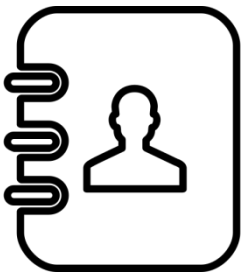
Inhalt

Änderungshistorie

Seit Version 1	kleine Textanpassungen
	5.3 LoadAll() zugefügt
	5.4 Umbenennung: *MyFriendBookApp* → *MyContactBookApp* (jetzt einheitlich)
	5.4 Übung: Skript **run.cmd** zum Start mit Doppelklick zugefügt
	5.5 Info zu Programmierstil **P**ascal**C**ase vs. **c**amel**C**ase
	5.7 Aufgabe b) angepasst, Aufgabe c) zugefügt
	5.8 Ab hier Dokument fortgeschrieben

1 Lernziele



- Entwicklung einer .NET Core Konsolenanwendung zur Kontaktverwaltung
- C# Projekt mit .NET Core Command Line Interface (CLI) erzeugen
- Benutzerinteraktion mit Kommandomenü programmieren
- Konzept einer gezielten Methodenstruktur nutzen
- Datentyp *String* und Verarbeitungsmethoden verwenden
- Arrays zur internen Datenhaltung nutzen
- Datentyp *Char*, Zeichensätze und Codepages kennen lernen
- Dateiablage in Textzeilenformat programmieren

2 Voraussetzungen

- Erste C# Konsolenanwendung mit Interaktion bereits entwickelt (z.B. Taschenrechner)
- Nutzung von geeigneten Programmierbüchern zu C#

3 Format

- Programmstruktur und Benutzerinteraktion wird vorgegeben
- Die Entwicklung erfolgt in Lernschritten stufenweise bis zur fertigen Anwendung
- Zu den Lernschritten werden passend Lerninhalte beigesteuert
- Einzelne Aufgaben und Fragestellungen setzten den Übungsfokus

4 Arbeitsweise

- Partnerarbeit mit stetigem Wechsel an der Tastatur (Programmierer ←→ Berater/Prüfer)
- Die Teilergebnisse werden stufenweise gesichert
- Die Studierenden führen ein Arbeitsprotokoll (*LS-MyContactBookApp-Protokoll.xlsx*)
- Zu den Übungen ist ein fortlaufendes Textdokument pro Partnergruppe für die Ergebnisse zu pflegen
- Informationsaustausch und damit Wissenstransfer innerhalb der Lerngruppe ist sehr erwünscht

5 Basisschritte

5.1 Funktionale Anforderungen

Das UML-Anwendungsfalldiagramm (Bild1 s.u.) zeigt die geforderten Funktionen des Programms. Dieses Diagramm ist für den deutschsprachigen Anwender angefertigt. Ein Leser ohne Programmierkenntnisse kann es verstehen.

Recherche:
Recherchieren Sie zu UML: Wofür steht UML? Welches ist die aktuelle Version der UML? Wie viele Diagrammarten gibt es zurzeit? Zu welcher Kategorie gehören Anwendungsfalldiagramme der UML?

Übrigens:
Ein Anwendungsfalldiagramm soll relevante Rollen (hier Anwender) und Vorgänge zeigen aber keine Abläufe mit Reihenfolgen und Kausalitäten darstellen. Für die Darstellung von Ablaufdetails sind andere Diagramme in der UML verantwortlich.

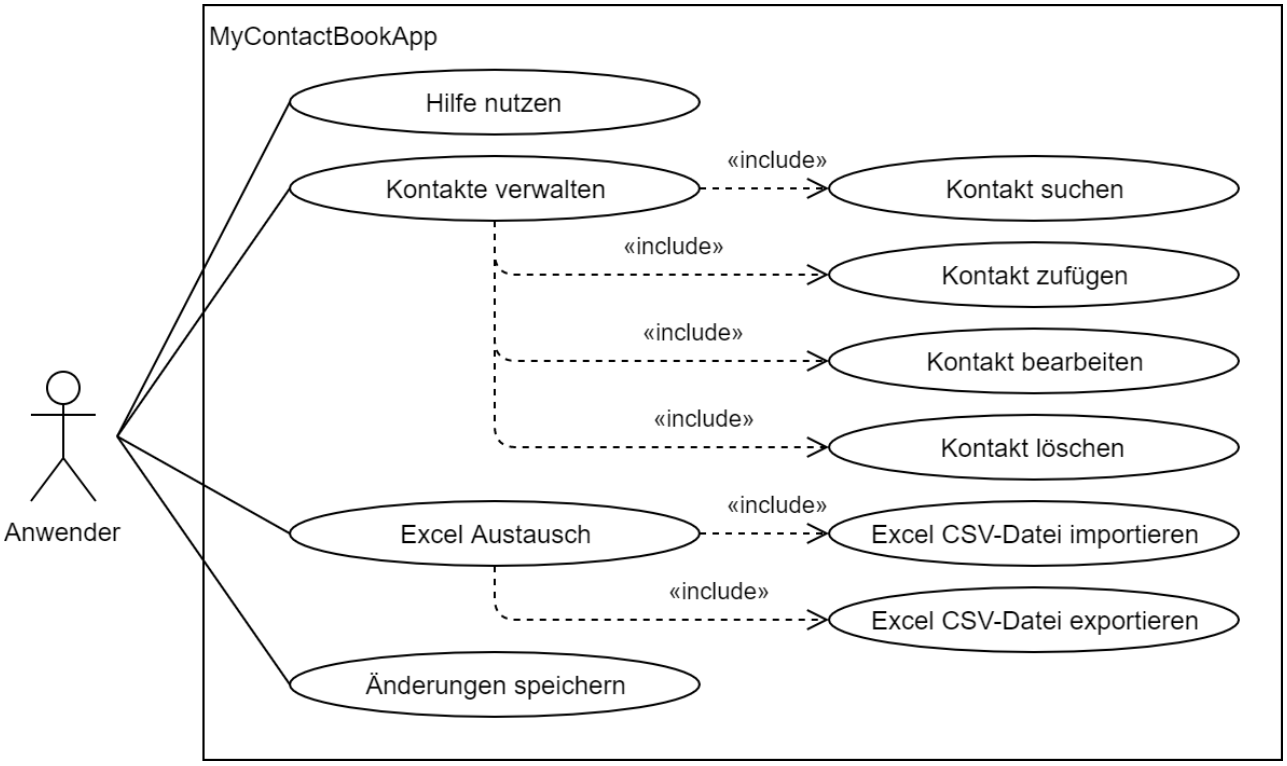


Bild 1 – Anwendungsfalldiagramm

5.2 Daten der Anwendung

Kontaktdaten sollen nach Vor- und Nachname geführt werden. Beliebige weitere Attribute sollen zusätzlich als Textwerte möglich sein, aber ohne geforderte Anzahl oder vorgegebenen Zweck. Denkbar sind damit nach Bedarf etwa Texte wie Telefon, Email, Adresse oder Notizen:

Kontakt als EBNF dargestellt:

```
Kontakte = {Kontakt};
Kontakt = (Vorname, Nachname) | Vorname | Nachname, {Attribut};
```

Recherche:
Recherchieren Sie BNF und EBNF im Vergleich, entschlüsseln Sie die Kontaktdefinition oben und notieren Sie diese dann in Form der BNF.

