

## 6.5 DATEIVERARBEITUNG

*von Michael Fischer*

Bevor es an die Verarbeitung von Dateien geht, muß zuerst die Frage geklärt werden, was Dateien überhaupt sind.

Daten werden zunächst im Arbeitsspeicher erzeugt und gespeichert.

Da alle Daten im Arbeitsspeicher und damit natürlich auch die Programmdateien bei der Programmbeendigung beziehungsweise einem Reset/Herunterfahren des Computers gelöscht werden, müssen alle Daten, die auch danach erhalten bleiben sollen, vor Beendigung des Programms auf einem Datenträger (beispielsweise auf einer Festplatte) gespeichert werden. Diese Speicherung erfolgt in allen modernen Dateisystemen strukturiert und nach Dateien getrennt.

Dateien sind Einheiten aus Daten, die unter einem bestimmten Namen auf einem Datenträger gespeichert werden. Die kleinste Einheit einer Datei ist in allen Dateisystemen ein Byte.

Im Normalfall handelt es sich beim Inhalt von Dateien um zusammenhängende Informationen. Zwar wird von keinem Dateisystem verlangt, daß Dateien streng nach Daten gleichen Typs getrennt werden müssen – so wäre es auch möglich, beispielsweise ein MP3-Stück zusammen mit einem Kochrezept in einer Datei abzuspeichern –, aber sinnvoll ist diese Trennung nicht nur, weil auf die Daten sonst unter Umständen nur noch mit speziellen Hilfsprogrammen zugegriffen werden kann, sondern grundsätzlich auch wegen der klar nachvollziehbaren Struktur – der Ordnung – innerhalb des Dateisystems.

Unter verschiedenen Dateisystemen gibt es verschiedene Regeln sowie Konventionen für Dateinamen.

So setzt sich der Dateiname unter MS Windows, das an die Tradition von CP/M und MS-DOS anknüpft, im Normalfall aus dem eigentlichen Namen und der durch einen Punkt abgetrennten Endung, die Aufschluß über den Inhalt gibt (beispielsweise .Exe für ausführbare Dateien, .Txt für Textdateien), zusammen.

Unix kennt traditionell zwar keine Dateierendungen, in der Praxis werden diese jedoch heute ebenso wie in Windows verwendet. Weder Windows noch Unix erzwingen dies.

.NET lehnt sich stark an das Dateisystemkonzept von Windows an, kennt also Laufwerksbuchstaben, UNC-Pfade, und verwendet die Konvention der Dateierendungen.

Allen modernen Dateisystemen gemeinsam ist

- ◆ die hierarchische Speicherung der Dateien in Verzeichnissen und Unterverzeichnissen,
- ◆ das Versehen der Verzeichnisse und Dateien mit Datum und Uhrzeit der letzten Änderung und
- ◆ die Vergabe von Rechten an den Verzeichnissen und Dateien.

Nun zur Verarbeitung von Dateien. Das sind alle Arbeiten, die sich auf das Dateisystem beziehen.

Dazu zählen nicht nur die gängigen Operationen mit dem Inhalt der Dateien wie das Öffnen, Lesen, Schreiben, Schließen, sondern auch solche, die die Struktur des Dateisystems betreffen, wie Verzeichnisse anlegen, Dateien verschieben, Dateigröße ermitteln und so weiter.

Gerade bei der Speicherung von Daten, bei denen es auf effizienten wahlfreien Zugriff ankommt, verwendet man heute in vielen Bereichen Datenbanken wie MS Access, MySQL oder Oracle. Dennoch sind Dateisysteme nicht wegzudenken.

Insbesondere bei der Speicherung von (nicht zu umfangreichen, schwer strukturierbaren) Daten auf Desktop-Systemen verwendet man aber nach wie vor Dateien; den Einsatz einer Datenbank müsste man in solchen Fällen als übertrieben und umständlich ansehen.

Das .NET-Framework besitzt kein eigenes Pfadsystem, sondern entspricht dem von Windows (mit Konzepten wie Laufwerksbuchstaben, UNC, Backslash als Trennzeichen und so weiter). Zunächst mag diese Tatsache enttäuschend wirken, erweckt .NET doch als java-ähnliches System den Eindruck, es wäre auf Plattformunabhängigkeit hin optimiert, jedoch könnte man mindestens einen guten Grund anführen, warum für .NET kein eigenes einheitliches Dateisystem quasi als Interface definiert wurde.

Es ist eh nur minimaler Aufwand erforderlich, um fast vollkommene Portierbarkeit auf Dateisystemebene zu erreichen, indem zentral das Basisverzeichnis gespeichert wird und alle Pfadangaben auf diesem Basisverzeichnis aufsetzen, was dank Unterstützung des Slash-Zeichens in Pfadangaben auf allen Plattformen möglich ist. Ganz abgesehen davon, daß auf Portierbarkeit hin entwickelte Anwendungen bevorzugt mit Datenbanken arbeiten dürften. Ein derartiges Dateisystem-Interface hätte sich somit kaum gelohnt.

Darüber, ob das vorrangige Ziel bei der Einführung von .NET die Portierbarkeit war, läßt sich übrigens lange diskutieren...

