование ReadAllTextAsync или WriteAllTextAsync может по-прежнему потреблять значительную часть памяти.

Решение:
асинхронные операции на основе потоков:

FileStream.ReadAsync()

FileStream.WriteAsync()

Асинхронная запись больших файлов с помощью FileStream.WriteAsync()

```
Ссылок: 0
async Task WriteLargeFileAsync()
{
    byte[] data = new byte[1024]; // Example data
    using (FileStream fs = new FileStream("largefile.bin", FileMode.Create,
                                          FileAccess.Write, FileShare.None, 4096, true))
    {
        await fs.WriteAsync(data, 0, data.Length);
    }
}
```

Асинхронное чтение больших файлов с помощью FileStream.ReadAsync()

```
Ссылок: 0
public async Task ReadLargeFileAsync()
{
    byte[] buffer = new byte[1024]; // Buffer for reading chunks
    using (FileStream fs = new FileStream("largefile.bin", FileMode.Open, FileAccess.Read, FileShare.None, 4096, true))
    {
        int bytesRead = await fs.ReadAsync(buffer, 0, buffer.Length);
        Console.WriteLine($"Bytes read: {bytesRead}");
    }
}
```

Обработка ошибок при асинхронных операциях с файлами

```
Ссылок: 0
public async Task ReadLargeFileAsync()
{
    byte[] buffer = new byte[1024]; // Buffer for reading chunks
    using (FileStream fs = new FileStream("largefile.bin", FileMode.Open, FileAccess.Read, FileShare.None, 4096, true))
    {
        int bytesRead = await fs.ReadAsync(buffer, 0, buffer.Length);
        Console.WriteLine($"Bytes read: {bytesRead}");
    }
}
```

LIVE

Сравнение производительности: синхронный и асинхронный режимы

Синхронная обработка файлов:

- Блокирует вызывающий поток.
- Подходит для небольших быстрых операций с файлами.

Асинхронная обработка файлов:

- Не блокирует основной поток.
- Идеально подходит для длительных операций ввода-вывода или при работе с большими файлами.

Итого

- Асинхронная обработка файлов улучшает отзывчивость приложения.
- async и await имеют решающее значение для написания неблокирующего кода.
- Используйте FileStream.WriteAsync() и FileStream.ReadAsync() для обработки больших файлов.
- Правильно обрабатывайте ошибки в асинхронных операциях.

Работа с Environment, Path

Работа с Environment, Path

- Используйте класс Environment для получения информации об операционной системе, путях к файлам и каталогах, специфичных для пользователя.

- Используйте класс Path для работы с путями к файлам и каталогам кроссплатформенным способом.

Работа с Environment

- Environment.CurrentDirectory: Получает или задает текущий рабочий каталог.

- Environment.MachineName: получает имя компьютера.
- Environment.UserName: получает имя пользователя, вошедшего в систему.
- Environment.GetFolderPath(): получает системные каталоги, такие как «Рабочий стол», «Документы» и т. д.

Работа с Path

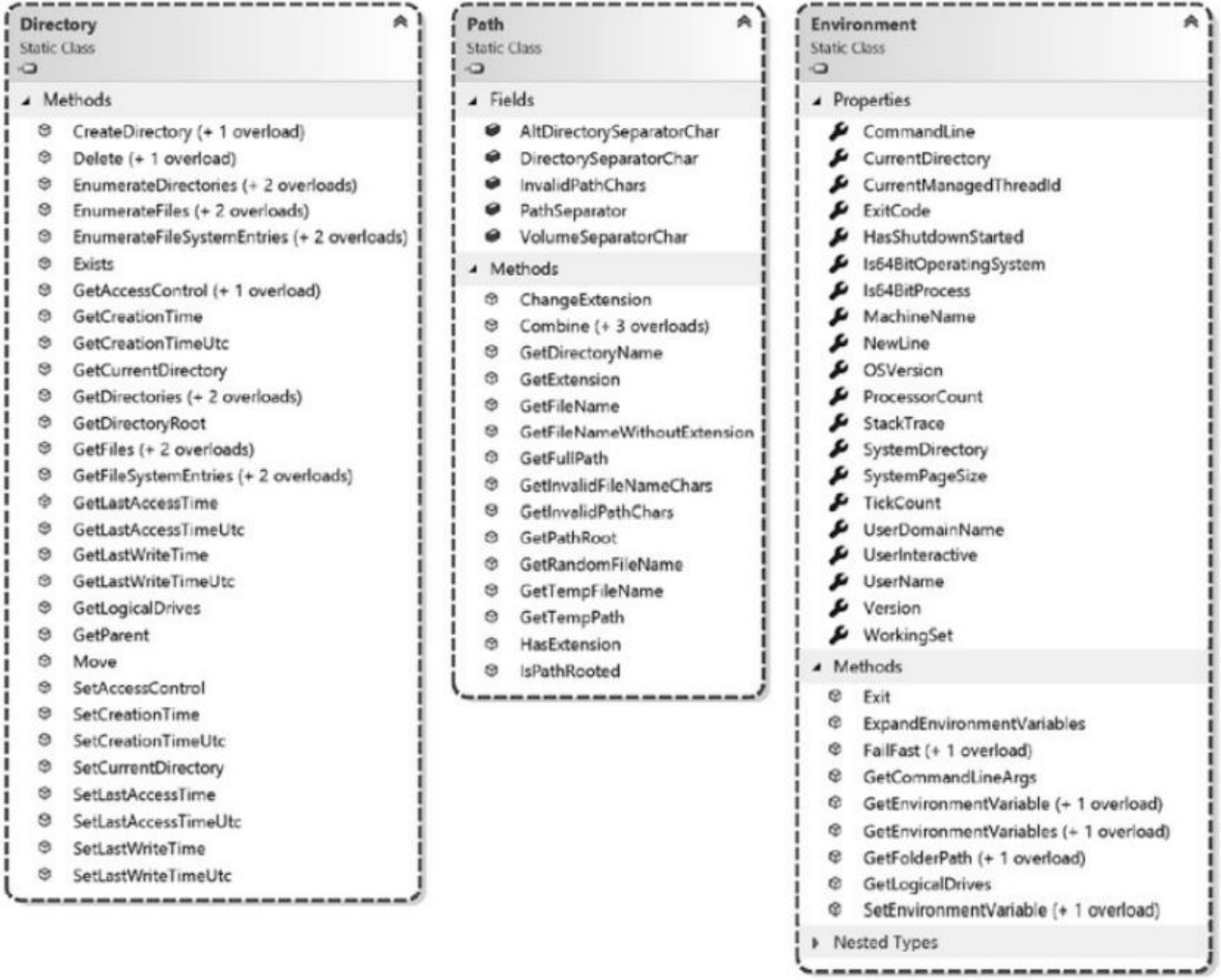
- Path.Combine(): объединяет несколько строк в один путь к файлу или каталогу.
- Path.GetFileName(): извлекает имя и расширение файла из пути.
- Path.GetDirectoryName(): извлекает часть пути к файлу, относящуюся к каталогу.
- Path.GetExtension(): извлекает расширение файла.