5.3 Methodenstruktur

Die Funktionen des Programms werden in eigenen Methoden implementiert. Die Hauptmethode *Main( )* beinhaltet eine Eingabemöglichkeit für Suchtext oder Kommandooptionen, über welche die Funktionen des Programms durch den Anwender genutzt werden können. Die in Tabelle 1 dargestellten Methoden sind vorgegeben und entsprechend den folgenden Lernschritten zu realisieren. Die weitern Kapitel beschreiben dabei die Anforderungen an die einzelnen Methoden. Dort kommen noch teilweise weitere Methoden hinzu.

Funktion im Anwendungsfalldiagramm	Methode im Quellcode	
---	static void Main( )	Hauptprogramm mit Startbild
Hilfe anzeigen	static void ShowHelp( )	Bedienungshilfe
Kontakt zufügen	static void ContactAdd( )	
Kontakt suchen	static void ContactSearch(string pattern)	Suchtext → Ergebnisliste
Kontakt bearbeiten	static void ContactEdit(int index)	Index der Anzeige
Kontakt löschen	static void ContactRemove(int index)	Index der Anzeige
Excel CSV-Datei importieren	static void ExcelImport(string path)	
Excel CSV-Datei exportieren	static void ExcelExport(string path)	
Änderungen speichern	static void SaveAll( )	Auf Wunsch des Anwenders
Daten der letzten Sitzung laden	static void LoadAll()	Beim Start des Programms

Tabelle 1 – Programmfunktionen und Methodenstruktur

5.4 Programmerstellung

Aufgabe:
Erstellen Sie ein neues *.NET Core C# Konsolenprojekt*.
Das Projekt soll den Namen ***MyContactBookApp*** bekommen.

Lernen Sie das.NET Core CLI (Command Line Interface) „*dotnet.exe*“ kennen und nutzen Sie es zunächst ohne Visual Studio wie folgt:

- Verzeichnis *MyContactBookApp* anlegen und dort eine Kommandozeilenbox (cmd.exe) öffnen
- Befehl **dotnet -h**“ zeigt an, was dotnet.exe alles anbietet
- Befehl **dotnet new console -lang C#** ein neues .NET Core Projekt wird dort erzeugt
- Befehl **dotnet build**“ das Projekt wird dort übersetzt
- Befehl **dotnet run**“ das Projekt wird dort gestartet und ggf. vorher kompiliert
- Explorerdoppelklick auf Datei *MyContactBookApp.csproj* startet Visual Studio mit dem Projekt
- Beim Speichern erstellt Visual Studio noch die altbekannte Solution-Datei *MyContactBookApp.sln*

Hinweis: Wir werden ab jetzt nicht mehr das herkömmliche *.NET Framework*, sondern das plattformunabhängige ***.NET Core*** *Framework* nutzen. Auf den Schulrechnern ist es bereits als Version 2.1 installiert. Sie müssen es ggf. auf Ihrem eigenen System mit dem *Visual Studio Installer* nachinstallieren. Höhere Versionen sind natürlich bereits ok. Dez. 2019 erwarten wir Version 3.1 LTS (Long Term Support). Siehe auch https://docs.microsoft.com/de-de/dotnet/core/ (ggf. auch mit „en-us“ statt „de-de“)

Aufpassen:
In Visual Studio sollten Sie nur die Dateien editieren, die zum Projekt gehören. Gern ist mal noch eine cs-Datei geöffnet, in der man arbeitet und sich wundert warum beim Debuggen der Code nicht erfasst wird.
Tip: Sicherheitshalber alle cs-Dateien schließen und per Projektmappenexplorerfenster gezielt öffnen.

Übung:
Legen Sie eine Skriptdatei (=Textdatei für den Kommandointerpreter) mit dem Namen **run.cmd** im Projektverzeichnis an, die den Text **@dotnet run** enthält. Wenn Sie diese Skriptdatei im Explorer doppelklicken, wird das Programm gestartet und ggf. vorher kompiliert.

5.5 Hauptprogramm und Hilfefunktion

Nach dem Starten der Anwendung soll folgende einfache Oberfläche erscheinen, von der aus der Anwender später direkt Kontakte suchen kann oder Programmfunktionen per Kommandooption nutzen kann. Der Rahmen des Konsolenfensters trägt den Titel des Programms. Ebenso erscheint eine Titelzeile im Textbereich.

Zunächst werden die Hilfefunktion, das Programmende und ein Entwicklungshinweis ausgegeben, wie es aus dem folgenden Bild 2 ersichtlich werden soll:

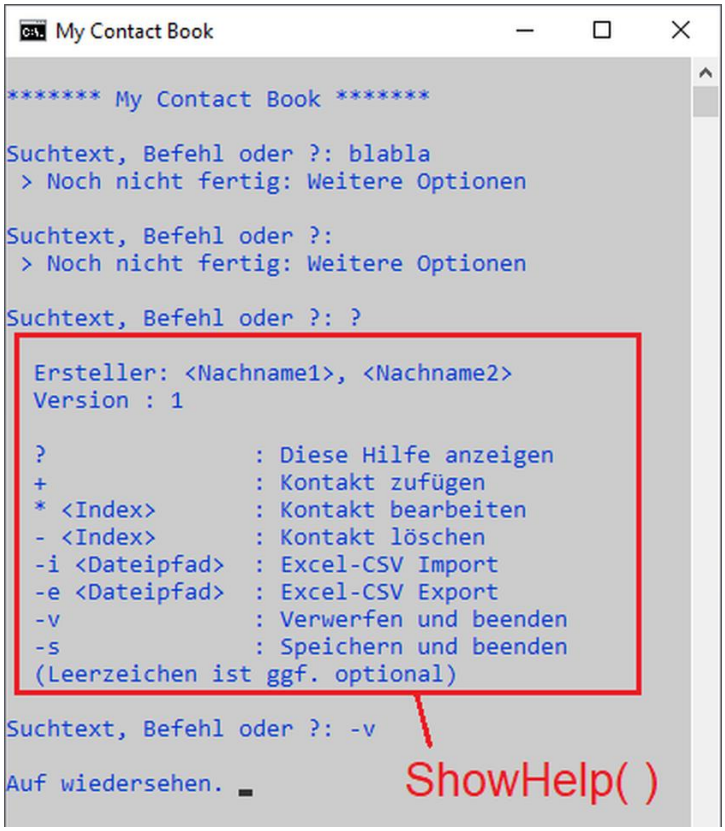


Bild 2 – Startbild und Hilfefunktion

Mit dem Kommandosymbol „?“ erhält der Anwender eine Hilfefunktion über alle Möglichkeiten.
Ferner kann der Anwender hier das Programmierteam (Ihre Namen) sehen und die aktuelle Version der Software, die mit 1 beginnt und mit jedem lauffähigen Zwischenstand hochgezählt werden soll.

