10 Dateizugriffe mit Visual Basic .NET

Das Thema Dateizugriffe gehört zu jenen Themen, die in praktisch sämtlichen auf Windows basierenden Programmiereinführungen nahezu gleich behandelt werden. Schuld daran ist nicht die »Einfallslosigkeit« der Autoren, sondern der Umstand, dass das Dateisystem den Rahmen und die Regeln vorgibt und die Umsetzung lediglich von der Syntax der jeweiligen Programmiersprache abhängt. Auch bei .NET ist das nicht anders. Da .NET auf dem Betriessystem (Windows oder Unix) aufsetzt und dieses nicht ersetzt, übernimmt es auch die Regeln, die durch das Betriebssystem vorgegeben werden.

Eine wichtige Neuerung gibt es dennoch. Bei .NET ist eine Datei lediglich ein logisches Konzept und nur ein Spezialfall unter mehreren. Im Mittelpunkt steht der *Strom* (engl. »stream«), der eine Aneinanderreihung von Zeichen repräsentiert. Ob diese Daten aber nun in eine Datei, direkt an die Ausgabekonsole oder vielleicht in ein Crypto-Modul zwecks Verschlüsselung fließen, ist dabei sekundär. Dies ist ein weiteres Beispiel für die Flexibilität, die mit .NET einhergeht. Eine Flexibilität, die für viele Visual Basic-Programmierer aber erst einmal bedeutet, sich mit neuen, grundlegenden Konzepten beschäftigen zu müssen. Doch auch hier erweist sich .NET als unser Freund, indem es die vertrauten »Dateizugriffsbefehle« (allerdings als Funktionen und nicht als Befehle, sowie mit einer leicht geänderten Syntax) zur Verfügung stellt, die Basic-Programmierer bereits seit »Äonen« kennen. Niemand wird daher gezwungen, sich mit Streams zu beschäftigen, doch früher oder später wird dies die normale Art und Weise darstellen, Dateiinhalte zu lesen und zu schreiben.

Ein weiterer Aspekt wird dazu führen, dass die Bedeutung der Standarddateizugriffsbefehle bei .NET stark zurückgehen wird: XML als alternatives Datenformat. In vielen Fällen wird es nahe liegender und praktischer sein, Daten im XML-Format über die zuständigen Klassen im *System.Xml*- oder *System.Data*-Namespace zu speichern, als dafür Dateizugriffsbefehle zu benutzen. Auch wenn die guten, alten Random-Dateien nach wie vor unterstützt werden, sollten sie für neue Projekte zugunsten von XML nicht mehr in Betracht gezogen werden.

Die Stichworte für dieses Kapitel:

- Allgemeine Dateizugriffe
- Die Rolle der Streams
- Textdateien schreiben und lesen
- Binäre Daten schreiben und lesen
- Überblick über den *System.IO*-Namespace
- Zugriffe auf das Dateisystem
- Arrays in einer Datei speichern und wieder auslesen
- Für alle Fälle: die Kompatibilitätsfunktionen
- Verzeichnisse überwachen
- Umgang mit unterschiedlichen Zeichensätzen – die *Encoding*-Klassen

10.1 Allgemeine Dateizugriffe

Auch wenn wirklich niemand bei Visual Basic .NET »gezwungen« wird, sich für simple Dateizugriffe mit dem anfangs ein wenig andersartigen Konzept der Streams zu beschäftigen, sollen diese Streams dennoch in diesem Kapitel im Mittelpunkt stehen (mehr zu den konventionellen Dateibefehlen in Kapitel 10.7). Los geht es mit kleinen Beispielen, die am besten den grundsätzlichen Umgang mit den allgegenwärtigen Streams deutlich machen.

Die Klassen für den Dateizlugriff befinden sich im Namespace System.IO. *Die Beispiele importieren den Namespace daher, wenngleich dies nicht zwingend notwendig ist.*

10.1.1 Ein kurzer Überblick

Sie werden in diesem Kapitel eine Reihe von ähnlich klingenden Stream-Klasen kennen lernen, die auf dem Konzept eines Streams basieren, das in Kapitel 10.2 ausführlicher vorgestellt wird. Als erste Orientierung schon einmal vorweg:

- Für das Lesen von Textdateien kommt die *StreamReader*-Klasse zum Einsatz.
- Das Schreiben von Textdateien übernimmt die *StreamWriter*-Klasse.

Für das Lesen von Daten gibt es beim *StreamReader* die Methoden *Read* und *ReadLine*, die Pendants der *StreamWriter*-Klassen lauten *Write* und *WriteLine*. Damit lässt sich bereits eine Menge bewerkstelligen, sodass die Streams

bei weitem nicht so »kompliziert« sind, wie es vielleicht bei oberflächlicher Betrachtung erscheinen mag.

10.1.2 Die StreamReader-Klasse

Die Aufgabe der *StreamReader*-Klasse ist es, Bytes mit einer bestimmten Codierung (standardmäßig UTF-8-Zeichensatz) aus einem Stream in eine Datei zu übertragen. Die *StreamReader*-Klasse erweitert die *TextReader*-Klasse. Die Tabelle 10.1 enthält die wichtigsten Mitglieder der *StreamReader*-Klasse.

Tabelle 10.1: Die wichtigsten Mitglieder der *StreamReader*-Klasse

Mitglied	Bedeutung
BaseStream-Eigenschaft	Gibt das zugrunde liegende *Stream*-Objekt zurück.
Close-Methode	Schließt das Objekt und gibt alle damit verbundenen Ressourcen frei.
Peek-Methode	Gibt das nächste zu lesende Zeichen zurück, ohne es tatsächlich zu lesen.
ReadBlock-Methode	Liest eine vorgegebene Anzahl an Zeichen ein.
ReadLine-Methode	Liest genau eine Zeile, d.h. alle Zeichen bis zum nächsten Zeilenumbruch/Neue Zeile-Zeichenpaar ein.
Read-Methode	Liest die nächsten Zeichen aus dem Strom.
ReadToEnd-Methode	Liest alle noch verbleibenden Zeichen in einem Rutsch.

Das folgende Beispiel zeigt, wie sich der Inhalt einer Textdatei (in diesem Fall *Msdos.sys*) einlesen lässt. Im Mittepunkt steht die *StreamReader*-Klasse.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_StreamReader.vb.

Listing 10.1: Auslesen einer Textdatei mit einer *StreamReader*-Klasse

```
' ======================================
' Lesen einer Textdatei mit StreamReader
' Visual Basic .NET Kompendium
' ======================================

Option Explicit
Imports System.IO
Imports System.Console
```

```
Class App

 Shared Sub Main ()
    Dim oSt As New StreamReader("C:\Msdos.sys")
    WriteLine(oSt.ReadToEnd())
    oSt.Close()
 End Sub

End Class
```

Das Auslesen der Textdatei findet in zwei Befehlszeilen statt und ist damit sehr kurz:

```
Dim oSt As New StreamReader("C:\Msdos.sys")
WriteLine(oSt.ReadToEnd())
```

Als Erstes wird die *StreamReader*-Klasse mit dem Namen der zu öffnenden Datei instanziiert. Anschließend liest die *ReadToEnd*-Methode alle Zeichen ein, sodass die *WriteLine*-Methode sie ausgeben kann.

Wie üblich geht es noch ein wenig kürzer, wobei diese »Kurzschreibweise« stets zu Lasten der Lesbarkeit geht und daher gerade am Anfang nicht empfohlen wird.

Der folgende Befehl öffnet eine Datei über ein *StreamReader*-Objekt und gibt ihren Inhalt gleichzeitig aus.


```
WriteLine(New StreamReader("C:\Msdos.sys").ReadToEnd())
```

Auch wenn diese Kompaktheit durchaus ihren Reiz hat, besitzt sie doch Nachteile. Neben der erschwerten Lesbarkeit wird in diesem Fall keine Variable für das *StreamReader*-Objekt angelegt, über das es sich wieder schließen ließe. Nachfolgende Zugriffe auf die Datei führen daher zu der Ausnahme, dass die Datei bereits von einem Prozess verwendet wird und daher nicht geöffnet werden kann.

Wenn eine Datei nicht gefunden wird

Gerade die Dateizugriffsbefehle sind fehlerträchtig, denn es kann immer passieren, dass eine Datei nicht dort ist, wo sie das Programm vermutet, ein Verzeichnis nicht existiert oder ein Netzwerklaufwerk nicht bereit ist. Alle diese »Fauxpas« führen bei Visual Basic .NET zu Ausnahmen (die »Nachfolger« der Laufzeitfehler früherer Versionen), die sich von der *System.IO.Exception*-Klasse ableiten (siehe Tabelle 10.2). Abgefangen werden Ausnahmen über den *Try*- Befehl, der in Kapitel 11 Thema ist. Eine Fehlernummer gibt es hier offiziell zwar nicht, lediglich eine Fehlermeldung. Doch da das *Err*-Objekt bei Visual Basic .NET nach wie vor unterstützt wird, kann man (wer unbedingt will oder muss) auch eine Fehlernummer erhalten.

Ausnahme	Wann tritt sie auf?
PathTooLongException	Der Verzeichnispfad ist zu lang.
DirectoryNotFoundException	Das Verzeichnis existiert nicht.
EndOfStreamException	Das Ende des Stroms (der Datei) wurde erreicht.
FileNotFoundException	Die Datei gibt es nicht.
FileLoadException	Der Dateiinhalt kann aus irgendeinem Grund nicht geladen werden.

Tabelle 10.2: Ausnahmen, die im Zusammenhang mit Dateizugriffen eine Rolle spielen

10.1.3 Die StreamWriter-Klasse

Die *StreamWriter*-Klasse ist das Pendant zur *StreamReader*-Klasse. Sie schreibt die Zeichen eines *Stream*-Objekts entweder zeichen- oder zeilenweise in die darunter liegende Datei. Die Tabelle 10.3 enthält die wichtigsten Mitglieder der *StreamWriter*-Klasse.