Получение информации о файлах и каталогах с помощью пути

- Path.Combine(): объединяет несколько строк в один путь к файлу или каталогу.
- Path.GetFileName(): извлекает имя и расширение файла из пути.
- Path.GetDirectoryName(): извлекает часть пути к файлу, относящуюся к каталогу.
- Path.GetExtension(): извлекает расширение файла.

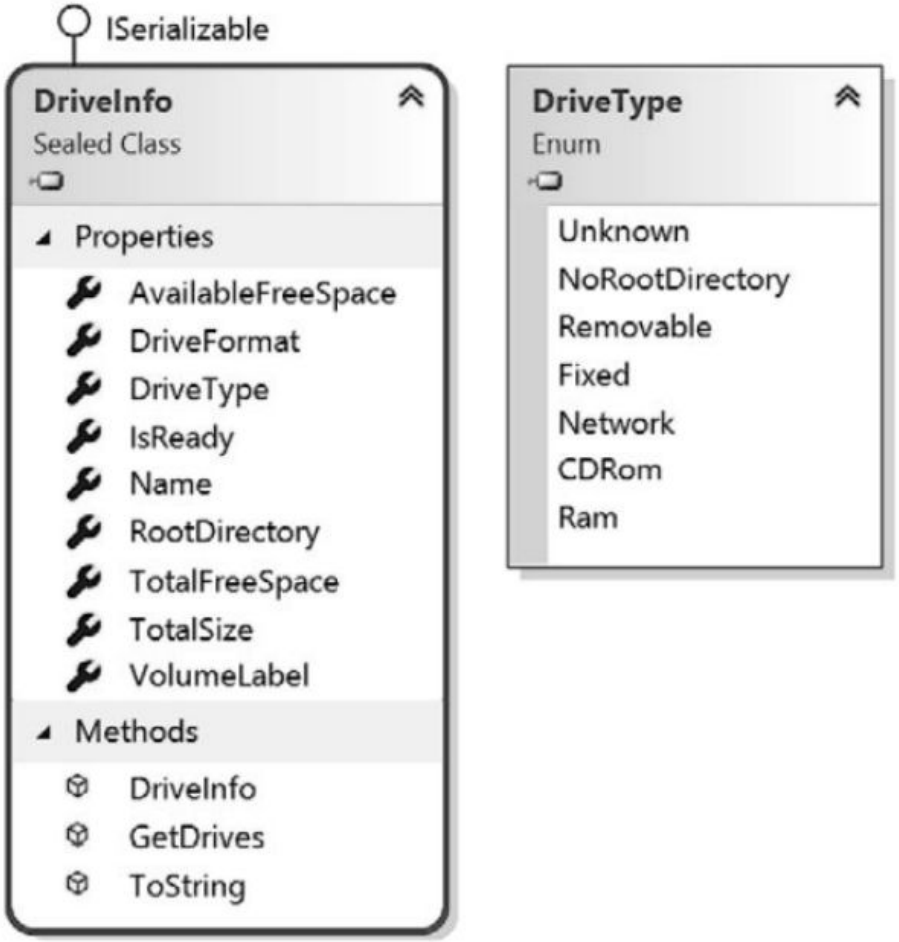
Обработка кроссплатформенных путей

- Windows использует обратные косые черты (\) для путей.
- Linux/macOS использует прямые косые черты (/).
- Используйте метод Path.Combine() для обеспечения совместимости.
- Path.DirectorySeparatorChar: Извлекает разделитель каталогов, специфичный для платформы.


