C#
Dateien und Streams

70-483 – Skill 4.1

Teil 1:
Laufwerke, Verzeichnisse, Dateien und Pfade

Grundlagen

- Das .NET Framework bietet eine Reihe von Klassen für den Umgang mit Laufwerken, Verzeichnissen, Dateien und Pfaden.
- Diese Klassen stellen Methoden zum Erstellen, Kopieren, Verschieben und Löschen von Verzeichnissen und Dateien zur Verfügung.
- Es ist auch möglich, Dateien direkt zu lesen oder zu schreiben.
- Dabei muss aber der gesamte Inhalt der Datei in den Speicher geladen werden, was je nach Größe zu Problemen führen kann.
- Dennoch kann der Einsatz bei kleinen Dateien sinnvoll sein.

Laufwerke

- Bei der Arbeit mit dem Dateisystem trifft man auf oberster Ebene zunächst auf die Laufwerke eines Systems.
- Das .NET Framework bietet die Klasse `DriveInfo` zur Arbeit mit Laufwerken an.
- Diese bietet die statische Methode `GetDrives()` an, die ein Array aus `DriveInfo`-Objekten zu allen vorhandenen Laufwerken liefert.
- Jedes `DriveInfo`-Objekt bietet diverse Eigenschaften zu dem jeweiligen Laufwerk an.
- Spezielle Methoden zur Arbeit mit Laufwerken gibt es aber nicht.

Laufwerke

- Mit der Klasse `DriveInfo` kann man Informationen über das Dateisystem sammeln.
- Die Properties sind bis auf `VolumeLabel` alle `readonly`.
- Die Property `IsReady` gibt an, ob das Laufwerk bereit ist.
- Die Property `DriveType` ist vom Typ `DriveType`, einer Enumeration.

```
DriveInfo[] DriveInfos =
    DriveInfo.GetDrives();

foreach (DriveInfo di in DriveInfos) {
    Console.Write("Drive: ");
    Console.WriteLine(di.Name);
    Console.Write("  Type: ");
    Console.WriteLine(di.DriveType);
    if (di.IsReady == true) {
        Console.Write("  Format: ");
        Console.WriteLine(di.DriveFormat);
        Console.Write("  Label: ");
        Console.WriteLine(di.VolumeLabel);
        Console.Write("  Drive size: ");
        Console.WriteLine(di.TotalSize);
        Console.Write("  Free total: ");
        Console.WriteLine(di.TotalFreeSpace);
        Console.Write("  Free for User: ");
        Console.WriteLine(
            di.AvailableFreeSpace);
    }
}
```

Verzeichnisse

- Auf der nächsten Ebene im Dateisystem trifft man auf die Verzeichnisse.
- Zur Arbeit mit Verzeichnissen bietet das .NET Framework die Klassen `Directory` und `DirectoryInfo` an.
- `Directory` ist eine statische Klasse und bietet sich für einmalige Operationen auf einem Verzeichnis an.
- `DirectoryInfo` muss instanziiert werden und bietet sich an, wenn mehrere Operationen auf demselben Verzeichnis notwendig sind.
- Die Properties und Methoden der beiden Klassen überschneiden sich zu großen Teilen.

Verzeichnisse

- Ein `DirectoryInfo`-Objekt kann für ein nicht existierendes Verzeichnis erzeugt werden.
- Das Verzeichnis kann dann mit dem Objekt erstellt werden.
- `GetFileSystemInfos()` liefert Infos über Dateien und Unterverzeichnisse.
- Das Verzeichnis kann über das Objekt auch gelöscht werden.

```
static void DirectoryWorks(string path) {
    DirectoryInfo di = new
        DirectoryInfo(path);

    if (!di.Exists) {
        di.Create();
        di.CreateSubdirectory("test");

        FileSystemInfo[] infos =
            di.GetFileSystemInfos("*");
        foreach (var fsi in infos) {
            Console.WriteLine(fsi.Name);
        }
        Console.ReadLine();

        di.Delete(true);
    }
    else {
        Console.WriteLine(
            "Verzeichnis existiert schon");
    }
}
```