Mitglied	Bedeutung
BaseStream-Eigenschaft	Gibt das zugrunde liegende *Stream*-Objekt zurück.
Encoding-Eigenschaft	Ermöglicht den Zugriff auf das zuständige *Encoding*-Objekt, über das eine Konvertierung von einem Zeichensatz in einen anderen durchgeführt werden kann (zur Auswahl stehen ASCII, Unicode, UTF-7 und UTF-8).
NewLine-Eigenschaft	Gibt den Zeichencode für das Zeichen zurück, mit dem eine Zeile beendet wird, oder legt es fest.
Close-Methode	Schließt das *StreamWriter*-Objekt und damit auch den darunter liegenden Stream.
Flush-Methode	Überträgt alle noch im Zwischenspeicher befindlichen Zeichen in den darunter liegenden Stream und damit in die Datei.
Write-Methode	Schreibt einen einzelnen Wert oder ein Array in den Stream.
WriteLine-Methode	Schreibt Zeichen in den Stream und schließt sie mit einem Zeilenende-Zeichen ab.

Tabelle 10.3: Die wichtigsten Mitglieder der *StreamWriter*-Klasse

Das folgende Beispiel ist ein Beispiel für den Schreibzugriff auf eine Textdatei. Es nimmt so lange Namen entgegen, die in einer Datei gespeichert werden, bis der Anwender nichts mehr eingibt. Dann wird der komplette Dateiinhalt wieder ausgegeben.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_StreamWriter.vb.

Listing 10.2: Speichern in einer Textdatei mit einer *StreamWriter*-Klasse

```
' ================================================
' Speichern in einer Textdatei mit StreamWriter
' Visual Basic .NET Kompendium
' ================================================
Option Explicit On
Imports System.IO
Imports System.Console

Class App

  Shared Sub Main()
    Dim stEingabe As String
    Dim i As Integer
    Dim obFi As FileStream = New _
     FileStream("Namen.dat", FileMode.Create)
    Dim obStw As StreamWriter = New StreamWriter(obFi)
    Do
     Write("Name? ")
     stEingabe = ReadLine()
     If stEingabe = "" Then Exit Do
        obStw.WriteLine(stEingabe)
        i += 1
    Loop
    obStw.Close()
    WriteLine("{0} Namen gespeichert!", i)
    i = 0
    obFi = New FileStream("Namen.dat", FileMode.Open)
    Dim obStr As StreamReader = New StreamReader(obFi)
    Do until obStr.Peek() = -1
       WriteLine(obStr.ReadLine)
       i += 1
    Loop
    obStr.Close()
    WriteLine("{0} Namen gelesen!", i)
  End Sub
End Class
```

10.1.4 Die FileStream-Klasse

Das *FileStream* ermöglicht den direkten Zugang zum Inhalt einer Datei in Gestalt eines Streams. Dass es hier nicht um die Datei, so wie sie durch das Dateisystem repräsentiert wird, sondern um ihren Inhalt geht, machen die Mitglieder deutlich, die in Tabelle 10.4 zusammengestellt sind. Es gibt keine *ReadLine*-Methode, mit der sich der Textinhalt zeilenweise lesen ließe. Das liegt ganz einfach daran, dass der Dateiinhalt nicht als Textstrom, sondern als allgemeiner Strom betrachtet wird.

Tabelle 10.4: Die wichtigsten Mitglieder der *FileStream*-Klasse

Mitglied	Bedeutung
Close-Methode	Schließt die darunter liegende Datei.
Flush-Methode	Bewirkt, dass alle noch im Arbeitsspeicher befindlichen Daten in das Gerät geschrieben werden.
Length-Eigenschaft	Länge des Streams in Byte.
Position-Eigenschaft	Aktuelle Position des Lese-/Schreibzeigers.
ReadByte-Methode	Liest ein einzelnes Byte aus dem Strom.
Read-Method	Liest einen Datenblock der angegebenen Größe aus dem Strom und überträgt die Zeichen in ein Byte-Array.
WriteByte-Methode	Schreibt ein einzelnes Byte in den Strom.
Write-Methode	Schreibt einen Datenblock aus einem Byte-Array in den Strom.

10.1.5 Der Unterschied zwischen FileStream und StreamReader

Am Anfang hören sich viele Klassen im Namespace *System.IO* recht ähnlich an. Es gibt eine *StreamReader*-, eine ähnlich klingende *TextReader*, eine *StreamWriter*- und eine ähnlich klingende *TextWriter*-, eine *FileStream*-Klasse sowie schließlich die *Stream*-Klasse. Eine komplette Übersicht über die wichtigsten Klassen gibt die Tabelle 10.6, während die Abbildung 10.3 die Klassen in einem Schaubild präsentiert. Letztere ist nur ein abstraktes Modell für einen (Daten-)Stream und wird in verschiedenen Klassen implementiert. Eine davon ist die *FileStream*-Klasse, die das Modell des Streams auf den Inhalt einer Datei überträgt, sodass dieser im Sinne des Stream-Modells angesprochen werden kann. Die *FileStream*-Klasse greift direkt auf den mit einer Datei verbundenen Stream zu. Hier lässt sich der Inhalt der Datei nur zeichen- oder blockweise lesen. In letzterem Fall erhält man ein Byte-Array, das in einen String konvertiert werden müsste. Geht es darum, einen Stream als einen Strom von Textzeichen zu behandeln, sind die Klassen

StreamReader und *StreamWriter* optimaler, denn hier gibt es Methoden wie *ReadLine* oder *WriteLine*, die eine komplette Textzeile lesen bzw. schreiben können.

StreamReader und *StreamWriter* sind spezialisierte Klassen, die sich von den allgemeineren Klassen *TextReader* und *TextWriter* ableiten. Diese sind wieder abstrakte Klassen, die einen allgemeinen Textleser bzw. Textschreiber darstellen. Sie bieten eine allgemeine Funktionalität, die in Spezialklassen (unter anderem *StreamWriter* und *HttpWriter*) implementiert wird. Übrigens, und das sollte in einem etwas fortgeschritteneren Kapitel keine Überraschung mehr sein, können Sie *TextReader* und *TextWriter* auch in eigenen Klassen implementieren. Sie erhalten damit eine kompatible Klasse, die überall dort eingesetzt werden kann, wo *TextReader* und TextWriter vorgesehen sind, aber die Funktionalität können sie neu festlegen.

TextReader und *TextWriter* werden auch von den Klassen *StringReader* und *StringWriter* implementiert. Dies sind spezielle Klassen, die nicht direkt etwas mit dem Thema Dateizugriffe zu tun haben. Sie ermöglichen das Lesen und Schreiben von Zeichenketten mit der Funktionalität, die durch *TextReader* und *TextWriter*-Klasse vorgegeben wird, bei einem *StringBuilder*-Objekt (mehr dazu in Kapitel 4.8).

10.1.6 Das Prinzip der Arbeitsteilung

Das Beispiel in Listing 10.2 soll nicht nur den Umgang mit der *StreamWriter*-Klasse an einem etwas größeren Beispiel veranschaulichen, es soll auch das Prinzip der Arbeitsteilung hervorheben, das beim Umgang mit Streams eine zentrale Rolle spielt. Bei .NET gibt es zwei Typen von Streams: Basis-Streams (engl. »basestreams«), die direkt auf einem Medium (etwa einer Datei) aufsetzen, und die Pass-Through-Streams. Letztere kann man sich als eine Art »Aufsatz« auf einen Basis-Stream vorstellen, der die vom Basis-Stream gelieferten Daten weiterverarbeitet. Ein Beispiel für einen solchen Pass-Through-Stream[1] ist die *CryptoStream*-Klasse. Deren Konstruktor erwartet ein bereits initialisiertes *Stream*-Objekt, dessen Inhalt verschlüsselt wird.

Auch das Beispiel zeigt, was bei den .NET-Streams mit Arbeitsteilung gemeint ist. Zuerst wird ein *FileStream*-Objekt angelegt. Dieses wird dazu benutzt, jeweils ein *StreamWriter*- und ein *StreamReader*-Objekt anzulegen. Auch wenn man das *FileStream*-Objekt direkt für den Dateizugriff benutzen kann, ist dies die etwas flexiblere und damit häufiger anzutreffende Variante. Das Prinzip der Arbeitsteilung besteht darin, dass sich das *FileStream*-Objekt um den Zugriff auf die Datei kümmert und zum Beispiel die Zugriffsmodalitäten festlegt. Das, was geschrieben oder gelesen werden soll, erhält es über ein *StreamWriter*- oder *StreamReader*-Objekt.

1 Der muss nun nicht mit »passt durch«, sondern mit »Durchreichen« übersetzt werden.

Hier noch einmal die wichtigsten »Erkenntnisse«, die Sie aus Listing 10.2 ziehen sollten:

- Das *FileStream*-Objekt steht zunächst lediglich für den Zugriff auf eine Datei. Beim Anlegen werden der Dateiname sowie ein Zugriffsmodus (Öffnen, Anlegen usw.) angegeben. Eine Übersicht über die zur Auswahl stehenden Modi gibt die Tabelle 10.5.
- Wird kein Verzeichnispfad angegeben, muss sich die Datei im gleichen Verzeichnis wie das Programm befinden. Bei Visual Studio .NET wäre dies das *bin*-Unterverzeichnis im Projektverzeichnis.
- Das eigentliche Speichern bzw. Lesen geschieht über ein *Stream*-Objekt, das vom Dateizugriff vollkommen unabhängig ist. Es ist das *StreamReader*-Objekt (*Lesen*) und das *StreamWriter*-Objekt (Schreiben).
- Um beim Einlesen festzustellen, ob noch Zeichen vorhanden sind, wird die *Peek*-Methode verwendet. Diese gibt das nächste zu lesende Zeichen zurück, ohne es aber tatsächlich zu lesen. Es wird also nur geprüft, ob noch ein Zeichen gelesen werden könnte. Gibt *Peek* -1 zurück, ist kein Zeichen mehr da.
- Am Ende muss (oder sollte) das *FileStream*-Objekt wieder geschlossen werden.

Tabelle 10.5: Die verschiedenen Möglichkeiten, eine Datei zu öffnen

Modi (Konstante)	Bedeutung
FileMode.Append	Es sollen Daten an die Datei angehängt werden. Existiert die Datei nicht, wird sie angelegt.
FileMode.Create	Die Datei wird in jedem Fall neu angelegt. Existiert die Datei bereits, wird sie überschrieben.
FileMode.CreateNew	Die Datei wird neu angelegt. Existiert die Datei bereits, wird eine Ausnahme ausgelöst.
FileMode.Open	Die Datei wird geöffnet, sofern die Datei bereits existiert und der Benutzer die Leseberichtigung besitzt. Existiert die Datei nicht, wird eine Ausnahme ausgelöst.
FileMode.OpenOrCreate	Zuerst wird versucht, die Datei zu öffnen. Existiert sie noch nicht, wird sie angelegt.
FileMode.Truncate	Legt fest, dass eine bereits vorhandene Datei geöffnet werden soll, die dann aber auf 0 Byte Größe gekürzt (engl. »to truncate«) wird. Setzt die Schreibberechtigung des Benutzers voraus.

Damit wissen Sie bereits das Wesentliche über das Thema Dateizugriffe. Wenn Sie die beiden kleinen Beispiele dieses Abschnitts als Grundgerüst verwenden, sollten Sie die meisten einfachen Aufgaben relativ problemlos lösen können.

10.2 Die Rolle der Streams

Bei allen Dateizugriffen unter .NET steht eine Einrichtung direkt oder indirekt im Mittelpunkt, die als *Stream* (zu Deutsch »Strom«) bezeichnet wird. Ein Stream ist ein Strom von Daten, also eine Folge von Bytes oder Zeichen, die von einer Quelle stammen und an ein Ziel transportiert werden. Streams besitzen besondere Eigenschaften:

- Sie besitzen einen Anfang und ein Ende. Es gibt einen internen (unsichtbaren) Positionszeiger, der stets die nächste Lese- und Schreibposition angibt. Dieser kann vom Programm sowohl indirekt durch eine Lese- oder Schreiboperation als auch direkt (*Position*-Eigenschaft) verschoben werden.
- Streams sind temporäre Einrichtungen. Mit dem Anlegen bzw. Öffnen des Streams wird dieser mit Daten gefüllt. Mit dem Schließen des Streams, dem Löschen des Streams oder der Variablen, über die der Stream angesprochen wird, gehen die Daten wieder verloren.
- Viele Methoden geben *Stream*-Objekte zurück oder erwarten diese als Argument. Streams sind damit bei .NET ein universelles »Transportmittel« für unstrukturierte Daten.
- Streams sind stets Spezialisten für eine bestimmte Aufgabe. Es gibt Streams, die für den Inhalt einer Datei ein »Lesegerät« darstellen, das deren Inhalt Textzeichen für Textzeichen liest. Es gibt Streams, die einfach nur Bytes in das mit dem Stream verbundene Medium (etwa eine Datei) schreiben und vieles mehr. Streams sind ein universelles Konzept, das sich für viele Aufgaben einsetzen lässt.

Flexibilität ist derer Grund, warum bei .NET Streams im Vordergrund stehen. Ein Stream steht lediglich für eine Folge von Zeichen, die im Arbeitsspeicher gehalten werden. Wie der Stream weiterverarbeitet wird, geht den Stream nichts an. Er kann in eine Datei geschrieben, auf einem beliebigen Ausgabegerät ausgegeben oder – und das ist besonders flexibel – einem anderen Objekt zugeführt werden, das den Stream weiterverarbeitet. Ein Beispiel wäre die Umwandlung eines kompletten Streams in ein anderes Zeichenformat. Soll ein Stream in einem anderen Zeichenformat gespeichert werden, wird er vor dem Abspeichern einem anderen Objekt zugeführt, das den Stream umwandelt. Da die Streams in einem allgemeinen Format vorliegen ist es für Programmierer (bei .NET betrifft dies vor allem Visual Basic-Pro-

grammierer) kein Problem einen weiteren Filter zu programmieren, der einen Stream auf eine ganz andere Weise weiterverarbeitet.

Ein Stream ist ganz allgemein ein Strom, d.h. eine Aneinanderreihung von Zeichen, die im Arbeitsspeicher gehalten werden. Viele .NET-Klassen arbeiten direkt mit Streams, indem sie sie zum Beispiel in eine Datei schreiben oder in ein anderes Format umwandeln.

In Kapitel 14 geht es um das Thema XML. Dort werden Streams ebenfalls eine zentrale Rolle spielen.

10.2.1 Die Stream-Klasse

Hinter allen Stream-Klassen steht die Klasse *Stream* im Namespace *IO* (IO steht für Input/Output). Es handelt sich um eine sehr allgemeine Klasse, die so allgemein ist, dass sie nicht direkt instanziiert werden kann; sie ist mit *MustInherit* deklariert. Der Befehl

```
Dim obSt As Stream = New Stream()
```

führt daher zu einer Fehlermeldung. Bei *Stream* handelt es sich um eine *abstrakte Basisklasse,* also eine Klasse, bei der für einige oder alle Mitglieder nur die Definition, aber nicht die Implementierung enthalten ist. Die Idee, die hinter der abstrakten Basisklasse *Stream* steckt, ist Vereinheitlichung. Der Programmierer kann sich auf der Grundlage der *Stream*-Klasse seine eigenen Klassen programmieren, die ein beliebiges Ausgabemedium auf beliebige Weise ansprechen. Doch da die neue Klasse die *Stream*-Klasse implementiert hat, ist sie kompatibel zu allen Objekten, deren Eigenschaften oder Methoden ein Objekt vom Typ *Stream* erwarten.

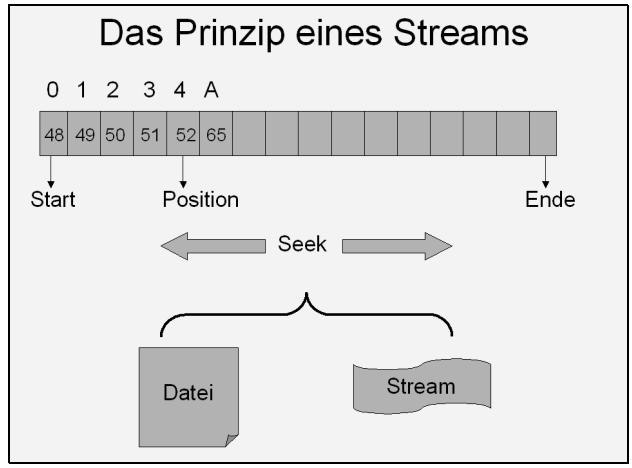


Abbildung 10.1: Das allgemeine Prinzip eines Streams

Allerdings müssen Sie sich nur in Ausnahmefällen die Mühe machen, *Stream* in einer Klasse zu implementieren, denn genau dies ist bereits in den folgenden, von der *Stream*-Klasse abgeleiteten Klassen geschehen:

- *BufferedStream*
- *FileStream*
- *MemoryStream*
- *System.Net.Sockets.NetworkStream*
- *System.Security.Cryptography.CryptoStream*

Bereits die Aufzählung macht deutlich, dass sich die *Stream*-Klasse nicht auf Dateizugriffe beschränkt, sondern ein allgemeines Konzept bei .NET darstellt. Im Folgenden soll lediglich die *FileStream*-Klasse betrachtet werden, da diese natürlich in diesem Kapitel im Mittelpunkt steht.

10.2.2 Die FileStream-Klasse

Die *FileStream*-Klasse implementiert die *Stream*-Klasse für einen Dateizugriff. Allerdings ist die *FileStream*-Klasse sehr allgemein gehalten, sie betrachtet den Dateiinhalt als eine Aneinanderreihung von Bytes. Der Konstruktor der *FileStream*-Klasse ist insgesamt 9-fach überladen:

```
Sub New (IntPtr, FileAccess)
Sub New (String, FileMode)
Sub New (IntPtr, FileAccess, Boolean)
Sub New (String, FileMode, FileAccess)
Sub New (IntPtr, FileAccess, Boolean, Integer)
Sub New (String, FileMode, FileAccess, FileShare)
Sub New (IntPtr, FileAccess, Boolean, Integer, Boolean)
Sub New (String, FileMode, FileAccess, FileShare, _
         Integer)
Sub New (String, FileMode, FileAccess, FileShare, _
         Integer, Boolean)
```

Die am häufigsten verwendete Variante dürfte jene sein, bei der ein Dateiname und ein Zugriffsmodus übergeben wird:

```
oStr = New FileStream(stPfad, FileMode.Open)
```

Bei *FileMode* handelt es sich um eine Enumeration, die folgende Konstanten umfasst: *Append*, *Create*, *CreateNew*, *Open*, *OpenOrCreate* und *Truncate*. Über den *FileAccess*-Parameter kann der Zugriffsmodus bestimmt werden. Die gleichnamige Enumeration umfasst dabei folgende Konstanten: *Read*, *ReadWrite* und *Write*. Für einen schreibgeschützten Zugriff lautet der Aufruf wie folgt:

```
oStr = New FileStream(stPfad, FileMode.Open, _
       FileAccess.Read)
```

Für die Situationen, in denen ein Mehrfachzugriff auf die Datei möglich ist, gibt es den Parameter *FileShare*. Die Enumeration umfasst folgende Werte: *Inheritable*, *None*, *Read*, *ReadWrite* und *Write*. Natürlich dürfen sich die Einstellungen nicht gegenseitig blockieren, d.h. bei einer Read-Modus-gewählten Datei kann für *FileAccess* nicht *Write* gewählt werden.

Die übrigen Parameter sind recht speziell und werden in der Online-Hilfe gut beschrieben. So gibt es den Parameter *useAsync*, der einen Stream asynchron öffnet, was laut Hilfe Performancevorteile verspricht. Auf dieses Thema wird lediglich in Kapitel 10.10 kurz eingegangen.

Die Rolle des Positionszeigers

Stellen Sie sich den nun geöffneten Stream als eine Aneinanderreihung von Bytes vor. Dabei wird die Gesamtlänge über die *Length*-Eigenschaft angegeben. Das Pendant zu dieser Nur-Lesen-Eigenschaft ist die *SetLength*-Eigenschaft, durch die sich ein Stream zum Beispiel verkürzen lässt. Die wichtigsten Mitglieder der *FileStream*-Klasse sind in Tabelle 10.4 zusammengestellt. Stellen Sie sich weiterhin einen unsichtbaren Zeiger vor, der die Position bestimmt, an der das nächste Byte über die *ReadByte*-Methode gelesen oder über die *WriteByte*-Methode geschrieben wird. Die aktuelle Position gibt die *Position*-Eigenschaft an. Diese Eigenschaft kann auch beschrieben werden, sodass sich auf einfache Weise die aktuelle Lese-/Schreibposition festlegen lässt. Eine Alternative, um den Positionszeiger zu verschieben, bietet die *Seek*-Methode. Ihr werden zwei Parameter übergeben: die Anzahl der Stellen, um die der Zeiger verschoben werden soll (hier sind auch negative Werte erlaubt), und die relative Position, auf die sich dieser Offset beziehen soll. Über die *SeekOrigin*-Enumeration stehen drei Werte zur Auswahl: *SeekOrigin.Begin*, *SeekOrigin.Current* und *SeekOrigin.End*.

Das Ende des Streams feststellen

Anders als man es vielleicht vermuten würde, gibt es keine EOF-Eigenschaft, die das Ende des Streams signalisiert. Das Pendant ist die *Position*-Eigenschaft, die dazu einfach mit der Länge des Streams (*Length*-Eigenschaft) verglichen werden muss.

Die folgende kleine Schleife liest alle Zeichen aus dem Stream aus.

```
Do While oStr.Position < oStr.Length
    txtStream.Text += oStr.ReadByte.ToString
Loop
```

Beachten Sie dabei aber, dass *ReadByte* lediglich ein einzelnes Byte liest (das aber in einen *Integer* konvertiert wird), was nicht bedeutet, dass es als Textzeichen vorliegt. In der Textbox *txtStream* erscheinen daher die ANSI-Codes der Textzeichen und nicht die Textzeichen selbst. Die *FileStream*-Klasse ist universell ausgelegt und macht sich nicht die Mühe, den Inhalt für den Programmierer zu interpretieren.

Aber was passiert, wenn die obige Schleife erneut aufgerufen wird? Gar nichts, denn der Positionszeiger befindet sich immer noch am Ende, sodass die Abfrage bereits von Anfang an erfüllt ist. Soll der Inhalt erneut gelesen werden, muss entweder die Position auf 0 gesetzt oder die *Seek*-Methode aktiviert werden. Hat man sich an die Idee eines Streams als eine Aneinanderreihung von Bytes erst einmal gewöhnt, ist im Grunde alles recht einfach.

Einen Block von Bytes in Textzeichen konvertieren

Wurde im letzten Abschnitt behauptet, mit Streams sei alles ganz einfach, so muss dies vorübergehend wieder relativiert werden. Ein Byte zu lesen ist einfach, doch was ist, wenn man einen ganzen Block lesen möchte? Dafür gibt es die *Read*-Methode, die wie folgt aufgerufen wird:

```
Function Read( ByVal array() As Byte, _
               ByVal offset As Integer, _
               ByVal count As Integer ) As Integer
```

Insgesamt erwartet die Methode drei Parameter: ein *Byte*-Array, das mit den gelesenen Bytes gefüllt wird, die Startposition, ab der der Stream gelesen werden soll, und die Anzahl der gelesenen Bytes. Zurückgegeben wird die Anzahl der tatsächlich gelesenen Bytes (falls mehr Bytes gelesen werden sollen, als im Stream noch vorhanden sind). Auch das ist nicht weiter kompliziert, wie der folgende Aufruf zeigt:


```
Dim LesePuffer(oStr.Length) As Byte
Dim AnzahlGelesen As Integer
oStr.Seek(0, SeekOrigin.Begin)
AnzahlGelesen = oStr.Read(LesePuffer, 0, oStr.Length)
```

Zwei Dinge gilt es zu beachten: Der übergebene Puffer muss in der benötigten Größe initialisiert werden und der Positionszeiger muss (bzw. sollte) zu Beginn auf 0 gesetzt werden, da der Offset zwar die Leseposition festlegt, aber nicht dazu führt, dass der Positionszeiger auf diesen Wert gesetzt wird. Befindet sich dieser etwa am Ende des Streams, werden trotz Offset 0 keine Bytes gelesen.

Doch was ist – jetzt kommt die ankündigte Schwierigkeit ins Spiel –, wenn es sich um eine Textdatei handelt und der Inhalt als Text behandelt werden soll? In diesem Fall muss das *Byte*-Array in einen String umgewandelt wer-

den. Dafür käme eine simple Schleife in Frage. Doch eleganter erledigt dies die *GetString*-Methode der *Encoding*-Klasse (Namespace *System.Text*), der man ein *Byte*-Array übergibt und die freundlicherweise einen String zurückgibt. Allerdings muss man den passenden Zeichensatz wählen, was im folgenden Beispiel durch den Default-Zeichensatz elegant und souverän gelöst wird.

```
Dim en As System.Text.Encoding = _
 System.Text.Encoding.Default
txtStream.Text = en.GetString(LesePuffer)
```

»Gewusst wie« ist auch hier das Motto oder etwas verfeinert »Es gibt zu einer nahe liegenden Lösung stets auch eine elegantere Lösung«, das sich wie ein roter Faden durch die .NET-Programmierung zieht.

Quasi zum lockeren Kennenlernen eines *FileStream*-Objekts finden Sie auf der Buch-CD ein kleines Beispielprogramm (*10_Stream.sln* im Verzeichnis *\Quellen\Kapitel10*), das die wichtigsten Eigenschaften eines *FileStream*-Objekts veranschaulichen soll. Es geht von einer Textdatei mit dem Namen *Zahlen.dat* aus, die sich im Anwendungsverzeichnis (also nicht im *bin*-Unterverzeichnis) befinden muss, und die ein paar beliebige Textzeichen enthält.

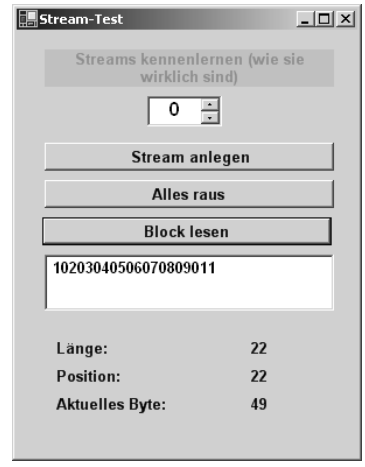


Abbildung 10.2: Das Projekt soll den Umgang mit einem *FileStream* veranschaulichen.

10.3 Binäre Daten schreiben und lesen

Binärdateien sind Dateien, deren Inhalt vom Programm als binäre Daten und nicht als Textdaten interpretiert wird. Der Unterschied ist nicht immer auf Anhieb verständlich, denn ein Textzeichen wird grundsätzlich immer binär gespeichert, anders wäre es gar nicht möglich. Die Unterscheidung wird einzig und allein in dem Programm getroffen, das auf den Dateiinhalt zugreifen möchte. Es muss sich entscheiden, ob es den Inhalt der Datei als eine Aneinanderreihung von Textzeichen betrachtet und Spezialzeichen, wie z.B. das Paar Wagenrücklauf (13)/Neue Zeile (10) gesondert behandelt oder ob alle Bytes gleich sind und sie keine spezielle Bedeutung erhalten. Dies wäre eine Binärdatei. Bei einer Binärdatei wird die Zahl 1234 als Bytefolge 34 12 gespeichert, bei einer Textdatei als Zeichenfolge 1 2 3 4, die gegebenenfalls durch ein 13/10-Paar abgeschlossen wird. Auch bezüglich der Behandlung von Umlauten wird der Unterschied zwischen Text- und Binärdateien deutlich. Bei einer Textdatei ist ein Ä eine Variante des Buchstabens A und wird bei einem Vergleich mit dem Buchstaben B als kleiner eingestuft. Bei einer Binärdatei ist ein und dasselbe Zeichen lediglich eine Zahl (der Zeichencode von Ä), die weit außerhalb des Bereichs der Buchstaben liegt und damit immer als »größer« als jeder Buchstabe eingestuft wird. Dies sind jene Situationen, in denen es auf eine Unterscheidung zwischen Text- und Binärdatei ankommt.

Die .NET-Basisklassen halten für das Schreiben im Textmodus die *TextWriter*-, für das Schreiben im Binärmodus die *BinaryWriter*-Klasse bereit. Es spricht nichts dagegen, auch Texte im Binärmodus zu speichern, nur dass dann zum Beispiel nicht automatisch ein Wagenrücklauf/Neue Zeile-Zeichenpaar angehängt wird.

Das folgende Beispiel speichert die Zahl 1234 zunächst in einer Textdatei:


```
Dim oFi As New FileStream("Zahlen.txt", FileMode.OpenOrCreate)
Dim oSt = New StreamWriter(oFi)
oSt.Writeline ("1234")
oSt.Close()
```

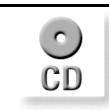
Durch diese Befehlsfolge werden insgesamt 6 Byte gespeichert. Pro Ziffer wird ein Byte gespeichert, auch wenn die interne Darstellung der Zeichen in Unicode erfolgt, wo jedes Zeichen zwei Byte umfasst. Außerdem wird durch *WriteLine* an das Ende ein Wagenrücklauf (13)/Neue Zeile (10)-Paar gehängt, sodass insgesamt 6 Byte gespeichert werden. Würde man stattdessen *Write* verwenden, würden nur 4 Byte gespeichert werden, da hier kein Zeilenende geschrieben wird.

Wird die Zahl dagegen über ein *BinaryWriter*-Objekt binär gespeichert, wird sie als eine 16-Bit-Zahl gespeichert (dazu muss der Typ aber explizit angegeben werden) und belegt somit nur 2 Byte in der Datei:

```
oFi = New FileStream ("Zahlen.dat", _
          FileMode.OpenOrCreate)
Dim oStB As New BinaryWriter(oFi)
oStB.Write (CType(1234, System.Int16))
oStB.Close
```

Das folgende Beispiel fasst die unterschiedlichen Arten, Daten in dieselbe Datei zu schreiben, noch einmal zusammen. Sein »Lerneffekt« besteht darin, dass es im Anschluss an das Schreiben einmal über *StreamWriter* und einmal über *BinaryWriter* die Größe des geschriebenen Streams in Byte anzeigt.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_BinaryWriter.vb.



Listing 10.3: Binär- und Textdateien im direkten Vergleich

```
' ===========================================
' Binärer Zugriff und Textzugriff auf Dateien
' Visual Basic .NET Kompendium
' ===========================================
Option Explicit On
Option Strict On
Imports System
Imports System.Console
Imports System.IO

Class App
  Shared Sub Main()
     Dim obFi As New FileStream("Zahlen.txt", _
         FileMode.OpenOrCreate)
     Dim obSt As New StreamWriter(obFi)
     obSt.Writeline("12345678")
     WriteLine("Dateigröße: {0}", obFi.Length)
     obSt.Close()
     obFi = New FileStream("Zahlen.dat", _
         FileMode.OpenOrCreate)
     Dim oStB As New BinaryWriter(obFi)
     oStB.Write(CType(12345678, System.Int32))
     WriteLine("Dateigröße: {0}", obFi.Length)
     oStB.Close()
     ReadLine()
  End Sub

End Class
```

10.4 Überblick über den System.IO-Namespace

Hinter allen Dateizugriffen stehen die Klassen im *System.IO*-Namespace, die in diesem Abschnitt kurz vorgestellt werden. Die Tabelle 10.6 fasst die wichtigsten Klassen in diesem Namespace zusammen.

Tabelle 10.6: Die wichtigsten Klassen im *System.IO*-Namespace im Überblick

Klasse	Bedeutung
BinaryReader	Liest binäre Daten aus einem Stream.
BinaryWriter	Schreibt binäre Daten in einen Stream. Durch Überschreiben einzelner Methoden kann für Zeichenketten ein individuelles Format festgelegt werden.
Directory	Steht für ein beliebiges Verzeichnis und stellt gemeinsame Methoden zur Verfügung, mit denen das Verzeichnis bearbeitet werden kann.
DirectoryInfo	Stellt zusätzliche Informationen über ein Laufwerk zur Verfügung und ermöglicht den Zugriff auf mehrere Verzeichnisse, etwa alle Unterverzeichnisse in einem Verzeichnis.
File	Steht für eine beliebige Datei auf der Ebene des Dateisystems und stellt gemeinsame Methoden zur Verfügung, mit denen die Datei bearbeitet werden kann. Enthält außerdem Methoden (*CreateText* und *OpenText*), mit denen sich ein *FileStream*-Objekt anlegen lässt.
FileInfo	Stellt zusätzliche Informationen über eine Datei zur Verfügung und ermöglicht den Zugriff auf mehrere Dateien (etwa alle Dateien in einem Verzeichnis).
FileStream	Stellt ein *Stream*-Objekt für den Dateizugriff zur Verfügung, der sowohl synchron (Default) als auch asynchron erfolgen kann.
FileSystemInfo	Stellt die Basisklasse für *FileInfo* und *DirectoryInfo* dar.
FileSystemWatcher	Ermöglicht das Überwachen von Verzeichnissen, sodass z.B. das Anlegen einer neuen Datei ein Ereignis auslöst.
Stream	Allgemeine Klasse für den Zugriff auf Daten, die als eine Folge von Bytes betrachtet werden.
StreamReader	Implementiert eine *TextReader*-Klasse, die den byteweisen Lesezugriff auf einen Stream unter Berücksichtigung einer bestimmten Decodierung ermöglicht.
StreamWriter	Implementiert eine *TextWriter*-Klasse, die den byteweisen Schreibzugriff auf einen Stream unter Berücksichtigung einer bestimmten Codierung ermöglicht.

Klasse	Bedeutung
StringReader	Implementiert eine *TextReader*-Klasse, um die Zeichen aus einem normalen String zu lesen.
StringWriter	Implementiert eine *TextWriter*-Klasse, um die Zeichen eines Strings in einem *StringBuilder*-Objekt zu speichern.
TextReader	Ermöglicht den Lesezugriff auf einen Stream mit Textzeichen.
TextWriter	Ermöglicht den Schreibzugriff auf einen Stream, um Textzeichen zu speichern.

Tabelle 10.6: Die wichtigsten Klassen im *System.IO*-Namespace im Überblick (Forts.)

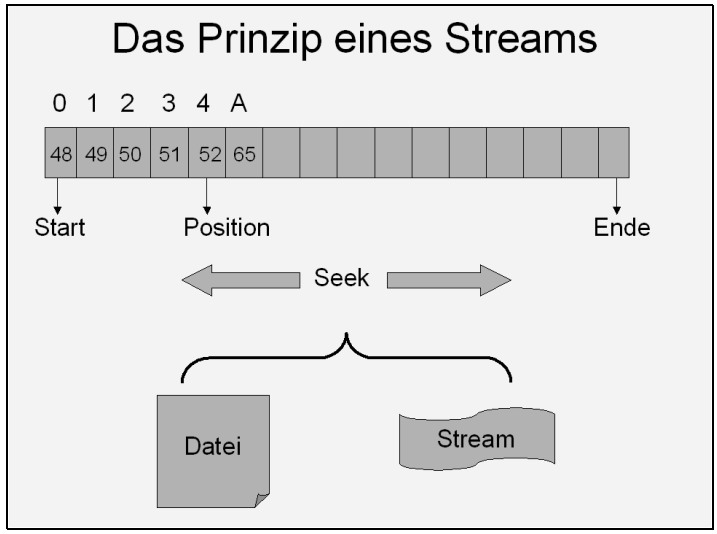


Abbildung 10.3: Ein anderer Blick auf die *System.IO*-Klassen

10.5 Zugriffe auf das Dateisystem

Neben dem Anlegen und Öffnen von Dateien, in denen ein Programm Daten speichert, möchte man auf andere *Dateien* und *Verzeichnisse* zugreifen können, die mit dem eigenen Programm nichts zu tun haben. Dazu gehört z.B. das Anlegen von Verzeichnissen und einiges mehr. Auch für diese Aufgaben stellen die Basisklassen mit *Directory* und *File* die passenden Klassen zur Verfügung.

Die Möglichkeiten der Klassen File *und* Directory *ähneln jenen, die bei früheren Visual Basic-Versionen über das* FileSystemObject*-Objekt der COM-Komponente Scripting Runtime (*Scrrun.dll*) zur Verfügung standen. Dies lässt sich übrigens auch unter Visual Basic .NET nutzen, wenngleich es aus Performancegründen nicht empfohlen wird.*

Das folgende Beispiel veranschaulicht ein paar der Möglichkeiten, die über die Klassen *File* und *Directory* zur Verfügung stehen. Es legt zunächst ein neues Verzeichnis an, wobei ein eventuell vorhandenes Verzeichnis ohne eine Bestätigung einzuholen gelöscht wird. Dafür, dass auch ein Verzeichnis mit Inhalt gelöscht wird, sorgt der zweite Parameter *True*, aber Vorsicht beim Aufruf dieser Methode. Im Beispiel werden in diesem Verzeichnis 255 Dateien angelegt, deren Namen anschließend ausgegeben werden.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_Directory.vb.

Listing 10.4: *File-* und *Directory-*Klasse an einem Beispiel

```
' ============================================
' Beispiel für Directory-Klasse
' Visual Basic .NET-Kompendium
' ============================================
Option Explicit On
Option Strict On

Imports System.Console
Imports System.IO

Class App
  Shared Sub Main()
    Dim stPfad As String = "C:\OneTime"
    If Directory.Exists(stPfad) = True Then
       Directory.Delete(stPfad, True)
    End If
    Directory.CreateDirectory(stPfad)
    WriteLine("Verzeichnis um {0} neu angelegt!", _
     Directory.GetCreationTime(stPfad))
    Write("Weiter mit Taste")
    ReadLine()
    Dim i As Integer
    Dim oFi As FileStream
    For i = 1 To 255
      oFi = File.Create(stPfad & "\Datei" & _
       i.ToString("000") & ".dat")
    Next
    Writeline("255 Dateien um {0} neu angelegt!", _
     File.GetCreationTime(oFi.Name))
    Write("Weiter mit Taste")
    ReadLine()
    Dim aDateien() As String = _
     Directory.GetFiles(stPfad, "*.dat")
    For i = 0 To aDateien.Length -1
       WriteLine(aDateien(i))
    Next
```

```
    Writeline("255 Dateinamen ausgegeben")
   End Sub
End Class
```

10.5.1 Die File-Klasse

Die *File*-Klasse steht für eine einzelne Datei. Die wichtigsten Mitglieder sind in Tabelle 10.4 zusammengestellt. Die meisten Mitglieder sind gemeinsame Mitglieder, die direkt auf die *File*-Klasse angewendet werden.

Das folgende Beispiel gibt den Zeitpunkt aus, an dem die Datei *Msdos.sys* erstellt wurde.

```
File.GetCreationTime("C:\Msdos.sys").ToString()
```

Tabelle 10.7: Interessante Mitglieder der *File*Klasse

Mitglied	Bedeutung
AppendText-Methode	Legt ein *StreamWriter*-Objekt für die angegebene Datei an und fügt den angegebenen Text hinzu.
CreateText-Methode	Legt ein *StreamWriter*-Objekt für die angegebene Datei an und trägt den angegebenen Text ein.
Delete-Methode	Löscht eine Datei.
Exists-Methode	Prüft, ob die angegebene Datei existiert.
GetAttributes-Methode	Gibt die Attribute der angegebenen Datei zurück.
GetCreationTime-Methode	Gibt den Zeitpunkt als *DateTime*-Objekt zurück, an dem die Datei angelegt wurde.
Move-Methode	Verschiebt die Datei in das angegebene Verzeichnis.
Open-Methode	Öffnet die Datei über ein *FileStream*-Objekt.
OpenText-Methode	Öffnet die Datei über ein *StreamReader*-Objekt.
SetAttributes-Methode	Setzt einzelne Dateiattribute.
SetLastAccessTime-Methode	Setzt den Zeitpunkt für den letzten Zugriff auf die Datei.

10.5.2 Die FileInfo-Klasse

Die *FileInfo*-Klasse ergänzt die *File*-Klasse und ist in manchen Fällen eine Alternative. Vor allem dann, wenn Attribute einer Datei angesprochen werden sollen, wie zum Beispiel die Größe, die von der *File*-Klasse nicht zur Verfügung gestellt werden.

Das folgende Beispiel gibt Namen und Größe der ersten Datei im Verzeichnis *C:\Eigene Dateien* aus.


```
Dim oFiInf As New FileInfo(Directory.GetFiles _
 ("C:\Eigene Dateien")(0))
WriteLine("Die Datei {0} ist {1} Bytes groß!", _
 oFiInf.Name, oFiInf.Length)
```

Mit File *und* FileInfo *gibt es zwei scheinbar sehr ähnliche Klassen. Ein Grund dafür ist, dass die statischen Methoden bei* File *bei jedem Aufruf interne Sicherheitsüberprüfungen durchführen, die Instanzenmethoden eines* FileInfo*-Objekts dagegen nicht. Dies ist aus Performancegründen vorteilhafter ist, wenn ein und dasselbe Objekt mehrfach nacheinander verwendet wird.*

Tabelle 10.8: Die interessantesten Mitglieder der *FileInfo*-Klasse

Mitglied	Bedeutung
Attributes-Eigenschaft	Gibt die *FileAttributes* der Datei zurück oder legt sie fest.
CopyTo-Methode	Kopiert die Datei in eine neue Datei.
Director-Eigenschaft	Gibt ein *Directory*-Objekt des übergeordneten Verzeichnisses zurück.
DirectoryName-Eigenschaft	Gibt den kompletten Pfad des übergeordneten Verzeichnisses zurück.
Exists-Methode	Gibt einen *True*-/*False*-Wert zurück, je nachdem, ob die Datei existiert oder nicht.
Extension-Eigenschaft	Gibt die Erweiterung der Datei zurück.
Length-Eigenschaft	Gibt die Größe der Datei zurück.
OpenRead-Methode	Öffnet die Datei als schreibgeschützten *FileStream*.
OpenWrite-Methode	Öffnet die Datei als *FileStream*, der nur über Schreibzugriff verfügt.

10.5.3 Die Directory-Klasse

Die *Directory*-Klasse steht für ein einzelnes Verzeichnis. Die wichtigsten Mitglieder sind in Tabelle 10.9 zusammengestellt. Wie bei der *File*-Klasse handelt es sich auch hier bei den meisten Mitgliedern um gemeinsame Mitglieder.

Das folgende Beispiel legt auf Laufwerk *C:* ein Verzeichnis mit dem Namen »Spezial« an, wenn dieses noch nicht existiert. In beiden Fällen wird der Zeitpunkt über *CreationTime* ausgegeben, an dem das Verzeichnis angelegt wurde.


```
If Directory.Exists("C:\Spezial") = False Then
  Dim oDInf As DirectoryInfo = _
   Directory.CreateDirectory("C:\Spezial")
  WriteLine("Verzeichnis um {0} angelegt!", _
   oDInf.CreationTime.ToString)
Else
  Dim oDInf As New DirectoryInfo("C:\Spezial")
   WriteLine("Verzeichnis wurde bereits " & _
        "um {0} angelegt!", oDInf.CreationTime.ToString)
End If
```

Tabelle 10.9: Interessante gemeinsame Mitglieder der *Directory*-Klasse

Mitglied	Bedeutung
CreateDirectory-Methode	Legt ein neues Verzeichnis an.
Delete-Methode	Löscht ein Verzeichnis, auf Wunsch auch mitsamt allen seinen Dateien und Unterverzeichnissen.
Exists-Methode	Prüft, ob das angegebene Verzeichnis existiert.
GetCurrentDirectory-Methode	Gibt den Pfad des aktuellen Verzeichnisses zurück.
GetFiles-Methode	Gibt die Namen aller Dateien in einem Verzeichnis als String-Array zurück.
Move-Methode	Verschiebt das Verzeichnis mitsamt seinem Inhalt in ein anderes Verzeichnis.

10.5.4 Die DirectoryInfo-Klasse

Ähnlich wie die *FileInfo*-Klasse die *File*-Klasse ergänzt, ergänzt auch die *DirectoryInfo*-Klasse die *Directory*-Klasse. Und wie schon bei *FileInfo* stellt *DirectoryInfo* ausschließlich Instanzenmitglieder zur Verfügung, von denen die interessantesten in Tabelle 10.10 zusammengestellt sind.

Das folgende Beispiel geht alle Dateien im Verzeichnis *C:\OneTime* durch (ändern Sie diesen Pfad gegebenenfalls) und gibt Namen und Größe jeder Datei aus.


```
Dim Dateien() As FileSystemInfo = _
 oDirInf.GetFileSystemInfos()
Dim oFiInf As FileInfo

For Each oFiInf In Dateien
  WriteLine("Die Datei {0} ist {1} Bytes groß!", _
   oFiInf.Name, oFiInf.Length)
Next
```

Tabelle 10.10: Die interessantesten Mitglieder der *DirectoryInfo*-Klasse

Mitglied	Bedeutung
Attributes-Eigenschaft	Gibt die *FileAttributes* des Verzeichnisses zurück oder legt sie fest.
CreationTime-Eigenschaft	Gibt den Zeitpunkt als *DateTime*-Objekt zurück, an dem das Verzeichnis angelegt wurde.
Exists-Methode	Gibt einen *True*-/*False*-Wert zurück, je nachdem, ob das Verzeichnis existiert oder nicht.
FullName-Eigenschaft	Vollständiger Name des Verzeichnisses.
LastWriteTime-Eigenschaft	Gibt den Zeitpunkt des letzten Schreibzugriffs als *DateTime*-Objekt zurück.
Root-Eigenschaft	Verzeichnispfad des Stammverzeichnisses.
Create-Methode	Legt ein neues Verzeichnis an.
CreateSubDirectory-Methode	Legt in dem Verzeichnis ein neues Unterverzeichnis an.
GetFileSystemInfos-Methode	Gibt für alle Dateien in dem Verzeichnis ein Array mit *FileSystemInfo*-Objekten zurück.

10.5.5 Die Path-Klasse

Eine »witzige« Klasse ist die *Path*-Klasse, wobei das Attribut »witzig« lediglich im Sinne von »was es bei .NET nicht alles gibt« verstanden werden soll. Die Klasse enthält knapp ein halbes Dutzend gemeinsamer Felder und ein Dutzend gemeinsamer Methoden, die den Umgang mit Verzeichnispfaden erleichtern. So gibt es eine *GetExtension*-Methode, die die Erweiterung aus einem Pfad zurückgibt, eine *ChangeExtension*-Methode, die eine Erweiterung elegant ändert, sowie eine *GetTempFileName*-Methode, die einen eindeutigen Namen für eine temporäre Datei zurückgibt und bereits eine Datei mit 0 Byte Größe anlegt.

Mitglied	Bedeutung
DirectorySeparatorChar-Feld	Gibt das Zeichen zurück, das auf der Plattform als Trennzeichen in Verzeichnisnamen benutzt wird.
ChangeExtension-Methode	Ändert die Erweiterung einer Datei.
Combine-Methode	Kombiniert zwei Teile eines Verzeichnispfades zu einem neuen Pfad.
GetExtension-Methode	Gibt die Erweiterung einer Datei zurück.
GetFilenameWithoutExtension-Methode	Gibt einen Dateinamen ohne seine Erweiterung zurück.
GetTempFileName-Methode	Gibt den frisch anlegten Namen für eine temporäre Datei zurück.
HasExtension-Methode	Gibt an, ob ein Verzeichnispfad auch eine Erweiterung besitzt.

Tabelle 10.11: Nützliche Mitglieder der *Path*-Klasse

10.6 Arrays in einer Datei speichern und wieder auslesen

Auch wenn es dafür keinen zwingenden Grund gibt, Arrays erfahren in diesem Kapitel eine Sonderbehandlung. Zwar setzt sich ein Array aus vielen gleichartigen Elementen zusammen, aber es wird sowohl beim Abspeichern als auch beim Einlesen als eine Einheit behandelt, was die Programmierung deutlich erleichtert.

Alles, was Sie zum Abspeichern und Laden eines Array an Befehlen benötigen, wurde im Zusammenhang mit den .NET-Basisklassen im *System.IO*-Namespace bereits vorgestellt. Anstelle weiterer Erläuterungen soll das folgende Beispiel das Prinzip (und damit hoffentlich auch alle Fragen) beantworten.

Das folgende Beispiel legt zunächst ein kleines Array an, belegt es mit Werten und speichert diese über ein *FileStream*- und *BinaryWriter*-Objekt in der Datei *Zahlen.dat* ab. Im Anschluss daran werden die gespeicherten Zahlen über ein *BinaryReader*-Objekt wieder gelesen.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_ArrayDatei.vb.

Listing 10.5: Abspeichern und Laden eines kompletten Arrays

```
' ============================================
' Abspeichern und Einlesen eines Arrays
' Visual Basic .NET-Kompendium
' ============================================
Option Explicit On
Option Strict On

Imports System
Imports System.Console
Imports System.IO

Class App

  Shared Sub Main()
    Dim aZahlen As Byte() = {11, 22, 33, 44, 55}
    Dim obFi As New FileStream("C:\Zahlen.dat", _
     FileMode.OpenOrCreate)
    Dim obStW As New BinaryWriter(obFi)
    obStW.Write(aZahlen)
    WriteLine("Array komplett geschrieben - {0} Byte", _
     obStW.BaseStream.Length)
    obStW.Close()
    obFi.Close()
    ReadLine()
    ' Das Array komplett löschen
    Array.Clear(aZahlen, 0, aZahlen.Length)
    ' Datei erneut öffnen
    obFi = New FileStream("C:\Zahlen.dat", _
     FileMode.Open)
    Dim obStR As New BinaryReader(obFi)
    ' 5 Bytes in das Array einlesen
    aZahlen = obStR.ReadBytes(5)
    Dim i As Integer
    ' Array komplett wieder ausgeben
    For i = 0 To aZahlen.Length -1
      Write(" {0} ", aZahlen(i))
    Next
    obStR.Close()
    obFi.Close()
    ReadLine()
  End Sub

End Class
```

10.7 Die Kompatibilitätsfunktionen (zur Entspannung)

Erfahrene Visual Basic-Programmierer, die ihr Basic vielleicht noch aus den guten alten Tagen von QuickBasic, GW-Basic und BasComp[2] kennen und sich gerade mit Visual Basic .NET und damit wieder mit einer völlig neuen Programmierwelt vertraut machen, werden sich vielleicht schon etwas länger fragen, warum alles scheinbar so unnötig kompliziert sein muss. Will (Microsoft-Gründer) Bill Gates damit etwa beweisen, dass man auch in Basic kompliziert programmieren kann[3]? Das ist natürlich nicht der Fall. Hinter den verschiedenen Stream-Klassen steht ein sehr flexibles Konzept, das am Anfang ein wenig kompliziert wirken mag, es aber natürlich nicht ist. Flexibilität ist das, was sich die Mehrheit der Programmierer wünscht und Flexibilität bringt am Anfang eine gewisse Komplexität mit sich. Lassen Sie sich also von den vielen (so viele sind es am Ende doch nicht) Stream-Klassen nicht verwirren. Sie werden das Konzept der verschachtelten Streams noch zu schätzen lernen. Doch der unter Umständen vertraute »Basic-Stil« ist auch bei Visual Basic .NET noch nicht ganz aus der Mode gekommen, denn im Namespace *Microsoft.VisualBasic* hält .NET die gute alten Dateizugriffsbefehle und -funktionen als Methoden bereit. Sie sind zwar nicht 100% syntaxkompatibel, ihren Vorbildern aber sehr ähnlich. Übrigens, diese Funktionen lassen sich auch von einem C#-Programm aus benutzen.

Bei allen »Kompatibilitätsfunktionen« handelt es sich um gewöhnliche Methoden der Klasse FileSystem *im Namespace* Microsoft.VisualBasic. *Der Objektkatalog ist der schnellste Weg, um sich einen kompletten Überblick zu verschaffen. Grundsätzlich spricht nichts dagegen, diese Funktionen weiterhin zu benutzen, denn es erspart oft eine längere Suche in der Klassenreferenz, doch da kein* Stream-*Objekt im Spiel ist, ist dieses Verfahren sehr viel unflexibler, wenn Dateiinhalte an andere Programmteile weitergegeben werden sollen. Daher beschränkt sich ihr Einsatzbereich nüchtern betrachtet auf Kompatibilitätsbrücken.*

Mitglied	Bedeutung
ChDir	Wechselt das aktuelle Verzeichnis.
CurDir	Gibt das aktuelle Verzeichnis zurück.

Tabelle 10.12: Ein paar »Kostproben« aus der *File-System*-Klasse im Namespace *Microsoft.VisualBasic*

2 Der Name eines Basic-Compilers aus den frühen achtziger Jahren.

3 Diese kleine humoristische Anspielung bezieht sich auf den Umstand, dass der kleine Bill in jungen Jahren mit seinem Schulfreund Paul einen Basic-Compiler für einen der ersten Heimcomputer geschrieben und damit den Grundstein für den weltweiten Erfolg einer Programmiersprache gelegt hat, die bei »richtigen« Programmierern nicht immer als optimal angesehen wird.

Tabelle 10.12: Ein paar »Kostproben« aus der *File-System*-Klasse im Namespace *Microsoft.VisualBasic* (Forts.)

Mitglied	Bedeutung
Dir	Gibt nach Übergabe eines Platzhalters den ersten oder nächsten Dateinamen im aktuellen Verzeichnis zurück, der dem Platzhalter entspricht.
EOF	Gibt *True* zurück, wenn das Ende einer Datei erreicht wurde.
FileClose	Schließt eine Datei.
FileCopy	Kopiert eine Datei.
FileGet	Führt einen Binärlesezugriff durch.
FileLen	Gibt die Länge einer Datei zurück.
FileOpen	Öffnet eine Datei.
FilePut	Führt einen Binärschreibzugriff durch.
FreeFile	Gibt die nächste freie Dateikanalnummer zurück.
GetAttr	Gibt die Dateiattribute zurück.
Input	Liest alle Zeichen bis zum nächsten Trennzeichen oder Zeilenende ein.
Kill	Löscht die angegebene Datei (ohne Bestätigung und nicht in den Papierkorb).
MkDir	Legt ein Verzeichnis an.
Print	Schreibt die angegebenen Werte in eine geöffnete Datei.
Rename	Gibt einer Datei einen anderen Namen.
RmDir	Entfernt ein (leeres) Verzeichnis.
Seek	Verschiebt den Positionszeiger für den Dateizugriff oder fragt seine Position ab.
Write	Schreibt in eine geöffnete Datei.

Das folgende Beispiel zeigt, wie sich eine Textdatei traditionell einlesen und anzeigen lässt.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_FileAlt.vb.

Listing 10.6: Die alten Dateizugriffsbefehle in Aktion – Methoden der *FileSystem*-Klasse

```
' ============================================
' Beispiel für die "alten" Dateizugriffsmethoden
' ============================================

Imports Microsoft.VisualBasic
Imports System.Console

Class App

  Shared Sub Main ()
    Dim DateiNr As Integer
    Dim Inhalt As String
    DateiNr = FreeFile()
    FileOpen(DateiNr, "C:\Msdos.sys", OpenMode.Input)
    Inhalt = InputString(DateiNr, LOF(DateiNr))
    WriteLine(Inhalt)
    FileClose(DateiNr)
  End Sub

End Class
```

Das folgende Beispiel schreibt ein paar Zahlen in die Datei *Zahlen.dat*, um sie anschließend wieder auszulesen.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_ArrayKompat.vb.

Listing 10.7: Array abspeichern und einlesen per Kompatibilitätsklasse

```
' ============================================
' Abspeichern und Einlesen eines Arrays
' Visual Basic .NET Kompendium
' ============================================

Option Explicit On
Option Strict Off

Imports System
Imports System.Console
Imports Microsoft.VisualBasic

Class App

  Shared Sub Main()
    Dim Probefeld As Byte() = {11, 22, 33, 44, 55}
    Dim i As Integer
    FileOpen(1, "C:\Zahlen.dat", OpenMode.Binary)
    FilePut(1, Probefeld)
    FileClose(1)
    Array.Clear(Probefeld, 0, Probefeld.Length)
```

```
    FileOpen(1, "C:\Zahlen.dat", OpenMode.Binary)
    FileGet(1, Probefeld)
    FileClose(1)
    For i = 0 To Probefeld.GetUpperBound(0)
      WriteLine(" {0} ", Probefeld(i))
    Next I
    ReadLine()
  End Sub
End Class
```

Das sieht doch aus der Sicht erfahrener Visual Basic-Programmierer schon gleich viel sympathischer und vertrauter aus. Es kommt garantiert kein wie auch immer geartetes *Stream*-Objekt vor, zumindest tritt keines direkt in Erscheinung. Eine besondere Rolle spielt wieder einmal der *Imports*-Befehl:

```
Imports Microsoft.VisualBasic
```

Durch ihn wird es möglich, alle Namen im Namespace *Microsoft.VisualBasic* ohne ständiges Voranstellen des Namespaces zu benutzen.

Das obige Beispiel funktioniert nur bei ausgeschalteter strenger Typenüberprüfung. Ansonsten meckert der Compiler, weil eine implizite Umwandlung eines Byte()*-Typs in einen* System.Array*-Typ nicht erlaubt ist. Abhilfe schafft ein weiteres Array, das explizit vom Typ* Array *deklariert wird und als Zwischenstation beim späteren Einlesen dient:*

```
Dim ax As Array

FileOpen(1, "C:\Zahlen.dat", OpenMode.Binary)
ax = CType(Probefeld, Array)
FileGet(1, ax)
FileClose(1)
Probefeld = CType(ax, Byte())
For i = 0 To Probefeld.GetUpperBound(0)
```

Man kann es daher drehen und wenden, wie man möchte, am Ende bestimmt auch hier .NET die Spielregeln.

10.8 Verzeichnisse überwachen

Das Schöne an den .NET-Basisklassen ist, dass sie relativ vollständig sind und dass bei ihrem Entwurf die Bedürfnisse Programmierer in den Mittelpunkt gestellt wurden. Zudem werden sie von Visual Basic auf die exakt gleiche Weise genutzt wie von C# und all den anderen schönen Programmiersprachen. Ein Thema, das vielen Visual Basic-Programmierern in der Vergangenheit nicht unerhebliche Kopfschmerzen bereitet hat und das sich

unter .NET ganz einfach und vor allem völlig konform zur restlichen Programmierung lösen lässt sind die Überwachungsfunktionen, die Windows automatisch zur Verfügung stellt. Diese ließen sich in der Vergangenheit nur über recht aufwändige API-Programmierung lösen. Diese Überwachungsfunktionen melden immer dann ein Ereignis, wenn in dem zu überwachenden Verzeichnis eine Datei angelegt, gelöscht oder umbenannt wird. Das kann zum Beispiel sehr nützlich sein, um festzustellen, wann Dateien in ein Webverzeichnis geladen werden. Bei .NET gibt es dazu die *FileSystemWatcher*-Klasse, die alle gewünschten Funktionen zur Verfügung stellt und zudem sehr einfach anzuwenden ist.

Auch Visual Studio .NET benutzt eine solche Überwachung und ist dadurch in der Lage zu erkennen, wenn geladene Quelltextdateien außerhalb der IDE verändert wurden.

Das folgende Beispiel richtet eine Verzeichnisüberwachung ein, die bewirkt, dass immer dann eine Meldung ausgegeben wird, wenn eine Datei in dem Verzeichnis *C:\Test* angelegt oder gelöscht wird. Das funktioniert übrigens auch mit einem Netzwerkverzeichnis. Da es sich um eine Konsolenanwendung handelt, bleibt die Überwachung nur so lange bestehen, wie das Programm aktiv ist. Aus diesem Grund sorgt die *ReadLine*-Funktion am Schluss dafür, dass das Programm nicht von alleine endet. Außerdem muss das zu überwachende Verzeichnis existieren, ansonsten ist eine Ausnahme die Folge.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_FileWatcher.vb.

Listing 10.8: eine Überwachung für Verzeichnisse einrichten

```
' ============================================
' Beispiel für eine Verzeichnisüberwachung
' Muss mit /r:System.dll kompiliert werden
' Visual Basic .NET Kompendium
' ============================================

Imports System.IO
Imports System.Console

Class App

  Shared Sub Main()
    Dim oFw As New FileSystemWatcher()
    With oFw
      ' Dieses Verzeichnis überwachen
      ' Auch Netzwerklaufwerke lassen sich überwachen
     .Path = "C:\Test"
      ' muss nicht unbedingt gesetzt werden
```

```
      ' .NotifyFilter = NotifyFilters.Attributes
´     ' Auch Unterverzeichnisse
       .IncludeSubdirectories = True
      ' Alle Dateien
       .Filter = "*.*"
    End With
    AddHandler oFw.Created, _
     New FileSystemEventHandler(AddressOf OnFileMelder)
    AddHandler oFw.Deleted, _
     New FileSystemEventHandler(AddressOf OnFileMelder)
    oFw.EnableRaisingEvents = True


    WriteLine("FileWatch aktiv...")
    ReadLine()
  End Sub

  Shared Sub OnFileMelder(ByVal Source As Object, _
   ByVal e As FileSystemEventArgs)
      WriteLine("Es ist was passiert... " _
       & e.FullPath & ":" & e.ChangeType.ToString)
  End Sub

End Class
```

10.8.1 Kurze Aufmunterung für zwischendurch

Leser, die es bis zu diesem Punkt geschafft haben (bildlich gesehen haben Sie auf dem Weg zum Gipfel bereits die Hälfte des Weges zurückgelegt), sind zwar schon einiges gewohnt, werden sich aber dennoch nach wie vor manches Mal fragen, woher um alles in der Welt man wissen soll, dass es etwa eine *FileSystemWatcher*-Klasse gibt. Auch hier lautet die allgemeingültige Antwort, sich ausführlich mit der Dokumentation des NET. Framework SDK zu beschäftigen. Ebenfalls sehr hilfreich sind der Objektbrowser der Visual Studio .NET-IDE und nicht zuletzt die vielen Beispiele sowohl in der Dokumentation als auch in diesem Buch (oder auf www.nanobooks.net). Man muss sich allerdings die Mühe machen, danach zu suchen und darf nicht erwarten, alles auf dem Präsentierteller hübsch verpackt zu erhalten[4].

4 Insbesondere jene Visual Basic-Programmierer, auf deren System die F1-Taste in der Vergangenheit zu einer Fehlermeldung führte, sollten sich hier angesprochen fühlen.

10.9 Umgang mit unterschiedlichen Zeichensätzen – die Encoding-Klassen

Zum Abschluss dieses Kapitels ein wichtiges Thema, das bisher bei gezeigten Dateizugriffen immer irgendwie über allem »schwebte«, aber nie angesprochen wurde. Es geht um die unterschiedlichen Zeichensätze, die beim Schreiben und Lesen von Textdateien über die Klassen *StreamReader* und *StreamWriter* eine Rolle spielen können. Rufen Sie sich dazu noch einmal die *StreamReader*-Klasse und ihre möglichen Konstruktorvarianten in Erinnerung. Diese ist insgesamt 10-fach überladen. Folgende Variationen stehen zur Auswahl:

Tabelle 10.13: Die Konstruktorvarianten für die *StreamReader*-Klasse im Überblick

Konstruktorvariante	Bedeutung
Sub New (Stream)	Die Instanz wird auf der Basis eines bereits vorhandenen Streams geöffnet.
Sub New (String)	Die Instanz wird mit einem Dateinamen geöffnet.
Sub New (Stream, Boolean)	Der Parameter *Boolean* gibt an, ob die Byte-Reihenfolge (Little-Endian und Big-Endian) automatisch erkannt werden soll.
Sub New (Stream, Enconding)	Über *Enconding* wird der Zeichensatz festgelegt.
Sub New (String, Boolean)	-
Sub New (String, Enconding)	Zusätzlich zum Dateinamen wird ein Zeichensatz angegeben.
Sub New (Stream, Enconding, Boolean)	-
Sub New (String, Enconding, Boolean)	-
Sub New (Stream, Enconding, Boolean, Integer)	Der Parameter *Integer* gibt die Mindestgröße des Puffers an.
Sub New (String, Enconding, Boolean)	-

Bei 6 der insgesamt 10 Varianten kommt ein *Encoding*-Parameter vor, der in den bisherigen Beispielen nicht genutzt wurde. Dieser Parameter steht nicht einfach nur für eine Konstante, sondern für die *Encoding*-Klasse im Namespace *System.Text*. Auch hier handelt es sich um eine abstrakte Basisklasse (*MustInherit*), die in folgenden Klassen implementiert ist:

- *System.Text.ASCIIEncoding*
- *System.Text.UnicodeEncoding*
- *System.Text.UTF7Encoding*
- *System.Text.UT8Encoding*

Standardmäßig arbeitet .NET mit Unicode, einem modernen 16-Bit-Zeichensatz, der international standardisiert ist und praktisch keine Wünsche offen lässt. Oder .NET arbeitet mit UTF-8, bei dem, sofern es nicht erforderlich ist, Zeichen nicht als 16-Bit-, sondern nur als 8-Bit-Einheit gespeichert werden. Das ist auch der Grund, warum die *Encoding*-Klasse in den bisherigen Beispielen nicht vorkam. Doch wie immer gibt es Ausnahmen. Eine solche Ausnahme liegt vor, wenn Daten über ein Netzwerk gesetzt werden sollen und 16 Bit pro Zeichen zu viel sein könnten oder wenn mit Internetanwendungen Daten ausgetauscht werden, die kein Unicode vertragen. Ein anderer Ausnahmefall liegt natürlich vor, wenn MS-DOS-Dateien gelesen (*ASCIIEncoding*) oder geschrieben werden sollen. Diese Dateien dürfen aber nicht mit jenen Textdateien verwechselt werden, die in der Eingabeaufforderung bearbeitet werden. Hier sind auch Umlaute usw. erlaubt, die im reinen ASCII-Zeichensatz nicht enthalten sind. Möchte man diese Dateien korrekt einlesen, muss *Encoding.Default* (ANSI-Zeichensatz) zum Einsatz kommen.

Tabelle 10.14: Die .NET-Zeichensätze im Überblick

Zeichensatz	Kurzportrait
ASCII	Der Uralt-Zeichensatz, der früher unter DOS und Windows zum Einsatz kam. Ursprünglich ein 7-Bit-Code, wurde jedoch fast ausschließlich auf 8 Bit erweitert.
UTF-7	Spielt beim Versenden von Unicode-Text per E-Mail eine Rolle, da viele Mail-Systeme nur 7-Bit-Zeichencodes vertragen (UTF = Universal Character Set Transformation Format).
UTF8	Unicode-Zeichensatz, bei der aber wahlweise 1 oder 2 Byte für die Darstellung eines Zeichens verwendet werden. Wird standardmäßig von *StreamWriter* benutzt.
Unicode	Der Standardzeichensatz bei .NET.

Das folgende Beispiel schreibt zunächst eine kleine Textdatei über eine *StreamWriter*-Klasse, wobei die Default-Codierung (ANSI) benutzt wird. Anschließend wird die Datei über eine *StreamReader*-Klasse einmal »falsch« (UTF8-Codierung) und einmal »richtig« (*Encoding.Default*) eingelesen.


```
' ============================================
' Beispiel für die Encoding-Klasse
' Visual Basic .NET-Kompendium
' ============================================
Imports System.Console
```

```
Imports System.IO
Imports System.Text

Class App

  Shared Sub Main ()
   Dim stPfad As String = "Test.txt"
   Dim oSW As New StreamWriter(stPfad, False, _
       Encoding.Default)
   With oSw
    .WriteLine("Dies ist ein kleiner ANSI-Text")
    .WriteLine("Jetzt kommen ein paar Umlaute:")
    .WriteLine("äääÄÄÄöööÖÖÖüüüÜÜÜßßßß")
    .WriteLine("Vielen Dank für Ihre Aufmerksamkeit")
    .Close
   End With
   WriteLine("Zuerst öffnen mit UTF8-Codierung")
   Dim oSr As New StreamReader(stPfad)
   WriteLine(oSr.ReadToEnd())
   oSr.Close()
   WriteLine("Jetzt öffnen mit ANSI-Codierung")
   oSr = New StreamReader(stPfad, Encoding.Default)
   WriteLine(oSr.ReadToEnd())
   oSr.Close()
  End Sub

End Class
```

10.10 Asynchrones Lesen einer Textdatei

Zum Schluss ein sehr interessantes Thema, das aber aus Platzgründen (und weil die damit einhergehenden Thread-Problematik den Rahmen des Buches sprengen würde) nur kurz angedeutet werden soll. Es geht um das asynchrone Lesen oder Schreiben eines Streams. Asynchron bedeutet, dass ein Stream geöffnet wird und sich ein zweiter Thread um das Lesen und Schreiben kümmert, während sich der Haupt-Thread anderen Dingen widmen kann. Sobald die Arbeit (oder ein Teil davon) erledigt ist, meldet sich der Thread wieder über eine vorher festgelegte Callback-Prozedur.

Das folgende kleine Beispiel dient lediglich dazu, das Prinzip eines asynchronen Lesevorgangs zu demonstrieren und gleichzeitig anzudeuten, dass es bei weitem nicht so kompliziert werden muss, wie es einige Beispiele zu diesem Thema suggerieren könnten.

Sie finden das Beispiel auf der Buch-CD in der Datei 10_AsyncStream.sln.

Listing 10.9: Ein Beispiel für einen asynchronen Lesevorgang

Das Beispiel besteht aus einem Windows-Formular mit einem Button und einer Textbox. Nach Anklicken des Buttons werden die ersten 100 Zeichen aus einer Textdatei (*C:\Msdos.sys*) asynchron gelesen und in der Textbox angezeigt.

```
Imports System.Text
Public Class Form1
  Inherits System.Windows.Forms.Form
  Private LesePuffer(100) As Byte
  Private Fs As FileStream
  Private Sub btnStart_Click _
          (ByVal sender As System.Object, _
           ByVal e As System.EventArgs) Handles btnStart.Click
    Fs = New FileStream("C:\Msdos.sys", FileMode.Open, _
     FileAccess.Read, FileShare.ReadWrite, 100, True)
    Dim AsMelder As Object
    Fs.BeginRead(LesePuffer, 0, 100, _
     AddressOf Fertig, AsMelder)
  End Sub

  Sub Fertig(ByVal Ar As System.IAsyncResult)
     If Ar.IsCompleted = True Then
        Fs.EndRead(Ar)
        txtLesePuffer.Text = _
         Encoding.Default.GetString(LesePuffer)
     End If
  End Sub
End Class
```

Das asynchrone Lesen wird mit der *BeginRead*-Methode eingeleitet. Etwas später wird die vorher vereinbarte Callback-Prozedur *Fertig* aufgerufen, der ein Parameter vom Typ *IAsyncResult* übergeben wird. Über dessen *IsCompleted*-Eigenschaft lässt sich feststellen, ob der Vorgang beendet wurde oder ob weitere Daten bereitstehen und der Puffer unter Umständen neu initialisiert werden muss. Ist *IsCompleted=True* sollte der Vorgang über die *EndRead*-Methode beendet werden. Bei kleinen Dateien ist dieser Lesemodus viel zu aufwändig und daher keine Alternative, aber bei sehr großen Dateien sollen sich laut Hilfe deutliche Performancegewinne ergeben.