

---

## Ein- und Ausgabe über Datenströme in C#

---

Objektorientiert (mit Vererbung) implementieren

---

### Inhaltsverzeichnis

---

|       |                                                                                                |    |
|-------|------------------------------------------------------------------------------------------------|----|
| 1     | Einleitung .....                                                                               | 2  |
| 2     | Datenpersistenz .....                                                                          | 2  |
| 3     | Datenströme (engl. stream).....                                                                | 3  |
| 3.1   | Byte-Datenströme .....                                                                         | 3  |
| 3.1.1 | Datentransfer: Programm (Variable) zur Senke (externer Datenspeicher).....                     | 3  |
| 3.1.2 | Datentransfer: Quelle (externer Datenspeicher) zum Programm (Variable).....                    | 4  |
| 3.2   | Wichtige Operationen der Stream-Objekte .....                                                  | 4  |
| 3.3   | Wichtige Methoden und Eigenschaften der Klasse: System.IO.Stream.....                          | 4  |
| 3.3.1 | Eigenschaften (Properties) .....                                                               | 4  |
| 3.3.2 | Methoden.....                                                                                  | 5  |
| 3.4   | Beispiel zum generellen Aufbau einer Datenstromumsetzung mit FileStream .....                  | 5  |
| 4     | Verwendung der Klasse: FileStream .....                                                        | 7  |
| 4.1   | Erzeugung eines neuen Datenstroms mit FileStream.....                                          | 7  |
| 4.1.1 | Erzeugung des Datenstromes ohne FileAccess-Modus .....                                         | 7  |
| 4.1.2 | Erzeugung eines neuen Datenstromes mit FileAccess-Modus .....                                  | 8  |
| 4.1.3 | Generelle Hinweise zur Erzeugung eines neuen Datenstroms .....                                 | 9  |
| 4.2   | Schliessen eines Datenstromes .....                                                            | 9  |
| 4.2.1 | Schliessen eines Datenstromes mittels der Methode: Close() .....                               | 9  |
| 4.2.2 | Schliessen eines Datenstromes nach Verlassen des „using-Blocks“ .....                          | 10 |
| 4.2.3 | Schliessen eines Datenstromes mit Ausnahmebehandlung .....                                     | 11 |
| 5     | Speicherung von Daten beliebigen Datentyps.....                                                | 13 |
| 5.1   | Schreiben und Lesen von Textdateien .....                                                      | 13 |
| 5.1.1 | Schreiben von Texten mittels StreamWriter-Objekt in eine Datei .....                           | 14 |
| 5.1.2 | Lesen von Texten mittels StreamReader-Objekt aus einer Datei .....                             | 15 |
| 5.1.3 | Beispiel zum Schreiben und Lesen von Textdateien mit der Einstellung:<br>Encoding.Default..... | 15 |
| 5.1.4 | Beispiel zum Schreiben und Lesen von Textdateien mit der Einstellung:<br>Encoding.UTF8 .....   | 16 |
| 5.1.5 | Unterschied zwischen einem StreamWriter und einem BinaryWriter .....                           | 17 |
| 5.2   | Schreiben und Lesen von Binärdateien.....                                                      | 17 |
| 5.2.1 | Schreiben von Binärwerten in eine Datei.....                                                   | 17 |
| 5.2.2 | Lesen von Binärwerten aus einer Datei.....                                                     | 18 |
| 5.2.3 | Beispiel zum Schreiben und Lesen von unterschiedlichen Datentypen .....                        | 18 |

|       |                                                                                                   |    |
|-------|---------------------------------------------------------------------------------------------------|----|
| 6     | Binäre Serialisierung von Objekten.....                                                           | 20 |
| 6.1   | Generelles zur Objektserialisierung .....                                                         | 20 |
| 6.1.1 | Erzeugung eines BinaryFormatter-Objektes für die Objektserialisierung.....                        | 21 |
| 6.1.2 | Schreiben von Objekten mittels BinaryFormatter-Objekt in eine Binärdatei.....                     | 21 |
| 6.1.3 | Lesen von Objekten aus einer Binärdatei .....                                                     | 22 |
| 6.2   | Beispiele wie Objekte in eine Binärdatei gespeichert werden können .....                          | 22 |
| 6.2.1 | Objektserialisierung von Objekten, welche in einem Array gespeichert sind .....                   | 23 |
| 6.3   | Objektserialisierung von Objekten, welche in einer Liste gespeichert sind.....                    | 24 |
| 6.4   | Objektserialisierung von Objekten, welche in einer ObservableCollection<br>gespeichert sind ..... | 25 |

## 1 Einleitung

---

Praktisch jedes Programm muss Daten aus externen Quellen einlesen und/oder Verarbeitungsergebnisse in externen Datenspeichern (Senken) ablegen. Bis zum jetzigen Zeitpunkt haben wir uns nur auf die Eingabe per Tastatur sowie die Ausgabe per Bildschirm beschränkt. Da es auch noch alternative Quellen und Senken gibt (z.B. Dateien, Netzwerkverbindungen, Datenbankserver usw.), wollen wir nun den Zugriff auf diese in diesem Dokument behandeln.

Wir beschränken uns in diesem Dokument auf einfache Verfahren, um Werte elementarer Typen (z.B. int, double), Zeichenfolgen (String) oder beliebige Objekte in Dateien zu schreiben bzw. von dort zu lesen. Wesentliche Teile der erlernten Techniken werden aber sinngemäss auch in der Netzwerkprogrammierung verwendbar sein.

## 2 Datenpersistenz

---

In der Informatik wird zwischen persistenter und volatiler Datenhaltung unterschieden. Persistenz ist hierbei die Eigenschaft, Daten auch über die Laufzeit eines Programmes oder Systems zu speichern. Häufig wird Persistenz auch einfach als "nicht flüchtige Datenspeicherung" definiert.

Ein Beispiel für persistente Datenhaltung ist die Festplatte. Die gespeicherten Daten werden auch nach Herunterfahren des Computers gespeichert.

Ein Gegenbeispiel ist der Halbleiterspeicher (RAM). Dieser ist flüchtig (volatil). Sobald der Speicher nicht mehr mit Strom versorgt wird, sind die zuvor gespeicherten Daten nicht mehr verfügbar.

### 3 Datenströme (engl. stream)

**Merke:**



**Unter einem so genannten Datenstrom verstehen wir ein Objekt, das byteorientierte Schreib- und Leseoperationen zwischen einer Anwendung und einem Speichermedium gestattet.**

Das Speichermedium kann im konkreten Fall unterschiedlichster Natur sein, wie zum Beispiel eine USB-Stick, eine Festplatte oder auch der Verbindungsendpunkt in einem Netzwerk.

Jede Klasse, die einen Datenstrom repräsentiert, ist per Definition von der Klasse: `System.IO.Stream` abgeleitet. Die Stream-Klasse (z.B. `FileStream`) sowie alle von ihr abgeleiteten Klassen stellen eine allgemeine Sicht auf Datenbehälter dar, der Anwender wird mit ihrer Hilfe von den speziellen Details bestimmter Ein- und Ausgabemedien isoliert. Die Stream-Klasse selbst ist abstrakt, aus diesem Grund legt sie im Gegensatz zu ihren Spezialisierungen noch kein konkretes Medium für die Ein- bzw. Ausgabe fest.

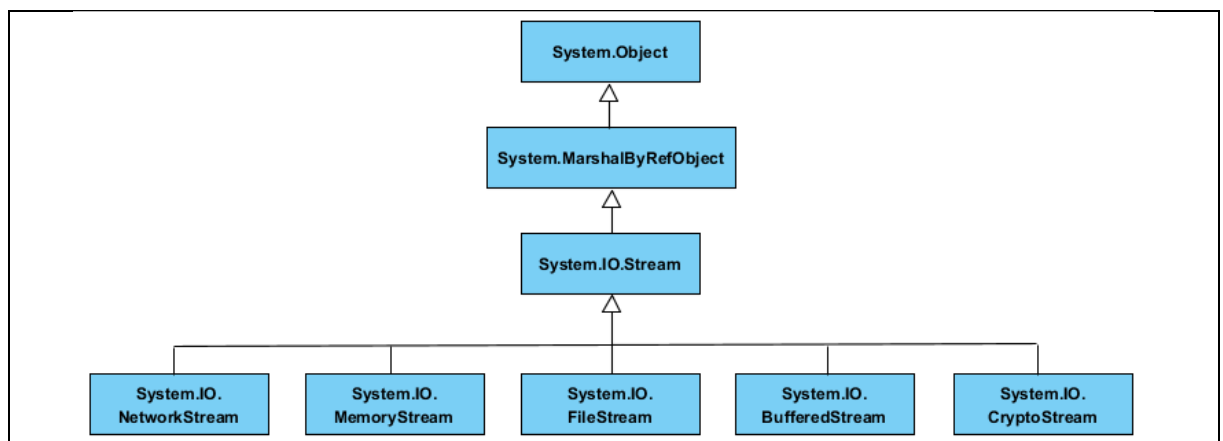


Abbildung 1: Klassenübersicht der unterschiedlichen Stream-Klassen

Die .NET - Klassen zur Datenein- und Datenausgabe befinden sich im Namensraum: `System.IO`, den wir folglich bei Quellcodedateien mit entsprechender Funktionalität in der Regel zu Beginn importieren:

```
using System.IO;
```

#### 3.1 Byte-Datenströme

Der Transfer von Daten erfolgt im Normalfall über Bytes. Wie dies genau funktioniert soll im Folgenden aufgezeigt werden:

##### 3.1.1 Datentransfer: Programm (Variable) zur Senke (externer Datenspeicher)

Ein Programm schreibt Daten in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Ausgabegerät, Netzverbindung):

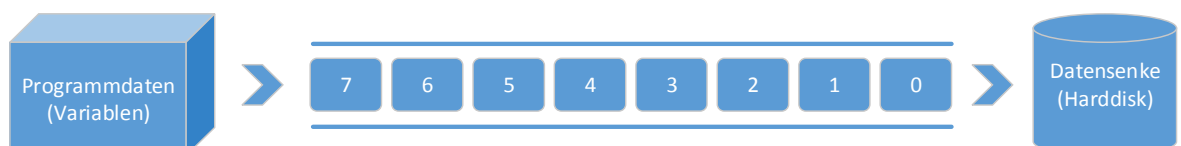


Abbildung 2: Datentransfer: Programm (Variable) zur Senke (externer Datenspeicher)

### 3.1.2 Datentransfer: Quelle (externer Datenspeicher) zum Programm (Variable)

Ein Programm liest Daten aus einem Eingabestrom, der aus einer Datenquelle (z.B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:

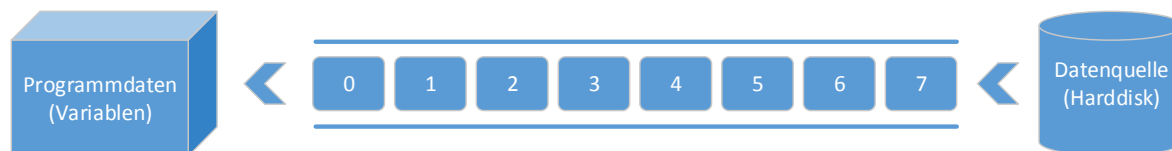


Abbildung 3: Datentransfer: Quelle (externer Datenspeicher) zum Programm (Variable)

## 3.2 Wichtige Operationen der Stream-Objekte

- **Lesen**

Ein Lesevorgang an einem Stream-Objekt bewirkt einen Datentransfer vom Speichermedium in einen Speicherbereich der Anwendung, typischerweise in ein Bytearray.

- **Schreiben**

Ein Schreibvorgang an einem Stream-Objekt bewirkt einen Datentransfer von der Anwendung hin zum Datenmedium. In der Regel sind die Daten in der Anwendung wiederum in einem Bytearray aufzubereiten.

- **Positionieren**

Unter der Positionierung versteht man die Möglichkeit, den sogenannten Positionszeiger, den jedes Stream-Objekt verwaltet, wahlfrei verändern zu können. Unter dem Positionszeiger kann man sich einen Index vorstellen, der in der zu transferierenden Datenmenge eine bestimmte Stelle identifiziert. Die Veränderung des Positionszeigers ist nicht für jedes Speichermedium sinnvoll oder gar durchführbar. Zum Beispiel unterstützen Netzwerkströme das Positionieren überhaupt nicht.

Die Fähigkeiten des Lesens, Schreibens und Positionierens werden nicht von allen Datenströmen gleichermassen unterstützt. Zu diesem Zweck gibt es in der Stream-Klasse die drei Eigenschaften: CanRead, CanWrite und CanSeek, mit deren Hilfe eine Anwendung einen Datenstrom konkret nach seinen Fähigkeiten befragen kann.

## 3.3 Wichtige Methoden und Eigenschaften der Klasse: System.IO.Stream

### 3.3.1 Eigenschaften (Properties)

|                                               |                                                                                                                                |
|-----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------|
| <code>public long Position {get; set;}</code> | Über dieses Property wird die aktuelle Position im Stream angesprochen, an der das nächste Byte gelesen bzw. geschrieben wird. |
| <code>public long Length {get;}</code>        | Mit diesem Property wird die Länge des Streams (z.B. die Dateigrösse) in Bytes angesprochen.                                   |
| <code>public bool CanRead {get;}</code>       | Gibt an, ob der aktuelle Datenstrom Lesevorgänge unterstützt.                                                                  |
| <code>public bool CanSeek {get;}</code>       | Gibt an, ob der aktuelle Datenstrom die Möglichkeit unterstützt, den Positionszeiger zu verändern.                             |
| <code>public bool CanWrite {get;}</code>      | Gibt an, ob der aktuelle Datenstrom Schreibvorgänge unterstützt.                                                               |

### 3.3.2 Methoden

|                                                                                                                                                                                                                                                                                                                                                                                                                                                                  |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>public int ReadByte()</b>                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| Mit dieser Methode wird ein Stream-Objekt aufgefordert, ein Byte per Rückgabewert vom Typ: <code>int</code> zu liefern und seine Position entsprechend zu erhöhen. Ist das Ende des Streams erreicht, wird keine Ausnahme geworfen, sondern der Rückgabewert -1 geliefert.                                                                                                                                                                                       |
| <b>public int Read(byte[] buffer, int offset, int count)</b>                                                                                                                                                                                                                                                                                                                                                                                                     |
| Mit dieser Methode wird ein Stream-Objekt aufgefordert, Anzahl: <code>count</code> Bytes zu liefern, im Byte-Array: <code>buffer</code> ab Position: <code>offset</code> abzulegen und seine Position entsprechend zu erhöhen. Als Rückgabewert erhält man die Anzahl der tatsächlich gelieferten Bytes, die bei unzureichendem Vorrat kleiner als <code>count</code> ausfallen kann.                                                                            |
| <b>public void WriteByte()</b>                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| Mit dieser Methode wird ein Stream aufgefordert, ein Byte zu schreiben und seine Position entsprechend zu erhöhen.                                                                                                                                                                                                                                                                                                                                               |
| <b>public void Write(byte[] buffer, int offset, int count)</b>                                                                                                                                                                                                                                                                                                                                                                                                   |
| Mit dieser Methode wird ein Stream-Objekt aufgefordert, Anzahl: <code>count</code> Bytes zu schreiben, die im Byte-Array: <code>buffer</code> ab Position: <code>offset</code> liegen, und seine Position entsprechend zu erhöhen.                                                                                                                                                                                                                               |
| <b>public void Flush()</b>                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Viele Stream-Objekte verwenden beim Schreiben aus Performancegründen einen Puffer, um die Anzahl der zeitaufwendigen Zugriffe auf eine angeschlossene Senke (z.B. Datei) möglichst gering zu halten. Mit der Methode: <code>Flush()</code> verlangt man die sofortige Ausgabe des Puffers, so dass der komplette Inhalt für Abnehmer zur Verfügung steht. Bei der Klasse: <code>FileStream</code> wird eine voreingestellte Puffergrösse von 4096 Bytes benutzt. |
| <b>public void Close()</b>                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| Schliesst einen geöffneten Datenstrom.                                                                                                                                                                                                                                                                                                                                                                                                                           |
| <b>public long Seek(long offset, SeekOrigin origin)</b>                                                                                                                                                                                                                                                                                                                                                                                                          |
| Diese Methode fordert einen Stream auf, seine Position relativ zu einem per <code>SeekOrigin</code> -Wert festgelegten Bezugspunkt: <code>origin</code> ( <code>Begin</code> , <code>Current</code> , <code>End</code> ) neu zu setzen. Als Rückgabe erhält man die neue Position.                                                                                                                                                                               |
| Beispiel:                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| <code>fs.Seek(-4, SeekOrigin.Current); //go back 4 bytes from crurrent place</code>                                                                                                                                                                                                                                                                                                                                                                              |
| <b>public abstract void SetLength (long value);</b>                                                                                                                                                                                                                                                                                                                                                                                                              |
| Legt die Länge des aktuellen Datenstroms fest.                                                                                                                                                                                                                                                                                                                                                                                                                   |

### 3.4 Beispiel zum generellen Aufbau einer Datenstromumsetzung mit FileStream

Beim nachfolgenden Programm soll das Byte-Array: `byteArrayWrite` in einem File mit dem Namen: `byteArray.bin` abgespeichert und anschliessen wieder in das Byte-Array: `byteArrayRead` zurückgelesen werden. Anschliessend wird das soeben erzeugte File: `byteArray.bin` wieder gelöscht.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace Byte_Array_im_File_sichern_V1_W0d
{
    class Program
    {
        static void Main(string[] args)
        {
            // define path and file name
            string fileName = @"j:\Daten\byteArray.bin";

            // define and initialize arrays
            byte[] byteArrayWrite = { 200, 201, 202, 203, 204, 205, 206, 207 };
            byte[] byteArrayRead = new byte[byteArrayWrite.Length];

            //create file stream
            FileStream fs = new FileStream(fileName, FileMode.Create);

            // write array to file
            fs.Write(byteArrayWrite, 0, byteArrayWrite.Length); // array name,
                                                                // start index,
                                                                // length of array

            //read from file
            fs.Position = 0; // set start position
            fs.Read(byteArrayRead, 0, byteArrayRead.Length); // read file values

            // output: values of byte array
            for (int count = 0; count < byteArrayRead.Length; count++){
                Console.Write (byteArrayRead[count] + ", ");
            }

            // close filestream
            fs.Close();

            // delete file
            File.Delete(fileName);
        }
    }
}
```

Codebeispiel 1: Schreiben und lesen eines Byte-Arrays

Ausgabe:


200, 201, 202, 203, 204, 205, 206, 207,

Hinweis:

Das File: byteArray.bin wird im Visual Studio kurzzeitig unter dem Pfad: Programmname\bin\Debug abgelegt bis es dann vom Programm selbst gelöscht wird. Am besten kann dies festgestellt werden, wenn der Befehl: File.Delete(fileName) kurzzeitig deaktiviert wird.

## 4 Verwendung der Klasse: FileStream

Mit einem FileStream-Objekt können Bytes aus einer Datei gelesen oder hineingeschrieben werden.

|                                                                                                        |                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br> | <b>Ein FileStream-Objekt beherrscht prinzipiell beide Transportrichtungen, kann aber auch auf unidirektionalen Betrieb eingestellt werden.</b> |
|--------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------|

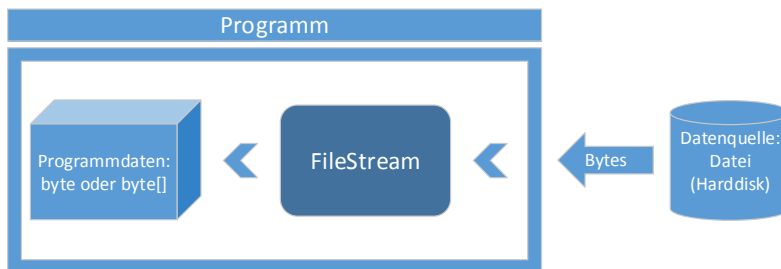


Abbildung 4: Lesen von einer Datei

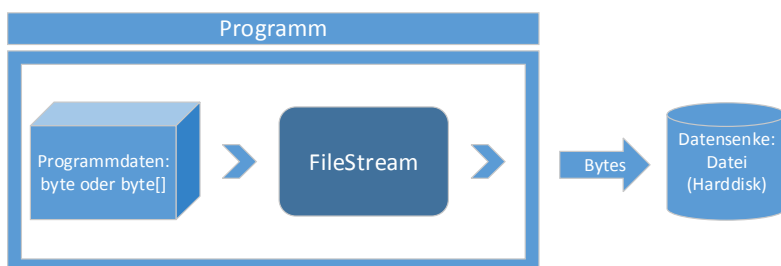


Abbildung 5: Schreiben in eine Datei

FileStream-Objekte verwenden einen Puffer, um die Anzahl der Zugriffe auf die angeschlossene Datei möglichst gering zu halten. Beim Schliessen einer Datei (Verlassen des using-Blocks oder Close()-Aufruf (siehe weiter unten)) werden gepufferte Schreibvorgänge automatisch ausgeführt. Die voreingestellte Puffergrösse beträgt 4096 Bytes. Dieser kann aber durch einen bestimmten Parameter bei gewissen Konstruktoren übersteuert werden.

### 4.1 Erzeugung eines neuen Datenstroms mit FileStream

#### 4.1.1 Erzeugung des Datenstromes ohne FileAccess-Modus

Erzeugung und Initialisierung einer neuen Instanz der Klasse: FileStream mit dem angegebenen Pfad und dem angegebenen Erstellungsmodus.

```
FileStream fileStreamName = new FileStream(fileName, FileMode.Modus);
```

Codeumsetzung 1: Erzeugung und Initialisierung eines Datenstroms

| Parameter:       | Beschreibung:                                                                                                                                                                       |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>fileName:</b> | Mit dem Filenamen wird die Bezeichnung des zu schreibenden oder lesenden Files angegeben. Im Weiteren ist es auch möglich, neben dem Filenamen auch den Pfad der Datei festzulegen. |
| <b>FileMode:</b> | Beim Erstellen eines FileStream-Objekts kann man den Öffnungsmodus über einen Wert des Enumerationstyps: FileMode wählen. Folgende Möglichkeiten stehen zur Verfügung:              |

|                     |                                                                                                                                                                                          |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Append</b>       | Die Datei wird geöffnet oder neu erzeugt. Beim diesem Öffnungsmodus befindet sich der Dateizeiger am Ende der Datei (hinter dem letzten Byte), und es ist kein lesender Zugriff möglich! |
| <b>Create</b>       | Ist die Datei noch nicht vorhanden, wird sie angelegt (wie bei CreateNew), anderenfalls wird sie überschrieben (wie bei Truncate).                                                       |
| <b>CreateNew</b>    | Es wird eine neue Datei angelegt, oder eine IOException geworfen, falls eine Datei mit dem gewünschten Namen bereits existiert.                                                          |
| <b>Open</b>         | Es wird eine vorhandene Datei geöffnet, oder eine IOException geworfen, falls keine Datei mit dem angegebenen Namen existiert.                                                           |
| <b>OpenOrCreate</b> | Es wird eine neue Datei erzeugt oder eine vorhandene geöffnet, jedoch im Unterschied zu Create nicht automatisch überschrieben.                                                          |
| <b>Truncate</b>     | Es wird eine vorhandene Datei geöffnet und entleert, oder eine IOException geworfen, falls keine Datei mit dem gewünschten Namen existiert.                                              |

```
using System.IO;

// path and file name definition
string fileName = @"j:\Daten\byteArray.bin";

//create file stream
FileStream fs = new FileStream(fileName, FileMode.Create);
```

Codebeispiel 2: Erzeugung und Initialisierung eines Datenstroms

#### 4.1.2 Erzeugung eines neuen Datenstromes mit FileAccess-Modus

Auf einen FileStream kann auf drei verschiedene Arten zugegriffen werden. Der Zugriff wird dabei über den Parameter: FileAccess des nachfolgenden Konstruktors festgelegt:

```
FileStream fileStreamName = new FileStream(fileName, FileMode.Modus,
                                           FileAccess.AccessModus);
```

Codeumsetzung 2: Erzeugung und Initialisierung eines Datenstroms mit Filezugriffsangabe

Bei diesem Parameter handelt es sich um einen Enumerationsdatentyp, der folgende Werte annehmen kann:

- `FileAccess.Read` → FileStream wird zum **Lesen** geöffnet
- `FileAccess.Write` → FileStream wird zum **Schreiben** geöffnet und
- `FileAccess.ReadWrite` → FileStream wird zum **Lesen & Schreiben** geöffnet.

```
using System.IO;

// path and file name definition
string fileName = @"j:\Daten\byteArray.bin";

//create file stream
fs = new FileStream(fileName, FileMode.Create, FileAccess.ReadWrite);
```

Codebeispiel 3: Erzeugung und Initialisierung eines Datenstroms mit Filezugriffsangabe



#### 4.1.2.1 Zugriffsmöglichkeiten auf das erstellte FileStream-Objekt

Bitte beachten Sie, dass nicht alle obenstehenden Möglichkeiten bei jedem FileMode zur Verfügung stehen. Folgende Tabelle gibt einen Überblick, welche Kombinationen erlaubt sind:

| Öffnungsmodus (FileMode): | Erlaubte FileAccess-Werte: | Default Einstellung: |
|---------------------------|----------------------------|----------------------|
| Append                    | Write                      | Write                |
| Create                    | Write, ReadWrite           | ReadWrite            |
| CreateNew                 | Write, ReadWrite           | ReadWrite            |
| Open                      | Read, Write, ReadWrite     | ReadWrite            |
| OpenOrCreate              | Read, Write, ReadWrite     | ReadWrite            |
| Truncate                  | Write, ReadWrite           | ReadWrite            |

#### 4.1.3 Generelle Hinweise zur Erzeugung eines neuen Datenstroms

- Die Klasse: `FileStream` besitzt diverse Konstruktoren zur Erzeugung einer Datenstrominstanz. Weitere Hinweise zu möglichen Konstruktoren finden sie unter:

[https://msdn.microsoft.com/de-de/library/system.io.filestream\(v=vs.110\).aspx](https://msdn.microsoft.com/de-de/library/system.io.filestream(v=vs.110).aspx)

- Die Pfadangabe für den Filenamen kann auch ohne das Zeichen: `@` definiert werden wie folgendes Beispiel zeigt:

```
string fileName = "j:\\Daten\\byteArray.bin";
```

Diese Art der Pfadangabe kommt von der Programmiersprache ANSI C hervor und stellt somit eine eher veraltet Art dar und sollte somit nicht mehr verwendet werden.

#### 4.2 Schliessen eines Datenstromes

Es ist wichtig Datenströme so früh wie möglich zu schließen, um (exklusive) Zugriffe durch andere Interessenten zu ermöglichen. So kann z.B. eine geöffnete Datei nicht geschlossen werden, bevor der Datenstrom beendet wird.

##### 4.2.1 Schliessen eines Datenstromes mittels der Methode: `Close()`

```
fileStreamName.Close(); // Schliesst den Datenstrom
```

Codeumsetzung 3: Schliessen eines Datenstroms mittels der Methode: `Close()`

```
// close file stream  
fs.Close();
```

Codebeispiel 4: Schliessen eines Datenstroms mittels der Methode: `Close()`

Mit der Methode: `Close()` wird ein Datenstrom geschlossen, was in den von Stream abgeleiteten Klassen folgende Massnahmen beinhaltet:

- Wenn ein Puffer vorhanden ist (z.B. bei der Klasse: `FileStream`), wird er durch einen automatischen Aufruf der Methode: `Flush()` entleert.
- Betriebssystem-Ressourcen (z.B. Datei-Handles, Netzwerk-Sockets) werden freigegeben.

|                               |                                                                                                                                                                                                                                                                                                               |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b><i>Nach einem Close()-Aufruf existiert das angesprochene .NET – Objekt weiterhin, doch führen Lese- bzw. Schreibversuche zu Ausnahmefehlern. Rufen Sie Close() also nicht auf, wenn solche Ausnahmefehler möglich sind, weil im Programm noch weitere Referenzen auf das Stream-Objekt existieren.</i></b> |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

## 4.2.2 Schliessen eines Datenstromes nach Verlassen des „using-Blocks“

Mit der using-Anweisung (nicht zu verwechseln mit der using-Direktive zum Importieren eines Namensraums) kann man einen Block definieren und Objekte erzeugen, die nur innerhalb des using-Blocks gültig (referenziert) sind. Beim Verlassen des Blocks werden automatisch alle verwendeten Ressourcen (Dateien, Netzwerkverbindungen) freigegeben.

|                               |                                                                                                                                   |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b><i>Per using-Anweisung wird das Schliessen eines Stroms elegant gelöst, ohne die Methode: Close() verwenden zu müssen.</i></b> |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------|

```
using (FileStream fileStreamName = new FileStream(fileName, FileMode.Modus))
{
    ...
}
```

Codeumsetzung 4: Schliessen eines Datenstroms mittels einem using-Block

```
...
//create and close file stream at the end of this block
using (FileStream fs = new FileStream(fileName, FileMode.Create))
{
    // write array to file
    fs.Write(byteArrayWrite, 0, byteArrayWrite.Length); // array name,
                                                         // start index,
                                                         // length of array

    //read from file
    fs.Position = 0; // set start position
    fs.Read(byteArrayRead, 0, byteArrayRead.Length); // read file values

    // output: values of byte array
    for (int count = 0; count < byteArrayRead.Length; count++)
    {
        Console.WriteLine(byteArrayRead[count] + ", ");
    }
}

// delete file
File.Delete(fileName);
...
```

Codebeispiel 5: Schliessen eines Datenstroms mittels einem using-Block

### 4.2.3 Schliessen eines Datenstromes mit Ausnahmebehandlung

In manchen Fällen ist es sinnvoll bzw. erforderlich, die von Ein-/Ausgabe – Methoden zu erwartenden Ausnahmen zu behandeln. Hierzu verwenden wir die try- und catch- Anweisungen.

*Hinweis:*

*Im Dokument: „Exception Handling in C#“ werden wir diesen Bereich genauer anschauen.*

#### 4.2.3.1 Ausnahmebehandlung ohne using-Block

```
...
// create file stream and initialize with null
FileStream fs = null;

//
try
{
    fs = new FileStream(fileName, FileMode.Create);    // initialise file stream

    // write array to file
    fs.Write(byteArrayWrite, 0, byteArrayWrite.Length);

    //read from file
    fs.Position = 0;                                // set start position
    fs.Read(byteArrayRead, 0, byteArrayRead.Length);  // read file values

    // output: values of byte array
    for (int count = 0; count < byteArrayRead.Length; count++)
    {
        Console.Write(byteArrayRead[count] + ", ");
    }
}

catch (Exception e)
{
    if (fs != null)
    {
        fs.Close();
    }
}

try
{
    // delete file
    File.Delete(fileName);
}
catch{}
...
```

*Codebeispiel 6: Schliessen eines Datenstromes mit Ausnahmebehandlung ohne using-Block*

#### 4.2.3.2 Ausnahmebehandlung mit using-Block

Mit Hilfe der using-Anweisung lässt sich obenstehendes Programm kürzen. Das Verhalten wird nicht verändert.

```
...
try
{
    using (FileStream fs = new FileStream(fileName, FileMode.Create))
    {
        // write array to file
        fs.Write(byteArrayWrite, 0, byteArrayWrite.Length);
        //read from file
        fs.Position = 0; // set start position
        fs.Read(byteArrayRead, 0, byteArrayRead.Length); // read file values

        // output: values of byte array
        for (int count = 0; count < byteArrayRead.Length; count++)
        {
            Console.Write(byteArrayRead[count] + ", ");
        }
    }

    // delete file
    File.Delete(fileName);
}

catch (Exception e)
{
    Console.WriteLine(e.Message);
}
...
```

*Codebeispiel 7: Schliessen eines Datenstromes mit Ausnahmebehandlung mit using-Block*

## 5 Speicherung von Daten beliebigen Datentyps

Bisher haben wir uns auf das Schreiben und Lesen von Byte-Datentypen beschränkt. In diesem Abschnitt wollen wir jetzt Verfahren kennenlernen, wie auch andere Datentypen (z.B.: int, double, string, Objekte, Strukturen) in ein File gespeichert oder von dort gelesen werden können.

Bei den Dateien auf einem Computer werden folgende zwei Grundtypen unterschieden:

- **Textdateien**

Diese Dateien können von uns mit Hilfe eines Texteditors gelesen und/oder bearbeitet werden, sofern der Texteditor die beim Erstellen der Datei verwendete Zeichenkodierung versteht. Per Programm lassen sich numerische Daten und Zeichenfolgen leicht in eine Textdatei schreiben, doch ist das Lesen numerischer Daten aus einer Textdatei mit erhöhtem Aufwand verbunden.

New features:

- \* Export nodes data into a CSV file.
- \* Compatibility with new Eaton UPSs: 9PX6KSP, 9PX8KSP, 9PX10KSP.
- \* SNMPv3 support for IPP data acquisition.
- \* IPP administrator has the possibility to definitely delete the old events from IPP database in order to reduce size and improve performance.

Abbildung 6: Beispiel einer Textdatei

- **Binärdateien**

In einer Binärdatei werden Daten im Wesentlichen genauso dargestellt wie im Arbeitsspeicher eines Computers, sodass Programme wenig Mühe haben, Daten beliebigen Typs in eine Binärdatei zu schreiben oder aus einer Binärdatei mit bekanntem Aufbau zu lesen. Für uns Menschen ist eine Binärdatei zum Lesen allerdings nicht geeignet. Öffnet man sie mit einem Texteditor, ist nur eine wirre Folge von Zeichen und Sonderzeichen zu sehen.

```
pMx1ZMqGAqAUIQce_____
qMx1ZMqGAqAUIQcdqMx1ZMqGAqAUIQcc_____
_____qMx1ZMqGAqAUIQcb_____qMx
```

Abbildung 7: Beispiel einer Binärdatei

### 5.1 Schreiben und Lesen von Textdateien

Wie Sie in den vorhergehenden Abschnitten gesehen haben, stellt die Klasse: Stream Operationen bereit, mit denen Sie unformatierte Daten byteweise lesen und schreiben können. Stream-Objekte bieten sich daher insbesondere für allgemeine Operationen an, beispielsweise für das Kopieren von Dateien. Die Klasse Stream beziehungsweise die daraus abgeleiteten Klassen sind aber weniger gut für textuelle Ein- und Ausgabeoperationen geeignet.

**Merke:**

!

**Um in eine Textdatei zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen: StreamWriter bzw. StreamReader, die jeweils über eine Instanzvariable mit einem FileStream-Objekt verbunden sind.**

Um den üblichen Anforderungen von Textoperationen zu entsprechen, stellt die .NET-Klassenbibliothek die beiden abstrakten Klassen: TextReader und TextWriter bereit. Objekte, die aus der Klasse Stream abgeleitet werden, unterstützen den vollständigen Satz an

E/A-Operationen, also sowohl das Lesen als auch das Schreiben. Nun wird die Bearbeitung auf zwei Klassen aufgeteilt, die entweder nur lesen oder nur schreiben können.

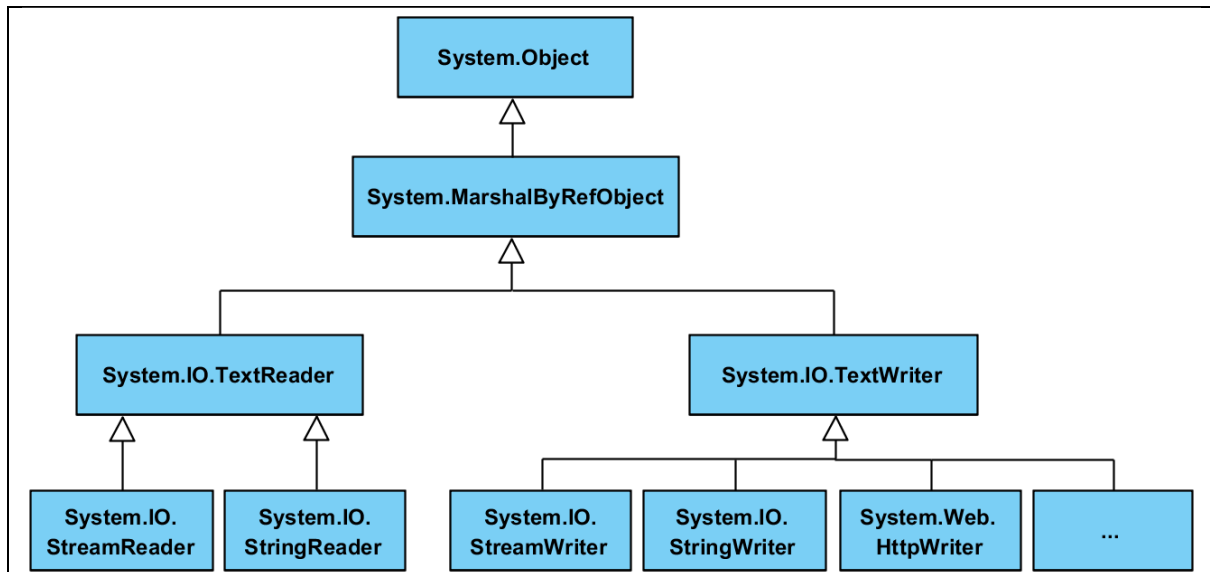


Abbildung 8: Klassenübersicht der Reader- und Writer-Klassen

|                               |                                                                                                                                                                             |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b>Mit einem TextWriter-Objekt kann man die Zeichenfolge von Variablen beliebigen Typs ausgeben.</b><br><b>Das Gegenstück TextReader liest stets eine Zeichenfolge ein.</b> |
|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 5.1.1 Schreiben von Texten mittels StreamWriter-Objekt in eine Datei

```
StreamWriter streamWriterObject = new StreamWriter(fileName);
```

Codeumsetzung 5: Erzeugung eines StreamWriter-Objektes

```
// create StreamWriter-object
StreamWriter sw = new StreamWriter(fileName);
```

Codebeispiel 8: Erzeugung eines StreamWriter-Objektes

Schreibt man per StreamWriter Texte in eine Datei, entsteht folgende Verarbeitungskette:

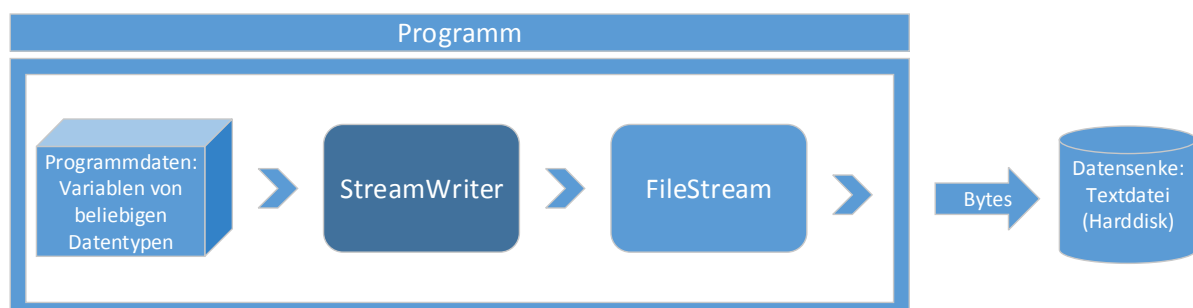


Abbildung 9: Verarbeitungskette beim Schreiben von Texten in eine Datei

### 5.1.2 Lesen von Texten mittels StreamReader-Objekt aus einer Datei

```
StreamReader streamReaderObject = new StreamReader(fileStreamObject);
```

Codeumsetzung 6: Erzeugung eines StreamReader-Objektes

```
// create StreamReader-object  
StreamReader sr = new StreamReader(new FileStream(fileName, FileMode.Open,  
                                           FileAccess.Read), Encoding.Default);
```

Codebeispiel 9: Erzeugung eines StreamReader-Objektes

Werden Texte per StreamReader von einer Datei gelesen, so entsteht folgende Verarbeitungskette:

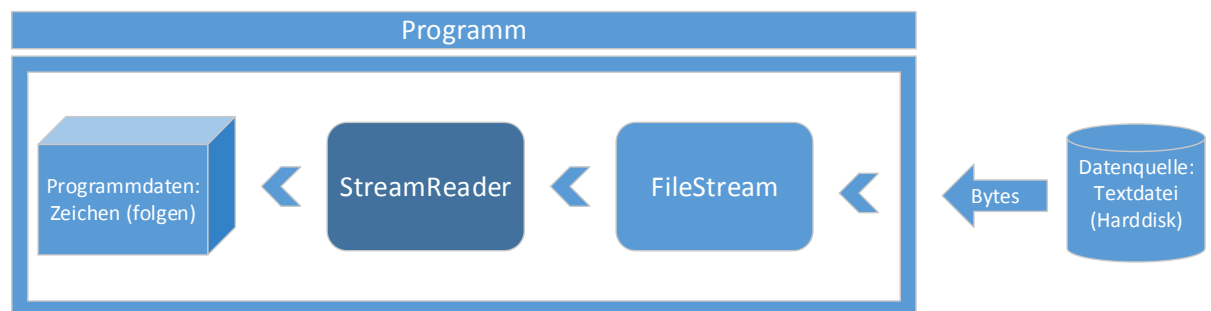


Abbildung 10: Verarbeitungskette beim Lesen eines Textes von einer Datei

### 5.1.3 Beispiel zum Schreiben und Lesen von Textdateien mit der Einstellung: Encoding.Default

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace StreamWriter_zum_Speichern_im_File_V1_W0d
{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = "textValues.txt";

            // create StreamWriter-object
            StreamWriter sw = new StreamWriter(fileName);

            sw.WriteLine(1234);
            sw.WriteLine(5.678);
            sw.WriteLine("Hello friends; Special symbols: ä ö ü ");
            sw.Close();

            // create StreamReader-object
            StreamReader sr = new StreamReader(new FileStream(fileName, FileMode.Open,
                                                             FileAccess.Read), Encoding.Default);

            // Note: text correct but wrong presentation of special symbols

            Console.WriteLine("Content of File: \"{0}\"\\":",
                              ((FileStream)sr.BaseStream).Name);
```

```
for (int count = 0; sr.Peek() >= 0; count++)
{
    Console.WriteLine("{0}:\t{1}", count, sr.ReadLine());
}
sr.Close();
}
```

Codebeispiel 10: Schreiben und Lesen von Texten mit den Klassen: StreamWriter und StreamReader sowie der Einstellung: Encoding.Default

Ausgabe:

```
Content of File: "C:\Users\Werner.WOTEC\Desktop\StreamWriter zum Spei-
chern im File_V1_WOd\StreamWriter zum Speichern im File_V1_WOd\bin
\Debug\textValues.txt":
0:      1234
1:      5.678
2:      Hello friends; Special symbols: Ã ¢ Ã ¢ Ã ¢
```

#### 5.1.4 Beispiel zum Schreiben und Lesen von Textdateien mit der Einstellung: Encoding.UTF8

Wie wir aus obigem Beispiel erkennen können, wird der Text selbst korrekt dargestellt. Die speziellen Symbole hingegen, werden falsch übersetzt. Dies hat etwas mit dem „Encoding“ zu tun. Verwenden wir wie oben angegeben die Einstellung: Encoding.Default geschieht dieser Fehler. Möchte man diesen Fehler beheben, so muss man nur das „Encoding“ auf: Encoding.UTF8 einstellen wie untenstehendes Beispiel zeigt:

```
...
// Note: text and special symbols are correct
StreamReader sr = new StreamReader(new FileStream(fileName, FileMode.Open,
    FileAccess.Read), Encoding.UTF8);
...
```

Codebeispiel 11: Schreiben und Lesen von Texten mit den Klassen: StreamWriter und StreamReader sowie der Einstellung: Encoding.UTF8

Ausgabe:

```
Content of File: "C:\Users\Werner.WOTEC\Desktop\StreamWriter zum Spei-
chern im File_V1_WOd\StreamWriter zum Speichern im Fi-
le_V1_WOd\bin\Debug\textValues.txt":
0:      1234
1:      5.678
2:      Hello friends; Special symbols: ä ö ü
```

Hinweis:

Per Voreinstellung schreiben bzw. lesen die StreamWriter bzw. -Reader Unicode-Zeichen unter Verwendung der platz sparenden UTF8-Kodierung. Sie können also die Angabe: Encoding.UTF8 auch weglassen.



### 5.1.5 Unterschied zwischen einem StreamWriter und einem BinaryWriter

Im Unterschied zu einem BinaryWriter (siehe nächsten Abschnitt) besitzt ein StreamWriter einen lokalen Puffer (Datentyp: char[], voreingestellte Größe: 1024). Daher muss ein StreamWriter unbedingt nach Gebrauch per Close() (oder Dispose()) geschlossen werden. Das zugrunde liegende Stream-Objekt wird dabei automatisch ebenfalls geschlossen.

## 5.2 Schreiben und Lesen von Binärdateien

|                           |                                                                                                                                                                                                                                                                                                          |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><b>!</b> | <b>Um Werte von einem beliebigen elementaren Datentyp (z.B. int, double) sowie Zeichenfolgen in einen binär organisierten Strom (z.B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse: BinaryWriter bzw. BinaryReader.</b> |
|---------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

*Hinweis:*

*Diese beiden Klassen stammen nicht von der Klasse: Stream ab sondern von der Klasse: Object.*

|                           |                                                                                                                                                             |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><b>!</b> | <b>Im Unterschied zu den „bidirektionalen“ Stream-Klassen sind für das Schreiben bzw. Lesen von elementaren Datenwerten „gerichtete“ Klassen zuständig.</b> |
|---------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|

### 5.2.1 Schreiben von Binärwerten in eine Datei

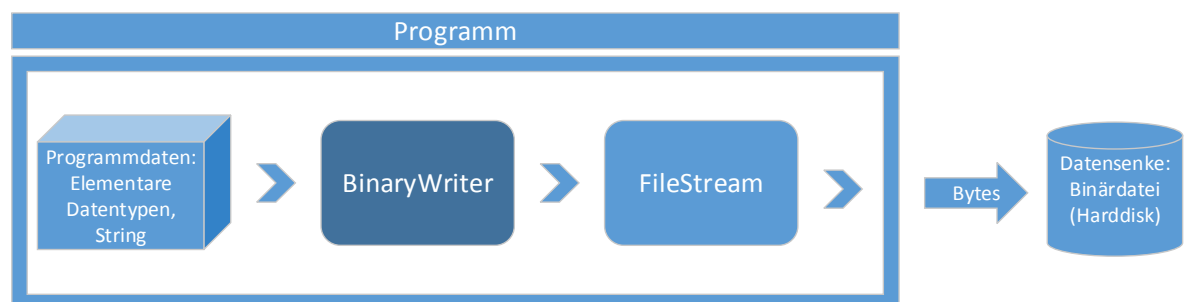
```
BinaryWriter binaryWriterObject = new BinaryWriter(fileStreamObject);
```

*Codeumsetzung 7: Erzeugung eines BinaryWriter-Objektes*

```
// create BinaryWriter-object  
BinaryWriter bw = new BinaryWriter(fileStreamWrite);
```

*Codebeispiel 12: Erzeugung eines BinaryWriter-Objektes*

Schreibt man per BinaryWriter in eine Datei, entsteht folgende Verarbeitungskette:



*Abbildung 11: Verarbeitungskette beim Schreiben von Binärwerten in eine Datei*

## 5.2.2 Lesen von Binärwerten aus einer Datei

```
BinaryReader binaryReaderObject = new BinaryReader(fileStreamObject);
```

Codeumsetzung 8: Erzeugung eines BinaryReader-Objektes

```
// create BinaryReader-object  
BinaryReader br = new BinaryReader(fileStreamRead);
```

Codebeispiel 13: Erzeugung eines BinaryReader-Objektes

Werden Binärdaten von einer Datei gelesen, so entsteht folgende Verarbeitungskette:

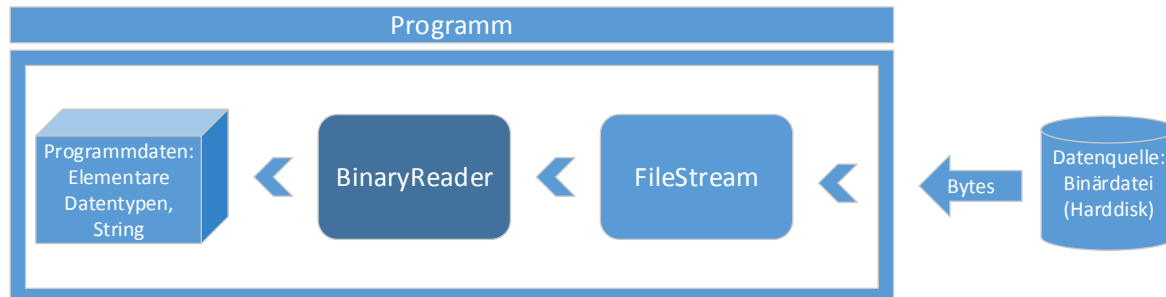


Abbildung 12: Verarbeitungskette beim Lesen der Binärwerte von einer Datei

## 5.2.3 Beispiel zum Schreiben und Lesen von unterschiedlichen Datentypen

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;

namespace BinaryWriter_zum_Speichern_im_File_V1_W0
{
    class Program
    {
        static void Main(string[] args)
        {
            string fileName = @"j:\Daten\binaryValues.bin";

            // write process
            try {
                using (FileStream fileStreamWrite = new FileStream(fileName,
                                                                    FileMode.Create))
                {
                    // create BinaryWriter-object
                    BinaryWriter bw = new BinaryWriter(fileStreamWrite);

                    // writing binary information's to the file
                    bw.Write(1234);           // int Value
                    bw.Write(5.678);         // double value
                    bw.Write("Hello friends"); // string value
                }
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
            }
        }
    }
}
```

```
// read process
try {
    using (FileStream fileStreamRead = new FileStream(fileName, FileMode.Open,
                                                    FileAccess.Read))
    {
        // create BinaryReader-object
        BinaryReader br = new BinaryReader(fileStreamRead);

        // reading binary information's from the file
        Console.WriteLine(br.ReadInt32() + "\n" + br.ReadDouble() + "\n" +
                          br.ReadString());
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
}
```

Codebeispiel 14: Schreiben und Lesen unterschiedlicher Datentypen mit den Klassen: BinaryWriter und BinaryReader

Ausgabe:

```
1234
5.678
Hello friends
```

## 6 Binäre Serialisierung von Objekten

|                               |                                                                                                                                                                                                                                                                                                                          |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b><i>Die Serialisierung ist in der Informatik eine Abbildung von strukturierten Daten auf eine sequenzielle Darstellungsform. Serialisierung wird hauptsächlich für die Persistierung von Objekten in Dateien und für die Übertragung von Objekten über das Netzwerk bei verteilten Softwaresystemen verwendet.</i></b> |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Sämtliche Daten eines Objektes werden durch die Attribute der Klassen beschrieben. Ein Anwender interessiert sich nicht für diese Details. Er arbeitet mit den Daten, manipuliert sie und erwartet ein fehlerfreies Laufzeitverhalten. Dazu zählt auch, dass nach dem Schliessen und dem späteren Neustart des Programms exakt der Zustand wieder hergestellt wird, den ein Objekt vor dem Schliessen hatte.

Die Daten einer Anwendung werden in verschiedenen Datentypen bereitgestellt. Doch welche sind notwendig, um ein bestimmtes Objekt wiederherzustellen? Zwangsläufig müssen das nicht alle sein, denn ein Objekt könnte auch Daten enthalten, die spezifisch für die aktuelle Laufzeitumgebung sind und nach dem erneuten Starten der Anwendung keine Bedeutung mehr haben.

Alle gespeicherten Daten sind einem bestimmten Datentyp zuzuordnen. Wenn der Inhalt des Memberattributs: name eines Objekts der Klasse: Client gesichert wird, darf dieser Wert nach dem Neustart nicht dem Attribut: name eines Objekts von der Klasse: Supplier zugeordnet werden.

Das .NET-Framework hilft uns bei dieser Problematik. Die Technologie, die sich dahinter verbirgt, wird als Serialisierung bezeichnet.

|                               |                                                                                                                                                                                                                                                                                 |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b><i>Die Serialisierung ist ein Prozess mit der Fähigkeit, ein im Hauptspeicher befindliches Objekt in ein bestimmtes Format zu konvertieren und in eine Datei zu schreiben. Das schliesst auch die Rekonstruktion der Objekte in ihrem ursprünglichen Format mit ein.</i></b> |
|-------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Die Serialisierung ist ein Prozess, der automatisch abläuft und bei dem der Name der Anwendung, der Name der Klasse und die Daten-Member eines Objekts binär gespeichert werden. Dadurch wird die spätere Rekonstruktion in einer exakten Kopie möglich.

### 6.1 Generelles zur Objektserialisierung

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können in C# Objekte tatsächlich in der Regel genau so einfach wie Variablen von elementarem Typ in einen Datenstrom geschrieben bzw. von dort gelesen werden. Die keinesfalls triviale Übersetzung eines Objekts mit all seinen Instanzvariablen und den enthaltenen (d.h. von Instanzvariablen referenzierten) Objekten in einen Bytestrom bezeichnet man recht treffend als Objektserialisierung. Beim Einlesen werden alle Objekte mit ihren Instanzvariablen wiederhergestellt und die Referenzen zwischen den Objekten in den Ausgangszustand gebracht.

|                               |                                                                                                                                                                                                                                                                                |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Merke:</b><br><br><b>!</b> | <b><i>Das Serialisieren der Instanzen eines Typs muss explizit bei der Definition über das Typ-Attribut: Serializable erlaubt werden. Beachten Sie bitte, dass das Attribut: Serializable nicht vererbbar ist und somit nicht auf abgeleitete Klassen übertragen wird.</i></b> |
|-------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Bei Bedarf können einzelne Felder über das Attribut: `NonSerialized` ausgeschlossen werden. Dies kommt z.B. dann in Frage, wenn:

- ein Feld aus Sicherheitsgründen nicht in den Ausgabestrom gelangen soll,
- ein Feld temporäre Daten enthält, so dass ein Speichern überflüssig bzw. sinnlos ist,
- ein Feld einen nicht-serialisierbaren Datentyp hat.

Wird ein solches Feld nicht von der Serialisierung ausgeschlossen, kommt es unter Umständen zu einer `SerializationException`.

Die Umsetzung der (De)Serialisierung übernimmt eine Instanz aus einer Klasse, die das Interface: `IFormatter` (Namensraum: `System.Runtime.Serialization`) implementiert.

Am einfachsten geht dies mit der Klasse: `BinaryFormatter` (Namensraum: `System.Runtime.Serialization.Formatters.Binary`), die ein kompaktes Binärformat verwendet.

### 6.1.1 Erzeugung eines `BinaryFormatter`-Objektes für die Objektserialisierung

```
IFormatter binaryFormatterObject = new BinaryFormatter();
```

*Codeumsetzung 9: Erzeugung eines `BinaryFormatter`-Objektes*

```
// create BinaryFormatter-object  
IFormatter bf = new BinaryFormatter();
```

*Codebeispiel 15: Erzeugung eines `BinaryFormatter`-Objektes*

### 6.1.2 Schreiben von Objekten mittels `BinaryFormatter`-Objekt in eine Binärdatei

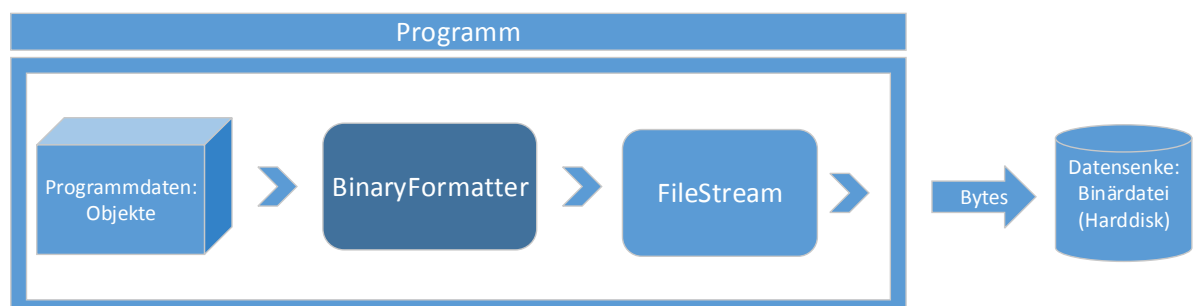
```
binaryFormatterObject.Serialize(fileStreamObject, objects);
```

*Codeumsetzung 10: Schreiben von Objekten unter Zuhilfenahme eines `BinaryFormatter`-Objekts in eine Datei*

```
// write BinaryFormatter-object  
bf.Serialize(fs, clients);
```

*Codebeispiel 16: Schreiben von Objekten unter Zuhilfenahme eines `BinaryFormatter`-Objekts in eine Datei*

Werden Objekte in eine Datei geschrieben, so entsteht folgende Verarbeitungskette:



*Abbildung 13: Verarbeitungskette beim Schreiben von Objekten in eine Binärdatei*

### 6.1.3 Lesen von Objekten aus einer Binärdatei

```
binaryFormatterObject.Deserialize(fileStreamObject);
```

Codeumsetzung 11: Lesen von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts aus einer Datei

```
// Reads an array of objects from a binary file  
Client[] recClients = (Client[]) bf.Deserialize(fs);
```

Codebeispiel 17: Lesen von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts aus einer Datei

Werden Objekte aus einer Datei gelesen, so entsteht folgende Verarbeitungskette:

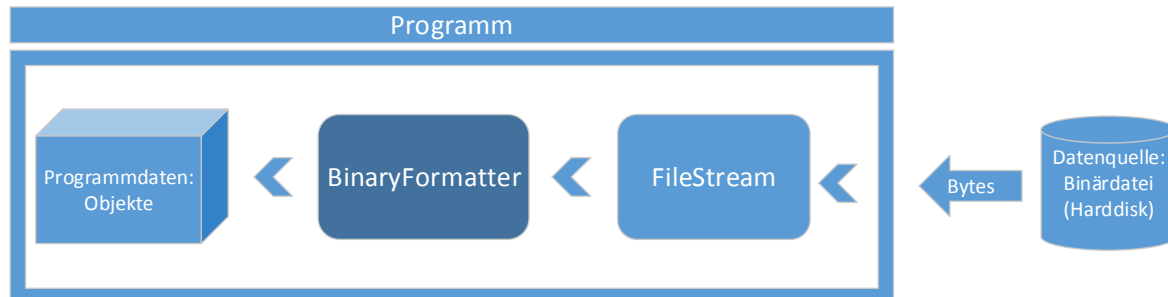


Abbildung 14: Verarbeitungskette beim Lesen von Objekten aus einer Binärdatei

## 6.2 Beispiele wie Objekte in eine Binärdatei gespeichert werden können

Bei den nachfolgenden Beispielen sollen immer wieder Objekte der Klasse: Client in eine Binärdatei abgespeichert werden. Die Klasse: Client ist dabei wie folgt definiert:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Runtime.Serialization;

namespace Objekte_in_File_speichern_V1_W0d
{
    [Serializable]
    class Client
    {
        private int clientNumber;
        private string firstName;
        private string surName;
        private double factor;
        [NonSerialized]
        int regionNumber;

        public Client(int pClientNumber, string pFirstName, string pSurName,
            double pFactor, int pRegionNumber)
        {
            clientNumber = pClientNumber;
            firstName = pFirstName;
            surName = pSurName;
            factor = pFactor;
            regionNumber = pRegionNumber;
        }

        public void printClient()
        {

```

```
        Console.WriteLine("Client number: " + clientNumber);  
        Console.WriteLine("First Name: " + firstName);  
        Console.WriteLine("Surname: " + surName);  
        Console.WriteLine("Factor: " + factor);  
        Console.WriteLine("Region number: " + regionNumber);  
    }  
}
```

Codebeispiel 18: Beispiel der Klasse: Client welche für die Serialisierung vorbereitet wird

Hinweis:

Im obigen Beispiel werden bei einem Objekt der Klasse: Client die folgenden Attributinhalt gespeichert: clientNumber, firstName, surName und factor. Das Attribut: regionNumber wird aber nicht serialisiert und somit auch nicht abgespeichert!

## 6.2.1 Objektserialisierung von Objekten, welche in einem Array gespeichert sind

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.IO;  
using System.Runtime.Serialization;  
using System.Runtime.Serialization.Formatters.Binary;  
  
namespace Objekte_in_File_speichern_V1_W0d  
{  
    class Program  
    {  
        static string fileName = "clientfile.bin";  
  
        static void Main(string[] args)  
        {  
            Client[] clients = new Client[2];  
            clients[0] = new Client(1, "Joe", "Hanson", 1.234, 6300);  
            clients[1] = new Client(2, "Tina", "Turner", 4.5, 8300);  
  
            Console.WriteLine("Safed Client;\n");  
            foreach (Client count in clients)  
            {  
                count.printClient();  
                Console.WriteLine();  
            }  
  
            FileStream fs = new FileStream(fileName, FileMode.Create);  
            IFormatter bf = new BinaryFormatter();  
  
            // Writes an array of objects to a binary file  
            bf.Serialize(fs, clients);  
  
            fs.Position = 0;  
            Console.WriteLine("\n\nReconstructed Clients;\n");  
  
            // Reads an array of objects from a binary file  
            Client[] recClients = (Client[]) bf.Deserialize(fs);  
  
            foreach (Client count in recClients)
```

```
        {  
            count.printClient();  
            Console.WriteLine();  
        }  
        fs.Close();  
    }  
}
```

Codebeispiel 19: Beispiel zur Speicherung von Objekten, welche in einem Array gespeichert sind

Ausgabe:

| Safed Client;       | Reconstructed Clients: |
|---------------------|------------------------|
| Client number: 1    | Client number: 1       |
| First Name: Joe     | First Name: Joe        |
| Surname: Hanson     | Surname: Hanson        |
| Factor: 1.234       | Factor: 1.234          |
| Region number: 6300 | Region number: 0       |
| Client number: 2    | Client number: 2       |
| First Name: Tina    | First Name: Tina       |
| Surname: Turner     | Surname: Turner        |
| Factor: 4.5         | Factor: 4.5            |
| Region number: 8300 | Region number: 0       |

Hinweis:

Obige Ausgabe gilt auch für die nächsten zwei Codebeispiele, welche für eine Liste und eine ObservableCollection geschrieben wurden.

## 6.3 Objektserialisierung von Objekten, welche in einer Liste gespeichert sind

```
using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using System.IO;  
using System.Runtime.Serialization;  
using System.Runtime.Serialization.Formatters.Binary;  
  
namespace Objekte_in_File_speichern_V2_W0d  
{  
    class Program  
    {  
        static string fileName = "clientfile.bin";  
  
        static void Main(string[] args)  
        {  
            List<Client> list1 = new List<Client>();  
            list1.Add(new Client(1, "Joe", "Hanson", 1.234, 6300));  
            list1.Add(new Client(2, "Tina", "Turner", 4.5, 8300));  
  
            Console.WriteLine("Safed Client;\n");  
            foreach (Client count in list1)  
            {  
                count.printClient();  
            }  
        }  
    }  
}
```



```
        Console.WriteLine();
    }

    FileStream fs = new FileStream(fileName, FileMode.Create);
    IFormatter bf = new BinaryFormatter();

    // Writes a list of objects to a binary file
    bf.Serialize(fs, list1);

    fs.Position = 0;
    Console.WriteLine("\n\nReconstructed Clients:\n");

    // Reads a list of objects from a binary file
    List<Client> recList = (List<Client>)bf.Deserialize(fs);

    foreach (Client count in recList)
    {
        count.printClient();
        Console.WriteLine();
    }
    fs.Close();
}
}
```

Codebeispiel 20: Beispiel zur Speicherung von Objekten, welche in einer Liste gespeichert sind

## 6.4 Objektserialisierung von Objekten, welche in einer ObservableCollection gespeichert sind

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;
using System.Collections.ObjectModel;

namespace Objekte_in_File_speichern_V3_W0d
{
    class Program
    {
        static string fileName = "clientfile.bin";

        static void Main(string[] args)
        {
            ObservableCollection<Client> oclist1 = new ObservableCollection<Client>();
            oclist1.Add(new Client(1, "Joe", "Hanson", 1.234, 6300));
            oclist1.Add(new Client(2, "Tina", "Turner", 4.5, 8300));

            Console.WriteLine("Safed Client;\n");
            foreach (Client count in oclist1)
            {
                count.printClient();
                Console.WriteLine();
            }

            FileStream fs = new FileStream(fileName, FileMode.Create);
            IFormatter bf = new BinaryFormatter();
        }
    }
}
```

```
// Writes an ObservableCollection of objects to a binary file
bf.Serialize(fs, oclist1);

fs.Position = 0;
Console.WriteLine("\n\nReconstructed Clients:\n");

// Reads an ObservableCollection of objects from a binary file
ObservableCollection<Client> recOcList =
    (ObservableCollection<Client>)bf.Deserialize(fs);

foreach (Client count in recOcList)
{
    count.printClient();
    Console.WriteLine();
}
fs.Close();
}
}
```

*Codebeispiel 21: Beispiel zur Speicherung von Objekten, welche in einer ObservableCollection gespeichert sind*

## Abbildungen:

---

|                                                                                     |    |
|-------------------------------------------------------------------------------------|----|
| Abbildung 1: Klassenübersicht der unterschiedlichen Stream-Klassen                  | 3  |
| Abbildung 2: Datentransfer: Programm (Variable) zur Senke (externer Datenspeicher)  | 3  |
| Abbildung 3: Datentransfer: Quelle (externer Datenspeicher) zum Programm (Variable) | 4  |
| Abbildung 4: Lesen von einer Datei                                                  | 7  |
| Abbildung 5: Schreiben in eine Datei                                                | 7  |
| Abbildung 6: Beispiel einer Textdatei                                               | 13 |
| Abbildung 7: Beispiel einer Binärdatei                                              | 13 |
| Abbildung 8: Klassenübersicht der Reader- und Writer-Klassen                        | 14 |
| <i>Abbildung 9: Verarbeitungskette beim Schreiben von Texten in eine Datei</i>      | 14 |
| <i>Abbildung 10: Verarbeitungskette beim Lesen eines Textes von einer Datei</i>     | 15 |
| Abbildung 11: Verarbeitungskette beim Schreiben von Binärwerten in eine Datei       | 17 |
| Abbildung 12: Verarbeitungskette beim Lesen der Binärwerte von einer Datei          | 18 |
| Abbildung 13: Verarbeitungskette beim Schreiben von Objekten in eine Binärdatei     | 21 |
| Abbildung 14: Verarbeitungskette beim Lesen von Objekten aus einer Binärdatei       | 22 |

## Codeumsetzung:

---

|                                                                                                         |    |
|---------------------------------------------------------------------------------------------------------|----|
| Codeumsetzung 1: Erzeugung und Initialisierung eines Datenstroms                                        | 7  |
| Codeumsetzung 2: Erzeugung und Initialisierung eines Datenstroms mit Filezugriffsangabe                 | 8  |
| Codeumsetzung 3: Schliessen eines Datenstroms mittels der Methode: Close()                              | 9  |
| Codeumsetzung 4: Schliessen eines Datenstroms mittels einem using-Block                                 | 10 |
| Codeumsetzung 5: Erzeugung eines StreamWriter-Objektes                                                  | 14 |
| Codeumsetzung 6: Erzeugung eines StreamReader-Objektes                                                  | 15 |
| Codeumsetzung 7: Erzeugung eines BinaryWriter-Objektes                                                  | 17 |
| Codeumsetzung 8: Erzeugung eines BinaryReader-Objektes                                                  | 18 |
| Codeumsetzung 9: Erzeugung eines BinaryFormatter-Objektes                                               | 21 |
| Codeumsetzung 10: Schreiben von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts in eine Datei | 21 |
| Codeumsetzung 11: Lesen von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts aus einer Datei   | 22 |

## Codebeispiele:

---

|                                                                                                                                        |    |
|----------------------------------------------------------------------------------------------------------------------------------------|----|
| Codebeispiel 1: Schreiben und lesen eines Byte-Arrays                                                                                  | 6  |
| Codebeispiel 2: Erzeugung und Initialisierung eines Datenstroms                                                                        | 8  |
| Codebeispiel 3: Erzeugung und Initialisierung eines Datenstroms mit Filezugriffsangabe                                                 | 8  |
| Codebeispiel 4: Schliessen eines Datenstroms mittels der Methode: Close()                                                              | 9  |
| Codebeispiel 5: Schliessen eines Datenstroms mittels einem using-Block                                                                 | 10 |
| Codebeispiel 6: Schliessen eines Datenstromes mit Ausnahmebehandlung ohne using-Block                                                  | 11 |
| Codebeispiel 7: Schliessen eines Datenstromes mit Ausnahmebehandlung mit using-Block                                                   | 12 |
| Codebeispiel 8: Erzeugung eines StreamWriter-Objektes                                                                                  | 14 |
| Codebeispiel 9: Erzeugung eines StreamReader-Objektes                                                                                  | 15 |
| Codebeispiel 10: Schreiben und Lesen von Texten mit den Klassen: StreamWriter und StreamReader sowie der Einstellung: Encoding.Default | 16 |

|                                                                                                                                        |    |
|----------------------------------------------------------------------------------------------------------------------------------------|----|
| Codebeispiel 11: Schreiben und Lesen von Texten mit den Klassen: StreamWriter und<br>StreamReader sowie der Einstellung: Encoding.UTF8 | 16 |
| Codebeispiel 12: Erzeugung eines BinaryWriter-Objektes                                                                                 | 17 |
| Codebeispiel 13: Erzeugung eines BinaryReader-Objektes                                                                                 | 18 |
| Codebeispiel 14: Schreiben und Lesen unterschiedlicher Datentypen mit den Klassen:<br>BinaryWriter und BinaryReader                    | 19 |
| Codebeispiel 15: Erzeugung eines BinaryFormatter-Objektes                                                                              | 21 |
| Codebeispiel 16: Schreiben von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts in<br>eine Datei                              | 21 |
| Codebeispiel 17: Lesen von Objekten unter Zuhilfenahme eines BinaryFormatter-Objekts aus<br>einer Datei                                | 22 |
| Codebeispiel 18: Beispiel der Klasse: Client welche für die Serialisierung vorbereitet wird                                            | 23 |
| Codebeispiel 19: Beispiel zur Speicherung von Objekten, welche in einem Array gespeichert sind                                         | 24 |
| Codebeispiel 20: Beispiel zur Speicherung von Objekten, welche in einer Liste gespeichert sind                                         | 25 |
| Codebeispiel 21: Beispiel zur Speicherung von Objekten, welche in einer Observable-<br>Collection gespeichert sind                     | 26 |

## Historie

| Vers. | Bemerkungen                               | Verantwortl. | Datum      |
|-------|-------------------------------------------|--------------|------------|
| 0.1   | - erstes Zusammenführen der Informationen | W. Odermatt  | 14.02.2016 |
| 1.0   | - letzte Layoutanpassungen gemacht        | W. Odermatt  | 07.04.2016 |

## Referenzunterlagen

| LfNr. | Titel / Autor / File / Verlag / ISBN                                                           | Dokument-Typ      | Ausgabe |
|-------|------------------------------------------------------------------------------------------------|-------------------|---------|
| 1     | „Programmiersprache C“, Implementierung C, Grundkurs / H.U. Steck                              | Theorieunterlagen | 2003    |
| 2     | „Programmieren in C“ / W. Sommergut / Beck EDV-Berater im dtv / 3-423-50158-8                  | Fachbuch          | 1997    |
| 3     | „C Kompakt Referenz“ / H. Herold / Addison Wesley / 3-8273-1984-6                              | Fachbuch          | 2002    |
| 4     | „C in 21 Tagen“ / P. Aitken, B. L. Jones / Markt + Technik / 3-8272-5727-1                     | Fachbuch          | 2000    |
| 5     | „C Programmieren von Anfang an“ / H. Erlenkötter / rororo / 3-499-60074-9                      | Fachbuch          | 2001    |
| 6     | „Programmieren in C“ / B.W. Kernighan, D.M. Ritchie / Hanser / 3-446-25497-3                   | Fachbuch          | 1990    |
| 7     | „Einstieg in Visual C# 2013“ / Thomas Theis / Galileo Computing / 978-3-8362-2814-5            | Fachbuch          | 2014    |
| 8     | „Visual C# 2012“ / Andreas Kühnel / Rheinwerk Computing / 978-3-8362-1997-6                    | Fachbuch          | 2013    |
| 9     | „Objektorientiertes Programmieren in Visual C#“ / Peter Loos / Microsoft Press / 3-86645-406-6 | Fachbuch          | 2006    |
| 10    | „Einführung in das Programmieren mit C# 4.0“ / Bernhard Baltes-Götz / Universität Trier        | Vorlesung         | 2011    |