Für die Anzeige der Hilfe soll die Methode *ShowHelp( )* s.o. in Tabelle 1 implementiert werden und aus der Hauptmethode *Main( )* aufgerufen werden, wenn der Anwender die Option „?“ mit der Eingabetaste abschickt.

Das Bild 2 zeigt den Verlauf. Dort sind auch weitere erste Programmreaktionen zu sehen.
Nach der Abschiedsnachricht muss der Anwender abschließend die Eingabetaste drücken.

Aufgabe:
Implementieren Sie die Methoden *Main( )* und *ShowHelp( )* geeignet, bis eine Interaktion à la Bild 2 möglich ist.

Programmierstil:
Methoden werden stets vorne groß geschrieben in der Schreibweise ***PascalCase*** *(=upper camel case)*. Dagegen werden Variablen klein geschrieben ***camelCase*** *(=lower camel case)*. Beides mit Wortgrenzen durch Großbuchstaben hervorgehoben.
In unserem Programm alle Variablen (später in der Objektorientierung nur private).

5.6 Alle Methoden anlegen mit Dummyfunktion

Aufgabe:
Implementieren Sie die restlichen Methoden aus Tabelle 1 mit Dummyfunktion so, dass sie ggf. beim Aufruf eine Hinweiszeile wie "Noch nicht fertig: …" ausgeben. Damit sind diese schon aufrufbar, bevor Sie fertig entwickelt sind.

5.7 Eingabefunktion

Ein Kontakt soll als *String-Array* **`string[ ]`** realisiert werden. Das erste Element soll den Vornamen, das zweite Element soll den Nachnamen und alle weiteren sollen Elemente sollen die optionalen Attributwerte, falls vorhanden, enthalten. Siehe auch oben die EBNF für Kontakte.

Vorname oder Nachname darf entfallen, also dann ein leerer *String* im entsprechenden Array-Element.

Recherche zu Array und String:

a) Was sind Arrays in C#?
b) Was bedeutet dabei typisiert?
c) Welche elementaren Datentypen gibt es in C# außer String?
a) Was ist der Unterschied zwischen *string* und *String* in C#?
b) Was bedeutet das C# Schlüsselwort *null*?
c) Welcher Unterschied besteht zwischen *null* und Leertext?
d) Wie bestimme ich die Anzahl der Elemente eines Arrays?
e) Welche Funktion hat die String-Methode *IsNullOrWhiteSpace()*?
f) Wobei helfen die Methoden String.Trim( ), String.TrimStart( ), String.TrimEnd( )?
g) Was macht ein Aufruf von s.Trim(' ', '\t')?
h) Was ist ein *Jagged-Array* im Gegensatz zu einem 2-dimensionalen Array?

Die Methode *ContactAdd( )* soll intern eine weiterere zu entwickelnde Methode *static string[ ] ContactRead( )* aufrufen, die den Anwender einen Kontaktdatensatz eingeben lässt und diesen dann als Array zurückgibt.

Aufgabe a):
Implementieren Sie eine Methode *ContactRead( )* geeignet wie folgt und integrieren Sie deren Aufruf in Methode *ContactAdd( )*. Die interne Funktion besteht aus drei Teilen:

- Vorname und Nachname vom Anwender einfordern (eins davon darf Leertext bleiben)
- Weitere Attribute eingeben, bis der Anwender die Eingabetaste ohne Texteingabe drückt.
- Übernehmen der Werte in ein Array der entsprechend notwendigen Länge.
  (Hinweis: Ein Puffer-Array wird benötigt, wegen der zunächst unbekannten Länge)

Alle Benutzereingaben sollten von umschließenden Leerzeichen (Space, Tab) befreit werden.
Wenn der Anwender „+“ (und Eingabetaste) eingibt, soll nun über den Aufruf von Methode *ContactAdd( )* die Methode **`static string[ ] ContactRead( )`** ablaufen. Prüfen Sie das Ergebnis mit dem Debugger.

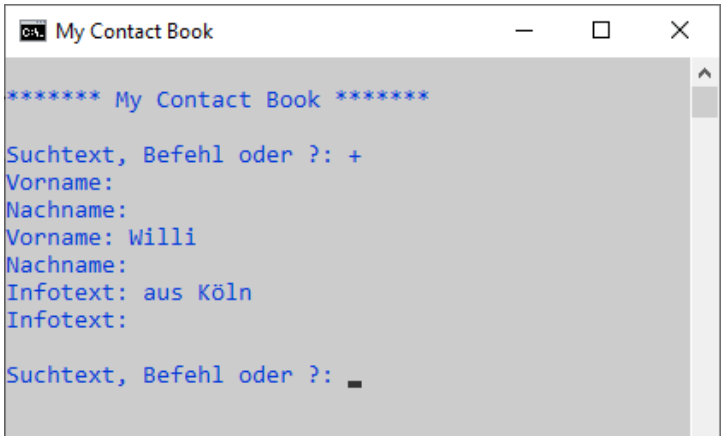


Bild 3 - Eingabe eines Kontaktes

Der Anwender vollzog hier (Bild 3) zwei Leereingaben, dann Vorname ohne Nachname gefolgt von Infotext und abschließende Leereingabe für Ende des Kontaktdatensatzes.
Rückgabe der Methode dann entsprechend: { "Willi ", " ", "aus Köln" }.

Aufgabe b):
Eine globale Array-Variable soll innerhalb der Methode *ContactAdd( )* den durch den Aufruf von *ContactRead( )* erhaltenen Datensatz aufnehmen. Ansatz: Jagged-Array. Das Array muss groß genug sein für eine noch unbekannte Menge möglicher Datensätze.

