# Zugriff auf das Dateisystem mit C#

# Dateitypen

- Text-Dateien
  - Zeichenbasiert
  - Unicode, ASCII oder andere Kodierung
  - Mit jedem Text-Editor wie z.B. notepad les- und änderbar
- Binär-Dateien
  - Byte-Bedarf richtet sich nach den abzuspeichernden Datentypen
  - Editierbar mit Hex-Editoren, in einem Text-Editor wird jedes byte oder die entsprechende byte-Zeichenbreite als Zeichen interpretiert, was nur bei Textfragmenten zu lesbaren Inhalten führen kann
  - Für das Lesen von binär kodierten Dateien muß der Aufbau der Datei bekannt sein, dies bedeutet z.B. beim Schreiben von Listen einen Kenner, wann die Liste endet oder die Angabe, wie viele Listenelemente abgespeichert wurden

- Besondere Datei-Formate
  - Textdateien zur Abbildung strukturierter Daten (cfg/ini-Dateien, xml-Dateien)
  - Ausführbare Dateien sind binäre Dateien mit entsprechenden Header-Informationen, die das Betriebssystem zum Starten benötigt

# Die Klassen FileInfo und DirectoryInfo

- Für ein einfaches Handling von Dateien und Verzeichnissen bietet .NET mit System.IO.FileInfo und System.IO.DirectoryInfo zwei Klassen an
- Die FileInfo-Klasse erlaubt den Zugriff auf Dateiattribute, aber ebenso Operationen wie Kopieren, Löschen oder Verschieben der Datei
- Die Klasse DirectoryInfo erlaubt analog den Zugriff auf Verzeichnisattribute wie Elternordner, enthaltene Dateien oder Verzeichnisse sowie Operationen in ähnlicher Form wie die FileInfo-Klasse (z.B. getFiles zum Abrufen aller Dateien in einem Ordner)
- Mit diesen beiden Klassen lassen sich
  - Verzeichnisbäume durchlaufen und
  - Dateien und Verzeichnisse manipulieren
- Die FileInfo-Klasse stellt darüber hinaus Methoden zur Verfügung, die ein Lesen und Schreiben von Dateien erlauben
- Die Klasse File stellt statische Methoden in ähnlicher Form wie bei einem Objekt der Klasse FileInfo bereit, hier finden sich aber Methoden, die auf das Vorhandensein einer Datei und das Manipulieren der Datei abzielen (z.B. Löschen)

# Die Klassen File und Directory

- Diese Klassen ähneln den Klassen FileInfo und DirectoryInfo, bieten z.T. die gleiche Funktionalität, jedoch auf Basis statischer Funktionen
- Darüber hinaus bieten die Klassen hilfreiche Funktionen, mit denen sich einfach in Dateien schreiben lässt (File.AppendText) oder eine ganze Textdatei mit einem Befehl in ein String-Array einlesen lässt (File.ReadAllLines)
- Mit der Directory-Klasse lassen sich ohne Objekt Verzeichnisse durchsuchen, löschen oder erstellen
- Die –Info-Pendants können durch ihre nicht-statische Struktur als Objekte gehalten werden und damit überall direkt verfügbar gemacht oder übergeben werden

# Beispiel: prüfen, ob Datei existiert

- Zusammensetzen eines Dateinamens mit Pfadangabe über Path.Combine
- Prüfen, ob Datei vorhanden ist, mit File.Exists
- Anzeige einer Dialogbox über MessageBox.Show mit vielen Überladungen, hier mit der Anzeige zweier Schaltflächen Ja-Nein und Ablegen des Ergebnisses in Variable vom Typ DialogResult

```
// Dateiname zusammensetzen, startpath ist Pfad der Anwendung selber
string mm_fileName = Path.Combine(Application.StartupPath, "datei.txt");

//Variable für Antwort auf Nachfrage...
DialogResult mm_ant = DialogResult.Yes;

if (File.Exists(mm_fileName)) // Ist die Datei bereits vorhanden???
{                             // wenn ja, dann nachfragen, ob überschrieben werden soll
    mm_ant = MessageBox.Show(mm_fileName + "ist bereits vorhanden.\r\n\r\n" +
       "Soll die Datei überschrieben werden?", Application.ProductName,
        /* Messagebox mit Ja-Nein-Schaltflächen */ MessageBoxButtons.YesNo,
        /* Icon in der MessageBox */               MessageBoxIcon.Question);
}
```

