

Grundlagen der Objektorientierung

Objektorientierte Softwareentwicklung –
von der objektorientierten Modellierung bis zum
objektorientierten Programm mit C#

ZERTIFIKATSKURS

WIRTSCHAFTSINFORMATIK

31 Mai 2012

G. Kohlruss, M. Michalowsky, G. Steinkamp

Inhaltsverzeichnis

1. Grundlagen der Objektorientierung	1
1.1 Klasse und Objekt	2
1.2 Grundlegende Datentypen.....	3
1.3 Klassen erzeugen	5
1.3.1 Erzeugen der Klasse Konto und Anlegen von Feldern	6
1.3.2 Properties	7
1.3.3 Der Konstruktor	8
1.3.4 Methoden	11
1.4 Erstellen einer Grafischen Benutzeroberfläche (GUI)	12
1.4.1 Erstellen von Labels.....	12
1.4.2 Erstellen von Textboxen und Buttons	13
1.5 Die Click-Methode	15
1.6 Programmanalyse	16
1.7 Nutzen des Konstruktors zur Objekterzeugung	16
1.8 Variablendeklaration.....	18
1.9 Nutzung von Properties (Set- und Get-Methoden)	18
1.10 Klassendiagramm und C#	19
2. Erweiterung von Klassendefinitionen	24
2.1 Klassendiagramm überarbeiten	24
2.2 Erläuterungen zum Konstruktor	25
2.3 Methoden schreiben und verwenden	26
3. Verzweigungen und Fehlerbehandlung	28
3.1 Entscheidungen treffen und Fehlerbehandlung	28
3.2 Test der selbst definierten Ausnahme	29
3.3 Entscheidungen treffen (Selektion)	31
4. Testen und Dokumentieren	34
4.1 Der Debugger.....	34
4.1.1 Einzelschrittverfahren	34
4.1.2 Haltepunkte	35

4.1.3 Überwachungsfenster	35
4.2 Kommentare erstellen und aufbereiten	36
5. ArrayLists und typisierte Listen.....	40
5.1 Vergleich von typisierten Listen und ArrayLists mit Regalen	40
5.2 Erzeugung von ArrayLists und List<Typ>.....	42
5.3 Tabellarische Darstellung	42
5.4 Wichtige Elemente zur Arbeit mit Auflistungen.....	46
5.5 Vorgehensweise.....	47
5.6 Die Objekte der Auflistung ausgeben	51
5.7 Durchlaufen von Listen mit Hilfe der foreach-Schleife	52
5.8 Erweiterung des Konto-Beispiels	53
5.8.1 Konto anlegen, aber nur falls Kontonummer nicht bereits vorhanden	54
5.8.2 Elemente einer Auflistung suchen	56
5.8.3 Sortieren einer Auflistung	57
5.9 Methoden und Eigenschaften der Klassen ArrayList und List<Typ>	60
6. Vererbung	63
6.1 Pflichtenheft: Kontoverwaltung Sparkasse Kleve.....	64
6.2 Nutzung des Steuerelements DataGridView (Tabellenanzeige).....	68
6.2 Nutzung von RadioButtons und Checkboxes.....	70
6.3 Eine Lösung mit den bisher bekannten Mitteln	71
6.4 Allgemeine Hinweise zur Vererbung	77
6.5 Methoden überschreiben	78
6.6 Die Basisklassenversion einer Methode aufrufen	79
6.7 Abstrakte Klassen.....	80
6.8 Abstrakte Methoden.....	80
6.9 Die Klasse Object	82
6.10 ArrayLists bzw. typisierte Listen und Vererbung	83
6.11 Zugriffsmodifizierer.....	83
6.12 Anwendung auf das Konto-Beispiel	84
6.11 Glossar: Schlüsselworte in C# im Zusammenhang zur Vererbung	87
7. Assoziationen in C#umsetzen	89

7.1 Ausgangsbeispiel: Kontoverwaltung für die Sparkasse Kleve	89
7.2 (mögliche) Umsetzung der Situation in ein Klassendiagramm:	91
7.3 Informationsteil zur Umsetzung von Assoziationen in C#	91
7.3.1 Wie werden Assoziationen in C# umgesetzt?	91
7.3.2 Listobjekte sollen in mehreren Formularen verfügbar sein, wie geht das?96	
7.3.3 Kundenverwaltung ergänzen.....	97
7.3.4 Implementierung von Aggregationen und Kompositionen	98
7.3.5 Exkurs: Weitere Alternativen zur Berücksichtigung verschiedener Leserichtungen am Beispiel des Kontoinhabers.....	104
Anhang I (Vollständige Lösung des Konto-Beispiels mit Vererbung)	109
Anhang II (Lösung des Konto-Beispiels mit Assoziationen)	118
8. Anhang I.....	A

1. Grundlagen der Objektorientierung

In den folgenden Kapiteln soll eine Einführung in die **objektorientierte Modellierung** und **Programmierung** anhand eines Kreditinstituts erfolgen. Für die Modellierung finden die Unified Modeling Language (**UML**) und für die Implementierung die Programmiersprache **C#** (sprich: C-Sharp) Verwendung. Als einführende Entwicklungsumgebung (IDE = Integrated Development Environment) wird **Microsoft Visual Studio 2010** herangezogen.

Für den Einstieg in die objektorientierte Programmierung wollen wir im Folgenden ein stark vereinfachtes Kontenverwaltungsprogramm erstellen. Im ersten Ansatz soll es nur möglich sein Konten zu erzeugen und den Inhalt der erzeugten Konten auszugeben. Im weiteren Verlauf des Skripts wird dieses Beispiel weiter ausgebaut.



Ausgangssituation

Bei der Sparkasse Kleve werden die folgenden Konten geführt.

S

EUR-KONTO

KONTOAUSZUG

5005878

Sparkasse Kleve BLZ 324 500 00

Kontoinhaber

Saldo

2.850,20 € +

Müller, Hans

S

S

EUR-KONTO

KONTOAUSZUG

5005612

Sparkasse Kleve BLZ 324 500 00

Kontoinhaber

Saldo

2.300,00 € -

Schmidt, Lolita

S

S

EUR-KONTO

KONTOAUSZUG

7205820

Sparkasse Kleve BLZ 324 500 00

Kontoinhaber

Saldo

2.850,20 € +

Jansen, Gregor

S

Die Geschäftsführung der Sparkasse Kleve hat die Erstellung einer zeitgemäßen Software zur Verwaltung ihrer Kunden und deren Konten in Auftrag gegeben.

1.1 Klasse und Objekt

Der klassische Ansatz der Programmierung, bei der es nur einfache Datentypen gab, um Daten aufzunehmen (Ganzzahl, Dezimalzahl, Zeichen, Zeichenketten etc.) ist in den vergangenen Jahrzehnten an seine Grenzen gestoßen. Das größte Problem bestand in einer häufig nicht mehr zu überschauenden Komplexität.

Hingegen ist der objektorientierte Ansatz ein neues Konzept der Softwareentwicklung, welches die Objekte der realen Welt in Software-Objekten abbildet und verarbeitet. Durch die Objektorientierung lassen sich komplexe Softwaresysteme wirtschaftlicher erstellen als frühere Konzepte (s. prozedurale Programmierung). Das Konzept wurde zwischen 1970 und 1980 entwickelt. Als erste objektorientierte Programmiersprache gilt Smalltalk-80.

Der „Objekt“-Begriff bildet den Kern des objektorientierten Konzepts. In der realen Welt treten Objekte in Form von Dingen (z. B. Buch, Auto), Personen (z. B. Kunde, Leser) oder Begriffen auf (z. B. Fremdsprache, Krankheit). Moderne Softwareentwickler versuchen, Objekte zu finden, die für eine Anwendung sinnvoll sind. Sie beschreiben, welche Eigenschaften die Objekte haben und welche Aktionen sie ausführen können. Hierbei reduzieren sie das reale Objekt auf die Eigenschaften und Aktionen, die im Zusammenhang der Anwendung maßgeblich sind. (Maßgeblich sind im Zusammenhang mit einer Kontenverwaltung sicher Name und Adresse einer Person, aber nicht die zugehörige Augenfarbe) Sie fassen somit Einheiten aus logisch zusammenhängenden Daten (Zustände) und Operationen (Verhalten) zu Objekten zusammen.

Die Objekte bilden eine geschlossene Einheit (Kapsel), d. h. ihre Daten werden versteckt (Information Hiding), so dass man nicht direkt von außen auf sie zugreifen kann. Dies ist eine wesentliche Voraussetzung für die Selbstverwaltung eines Objektes. Man sagt: Das Objekt kapselt die Daten. Ein Zugriff auf die Objektdaten ist nur möglich, indem man von außen eine Nachricht an das Objekt sendet.

In der folgenden Abbildung wird die Klasse **Konto** mit einigen konkreten Objekten visualisiert. Allerdings ist hier die Klasse noch auf die Eigenschaften reduziert, auf die Darstellung der Aktionen (Operationen) wurde noch verzichtet und die Eigenschaften sind nicht gekapselt.

Die Software für die Kontenverwaltung behandelt jedes Konto als **Objekt** mit

- ⇒ identischen **Attributen** (Kontonummer, Kontoinhaber, Saldo), aber
- ⇒ unterschiedlichen **Attributwerten**.

Die allgemeine, **abstrakte** Beschreibung der Struktur dieser Objekte wird als **Klasse** bezeichnet. Vereinfacht kann man auch von dem **Bauplan** für die Objekte sprechen. Das **Objekt**¹ ist somit eine **konkrete** Ausprägung einer Klasse.

¹ In der Literatur wird gleichbedeutend auch von Instanz gesprochen.

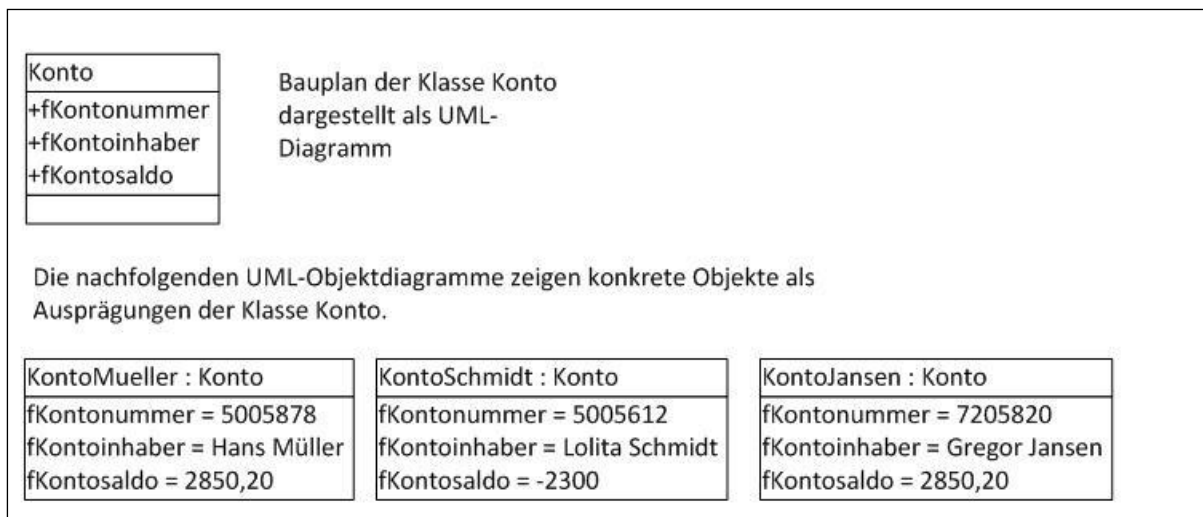


Abbildung 1-1 UML-Klassen und Objektdiagramme

Möchte man die dargestellte Klasse und die zugehörigen Objekte in C# implementieren muss man für jedes Attribut einen Datentyp festlegen. Hiermit wird festgelegt welcher Speicherbereich für ein Attribut reserviert wird.

In der folgenden Grafik werden die obigen Klassen- und Objektdiagramme um die Datentypen (siehe Tabelle auf Seite 4) erweitert.



Abbildung 1-2 UML-Klassen- und Objektdiagramme mit Datentypen

1.2 Grundlegende Datentypen

C# unterstützt die üblichen Datentypensätze. Für jeden von C# unterstützten Datentyp ist ein entsprechender .NET Common Language Runtime-Typ vorhanden. Der Datentyp **int** in C# wird beispielsweise dem Typ **System.Int32** in der Laufzeitumgebung zugeordnet. **System.Int32** kann fast überall dort verwendet werden, wo **int** zum Einsatz kommt. Dieses Vorgehen wird jedoch nicht empfohlen, da es die Lesbarkeit des Codes herabsetzt. Die grundlegenden Datentypen werden in der nachstehenden Tabelle beschrieben. Die Laufzeittypen befinden sich allesamt im System-Namespace der .NET Common Language Runtime.²

² <http://openbook.galileocomputing.de/csharp/kap03.htm>

Typ	Laufzeittyp	Beschreibung	Bereich (Bits)
byte	Byte	Byte-Wert ohne Vorzeichen	0 bis 255
sbyte	SByte	Byte-Wert mit Vorzeichen	-128 bis 127
short	Int16	Short-Wert mit Vorzeichen	-32.768 bis 32.767
ushort	UInt16	Short-Wert ohne Vorzeichen	0 bis 65535
int	Int32	Integer-Wert mit Vorzeichen	-2.147.483.648 bis 2.147.483.647
uint	UInt32	Integer-Wert ohne Vorzeichen	0 bis 4294967295
long	Int64	Langer integer-Wert mit Vorzeichen	-922337203685477508 bis 922337203685477507
ulong	UInt64	Langer integer-Wert ohne Vorzeichen	0 bis 18446744073709551615
float	Single	Gleitkommazahl mit einfacher Genauigkeit	-3.402823e38 bis 3.402823e38
double	Double	Gleitkommazahl mit doppelter Genauigkeit	-1.79769313486232e308 bis 1.79769313486232e308
decimal	Decimal	Zahl mit fester Genauigkeit	$\pm 1,0 \times 10e-28$ zu $\pm 7,9 \times 10e28$
string	String	Eine Folge von Zeichen	
char	Char	Ein einzelnes Unicode-Zeichen	Im Text verwendete Unicode-Symbole (16 Bit)
bool	Boolean	Wahrheitswert	True oder False (8 Bit)
object	Object	Alle anderen Typen stamen von ihm ab, daher kann er für jeden anderen Typ eingesetzt werden.	64 Bit für interne Verwaltungsinformationen + 32 bzw. 64 Bit für den Adressverweis auf die Daten.

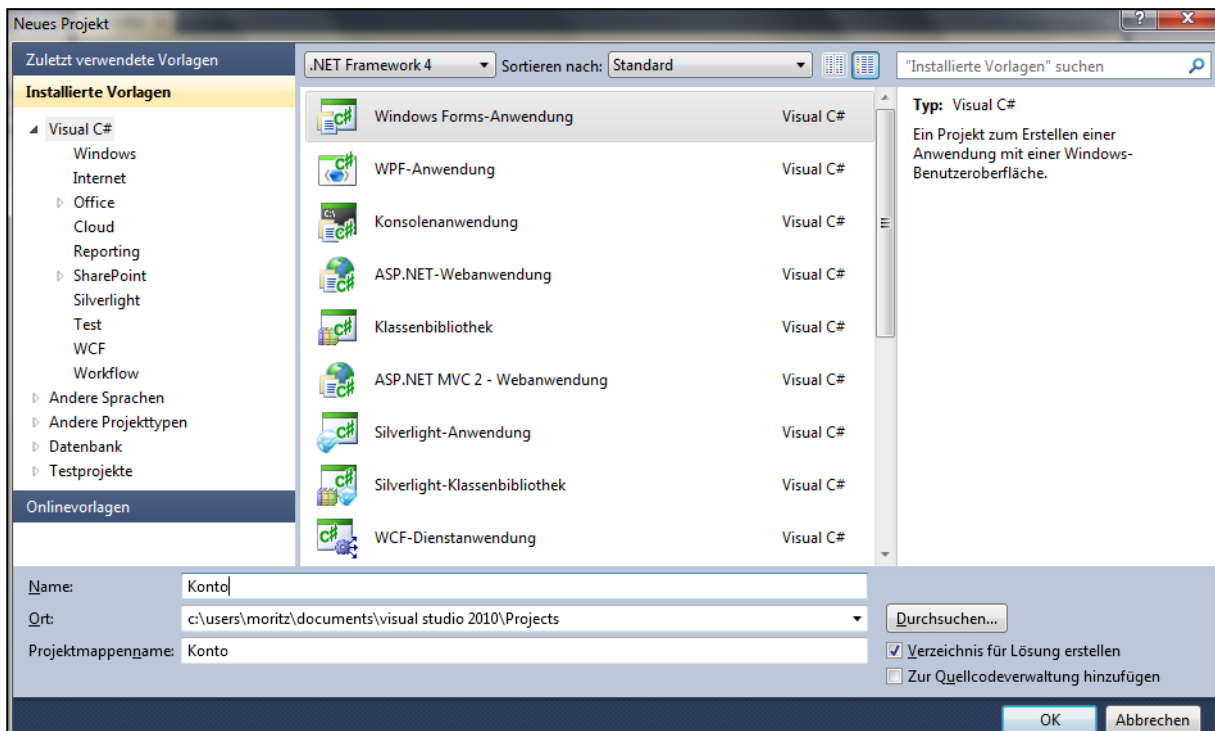
1.3 Klassen erzeugen

Mit der Entwicklungsumgebung *Microsoft Visual Studio 2010* kann die Programmierung nachvollzogen werden:

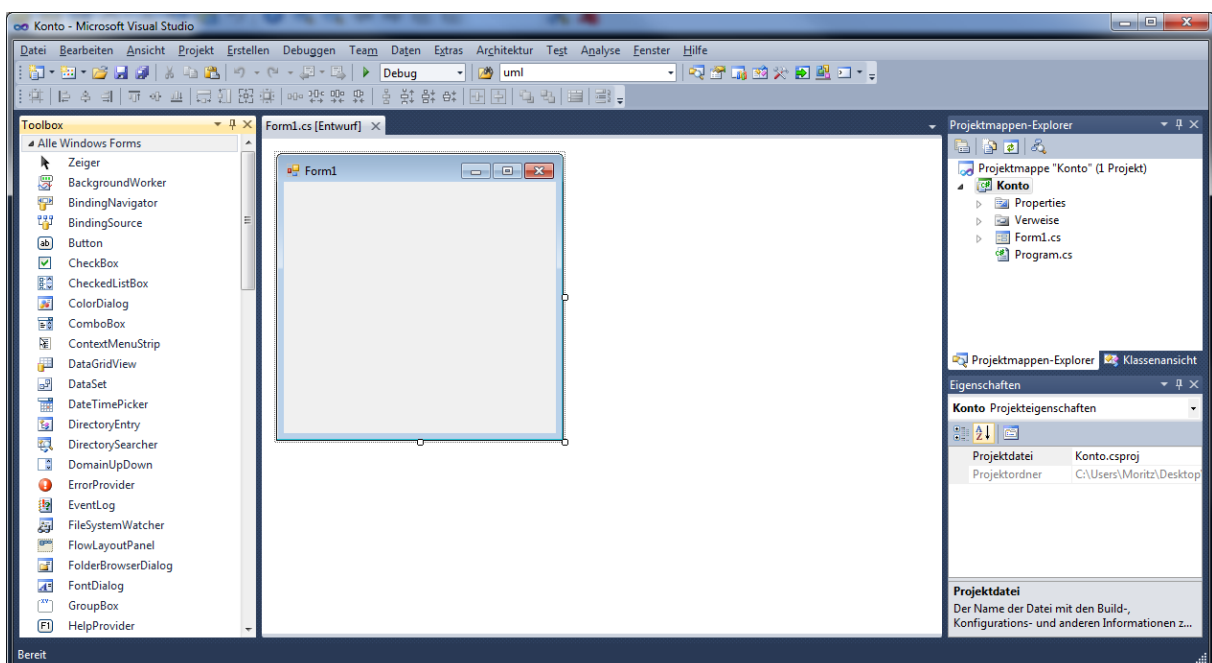


Aufruf des Programms Microsoft Visual Studio 2010

- ⇒ Aufruf des Programms Microsoft Visual Studio 2010
- ⇒ Erstellen eines neuen Projektes über „Projekt neu...“ – evtl. Ändern der Sprache auf C# - Auswählen der **Windows Forms-Anwendung** – Name: „Konto“ eingeben.



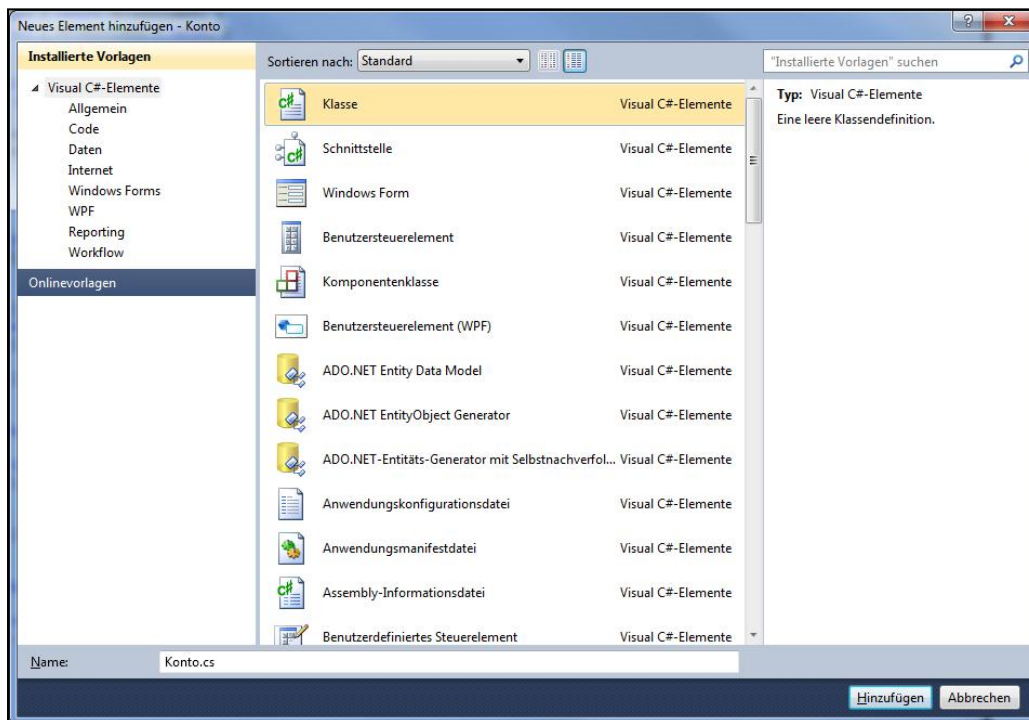
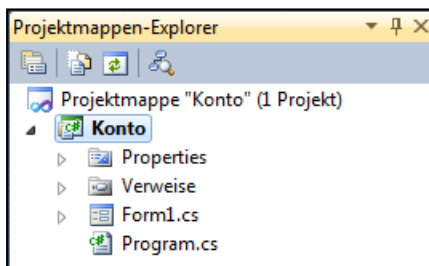
Nach Bestätigen mit <OK> wird in etwa die untenstehende Ansicht dargestellt. Sollte eines der Fenster nicht vorhanden sein, lässt es sich über das Menü <Ansicht> einblenden.



Im linken Bereich des Fensters ist die Toolbox zu sehen. Sie enthält Steuerelemente für die Oberflächengestaltung. Mittig ist das eigentliche Arbeitsfenster zum Gestalten von Oberflächen und Schreiben von Code zu sehen. Rechts oben ist der Projektmappenexplorer angezeigt, der die Gesamtstruktur des Projektes wiedergibt. Das Eigenschaftenfenster darunter ermöglicht Einstellungen für das jeweils aktive Steuerelement in der Entwurfssicht vorzunehmen. Dies ist die Standardeinstellung der Entwicklungsumgebung Visual Studio 2010. Sollte sich eines der Fenster an einer anderen Stelle befinden, kann dieses per Drag & Drop individuell positioniert werden. Mit Hilfe der „Stecknadel“ kann man die gewählte Position fixieren. Sollte eines der genannten Fenster geschlossen sein, kann man diese über das Menü Ansicht hinzufügen.