Lösungsansatz:
```
static string[][] contacts = new string[100][]; // später höher setzen z.B. 1000
contacts[count++] = ContactRead();
```

Fragen dazu:
Verbalisieren Sie den Unterschied zwischen globalen und lokalen Variablen.
Und warum müssen die Variablen **`contacts und count`** gobal sein?
Was genau macht die Anweisung **`contacts[count++] = ContactRead();`**?

Aufgabe c):
Setzen Sie im Code den Anfangswert von **`count`** auf **`100`** oder höher. Sie bekommen dann beim Anlegen eines neuen Datensatzes eine **`IndexOutOfRangeException`**. Bauen Sie ein Tor auf mit „**`try {…} catch {…}`**“, um das Problem zu verhindern und um den Anwender geeignet zu informieren. Thema Exception-Handling.

Nutzen **`Sie dabei contacts.Length ,um das Limit anzuzeigen.`**
So soll es aussehen:

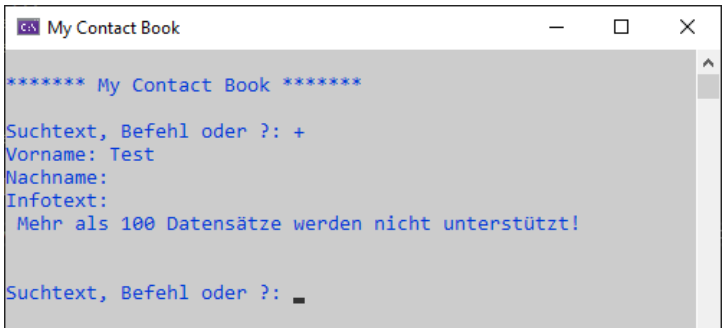


Bild 4 - Datenüberlauf anzeigen

Wenn die Lösung funktioniert, setzen Sie im Code den Anfangswert von Variable **`count`** zurück auf **`0`**;

5.8 Anzeigefunktion

Nun müssen die Datensätze auch übersichtlich als Liste zu sehen sein. Wenn der Anwender einen Suchtext oder auch keinen Text (und Eingabetaste) eingibt, wird automatisch die Suche durchgeführt und das Ergebnis angezeigt. Ohne Suchtext sollen natürlich alle Datensätze angezeigt werden. Im folgenden Bild sehen Sie, wie zwei Datensätze nach ihrer Eingabe so dann aufgelistet werden:

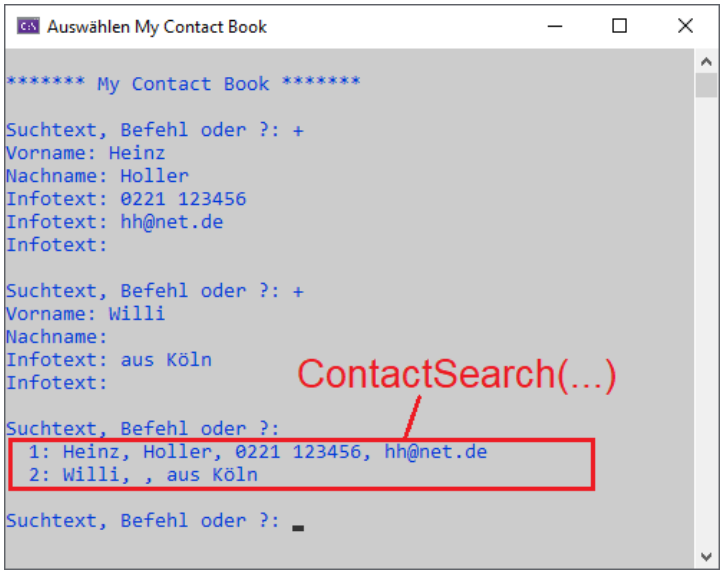


Bild 5 - Listenanzeige

Sie sehen, dass jeder Datensatz durch eine laufende Nummer gefolgt von einem Zeilentext für jeden Kontakt repräsentiert wird.

Ansichtssache:
Beachten Sie auch, dass Anwender als Nichtprogrammierer gerne von 1 bis n zählen, wohingegen Programmierer Datensätze gerne von 0 bis n-1 indizieren.

Aufgabe a):
Da wir später beim Speichern auch noch Textzeilen für einen Kontaktdatensatz erzeugen wollen, ist es nützlich eine neue Methode *ContactToString(…)* zu realisieren, die diese Aufgabe erfüllt - also folgendes liefert:

EBNF: Kontakttext = Attribut, { Trenntext, Attribut }

Dabei sollen der Kontaktdatensatz und der Trenntext als Methodenparameter übergeben werden.
Im Zusammenhang mit der Ausgabe oben würde die Methode etwa so aufgerufen werden:

```
string text = ContactToString(contact, ", ");
```

Das Zusammenfügen des Textes aus seinen Teilen kann entweder mit dem + Operator erfolgen oder mit Hilfe der Klasse *StringBuilder*. Hier Anregungen dazu:

```
using System.Text;
StringBuilder sb = new StringBuilder();
sb.Append(text1);
sb.Append(text2);
string text = sb.ToString();
```

Probieren Sie zu Übungszwecken beides aus. Außerdem müssen Sie natürlich entscheiden, welche Schleifenart am besten geeignet ist, um die Attribute des Kontaktes dabei zu durchlaufen.

Aufgabe b):
Um die Datensätze listenartig in Methode `ContactSearch( )` auf der Konsole anzuzeigen, eignet sich günstig eine formatierte Textausgabe wie folgt: `Console.WriteLine("{0,3}: {1}", i + 1, text);`
Der Erste Platzhalter bekommt die Mindestbreite 3 Zeichen im Konsolenbild.
Vernachlässigen Sie den Suchparameter zunächst, so dass stets alle Datensätze angezeigt werden.
Wenn Ihr Ergebnis die Kontakte wie in Bild 5 ausgibt, klopfen Sie sich auf die Schulter.

5.9 Datensatz editieren

Eine Editionsfunktion (Bearbeitungsfunktion) für einen bestehenden Datensatz kann in einem Konsolenprogramm sehr aufwendig sein. Da wir aber schon die brauchbare Methode *ContactRead( )* für die Eingabe der Kontaktdaten haben, können wir diese wiederverwenden. Dabei soll einfach die Kontaktliste am zu editierenden Index einen neuen Datensatz erhalten, der vom Anwender mit *ContactRead( )* neu eingegeben werden kann. Dies ist für den Anwender nicht ganz edel, da er alle Werte des Datensatzes neu eingeben muss. Dafür ist der Programmieraufwand sehr überschaubar. Wir entscheiden uns deswegen hier für eine wirtschaftliche Priorität.

Kernfunktionalität also: `contacts[index] = ContactRead();`

Natürlich muss der Index sicher aus dem Optionskommando, das der Anwender eingegeben hat, ermittelt werden. Dabei sind Zeilennummer (1…n) und Index (0…n-1) zu unterscheiden.

Coding Tip zum Optionskommando:

```
if (input.StartsWith("*")) ...
```

Coding Tip zu out-Parametern:

```
int num
if (int.TryParse(input.Substring(1), out num)) ...
// einfacher als Einzeiler
if (int.TryParse(input.Substring(1), out int num)) ...
```

Aufgabe:
Realisieren Sie die Bearbeitungsfunktion geeignet im Programm

6 Daten Speichern

6.1 Betrachtungen zu Textdaten

Textdaten müssen im Arbeitsspeicher und auf Datenträgern effizient gespeichert werden. Texte (Datentyp String) bestehen aus einzelnen Buchstaben (Datentyp *Char*) und müssen dafür ein geeignetes Speicherformat aufweisen.

Recherche:
Als Entwickler müssen Sie bei der Verarbeitung von Zeichen und Texten einordnen können, wie Textdaten für Speicherprozesse kodiert werden können. Das übergeordnete Thema heißt Zeichensätze. Versuchen Sie zu folgenden Kürzeln Zusammenhänge, Gemeinsamkeiten und Unterschiede zu ermitteln. Was steckt dahinter?

ASCII, 7bit-ASCII, 8bit-ASCII, Windows-1252, ISO 8859-1, Unicode, UTF8, UTF16

Info:
Die .NET Plattform speichert Texte im Arbeitsspeicher in einem auf **zwei** Byte beschränkten UTF16 Format.
Der Datentyp *char* (entspricht *Char*) definiert ein Zeichen als Einheit aus zwei Bytes.
Der Datentyp string (entspricht String) organisiert *Strings* als Verkettungen solcher *Char*-Werte.
String-Variablen können wie Char-Arrays angesprochen werden. Etwa so:
`char c = "Köln"[1]; // => 'ö'.`

Zeilenumbrüche:
Zeilenumbrüche werden in Texten durch Sonderzeichen repräsentiert. Historisch bedingt gibt es verschiedene Formate:

Betriebssystem	Zeichensatz	Abkürzung	Code Hex	Code Dezimal	Escape-Sequenz
Unix, Linux, Android, macOS, AmigaOS, BSD, weitere	ASCII	LF	0A	10	\n
Windows, DOS, OS/2, CP/M, TOS (Atari)		CR LF	0D 0A	13 10	\r\n
Mac OS Classic, Apple II, C64		CR	0D	13	\r
AIX OS & OS/390	EBCDIC	NL	15	21	\025

Tabelle 2 - Quelle: https://de.wikipedia.org/wiki/Zeilenumbruch

Text mit Zeilenumbruch:

Üblich in C#:	"Alfa**\n**Beta"	// Text mit Zeilenvorschub (Line Feed **LF**)
Nicht selten in C#:	"Alfa**\r\n**Beta"	// Text mit Wagenrücklauf (Carriage Return **CR**)
Unüblich in C#:	"Alfa**\r**Beta"	// Text mit **CR** und **LF**

Text mit Tabulatorzeichen:
"Alfa**\t**Beta" // Text mit Tabulatorzeichen (Tab, Code 9)

Zeilenumbruch in der Konsole – vier Varianten:
```
Console.WriteLine("Hallo"); // Text und Zeilenumbruch
Console.Write("Hallo" + Environment.NewLine); // Sytemkonstante nutzen
Console.Write("Hallo\r\n"); // Bei Windows Systemen
Console.Write("Hallo\n"); // Gleiche Erscheinung auf dem Bildschirm
```

Wenn Sie das verwirrend finden … willkommen im Club. Tatsächlich muss der Entwickler bei der Verarbeitung von Textdaten immer diese historisch bedingt unterschiedlichen Varianten bedenken und ggf. programmiertechnisch berücksichtigen.

6.2 Speichern der Daten

Nun soll es praktisch werden. Zum Speichern der Daten sollen die Kontakte als Textzeilen ausgegeben werden. Dabei sollen folgende Anforderungen gelten:

- Erste Zeile soll den Text „*MyContactBookApp*“ als Programmerkennung enthalten
- Dann folgen alle Kontakte als Zeilen mit Tabulator (s.o.) als Trennzeichen
- Die Attributwerte dürfen daher keine Tabulatoren enthalten!
- Zeilenumbrüche sollen durch *WriteLine()* oder durch *Environment.NewLine* (s.o.) erfolgen

Nun fragen Sie sich, wie die Daten in die Datei kommen? Das Prinzip kennen Sie bereits von der Konsole: Mit den Methoden *Write( )* und *WriteLine( )* schreiben Sie fortlaufend Daten auf/in ein Textmedium, das diesmal kein Konsolenfenster ist, sondern eine Datei ist. Die Klasse *StreamWriter* ist Ihre Programmierschnittstelle dazu. ‚Steam‘ deutet schon an, dass Daten strömen. Schauen wir uns typischen Code an:

```
using System;
using System.IO;
using System.Text;
...
string desktop = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
string file = Path.Combine(desktop, "daten.mcb");
using (StreamWriter sw = new StreamWriter(file, false, new UTF8Encoding(true)))
{
  sw.WriteLine("MyContactBookApp");
  sw.WriteLine("und weiter geeignet programmieren");
}
```

Aufgabe:
Übernehmen Sie diesen Code in die Methode *SaveAll()* geeignet und sorgen Sie dafür, dass der Anwender mit der korrekten Option diese Methode zum Speichern aufrufen kann. Suchen Sie dann in der Entwicklerhilfe von Microsoft oder in einem Programmierbuch Beschreibungen zu den Elementen des Codes (Methoden, Klassen, Eigenschaften), um den Ablauf zu verstehen. Letztlich ist der Code aber auch einigermaßen selbsterklärend. Es ist auch interessant, mit dem Debugger schrittweise durch diesen Code zu laufen und sich die Variablenwerte zur Laufzeit anzuschauen.

Trotz Ihrer Recherche sei hier kurz erläutert:
Wir speichern hier Text überschreibend in eine Datei im UTF8 Format mit einer Formatkennung BOM (Byte Order Mark). Das sind vorangestellte Bytes in der Datei, an denen die lesende Software erkennen kann, dass der folgende Text im UTF8 Format vorliegt. Dies sollte heutzutage Standard sein. Mit dem **`using {...}`** Block wird sichergestellt, dass das erstellte *StreamWriter*-Objekt alle internen Systemressourcen für den Dateizugriff wieder frei gibt. (Für Fortgeschrittene: Es wird am Ende des Blocks die *Dispose( )*-Methode von *StreamWriter* sicher aufgerufen).