Dateien

- Für die Arbeit mit Dateien stellt das .NET Framework die Klassen `File` und `FileInfo` zur Verfügung.
- Analog zu `Directory` und `DirectoryInfo` ist auch `File` eine statische Klasse und `FileInfo` muss instanziiert werden.
- Beide Klassen bieten diverse Properties an, die über den Zustand einer Datei Auskunft geben.
- Darüber hinaus gibt es diverse Methoden, die das Erstellen, Kopieren, Verschieben und Löschen ermöglichen.
- Auch hier sollte `File` für einmalige Aktionen und `FileInfo` für mehrfache Aktionen verwendet werden.

Dateien

- Mit einem `FileInfo`-Objekt wird eine Datei erstellt.
- Die Methode `CreateText()` gibt ein `StreamWriter`-Objekt zurück.
- Die Methode `MoveTo()` verschiebt die Datei.
- Die Methode `CopyTo()` kopiert die Datei, behält aber die ursprüngliche Datei im Fokus.

```
static void Files() {
    FileInfo fi = new FileInfo("test.txt");

    if (!fi.Exists) {
        using (StreamWriter sw =
            fi.CreateText()) {
            sw.WriteLine("Ein wenig Inhalt");
        }
        Console.WriteLine(fi.FullName);
        Console.ReadLine();

        fi.MoveTo(@"C:\Temp\___test.txt");
        Console.WriteLine(fi.FullName);
        Console.ReadLine();

        fi.CopyTo(@".\neuertest.txt");
        Console.WriteLine(fi.FullName);
        Console.ReadLine();

        fi.Delete();
    }
}
```

Pfade

- Um das Arbeiten mit Pfadangaben zu vereinfachen, bietet das .NET Framework die statische Klasse `Path`.
- Sie bietet diverse Methoden um Informationen aus Pfadangaben zu extrahieren wie z. B. den Dateinamen, das Wurzelverzeichnis und andere Dinge.
- Darüber hinaus gibt es Methoden, die bei der Erstellung korrekter Pfadangaben unterstützen.
- Außerdem gibt es einige Felder in denen plattformspezifische Trennzeichen für Pfade oder nicht erlaubte Zeichen innerhalb von Pfaden gespeichert sind.

Pfade

Aktuelles Verzeichnis ermitteln

Pfad mit Trennzeichen zusammen-bauen

```
static void Paths() {
    string path = Directory.GetCurrentDirectory();

    string[] parts = path.Split(Path.DirectorySeparatorChar);
    foreach (string s in parts) {
        Console.WriteLine(s);
    }

    StringBuilder sb = new StringBuilder(parts[0]);
    for (int i = 1; i < parts.Length; i++) {
        sb.Append(Path.DirectorySeparatorChar);
        sb.Append(parts[i]);
    }
    string newPath = sb.ToString();
    Console.WriteLine(newPath);

    bool error = newPath
        .Intersect(Path.GetInvalidPathChars())
        .Any();
    Console.WriteLine(error);
}
```

Pfad am Trennzeichen splitten

Ermitteln, ob Pfad ungültige Zeichen enthält

Verzeichnisse überwachen

- Mit einem Objekt der Klasse `FileSystemWatcher` können Veränderungen in Verzeichnissen überwacht werden.
- Die Klasse feuert Events, wenn im überwachten Verzeichnis Dateien erstellt, verändert oder gelöscht werden.
- Für diese Events kann man sich mit Methoden registrieren, die dann bei Eintritt des Ereignis ausgeführt werden.
- So können z. B. alle Aktivitäten in einem Verzeichnis in einem Log-File protokolliert werden.
- Oder Dateien die dort abgelegt werden, können automatisiert in andere Verzeichnisse verschoben werden.