Directory
Static Class
Methods
CreateDirectory (+ 1 overload)
Delete (+ 1 overload)
EnumerateDirectories (+ 2 overloads)
EnumerateFiles (+ 2 overloads)
EnumerateFileSystemEntries (+ 2 overloads)
Exists
GetAccessControl (+ 1 overload)
GetCreationTime
GetCreationTimeUtc
GetCurrentDirectory
GetDirectories (+ 2 overloads)
GetDirectoryRoot
GetFiles (+ 2 overloads)
GetFileSystemEntries (+ 2 overloads)
GetLastAccessTime
GetLastAccessTimeUtc
GetLastWriteTime
GetLastWriteTimeUtc
GetLogicalDrives
GetParent
Move
SetAccessControl
SetCreationTime
SetCreationTimeUtc
SetCurrentDirectory
SetLastAccessTime
SetLastAccessTimeUtc
SetLastWriteTime
SetLastWriteTimeUtc
Path
Static Class
Fields
AltDirectorySeparatorChar
DirectorySeparatorChar
InvalidPathChars
PathSeparator
VolumeSeparatorChar
Methods
ChangeExtension
Combine (+ 3 overloads)
GetDirectoryName
GetExtension
GetFileName
GetFileNameWithoutExtension
GetFullPath
GetInvalidFileNameChars
GetInvalidPathChars
GetPathRoot
GetRandomFileName
GetTempFileName
GetTempPath
HasExtension
IsPathRooted
Environment
Static Class
Properties
CommandLine
CurrentDirectory
CurrentManagedThreadId
ExitCode
HasShutdownStarted
Is64BitOperatingSystem
Is64BitProcess
MachineName
NewLine
OSVersion
ProcessorCount
StackTrace
SystemDirectory
SystemPageSize
TickCount
UserDomainName
UserInteractive
UserName
Version
WorkingSet
Methods
Exit
ExpandEnvironmentVariables
FailFast (+ 1 overload)
GetCommandLineArgs
GetEnvironmentVariable (+ 1 overload)
GetEnvironmentVariables (+ 1 overload)
GetFolderPath (+ 1 overload)
GetLogicalDrives
SetEnvironmentVariable (+ 1 overload)
Nested Types

Работа с диском

- Фиксированные диски: основные жесткие диски, на которых установлена операционная система (DriveType.Fixed).
- Съемные диски: USB-накопители, внешние жесткие диски (DriveType.Removable).
- Сетевые диски: общие сетевые диски (DriveType.Network).
- CD/DVD-приводы: оптические носители (DriveType.CDRom).


ISerializable
DriveInfo
Sealed Class
Properties
AvailableFreeSpace
DriveFormat
DriveType
IsReady
Name
RootDirectory
TotalFreeSpace
TotalSize
VolumeLabel
Methods
DriveInfo
GetDrives
ToString
DriveType
Enum
Unknown
NoRootDirectory
Removable
Fixed
Network
CDRom
Ram

Исключения, связанные с диском, и обработка ошибок

- Распространенные исключения для обработки:
- IOException: Общая ошибка ввода-вывода (например, диск не готов).
- UnauthorizedAccessException: Отсутствие разрешений на доступ к диску.
- DriveNotFoundException: Диск недоступен.

```
try
{
    // Perform file operations
}
catch (UnauthorizedAccessException ex)
{
    Console.WriteLine("Access denied. Please check your permissions.");
}
catch (IOException ex)
{
    Console.WriteLine("An I/O error occurred: " + ex.Message);
}
```

Класс кодировки System.Text.Encoding

- ASCII Кодирует ограниченный диапазон символов, используя семь младших битов байта.
- UTF-7 Представляет символы в виде последовательностей 7-разрядных символов ASCII. Символы Юникода, не относящиеся к ASCII, представляются в виде escape-последовательности символов ASCII.
- UTF-8 Представляет каждую кодовую точку Юникода в виде последовательности от одного до четырех байтов. UTF-8 поддерживает 8-разрядный размер данных и хорошо работает со многими операционными системами.
- UTF-16 Представляет каждую кодовую точку Юникода в виде последовательности одного или двух 16-разрядных целых чисел.
- UTF-32 Представляет каждую кодовую точку Юникода в виде 32-разрядного целого числа.

Список материалов для изучения

1. https://www.youtube.com/watch?v=zHiWqnTWsn4&t=2350s
2. https://www.c-sharpcorner.com/UploadFile/damubetha/solid-principles-in-C-Sharp/

Вопросы?

Ставим “+”,
если вопросы есть

Ставим “–”,
если вопросов нет

Рефлексия

Рефлексия

С какими впечатлениями уходите с вебинара?

Как будете применять на практике то,
что узнали на вебинаре?