.NET versteht Backslashes (»\«) wie einfache Schrägstriche (»/«) als Trennzeichen in Pfaden, wobei auch Mischformen gültig sind:

```
C:\WINDOWS\SYSTEM
C:/WINDOWS/SYSTEM
C:\WINDOWS/SYSTEM
```

Die Verwendung des Schrägstrichs hat einen Vorteil: Der Backslash ist ja als Symbol für Steuerzeichen reserviert. In einer Pfadangabe wie `\newdata` würde beispielsweise das `\n` als Zeilenumbruch verstanden werden. Deswegen sollte bei der Verwendung des Backslashes dieser entweder verdoppelt werden:

```
"C:\\WINDOWS\\SYSTEM"
```

Der Compiler versteht »\\« als einen Backslash und legt diesen dann so in der Assembly ab. Oder es sollte der Klammeraffe vor Pfadangaben gesetzt werden:

```
@"C:\WINDOWS\SYSTEM"
```

Eine Alternative zur Verdoppelung des Backslashes bietet C# nämlich durch die @-Notation.

Die Angaben

```
@"C:\WINDOWS\SYSTEM"
```

und

```
"C:\\WINDOWS\\SYSTEM"
```

bezeichnen also denselben Pfad.

Zur Vermeidung mysteriöser Fehler beim Zugriff auf Verzeichnisse und Dateien sollte man sich gleich von Anfang an eine korrekte Schreibweise für Pfadangaben angewöhnen, also entweder die @-Schreibweise oder die mit verdoppelten Backslashes.

### 6.5.1 Auf Dateiinhalte zugreifen

Die .NET-Klassenbibliotheken unterscheiden zwischen zwei Methoden, um auf Dateiinhalte zuzugreifen:

- ◆ Binärzugriff: Auf die Daten kann byte- beziehungsweise blockweise zugegriffen werden. (Klassen `BinaryReader/BinaryWriter`)
- ◆ Textzugriff: Auf die Daten kann zeilenweise zugegriffen werden. Als Trennzeichen gelten Zeilenendezeichen. (Klassen `StreamReader/StreamWriter`)

Das .NET-Framework bietet eine breite Palette an Klassen für den Umgang mit Dateien und Verzeichnissen an:

<b>Klasse</b>	<b>Aufgabe</b>
BinaryReader	Lesen von Binärdaten
BinaryWriter	Schreiben von Binärdaten
BufferedStream	Puffer für Lese- und Schreibzugriff auf anderen Stream
Directory	Statische Methoden für den Verzeichniszugriff
DirectoryInfo	Instanzmethoden für den Verzeichniszugriff
DirectoryNotFoundException	Ausnahme im Falle eines fehlgeschlagenen Pfad-Zugriffes
EndOfStreamException	Ausnahme im Falle eines Zugriffs über das Streamende hinaus
ErrorEventArgs	Stellt Daten für das Error-Event zur Verfügung
File	Statische Methoden für den Dateizugriff
FileInfo	Instanzmethoden für den Dateizugriff
FileLoadException	Ausnahme, wenn Datei zwar vorhanden ist, aber nicht geöffnet werden kann
FileNotFoundException	Ausnahme im Falle des Zugriffs auf nicht existierende Datei
FileStream	Methoden für sowohl synchronen als auch asynchronen Dateizugriff
FileSystemEventArgs	Stellt Daten für Directory-Events zur Verfügung: Changed, Created, Deleted.
FileSystemInfo	Basisklasse für DirectoryInfo und FileInfo
FileSystemWatcher	Prüft auf Änderungen im Dateisystem
InternalBufferOverflowException	Ausnahme im Falle eines Überlaufs des internen Puffers
IODescriptionAttribute	Setzen der Beschreibung, die Entwicklungstools anzeigen können, wenn Events, Ergänzungen oder Eigenschaften angesprochen werden
IOException	Ausnahme im Falle eines I/O-Fehlers (beispielsweise bei Hardwaredefekten)
MemoryStream	Generieren eines Streams, der ausschließlich mit dem Arbeitsspeicher arbeitet

*Tabelle 6.5: Klassen für den Umgang mit Dateien und Verzeichnissen (Teil 1)*

Path	String-Operationen, die Pfad-Informationen enthalten (beispielsweise Ermitteln des kompletten Pfades zu einer Datei)
PathTooLongException	Ausnahme im Falle einer Überschreitung der (vom System vorgegebenen) maximalen Pfadlänge
RenamedEventArgs	Liefert Daten für das Renamed-Event
Stream	Abstrakte Basisklasse für alle Streams
StreamReader	Stream zum Lesen aus Textdateien
StreamWriter	Stream zum Schreiben in Textdateien
TextReader	Abstrakte Basisklasse zum sequentiellen Lesen
TextWriter	Abstrakte Basisklasse zum sequentiellen Schreiben

Tabelle 6.5: Klassen für den Umgang mit Dateien und Verzeichnissen (Teil 2)

Für einige dieser Klassen folgen noch Beispiele.