Verzeichnisse überwachen

Directory festlegen

Überwachung filtern

Überwachung starten

```
static void Main(string[] args) {
    string currentPath = Directory.GetCurrentDirectory();

    using (FileSystemWatcher watcher = new FileSystemWatcher(currentPath)) {

        watcher.NotifyFilter = NotifyFilters.FileName |
            NotifyFilters.LastWrite;

        watcher.Created += Watcher_Changed;
        watcher.Changed += Watcher_Changed;
        watcher.Deleted += Watcher_Changed;

        watcher.EnableRaisingEvents = true;

        Console.WriteLine("Watcher startet ...");
        Console.ReadKey();
    }
}

static void Watcher_Changed(object source, FileSystemEventArgs e) {
    Console.WriteLine($"Datei {e.Name} {e.ChangeType}");
}
```

Instanz erzeugen

Für Events registrieren

Methode für Event-Aufrufe

Teil 2:
Streams

Grundlagen

- Streams kapseln im weitesten Sinne eine Datenquelle, aus der gelesen und in die geschrieben werden kann, wobei die aktuelle Position des Streams ebenfalls gespeichert wird.
- Ein Stream kann als Quelle eine Datei, den Hauptspeicher oder auch eine Netzwerkverbindung kapseln.
- Streams werden unterschieden nach:

  Base-Streams/Backing-Store-Streams:
  Kapseln direkt eine Datenquelle zum Lesen und/oder Schreiben.

  Pass-through-Streams/Decorator-Streams:
  Ergänzen Base-Streams bzw. Backing-Store-Streams um weitere Funktionalitäten und können hintereinander in Reihe geschaltet werden.

Basisklasse

- Die abstrakte Klasse `Stream` ist die Basisklasse für alle anderen Stream-Klassen im .NET Framework.
- Sie stellt die grundlegenden Eigenschaften und Methoden bereit, die von ihren Subklassen geerbt und teilweise überschrieben werden.
- Die Methoden der von `Stream` abgeleiteten Klassen arbeiten alle nur mit `Byte`-Sequenzen.
- Daten, die in Streams geschrieben oder aus Streams gelesen werden, müssen also noch entsprechend gewandelt werden.

Subklassen von `Stream`

- `FileStream` (Base-Stream)

  Ein Stream zum Lesen und Schreiben aus Dateien.

- `MemoryStream` (Base-Stream)

  Ein Stream der den Arbeitsspeicher als Sicherungsspeicher nutzt.

- `NetworkStream` (Base-Stream)

  Ein Stream für den Zugriff auf eine Netzwerk-Ressource.