Die Klasse *Path* im Namespace *System.IO* ist sehr praktisch für Dateipfadarbeiten mit u.a. den Methoden:
→ *Combine(), GetDirectoryName(), GetFileName(), GetFileNameWithoutExtension(), GetExtension().*

Das Resultat ist eine Textdatei, die Sie z.B. mit Notepad++ anschauen können, auch um zu prüfen, ob die BOM Kennung der Datei erkannt wird. Siehe Screenshot hier. Probieren Sie es dann auch aus.

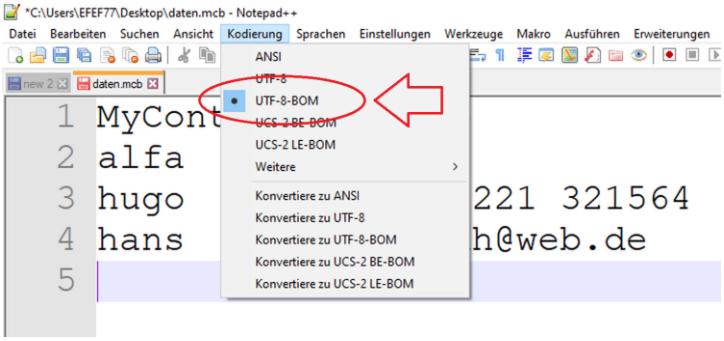


Bild 6 - Notepad++ erkennt BOM Kennung

Die Methode *ContactToString(…)* aus dem oberen Kapitel hilft Ihnen hier wieder die Kontakte zeilenweise in Text umzuwandeln und damit die Speicherfunktion fertig zu stellen - diesmal mit einem anderen Trenntext.

Es wäre auch nett, wenn der Anwender eine Quittung zu sehen bekommt, dass erfolgreich gespeichert wurde. Beschreiben Sie, was folgender Code macht:

```
Console.WriteLine(
      "{0} {1} gespeichert", count,
      count == 1 ? "Datensatz wurde" : "Datensätze wurden");
```

Kniffelig:
Wie können Sie sicherstellen, dass die Speicherdatei keine unzulässigen Tabulatoren enthält. Diskutieren Sie verschiedene Lösungsmöglichkeiten und arbeiten Sie die sinnvollste in Ihr Programm ein.

6.3 Laden der Daten

Nun soll das Programm direkt beim Start schon die Daten laden, falls die Datei mit Kontaktdaten aus einer früheren Sitzung bereits besteht und dort die erste Textzeile die oben geforderte Programmerkennung „*MyContactBookApp*“ enthält. Der notwendige Programmcode ähnelt ein wenig dem Code für das Speichern:

```
using (StreamReader sw = new StreamReader(GetDataFile(), true)) {
  if (!sw.EndOfStream) {
    if (sw.ReadLine() == "MyContactBookApp") { … usw. … }
  }
}
```

Hier jetzt mit einer neuen Methode *GetDataFile()*, die redundanten Code erspart, der auch bei der Speicherfunktion ersetzt werden kann.

```
static string GetDataFile() {
  string desktop = Environment.GetFolderPath(Environment.SpecialFolder.Desktop);
  return Path.Combine(desktop, "daten.mcb");
}
```

Interessant hier ist, dass der zum *StreamWriter* passende *StreamReader* durch den zweiten Parameter **`true`** die Vorgabe bekommt, das Encoding aus der BOM Kennung (Byte Order Mark) selbst zu bestimmen.

Ferner kann bzw. sollte vor dem Lesen mit *ReadLine()* über Eigenschaft *EndOfStream* geprüft werden, ob noch Daten zum Lesen bereitstehen. Leseversuche würden andernfalls einen Fehler (Exception) werfen. Durch ein aufeinanderfolgendes Aufrufen von *ReadLine()* werden die verfügbaren Textdaten immer bis zum nächsten Zeilenumbruch (bzw. gegebenen Falls bis zum Dateiende) gelesen, wobei die Zeilenumbruchzeichen selbst beim Lesen intern verbraucht werden. Die Datei soll durch eine geeignete Schleife bis zum Ende ausgelesen werden.

Zerlegen einer Zeile:
Eine Textzeile kann mit der Methode *Split(…)* unter Angabe eines Trennzeichens zerlegt werden wie folgt:
`string[] contact = sw.ReadLine().Split('\t');`

Aufgabe:
Mit diesem Wissen können Sie das Einlesen der Daten in die Methode *LoadAll()* siehe Tabelle 1 programmieren und diese beim Programmstart entsprechend aufrufen.

So sollte es für den Anwender z.B. aussehen:

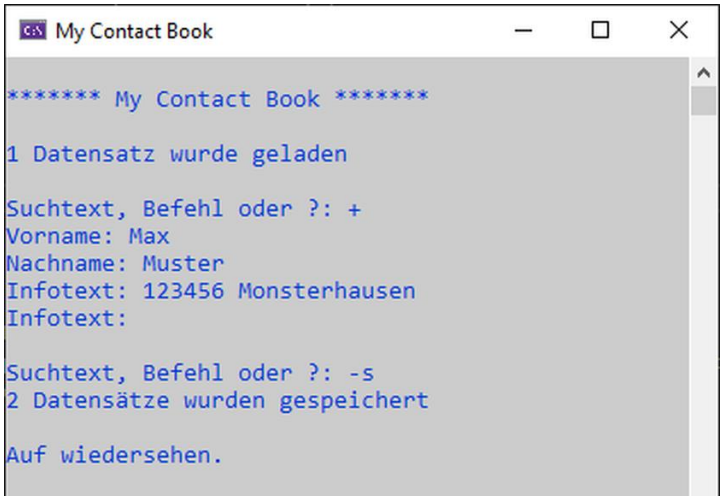


Bild 7 - Laden und Speichern

7 Sonstiges

7.1 Filtern der Ausgabe mit Suchmuster

Da Sie jetzt mehrere Datensätze speichern und laden können, müssen Sie Testdaten nicht jedes Mal neu eingeben. Halten Sie sich eine Datei vor, in der eine geeignete Menge von Testdaten vorliegt.

Jetzt ist also ein guter Zeitpunkt, die Filterfunktion zu realisierten. Methode *ContactSearch(string pattern)* muss nun das Suchmuster auswerten, um die Anzeige zu filtern.