Die am meisten gebrauchten Klassen für Dateien sind *FileStream*, *BinaryReader* beziehungsweise *BinaryWriter*, *StreamReader* beziehungsweise *StreamWriter* und *File* beziehungsweise *FileInfo*, für Verzeichnisse *Directory* beziehungsweise *DirectoryInfo*. Die Klassen *FileInfo* beziehungsweise *DirectoryInfo* bietet zwar ähnliche und gleiche Methoden mit der jeweils selben Funktionalität wie die Klassen *File* beziehungsweise *Directory*, jedoch sind letztere statisch deklariert, das heißt, es kann auf sie zugegriffen werden, ohne daß ein Objekt erzeugt werden muß. Somit bietet sich der Einsatz der Klasse *File* dann an, wenn nur ein einziger Zugriff auf eine bestimmte Datei erfolgen soll, ansonsten der der Klasse *FileInfo*.

Möchte man in einem Programm beispielsweise nur mal schnell eine Datei kopieren, genügt der Zugriff über die statische Klasse *File*:

```
File.Copy(@"original", @"kopie");           // Datei kopieren
```

Sollen dagegen mehrere Operationen auf dieselbe Datei angewandt werden, empfiehlt sich das Anlegen eines Objekts von *FileInfo*:

```
FileInfo f = new FileInfo("original");
System.Console.WriteLine(f.Length + " Byte werden kopiert"); // Dateilänge
                                                                // ausgeben
f.CopyTo("kopie");                                           // Datei kopieren
```

Die *Exception*-Klassen wie beispielsweise *FileNotFoundException* und *PathTooLongException* sind durch ihre Namen selbsterklärend und können wie üblich über *try-catch*-Blöcke abgefangen werden, die die relevante Operationen umschließen.

## 6.5.2 Verwendung der Streamklassen

Unter einem Stream versteht man in .NET einen Datenaustauschkanal, über den Daten zwischen Anwendung und verschiedenen Datenquellen und -zielen ausgetauscht werden (beispielsweise Schreiben/Lesen in/von Dateien, Speicher oder Netzwerk).

Zum Zugriff auf den Inhalt von Dateien (Lesen und Schreiben) werden im .NET-Framework grundsätzlich Streamklassen eingesetzt.

Die Verwendung von Streamklassen bietet die Vorteile, daß sich Programmänderungen, die darin bestehen, den Zugriff auf Dateien durch Zugriffe auf andere Medien (wie Speicher oder Netzwerk) zu ersetzen, bequem durchführen lassen, indem einfach die Klasse geändert wird.

Programme, die auf Dateien zugreifen, lassen sich natürlich auch leicht um Zugriffe auf andere Medien ergänzen.

Zunächst ein Beispielprogramm zur Verwendung der Streamklassen, das die Textdatei *fa.txt* anlegt, den String »Holadiho!« hineinschreibt, diesen anschließend wieder aus der Datei ausliest und ausgibt. Die Datei wird danach gelöscht. Wie bei allen Programmen, die mit Dateien beziehungsweise Streams arbeiten, muß der Namespace *System.IO* eingebunden werden.

```
using System;
using System.IO;

public class FileAccessDemo {
    static void Main() {
        string filename = @"fa.txt";
        string schreibtext = "Joladiho!";
        string lesetext;

        // Datei fa.txt im aktuellen Verzeichnis erstellen
        // und zum Schreiben öffnen:
        StreamWriter sw = File.CreateText(filename);
        sw.Write(schreibtext);           // String in Dateispeicher schreiben
        sw.Close();                     // Datei schließen

        StreamReader sr = new StreamReader(filename); // Datei zum Lesen öffnen
        lesetext = sr.ReadToEnd();       // String aus Datei lesen
        Console.Write(lesetext);        // String ausgeben
        sr.Close();                     // Datei schließen

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

Hier einige Codebeispiele für den Gebrauch der Dateiverarbeitungs-Klassen:

### *Umgang mit Textdateien*

Textdatei auslesen:

```
// Textlese-Stream auf Datei hamster.txt:
StreamReader sr = new StreamReader("hamster.txt");
string line;

// alle Zeilen aus Stream lesen:
while ((line = sr.ReadLine()) != null)
    Console.WriteLine(line);           // gelesene Zeile ausgeben

sr.Close();                           // Datei schließen
```

Textdatei erstellen:

```
// Datei stallhase.txt im aktuellen Verzeichnis erstellen
File.CreateText("stallhase.txt");
```

Textdatei erstellen und beschreiben:

```
String wt = "watschenwauwau";

// Datei hund.txt im aktuellen Verzeichnis neu anlegen
StreamWriter sw = File.CreateText(@"hechelhund.txt");
sw.Write(wt);           // String in Dateispeicher schreiben
sw.Close();            // Datei schließen
```

Textdatei öffnen, Dateizeiger an das Ende setzen und über Stream beschreiben:

```
String zeile = "lalala";
// setzt Dateizeiger ans Ende:
StreamWriter sw = File.AppendText(@"songtext.txt");
sw.WriteLine(zeile);           // schreibt den String in die Textdatei
sw.Close();                   // Datei schließen
```

### *Umgang mit Binärdateien*

Binärdatei erstellen und beschreiben:

```
string filename = @"integer.dat";

// Datei "integer.dat" neu anlegen und Filestream darauf öffnen:
FileStream fs = new FileStream(filename, FileMode.CreateNew);
```