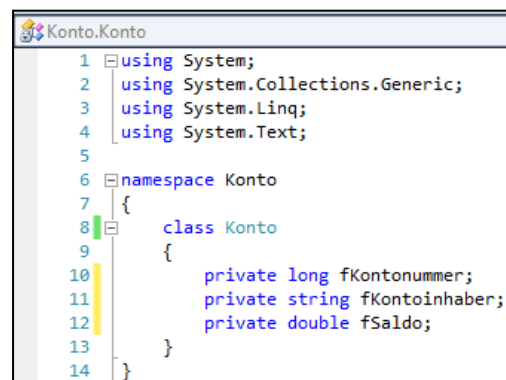
1.3.1 Erzeugen der Klasse Konto und Anlegen von Feldern

Im Projektmappen-Explorer wählen Sie die Projektdatei **Konto** mit der rechten Maustaste an und legen über die Befehlsfolge <Hinzufügen> <Klasse...> eine neue Klasse mit dem Namen **Konto.cs** an.



Nach Erstellen der Klasse **Konto** öffnet sich das zugehörige Code-Fenster. Im Body der Klasse wird die Deklaration der Attribute (in C# **Felder** genannt) gemäß der nebenstehenden Abbildung eingetragen.

Es handelt sich in diesem Fall um die Attribute Kontonummer, Kontoinhaber und Saldo, die mit den



zugehörigen Felddatentypen deklariert werden. Um zu kennzeichnen, dass es sich um Felder handelt, beginnt in diesem Skript die Deklaration mit einem kleinen **f**, gefolgt von dem Feldnamen³.

Die Vereinbarung der Variablen (Felder) beginnt in der Regel mit dem Schlüsselwort **private**. Dies bedeutet, dass eine **direkte** Veränderung der Variablen nur innerhalb der Klasse möglich ist. Dadurch wird das **Geheimnisprinzip** gewahrt, der unmittelbare Datenzugriff von außen wird nicht gestattet (= **Kapselung**), d. h. in solch einem Fall kann eine Veränderung nur durch den Aufruf öffentlich zugänglicher Methoden (Operationen) der Klasse erfolgen.

1.3.2 Properties

Damit nun jede Klasse dem Prinzip der Selbstverwaltung folgend, den Zugriff auf die Felder selbst regeln kann, wird zu jedem Feld die Klasse um eine Operation für den lesenden und/oder für den schreibenden Zugang erweitert. Die Zugriffsoperationen (Zugriffsmethoden) für ein Feld werden in C# in einem speziellen Konstrukt zusammengefasst, das als **property** (Eigenschaft) bezeichnet wird. Eine Property ist ein Codeblock mit einer Deklarationszeile gefolgt von einem geschweiften Klammerpaar. Die Deklarationszeile hat den Aufbau

Sichtbarkeitsattribut	Datentyp	Propertybezeichner
z.B. public	long	Kontonummer

Hierbei bedeutet **public** öffentlich, also von außen zugreifbar, der angegeben Datentyp wird häufig mit dem Datentyp des zugehörigen Feldes übereinstimmen und der Bezeichner der Property entspricht dem Feldnamen nur wird nun der dem Feldbezeichner vorangestellte Kleinbuchstabe „f“ weggelassen.

Das geschweifte Klammerpaar enthält die Zugriffsmethode **get** und/oder **set**. Die Zugriffsmethode „get“ regelt das Auslesen eines Feldes, die „set“-Methode das Setzen eines Feldes.

Im einfachsten Fall wird der Inhalt des Feldes bei Aufruf der get-Methode unverändert zurückgegeben und der beim Aufruf der set-Methode übergebene Wert unverändert übernommen. Dieser einfache Fall wird durch das folgende Code-Fragment erreicht:

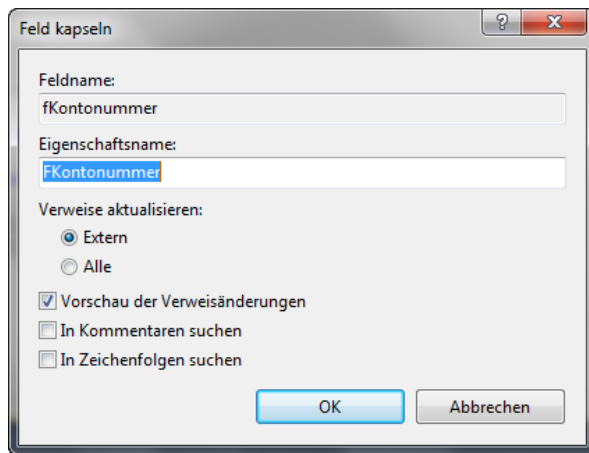
```
get{return fKontonummer;}  
set{fKontonummer=value}
```

```
Konto.Konto  
1 using System;  
2 using System.Collections.Generic;  
3 using System.Linq;  
4 using System.Text;  
5  
6 namespace Konto  
7 {  
8     class Konto  
9     {  
10         private long fKontonummer;  
11  
12         public long Kontonummer  
13         {  
14             get { return fKontonummer; }  
15             set { fKontonummer = value; }  
16         }  
17  
18         private string fKontoinhaber;  
19  
20         public string Kontoinhaber  
21         {  
22             get { return fKontoinhaber; }  
23             set { fKontoinhaber = value; }  
24         }  
25  
26         private double fSaldo;  
27  
28         public double Saldo  
29         {  
30             get { return fSaldo; }  
31             set { fSaldo = value; }  
32         }  
33     }  
34 }
```

³ In der Fachliteratur werden Felder häufig nur durch den Start mit einem Kleinbuchstaben gekennzeichnet. Da es aber auch Programmiersprachen gibt, die zwischen Groß- und Kleinbuchstaben keinen Unterschied machen, wie z.B. VB.NET, haben wir uns für den Kennbuchstaben „f“ entschieden.

Hierbei ist **value** der Parameter, in dem sich der zu setzende Wert befindet. Der Inhalt von value wird mit Hilfe des Zuweisungsoperators „=“ in das entsprechende Feld geschrieben.

Da diese Grundstruktur im Prinzip für alle Properties gleich ist, kann diese auch automatisch erzeugt werden. Hierzu geht man auf den Namen des Feldes in der Felddeklarationszeile und wählt im Kontextmenü den Befehl „Umgestalten->Feld kapseln“ aus.



In dem Popup-Menü ändert man den vorgeschlagenen Eigenschaftsnamen ab (löschen des führenden „F“) und klickt die Schaltfläche „OK“.

Nun kann diese Grundstruktur je nach Anwendungskontext abgewandelt werden. Z. B. wäre es denkbar, dass aus Datenschutzgründen nicht die gesamte Kontonummer ausgelesen werden darf sondern nur die ersten vier Stellen der Kontonummer (so verfahren z.B. einige Online-Shops, um dem Einkäufer eine Kontrolle der Bankverbindung zu ermöglichen, ohne die gesamte Kontonummer anzuzeigen). Desweiteren kann mit der Kontonummer im Anwendungskontext nicht sinnvoll gerechnet werden, daher wird die ausgelesene Kontonummer als Text zurückgegeben. Außerdem wird die Kontonummer ja nur einmalig angelegt und darf im Nachhinein nicht von außen verändert werden. In diesem Fall wird in der Property die set-Methode weggelassen. Die Property Kontonummer müsste dann wie folgt geändert werden:

```
private long fKontonummer;

public string Kontonummer
{
    get { return fKontonummer.ToString().Substring(0,3); }
}
```

Später wird noch darauf eingegangen, wie Properties von außen genutzt werden. Aber hierzu muss man erst wissen, wie denn Objekte zu einer Klasse angelegt werden können. Hierauf geht der folgende Abschnitt ein.

1.3.3 Der Konstruktor

Der Konstruktor ist eine spezielle Methode, die dazu dient, Objekte zur Laufzeit der Anwendung zu erzeugen (Instanziierung) und bei ihrer Erzeugung Werte zuzuweisen (Initialisierung). Dabei ist es wichtig zu beachten, dass es innerhalb einer Klasse mehrere Konstruktoren geben kann - je nachdem, ob man dem Benutzer der Anwendung erlauben möchte, seine Objekte auf verschiedene Art und Weise zu erzeugen.⁴

⁴ Einstieg in Visual C# 2010 – Thomas Theis S. 182

Grundsätzlich entspricht der Konstruktorname dem Klassennamen. Da Konstruktoren die oben beschriebene feste Aufgabe haben und nie einen Wert zurückgeben, entfällt die Angabe eines Rückgabetyps.

Der Konstruktor befindet sich in den Zeilen 34-39 der folgenden Abbildung. In Zeile 34 steht der Methodenkopf, in dem die Sichtbarkeit (in der Regel public), der Bezeichner (=Klassenname) und eine in runden Klammern eingeschlossene Parameterliste angegeben wird. Der Rumpf des Konstruktors wird von geschweiften Klammern eingeschlossen. In unserem Fall weisen wir den Variablen (Feldern) die entsprechenden Parameterwerte **pKontonummer**, **pKontoinhaber** und **pSaldo** (das Präfix **p** steht in diesem Fall für Parameter) zu, die dem Konstruktor beim Aufruf übergeben werden müssen. Dies entspricht prinzipiell dem Setzen von Feldern in Eigenschaften. Da aber nun mehrere Felder gleichzeitig gesetzt werden können, kann hier kein Standardparameter „value“ verwendet werden, sondern jeder benötigte Parameter hat einen eigenen Namen und einen eigenen Datentyp, der in der Parameterliste eingeführt werden muss. Wird für eine Klasse kein Konstruktor definiert, so besitzt diese immer einen leeren (parameterlosen) Standardkonstruktor. Dies hat zur Folge, dass die Felder

(= Instanzvariablen) mit Standardwerten bei Verwendung dieses Konstruktors zur Objekterzeugung initialisiert werden (Z.B. mit 0 bei numerischen Datentypen). Sobald aber ein Konstruktor hinzugefügt wird, steht der Standardkonstruktor nicht mehr automatisch zur Verfügung, sondern müsste falls gewünscht im Klassencode ergänzt werden.

Nun bleibt noch zu klären unter welchen Voraussetzungen es möglich ist zu einer Klasse mehrere Konstruktoren zu erstellen. Wie ist der Compiler in der Lage zu wissen welcher Konstruktor gemeint ist,

Im Skript haben wir zur Verdeutlichung der Buchstabe **p** den Parameternamen vorangestellt. Üblicher ist es in der Praxis Felder und zugehörige Parameter gleich zu benennen. Damit in diesem Fall das Programm Feld und Parameter unterscheiden kann, gibt es das Schlüsselwort **this**. Dieses muss bei Mehrdeutigkeit verwendet werden, wenn man sich auf eine Eigenschaft oder ein Feld beziehen will, anstelle eines aktuellen gleichnamigen Parameters.

Auch wenn in dem Kontext keine Mehrdeutigkeit vorliegt, gilt die Verwendung des Schlüsselwortes **this** bei jedem Verweis auf ein Feld, eine Eigenschaft oder eine Methode als guter Stil.

(s. Golo Roden, Auf der Fährte von C#, S. 60)

```
Konto.Konto
1 using System;
2 using System.Collections.Generic;
3 using System.Linq;
4 using System.Text;
5
6 namespace Konto
7 {
8     class Konto
9     {
10         private long fKontonummer;
11
12         public long Kontonummer
13         {
14             get { return fKontonummer; }
15             set { fKontonummer = value; }
16         }
17
18         private string fKontoinhaber;
19
20         public string Kontoinhaber
21         {
22             get { return fKontoinhaber; }
23             set { fKontoinhaber = value; }
24         }
25
26         private double fSaldo;
27
28         public double Saldo
29         {
30             get { return fSaldo; }
31             set { fSaldo = value; }
32         }
33
34         public Konto(long pKontonummer, string pKontoinhaber, double pSaldo)
35         {
36             fKontonummer = pKontonummer;
37             fKontoinhaber = pKontoinhaber;
38             fSaldo = pSaldo;
39         }
40     }
41 }
```

wenn alle Konstruktoren denselben Bezeichner (=Klassenname) besitzen? Dieses ist dann möglich, wenn beim Aufruf der Methode eindeutig entschieden werden kann, welche Methode gemeint ist. Dies ist bei identischen Methodennamen möglich, wenn die Anzahl der Parameter verschieden sind oder bei gleicher Parameteranzahl, falls die Datentypen hinreichend unterschiedlich sind. Den Bezeichner einer Methode, die Anzahl der Parameter und die Auflistung der Typen der Parameter bezeichnet man zusammen als Signatur einer Methode. Man darf also einen weiteren Konstruktor ergänzen, wenn seine Signatur eindeutig ist. Dieser Vorgang wird in der Objektorientierung als **Überladung** bezeichnet, und kann nicht nur auf Konstruktoren sondern auch auf beliebige Methoden und Operatoren angewendet werden.

Beispiel:

*Die Klasse **Konto** (s.o.) soll um weitere Konstruktoren mit nur einem Parameter ergänzt werden.*

```
(1)  Konto(double pKontostand)
    {
        fKontostand = pKontostand;
    }

(2)  Konto(string pKontoinhaber)
    {
        fKontoinhaber= pKontoinhaber;
    }

(3)  Konto(long pKontonummer)
    {
        fKontonummer= pKontonummer;
    }
```

Die gleichzeitige Ergänzung von (1) und (2) ist möglich, nicht aber die zusätzliche Ergänzung von (3). Denn bei einem Aufruf mit dem Wert 10000 kann nicht unterschieden werden, ob ein Kontostand oder eine Kontonummer gemeint sind, da der Wert 10000 implizit in einen double-Wert umgewandelt werden kann. Während ein Parameterwert „Müller“ vom Typ her eindeutig dem Konstruktor (2) zugeordnet werden kann.⁵

Exkurs

Das Schreiben überladener Konstruktoren, birgt die Gefahr, dass sich identische Codezeilen in mehreren Konstruktoren befinden (Redundanz). Dies sollte vermieden werden, da jede Änderung an solchen Zeilen dann ja an mehreren Stellen erfolgen muss.

Zu dem bereits vorhandenen Konstruktor der Klasse **Konto** soll ein weiterer Konstruktor ergänzt werden, dem nur Kontonummer und Kontoinhaber übergeben werden soll, da häufig Konten mit dem Saldo 0 EUR angelegt werden.

Um nun nicht redundanten Code schreiben zu müssen, kann man bei der Codierung von zusätzlichen Konstruktoren bereits bestehende Konstruktoren verwenden.

⁵ Das Thema Typumwandlung (Casten, Konvertierung) wird in einem eigenen Abschnitt erläutert.

Also anstelle von:

```
public Konto(int pKontonummer, string pKontoinhaber)
{
    fKontonummer = pKontonummer;
    fKontoinhaber = pKontoinhaber;
    fSaldo=0
}
```

kann man redundanzfrei schreiben:

```
public KONTOKonto(int pKontonummer,
string pKontoinhaber):this(pKontonummer, pKontoinhaber, 0)
{
}
```

Zusätzlich können hier noch eigene Anweisungen ergänzt werden, die erst ausgeführt werden, wenn der hinter dem Doppelpunkt erfolgte Konstruktoraufruf abgearbeitet ist.

1.3.4 Methoden

Soll ein Objekt weitere Operationen ausführen, die sich beispielsweise auf mehrere Felder gleichzeitig beziehen (z. B. Auszahlen, Ausgeben von Kontoinformationen), dann sind dafür Eigenschaften ungeeignet. In solchen Fällen benötigt man eine Methode, die innerhalb einer Klasse definiert wird.

Eine Methode ist ein Codeblock, der eine Reihe von Anweisungen enthält. Methoden werden innerhalb einer Klasse deklariert, indem die Sichtbarkeit (z.B. private oder public), der Rückgabewert, der Name der Methode gefolgt von runden Klammern, die eine durch Kommata getrennte Auflistung der Parameter mit den zugehörigen Datentypen enthält. Die **formalen Parameter** der Parameterliste sind Variablen mit begrenzter Gültigkeit in zeitlicher und örtlicher Hinsicht. Mit dem Methodenaufruf werden den formalen Parametern entsprechende Werte übergeben, die während der Laufzeit der Methode Gültigkeit haben. Nach Abarbeitung der Methode stehen diese Variablen nicht mehr zur Verfügung.

Leere Klammern geben an, dass die Methode keine Parameter benötigt. Der Methodendeklaration im Methodenkopf folgt ein geschweiftes Klammerpaar, das die auszuführenden Anweisungen enthält. Methoden lassen sich klassifizieren in solche, die

- ⇒ Informationen über den Zustand eines Objekts zurückgeben (**Anfragen**),
- ⇒ Veränderungen am Zustand eines Objekts vornehmen (**Aufträge**).

Es gibt aber auch Mischformen. Z.B: eine Methode, die den Auftrag „Auszahlen“ durchführt, aber auch die Information zurückgibt, ob der Auftrag vollständig ausgeführt werden konnte.

Gibt eine Methode einen Wert zurück, so wird der Datentyp des Rückgabewertes im Methodenkopf angegeben. In diesem Fall muss jeder mögliche Codepfad mit einer **return**-Anweisung enden, die einen Wert des angekündigten Datentyps zurückgibt.

Führt die Methode nur einen Auftrag aus, ohne einen Wert zurückzugeben, so wird anstelle eines Datentyps das Schlüsselwort **void** angegeben. Solche Methoden enthalten dann auch keine return-Anweisung.

Beispiel (ohne Rückgabotyp)

```
public void Einzahlen(double pBetrag)
{
    this.Saldo+=pBetrag;
}
```

Beispiel (mit Rückgabotyp)

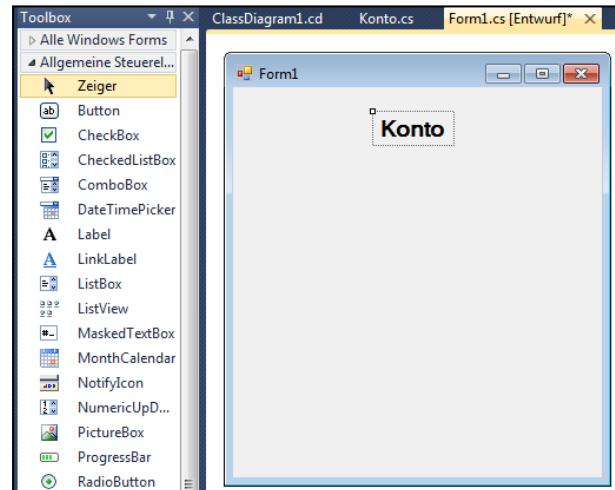
```
public bool Auszahlen(double pBetrag)
{
    if (fSaldo - pBetrag > -fDispo)
    {
        this.Saldo -= pBetrag;
        return true;
    }
    else
    {
        return false;
    }
}
```

1.4 Erstellen einer Grafischen Benutzeroberfläche (GUI)

Um den Sachverhalt anhand des Kontobeispiels zu verdeutlichen und die Daten eingeben bzw. auslesen zu können, erstellen Sie nun eine **grafische Benutzeroberfläche** (GUI = Graphical User Interface). Dazu wählen Sie im Projektmappenexplorer **Form1.cs** an.

1.4.1 Erstellen von Labels

Ziehen Sie aus der Toolbox am linken Bildrand ein **Label** in das Formular und weisen Sie ihm in den Eigenschaften am rechten unteren Bildrand den Namen **lblKonto** zu (Namenskonvention: Alle Bezeichner für Labels beginnen mit **lbl**⁶). Das Label dient in diesem Fall als Beschriftungsfeld und soll den Text **Konto** enthalten, den Sie ebenfalls im Eigenschaften-Fenster eingeben. Stellen Sie zudem unter **Font** die Schriftgröße 14 und Fettdruck (Bold=True) ein und positionieren Sie das Label mittig im oberen Bereich des Formulars.

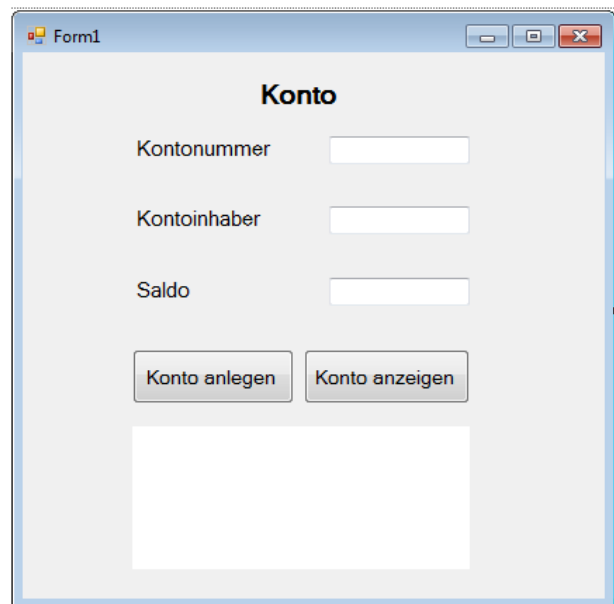


Erstellen Sie analog drei weitere Labels mit den Namen **lblKontonummer**, **lblKontoinhaber** und **lblSaldo**, die den Text Kontonummer, Kontoinhaber und Saldo enthalten und in Schriftgröße 11 formatiert sind. Ordnen Sie diese Labels halb links unter dem Label Konto an. Legen Sie noch ein weiteres Label an, das Sie **lblAusgabe** nennen und am unteren Bildrand anordnen. Dieses Label soll keine Beschriftung erhalten, da es später der Ausgabe von Daten dienen soll. Um es für den Nutzer auch schon vor der Programmausführung sichtbar zu machen, bietet es sich an dem Label die Eigenschaften **BackColor <ButtonHighlight>** und **AutoSize <False>** zuzuweisen.

⁶ Eine vollständige Liste der verwendeten Namenskonventionen für Steuerelemente finden Sie in Anhang I.

1.4.2 Erstellen von Textboxen und Buttons

Im nächsten Schritt erstellen Sie Textboxen, die die Eingabe von Werten ermöglichen sollen. Ziehen Sie dazu aus der Toolbox zunächst eine **Textbox** in den Formularbereich und weisen Sie ihr den Namen **tbxKontonummer** zu (Namenskonvention: Bezeichner beginnt mit tbx). Verfahren Sie in derselben Weise mit Textboxen für den Kontoinhaber und den Saldo. Der Formulargestalter von Visual Studio gibt Ihnen hier Hilfen zur Anordnung der Steuerelemente innerhalb des Formulars. Ordnen Sie die Textboxen rechts neben den zugehörigen Labels an.



Die Eingabe von Daten ist durch das Einrichten der Textboxen vorbereitet. Um sie aber tatsächlich ein- und auslesen zu können, benötigt man Buttons. Ziehen Sie zwei Buttons in das Formularfeld, die Sie mit **btnAnlegen** und **btnAnzeigen** benennen und mit **Konto anlegen** bzw. **Konto anzeigen** beschriften (Namenskonvention: btn wird den Bezeichnern von Buttons vorangestellt).

Zur Vorbereitung des Ein- und Auslesens der Daten ist nun ein Zwischenschritt notwendig. Es muss eine neue Methode für die Ausgabe des Textes erstellt werden. Wechseln Sie dazu in die **Konto.cs** und legen Sie in der Klasse **Konto** eine neue Methode mit dem Namen **Ausgabe** und dem Felddatentyp **String** an. Mit Hilfe dieser Methode soll der Text ausgegeben werden, der in die zuvor erstellten Textboxen eingegeben wurde.

Um die Ausgabe später nicht einzellig darzustellen, sondern bündig untereinander, behilft man sich einer so genannten Escape-Sequenz **\n** (=new line), um nach der Ausgabe jedes Feldes in eine neue Zeile zu springen (siehe Code-Abbildung auf der Folgeseite, Zeile 44).