Frage:
Welchen Nutzen hat der Aufruf von ToLower() in folgendem Code:
`bool treffer = attr.ToLower().Contains(pattern.ToLower());`

Hier eine Demonstration der Filterfunktion

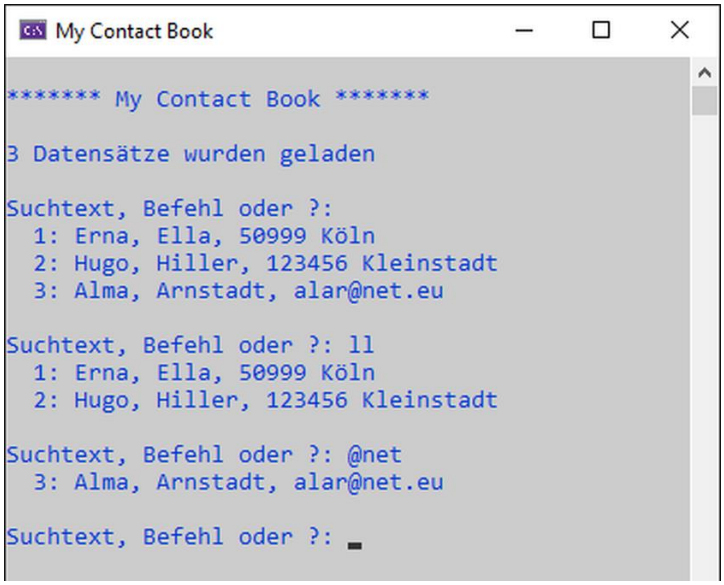


Bild 8 – Filterfunktion

Aufgabe:
Realisieren Sie die Filterfunktion geeignet im Programm.

7.2 Löschen von Daten

Nun soll geübt werden, wie in einem festen Array (keine dynamische Liste) ein Wert dort gelöscht werden kann. Folgendes Bild soll helfen:

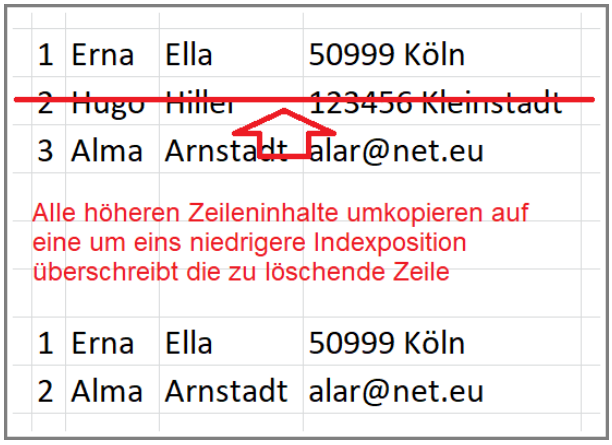


Bild 9 – Löschen durch aufrücken

Folgendes sollte beachtet werden

- Zeilennummer (1…n) und Index (0…n-1) sind zu unterscheiden
- Löschindex muss in einem korrekten Bereich liegen (0…**`count`**) (siehe 5.7)
- Die Zählvariable **`count`** (siehe 5.7) muss am Ende angepasst werden

Für den Anwender ist es sinnvoll, wenn nach dem Löschvorgang die Daten erneut aufgelistet werden, damit die neu entstandene Nummerierung ersichtlich ist. Vielleicht wenden Sie dabei auch das letzte Suchmuster wieder an. Nach dem Löschen kann dafür einfach `ContactSearch(recentPattern)` aufgerufen werden, wobei eine Variable **`recentPattern`** vorzusehen ist, die das Muster der letzten Suche speichert muss.

Die Auswertung der Zeilennummer aus dem Optionskommando kann entsprechend wie bei der Editierfunktion in Kapitel 6 erfolgen.

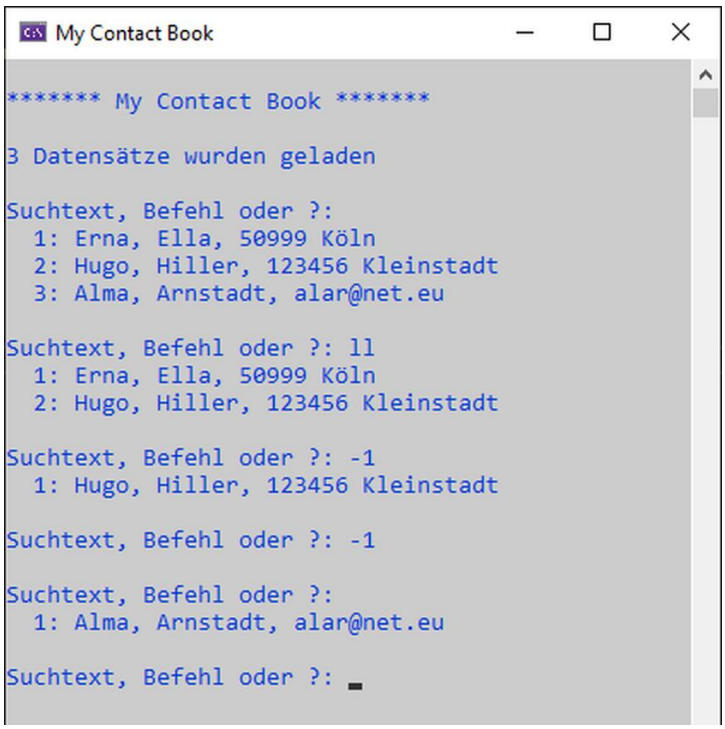


Bild 10 – Löschfunktion in Aktion

Aufgabe:
Realisieren Sie die Löschfunktion entsprechend im Programm

8 Excel CSV

8.1 Excel Importfunktion

Da tabellarische Daten in Excel oft schon existieren und/oder dort auch sehr ergonomisch erstellt und geändert werden können, ist es sehr sinnvoll, wenn Sie über die Kompetenz verfügen, solche Daten auch in einem textbasierten Programm zu verarbeiten.

Beginnen wir mit dem Erstellen von Kontaktdatensätzen mit Excel und speichern diese in eine Textdatei im CSV-Format.

Info:
CSV (Char Separated Values) ist ein tabellarisches Textformat, wo die Spaltenwerte innerhalb der Zeilen mit einem Trennzeichen markiert werden. Trennzeichen sind bei CSV Formaten meist Semikolon, Komma oder Tabulator. Excel verwendet hier konkret das Semikolon.

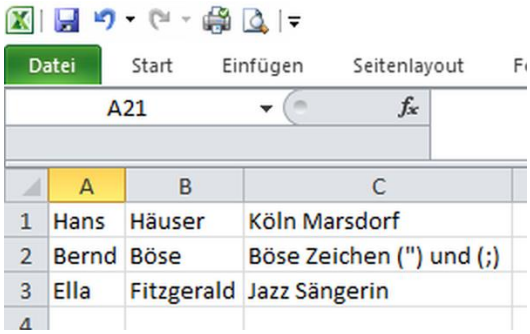


Bild 11 – Daten mit Excel erzeugen

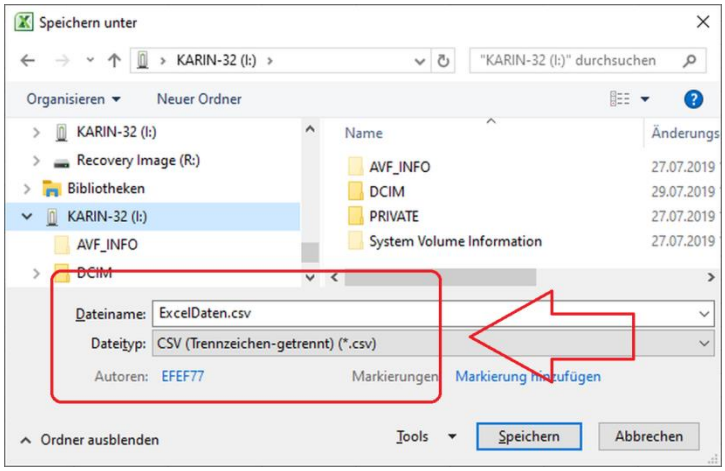


Bild 12 – Daten als CSV –Textdatei speichern

Das Ergebnis in Datei *ExcelDaten.csv* sieht wie folgt aus:

```
Hans;Häuser;Köln Marsdorf
Bernd;Böse;"Böse Zeichen ("") und (;)"
Ella;Fitzgerald;Jazz Sängerin
```

Wenn Sie die Importfunktion *ExcelImport(string path)* vergleichbar mit der Methode *LoadAll()* programmieren aber dabei schon das Semikolon als Trennzeichen verwenden, werden Sie zu folgendem Ergebnis kommen. Hier (s. Bild 13) wurde die Importfunktion additiv realisiert. Also die bestehenden Datensätze bleiben erhalten:

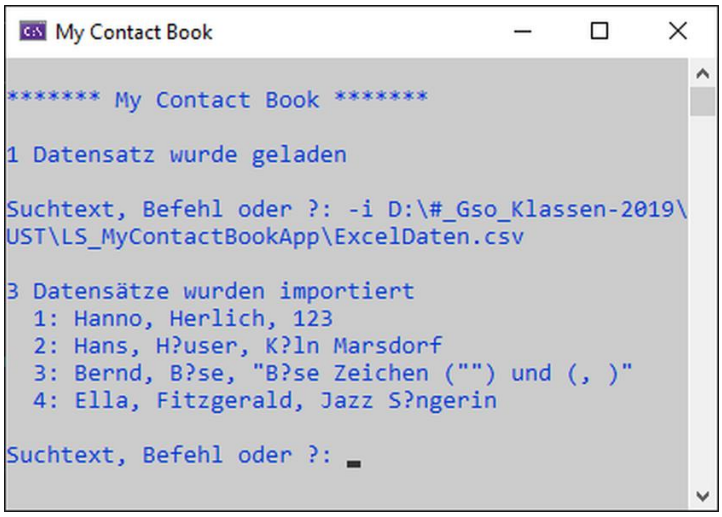


Bild 13 – Fast schon brauchbare Import

Problem1:
Die Umlaute. Lösung: Excel unter Windows verwendet als Textdateiformat den für Westeuropa üblichen Zeichensatz ISO-8859-1. So bekommen Sie das Problem in den Griff

```
StreamReader sw = new StreamReader(path, Encoding.GetEncoding("iso-8859-1"))
```

Problem2:
Wenn in einer Zelle besondere Zeichen wie (") oder (;) vorkommen, setzt Excel in der CSV-Datei um den Zellenwert herum doppelte Anführungszeichen. Ferner werden innerhalb der Zelle dann die (")-Zeichen doppelt codiert, wie sie es vergleichsweise in Bild 11 und am Inhalt von *ExcelDaten.csv* (s.o.) sehen können.
Leider müssen Sie dieses Problem bei der Zerlegung und Aufbereitung der Zeilentexte beim Import geeignet programmieren. Also los …

Tips:

```
cell = cell.Substring(1, cell.Length-2);
cell = cell.Replace("\"\"", "\"");
```

Aufgabe:
Realisieren Sie die Importfunktion entsprechend im Programm

8.2 Excel Exportfunktion

Beim Export gehen sie einerseits ähnlich wie bei Methode *SaveAll()* vor und nutzen aber auch wieder das Wissen aus Methode *ExcelImport(string path)* über den Zeichensatz ISO-8859-1 und Problem 2, um zu einer für Excel verdaulichen Lösung zu kommen. Die Ergebnisdatei des Exports können Sie im Explorer durch Doppelklick oder Kontextmenü „Öffen mit“ per Excel öffnen und bearbeiten.

Tip:
Die bestehende Funktion ContactToString() könnte wie folgt erweitert werden und bliebe dabei aber abwärtskompatibel (Thema Defaultwerte für Parameter):
`string ContactToString(string[] contact, string separator, bool csv = false)`
Innerhalb der Methode kann der Parameter `csv` dann situativ geeignet verwendet werden.

Aufgabe:
Realisieren Sie die Exportfunktion entsprechend im Programm

9 Weiteres

9.1 Sortierproblem

Diskussionspunkt:
Nun entsteht die Reihenfolge der Kontaktdaten im Array ja eher zufällig durch die Eingaben des Anwenders. Für den Anwender wäre es sicher nützlich wenn die Auflistung sortiert erscheint. Zum Beispiel nach Vorname oder nach Nachname. Sollen, Ihrer Meinung nach, die Daten also erst bei der Auflistung sortiert angezeigt werden, oder sollen sie bereits von Beginn an sortiert gehalten werden? Welche Aspekte sprechen für das eine oder das andere? Verschriftlichen Sie die Überlegung.

9.2 Doublettenproblem

Diskussionspunkt:
Wenn bei der Eingabe oder beim Import Namen und/oder Namenskombinationen aus Vor und Nachname doppelt vorkommen, besteht möglicherweise ein Konflikt. Welche verschiedenen Konfliktfälle sind aus Sicht der Anwender wohl problematisch und welche Lösungsstrategien könnten hier Abhilfe schaffen. Verschriftlichen Sie diese Problematik und Ihre Lösungsvorschläge.

◆