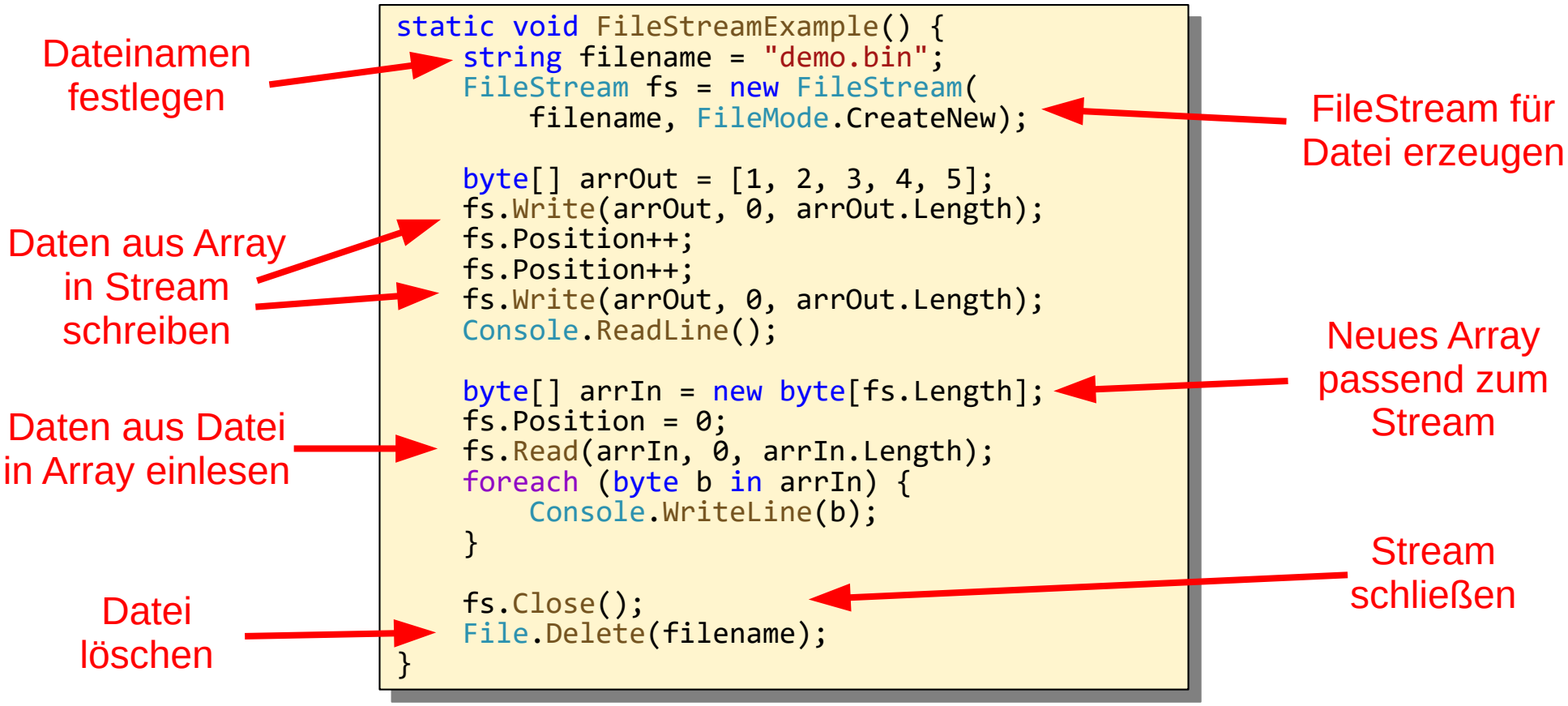
- `GZipStream` (Pass-through-Stream)

  Ein Stream zum Komprimieren und Dekomprimieren anderer Streams.

- `CryptoStream` (Pass-through-Stream)

  Ein Stream zum ver- und entschlüsseln anderer Streams.

FileStream




```
static void FileStreamExample() {
    string filename = "demo.bin";
    FileStream fs = new FileStream(
        filename, FileMode.CreateNew);

    byte[] arrOut = [1, 2, 3, 4, 5];
    fs.Write(arrOut, 0, arrOut.Length);
    fs.Position++;
    fs.Position++;
    fs.Write(arrOut, 0, arrOut.Length);
    Console.ReadLine();

    byte[] arrIn = new byte[fs.Length];
    fs.Position = 0;
    fs.Read(arrIn, 0, arrIn.Length);
    foreach (byte b in arrIn) {
        Console.WriteLine(b);
    }

    fs.Close();
    File.Delete(filename);
}
```

GZipStream

```
GZipCompress("test.txt", "test.txt.gz");
GZipDecompress("test.txt.gz", "neu.txt");

static void GZipCompress(string srcFile, string dstFile) {
    using (FileStream srcFS = File.OpenRead(srcFile)) {
        using (FileStream dstFS = File.Create(dstFile)) {
            using (GZipStream gzStream = new GZipStream(dstFS, CompressionMode.Compress)) {
                srcFS.CopyTo(gzStream);
            }
        }
    }
}

static void GZipDecompress(string srcFile, string dstFile) {
    using (FileStream srcFS = File.OpenRead(srcFile)) {
        using (FileStream dstFS = File.Create(dstFile)) {
            using (GZipStream gzStream = new GZipStream(srcFS, CompressionMode.Decompress)) {
                gzStream.CopyTo(dstFS);
            }
        }
    }
}
```

Reader-/Writer-Klassen

- Die `Stream`-Klassen im .NET-Framework arbeiten alle nur mit `Byte`-Daten.
- Wenn man andere Datentypen wie z. B. `int` oder `string` verarbeiten möchte, sind immer Umwandlungen nötig.
- Um komfortabel mit diesen Datentypen arbeiten zu können, gibt es spezielle Reader- und Writer-Klassen.
- Sie verwenden für die Verbindung zur Datenquelle ein `Stream`-Objekt, das im Konstruktor angegeben wird.
- Im Gegensatz zu den `Stream`-Klassen, gibt es für Lese- und Schreib-Operationen jeweils eine eigene Klasse.

StreamReader/StreamWriter

- Ein `StreamWriter` oder ein `StreamReader` benötigt ein `Stream`-Objekt mit dem er arbeitet.
- Dieses kann auf verschiedene Arten erzeugt werden.
- Den `FileStream` sollte man nicht vor dem `StreamWriter` schließen, da letzterer einen Puffer besitzt.

```
static void StreamReaderWriter() {
    // Datei erstellen
    string filename = "StreamWriter.txt";
    FileStream fs = File.Create(filename);

    // Text in Datei schreiben
    StreamWriter sw = new StreamWriter(fs);
    sw.WriteLine("Ein\nwenig\nText");
    sw.Close();

    // Text aus Datei lesen
    using (StreamReader sr =
        File.OpenText(filename)) {
        string l;
        while ((l = sr.ReadLine()) != null) {
            Console.WriteLine(l);
        }
    }

    // Datei wieder löschen
    File.Delete(filename);
}
```

BinaryReader/BinaryWriter

- Auch der `BinaryWriter` bzw. `BinaryReader` braucht ein `Stream`-Objekt zum Arbeiten.
- Daten beliebigen Typs können mittels `Write()` in die Datei geschrieben werden.
- Beim Lesen aus der Datei muss die Reihenfolge der Datentypen bekannt sein, da hier spezielle Methoden je Datentyp verwendet werden.

```
static void BinaryReaderWriter() {
    string filename = "BinaryWriter.bin";
    FileStream fs = File.Create(filename);

    // Binäre Daten in Datei schreiben
    BinaryWriter bw = new BinaryWriter(fs);
    bw.Write("Hallo");
    bw.Write(42);
    bw.Write(true);
    bw.Close();

    // Binäre Daten aus Datei lesen
    fs = File.OpenRead(filename);
    using (BinaryReader br =
        new BinaryReader(fs)) {
        Console.WriteLine(br.ReadString());
        Console.WriteLine(br.ReadInt32());
        Console.WriteLine(br.ReadBoolean());
    }

    File.Delete(filename);
}
```

StreamReader mit MemoryStream

```
static void MemoryStreamExample() {
    // Inhalt einer Datei lesen und im MemoryStream speichern
    // Die Datei wird direkt danach geschlossen
    string srcFile = "ToMemory.txt";
    using (MemoryStream ms = new MemoryStream(File.ReadAllBytes(srcFile))) {

        // Zugriff auf den MemoryStream mit einem StreamReader
        using (StreamReader sr = new StreamReader(ms)) {
            // Ausgabe des Inhalts zeilenweise
            string l;
            while ((l = sr.ReadLine()) != null) {
                Console.WriteLine(l);
            }

            // Den Inhalt in eine neue Datei schreiben
            string dstFile = "FromMemory.txt";
            using (FileStream fs = new FileStream(dstFile, FileMode.Create)) {
                ms.WriteTo(fs);
            }
        }
    }
}
```