Gehen Sie nun zurück in das Formular „Form1.cs [Entwurf]“ und benennen Sie es im Projektmappen-Explorer zur besseren Übersichtlichkeit in **frmKonto.cs** um. Mit einem Doppelklick auf einen Bereich des Formulars, der nicht durch ein Steuerelement (z.B. Label, Textbox, Button) belegt ist, gelangen Sie automatisch in das **FormLoad**-Ereignis. Hier können Anweisungen eingegeben werden, die ausgeführt werden, wenn das Formular aufgerufen wird. So können an dieser Stelle z.B. Objekten konkrete Werte zugewiesen werden, die beim Starten des Programms angezeigt werden. Dies ist in diesem frühen Stadium der Programmierung die einzige Möglichkeit Daten dauerhaft im Programm zu halten, da eine dauerhafte Speicherung (z.B. in einer Datenbank) noch nicht realisiert wurde. Ein Beispiel für das Anlegen eines Objektes im **frmKonto_Load**-Ereignis könnte wie folgt aussehen:

```
21 private void frmKonto_Load(object sender, EventArgs e)
22 {
23     Konto KontoMueller;
24     KontoMueller = new Konto(5005878, "Müller, Hans", 2850.20);
25     lblAusgabe.Text = KontoMueller.Ausgabe();
26 }
```

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5
6  namespace Konto
7  {
8      class Konto
9      {
10         private long fKontonummer;
11
12         public long Kontonummer
13         {
14             get { return fKontonummer; }
15             set { fKontonummer = value; }
16         }
17
18         private string fKontoinhaber;
19
20         public string Kontoinhaber
21         {
22             get { return fKontoinhaber; }
23             set { fKontoinhaber = value; }
24         }
25
26         private double fSaldo;
27
28         public double Saldo
29         {
30             get { return fSaldo; }
31             set { fSaldo = value; }
32         }
33
34         public Konto(long pKontonummer, string pKontoinhaber, double pSaldo)
35         {
36             fKontonummer = pKontonummer;
37             fKontoinhaber = pKontoinhaber;
38             fSaldo = pSaldo;
39         }
40
41         public string Ausgabe()
42         {
43             string ausgabeText;
44             string neueZeile = "\n";
45             ausgabeText = "Kontonummer: " + Kontonummer.ToString() + neueZeile;
46             ausgabeText = ausgabeText + "Kontoinhaber: " + Kontoinhaber + neueZeile;
47             ausgabeText = ausgabeText + "Saldo: " + Saldo.ToString("C");
48             return ausgabeText;
49         }
50     }
51 }

```

Instanzvariable (Feld) (points to line 10)

Properties (bracketed lines 12-32)

Konstruktor (bracketed lines 34-39)

Ausgabe-methode (bracketed lines 41-49)

1-3: Code der Klasse Konto

Im Quelltext des Formulars wird das Verhalten der einzelnen Steuerelemente beschrieben, d.h. in unserem Fall soll beschrieben werden was passiert, wenn ein Button angeklickt wird. Dazu müssen wir dem Programm zunächst sagen, dass wir ein neues Objekt innerhalb der Klasse Konto anlegen wollen.

Dies geschieht durch das Einfügen der Anweisung

```
private Konto MeinKonto;
```

unterhalb der partiellen ⁷ Klassendefinition (`public partial class frmKonto : Form`) im Quelltext des Formulars (Abbildung unten: Zeile 14).

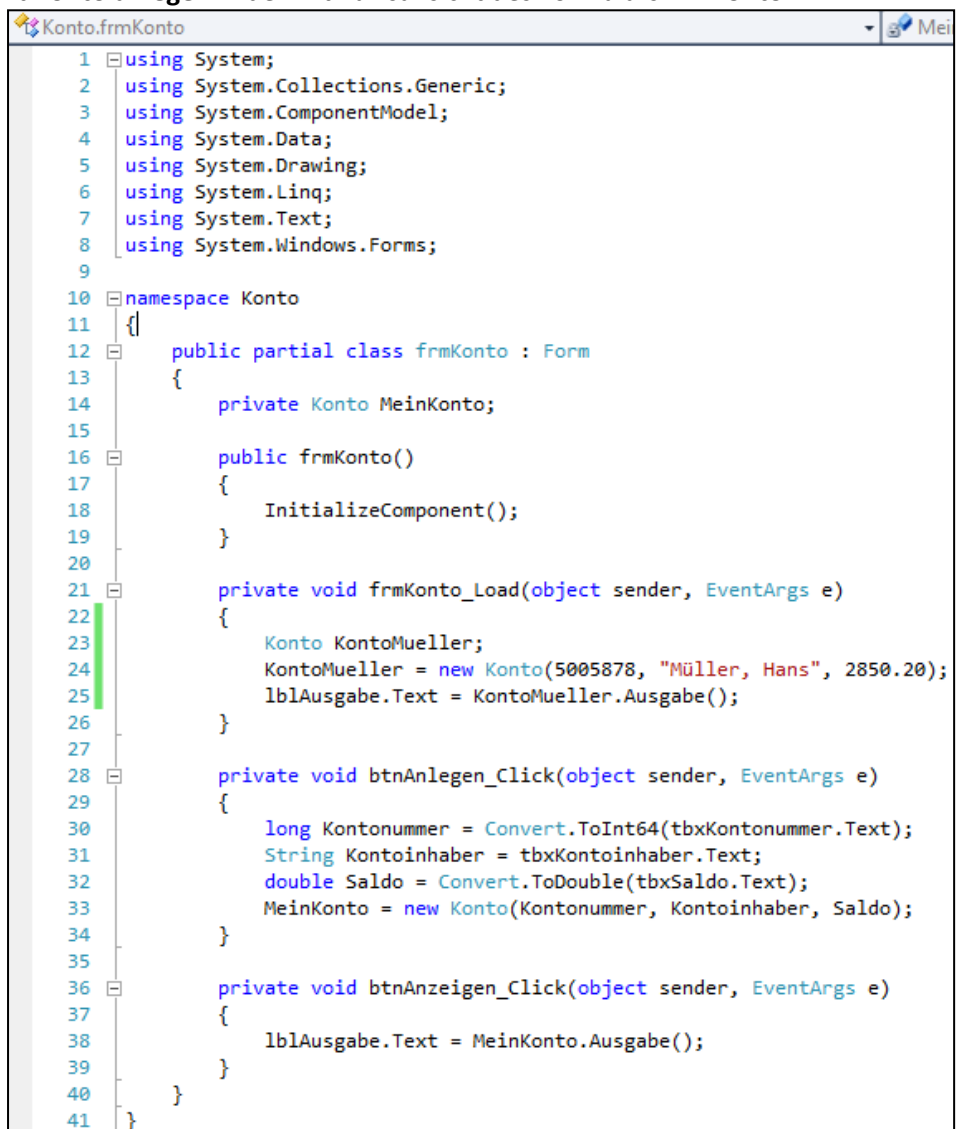
- ! Beachten Sie, dass der Gültigkeitsbereich der Variable so gewählt ist, dass von sämtlichen Formalmethoden darauf zugegriffen werden kann.
- ! Beachten Sie weiter, dass hier noch kein konkretes Objekt instanziiert ist, sondern nur die Deklaration erfolgt. D.h. es liegen im Gegensatz zum Objekt **einKonto** noch keine Daten für das Objekt **MeinKonto** vor, das Objekt hat den Standardwert **null**. „null“ ist ein spezielles Schlüsselwort in C# und bedeutet, dass hier noch kein Adressverweis gespeichert ist.

1.5 Die Click-Methode

Bei Klick auf die Buttons sollen Daten eingegeben bzw. ausgelesen werden können. Per Doppelklick auf den Button mit der Aufschrift **Konto anlegen** in der Entwurfsansicht des Formulars **frmKonto**

wird die zugehörige Methode generiert, allerdings ist der Methodenrumpf zunächst leer. In der Ereignisroutine des Buttons **btnAnlegen** (Zeile 28 – 34) werden die Werte der Textboxen ausgelesen in Hilfsvariablen mit geeigneten Datentypen übertragen und dann zur Objekterzeugung verwendet.

Die Methoden der Konvert-klassse sorgen für die geeignete Konvertierung der Datentypen, so dass sie zu den Felddatentypen des zu erzeugenden Konto-Objekts passen. In Zeile 33 wird mit den eingelesenen Werten das Objekt **MeinKonto** initialisiert und erzeugt (Details im Folgeabschnitt). Da eine Textbox grundsätzlich nur Text



```
1 using System;
2 using System.Collections.Generic;
3 using System.ComponentModel;
4 using System.Data;
5 using System.Drawing;
6 using System.Linq;
7 using System.Text;
8 using System.Windows.Forms;
9
10 namespace Konto
11 {
12     public partial class frmKonto : Form
13     {
14         private Konto MeinKonto;
15
16         public frmKonto()
17         {
18             InitializeComponent();
19         }
20
21         private void frmKonto_Load(object sender, EventArgs e)
22         {
23             Konto KontoMueller;
24             KontoMueller = new Konto(5005878, "Müller, Hans", 2850.20);
25             lblAusgabe.Text = KontoMueller.Ausgabe();
26         }
27
28         private void btnAnlegen_Click(object sender, EventArgs e)
29         {
30             long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
31             String Kontoinhaber = tbxKontoinhaber.Text;
32             double Saldo = Convert.ToDouble(tbxSaldo.Text);
33             MeinKonto = new Konto(Kontonummer, Kontoinhaber, Saldo);
34         }
35
36         private void btnAnzeigen_Click(object sender, EventArgs e)
37         {
38             lblAusgabe.Text = MeinKonto.Ausgabe();
39         }
40     }
41 }
```

⁷ Partielle Klassen sind eine Möglichkeit, Klassendefinitionen auf mehrere Dateien zu verteilen. C# nutzt dies u.a. um den vom Designer erzeugten Code einer Formalklasse von dem vom Benutzer erstellten Code zu trennen.

erkennt und auch ausgeben würde, müssen die eingegebenen Werte bei Saldo und Kontonummer in ihre entsprechenden Felddatentypen konvertiert werden (Zeilen 30 und 32). Bei Klick auf den Button **Konto anzeigen (btnAnzeigen)** sollen die eingegeben Daten angezeigt werden. Man gelangt wiederum durch einen Doppelklick auf den Button **Konto anzeigen** im Entwurf des Formulars **frmKonto** an die entsprechende Stelle im Quelltext und verknüpft hier das Label **lblAusgabe** mit dem neu angelegten Objekt **MeinKonto** (Zeilen 36-39).

Ein einfaches Programm zur Eingabe von Kontoinformationen ist nun funktionsfähig. Sie können es testen, indem Sie das Debugging mit Klicken auf die grüne „Play“-Taste oder Drücken der Taste F5 starten.

Auftrag

Geben Sie die Kontoinformationen des Kunden Gregor Jansen in das Programm ein und lassen Sie sich diese Daten anschließend wieder anzeigen.

! Da das Programm noch über keine Datenbankfunktionalität oder ähnliche externe Speicherung verfügt, können die neu eingegebenen Informationen leider nicht dauerhaft gespeichert werden.

1.6 Programmanalyse

Zu Beginn des Programmcodes (Seite 14) werden mittels der **using-Anweisung** von Visual Studio unterschiedliche vordefinierte Namensräume zugewiesen, da einige Namensräume, wie bspw. **System**, von mehreren Komponenten genutzt werden. Darauf folgt die benutzerdefinierte Einrichtung des Namensraums (namespace) **Konto** durch den Nutzer, der auch gleichzeitig den Dateinamen bestimmt.

Es folgt die Vereinbarung der Klasse **Konto**. Der Klassenkörper wird durch ein geschweiftes Klammernpaar { ... } eingerahmt.

Innerhalb des Klassenkörpers können die Bereiche

- ⇒ Instanzvariablen (Zeilen 10, 18, 26),
- ⇒ Konstruktor (Zeile 34 – 39),
- ⇒ Properties (Zeilen 12-16, 20-24 und 28-32),
- ⇒ Allgemeine Methode (Zeile 41 – 49)

unterschieden werden.

1.7 Nutzen des Konstruktors zur Objekterzeugung

Es ist für das Verständnis des Programmcodes wichtig zu wissen, dass der Aufruf des Konstruktors zwei Schritte veranlasst:

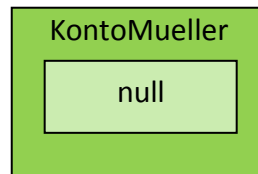
- ⇒ die **Objektinstanziierung** (Adressverweis auf den Datenbereichs des Objekts)
- ⇒ die **Objektinitialisierung** (Füllen der Daten eines Objektes an Hand der übergebenen Parameter sowie der Anweisungen im Konstruktor)

Beispiel:

In der folgenden Code-Zeile wird eine Variable vom Typ **Konto** deklariert:

```
Konto KontoMueller;
```

Durch die **Objektdeklaration** wird eine Variable namens **KontoMueller** vom Typ **Konto** im Hauptspeicher angelegt und standardmäßig mit dem Wert **null** initialisiert.



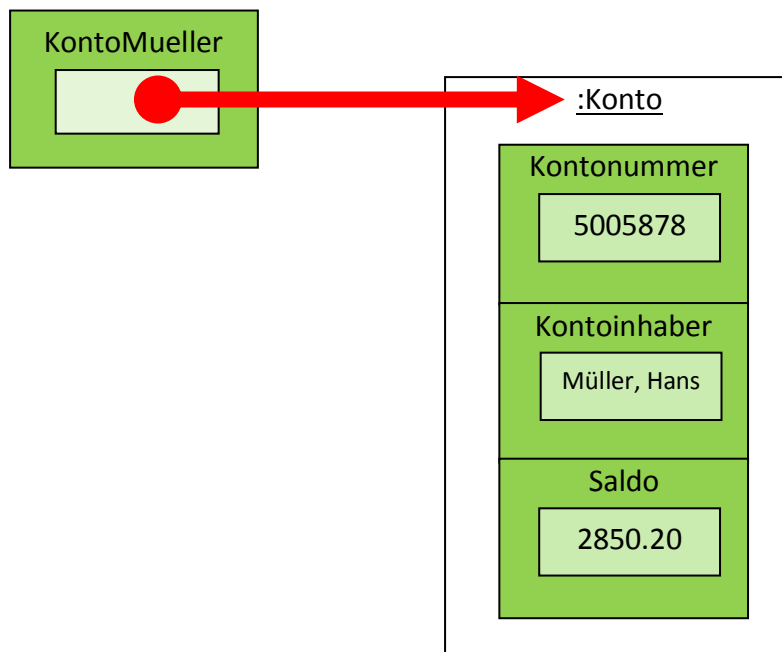
In C# sieht die Objektdeklaration wie folgt aus:

```
private Konto KontoMueller;
```

Danach erfolgt die Objektinstanziierung durch den Aufruf des Konstruktors, bei der die Adresse des zugehörigen Datenbereichs zugeordnet wird und die Objektdaten gemäß den Anweisungen im Konstruktor initialisiert werden.

```
KontoMueller = new Konto(5005878, „Müller, Hans“, 2850.20);
```

Dies führt zu dem folgenden Ergebnis im Hauptspeicher führt.



Nach der Instanziierung ist ein Speicherbereich für das Objekt und dessen Variablen bereitgestellt worden. Diese Speicheradresse wird in der Variablen **KontoMueller** hinterlegt.



Deklaration und Instanziierung eines Objektes kann auch als Einzeiler geschrieben werden. Allerdings kann man dies nicht immer nutzen, da häufig die Deklaration außerhalb einer speziellen Methode geschrieben werden muss, damit sie z. B. in der gesamten Formalklasse zur Verfügung steht.

```
Private Konto KontoMueller = new KONT0(5005878, „Müller, Hans“, 2850.20);
```

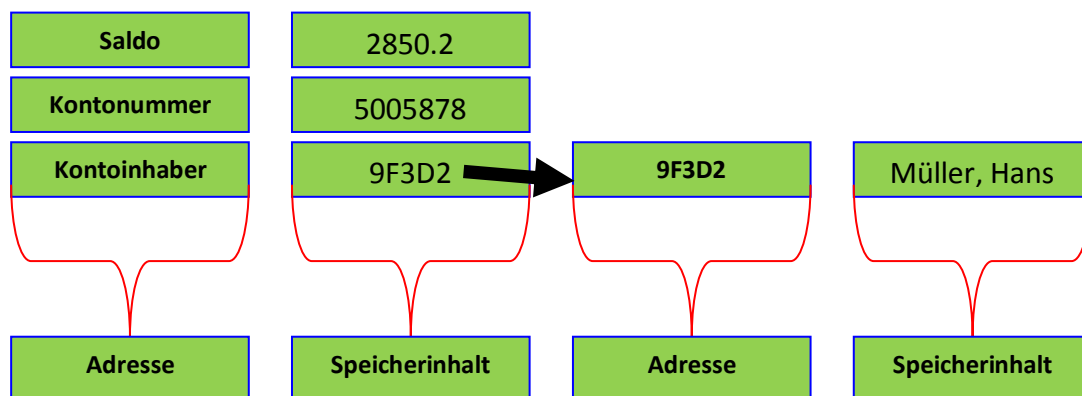
1.8 Variablendeklaration

Die Variablendeklarationen sind bisher wie folgt aufgebaut:

private Datentyp bezeichnerVariable;

Einfache Datentypen (long, double) beginnen mit einem Kleinbuchstaben; **Referenztypen** (String) können mit einem Großbuchstaben beginnen – es werden in C# allerdings auch Kleinbuchstaben akzeptiert.

- ⇒ Bei einfachen Datentypen enthält der durch die Speichervariable angesprochene Hauptspeicherplatz den das Objekt beschreibenden Attributwert (z. B. Saldo: 2850.20);
- ⇒ bei Referenztypen befindet sich in dem Speicherbereich ein Adressverweis auf den Speicherbereich, in dem sich die Attributwerte befinden.



1.9 Nutzung von Properties (Set- und Get-Methoden)

Properties bestehen aus einer Deklarationszeile (s.u.) sowie einem Block, in der die zugehörige get- und set-Methode festgelegt wird.

Deklarationszeile: `public long Kontonummer (public Felddatentyp Bezeichner).`

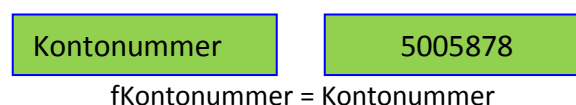
Da der Sichtbarkeitsbereich public ist, sind sie von jedem Ort aus aufrufbar.

Für die Nutzung von Properties von außerhalb nutzt C# den Zuweisungsoperator „=“.

Die Anweisung:

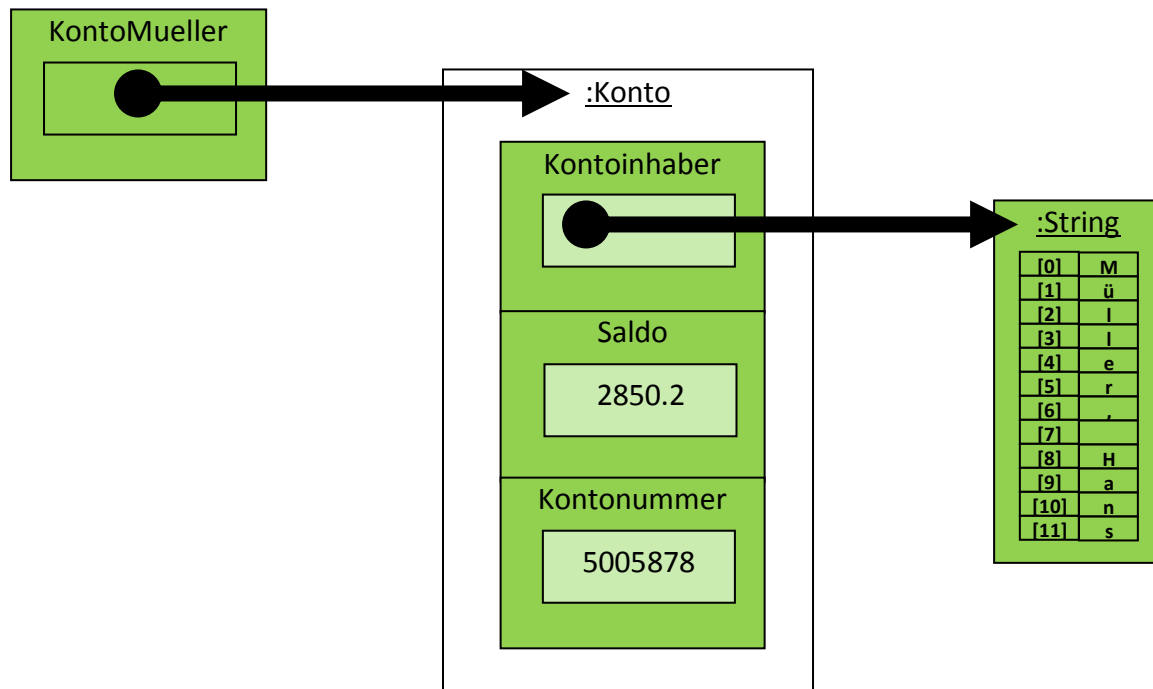
```
KontoMueller.Kontonummer=5005878
```

bedeutet, dass an das Objekt **KontoMueller** der Auftrag **set {fKontonummer = 5005878}** gesendet wird. Der aktuelle Parameter (5005878) wird entsprechend der Grafik an den formalen Parameter **value** übergeben. Anschließend erfolgt innerhalb der Methode die Wertzuweisung an die Instanzvariable:



In C# ist der Aufruf der set-Methode implizit, während z.B. in Java die set-Methode explizit genannt wird. C# erkennt den Aufruf des set-Auftrags daran, dass der Propertyname links vom Zuweisungsoperator „=“ steht.

Nach Abarbeitung der beiden anderen Set-Aufträge ergibt sich die folgende Situation im Hauptspeicher:



Wird die Property genutzt, um Informationen auszulesen (get-Auftrag), benötigt man einen Container (Variable) gleichen Datentyps der die angefragte Information aufnehmen kann.

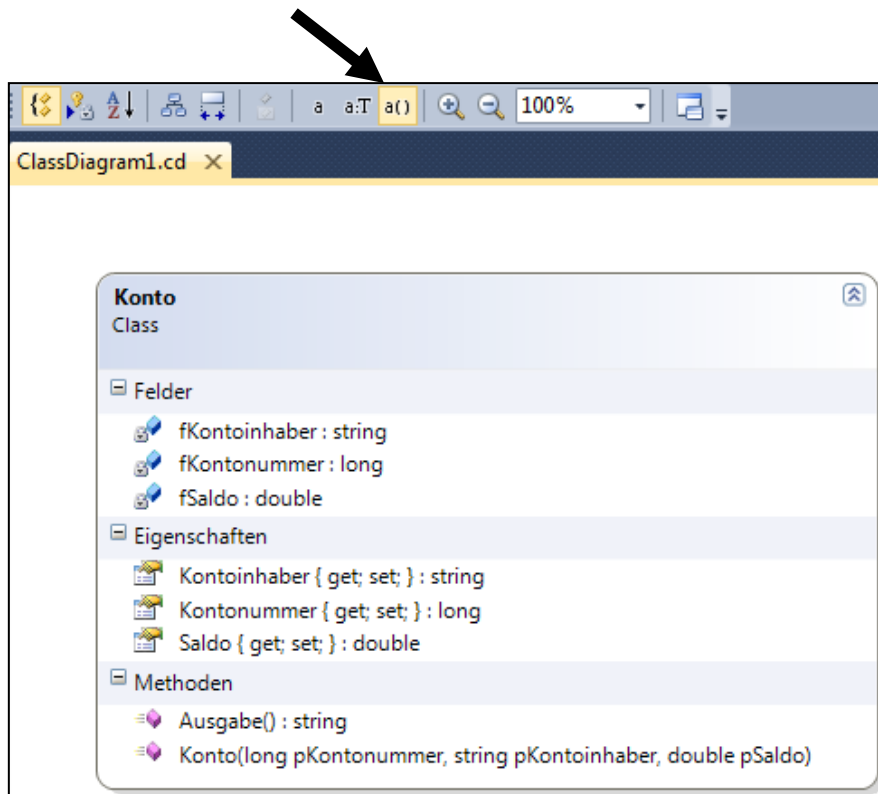
Auch hier wird wieder der Zuweisungsoperator „=" verwendet, während z. B. bei der Programmiersprache Java die get-Methode explizit angegeben werden muss.

```
int Kontonummer = KontoMueller.Kontonummer
```

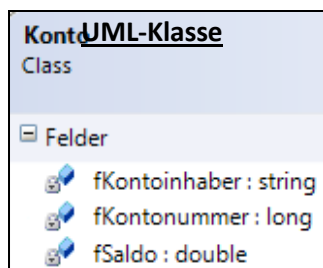
Daran, dass die Property nun rechts vom Zuweisungsoperator „=" steht, erkennt der Compiler, dass nun die parameterlose get-Methode benötigt wird. Die Rückgabe erfolgt gemäß der im get-Block enthaltenen Anweisungen. Ist der Datentyp der aufnehmenden Variable verschieden von dem Datentyp der Property wird zwangsläufig ein Fehler provoziert.

1.10 Klassendiagramm und C#

Im Rahmen der Softwareentwicklung steht vor der **Objektorientierten Programmierung (OOP)** die objektorientierte Modellierung. Grundlegend für die statische Modellierung ist das Klassendiagramm, das in seinen Grundzügen schon zu Beginn dieses Kapitels erstellt wurde. Visual Studio 2010 (leider nicht die Express-Version) bietet uns an dieser Stelle die Möglichkeit ein automatisch generiertes Klassendiagramm auf Basis unserer Programmierung anzuzeigen. Wählen Sie dazu mit der rechten Maustaste im Projektmappenexplorer **Konto.cs** an und klicken Sie auf **Klassendiagramm anzeigen**. Ihnen wird nun das Klassendiagramm der Klasse **Konto** angezeigt. Zum Anzeigen der vollständigen Signatur klicken Sie links oberhalb des Klassendiagramms auf die Schaltfläche **a()**.



Klassendiagramm (UML) und Implementierung (C#-Code) sind ähnlich aufgebaut, unterscheiden sich jedoch in Details.



C#-Code

```
class Konto
{
    private long fKontonummer;
    private string fKontoinhaber;
    private double fSaldo;
}
```

Die **Zugriffsmodifizierer** werden in der UML symbolisch dargestellt. In der UML steht der **Datentyp** hinter der Variablen, bei der Codierung umgekehrt. Außerdem steht in der UML noch ein Doppelpunkt vor dem Datentyp.

Ähnlich wie bei den Variablen sind auch bei den **Methoden** die Positionierungen der einzelnen Elemente zwischen der UML und der Codierung unterschiedlich.

MERKE

Die Modellierung (UML) geht der Implementierung (C#) voraus. Die Darstellung und die verwendete Terminologie weichen voneinander ab.

So spricht man in der UML von Attributen und Operationen und bei der Programmierung von Datenfeldern bzw. Instanzvariablen und Methoden.

Übungen zu Kapitel 1

Aufgabe 1

Zeichnen Sie das Klassendiagramm für das **Konto** und ergänzen Sie es um die Instanzvariable sowie die Property (Get- und Set-Methode) für das **Kreditlimit**.

☐	S EUR-KONTO 5005878 Sparkasse Kleve BLZ 324 500 00	KONTOAUSZUG Kontoinhaber Müller, Hans Saldo 2.850,20 € + Kreditlimit 6.000,00 €
☐		

☐	S EUR-KONTO 5005612 Sparkasse Kleve BLZ 324 500 00	KONTOAUSZUG Kontoinhaber Schmidt, Lolita Saldo 2.300,00 € - Kreditlimit 5.000,00 €
☐		

☐	S EUR-KONTO 7205820 Sparkasse Kleve BLZ 324 500 00	KONTOAUSZUG Kontoinhaber Jansen, Gregor Saldo 2.850,20 € + Kreditlimit - €
☐		

Anschließend vervollständigen Sie die Codierung und testen das Programm!

Aufgabe 2

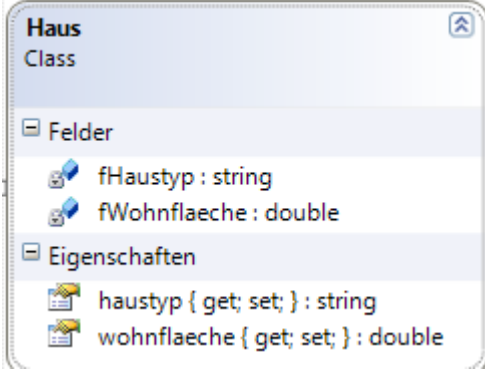
Überprüfen Sie, ob Sie die Bedeutung bzw. den Inhalt der folgenden Begriffe erläutern können und legen Sie anschließend ein Glossar mit diesen Begriffen an:

- ⇒ Projekt, Klasse und Objekt,
- ⇒ int, double und String,
- ⇒ primitive Datentypen und Referenztypen,
- ⇒ Instanzvariable und Attribute,
- ⇒ Deklarieren, Instanziiieren und Initialisieren,
- ⇒ Konstruktor,
- ⇒ Bezeichner und Datenfelder,
- ⇒ Methoden.

Aufgabe 3

Ordnen Sie die folgenden Begriffe den untenstehenden Aussagen, Darstellungen oder Beschreibungen zu.

1. **Objekt**
2. **Attribut**
3. **Klasse**
4. **Methode**
5. **Bezeichner**

a)	Merkmale wie Telefon, Familienname, Kontostand.	
b)	Diese Operationen dienen dazu, Attributswerte in einem Objekt zu setzen, zu ändern oder zu lesen.	
c)	Darstellung individueller Betrachtungsgegenstände im Hauptspeicher eines Computers auf der Basis einer Klasse.	
d)	hausMeier : Haus	
e)	Namen, die der Programmierer für Methoden oder Variablen wählen muss.	
f)	Abstrakte Zusammenfassung von individuellen Objekten mit identischen Attributen und Operationen, die an diesen Objekten ausgeführt werden können.	
g)	 <pre> classDiagram class Haus { fHaustyp : string fWohnflaeche : double haustyp { get; set; } : string wohnflaeche { get; set; } : double } </pre>	

Aufgabe 4

Gegeben sei die folgende Codierung einer C#-Klasse namens **Haus**.

Geben Sie durch Einsetzen der folgenden Zahlen an, wo sich die folgenden Elemente in der Codierung wiederfinden.

1. Instanzvariable
2. Parameter Variable
3. Konstruktor
4. Klassenbezeichner
5. Referenzdatentyp
6. primitiver Datentyp
7. Signatur
8. Kommentar

```
namespace Haus {  
    class Haus  
    {  
        private string fHaustyp;  
        private double fWohnflaeche;  
  
        public Haus(string haustyp, double wohnflaeche);  
  
        public string haustyp  
        {  
            get { return fHaustyp; }  
            set { fHaustyp = value; }  
        }  
  
        public double wohnflaeche  
        {  
            get { return fWohnflaeche; }  
            set { fWohnflaeche = value; }  
        }  
    }  
}  
// Nicht vorhanden
```

2. Erweiterung von Klassendefinitionen



Situationsbeschreibung

Nach Vorstellung des bisherigen Programms äußern die Sachbearbeiter aus den Fachabteilungen den Wunsch, dass bei der Anlage eines Kontos nach wie vor sofort eine Kontonummer vergeben sowie der Kontoinhaber erfasst wird und Saldo und Kreditlimit zugewiesen werden können.

Aber sie erklären, dass sich die Kontonummer während der Existenz des Kontos nicht mehr verändert. Der Name des Kontoinhabers kann sich jedoch durch Änderung des Ehestands sowie Tod und Erbschaft verändern.

Auch weisen sie darauf hin, dass der Kontostand nur durch Einzahlungen und Auszahlungen geändert werden darf.

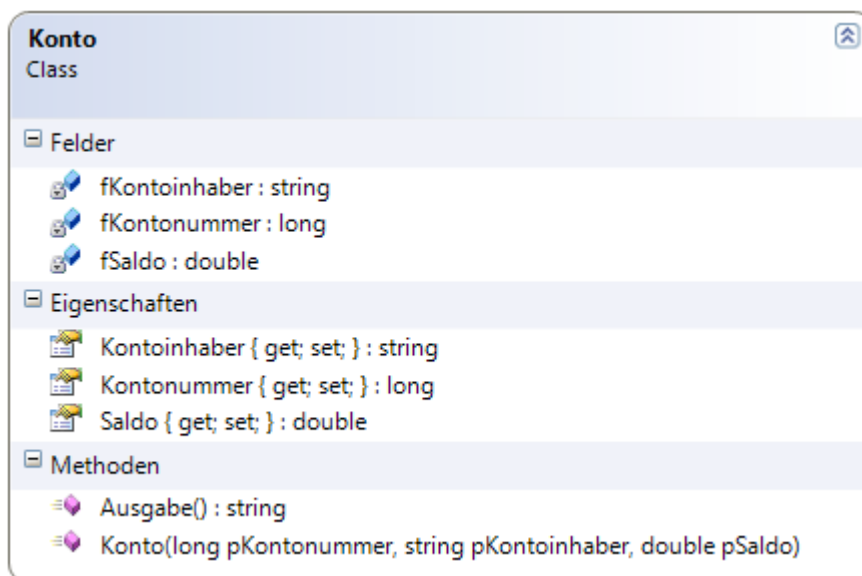
Die EDV-Verantwortlichen versprechen eine Lösung herbeizuführen.

2.1 Klassendiagramm überarbeiten

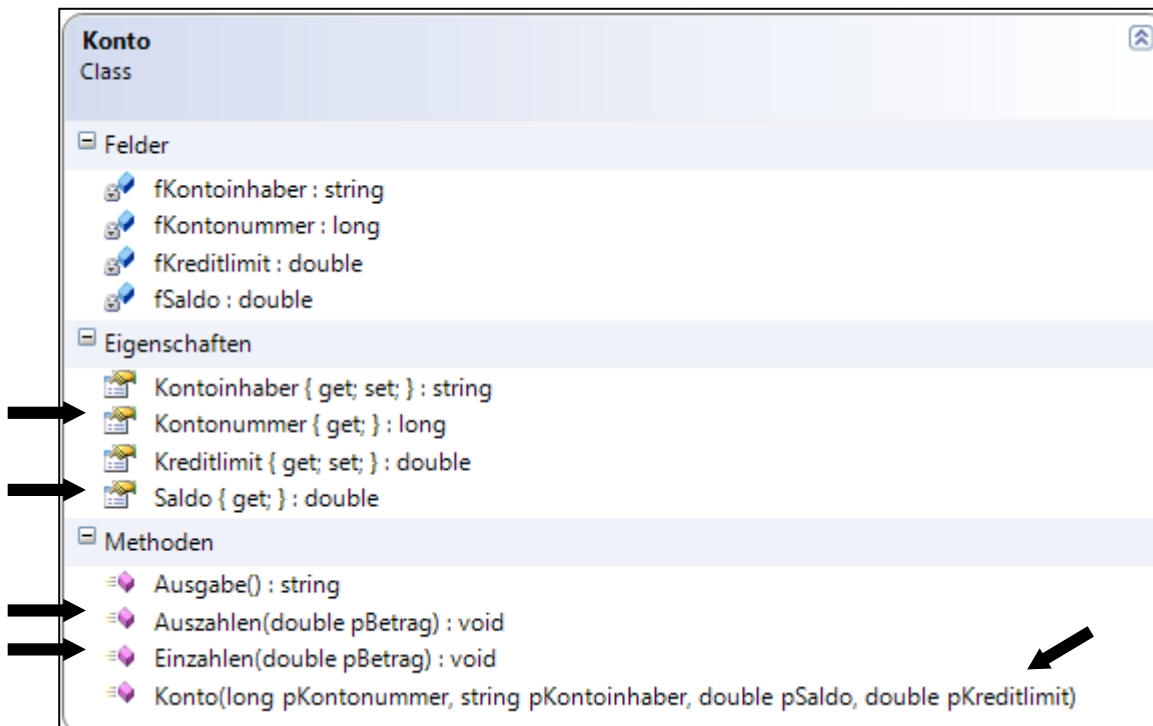
Die Softwareentwickler überarbeiten das bisherige Klassendiagramm entsprechend den an sie herangetragenen Wünschen wie folgt:

- ⇒ der Konstruktor wird um die formalen Parameter ergänzt;
- ⇒ die Set-Methoden der Properties Kontonummer und Saldo werden gelöscht;
- ⇒ der Saldo kann nur noch durch die Methoden „Einzahlen“ und „Auszahlen“ geändert werden.

Ausgangssituation: Klassendiagramm entsprechend Kapitel 1.



Zielsituation: Klassendiagramm nach Überarbeitung.



2.2 Erläuterungen zum Konstruktor

Die Anfangswerte wurden bisher im Konstruktor mit Parametern direkt den Feldern zugewiesen. Dies ist besonders dann notwendig, wenn die Set-Methoden der entsprechenden Felder nicht vorhanden sind. Ist die Set-Methode vorhanden, können den Feldern über Verwendung der Properties Werte zugewiesen werden. Der angepasste Konstruktor sieht nun folgendermaßen aus:

```
public Konto(long pKontonummer, string pKontoinhaber, double pSaldo, double pKreditlimit)
{
    fKontonummer = pKontonummer;
    Kontoinhaber = pKontoinhaber;
    fSaldo = pSaldo;
    Kreditlimit = pKreditlimit;
}
```

Will man eine logische Überprüfung der zu initialisierenden Werte vornehmen, so kann dies direkt im Konstruktor erfolgen oder mit Hilfe einer privaten Methode, die im Konstruktor aufgerufen wird.

Beispielsweise enthalten die Kontonummern der Kreditinstitute an der letzten Stelle eine Prüfziffer, die mittels eines komplexen Algorithmus innerhalb der privaten Methode ermittelt und überprüft werden müsste.

2.3 Methoden schreiben und verwenden

Unter einer Methode versteht man eine Menge zusammengehörender Anweisungen, die es erlauben eine bestimmte Aufgabe auszuführen. Methoden können keinen, einen oder mehrere Aufrufparameter haben und können einen oder keinen Funktionswert zurückliefern. Die Information hierzu findet sich direkt in der Definitionszeile zu einer Methode (Methodenkopf).

Der Aufbau des Methodenkopfs wird am folgenden Beispiel erläutert:

1

2

3

4

```
public void Einzahlen(double pBetrag)
{
    fSaldo = fSaldo + Math.Max(pBetrag, 0);
}
```

1	Sichtbarkeitsattribut	In der Regel sind Methoden public.
2	Art des Rückgabewertes	Der Rückgabewert legt fest, welchen Datentyp das Ergebnis eines Methodenaufrufs besitzt. void wird angegeben, wenn der Aufruf kein Ergebnis zurückgibt.
3	Funktionsname	Funktionsnamen sind frei wählbar, müssen aber wie bei Variablen gültige C#-Bezeichner sein. Das heißt, sie sollten mit einem Buchstaben beginnen. Es dürfen Ziffern und Buchstaben folgen. Einzige erlaubte Sonderzeichen sind „_“ und „@“. Sie dürfen keinem in C# reservierten Schlüsselwort entsprechen.
4	Parameterliste	Die Parameterliste enthält Datentyp und Bezeichner jedes verwendeten Parameters. Mehrere Parameter werden durch Komma getrennt. Parameter können auch ganz fehlen aber die öffnende und schließende runde Klammer der Parameterliste muss vorhanden sein. Im vorliegenden Fall besteht die Parameterliste aus einem Parameter von Datentyp double.

Der sich der Signatur anschließende Block (gekennzeichnet durch geschweifte Klammern) enthält die Anweisungen, die festlegen welche Aufgabe die Methode erfüllt. Dieser Block wird auch Methodenrumpf genannt. In unserem Beispiel wird zum bisherigen Saldo der Einzahlungsbetrag hinzuaddiert. Um zu verhindern, dass die Einzahlung auch für Auszahlungen genutzt wird, werden negative Werte ignoriert. Hierzu wird die von C# im Namensraum „System“ bereitgestellte Klasse **Math** genutzt, die eine Vielzahl von Methoden für die üblichen mathematischen Funktionen bereitstellt. Bei der Angabe von zwei numerischen Parametern gibt die Methode **Max** den größeren der beiden Werte zurück.

Anmerkung: Hierbei handelt es sich um statische Methoden (Schlüsselwort: *static*), die klassenweit definiert sind. Deshalb können diese auch direkt mit der Klasse aufgerufen werden ohne dass zuvor ein Objekt instanziiert werden muss.

Merke: Es handelt sich bei der Verwendung des Gleichheitszeichens nicht um eine Gleichung, sondern um eine Wertzuweisung!

In folgender Button-Click-Routine wird die Methode **Einzahlen()** verwendet. Um eine Methode einer Klasse aufzurufen benutzt man den Namen eines Objektes, gefolgt von einem Punkt und dem Namen der Methode. Im Anschluss werden in runden Klammern die Parameter angegeben. Im untenstehenden Codebeispiel wird die Methode **Einzahlen()** für das Objekt **MeinKonto** aufgerufen. Übergeben wird der Wert, der in der Variable **Betrag** gespeichert ist.

```
private void btnEinzahlen_Click(object sender, EventArgs e)
{
    double Betrag = Convert.ToDouble(tbxBetrag.Text);
    MeinKonto.Einzahlen(Betrag);
}
```

Die folgenden Screenshots zeigen die Wirkung des Aufrufs.

Konto

Kontonummer: 5005878
 Kontoinhaber: Müller, Hans
 Saldo: 2850,20
 Kreditlimit: 6000,00

Konto anlegen Konto anzeigen

Kontonummer: 5005878
 Kontoinhaber: Müller, Hans
 Saldo: 2.850,20 €
 Kreditlimit: 6.000,00 €

Ein-/Auszahlungen

Betrag:

Einzahlen Auszahlen

- ein Objekt wird initialisiert und angezeigt.
- aktueller Saldo: 2.850,20 €

Nach Eingabe des Betrages und Klicken der Schaltfläche **Einzahlen** wird der Saldo geändert, wie uns das anschließende Klicken der Schaltfläche **Konto anzeigen** auch ausgibt.

Konto

Kontonummer: 5005878
 Kontoinhaber: Müller, Hans
 Saldo: 2850,20
 Kreditlimit: 6000,00

Konto anlegen Konto anzeigen

Kontonummer: 5005878
 Kontoinhaber: Müller, Hans
 Saldo: 3.200,20 €
 Kreditlimit: 6.000,00 €

Ein-/Auszahlungen

Betrag: 350

Einzahlen Auszahlen

Aufgabe 1

Überarbeiten Sie Ihr Kontoprojekt entsprechend der obigen Ausführungen! Die Programmierung der Schaltfläche **Auszahlen** wird im folgenden Kapitel erläutert.

3. Verzweigungen und Fehlerbehandlung



Situationsbeschreibung

Nach Vorstellung des überarbeiteten Programms wird von den Sachbearbeitern an die Softwareentwickler als weitere Anforderung formuliert, dass Auszahlungen nicht zu einer Überschreitung des Kreditlimits führen dürfen.

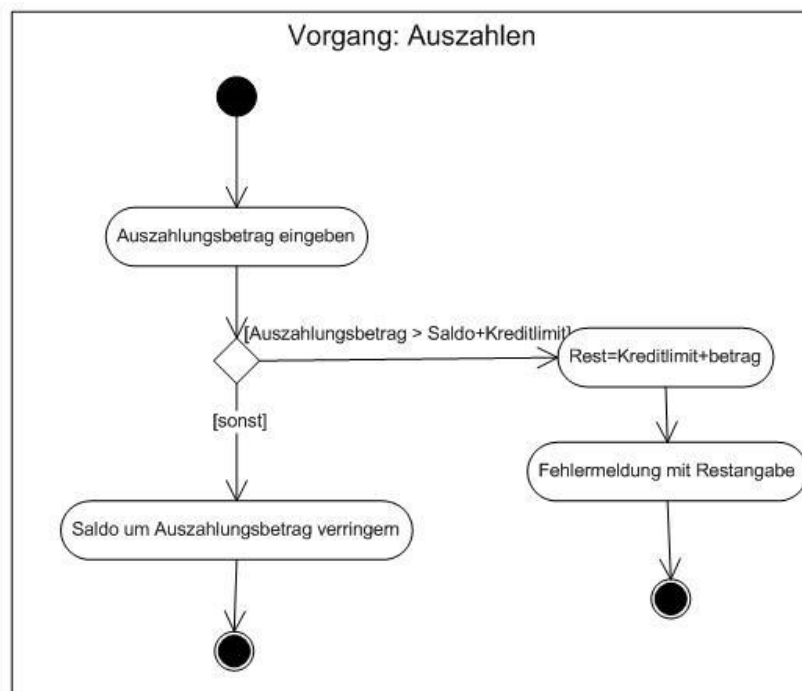
Kontoüberziehungen im Rahmen des Kreditlimits sind möglich. Bei Überschreitungen erfolgt ein Hinweis, dass die Abbuchung in entsprechender Höhe nicht möglich ist.

3.1 Entscheidungen treffen und Fehlerbehandlung

Die Anforderung der Fachabteilung wird von den Softwareentwicklern als **Pseudocode** formuliert:

```
WENN      gewünschter Auszahlungsbetrag <= Saldo + Kreditlimit DANN
          Auszahlung durchführen
SONST
          Warnhinweis mit maximal möglicher Kontobelastung
ENDE WENN
```

und anschließend in ein Aktivitätsdiagramm⁸ überführt.



⁸ Ausführliche Informationen zum Erstellen von Aktivitätsdiagrammen finden sich im UML-Skript.

Dieses Diagramm dient den Programmierern als Grundlage für die Erstellung des folgenden C#-Codes.

```
public void Auszahlen(double pBetrag)
{
    if (pBetrag > fSaldo + fKreditlimit)
    {
        throw new Exception("Auszahlung in Höhe von " + pBetrag + " € nicht möglich."
            + "\n" + "Maximal möglicher Auszahlungsbetrag: " + (Saldo + Kreditlimit) + " € ");
    }
    else
    {
        fSaldo = fSaldo - Math.Max(pBetrag,0);
    }
}
```

Um in einer Methode zum Ausdruck zu bringen, dass ein Fehler aufgetreten ist, wirft man eine Ausnahme. Hierzu dient das Schlüsselwort **throw**. Außerdem gibt es eine **Exception-Klasse** von der benutzerdefinierte Objekte erzeugt werden können. Beim Aufruf des Konstruktors kann ein eigener Fehlertext vergeben werden (Message). In unserem Fall kann das Werfen des Fehlers bei Überziehung des Kreditlimits wie folgt implementiert werden:

Wird mit **throw** ein Ausnahmeobjekt geworfen, so wird das Objekt an die rufende Methode weitergereicht und die aktuelle Methode wird verlassen.

Im folgenden Code (erscheint beim Doppelklick auf den **Auszahlen**-Button im Formular frmKonto.cs) sieht man wie das Fehlerobjekt aufgefangen wird (**try-catch**):

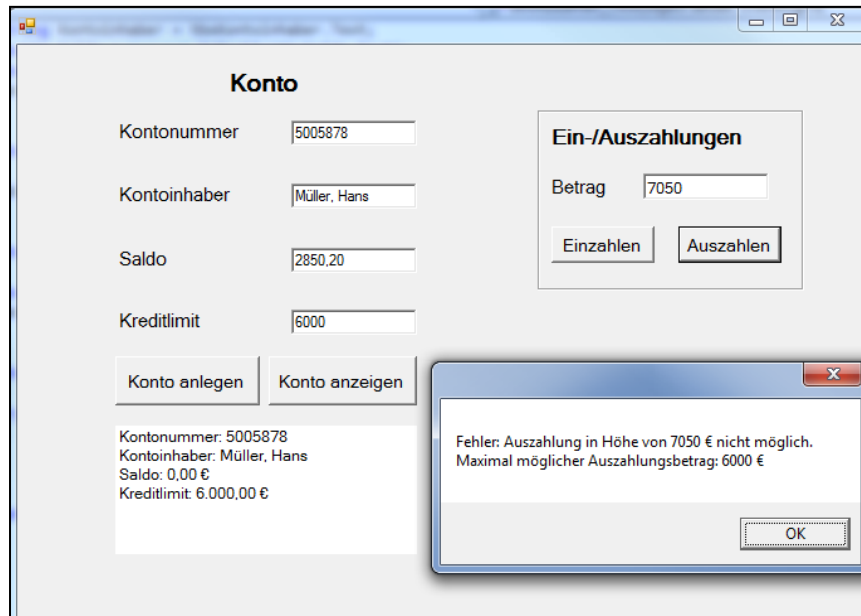
```
private void btnAuszahlen_Click(object sender, EventArgs e)
{
    try
    {
        double Betrag = Convert.ToDouble(tbxBetrag.Text);
        MeinKonto.Auszahlen(Betrag);
    }
    catch (Exception ex)
    {
        MessageBox.Show("Fehler: " + ex.Message);
    }
}
```

Alle Anweisungen, die zu einer Ausnahme führen können, sind im try-Block untergebracht. Die Reaktion auf die Ausnahme erfolgt im catch-Block, dem das geworfene Exception-Objekt übergeben wird. Es wird empfohlen Konvertierungsanweisungen immer mit try-catch auszuführen, um Bedienungsfehler der Oberfläche abzufangen.

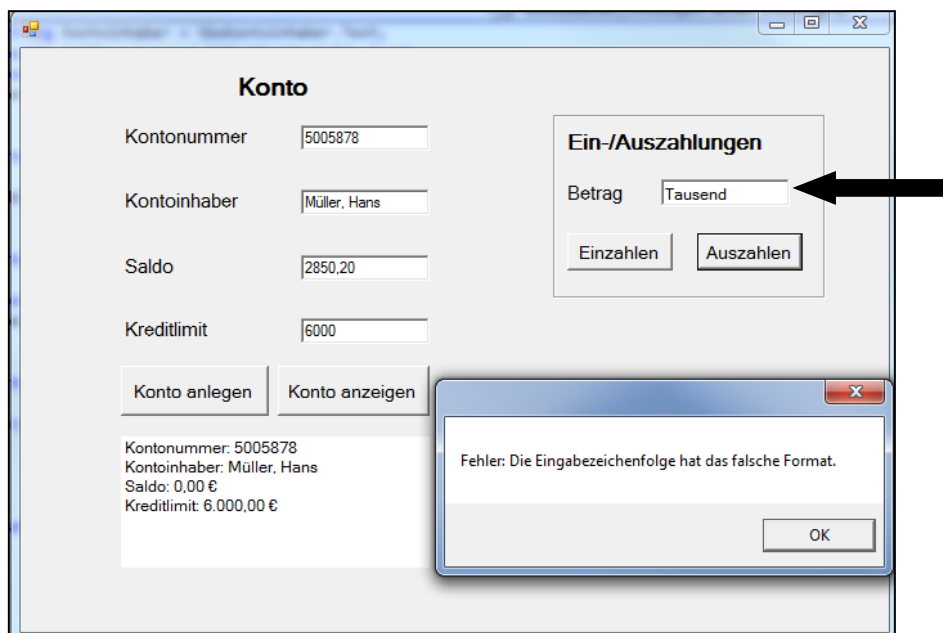
3.2 Test der selbst definierten Ausnahme

Die Programmierer testen das Programm nach Eingabe der Ausgangswerte im Anschluss an die Einzahlung von 350,00 €, indem sie eine Auszahlung von 3.200,20 € veranlassen. Erwartungsgemäß wird der Auszahlungswunsch akzeptiert, da das Kreditlimit nicht ausgeschöpft wird.

Anschließend überprüfen die Softwareentwickler, wie sich das Programm bei einem gewünschten Auszahlungsbetrag von 7.050,00 € verhält. Erwartungsgemäß wird die Auszahlung verweigert und es erfolgt ein Hinweis zu dem maximal möglichen Auszahlungsbetrag.



Ohne weiteren Code schreiben zu müssen, werden nun auch Eingabefehler des Benutzers sinnvoll abgefangen (hier wird die Klasse `System.InvalidCastException` verwendet, die im Namensraum „System“ definiert ist).



3.3 Entscheidungen treffen (Selektion)

Die `if`-Anweisung (= Selektion) hat den folgenden allgemeinen Aufbau:

```
if (Bedingung)
{
    Anweisungen bei erfüllter Bedingung – true
}
else
{
    Anweisungen bei nicht erfüllter Bedingung – false
}
```

Im Beispiel oben (Seite 23) wird der `If`-Block ausgeführt, wenn die Auswertung der Bedingung

`(pBetrag > fSaldo + fKreditlimit)`

den Wert `true` liefert. Ist das Ergebnis `false` wird der `Else`-Block ausgeführt (zweiseitige Auswahl). Der `Else`-Block muss nicht vorhanden sein, in diesem Fall spricht man von einseitiger Auswahl.

Eine Bedingung muss immer so formuliert werden, dass nur das Ergebnis `true` oder `false` möglich ist. Bedingungen können mit Hilfe von Vergleichs- und logischen Operatoren formuliert werden, die in folgender Übersicht zusammengestellt sind.

Vergleichsoperatoren

Operator	Funktion	Beispiel
<code>==</code> ⁹	gleich	<pre>int a=5; int b=7; bool gleich = (a==b);</pre> Die Variable <code>gleich</code> enthält den Wert <code>false</code> , da <code>a</code> und <code>b</code> verschieden sind.
<code>!=</code>	ungleich	<pre>bool ungleich = (a!=b);</pre> Die Variable <code>ungleich</code> enthält den Wert <code>true</code> , da <code>a</code> und <code>b</code> verschieden sind.
<code><</code>	kleiner	
<code><=</code>	kleiner gleich	
<code>></code>	größer	
<code>>=</code>	größer gleich	

⁹ Nicht zu verwechseln mit dem einfachen Gleichheitszeichen für Zuweisungen (häufige Fehlerquelle).

Logische Operatoren

Operator	Funktion	Beispiel/Erläuterung
&&	AND	<pre>bool a=true; bool b=false; bool aANDb = (a&&b);</pre> Die Variable aANDb enthält den Wert <code>false</code> , da a und b nicht beide <code>true</code> sind.
	OR	<pre>bool c = (a b);</pre> Die Variable c enthält den Wert <code>true</code> , da a <code>true</code> ist..
^	XOR	Eine von beiden Bedingungen ist wahr und die andere falsch.
!	NOT	<pre>bool c = !a;</pre> Die Variable c enthält den Wert <code>false</code> , da a <code>true</code> ist.

Übung zu Kapitel 3

Aufgabe 1

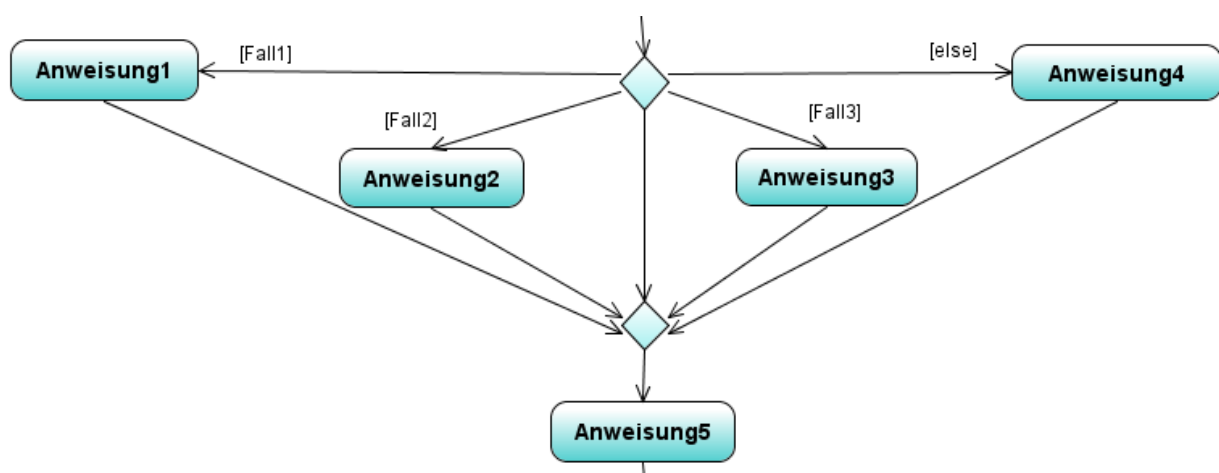
Verändern Sie die Methode **Auszahlen()** entsprechend den obigen Ausführungen und testen Sie diese mit selbst gewählten Testdaten!

Exkurs: Mehrseitige Auswahl

Immer dann, wenn eine Variable nur eine bestimmte Anzahl von Werte annehmen kann, und man diese Werte vergleichen muss, so bietet sich eine Mehrfachauswahl an.

Eine Mehrfach-Auswahl kann anstelle einer Verschachtelung mehrerer if-Anweisungen auch in Form einer Switch-Anweisung realisiert werden. Dabei wird der Wert des Switch-Ausdrucks mit aufgezählten Werten verglichen. Der Datentyp des Switch-Ausdrucks muss ganzzahlig (byte, short, int) oder char sein. Auch die Verwendung von Aufzählungstypen ist möglich.

UML-Aktivitätsdiagramm:



Syntax:

```
switch(Ausdruck) {  
    case wert1:  
        Anweisung1;  
        break;  
    case wert2:  
        Anweisung2;  
        break;  
    case wert3:  
        Anweisung3;  
        break;  
    default:  
        Anweisung4;  
}
```

Vorgehensweise:

- Trifft ein Fall zu, so werden die zugehörigen Anweisungen ausgeführt. Falls eine „break“-Anweisung vorhanden ist, springt das Programm an das Ende der Switch-Anweisung. Falls „break“ fehlt, werden alle Anweisungen bis zum nächsten „break“ oder bis zum Ende der Switch-Anweisung ausgeführt.
- Trifft kein Fall zu, so werden die default-Anweisungen ausgeführt.

Aufgabe 2

Die Klasse **Konto** wird um das Feld **fTyp** erweitert. **fTyp** kann folgende Werte annehmen.

1	PrivatKonto	Kontogebühr=5 EUR
2	Geschäftskonto	Kontogebühr=10 EUR
3	JugendKonto	Kontogebühr=0 EUR
4	MitarbeiterKonto	Kontogebühr=0 EUR
5	AuslandsKonto	Kontogebühr =20 EUR.

Erweitern Sie ihr Kontoprojekt so, dass für jedes Objekt der Klasse **Konto** eine Methode **berechneGebühr()** aufgerufen werden kann, die die Kontogebühr zurückgibt. Falls der Typ einen anderen als die oben angegebenen Werte hat, wird ein Fehler mit passender Meldung geworfen.

4. Testen und Dokumentieren

4.1 Der Debugger

Visual Studio 2010 besitzt einen Debugger¹⁰, mit dem die Programmausführung hinsichtlich der Ablaufstrukturen und Variableninhalte zwecks des Auffindens von Fehlern überwacht werden kann.

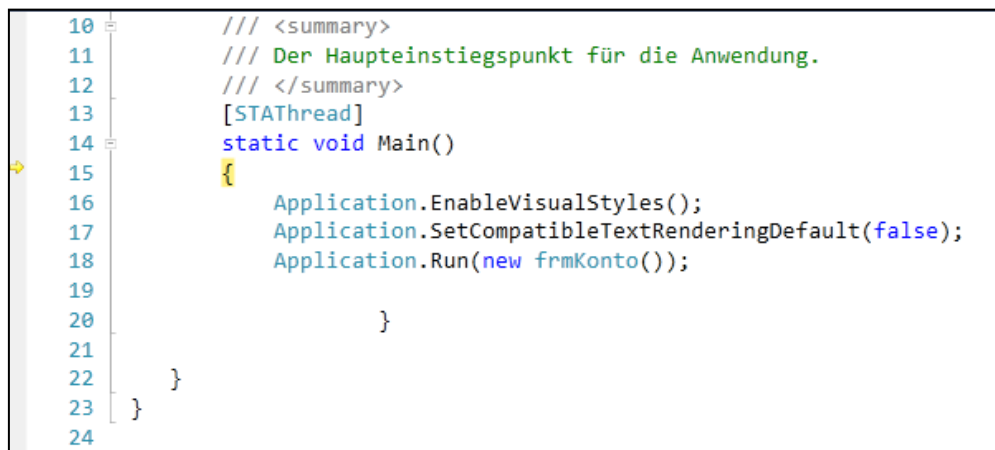
Der Debugger kann zur besseren Nachvollziehbarkeit des Programmablaufs in Einzelschritten ausgeführt werden. Dies geschieht über die Taste F11 oder über den Menüpunkt <Debuggen> - <Einzelschritt>.

4.1.1 Einzelschrittverfahren

Sie können ein Programm im Einzelschrittverfahren ablaufen lassen, um sich bei jedem einzelnen Schritt die aktuellen Inhalte der Variablen und Steuerelemente anzuschauen. Dabei beginnen Sie mit dem Menüpunkt **DEBUGGEN – EINZELSCHRITT** (alternativ: **Taste F11** drücken).

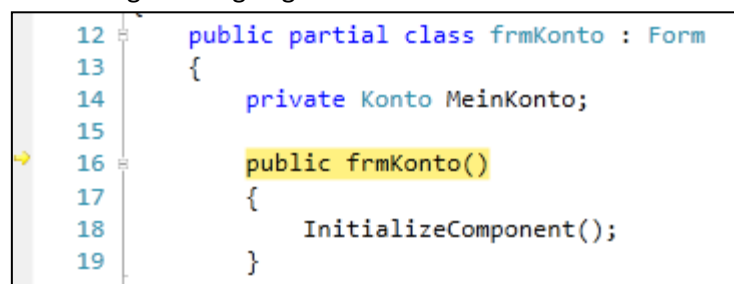
Nach dem Start des Einzelschrittverfahrens startet die Anwendung, zunächst mit einigen automatisch erzeugten Teilen des Programmcodes.

- Es beginnt mit der Methode `Main()`, mit der jedes C#-Programm beginnt. Ein gelber Pfeil vor einer gelb markierten Zeile kennzeichnet den Punkt, an dem das Programm gerade angehalten wurde und auf die Reaktion des Entwicklers wartet.



```
10  /// <summary>
11  /// Der Haupteinstiegspunkt für die Anwendung.
12  /// </summary>
13  [STAThread]
14  static void Main()
15  {
16      Application.EnableVisualStyles();
17      Application.SetCompatibleTextRenderingDefault(false);
18      Application.Run(new frmKonto());
19
20  }
21
22 }
23
24 }
```

- In Zeile 18 sieht man mit welchem Formular diese Anwendung startet. Hier wird ein neues Objekt der Klasse `frmKonto` erzeugt.
- Nach einigen weiteren Einzelschritten (F11) wechselt das Programm in die Klasse `frmKonto.cs`. Es durchläuft dort u.a. die Methode `InitializeComponent()`, in der die Eigenschaften und das Verhalten der Steuerelemente festgelegt werden, sowie man dies zuvor im Oberflächendesigner festgelegt hat.



```
12  public partial class frmKonto : Form
13  {
14      private Konto MeinKonto;
15
16      public frmKonto()
17      {
18          InitializeComponent();
19      }
20  }
```

¹⁰ „debug ... (computing) to look for and remove the faults in a computer program“ aus: OXFORD Advanced Learner's dictionary

- Nach einigen weiteren Einzelschritten, in denen unter anderem die Steuerelemente eingelesen werden, erscheint das Formular, in dem Kontonummer, Kontoinhaber, Saldo und Kreditlimit eingegeben werden können. Betätigen Sie den Button **Konto anlegen**, wird nun die Ereignismethode angezeigt. Nach zwei weiteren Einzelschritten steht das Programm in der Zeile mit `string Kontoinhaber = tbxKontoinhaber.Text;`
- Platzieren Sie den Cursor über einer Variablen oder eine Steuerelementeigenschaft (z.B. über der Variablen **Kontonummer**), so sehen Sie den aktuellen Wert. Sie können erkennen, dass die Variable **Saldo** noch den Wert 0 hat, da die aktuell markierte Anweisung noch nicht ausgeführt wurde.
- Bereits nach dem nächsten Einzelschritt hat die Variable **Saldo** den Wert 2850.2. Nach Durchführung aller Einzelschritte erscheint das Ergebnis des Programms wie gewohnt in der Anwendung.

Dieses Beispiel zeigt, dass Sie mit dem Einzelschrittverfahren bereits den Ablauf eines Programms stückweise verfolgen können und so den Ursprung eines logischen Fehlers leichter lokalisieren können.

4.1.2 Haltepunkte

Bei umfangreichem Programmcode kann das Einzelschrittverfahren sehr lange dauern. Um genau zu einer bestimmten Anweisung zu gelangen wo man den Ursprung eines Fehlers vermutet, behilft man sich so genannter Haltepunkte.

Das setzen eines Haltepunktes geschieht mithilfe des Menüpunkts **DEBUGGEN – HALTEPUNKT UMSCHALTEN**. Alternativ kann die **Taste F9** verwendet werden, oder einfach in den linken Rand der Zeile geklickt werden. Es wird ein Haltepunkt in der Zeile gesetzt, in der sich der Cursor befindet, bzw. in dessen Randbereich man geklickt hat. Im Beispiel bietet sich hierfür die Zeile

`long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);`

an in der die Kontonummer eingelesen und in eine Ganzzahl umgewandelt wird (Abbildung unten).

```

26 private void btnAnlegen_Click(object sender, EventArgs e)
27 {
28     long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
29     string Kontoinhaber = tbxKontoinhaber.Text;
30     double Saldo = Convert.ToDouble(tbxSaldo.Text);
31     double Kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
32     MeinKonto = new KONTO(Kontonummer, Kontoinhaber, Saldo, Kreditlimit);
33 }

```

Das Programm starten Sie nun über die Funktionstaste F5. Es unterbricht vor der Ausführung der Zeile mit dem Haltepunkt. Ab diesem Punkt können Sie das Programm wiederum im Einzelschrittverfahren ablaufen lassen und die Werte der Variablen wie oben beschrieben kontrollieren. Sie können auch mehrere Haltepunkte setzen. Sie entfernen einen Haltepunkt, indem Sie in den Cursor in der entsprechenden Zeile platzieren und F9 betätigen oder einfach auf den Haltepunkt selbst klicken.

4.1.3 Überwachungsfenster

Das Überwachungsfenster bietet während des Debuggens eine weitere komfortable Lösung zur Variablenkontrolle. Sie können es während des Debuggens über den Menüpunkt **DEBUGGEN – FENSTER – ÜBERWACHEN** einblenden.

Dort können Sie die Namen von Variablen oder von Steuerelement-Eigenschaften in der Spalte Name eingeben. In der Spalte Wert erscheint dann jeweils der aktuelle Wert beim Ablauf der Einzelschritte (siehe Abbildung unten). Auf diese Weise lässt sich die Entwicklung mehrerer Werte gleichzeitig komfortabel verfolgen.

Überwachen 1		
Name	Wert	Typ
Kontonummer	5007878	long
Kontoinhaber	"Müller, Hans"	string
Saldo	2850.2	double
Kreditlimit	6000.0	double

4.2 Kommentare erstellen und aufbereiten

Computerprogramme werden in Projekten arbeitsteilig erstellt. Mitarbeiter in solchen Projekten müssen Klassen verwenden, die sie nicht selbst geschrieben haben. Hier ist es wichtig, dass durch eine entsprechende Dokumentation die Funktionsweise einer Klasse beschrieben wird.

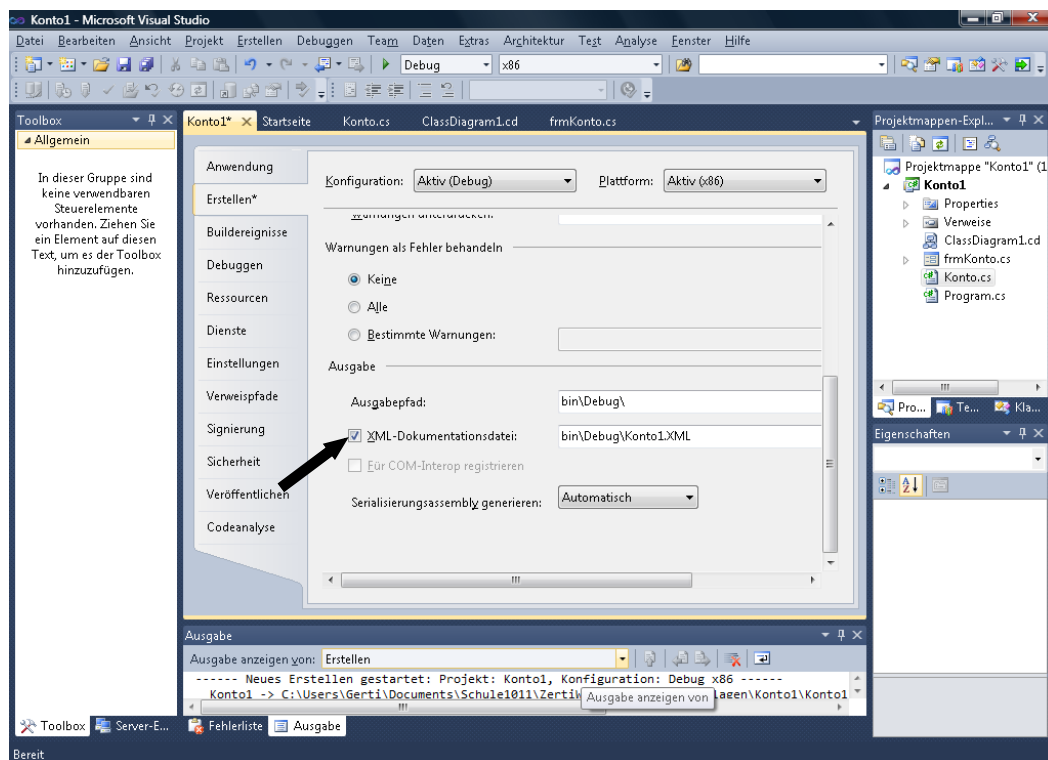
Auch wird es während des laufenden Betriebs notwendig sein, Klassen zu überarbeiten. Da in vielen Fällen eine personelle Identität von Ersteller und Modifizierer nicht gegeben sein dürfte, erleichtert eine aussagekräftige Dokumentation die notwendigen Anpassungen.

Neben dem „Einstreuen von Kommentaren“ mit Hilfe der Steuerzeichen `//` für einzeilige Kommentare und `/* */` für mehrzeilige Kommentare gibt es die Möglichkeit z.B. Klassen strukturiert zu kommentieren. Hierzu beginnt man den Kommentar in der Zeile über einem Member (Codeblock) mit drei Schrägstrichen `///`. Dann wird ein XML-Kommentar erzeugt, für den feste Strukturmöglichkeiten zur Verfügung stehen. Die XML-Elemente **summary**, **remarks** und **example** können vor jeder beliebigen Deklaration verwendet werden. Bei Methoden können zusätzlich die Parameter, der Rückgabewert und eventuell ausgelöste Ausnahmen beschrieben werden. Außer diesen Strukturierungsmerkmalen können noch weitere Formatierungstags verwandt werden:

Tag	Verwendung	Attribute
c	Text wird als einzeiliger Sourcecodeauszug formatiert	keine
code	Für mehrzeilige Sourcecodeauszüge	keine
description	Beschreibung	keine
item	Punkt in einer Liste	keine
list	Liste	type
listheader	Listenkopfzeile	keine
para	Fügt einen Absatz ein (wie <code><p></code> in HTML)	keine
paramref	Zeichnet einen Text als Parameter aus.	name *
see	Fügt einen Verweis innerhalb eines XML-Kommentars ein	cref *
term	Begriff, der durch <code><description></code> definiert wird	keine
value	Beschreibt eine Eigenschaft	keine

Sämtliche Kommentare kann man nun direkt beim Kompilieren extrahieren und in eine XML-Datei ausgeben lassen. Hierfür ist folgende Vorbereitung zu treffen:

Zu öffnen ist das Eigenschaftsfenster des Projektes (**Projekt -> Konto-Eigenschaften...**) und dann unter dem Menüpunkt „Erstellen“ aktivieren der XML-Dokumentationsdatei. Bei Bedarf kann zudem der relative Pfad angepasst werden. Default-Einstellung ist der Pfad bin\Debug\.



Die erzeugte Dokumentationsdatei für die Klasse **Konto** sieht dann beispielsweise so aus:

```
<?xml version="1.0"?>
<doc>
  <assembly>
    <name>Konto</name>
  </assembly>
  <members>
    <member name="T:Konto.KONTO">
      <summary> Hier werden alle Felder und Methoden der Klasse Konto erfasst </summary>
      <remarks> Copyright 2011, MM </remarks>
    </member>
    <member name="F:Konto.KONTO.fKontonummer">
      <summary> Zunächst ohne festen Erstellungsalgorithmus </summary>
    </member>
    <member name="F:Konto.KONTO.fKontoinhaber">
      <summary> Hier wird der Name des Kontoinhabers festgehalten </summary>
    </member>
    <member name="M:Konto.KONTO.#ctor(System.Int64,System.String,System.Double,System.Double)">
      <summary> vollständiger Konstruktor </summary>
      <param name="pKontonummer"> N.N. falls noch nicht bekannt </param>
      <param name="pKontoinhaber"/>
      <param name="pSaldo"> nur positive Werte zugelassen </param>
      <param name="pKreditlimit"> nur positive Werte zugelassen </param>
    </member>
    <member name="M:Konto.KONTO.Einzahlen(System.Double)">
      <summary> Kurzschreibweise genutzt </summary>
      <param name="pBetrag"/>
    </member>
    <member name="P:Konto.KONTO.Kontonummer">
      <summary> readonly </summary>
    </member>
    <member name="P:Konto.KONTO.Saldo">
      <summary> readonly </summary>
    </member>
    <member name="M:Konto.Program.Main">
      <summary> Der Haupteinstiegspunkt für die Anwendung. </summary>
    </member>
    <member name="F:Konto.frmKonto.components">
      <summary> Erforderliche Designervariable. </summary>
    </member>
    <member name="M:Konto.frmKonto.Dispose(System.Boolean)">
      <summary> Verwendete Ressourcen bereinigen. </summary>
      <param name="disposing"> True, wenn verwaltete Ressourcen gelöscht werden sollen; andernfalls False. </param>
    </member>
    <member name="M:Konto.frmKonto.InitializeComponent">
      <summary> Erforderliche Methode für die Designerunterstützung. Der Inhalt der Methode darf nicht mit dem Code-Editor geändert werden. </summary>
    </member>
    <member name="T:Konto.Properties.Resources">
      <summary> Eine stark typisierte Ressourcenklasse zum Suchen von lokalisierten Zeichenfolgen usw. </summary>
    </member>
    <member name="P:Konto.Properties.Resources.ResourceManager">
      <summary> Gibt die zwischengespeicherte ResourceManager-Instanz zurück, die von dieser Klasse verwendet wird. </summary>
    </member>
    <member name="P:Konto.Properties.Resources.Culture">
      <summary> Überschreibt die CurrentUICulture-Eigenschaft des aktuellen Threads für alle Ressourcenlookups, die diese stark typisierte Ressourcenklasse verwenden. </summary>
    </member>
  </members>
</doc>
```

Dies sieht in der Tat zunächst sehr unübersichtlich aus. Es gibt aber entsprechende Tools wie z.B. **NDoc** oder **Sandcastle** die diese XML-Dateien als gut lesbares HTML-Dokument aufbereiten. Man kann aber auch weitgehend ohne zusätzliche Hilfsmittel auskommen, wenn man eine passende XSL-Schemadatei zur Anzeige verwendet. Eine solche Schemadatei findet man z.B. unter dem folgenden Link zum Download:

<http://dotnet.jku.at/docview/>

Man lädt die XSL-Datei und die XML-Datei in denselben Ordner und öffnet die XML-Datei zum Editieren z. B. mit Visual Studio 2010. Anschließend ergänzt man hinter der Zeile

```
<?xml version="1.0"?>
```

die Zeile

```
<?xml-stylesheet type="text/xsl" href="Documentation.xsl"?>
```

Öffnet man nun die XML-Datei mit Hilfe eines Browsers (z.B. mit dem Internet Explorer) so wird die Datei wie folgt aufbereitet:

Konto

Hier werden alle Felder und Methoden der Klasse Konto erfasst

copyright 2010, KU

Fields

fKontonummer

Zunächst ohne festen Erstellungsalgorithmus

fKontoinhaber

Hier wird der Name des Kontoinhabers festgehalten

Später zu ersetzen durch Objekt

Properties

Kontonummer

readonly

Saldo

readonly

Bezieht man nun noch eine geeignete CSS-Datei mit ein, ist das Ergebnis schon sehr ansprechend:

Konto1
Properties.Resources Eine stark typisierte Ressourcenklasse zum Suchen von lokalisierten Zeichenfolgen usw.
Properties ResourceManager Gibt die zwischengespeicherte ResourceManager-Instanz zurück, die von dieser Klasse verwendet wird. Culture Überschreibt die CurrentUICulture-Eigenschaft des aktuellen Threads für alle Ressourcenlookups, die diese stark typisierte Ressourcenklasse verwenden.
Konto Hier werden alle Felder und Methoden der Klasse Konto erfasst copyright 2010, KU
Fields fKontonummer Zunächst ohne festen Erstellungsalgorithmus fKontoinhaber Hier wird der Name des Kontoinhabers festgehalten Später zu ersetzen durch Objekt

Um den Bezug zur CSS-Datei herzustellen wurde die XSL-Datei zwischen dem HTML-Tag und dem BODY-Tag um folgende Zeilen ergänzt:

```
<HTML>
  <HEAD>
    <TITLE>
      <xsl:value-of select="doc/assembly/name" />
    </TITLE>
    <LINK rel="stylesheet" type="text/css" href="doc.css" />
  </HEAD>
  <BODY>
```

Die Datei doc.css kann auf der Seite <http://dotnet.jku.at/docview/> heruntergeladen werden.

Zusätzlich ist es möglich an beliebigen Stellen im Code einzeilige Kommentare (Steuerzeichen //) und mehrzeilige Kommentare (zwischen /* und */) einzufügen. Diese erscheinen aber nicht in der zuvor vorgestellten XML-Datei.

Will man diese ebenfalls aufbereiten können externe Tools wie **Ndoc** oder **Sandcastle** verwendet werden.

Aufgabe 1

Dokumentieren Sie das Konto-Projekt in Analogie zu der Darstellung in Kapitel 4.2.

5. ArrayLists und typisierte Listen

Die Daten, die ein Benutzer über die Eingabefelder an der Grafischen Oberfläche eingibt, stehen nur während der Ausführung der Eingabeprozedur zur Verfügung. Wenn der Benutzer Daten für ein neues Objekt eingibt, werden die alten Objektdaten überschrieben. Auf zuvor eingegebene Objektdaten kann er somit im Nachhinein nicht mehr zugreifen.

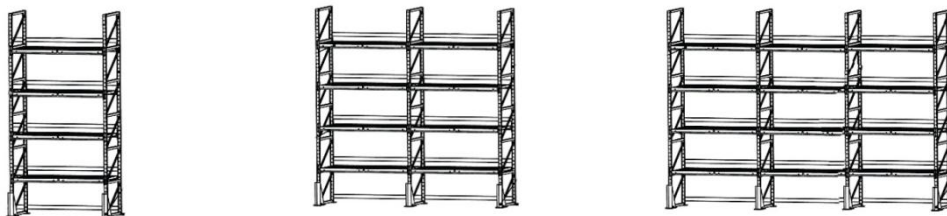
In der Praxis sollen die eingegebenen Daten natürlich nicht verloren gehen, sondern dem Programm weiterhin zur Verfügung stehen. Hierzu müssen die Daten zwischengespeichert werden. Für diese Zwischenspeicherung stehen Objektsammlungen bzw. Datencontainer des Namensraums `System.Collections` bzw. `System.Collections.Generic` zur Verfügung. Hierbei ist die `ArrayList` eine sehr flexible Datenstruktur, die eine Liste von Werten beliebigen Datentyps (z.B. auch Objekte einer Klasse) speichern kann. Sehr einfach können Werte hinzugefügt, eingefügt und gelöscht werden. Die Flexibilität beruht darauf, dass die Struktur dynamisch wächst, man muss also nicht im Vorfeld wissen, wie viele Elemente maximal eingefügt werden.

Die `ArrayList` ist teilweise vergleichbar mit einem Array, hat gegenüber Arrays aber den Vorteil, dass der Container eine dynamische Größe hat. Die Größe einer `ArrayList` muss nicht von vornherein angegeben werden. Zudem kann sie jederzeit vergrößert und verkleinert werden.

Allerdings sind die einzelnen Elemente einer `ArrayList` alle vom Datentyp `Object`. D. h., dass beliebige Werte aufgenommen werden können, wenn aber die einzelnen Werte wieder in der ursprünglichen Form verwendet werden sollen, müssen diese Werte erst gecastet (konvertiert) werden. Dies kostet Laufzeit und ist zudem fehleranfällig. Wird nicht der korrekte Cast durchgeführt entsteht ein Fehler zur Laufzeit. Bei generischen Auflistungen ist der Vorteil, dass für die aufzunehmenden Werte ein Typ festgelegt wird. Dies kann dann auch ein selbstdefinierter Datentyp z.B. eine Klasse sein. Die Auflistungsklasse `List` entspricht in ihrer Verwendung der nichtgenerischen Klasse `ArrayList`, mit dem Vorteil, dass die Auflistung typsicher ist. Der Compiler kann Fehler durch falsch eingegebene Datentypen bereits zur Compilierzeit abfangen und nicht erst zur Laufzeit.

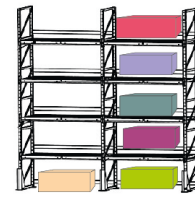
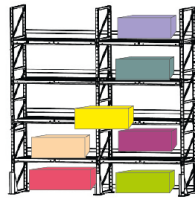
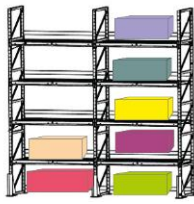
5.1 Vergleich von typisierten Listen und ArrayLists¹¹ mit Regalen

Eine `ArrayList` ist wie ein Regal, das sich automatisch vergrößert, wenn Sie mehr Platz benötigen.

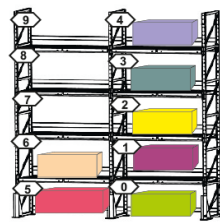


In einer `ArrayList` können allerdings nur Pakete (Objekte) gelagert werden. Der Lagerplatz wird lückenlos ausgenutzt. Wird ein Paket entnommen, kommen Heinzelmännchen und stapeln den Rest um.

¹¹ nach Werner Löhr



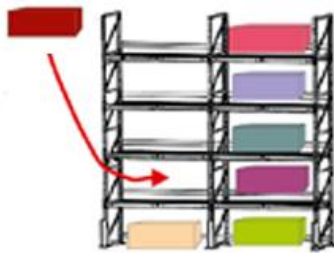
Ein Heinzelmännchen zählt stets die Objekte der Sammlung.



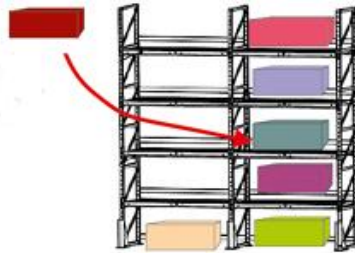
ArrayList



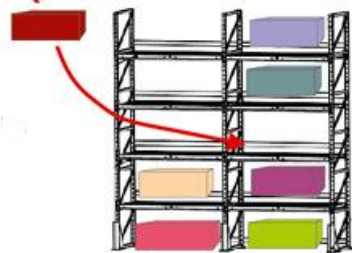
Ein neues Objekt wird im Normalfall am Ende hinzugefügt.



Falls Sie jedoch eine vordere Position haben wollen, stapeln die Heinzelmännchen um.



Nach dem Umstapeln kann das Objekt an der gewünschten Position eingelagert werden.



Man kann nicht irgendein Regal nutzen, sondern muss für jeden Zweck ein geeignetes Regal kaufen.

List <Flasche>



List <Topfblume>



List <Schuh>



Auf die Elemente einer ArrayList kann man nacheinander zugreifen. Man kann dies manuell tun, wie z. B. der Bäcker, der ein Backblech nach dem anderen (wie ein Iterator) zur Kontrolle aus dem Ofen herausschiebt. Der Zugriff lässt sich aber auch automatisieren, wie z.B. bei einem automatischen Vollauszug.

5.2 Erzeugung von ArrayLists und List<Typ>

Bei der Deklaration einer List ist in spitzen Klammern zusätzlich der Typ der Objekte anzugeben, die in der Liste zugelassen werden sollen.

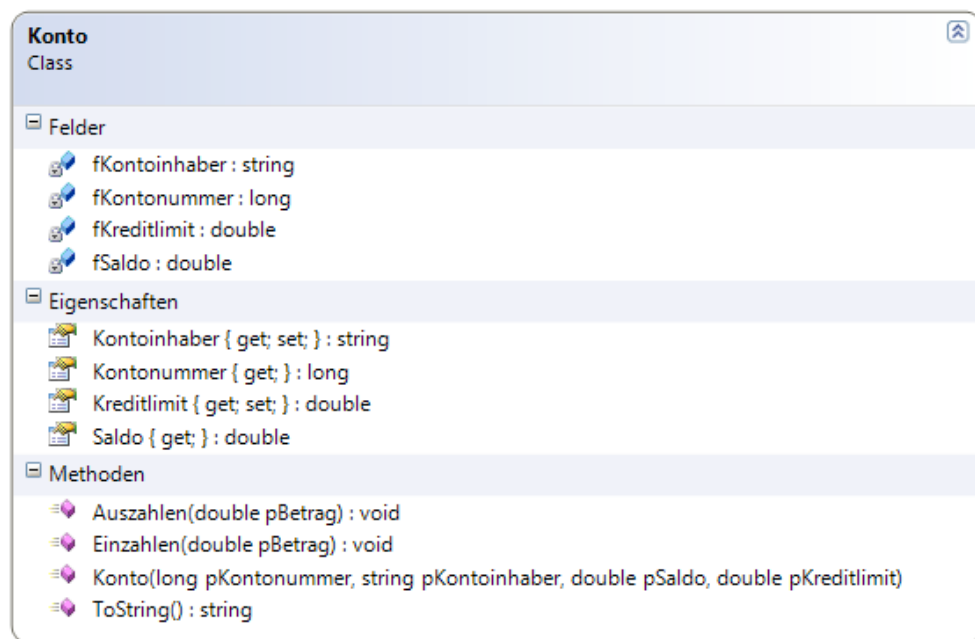
Syntax:

```
ArrayList listenName;  
listenName = new ArrayList()  
  
List<Typ> listenName;  
listenName = new List<Typ>()
```

5.3 Tabellarische Darstellung

Um die Komplexität des Quellcodes möglichst gering zu halten, setzen wir an dieser Stelle auf dem Kontoverwaltungs-Beispiel aus Kapitel 4 auf, reduzieren dieses allerdings um die XML-Kommentare. Die **Ausgabe()**-Methode, die eine formatierte Ausgabe der Informationen zu einem Objekt liefert, wird nun durch die **ToString()**-Methode ersetzt. Die ToString-Methode ist nämlich ohnehin für jede Klasse automatisch vorhanden, da jede Klasse ja vom Datentyp Object abgeleitet ist. Nur liefert standardmäßig die Methode ToString() den vollqualifizierten Namen eines Objekts zurück, z.B. Kontoverwaltung.Konto. Da diese Information i.d.R. keinen Nutzen hat, sollte jede benutzerdefinierte Klasse die ToString()-Methode geeignet überschreiben. Daher nutzen wir ab diesem Abschnitt die ToString()-Methode¹², um Informationen über die Objekte einer Klasse formatiert anzuzeigen. Zusätzlich zu der bisherigen Methodendefinition muss noch das Schlüsselwort „**override**“ zwischen dem Sichtbarkeitsattribut und dem Rückgabetyt eingefügt werden. So wird sichergestellt, dass der Programmierer diesen Schritt bewusst macht und nicht aus Versehen eine bestehende Methode überschreibt.

Basis für diesen Abschnitt ist die Klasse **Konto** mit folgender Struktur:



¹² Wir zeigen dieses Vorgehen exemplarisch für die ToString-Methode der Klasse Object. Ähnlich sollte man mit den weiteren Methoden der Object-Klasse verfahren, also mit Equals(), GetHashCode(), GetType().

Die Tabellarische Darstellung von Ergebnissen ist oft erwünscht. Anstelle der bisherigen Auflistung der Objektdaten untereinander möchte man häufig die Daten mehrerer Objekte tabellarisch anzeigen (s. Abbildung):

Nr.	Kontonummer	Kontoinhaber	Saldo	Kreditlimit
1	5005878	Müller, Hans	2850,20€	6000,00€
2	5005612	Schmidt, Lolita	2300,00€	5000,00€
3	7205822	Merck, Paul	2850,20€	0,00€

Anzahl der Konten: 3				

Hierfür kann man in C# ein eigenes Steuerelement nutzen (DataGridView), das wir an späterer Stelle näher vorstellen möchten, oder die Ausgabe so formatieren, dass jedes Objekt eine Zeile benötigt und jede Information an festen Positionen in der Zeile steht. Hierfür ist Voraussetzung, dass man in der Oberfläche für die Darstellung der Tabelle immer eine Festbreitenschrift, also eine Schrift in der jedes Zeichen den gleichen Raum einnimmt, verwendet. Ein solcher Schrifttyp ist z.B. **Courier New**.

Hilfreich zum Entwerfen einer Tabelle sind Vorüberlegungen, in denen man sich für jede dargestellte Information überlegt, wie viel Zeichen diese maximal beanspruchen darf. Als Planungsinstrument für die obenstehende Tabelle kann z.B. die folgende Tabelle helfen:

Überschrift	Länge der Über- schrift	Darstellung (Maximum)	Maximale Länge+Leer- plätze	Rechts - bzw. links- bündig	Format- anweisung
Nr.	3	999	3+3	lb	{0,-6}
Kontonummer	11	0050050878	11+2	lb	{1,-13}
Kontoinhaber	12	Müller, Hans-Di	15	lb	{2,-15}
Saldo	5	1112540,12 €	10	rb	{3,10:F2}
Kreditlimit	10	12000,00 €	10	rb	{4,10:F2}

Im konkreten Fall ist die Methode ToString() in der Klasse Konto wie folgt überschrieben (Die erste Spalte Nr. wird hier noch vernachlässigt, diese wird erst für die Objektsammlung Kontenliste interessant):

```
public override string ToString()
{
    string ausgabeText = "";
    string neueZeile = "\n";
    ausgabeText += String.Format("{0,-13} {1,-15} {2,10:F2}€ {3,10:F2}€",
        Kontonummer, Kontoinhaber, Saldo, Kreditlimit) + neueZeile;
    return ausgabeText;
}
```

Die statische Methode *Format* der Klasse String erlaubt die formatierte Ausgabe von einem oder mehreren Werten. Die Methode Format erhält als Parameter einen **Formatstring** mit den gewünschten Ausgabeoptionen sowie die Liste der auszugebenden Werte. Der Formatstring muss

nun für jeden auszugebenden Wert einen nummerierten Platzhalter einfügen (oben in der Planungstabelle fettgedruckt).

Der Aufruf ersetzt {0} durch den Wert von Kontonummer, {1} durch den Wert von Kontoinhaber, {2} durch den Wert von Saldo sowie {3} durch den Wert von Kreditlimit. Die Platzhalterangabe kann nun durch spezielle Formatoptionen erweitert werden.

{Platzhalternr [, Feldbreite] [:Formatierungscode [, Präzisierung]]}

Die eckigen Klammern besagen, dass die Angabe dieser Optionen optional ist. Die Feldbreite ist vorzeichenbehaftet: positiv = rechtsbündig, negativ=linksbündig. Die Präzisierung gibt i. d. R. die Anzahl der Nachkommastellen an.

Die folgenden Code-Snippets (Auszüge) zeigen die Wirkung einiger wichtiger Formatierungscodes:

```
double x = 12000.3457;
double z = -123214.345;
int k = 123456;
int l = -1234321;
lblAusgabe.Text = "| Format-Aufruf      |      Ausgabe      |";
lblAusgabe.Text += "\n| ({0,18:f2})\n,x) | " + String.Format("{0,18:f2} |", x);
lblAusgabe.Text += "\n| ({0,-18:f2})\n,x) | " + String.Format("{0,-18:f2} |", x);
lblAusgabe.Text += "\n| ({0,18:c2})\n,x) | " + String.Format("{0,18:c2} |", x);
lblAusgabe.Text += "\n| ({0,18:c2})\n,z) | " + String.Format("{0,18:c2} |", z);
lblAusgabe.Text += "\n| ({0,18:d10})\n,k) | " + String.Format("{0,18:d10} |", k);
lblAusgabe.Text += "\n| ({0,18:n4})\n,x) | " + String.Format("{0,18:n4} |", x);
```

Hierbei werden folgende Formatierungscodes verwendet:

Formatierungscode	Wirkung
f, F	Feste Anzahl von Nachkommastellen (Standardwert 2)
n, N	Wie f, aber mit Tausender Punkten
c, C	Währungsformat (abh. von Ländereinstellung)
d, D	Dezimalformat mit führenden Nullen

An Hand der folgenden Abbildung kann man die Wirkung der oben verwendeten Formatierungscodes nachvollziehen:

Format-Aufruf	Ausgabe
({0,18:f2})\n,x)	12000,35
({0,-18:f2})\n,x)	12000,35
({0,18:c2})\n,x)	12.000,35 €
({0,18:c2})\n,z)	-123.214,35 €
({0,18:d10})\n,k)	0000123456
({0,18:n4})\n,x)	12.000,3457

Falls man mit Hilfe der Formatcodes die gewünschte Ausgabeform nicht erreichen kann, ist es auch möglich benutzerdefinierte Formate (ähnlich wie in Excel) zu verwenden.

Hierzu noch einige weitere Beispiele:

```
lblAusgabe.Text = "| Format-Aufruf | Ausgabe |";
lblAusgabe.Text += "\n| \"{0,18:0.00}\" |, x) |" + String.Format("{0,18:0.00} |", x);
lblAusgabe.Text += "\n| \"{0,18:#,###.00}\" |, x) |" + String.Format("{0,18:#,###.00} |", x);
lblAusgabe.Text += "\n| \"{0,-18:#,###. #}\" |, x) |" + String.Format("{0,-18:#,###. #} |", x);
lblAusgabe.Text += "\n| \"{0,18:0.00%}\" |, x) |" + String.Format("{0,18:0.00%} |", x);
lblAusgabe.Text += "\n| \"{0,18:####E+0}\" |, x) |" + String.Format("{0,18:####E+0} |", x);
```

Zugehörige Ausgabe:

Format-Aufruf	Ausgabe
("{0,18:0.00}", x)	12000,35
("{0,18:#,###.00}", x)	12.000,35
("{0,-18:#,###. #}", x)	12.000,3
("{0,18:0.00%}", x)	1200034,57%
("{0,18:####E+0}", x)	1200E+1

Weiter sei noch darauf hingewiesen, dass es auch für die Formatierung von Datum und Zeit hilfreiche Formatcodes gibt. Hierzu ebenfalls ein paar Beispiele. Eine ausführliche Behandlung der möglichen Codes finden Sie z. B. unter <http://openbook.galileocomputing.de/csharp/kap30.htm>.

Format-Aufruf	Ausgabe
("{0:d} {1:h:mm}", h, h)	01.11.2010 3:38
("{0:dddd, 'den' dd.MMMM yyyy}", s)	Montag, den 01.November 2010
(" {0:t}", h)	15:38
(" {0:T}", h)	15:38:43
(" {0:T}", h)	15:38:43

Die laufende Nummer, sowie die Überschriften der Tabellenspalten werden erst bei der ToString()-Methode der Kontenliste berücksichtigt.

5.4 Wichtige Elemente zur Arbeit mit Auflistungen

Die einzelnen Aufgaben der Auflistung sollen an Hand der Button_Click-Routinen der folgenden Anwendung demonstriert werden:

The screenshot shows a Windows application window titled "Sparkasse Kleve". It contains several input fields and buttons for account management. On the right, there is a sub-window titled "Ein-/Auszahlungen" with a "Betrag" field and "Einzahlen" and "Auszahlen" buttons. Below the input fields are four buttons: "Konto anlegen", "Alle Konten anzeigen" (highlighted with a blue border), "Felder leeren", and "Konto löschen". At the bottom, there is a table with five columns: "Nr.", "Kontonummer", "Kontoinhaber", "Saldo", and "Kreditlimit". The table contains three rows of data. Below the table, it says "Anzahl der Konten: 3".

Nr.	Kontonummer	Kontoinhaber	Saldo	Kreditlimit
1	5005878	Müller, Hans	2850,20€	6000,00€
2	5005612	Schmidt, Lolita	2300,00€	5000,00€
3	7205822	Merck, Paul	2850,20€	0,00€

Anzahl der Konten: 3

Eine Liste füllen und ausgeben

Im Kontobeispiel werden beim **frmKonto_Load**-Ereignis drei neue Konten angelegt. Beim Betätigen des **Alle Konten anzeigen**-Buttons werden die zuvor angelegten Kontodaten angezeigt. (s.o.) Die eingegebenen Daten sollen in einer Auflistung zwischengespeichert werden und ihr sollen über Eingaben in der Oberfläche neue Konten hinzugefügt werden können. Die Auflistung wird in einer eigenen Fachklasse verwaltet. Die Schaltflächen geben wichtige Hinweise auf Funktionen, die unsere Fachklasse zur Verwaltung der Auflistung bieten muss: Das Einfügen und Löschen von Elementen der Auflistung. Dieses sind die **Add()**-Methode, die ein als Parameter übergebenes Element am Ende der Liste einfügt, und die **Remove()**-Methode, die ein als Parameter übergebenes Element aus der Liste entfernt. Diese Methoden stehen automatisch zur Verfügung, wenn die erstellte Verwaltungsklasse auf **ArrayLists** oder **typisierten Listen** basiert. Für das Anzeigen aller Konten benötigt man eine Möglichkeit, alle Elemente einer Auflistung zu durchlaufen (**Foreach-Schleife**). Wichtig ist, dass man im Zusammenhang mit Auflistungsklassen immer den Namensraum **System.Collections** mit einbindet, damit man die typisierten Listen und **ArrayLists** nutzen kann.

Die Notwendigkeit zum Erstellen einer neuen Klasse **Kontenliste.cs** ergibt sich dadurch, dass sich viele der Codezeilen mit der Verwaltung unserer Kontoliste beschäftigen. Würden diese alle in der Oberflächenprogrammierung integriert werden, müssten diese beim Austausch der GUI z.B. durch ein Web-Interface gänzlich neu programmiert werden. D.h. Aufgaben der Fachschicht haben sich in der Präsentationsschicht breit gemacht. Dies gilt es zu verhindern. Wir benötigen eine eigene Fachklasse, die Verwaltungsaufgaben der Kontoauflistung beschreibt.

5.5 Vorgehensweise

1. Zunächst ersetzen wir die Ausgabemethode in der Klasse **Konto.cs** durch die überschriebene **ToString()-**Methode (siehe vorhergehende Seite)
2. In einem nächsten Schritt legen wir eine neue Klasse **Kontenliste.cs** an, die die Eigenschaften der Auflistungsklasse **List<Typ>** erbt. Dieser Sachverhalt wird durch folgende Syntax innerhalb der Klassendefinition der Klasse KONTENLISTE dargestellt:

```
public class Kontenliste:List<Konto>
{
}
```

Die neue Klasse **Kontenliste.cs** stellt seinen Objekten automatisch alle Methoden und Properties zur Verfügung, die auch die Klasse **List<Typ>** schon hat.

3. In diesem leeren Zustand können schon einige Funktionalitäten genutzt werden, z.B. das **Einfügen neuer Elemente** in ein Objekt der Klasse Kontenliste funktioniert bereits:

```
20 private void frmKonto_Load(object sender, EventArgs e)
21 {
22     //erster Schritt dient nur der Maximierung des Formulars zur Laufzeit.
23     this.WindowState = FormWindowState.Maximized;
24     //Deklaration und initialisierung mit einer leeren Auflistung
25     eineKontenListe = new Kontenliste();
26     Konto einKonto = new Konto(5005878, "Müller, Hans", 2850.20, 6000.00);
27     //In der Folgezeile erhält die Auflistung das erste Element vom Typ Konto.
28     eineKontenListe.Add(einKonto);
29     // Man muss gar nicht erst ein eigenes Objekt vom Typ Konto anlegen, sondern kann das Erzeugen eines
30     // neuen Kontos direkt mit dem Ergänzen der Liste verbinden:
31     eineKontenListe.Add(new Konto(5005612, "Schmidt, Lolita", 2300.00, 5000.00));
32     eineKontenListe.Add(new Konto(7205822, "Merck, Paul", 2850.20, 0.00));
33 }
34
```

4. Auch das **Anlegen eines neuen Kontos** über die Oberfläche ist bereits möglich, ohne die Klasse zur Auflistung zu erweitern.

```
35 private void btnAnlegen_Click(object sender, EventArgs e)
36 {
37     // Einlesen aller für ein neues Konto benötigten Werte über die Oberfläche
38     // Auf das Abfangen von Eingabefehlern in der Oberfläche wird hier verzichtet.
39     long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
40     string Kontoinhaber = tbxKontoinhaber.Text;
41     double Saldo = Convert.ToDouble(tbxSaldo.Text);
42     double Kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
43     //Anlegen eines neuen Kontos und Ergänzung der Kontenliste (in zwei Schritten)
44     Konto einKonto = new Konto(Kontonummer, Kontoinhaber, Saldo, Kreditlimit);
45     eineKontenListe.Add(einKonto);
46     // Alternativ in einem Schritt:
47     // eineKontenListe.Add(new Konto(Kontonummer, Kontoinhaber, Saldo, Kreditlimit));
48 }
49
```

5. Nun soll es um die **Ausgabe aller Konten** in einem Label gehen, so wie es der Screenshot zu unserer Anwendung vorsieht.

Hierzu müssen alle Objekte der Auflistung durchlaufen werden. Hierbei kann man auf die Methode `ToString()` der Klasse `Konto` zurückgreifen, aber es müssen noch Ergänzungen vorgenommen werden: Es soll vorne noch zusätzlich eine laufende Nummer angezeigt werden, Überschriften müssen ergänzt werden sowie die Statuszeile mit der Anzahl der angezeigten Konten. Dies soll nicht alles in die Oberflächenprogrammierung ausgelagert werden, sondern hier ist es sinnvoll **die Klasse `Kontenliste`** um eine entsprechend angepasste `ToString()`-Methode zur Ausgabe aller Elemente der Auflistung zu erweitern:

```
10 public override string ToString()
11 {
12     string ausgabe = "";
13     string neueZeile = "\n";
14     // nun werden alle Objekte der Auflistung durchlaufen...
15     foreach (Konto konto in this)
16     {
17         //... und ausgegeben
18         ausgabe += String.Format("{0,-5} {1}", this.IndexOf(konto) + 1, konto.ToString());
19     }
20     ausgabe += neueZeile + "-----" + neueZeile;
21     //Hier wird die Ausgabe um eine Statuszeile ergänzt.
22     ausgabe += "Anzahl der Konten: " + this.Count.ToString();
23     return ausgabe;
24 }
25
```

Neue Methode der Klasse `Kontenliste`

In Zeile 10 (Abbildung auf S. 37) beginnt die Methode **`ToString()`**, die eine **`foreach-Schleife`** enthält. Diese wird zu einem späteren Zeitpunkt noch näher erläutert.

Nur so viel an dieser Stelle: es werden mittels der Schleife (Zeile 15) alle Elemente der Liste durchlaufen. Jedes Objekt der Kontenliste wird nacheinander in der Variablen **`Konto`** erfasst. Anschließend werden die Anweisungen im Schleifenrumpf (Zeile 16 – 19) ausgeführt. In Zeile 18 wird die laufende Nummer angezeigt. Diese könnte man selbst bilden, in dem man einen Schleifenzähler einbaut, der bei jedem Durchlauf um 1 erhöht wird. In dem oben dargestellten Code wird die Methode **`IndexOf()`** von Auflistungen verwendet, die zu dem übergebenen Objekt die Position (des ersten Vorkommens) in der Liste angibt. Da in C# die Zählung immer mit 0 beginnt, wird zur Position jeweils noch 1 addiert.

Die Ausgabe wird in Zeile 19 durch die Statuszeile erweitert, die die Anzahl der Konten der Auflistung angibt. Hierzu wird die Property **`Count`** verwendet, die standardmäßig für jede Auflistung die Anzahl der enthaltenen Elemente angibt.

Ist diese Methode nun fertig geschrieben, ist die Button-Programmierung der Schaltfläche „Alle Anzeigen“ nun nur noch ein „Einzeiler“:

```
50 private void btnAnzeigen_Click(object sender, EventArgs e)
51 {
52     lblAusgabe.Text = eineKontenListe.ToString();
53 }
54
```

6. Nun fehlt zur Vervollständigung der Beispielsanwendung, noch das Löschen eines Kontos. Die Kontonummer wird in der zugehörigen Textbox eingetragen und nach Klicken der Schaltfläche „Konto löschen“ wird das zugehörige Konto gelöscht. Für diesen Schritt fehlt noch eine Methode in der Klasse `Kontenliste` zum Auffinden des zugehörigen Kontos.

```

40 public int sucheKontoIndex(long pKontonummer)
41 {
42     int index = 0;
43     foreach (Konto einKonto in this)
44     {
45         if (einKonto.Kontonummer == pKontonummer)
46         {
47             return index;
48         }
49         index++;
50     }
51     throw new Exception("Konto nicht vorhanden!");
52 }
53

```

Weitere Methode der Klasse Kontenliste.

Da diese Methode einen Fehler wirft, falls kein Konto zu der angegebenen Kontonummer vorhanden ist, sollte man den Methodenaufruf in der Oberflächenschicht in einen try-catch-Block integrieren.

```

91 private void btnLöschen_Click(object sender, EventArgs e)
92 {
93     try
94     {
95         long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
96         int index = eineKontenListe.sucheKontoIndex(Kontonummer);
97         eineKontenListe.RemoveAt(index);
98         btnAnzeigen_Click(this, e);
99     }
100     catch (Exception ex)
101     {
102         lblAusgabe.Text = ex.ToString();
103     }
104 }

```

Aktualisierung der Anzeige!

In Zeile 96 wird die oben erstellte Methode `sucheKontoIndex()` genutzt, um den Index zu finden. Mit der Standardmethode `RemoveAt()` wird der Listeneintrag für den Index gelöscht, der dieser Methode übergeben wird.

In Zeile 98 wird deutlich, dass `Button_Click`-Methoden nicht nur durch ein Klicken des Anwenders auf der Oberfläche ausgelöst werden können, sondern dass man die Ausführung dieser Routine auch im Programm erzwingen kann.

- Die zuletzt hinzugefügte Methode `SucheKontoIndex()` ist ebenfalls nützlich, um den Button für Ein- und Auszahlen zu implementieren. Denn man kann auf einzelne Elemente einer Auflistung über einen **Indexer** zugreifen. Hierzu gibt man hinter dem Bezeichner der Auflistung die Position (den Index) desjenigen Elements an, auf das man zugreifen möchte. Der Index ist in C# immer nullbasiert, d.h. das erste Element einer Auflistung hat immer den Index 0.

```

63 private void btnEinzahlen_Click(object sender, EventArgs e)
64 {
65     double Betrag = Convert.ToDouble(tbxBetrag.Text);
66     long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
67     int index = eineKontenListe.sucheKontoIndex(Kontonummer);
68     eineKontenListe[index].Einzahlen(Betrag);
69     btnAnzeigen_Click(this, e);
70 }

```

Analog für das Auszahlen:

```
72 private void btnAuszahlen_Click(object sender, EventArgs e)
73 {
74     try
75     {
76         double Betrag = Convert.ToDouble(tbxBetrag.Text);
77         long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
78         int index = eineKontenListe.sucheKontoIndex(Kontonummer);
79         eineKontenListe[index].Auszahlen(Betrag);
80     }
81     catch (Exception ex)
82     {
83         MessageBox.Show("Fehler: " + ex.Message);
84     }
85     finally
86     {
87         btnAnzeigen_Click(this, e);
88     }
89 }
```

8. Nun fehlt zur Vervollständigung der Anwendung lediglich das Leeren der Felder. Hierzu gibt es die Möglichkeit jede Textbox anzusprechen und diese zu leeren, so wie in folgendem Code-Snippet angegeben:

```
private void btnLeeren_Click(object sender, EventArgs e)
{
    tbxKontonummer.Clear();
    tbxKontoinhaber.Clear();
    tbxSaldo.Clear();
    tbxKreditlimit.Clear();
}
```

Dies ist für vier Textboxen noch schnell notiert. Man kann sich aber auch große Erfassungsmasken mit mehr als zehn Eingabefeldern vorstellen. Hier ist es dann nützlich zu wissen, dass sämtliche Oberflächenelemente ebenfalls in die zu einem Formular gehörende Objektsammlung Controls gehören. Diese lassen sich dann wieder mit einer foreach-Schleife durchlaufen. Falls das jeweilige Control eine Textbox ist, wird diese geleert. Also erhalten wir folgende Alternativlösung zu dem oben angegebenen Code-Snippet:

```
private void btnLeeren_Click(object sender, EventArgs e)
{
    foreach (Control c in this.Controls)
    {
        if (c is TextBox)
        {
            ((TextBox)c).Clear();
        }
    }
}
```

Boolsche Prüfung: Ist das Control vom Typ TextBox?

Die Objekte die Textboxen sind, werden von dem allgemeineren Datentyp Control auf den Datentyp TextBox konvertiert.

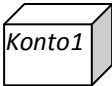
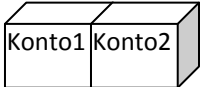
9. Nun ist die Anwendung fertig. Eine Optimierung fehlt allerdings noch. Die Add-Methode arbeitet noch nicht optimal. Denn zurzeit wäre es möglich zwei Konten mit derselben Kontonummer einzufügen.
Diesen Punkt wird bei der Ausweitung der hier behandelten Beispielanwendung noch einmal behandelt.

5.6 Die Objekte der Auflistung ausgeben

Der Zugriff auf alle Elemente einer Liste kann auf unterschiedliche Weise erfolgen. Sehr praktisch ist hier der Einsatz von Schleifen. Besonders praktisch ist die Verwendung der foreach-Schleife.

Ausgabe einzelner Konten der Auflistung

Die Elemente in einer Auflistung haben eine implizite Nummerierung (Indexer) oder Position, die mit null beginnt. Die Position eines Elementes in einer Sammlung wird üblicherweise sein Index genannt. Das erste eingefügte Element in einer Sammlung hat den Index 0, das zweite den Index 1 und so weiter.

Syntax	Beschreibung	Ergebnis
<code>Kontenliste</code> eineKontenListe; <code>eineKontenListe = new Kontenliste();</code>	Erstellt eine (leere) typisierte Auflistung	
<code>Konto</code> einKonto = <code>new Konto();</code> <code>eineKontenListe.Add(einKonto);</code>	Fügt ein Element hinzu	
<code>eineKontenListe.Add(einKonto = new Konto());</code>	Fügt ein weiteres Element hinzu	
<code>eineKontenListe.Add(einKonto = new Konto());</code>	Fügt ein weiteres Element hinzu	

Beispiel:

<i>eineKontenListe</i>	
Index	Listenelement
0	KontoMueller
1	KontoSchmidt
2	KontoMerck
...	...
...	...

Zur manuellen Ausgabe müssen die einzelnen Objekte über den entsprechenden Index ausgelesen werden.

Syntax:

```
KLASSE klassenobjekt = new Klasse();

klassenobjekt = liste[0];
MessageBox.Show(klassenobjekt.ToString());

klassenobjekt = liste[1];
MessageBox.Show(klassenobjekt.ToString());

klassenobjekt = liste[2];
MessageBox.Show(klassenobjekt.ToString());
```

Beispiel:

```
Konto einKonto = new Konto()
einKonto = eineKontenListe[0];
MessageBox.Show(einKonto.ToString());
```

liefert:



5.7 Durchlaufen von Listen mit Hilfe der foreach-Schleife

Die foreach-Anweisung vereinfacht das Durchlaufen von Arrays, ArrayLists und anderen Auflistungen. Sie ist die Standardtechnik, wenn alle Elemente einer Sammlung durchlaufen werden sollen. Sie ist leicht zu schreiben¹³ und sicher, weil jederzeit garantiert ist, dass sie zu einem Ende kommt.

Syntax:

```

      Deklaration der      Collection-
      Iterationsvariablen   name
      {
foreach (Typ typ in Auflistung)
{
    Anweisung1;
    Anweisung2;
    Usw.
}

```

¹³ Sie benötigt z.B. keinen Schleifenindex

Beispiel:

```
10 public override string ToString()
11 {
12     string ausgabe = "";
13     string neueZeile = "\n";
14     // nun werden alle Objekte der Auflistung durchlaufen...
15     foreach (Konto konto in this)
16     {
17         //... und ausgegeben
18         ausgabe += String.Format("{0,-5} {1}", this.IndexOf(konto) + 1, konto.ToString());
19     }
20     ausgabe += neueZeile + "-----" + neueZeile;
21     //Hier wird die Ausgabe um eine Statuszeile ergänzt.
22     ausgabe += "Anzahl der Konten: " + this.Count.ToString();
23     return ausgabe;
24 }
25
```

Erläuterung der foreach-Schleife (Zeile 15-19):

Für jedes Objekt **in** der Kontoliste (gekennzeichnet durch den **this-Operator**) weise das aktuelle Element der Auflistung der Variablen **Konto** zu und führe die Anweisungen im Schleifenrumpf aus. Allgemein muss eine Iterationsvariable deklariert und initialisiert werden, die ein einzelnes Element aus einer Collection aufnimmt. Dabei ist darauf zu achten, dass der Datentyp der Iterationsvariablen mit dem Typ der Elemente der Collection kompatibel sein muss (hier: **Konto**). Danach muss eine Referenz auf die Collection (hier: **this**) erfolgen.

Allgemein ist der Einsatz einer foreach-Schleife für sämtliche Aufzählungen möglich, die die Schnittstelle **IEnumerator** implementieren. Intern wandelt der Compiler foreach-Schleifen in eine while-Schleife um.

5.8 Erweiterung des Konto-Beispiels

Das Konto-Beispiel soll an dieser Stelle um einige Funktionen erweitert werden. So soll es dem Benutzer ermöglicht werden nach einer bestimmten Kontonummer zu suchen und sich die zugehörigen Kontoinformationen anzeigen zu lassen. Weiterhin soll die Möglichkeit gegeben werden die Konten z.B. alphabetisch nach dem Namen des Kontoinhabers zu sortieren. Außerdem soll die Kontonummer einer Kontenauflistung eindeutig sein.

The screenshot shows a window titled "Sparkasse Kleve". It contains several input fields and buttons. On the left, there are labels and input boxes for "Kontonummer" (5005878), "Kontoinhaber", "Saldo", and "Kreditlimit". On the right, there is a section titled "Ein-/Auszahlungen" with a "Betrag" input box and "Einzahlen" and "Auszahlen" buttons. Below these are six buttons: "Konto anlegen", "Alle Konten anzeigen", "Felder leeren", "Konto zu Kontonummer löschen", "Konto zu Kontonummer suchen" (highlighted with a blue border), and "Alle Konten sortiert anzeigen". At the bottom, there is a text area displaying account details: "Kontonummer: 5005878", "Kontoinhaber: Müller, Hans", "Saldo: 2.850,20 €", and "Kreditlimit: 6.000,00 €", followed by a dashed line.

5.8.1 Konto anlegen, aber nur falls Kontonummer nicht bereits vorhanden

Für den Vergleich zweier Objekte (hier zweier Konten) muss man zunächst definieren, woran die Gleichheit zweier Objekte festgemacht wird. Da jede Kontonummer in der Anwendung eindeutig ist, sind zwei Konten genau dann gleich, wenn ihre Kontonummern gleich sind.

Es wurde bereits erwähnt, dass jede Klasse implizit vom Typ `object` abstammt. D.h. jede Klasse verfügt auch bereits über die Methode `Equals(object obj)`. Analog zu der Methode `ToString()` sollte diese Methode passend zur jeweiligen Klasse überschrieben werden. Im Folgenden ist eine sinnvolle Implementierung dieser überschriebenen Methode abgebildet:

```
public override bool Equals(Object obj)
{
    if (obj == null) return false;

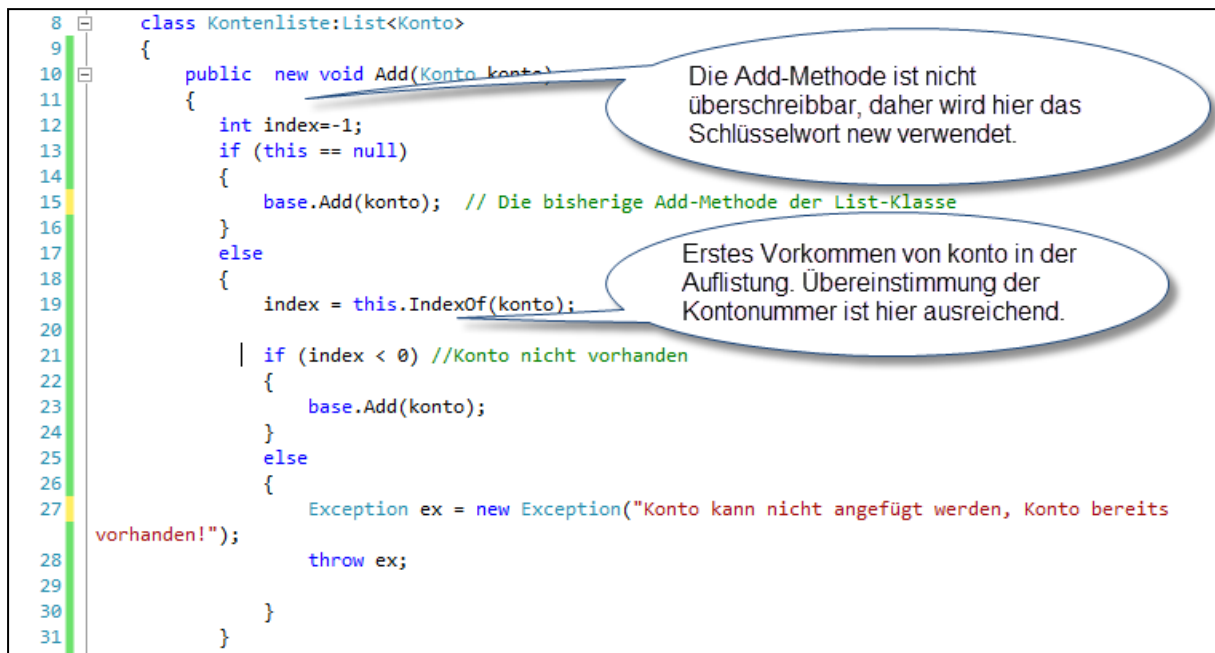
    if (this.GetType() != obj.GetType()) return false;

    // Folgende Umwandlung ist sicher, da Typgleichheit überprüft!
    Konto Konto = (Konto)obj;

    // Zwei Konten sind dann gleich, wenn ihre Kontonummer
    übereinstimmt!
    if (!Object.Equals(Kontonummer, Konto.Kontonummer)) return false;
    //Alternativ: Kontonummer!=Konto.Kontonummer

    return true;
}
```

Nun lässt sich beim Einfügen neuer Konten in die Kontenliste die Existenz mit Hilfe der `IndexOf()`-Methode von Auflistungen prüfen. Hierzu wird die `Add`-Methode der typisierten `List`-Klasse überschrieben:



Der folgende Test zeigt die Wirkung der beiden neu hinzugefügten Methoden. In der Buttonprogrammierung wurde lediglich die Add-Methode in einen try-catch-Block eingebettet:

```

private void btnAnlegen_Click(object sender, EventArgs e)
{
    // Einlesen aller für ein neues Konto benötigten Werte über die
    // Oberfläche
    // Auf das Abfangen von Eingabefehlern in der Oberfläche wird hier
    // verzichtet.
    long Kontonummer = Convert.ToInt64(tbxKontonummer.Text);
    string Kontoinhaber = tbxKontoinhaber.Text;
    double Saldo = Convert.ToDouble(tbxSaldo.Text);
    double Kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
    //Anlegen eines neuen Kontos und Ergänzung der Kontenliste (in zwei
    //Schritten)
    Konto einKonto = new Konto(Kontonummer, Kontoinhaber, Saldo,
    Kreditlimit);
    try
    {
        eineKontenliste.Add(einKonto);
    }
    catch (Exception ex)
    {
        MessageBox.Show(ex.Message.ToString());
    }
}

```

5.8.2 Elemente einer Auflistung suchen

Eine Auflistung lässt sich nach einem bestimmten Element durchsuchen, indem man in die Schleifen zum Durchlaufen der Sammlung eine Bedingung einbaut, die den Durchlauf stoppt, wenn das Element gefunden wurde.

Beispiel:

Bei Eingabe einer Kontonummer soll es möglich sein, die Daten des entsprechenden Kontos angezeigt zu bekommen. Ist die Kontonummer nicht vorhanden, soll eine entsprechende Fehlermeldung ausgegeben werden.

The screenshot shows the 'Sparkasse Kleve' web application interface. On the left, there are input fields for 'Kontonummer' (containing '5005612'), 'Kontoinhaber', 'Saldo', and 'Kreditlimit'. On the right, there is a section titled 'Ein-/Auszahlungen' with a 'Betrag' input field and 'Einzahlen' and 'Auszahlen' buttons. Below these are six buttons: 'Konto anlegen', 'Alle Konten anzeigen', 'Felder leeren', 'Konto zu Kontonummer löschen', 'Konto zu Kontonummer suchen' (highlighted with a blue border), and 'Alle Konten sortiert anzeigen'. At the bottom, a table displays account details:

Kontonummer	Kontoinhaber	Saldo	Kreditlimit
5005612	Schmidt, Lolita	2300,00€	5000,00€

Oder Anzeige, falls Kontonummer nicht vorhanden:

The screenshot shows the same 'Sparkasse Kleve' web application interface, but with the 'Kontonummer' field containing '2012'. The 'Konto zu Kontonummer suchen' button remains highlighted. At the bottom of the interface, a message reads 'Konto nicht vorhanden!'.

Natürlich könnte man wie beim Hinzufügen von Konten hier die `IndexOf()`-Methode nutzen. Im Folgenden soll aber eine eigene Methode geschrieben werden, da dieses Prinzip sich dann auch auf andere Felder (z.B. Kontoinhaber) übertragen lässt.

Der Quellcode der Methode muss der Klasse **Kontenliste.cs** hinzugefügt werden.