# Lesen und Schreiben in Dateien

- Dateien können in verschiedener Weise geöffnet werden:
  - Textdatei
  - Binärdatei
- Die Klasse StreamWriter kann benutzt werden, um Daten in eine Textdatei zu schreiben
- Die Klasse StreamReader analog dazu, um Daten aus einer Textdatei zu lesen
- Analog zu StreamWriter und –Reader bieten die Klassen BinaryReader und BinaryWriter die Möglichkeit, Dateien binär zu schreiben bzw. zu lesen
  - Für diese beiden Klassen wird die Datei nicht als String im Konstruktor übergeben, sondern als zuvor erzeugtes FileStream-Objekt
  - Das Filestream-Objekt wird mit Dateinamen, Typ des Zugriffs (Append, Truncate, Open, …) und der Art des Zugriffs (Read, ReadWrite oder Write) und dem möglichen gleichzeitigen Zugriff von anderen Benutzern (Read, ReadWrite, Write, None, …)
  - Auch StreamReader und StreamWriter lassen sich mit einem FileStream-Objekt im Konstruktor erzeugen
- Darüber hinaus ist es möglich, sich auch aus einem FileInfo-Objekt StreamReader bzw. StreamWriter aus Methodenaufrufen direkt erzeugen zu lassen (mit CreateText(), AppendText() oder OpenText())

# Beispiel: Schreiben in eine Textdatei mit einem StreamWriter

- StreamWriter enthält ähnliche Methoden wie die Console, allerdings nur die Ausgabe-Methoden, Lese-Methoden werden über StreamReader angeboten

```
if (DialogResult.Yes == mm_ant) //Wenn nicht mit nein abgebrochen wurde...
{
    StreamWriter mm_streamwriter = null; //Streamwriter Variable definieren
    try
    {
        //StreamWriter-Instanz für die Datei erzeugen
        mm_streamwriter = new StreamWriter(mm_fileName, /* append */ false,
            /*Kodierung der Datei*/ Encoding.GetEncoding("windows-1252"));
        // Datei zeilenweise schreiben
        mm_streamwriter.WriteLine("Zeile 1: Hallo");
        mm_streamwriter.WriteLine("Zeile 2: Welt!");
        // StreamWriter und damit auch Datei schließen
        mm_streamwriter.Close();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Fehler beim Öffnen, Schreiben oder Schließen der Datei '" +
            mm_fileName + "': " + ex.Message, Application.ProductName,
            MessageBoxButtons.OK, MessageBoxIcon.Error);
        //Bei Fehler Verlassen der Methode
        return;
    }
}
```

# Beispiel: Lesen mit StreamReader, der über FileStream erzeugt wird

- FileStream-Objekt mit zum Stream-Objekt passenden Zugriff – Erzeugen eines StreamWriters im unteren Beispiel führt zu Laufzeitfehler
- StreamReader wird erzeugt und bietet ähnliche Methoden wie Console an – zumindest die lesenden Methoden

```
// Dateiname zusammensetzen, startpath ist Pfad der Anwendung selber
string mm_fileName = Path.Combine(Application.StartupPath, "datei.txt");
FileStream mm_fs = new FileStream(mm_fileName,
    /* Öffnen oder Anlegen - Datei wird immer gefunden */ FileMode.OpenOrCreate,
    /* Datei zum Lesen öffnen */ FileAccess.Read ,
    /* Andere dürfen jetzt nicht an die Datei ran */ FileShare.None);
//StreamReader aus FileStream erstellen..
StreamReader mm_sr = new StreamReader(mm_fs);
//Eine Zeile Einlesen
string mm_line = mm_sr.ReadLine();
```