```
// Schreib-Binärstream auf dem Filestream aufsetzen:
BinaryWriter w = new BinaryWriter(fs);

for (int i = 1; i <= 3; i++)
    w.Write((int) i);                // schreibt Schleifenzähler in Datei

w.Close();                          // Datei schließen
```

Äquivalent zu

```
FileStream fs = new FileStream(filename, FileMode.OpenOrCreate,
                               FileAccess.Write, FileShare.None)
```

ist der Methodenaufruf

```
FileStream fs = File.OpenWrite(filename)
```

Aus Binärdatei lesen:

```
string filename = @"integer.dat";

// Datei integer.dat zum Lesen öffnen:
FileStream fs = new FileStream(filename, FileMode.Open, FileAccess.Read);

// Lese-Binärstream auf dem Filestream aufsetzen:
BinaryReader r = new BinaryReader(fs);

for (int i = 1; i <= 3; i++)
    Console.WriteLine(r.ReadInt32()); // 3 int-Werte aus Datei lesen
r.Close();                          // BinaryReader schließen
fs.Close();                          // FileStream schließen
```

Äquivalent zu

```
FileStream fs = FileStream(filename, FileMode.Open,
                           FileAccess.Read, FileShare.Read)
```

ist der Methodenaufruf:

```
FileStream fs = File.OpenRead(filename)
```

### Allgemeine Dateioperationen

Nachfolgenden geht es um Operationen, die sich mit Dateien in ihrer Gesamtheit, nicht jedoch mit ihrem Inhalt beschäftigen.

Dateigröße ermitteln:

```
FileInfo fi = new FileInfo(@"zoobewohner.txt");
Console.Write(fi.Length);
```

Zunächst wird ein FileInfo-Objekt für die Datei *fa.txt* erstellt.

An dieser Stelle werden noch keine Dateinfos eingeholt! Die Datei muß noch nicht einmal angelegt sein. Dann wird die Länge der Datei *fa.txt* im aktuellen Verzeichnis in Byte zurückgegeben. Bei *Length* handelt es sich um eine Eigenschaft der Instanz *fi*.

*Arbeit mit den Dateiattributen*

In .NET sind folgende Einzelattribute für Dateien bekannt:

Bezeichnung	Wert	Bedeutung
ReadOnly	1	Datei schreibgeschützt?
Hidden	2	Versteckte Datei?
System	4	Systemdatei?
Directory	16	Handelt es sich um ein Verzeichnis?
Archive	32	Soll Datei bei Backups berücksichtigt werden?
Temporary	256	Temporäre Datei (Datei, die nur vorübergehend für Zwischenspeicherungen existiert)?
SparseFile	512	SparseFile (große Datei, die zum Teil aus unbeschriebenen Blöcken besteht)?
ReparsePoint	1024	Besteht der Dateiinhalt aus einem Block aus Verweisen auf Dateien oder Verzeichnisse?
Compressed	2048	Handelt es sich um eine komprimierte Datei?
Offline	4096	Die Daten der Datei sind nicht sofort verfügbar
NotContentIndexed	8192	Wird die Datei von der Indizierung des Betriebssystems ausgenommen?
Encrypted	16384	Ist die Datei verschlüsselt?

*Tabelle 6.6: Dateiattribute*

Die einzelnen Attribute werden addiert. Das Ergebnis, also die Summe, ist das Dateiattribut.

Dateiattribute ermitteln:

```
Console.WriteLine(File.GetAttributes(@"pavian.txt"));
```

Hier werden über die statische Klasse *File* die Attribute der Datei *pavian.txt* ermittelt.

Angenommen, die Datei habe die Attribute *Archive* und *ReadOnly* gesetzt, dann beträgt der hier zurück- und ausgegebene Dateiattributswert 33 (= 1 für *ReadOnly* + 32 für *Archive*).

Dateiattribute setzen:

```
File.SetAttributes(@"cryptic.dat", FileAttributes.Encrypted);
```

Hier wird eine Datei mit dem Dateiattribut *Encrypted* versehen. Die bisher bestehenden Dateiattribute werden überschrieben. Damit die bestehenden Dateiattribute erhalten bleiben, müssen sie erst gelesen und dann per OR-Verknüpfung mit dem zu setzenden Wert geschrieben werden.

```
string dateiname = @"staubsauger.txt";
File.SetAttributes(dateiname,
    File.GetAttributes(dateiname) | FileAttributes.Encrypted);
```

Datei kopieren:

```
File.Copy(@"uhu.txt", @"uhu.bak"); // Datei uhu.txt nach uhu.bak kopieren
```

Dateieigenschaften wie Erstellungsdatum und Dateiattribute werden bei der Kopie direkt vom Original übernommen.

Datei umbenennen beziehungsweise verschieben:

```
// Umbenennen der Datei haukatze.txt nach hauskatze.txt:
File.Move(@"haukatze.txt", @"hauskatze.txt");
File.Move(@"hauskatze.txt", @".."); // Schlägt fehl!..
File.Move(@"hauskatze.txt", @"..\hauskatze.txt"); // Das geht!
```

Die Verschiebung funktioniert auch über Laufwerksgrenzen hinweg.

Es ist also beispielsweise möglich, die Datei *c:\hauskatze.txt* nach *e:\hauskatze.txt* zu verschieben.

Wie durch die Kommentare in den Beispielen schon angedeutet, funktioniert die Verschiebung in die nächsttiefere Verzeichnisebene über ein bloßes relatives *@..* nicht, über den relativen Pfad, also beispielsweise *@..\hauskatze.txt* dagegen schon.

Datei löschen:

```
File.Delete("kellerassel.txt"); // löscht Datei kellerassel.txt
```

Dateiexistenz prüfen:

```
if (File.Exists("monsteradressen.txt"))
    Console.WriteLine("Datei monsteradressen.txt existiert!");
else
    Console.WriteLine("Datei monsteradressen.txt existiert nicht!");
```

Dateiendung ändern:

```
string result;  
result = Path.ChangeExtension("radardaten.txt", ".bak");  
// result liefert radardaten.bak
```

Die String-Variable *result* liefert den neuen Pfadnamen.

Pfadname auf Existenz einer Endung prüfen:

```
Console.WriteLine(Path.HasExtension(@"blatt.laus")); // -> true  
Console.WriteLine(Path.HasExtension(@"blattlaus")); // -> false
```

Datumsinformationen von Dateien/Verzeichnissen:

.NET kennt drei Timestamps bei Dateien:

- ◆ Den Erstellungszeitpunkt,
- ◆ den Zeitpunkt des letzten Zugriffs und
- ◆ der Zeitpunkt des letzten Schreibzugriffs.

Diese Daten können über Methoden der Klassen *File* sowie *Directory* gelesen und gesetzt werden, die nachfolgend anhand von Beispielen vorgestellt werden.

Parallel zu diesen Methoden gibt es in der .NET-Version 1.1 noch die UTC-Varianten (Universal Time Conversion) wie beispielsweise *File.GetCreationTimeUtc(string pfad)*, die sich auf die sogenannte Greenwich-Time beziehen. UTC-Zeitangaben sind vor allem für Internet-Anwendungen interessant, um mit einer gemeinsamen Zeitbasis zu arbeiten, andere Anwendungen arbeiten jedoch üblicherweise mit der jeweiligen Ortszeit.

Datum des letzten Zugriffs auf eine bestimmte Datei ermitteln:

```
string pfad = @"e:\mp3\dackeltrance.mp3";  
DateTime dt = File.LastAccessTime(pfad);  
Console.WriteLine("Letzter Zugriff auf {0} am {1}", pfad, dt);
```

Hier werden Datum und Uhrzeit des letzten Zugriffs auf die Datei *c:\backup\hundekartei.bak* zurückgegeben.

Datum des letzten Schreibzugriffs auf eine bestimmte Datei ermitteln:

```
File.LastWriteTime(String pfad);
```

Analog zu *File.LastAccessTime()*, vergleiche vorheriges Beispiel »Datum des letzten Zugriffs auf eine bestimmte Datei ermitteln«.

Erstellungsdatum einer Datei ermitteln:

```
File.CreationTime(String pfad);
```

Analog zu *File.GetLastAccessTime()*, vergleiche »Datum des letzten Zugriffs auf eine bestimmte Datei ermitteln«.

Erstellungsdatum einer Datei ändern:

```
System.Globalization.CultureInfo info =
    // Sprache/Land = deutsch/Deutschland
    new System.Globalization.CultureInfo("de-DE", false);
    // deutscher Kalender
System.Globalization.Calendar calendar = info.Calendar;
System.DateTime dt = new System.DateTime(2001,        // Jahr
                                           03,         // Monat
                                           05,         // Tag
                                           13,         // Stunde
                                           27,         // Minute
                                           48,         // Sekunden
                                           89,         // Millisekunden
                                           calendar);  // deutscher Kalender
File.SetCreationTime(@"e:\mp3\pudelboogie.mp3", dt);
```

Datum des letzten Zugriffs auf bestimmte Datei ändern:

```
File.SetLastAccessTime(String filename, DateTime dateTime)
```

Analog zu *File.SetCreationTime()*, vergleiche vorhergehendes Beispiel »Erstellungsdatum einer Datei ändern«.

Datum des letzten Schreibzugriffs auf bestimmte Datei ändern:

```
File.SetLastWriteTime(String filename, DateTime dateTime)
```

Analog zu *File.SetCreationTime()*, vergleiche vorhergehendes Beispiel »Erstellungsdatum einer Datei ändern«.

Temporäre Datei vom System anlegen lassen:

```
System.Console.WriteLine(Path.GetTempFileName());
// liefert einen temporären Dateinamen wie z.B.
// C:\DOKUME-1\muellem\LOKALE-1\Temp\tmp117.tmp
```

Die Methode *Path.GetTempFileName()* gibt als String den Namen einer temporären Datei zurück, die automatisch im *Temp*-Verzeichnis angelegt, nicht jedoch geöffnet wird. Sie steht fortan zum Speichern temporärer Daten zur Verfügung, kann allerdings bei Beendigung des Programms vom System gelöscht werden, so daß temporäre Dateien zum Speichern von Daten über das mögliche Programmende hinaus nicht in Frage kommen.

### Verzeichnisoperationen

Der folgende Abschnitt beschäftigt sich mit Operationen, die sich auf Verzeichnisse (Ordner) beziehen.

#### Aktuelles Verzeichnis

Alle Operationen mit relativen Pfadangaben beziehen sich auf das aktuelle Verzeichnis einer .NET-Applikation, das sich sowohl ermitteln als auch setzen läßt. Standardmäßig entspricht es dem Verzeichnis, aus dem die laufende Applikation gestartet wurde. Jede Applikation führt ihr eigenes aktuelles Verzeichnis.

Aktuelles Verzeichnis ermitteln:

```
System.Console.WriteLine(Directory.GetCurrentDirectory());
```

Aktuelles Verzeichnis setzen:

```
// aktuelles Verzeichnis auf c:\mysql setzen:  
Directory.SetCurrentDirectory(@"c:\mysql");
```

Das Verzeichnis muß beim Aufruf von *Directory.SetCurrentDirectory* (*pfad*) bereits vorhanden sein.

Verzeichnis erstellen:

```
// Verzeichnis c:\test erstellen:  
Directory.CreateDirectory(@"c:\test");
```

Unterverzeichnis erstellen:

```
DirectoryInfo dir = new DirectoryInfo(@"f:\test");  
// Unterverzeichnis \subdir in f:\test erstellen:  
dir.CreateSubdirectory("subdir");
```

Verzeichnis löschen:

```
// leeres Verzeichnis c:\test\subdir löschen:  
Directory.Delete(@"c:\test\subdir");
```

Das Verzeichnis muß leer sein. Ganze Verzeichnisbäume (Verzeichnisse mit Unterverzeichnissen, die nicht leer sein müssen) können so gelöscht werden:

```
Directory.Delete(@"c:\test\", true);
```

Hier wird das Verzeichnis *c:\test* mit allen darin enthaltenen Dateien und Unterverzeichnissen rekursiv gelöscht.

Verzeichnisexistenz prüfen:

```
// prüfen ob Verzeichnis c:\data vorhanden ist:
if (Directory.Exists(@"c:\data"))
    System.Console.WriteLine("Verzeichnis existiert");
else
    System.Console.WriteLine("Verzeichnis existiert nicht");
```

Verzeichnis umbenennen beziehungsweise verschieben:

```
Directory.Move(@"c:\hundetaten", @"c:\hundedaten");
// benennt Verzeichnis c:\hundetaten nach c:\hundedaten um

Directory.Move(@"c:\dackelaten", @"c:\hundedaten\dackelaten");
// verschiebt das Verzeichnis c:\dackelaten
// nach c:\hundedaten\dackelaten
```

Damit Verzeichnisse verschoben werden können, müssen sie nicht leer sein.

Alle Dateien eines Verzeichnisses ermitteln:

```
string path = @"c:\"; // Pfad c:\ festlegen
DirectoryInfo dir = new DirectoryInfo(path);
// lese die Infos aller Dateien in c:\ in ein Array:
FileInfo[] fil = dir.GetFiles();
Console.WriteLine("Dateien im Verzeichnis {0} :", path);
foreach (FileInfo filename in fil) // Dateinamen aller Dateien
    Console.WriteLine(filename.Name); // aus dem Array ausgeben
```

Über *Directory.GetFiles(path)* werden alle Dateien im angegebenen Pfad eingeholt, dies geschieht unabhängig vom Dateiattribut. Die Dateien von Unterverzeichnissen ab dem angegebenen Pfad werden nicht mitgezählt.

Anzahl der Dateien eines Verzeichnisses ermitteln:

```
string path = @"f:\backup";
Console.WriteLine("Anzahl der Dateien im Verzeichnis {0} : {1}",
    path, DirectoryInfo.GetFiles(path).Length);
```

Über *Directory.GetFiles()* werden alle Dateien im angegebenen Pfad eingeholt, dies geschieht unabhängig vom Dateiattribut. Die Dateien von Unterverzeichnissen ab dem angegebenen Pfad werden nicht mitgezählt.

Anzahl der Dateien im Verzeichnis mit bestimmter Zeichenkette im Dateinamen ermitteln:

```
string path = @"c:\"; // Pfad c:\ festlegen
DirectoryInfo dir = new DirectoryInfo(path);
// lese die Infos aller Dateien mit "x" im Namen in ein Array:
```

```
FileInfo[] fil = dir.GetFiles("*x*");
Console.WriteLine("Anzahl Dateien mit x im Namen im Verzeichnis {0} : {1}",
    path, fil.Length);
```

Das Beispiel gibt die Anzahl der Verzeichnisse und Dateien aus, die ein »x« im Namen haben.

Dabei werden alle Dateien mit einem »x« im Dateinamen im angegebenen Pfad gezählt, unabhängig vom Dateiattribut. Die Dateien von Unterverzeichnissen ab dem angegebenen Pfad werden nicht mitgezählt.

Dateinamen aus Pfadangabe extrahieren:

```
Console.WriteLine(Path.GetFileName(@"C:\mydog\dackel.txt")); // -> dackel.txt
```

Das ist eine reine String-Operation. Es findet kein Zugriff auf den *Path.GetFileName* (*pfad*) übergebenen Pfad statt, also muß der Pfad nicht vorhanden sein.

Dateinamen ohne Dateiendung aus Pfadangabe extrahieren:

```
Console.WriteLine(Path.GetFileNameWithoutExtension(@"C:\tiere\iltis.txt"));
// -> iltis
```

Hierbei handelt es sich um eine reine String-Operationen. Es findet kein Zugriff auf den *Path.GetFileNameWithoutExtension* (*pfad*) übergebenen Pfad statt, also muß der Pfad nicht existieren.

Pfade kombinieren:

```
string pfad1 = @"c:\dackelclub";
string pfad2 = @"bodo.txt";
string pfad3 = @"kampfduckel\";
string pfad4 = @"";
string pfad5 = null;

Console.WriteLine(Path.Combine(pfad1, pfad2)); // c:\dackelclub\bodo.txt
Console.WriteLine(Path.Combine(pfad1, pfad3)); // c:\dackelclub\kampfduckel\
Console.WriteLine(Path.Combine(pfad1, pfad4)); // c:\dackelclub
Console.WriteLine(Path.Combine(pfad1, pfad5)); // System.ArgumentNullException
```

Hierbei handelt es sich um eine reine String-Operationen. Es findet kein Zugriff auf die *Path.Combine* (*pfad1*, *pfad2*) übergebenen Pfade statt, also müssen die Pfade nicht existieren.

Parent-Verzeichnis eines bestimmten Pfades ermitteln:

```
string pfad1 = @"c:\mysql\bin";
string pfad2 = @"c:\gemeinde\vereine\sport\handball\mitglieder";
DirectoryInfo di;
// Parent-Verzeichnis von pfad1 ermitteln
```

```

di = Directory.GetParent(pfad1);
System.Console.WriteLine(di.Name);          // liefert mysql
// Parent-Verzeichnis von pfad2 ermitteln
di = Directory.GetParent(pfad2);
System.Console.WriteLine(di.Name);          // liefert handball

```

Das ist eine reine String-Operation. Es findet kein Zugriff auf den *Directory.GetParent(pfad)* übergebenen Pfad statt, also muß der Pfad nicht vorhanden sein.

Root-Verzeichnis eines bestimmten Pfades ermitteln:

```

string pfad = @"c:\hundedaten\dackeldaten";
System.Console.WriteLine(Directory.GetDirectoryRoot(pfad)); // c:\

```

Hierbei handelt es sich um eine reine String-Operation. Es findet kein Zugriff auf den *Directory.GetDirectoryRoot(pfad)* übergebenen Pfad statt, also muß der Pfad nicht existieren.

Vorhandene System-Laufwerke (inklusive Netzwerk) ermitteln:

```

string[] drives = Directory.GetLogicalDrives();
Console.WriteLine("Vorhandene Laufwerke: ");
for (int i = 0; i < drives.Length; i++)
    Console.WriteLine(drives [i] + " ");

```

Die Laufwerke werden von der Methode *Directory.GetLogicalDrives()* in einem String-Array gespeichert. Über die Eigenschaft *Length* eines Arrays bekommt man die Länge, in diesem Fall die Anzahl der Laufwerke im System, heraus.

Temporäres Verzeichnis des Systems ermitteln:

```

System.Console.WriteLine(Path.GetTempPath());
// liefert einen Pfad wie C:\DOKUME-1\muellem\LOKALE-1\Temp

```

Beim temporären Verzeichnis handelt es sich um das Systemverzeichnis, in dem Programme vorübergehend Dateien speichern können. In der Praxis verwendet man es häufig, um Daten an externe Anwendungen zu übergeben, die diese nur über das Dateisystem akzeptieren. Eine Anwendung zum Umgang mit Archiven muß beispielsweise, wenn eine Textdatei aus dem Archiv betrachtet werden soll, die archivierte Datei in das temporäre Verzeichnis speichern und Notepad mit dieser Datei aufrufen. Da solche temporären Daten in diesem Verzeichnis nach Beendigung einer Applikation vom System gelöscht werden können, kommt es für längerfristige Auslagerungen nicht in Frage.

Prüfen, ob es sich um absolute oder relative Pfadangabe handelt:

```
string abspfad = @"c:\absolut";  
string relpfad = @"relativ";  
System.Console.WriteLine(Path.IsPathRooted(abspfad));           // liefert True  
System.Console.WriteLine(Path.IsPathRooted(relpfad));           // liefert False
```

Hierbei handelt es sich um eine reine String-Operation. Es findet kein Zugriff auf den *Path.IsPathRooted(pfad)* übergebenen Pfad statt, also muß der Pfad nicht existieren.

### 6.5.3 Isolierte Speicherung

Will eine Anwendung ihre Einstellung oder andere Daten speichern, stellen sich drei Probleme:

- ◆ Anwendungen, auch wenn sie vertrauenswürdig sind, sollen nicht die Möglichkeit haben, die Einstellungen beliebiger anderer Anwendungen zu lesen oder zu überschreiben.
- ◆ Nicht vertrauenswürdige Anwendungen sollen keinen Zugriff auf das gesamte Dateisystem haben, aber dennoch ihre eigenen Daten lesen und schreiben können.
- ◆ Serveranwendungen müssen oft Einstellungen von vielen unterschiedlichen Benutzern speichern und brauchen dazu ein System, um die Datensätze voneinander zu trennen.

In .NET wird dies mit dem Konzept der isolierten Speicherung gelöst:

- ◆ Eine Anwendung kann ihren eigenen isolierten Speicher anfordern, auf den keine andere .NET-Anwendung zugreifen darf.
- ◆ Mit passenden Sicherheitseinstellungen kann man auch verhindern, daß bestimmte, nicht vertrauenswürdige Anwendungen auf irgendetwas anderes als ihren eigenen »isolated Storage« Zugriff haben.
- ◆ Serveranwendungen können beispielsweise mehrere Bereiche von isolierter Speicherung anfordern, um die Daten von unterschiedlichen Benutzern zu speichern, ohne selbst ein Verwaltungssystem implementieren zu müssen.

In der Praxis wird das vom Framework so gelöst, daß isolierte Speicherung als Unterverzeichnis im Profil des Anwenders abgelegt wird. Isolierte Speicherung ist daher nicht geeignet, sensible Informationen wie unverschlüsselte Passwörter sicher zu speichern, da mit »unmanaged« (nicht .NET-) Anwendungen weiterhin auf das gesamte Dateisystem zugegriffen werden kann.

Für die isolierte Speicherung gibt es im Namespace *System.IO.IsolatedStorage* folgende Klassen:

<b>Klasse</b>	<b>Beschreibung</b>
IsolatedStorage	Abstrakte Basisklasse
IsolatedStorageException	Ausnahme, die auftritt, wenn bei einer Operation mit isoliertem Speicher ein Fehler auftritt
IsolatedStorageFile	Klasse zum Zugriff auf isolierten Speicher
IsolatedStorageFileStream	Klasse zum Stream-Zugriff auf isolierten Speicher

*Tabelle 6.7: Klassen für isolierte Speicherung*

Üblicherweise wird zunächst über den Aufruf der Methode *GetStore()* der Klasse *IsolatedStorageFile*

```
IsolatedStorageFile isoStore = IsolatedStorageFile.GetStore(
    IsolatedStorageScope.User | IsolatedStorageScope.Assembly, null, null);
```

ein isolierter Speicherbereich angefordert.

Auf diesen kann man dann über die zur Verfügung gestellte Stream-Klasse zugreifen:

```
StreamReader r = new StreamReader(new IsolatedStorageFileStream(
    filename, FileMode.Open, isoStore));
String isoContent = r.ReadLine();
// ...
```

Im folgenden Beispiel wird eine Datei *isofile.txt* im isolierten Speicher erstellt und der String »miep miep« hineingeschrieben.

```
using System;
using System.IO;
using System.IO.IsolatedStorage;

public class IsolatedStorageDemo {
    const string filename = "isofile.txt";
    const string filestring = "miep miep";

    public static void Main() {
        IsolatedStorageFile isoStore =
            IsolatedStorageFile.GetStore(IsolatedStorageScope.User |
            IsolatedStorageScope.Assembly, null, null);

        // String in den isolierten Speicher schreiben
        StreamWriter w = new StreamWriter(new IsolatedStorageFileStream(filename,
            FileMode.CreateNew, isoStore));

        w.WriteLine(filestring);
        w.Close();
    }
}
```

```
// String aus isoliertem Speicher lesen
StreamReader r = new StreamReader(new IsolatedStorageFileStream(filename,
                                                                    FileMode.Open, isoStore));

String isoContent = r.ReadLine();
r.Close();
Console.WriteLine("Die Datei " + filename + " enthält:\n{0}", isoContent);
}
}
```

Auf diese Daten hat nur Code des gleichen Assemblies Zugriff. Andere Assemblies bekommen beim Lesezugriff nicht mal eine Zugriffsverweigerung, sondern finden die Daten erst gar nicht vor – für sie existieren diese Daten nicht, was in einer *FileNotFoundException* resultiert.

Ausprobieren können Sie das, indem Sie ein weiteres Programm erstellen, das aus der selben Datei zu lesen versucht, beispielsweise indem Sie aus dem Beispielprogramm den Teil

```
// String in den isolierten Speicher schreiben
StreamWriter w = new StreamWriter(new IsolatedStorageFileStream(filename,
                                                                    FileMode.CreateNew, isoStore));

w.WriteLine(filestring);
w.Close();
```

auskommentieren und dann unter einem anderen Dateinamen kompilieren und ausführen.

## 6.6 DELEGATEN

von Michael Fischer

Delegaten sind objektorientierte, typsichere Funktionszeiger, die Referenzen auf Methoden speichern. Über den Aufruf von Delegaten können die referenzierten Methoden indirekt ausgeführt werden und somit beispielsweise ganz flexibel als Methodenparameter (Callbacks) übergeben werden.

In C# halten Delegaten sowohl durch die Klasse *System.Delegate* als auch durch das Schlüsselwort *delegate* Einzug.

Weiterhin lassen sich über sogenannte MulticastDelegaten Listen von Funktionszeigern führen, bei deren Aufruf die darin gespeicherten Funktionen der Reihe nach abgearbeitet werden. MulticastDelegaten spielen vor allem bei der Event-Abarbeitung in .NET eine Rolle.

Das folgende Beispielprogramm soll zeigen, wie eine Methode per Delegat aufgerufen wird: