Objekte und Klassen (Teil 3)

Im Teil 2 dieses Themas wurde die Klasse Person implementiert. Es wurden Objekte vom Typ Person erstellt und in einem generischen Container vom Typ **`List<T>`** abgelegt.

Im nächsten Schritt sollen die Daten der **`Person`**-Objekte in eine Textdatei geschrieben werden, so dass diese beim nächsten Programmstart eingelesen und hiervon wieder **`Person`**-Objekte erstellt werden können. Diese Fähigkeit bezeichnet man in der Informatik als Persistenz[1].

Schreiben und Lesen von Objekt-Daten

Damit in der Folge nicht immer wieder Daten von Personen eingegeben werden müssen, sollen die Daten der Person-Objekte beim Schließen des Programms in eine Textdatei geschrieben werden. Beim nächsten Programmstart sollen aus diesen Daten dann wieder Person-Objekte erstellt werden. Hierzu wird zwei Methoden implementiert; eine für das Schreiben und eine für das Lesen der Daten.

Die Programmiersprache **`C#`** stellt für das Schreiben und Lesen von Dateien verschiedene Klassen[2] zur Verfügung. Diese Klassen befinden sich im Namespace **`System.IO`**, der mit einer **`using`**-Direktive einzubinden ist.

Prinzipiell kann man unterscheiden zwischen:

1. **`BinaryReader`** bzw. **`BinaryWriter`**: Lesen und Schreiben von primitiven Datentypen als Binärwerte.
2. **`StreamReader`** und **`StreamWriter`**: Lesen und Schreiben von Zeichen mithilfe eines Codierungswerts, der die Zeichen in Bytes konvertiert und umgekehrt.

Für das Projekt „Adressbuch“ sollen die Daten in Textform geschrieben und gelesen werden. Daher finden die beiden Klassen **`StreamReader`** und **`StreamWriter`** hier Anwendung.

Speichern mit einem Objekt der `StreamWriter-Klasse`[3]

Das Beispiel im Kasten zeigt die Anwendung der Klasse **`StreamWriter`**.

Mit der Anweisung **`StreamWriter sw = new StreamWriter(@"personen.txt")`** wird ein Objektvariable mit dem Namen **`sw`** vom Typ **`StreamWriter`** erzeugt und mittels **`new`**-Operator auch direkt initialisiert.

Der Konstruktor-Methode wird als Parameter der Name der zu erstellenden Datei (in diesem Beispiel: **`personen.txt`**) übergeben. Das Zeichen „`@`“ bewirkt, dass der Dateiname als Text interpretiert wird. Dies ist v.a. bei der Angabe von Pfaden wichtig.

```
// Speichere alle Daten einer Person

using (StreamWriter sw =
       new StreamWriter(@"personen.txt"))
{
    foreach (Person p in personen)
    {
        sw.WriteLine(p.FirstName + ";" +
                     p.LastName + ";" +
                     p.Birthday.ToShortDateString()
                    );
    }
}
```

1 siehe auch **`https://de.wikipedia.org/wiki/Persistenz_(Informatik)`** abgerufen am 23. Mai 2017

2 Eine Beschreibung aus der MSDN zu Datei- und Stream E/A findet sich hier: **`https://msdn.microsoft.com/de-de/library/k3352a4t(v=vs.110).aspx`** abgerufen am 23. Mai 2017

3 **`https://msdn.microsoft.com/de-de/library/system.io.streamwriter%28v=vs.110%29.aspx`** abgerufen am 23. Mai 2017

Die Erstellung des Objektes erfolgt in Verbindung mit einer **`using`**-Anweisung.

Dies hat den Vorteil, dass ein derart erstelltes Objekt automatisch der Speicherbereinigung zugeführt wird, wenn es nicht mehr benötigt wird. Bei schreibenden Dateioperationen ist ansonsten der Aufruf einer Methode zum Schließen der Schreiboperationen erforderlich.

In diesem Fall müsste nach Ausführung aller Schreiboperationen der Befehl **`sw.Close()`** aufgerufen werden. Aufgrund der **`using`**-Anweisung kann dies jedoch entfallen!

Innerhalb des **`using`**-Blocks werden nun alle Schreibanweisungen auf diese Datei durchgeführt. Im obigen Beispiel werden nun mit einer foreach-Schleife die Daten aller Person-Objekte im Container personen in eine Datei geschrieben.

Für das Schreiben von Daten verfügt ein **`StreamWriter`**-Objekt über zwei[4] (für das Projekt relevante) Methoden: **`Write()`** und **`WriteLine()`**.

Die letztere fügt automatisch am Ende der zu schreibenden Daten einen Zeilenabschluss hinzu.

Jede im obigen Beispiel zu schreibende Zeile enthält den Vornamen, den Namen und das Geburtsdatum einer Person **`p`** aus dem Container **`personen`**. Die einzelnen Bestandteile werden mit dem Zeichen „**+**“ verkettet. Zwischen den Bestandteilen wird noch ein Semikolon eingefügt, das als Trennzeichen fungiert.

Das Ergebnis der Schreiboperationen in die Datei **`personen.txt`** könnte dann so aussehen.

```
Uwe;Homm;23.05.2017
Friedhelm;Ernst;16.03.2010
Karin;Ramsthaler;08.02.2015
Christine;Schmidt;03.10.1998
```

Die erstellte Datei befindet sich im gleichen Verzeichnis wie die Programmdatei mit der Erweiterung **`exe`**.

Soll die Datei in einem anderen Verzeichnis erstellt werden, ist ein Pfad anzugeben; z.B. **`"@D:\Testverzeichnis\personen.txt"`**.

Aufgaben

1. Ändern Sie zuerst die aktuelle Version der Klasse **`Person`** ab. Jede Person soll als Attribut eine numerische ID haben (Datentyp **`uint`**, Attributname **`id`**). Hierzu ist ein Property namens **`Id`** zu implemetieren, welches **nur** den **`get`**-Teil enthält. Der Wert von **`id`** soll automatisch anhand einer Klassenvariablen **`lastid`** erzeugt werden. Der zuletzt verwendete Wert von **`id`** wird in dieser Klassenvariablen mit der Bezeichnung **`lastid`** verwaltet. Diese Klassenvariable soll von außen nicht zugreifbar sein!
2. Implementieren Sie in der Klasse **`Form1`** (falls Sie den Standardnamen verwendet haben) eine nicht-öffentliche Methode namens **`storeAllPersons():void`**, mit der alle Person-Objekte in eine Datei **`personen.txt`** geschrieben werden.
3. Damit beim Beenden des Programms, genauer gesagt beim Schließen des Formulars, alle Personen-Objekt gespeichert werden, fügen Sie der Klasse **`Form1`** einen Eventhandler zu, der bei diesem Ereignis aufgerufen wird. Das Event hat den Namen **`FormClosing`**. Sie finden es in der Entwurfsansicht, wenn Sie das Formular anklicken und im Eigenschaften-Fenster auf das Blitz-Symbol klicken. Hier sehen Sie alle Events, auf die das Formular reagieren kann. Wenn Sie nun einen Doppelklick auf das o.g. Event ausführen, wird ein Eventhandler erzeugt. In diesem Eventhandler rufen Sie die Methode aus Aufgabe 2 auf.

4 Eigentlich sind es vier Methoden. Die beiden anderen werden nicht betrachtet.

Lesen mit einem Objekt der `StreamReader-Klasse`[5]

Das Beispiel im Kasten zeigt eine Anwendung der Klasse **`StreamReader`**. In der Zeile **`01`** wird innerhalb der runden Klammern eine **`StreamReader`**-Objektvariable namens **`sr`** erstellt und mittels **`new`**-Operator und Konstruktor initialisiert.

```
01 using (StreamReader sr = new StreamReader(@"personen.txt"))
02 {
03     // Datei zeilenweise lesen bis zum Ende der Datei
04     string zeile;
05     while ((zeile = sr.ReadLine()) != null)
06     {
07         // Nun die Zeile splitten anhand
08         // des Trennzeichens ';'
09         string[] inhalt = zeile.Split(';');

10         // Werte vorbereiten für Anlegen
11         // eines Person-Objekts
12         uint i = Convert.ToUInt32(inhalt[0]);   // ID
13         string v = inhalt[1]; // Vorname
14         string n = inhalt[2];   // Name
15         string g = inhalt[3];   // Geburtstag
16         DateTime gb = Convert.ToDateTime(g);

17         // Person-Objekt erstellen
18         Person p = new Person(i, v, n, gb);
19         // Person-Objekt in die Liste personen
20         personen.Add(p);
21     }
22 }
```

Die Konstruktor-Methode erhält den Namen der zu lesenden Textdatei als Parameter.

Der Kopf der nachfolgenden **`while`**-Schleife in Zeile **`05`** ist etwas tricky, da hier mehrere Dinge passieren!

In der innersten Klammer in Zeile **`05`** erfolgt der eigentliche Lesevorgang durch die Methode **`ReadLine()`**, die jeweils eine ganze Zeile aus der Textdatei liest. Diese Methode liefert die gelesene Zeile zurück und speichert diese in der **`string`**-Variablen namens **`zeile`**. Wird irgendwann das Ende der Textdatei erreicht, liest die Methode **`ReadLine()`** den Wert **`null`**. Deshalb wird der gerade mit **`ReadLine()`** gelesene und in **`zeile`** gespeicherte Wert mit dem Wert **`null`** verglichen. Liefert diese Bedingung den Wert **`true`**, ist das Ende der Textdatei erreicht; andernfalls wird das Lesen fortgesetzt.

Nach jeder erfogreich gelesenen Zeile, wird nun der Schleifenrumpf (Zeile **`07`** bis **`20`**) ausgeführt.

```
1;Uwe;Homm;23.05.2017
2;Friedhelm;Ernst;16.03.2010
3;Karin;Ramsthaler;08.02.2015
4;Christine;Schmidt;03.10.1998
```

Hier muss nun die gelesene Zeile in ihre Bestandteile zerlegt werden, um daraus die Werte zu gewinnen, die für die Erstellung eines Person-Objekts erforderlich sind. Jede Zeile enthält vier Werte, die jeweils durch das Zeichen „**`;`**“ getrennt werden. Der erste Wert ist die ID, der zweite der Vorname, der dritte Wert ist der Nachname und der vierte das Geburtsdatum im Format **`TT.MM.JJJJ`**.

In der Zeile **`09`** wird nun die Zerlegung durchgeführt. Hierzu wird eine Methode der Klasse **`string`** verwendet: die Methode **`Split()`**. Diese Methode erhält als Parameter das gewünschte Trenn**zeichen**; in diesem Fall eben das Semikolon. Die Betonung liegt auf Zeichen, da dieses Zeichen eben rechts und links von einem einzelnen Anführungszeichen begrenzt wird – und nicht wie bei einem String von doppelten Anführungszeichen!

0	1
1	Uwe
2	Homm
3	23.05.2017

Der Rückgabewert der **`Split()`**-Methode ist ein *eindimensionales Array* vom Typ **`string`**. Ein eindimensionales Array ist mit einer einspaltigen Tabelle vergleichbar: In der ersten Zeile der „Tabelle“ (mit der Zeilennummer 0) wird der erste Wert der gelesenen Zeile gespeichert; in der zweiten Zeile (Zeilennummer 1) der zweite Wert, usw. Man spricht bei einem Array aber nicht von Zeilennummern, sondern von einem *Index*. Das erste Array-Element hat den Index 0, das zweite Array-Element den Index 1, das dritte Array-Element den Index 2 usw.

5 **`https://msdn.microsoft.com/de-de/library/system.io.streamwriter%28v=vs.110%29.aspx`** abgerufen am 23. Mai 2017

Bei einem eindimensionalen Array mit z. B. 15 Werten, kann mittels Index 0 bis 14 auf die einzelnen Werte zugegriffen werden. Ein Array kann von jedem beliebigen Datentyp erstellt werden.

Ein Array ist ein anderer Containertyp, eine Alternative zur Klasse `List<T>`.

Arrays werden so erzeugt:

Hinter dem Datentyp wird ein eckiges Klammernpaar „`[]`“gesetzt, wodurch festgelegt wird, dass es sich um ein Array handelt. Anschließend folgt der Name des Arrays. Mit dem **`new`**-Operator wird nun das Array in der gewünschten Größe, d. h. wie viele Elemente das Array speichern kann, definiert.

```
Datentyp[] arrayname = new Datentyp[Anzahl];
```

Beispiele

```
int[] lottozahlen = new int[6];
double[] gehaelter = new double[1000];
Person[] personen = new Person[100];
```

Zugriff auf die Werte in einem Array

```
int zahl = lottzahlen[3];
double gehalt = gehaelter[999];
Person p = personen[55];
```

Auf die im Array gespeicherten Werte kann nun durch Angabe des betreffenden Index zugegriffen werden. Die **`Split()`**-Methode liefert also ein eindimensionales **`string`**-Array mit genau so vielen Elementen (sprich: Strings) zurück, wie in der Textzeile vorhanden sind.

Die vier Strings in unserem Beispiel befinden sich in dem Array mit dem Namen **`inhalt`** und über die Indizes 0 bis 3 (**`inhalt[0]`**, **`inhalt[1]`**, **`inhalt[2]`**, **`inhalt[3]`**) kann nun auf diese Strings zugegriffen werden.

Diese Inhalte müssen nun noch in zwei Fällen in den richtigen Datentyp zur Erstellung eines Person-Objekts umgewandelt werden. Aus dem String in **`inhalt[0]`**, der ja die ID enthält, muss noch ein Integer-Wert gemacht werden (Zeile **`12`**). Und aus dem String in **`inhalt[3]`**, der ja das Geburtsdatum enthält, muss noch ein **`DateTime`**-Objekt gemacht werden (Zeile **`16`**).

Nachdem nun alle Informationen zum Erzeugen eines **`Person`**-Objektes in Variablen vorliegen, wird das **`Person`**-Objekt erzeugt (Zeile **`18`**) und der Liste mit den **`Person`**-Objekten hinzugefügt (Zeile **`20`**).

Aufgaben

4. Implementieren Sie in der Klasse **`Form1`** (falls Sie den Standardnamen verwendet haben) eine nicht-öffentliche Methode **`readAllPersons():void`**, mit der nun beim Programmstart alle Einträge in der Datei **`personen.txt`** gelesen und als **`Person`**-Objekte in der Liste **`personen`** eingetragen werden.
   Damit dies bei jedem Programmstart automatisch durchgeführt wird, rufen Sie diese Methode in einem geeigneten Eventhandler auf. Sie können hierzu einen Eventhandler des Formulars **`Form1`** verwenden, nämlich den **`Load`**-Eventhandler. Das Ereignis **`Load`** tritt immer dann ein, wenn das betreffende Formular - in diesem Fall **`Form1`** - geladen wird.
5. Testen Sie ausgiebig, dass Schreiben und Lesen der Datei **`personen.txt`**. Testen Sie auch, dass die ID richtig geführt wird. Löschen Sie in der Datei eine Zeile und kontrollieren Sie, ob nach einem erneuten Programmstart und dem Hinzufügen einer neuen Person die ID richtig geführt wird. Sie darf sich nicht ändern!