# XML-Dateien

- XML (Extensible Markup Language) ist eine Auszeichnungssprache
  - Darstellung hierarchisch strukturierter Daten in Form von Textdaten
  - plattform- und implementationsunabhängig
  - Austausch von Daten zwischen Computersystemen insbesondere über das Internet.
  - Definiert und herausgegeben vom World Wide Web Consortium (W3C)
  - Basis vieler abgeleiteter Sprachen wie XHTML,
  - Ablösung der bisher weitgehend eingesetzten Ini-Dateien

- Das Format einer XML-Datei wird i.d.R. festgelegt durch
  - DTD-Datei (document type definition)
  - XML-Schema (XSD xml-scheme-definition)

- Mit gegebener xml-Datei lassen sich in Visual Studio Schemadateien erzeugen
- Die Schemadatei muss in der Regel nachträglich an die genauen Gegebenheiten angepasst werden
- Eine XML-Datei kann bei Vorhandensein eines Schemas im XML-Editor oder beim Import in C# auf Gültigkeit geprüft werden, ohne Schema nicht.

# Beispiel einer XML-Datei

```
<?xml version="1.0"?>
<catalog>
   <book id="bk101">
      <author>Gambardella, Matthew</author>
      <title>XML Developer's Guide</title>
      <genre>Computer</genre>
      <price>44.95</price>
      <publish_date>2000-10-01</publish_date>
      <description>An in-depth look at creating applications
      with XML.</description>
   </book>
   <book id="bk102">
      <author>Ralls, Kim</author>
      <title>Midnight Rain</title>
      <genre>Fantasy</genre>
      <price>5.95</price>
      <publish_date>2000-12-16</publish_date>
      <description>A former architect battles corporate zombies,
      an evil sorceress, and her own childhood to become queen
      of the world.</description>
   </book>
</catalog>
```

Quelle: Microsoft

# XML mit DTD

```
<!ELEMENT adressen (adresse+)>              //+:1..n, *:0..n, ?:0..1
<!ELEMENT adresse (name)>                   //1 name unterhalb von adresse
<!ATTLIST adresse                           //zwei Attribute an der adresse
 mail CDATA #IMPLIED                        //mail optional mit gültigem xml-Text
 verteiler (yes|no) "yes">                  //verteiler mit default="yes"
<!ELEMENT name (Vorname, Nachname)>         //An name hängen zwei Elemente
<!ELEMENT Vorname (#PCDATA)>                //Das Element enthält parsed character data
<!ELEMENT Nachname (#PCDATA)>               //diese werden als xml-Daten geparst

<?xml version="1.0"?>
<!DOCTYPE adressen SYSTEM "adressen.dtd">
<adressen>
        <adresse mail="susi.sorglos@web.de" verteiler="no">
                <name>
                        <Vorname>Susi</Vorname>
                        <Nachname>Sorglos</Nachname>
                </name>
        </adresse>
        <adresse mail="paul.baumann@gmx.de">
                <name>
                        <Vorname>Paul</Vorname>
                        <Nachname>Baumann</Nachname>
                </name>
        </adresse>
</adressen>
```

# XSD-Schema

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">  Standard-Namespace xs

<xs:element name="shiporder">
  <xs:complexType>
    <xs:sequence>
      <xs:element name="orderperson" type="xs:string"/>
      <xs:element name="shipto">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="name" type="xs:string"/>
            <xs:element name="address" type="xs:string"/>
            <xs:element name="city" type="xs:string"/>
            <xs:element name="country" type="xs:string"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
      <xs:element name="item" maxOccurs="unbounded">
        <xs:complexType>
          <xs:sequence>
            <xs:element name="title" type="xs:string"/>
            <xs:element name="note" type="xs:string" minOccurs="0"/>
            <xs:element name="quantity" type="xs:positiveInteger"/>
            <xs:element name="price" type="xs:decimal"/>
          </xs:sequence>
        </xs:complexType>
      </xs:element>
    </xs:sequence>
    <xs:attribute name="orderid" type="xs:string" use="required"/>
  </xs:complexType>
</xs:element>

</xs:schema>
```

Quelle: w3c

# XML mit xsd-Schema

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
  <orderperson>John Smith</orderperson>
  <shipto>
    <name>Ola Nordmann</name>
    <address>Langgt 23</address>
    <city>4000 Stavanger</city>
    <country>Norway</country>
  </shipto>
  <item>
    <title>Empire Burlesque</title>
    <note>Special Edition</note>
    <quantity>1</quantity>
    <price>10.90</price>
  </item>
  <item>
    <title>Hide your heart</title>
    <quantity>1</quantity>
    <price>9.90</price>
  </item>
</shiporder>
```

# XML mit dem XmlTextReader

```
(using System.Xml;)
…
System.Xml.XmlTextReader reader = new
System.Xml.XmlTextReader("c:\\IntroToVCS.xml");
          string contents = "";
          while (reader.Read())
          {
             reader.MoveToContent();
             if (reader.NodeType == System.Xml.XmlNodeType.Element)
                contents += "<"+reader.Name + ">\n";
             if (reader.NodeType == System.Xml.XmlNodeType.Text)
                contents += reader.Value + "\n";
          }
Console.Write(contents);
```

# XML mit DOM (Document Object Model)

```
using System.Xml;
…
XmlDocument doc = new XmlDocument();
doc.Load("articles.xml");

XmlElement root = doc.DocumentElement;

foreach(XmlNode article in root.ChildNodes)
{
        …
}
```

```
<?xml version="1.0" encoding="Windows-1252"?>
<articles>
 <article>
   <title></title>
   <author></author>
   <url></url>
   <date></date>
   <description></description>
 </article>
</articles>
```

# Aufgabe

- Erstellen Sie ein Konzept für eine Klasse, die einen einfachen Zugriff auf alte Ini-Dateien ermöglicht
  - Welche Möglichkeiten sollte Ihre Klasse für den Zugriff bieten
  - Definieren Sie alle notwendigen Methoden und Klassen mit Hilfe der UML

```
[fonts]
Norm=courier

[Mail]
MAPI=1

[MCI Extensions.BAK]
3g2=MPEGVideo
3gp=MPEGVideo
3gp2=MPEGVideo
mp4v=MPEGVideo
mts=MPEGVideo
ts=MPEGVideo
tts=MPEGVideo
```

# Serialisierung – Basisklasse zum Schreiben in Datei

```
[Serializable()]
public class Auto : ISerializable
{
   private string marke;
   private string modell;
   private int jahr;

   public Auto()
   {
   }

   public Auto(SerializationInfo info, StreamingContext ctxt)
   {
      this.marke = (string)info.GetValue("Marke", typeof(string));
      this.modell = (string)info.GetValue("Modell",typeof(string));
      this.jahr = (int)info.GetValue("Jahr", typeof(int));
   }

   public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
   {
      info.AddValue("Marke", this.marke);
      info.AddValue("Modell", this.modell);
      info.AddValue("Jahr", this.jahr);
   }
}
```

# Optionen beim Serialisieren

- Ohne weitere Angaben werden alle öffentlichen Eigenschaften in das File geschrieben
- Zum Deserialisieren muss jede zu serialisierende Klasse über einen parameterlosen Konstruktor verfügen, damit der Deserialisierer das neu zu erstellende Objekt erzeugen kann
- Das Serialisieren sollte als nicht static-Methode direkt in die zu serialisierende Klasse eingebaut werden, sondern am zu serialisierenden Objekt gerufen werden; die Methode zum Deserialisieren muss jedoch immer static sein, da das zu deserialisierende Objekt erst erstellt werden muss (daher Standardkonstruktor)
- Durch Implementieren der Schnittstelle ISerializable kann das Verhalten der zu serialisierenden Klasse explizit angepasst werden, da mit der Methode GetObjectData genau definiert werden kann, welche Eigenschaften serialisiert werden können
- Bei XML-Serializern ist es möglich, durch zusätzliche Angaben Kindknoten für Felder/Listen [XmlArray]  zu definieren, dabei auch verschiedene Klassen zu berücksichtigen
- Des weiteren ist es möglich, den XML-Tags eigene Namen zu geben  (z.B. [XmlArray(“Anforderungen“)]
- Achtung – beim Versionswechsel des Programms muss Aufwand betrieben werden, alte Objekte wieder einzulesen – der BinaryFormatter schreibt Assembly-Infos in die Datei

# Serialisierung – Sammelklasse für Objekte

```
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

[Serializable()]
public class ObjectToSerialize : ISerializable
{
   private List<Auto> autos;

   public List<Auto> Autos
   {
      get { return this.autos; }
      set { this.autos = value; }
   }

   public ObjectToSerialize()
   {
   }

   public ObjectToSerialize(SerializationInfo info, StreamingContext ctxt)
   {
      this.autos = (List<Auto>)info.GetValue("Autos", typeof(List<Auto>));
   }

   public void GetObjectData(SerializationInfo info, StreamingContext ctxt)
   {
      info.AddValue("Autos", this.autos);
   }
}
```

# Serialisierung – Datenformatierung für Datei

```
using System.IO;
using System.Runtime.Serialization;
using System.Runtime.Serialization.Formatters.Binary;

public class Serializer
{
   public Serializer()
   {
   }

   public void SerializeObject(string filename, ObjectToSerialize objectToSerialize)
   {
      Stream stream = File.Open(filename, FileMode.Create);
      BinaryFormatter bFormatter = new BinaryFormatter();
      bFormatter.Serialize(stream, objectToSerialize);
      stream.Close();
   }

   public ObjectToSerialize DeSerializeObject(string filename)
   {
      ObjectToSerialize objectToSerialize;
      Stream stream = File.Open(filename, FileMode.Open);
      BinaryFormatter bFormatter = new BinaryFormatter();
      objectToSerialize = (ObjectToSerialize)bFormatter.Deserialize(stream);
      stream.Close();
      return objectToSerialize;
   }
}
```

# Serialisierung – das eigentliche Schreiben

```
List<Auto> autos = new List<Auto>();

//Speichern der Autos-Liste in eine Datei
ObjectToSerialize objectToSerialize = new ObjectToSerialize();
objectToSerialize.Autos = autos;

Serializer serializer = new Serializer()
serializer.SerializeObject("outputFile.txt", objectToSerialize);

//Zurücklesen der Daten aus der Datei

objectToSerialize = serializer.DeSerializeObject("outputFile.txt");
autos = objectToSerialize.Autos;
```

# Mit XSD eine C#-Klasse erstellen

- Mit dem Tool xsd (Befehlszeilentool über die Microsoft Visual Studio Eingabeaufforderung) lassen sich aus einer xsd-Datei C#-Klassen zur Abbildung der xsd-Struktur im Programm abbilden
- Dabei kann durch Befehlsparameter eingestellt werden, ob Felder als Properties oder Fields, mit oder ohne linq-Unterstützung usw. erstellt werden.
- Die derart erstellten Klassen lassen sich dann leicht serialisieren und können direkt aus entsprechenden xml-Dateien importiert werden:

```
XML_Konfig mm_konfig;
XmlSerializer mm_xser = new XmlSerializer(typeof(XML_Konfig));
TextReader mm_textReader = new StreamReader(@"H:\Konfiguration.xml");
mm_konfig = (XML_Konfig) mm_xser.Deserialize(mm_textReader);
mm_textReader.Close();

Oder

TextWriter mm_textWriter = new StreamWriter(@"H:\Konfiguration.xml");
mm_xser.Serialize(mm_textWriter, mm_Konfig); //mit gültigem Objekt!
```