```
66 public int SucheKontoIndex(long pKontonummer)
67 {
68     int index = 0;
69     foreach (Konto einKonto in this)
70     {
71         if (einKonto.Kontonummer == pKontonummer)
72         {
73             return index;
74         }
75         index++;
76     }
77     throw new Exception("Konto nicht vorhanden!");
78 }
```

Die Methode **SucheKontoIndex()** durchsucht die Kontenliste nach der gewünschten Kontonummer und gibt die Position (den Index) in der Liste zurück. Ein Fehler wird geworfen, falls noch kein Konto mit der vorgegebenen Kontonummer in der Liste vorhanden ist.

In den Quellcode des Button_Click-Events **btnKontosuchen_Click** gelangt man wiederum durch einen Doppelklick auf den Button **Konto zu Kontonummer suchen** im Entwurf des Formulars.

Folgende Codezeilen sind in der Button_Click-Methode zu ergänzen:

```
122 private void btnKontoSuchen_Click(object sender, EventArgs e)
123 {
124     try
125     {
126         long kontonr = Convert.ToInt64(tbxKontonummer.Text);
127
128         Konto einKonto = eineKontenListe[eineKontenListe.SucheKontoIndex(kontonr)];
129         lblHeadline.Text = "Kontonummer   Kontoinhaber   Saldo   Kreditlimit ";
130         lblAusgabe.Text = einKonto.ToString();
131     }
132     catch (Exception ex)
133     {
134         MessageBox.Show(ex.Message);
135     }
136 }
137 }
```

In Zeile 128 erfolgt der Methodenaufruf der Methode **SucheKontoIndex()**. Wird ein übereinstimmendes Konto in der Liste gefunden, werden die Detailinformationen angezeigt. Andernfalls wird der Catch-Block ausgeführt und die Fehlermeldung „Konto nicht vorhanden“ in einer MessageBox angezeigt.

5.8.3 Sortieren einer Auflistung

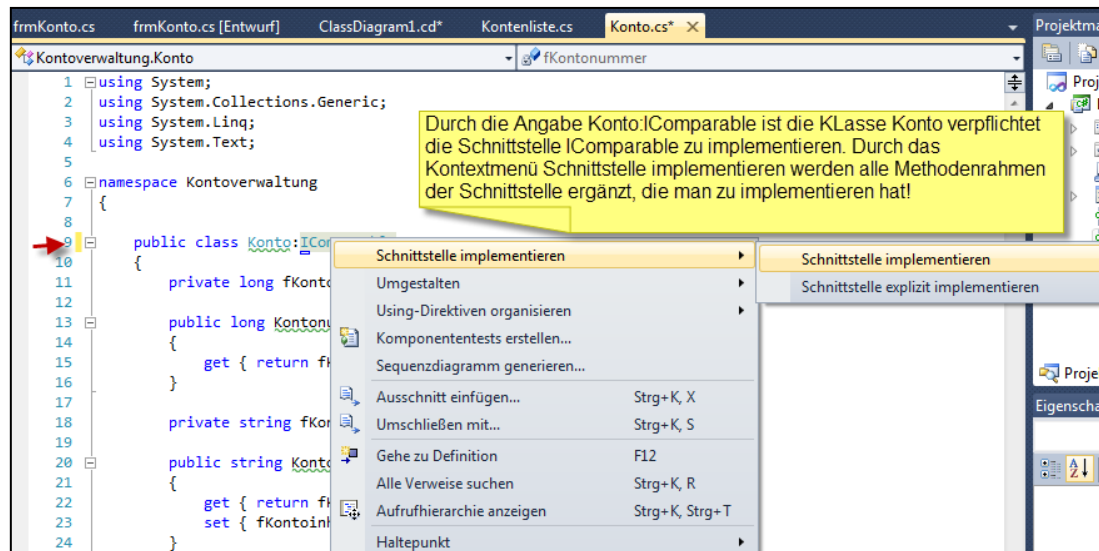
Die Daten einer Auflistung werden in der Reihenfolge ausgegeben, in der sie zur Auflistung hinzugefügt wurden, d. h. nicht alphabetisch. Zum Sortieren von Auflistungen steht die Methode **sort()** zur Verfügung. Allerdings muss für den Typ der Auflistung eine Sortierung definiert sein.

Was heißt, dass ein Konto größer als ein anderes Konto ist?

Das muss für selbstgeschriebene Typen erst definiert werden. Zu diesem Zweck kann man das Interface **IComparable** in der Fachklasse (in diesem Fall **Konto.cs**) implementieren. Interfaces enthalten nur die Signaturen von Methoden und Properties. Eine Klasse die ein Interface implementiert, muss alle Methoden des Interfaces „ausprogrammieren“.

Syntax: **class Klassenname: Interfacename**

Die obige Syntax beschreibt die Vereinbarung, dass die definierte Klasse, das hinter dem Doppelpunkt angegebene Interface implementieren muss. Markiert man den Interfacenamen und wählt im Kontextmenü den Eintrag „Schnittstelle implementieren“ so werden die zu implementierenden Methoden direkt eingefügt.



Ein großer Vorteil von Interfaces ist, dass eine Klasse mehrere Interfaces implementieren kann, aber nur von einer Klasse erben kann.

Im Kontobeispiel wird bei der Angabe von

`class Konto:IComparable`

und Klicken des Kontextmenüs „Schnittstelle implementieren“ in der Klasse `Konto` folgender Code angelegt:

```
public int CompareTo(Konto other)
{
    throw new NotImplementedException();
}
```

In der Regel wird diese Methode implementiert, in dem man auf schon bestehende `CompareTo`-Methoden der Standarddatentypen nutzt. Soll alphabetisch nach dem Kontoinhaber sortiert werden, so kann die Methode wie folgt implementiert werden:

```
public int CompareTo(Konto anderesKonto)
{
    return anderesKonto.Kontoinhaber.CompareTo(this.Kontoinhaber);
}
```

Die **sort()**-Methode steht allen Auflistungsklassen zur Verfügung und greift auf die **compareTo()**-Methode zu, die in der Fachklasse implementiert sein muss. Die **compareTo()**-Methode überprüft die Elemente einer Liste auf ihre Ordnung. Diese Sort-Methode kann man nun bei der Button-Programmierung des Buttons „Alle Konten sortiert anzeigen“ verwenden.

```
private void btnSortierenKontoinhaber_Click(object sender, EventArgs e)
{
    eineKontenListe.Sort();
    btnAnzeigen_Click(this, e);
}
```

Test der Sortierung:

The screenshot shows a Java Swing window titled "Sparkasse Kleve". It contains several input fields for account details: "Kontonummer", "Kontoinhaber", "Saldo", and "Kreditlimit". To the right, there is a sub-window titled "Ein-/Auszahlungen" with a "Betrag" field and "Einzahlen" and "Auszahlen" buttons. Below these are six buttons: "Konto anlegen", "Alle Konten anzeigen", "Felder leeren", "Konto zu Kontonummer löschen", "Konto zu Kontonummer suchen", and "Alle Konten sortiert anzeigen" (which is highlighted with a blue border). At the bottom, there is a table with 5 columns: "Nr.", "Kontonummer", "Kontoinhaber", "Saldo", and "Kreditlimit". The table contains 3 rows of data. Below the table, it says "Anzahl der Konten: 3".

Nr.	Kontonummer	Kontoinhaber	Saldo	Kreditlimit
1	7205822	Merck, Paul	2850,20€	0,00€
2	5005878	Müller, Hans	2850,20€	6000,00€
3	5005612	Schmidt, Lolita	2300,00€	5000,00€

Anzahl der Konten: 3

Es ist auch möglich weitere Sortierungen z.B. nach Kontonummern zu berücksichtigen. Hierzu muss in einer eigenen Klasse die `Comparable`-Schnittstelle implementiert werden. Dieses Interface besitzt lediglich die Methode `Compare`, der zu vergleichende Objekte übergeben werden.

Zunächst finden Sie aber noch eine Zusammenfassung der wesentlichen Methoden und Properties der Auflistungstypen `ArrayList` und `List`<>.

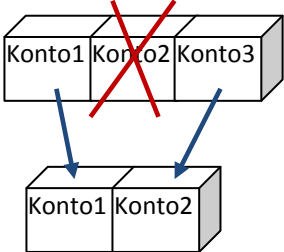
5.9 Methoden und Eigenschaften der Klassen ArrayList und List<Typ>

Folgende Methoden und Eigenschaften der ArrayList sind das wichtigste Handwerkszeug, um mit der Datenstruktur effektiv arbeiten zu können:

Handwerkszeug	Welche Aufgabe?	Wie nutzt man die Methode (Beispiele)?	Besonderheiten für generische Auflistung List<Typ>
Konstruktor	Anlegen und initialisieren einer neuen Aufzählung	<pre>ArrayList Kontoliste; Kontoliste = new ArrayList();</pre>	<pre>List<Konto> Kontoliste; Kontoliste = new List<Konto>();</pre>
Add (Methode)	Hinzufügen neuer Listeneinträge am Ende der Liste	Methodenaufruf: <pre>Add(Object pObj),</pre> als Parameter wird der neue Listeneintrag übergeben. <pre>Konto Konto1= new Konto(); Kontoliste.Add(Konto1);</pre>	Beim Hinzufügen wird Typsicherheit geprüft. <pre>Add(Typ pObj)</pre>
Indexer	Zugriff auf ein bestimmtes Element der Liste	An den Namen der ArrayList wird in eckigen Klammern die Position des gesuchten Elementes angegeben. Die Zählung beginnt bei 0. Fehler, falls Index größer als Anzahl der Elemente -1. <pre>MessageBox.Show(((Konto) Kontoliste[0]). ToString());</pre> (Cast hier erforderlich!)	Analog Hier: Kein Cast erforderlich: <pre>MessageBox.Show((Kontoliste[0]. ToString())</pre>
Count (Property)	Gibt die Anzahl der Elemente in der Liste wieder.	<pre>Kontoliste.Count();</pre> gibt den Wert 3 zurück, falls drei Konten in der Auflistung sind.	analog
Insert (methode)	Fügt ein Element an einer bestimmten Position der Liste ein.	Methodenaufruf: <pre>Insert(int pIndex, Object pObj)</pre> Aufruf der Methode Insert mit den Parametern pIndex (Wo soll das Element hinterher stehen?) und dem einzufügenden Wert. <pre>Kontoliste.Insert (1,meinKonto);</pre> fügt das Objekt meinKonto an die 2. Position der Liste zwischen Konto1 und Konto2 ein.	Methodenaufruf: <pre>Insert(int pIndex, Typ pObj)</pre> Beispiel analog.

Handwerkszeug	Welche Aufgabe?	Wie nutzt man die Methode (Beispiele)?	Besonderheiten für generische Auflistung List<Typ>
foreach-Schleife	Mit dieser Schleife können alle Einträge einer Liste problemlos durchlaufen werden, um z.B. alle Elemente aus zugeben.	<pre>foreach ((Konto) Konto in Kontoliste) { Ausgabe+=Konto.ToString(); }</pre>	<pre>foreach (Konto Konto in Kontoliste) { Ausgabe+=Konto.ToString(); }</pre>
RemoveAt Methode	Löscht das Listenelement an der angegebenen Position	Methodenaufruf: RemoveAt(int pIndex)	analog
Remove (Methode)	Löscht das erste Vorkommen desjenigen Listenelements mit dem Übereinstimmung mit dem übergebenen Parameter festgestellt wird.	Methodenaufruf: Remove(Object pObj)	Methodenaufruf: Remove(Typ pObj)
Clear (Methode)	Löscht alle Elemente einer Auflistung	Methodenaufruf: Clear() Beispiel: Kontoliste.Clear();	Analog
Contains (Property)	Prüft ob sich ein übergebenes Objekt in der Auflistung befindet.	Methodenaufruf Contains(Object item); gibt den Wert true zurück, falls der übergebene Wert in der Auflistung enthalten ist.	Methodenaufruf Contains(Typ item); Beispiel: Kontoliste.Contains(meinKonto)
Sort (Methode)	Sortiert die Elemente der Auflistung. Die Methode ist überladen. Ohne Parameter wird die Sortierung durch die im Typ (Klasse) vorhandene Implementierung der Schnittstelle IComparable vorgegeben. Sie können aber auch zusätzliche Sortierungen definieren, in dem Sie die Schnittstelle IComparer in einer eigenen Klasse implementieren.	Methodenaufruf: Sort() oder Sort(new ElementComparer()) Beispiel: <pre>SortByKontoInhaber icomparer = new SortByKontoInhaber(); Kontoliste.Sort(icomparer);</pre>	Analog
IndexOf (Methode)	Gibt den Index des ersten Vorkommens zurück	int i =IndexOf(meinKonto)	Basiert auf der Implementierung von Equals()

Veranschaulichung:

Syntax	Beschreibung	Ergebnis
<code>Kontoliste.add(Konto1);</code> <code>Kontoliste.add(Konto2);</code> <code>Kontoliste.add(Konto3);</code>		
<code>Kontoliste.remove(Konto2);</code> oder <code>Kontoliste.remove(1);</code>		

6. Vererbung

Ausgangsbeispiel: Kontoverwaltung für die Sparkasse Kleve

Situationsbeschreibung

Der Leiter der Kundenbetreuung weist darauf hin, dass die Konditionen der unterschiedlichen Kontenmodelle wie z.B. bei einem Sparkonto auf der einen Seite oder einem Girokonto auf der anderen Seite bei der jetzigen Lösung nicht angemessen berücksichtigt werden.

Der Leiter der DV-Abteilung verspricht dem Leiter der Kundenbetreuung eine Lösung und beauftragt einen Informatiker mit der Analyse sowie der programmtechnischen Umsetzung des Problems. Der Informatiker analysiert die Ausgangsgrundlage auf der Basis der ihm zur Verfügung gestellten Informationen zu den Kontenmodellen Sparkonto sowie Girokonto und erstellt ein Pflichtenheft.

Sparkasse Kleve

Sparkonto - Zinsen ab dem ersten Euro

Ihr Geld ist täglich für Sie verfügbar. Zuzahlungen sind jederzeit möglich und schon ab dem ersten Euro kassieren Sie attraktive Zinsen. Auszahlungen können jederzeit bis auf 0,00 EUR vorgenommen werden.

Die Zinsen werden dem Sparkonto am Ende eines Kalenderjahres gutgeschrieben. Kontoauszüge sowie Zinsbescheinigungen werden ebenfalls nur einmal jährlich am Ende eines Kalenderjahres erstellt. Die derzeitigen Zinsen betragen 0,5 % p. a.

Sparkasse Kleve

Girokonto

Für Ihre regelmäßigen Transaktionen wie Mietzahlungen oder Gehaltszahlungen lohnt sich unser Girokonto. Ihr Geld ist täglich für Sie verfügbar. Zuzahlungen sind jederzeit möglich. Ein Dispokredit kann Ihnen nach Rücksprache gewährt werden. Auszahlungen sind in diesem vereinbarten Rahmen jederzeit möglich. Die anfallenden Überziehungszinsen betragen 12,5 % p. a. vom in Anspruch genommenen Dispokredit und werden dem Girokonto vierteljährlich belastet. Gebühren werden nicht berechnet. Kontoauszüge können jederzeit erstellt werden.

6.1 Pflichtenheft: Kontoverwaltung Sparkasse Kleve

Zinsgutschriften bzw. Zinsbelastungen werden als Einzahlungen bzw. Auszahlungen behandelt. Der Zinszahlungszeitpunkt soll vorläufig noch unberücksichtigt bleiben.

Programmdaten

Folgende Daten sollen über jedes Konto ohne weitere Angabe des Kontenmodells gespeichert werden:

/D10/ Kontoinhaber

/D40/ Kontonummer

/D50/ Kontensaldo

Folgende Daten sollen über jedes **Sparkonto** gespeichert werden:

/D10/ Kontoinhaber

/D40/ Kontonummer

/D50/ Kontensaldo

Folgende Daten sollen über jedes **Girokonto** gespeichert werden:

/D10/ Kontoinhaber

/D40/ Kontonummer

/D50/ Kontensaldo

/D60/ Kreditlimit in €

Programmfunktionen

Für Giro- und Sparkonten sind folgende Operationen möglich:

/F10/ Es sollen lediglich Objekte vom Typ Sparkonto bzw. Girokonto erzeugt werden können. Es gibt keine Objekte zum Typ Konto.

/F21/ Bei der Erzeugung eines Sparkontos sind die Kontonummer und der Kontoinhaber sofort zu setzen. Für Sparkonten werden 7-stellige Nummern verwendet, die mit 2,3 oder 4 beginnen. Für Girokonten werden ebenfalls 7-stellige Nummern verwendet, die mit 5,6 oder 7 beginnen.

/F22/ Bei der Erzeugung eines Girokontos sind die Kontonummer sowie das Kreditlimit sofort zu setzen.

/F30/ Bei einer Konteneröffnung soll eine sofortige Bankeinzahlung möglich sein.

/F32/ Bei der Verwaltung aller Konten ist sicherzustellen, dass keine Kontonummer doppelt vergeben wird.

/F40/ Alle Daten - bis auf die Kontonummer und Kontostand - müssen einzeln geschrieben werden können.

/F50/ Die Daten der Instanzvariablen müssen einzeln ausgelesen werden können.

/F60/ Sowohl auf einem Sparkonto als auch auf einem Girokonto sind Einzahlungen, Auszahlungen und Überweisungen möglich.

/F62/ Für Spar- und Girokonten steht die Methode **Einzahlen()** zur Verfügung. Für beide Kontentypen gibt es beim Einzahlen keinen Unterschied. Der aktuelle Kontostand erhöht sich um den eingezahlten Betrag.

/F64/ Für Sparkonten steht die Methode **Auszahlen()** zur Verfügung. Der Kontostand wird um den auszahlenden Betrag verringert, aber nur dann wenn das Mindestguthaben von 1 EUR nicht unterschritten wird. Andernfalls wird die Auszahlung nicht durchgeführt, die Methode gibt einen Fehler mit entsprechender Fehlermeldung zurück.

/F66/ Für Girokonten steht die Methode **Auszahlen()** zur Verfügung. Der Kontostand wird um den auszahlenden Betrag verringert, aber nur dann wenn das Kreditlimit hierbei nicht überschritten wird. Andernfalls wird die Auszahlung nicht durchgeführt, die Methode gibt einen Fehler mit entsprechender Fehlermeldung zurück.

/F68/ Die Methode **Überweisen()** steht sowohl Girokonten als auch Sparkonten zur Verfügung. Hierbei handelt es sich um eine Transaktion¹⁴ der Hintereinanderausführung der Methoden Auszahlen() vom Senderkonto und Einzahlen() auf das Empfängerkonto jeweils mit demselben Betrag. Ist das Auszahlen() nicht möglich (vgl. F64 und F66), dann wird auch das Einzahlen() nicht durchgeführt!

Folgende Operationen bietet die Benutzerschnittstelle (das Formular zur Verwaltung der Konten):

/F70/ Alle Kontenobjekte werden mit allen Werten der Instanzvariablen bereits beim Laden des Formulars tabellarisch angezeigt.

/F72/ Bei einer Einzahlung wird auch die Anzeige automatisch aktualisiert (der neue Kontostand wird angezeigt).

/F74/ Bei einer Auszahlung wird auch die Anzeige automatisch aktualisiert (der neue Kontostand wird angezeigt).

/F80/ Die Kontenansicht kann auf Girokonten bzw. Sparkonten eingeschränkt werden. Die Eingeschränkte Sicht kann wieder auf die Sicht aller Konten erweitert werden.

/F90/ Über das Formular kann ein neues Konto erfasst werden. Dieses Konto wird automatisiert in der Kontoaufstellung angezeigt.

¹⁴ Eine Transaktion sind zusammengehörende Arbeitsschritte, die entweder ganz oder gar nicht durchgeführt werden. Bei einem Fehler wird der Zustand der beteiligten Objekte wieder in den Zustand vor Ausführung der Transaktion versetzt.

Testfälle

/T10/Folgende Sparkonto-Objekte sind zu erzeugen und mit den entsprechenden Werten zu belegen.

(Die ersten beiden Konten sind bereits beim Aufruf zugewiesen, das dritte wird über die Oberfläche erzeugt)

Objekte/ Instanzvariablen	<i>SparkontoMueller</i>	<i>SparkontoSchmidt</i>	<i>SparkontoMeier</i>
Kontonummer	3103901	3004813	3200450
Kontoinhaber	Hans Müller	Heinz Schmidt	Sabine Meier
Einzahlung bei Konteneröffnung	2000,00 €	5000,00 €	1,00 €
=Saldo	2000,00 €	5000,00 €	1,00 €

/T20/Folgende Girokonto-Objekte sind zu erzeugen und mit den entsprechenden Werten zu belegen.

(Die ersten beiden Konten sind bereits beim Aufruf zugewiesen, das dritte wird über die Oberfläche erzeugt)

Objekte/ Instanzvariablen	<i>GirokontoMueller</i>	<i>GirokontoSchmidt</i>	<i>GirokontoJansen</i>
Kontonummer	5005875	5005612	7205820
Kontoinhaber	Hans Müller	Lola Schmidt	Gregor Jansen
Kreditlimit	6000,00 €	5000,00 €	0,00 €
Einzahlung bei Konteneröffnung	1000,00 €	2000,00 €	3000,00 €
=Saldo	1000,00 €	2000,00 €	3000,00 €

/T30/Folgende Kontobewegungen sind mit Hilfe des Programms durchzuführen und zu überprüfen:

Datum	Art der Konto- bewegung	Konto- nummer	Betrag	ZielkontoNr	Ergebnis
15.01.2012, 9:05:45	Einzahlung	5005875	720,00 €	-	Saldo= 1720,00 €
15.01.2012, 9:15:05	Auszahlung	7205820	1800,00€	-	Saldo= 1200,00 €
16.01.2012, 13:25:05	Auszahlung	7205820	1950,00€	-	Saldo= 1200,00 €, Fehlermeldung!
16.01.2012, 13:25:15	Auszahlung	5005875	3000,00€	-	Saldo= -1280,00 €
16.01.2012, 13:45:56	Auszahlung	5005875	5000,00€	-	Saldo= -1280,00 € Fehlermeldung!

16.01.2012, 15:20:00	Einzahlung	3200450	99,00€	-	Saldo= 100,00 €
17.01.2012, 10:20:00	Auszahlung	3200450	99,00€	-	Saldo= 1,00 €
17.01.2012, 10:20:12	Auszahlung	3200450	1,00€	-	Saldo= 1,00 €; Fehlermeldung!
17.01.2012, 10:20:30	Überweisung	5005612	250	5005875	A: 1750,00 € Z:-1030,00 €
17.01.2012, 10:23:00	Überweisung	7205820	1201	5005875	A: 1200,00 € Z:-1030,00 € Fehlermeldung!
17.01.2012, 10:27:48	Überweisung	3103901	1000	5005875	A: 1000,00 € Z:- 30,00 €
18.01.2012, 11:02:02	Überweisung	3103901	1000	5005875	A: 1000,00 € Z:- 30,00 € Fehlermeldung!

Qualitätskriterien

/Q10/ Die Oberfläche ist übersichtlich und anwenderfreundlich zu gestalten

/Q20/ Das Programm sollte gut wartbar sein. Daher ist redundanter Code¹⁵ zu vermeiden.

¹⁵ Redundanter Code ist doppelter und in der Regel überflüssiger Code. Es gehört zu einem guten Programmierstil doppelten Code zu vermeiden. Das Programm wird hierdurch unnötig größer und schwerer wartbar.

6.2 Nutzung des Steuerelements DataGridView (Tabellenanzeige)

Bisher haben wir eine tabellarische Anzeige durch Nutzung der `String.Format()`-Methode erreicht. Dies ist zuweilen sehr mühsam und relativ unflexibel, da man feste Anzeigenbreiten vorgibt. In C# bietet sich deshalb die Nutzung des Steuerelements `DataGridView` an. Dieses Steuerelement kann sogar an Datenquellen wie typisierte Listen gebunden werden und die tabellarische Darstellung läuft weitgehend automatisch.

Im Folgenden ist die Handhabung dargestellt:

Wir benötigen eine Verwaltungsklasse (Kontenliste) um alle Konten verwalten zu können. Hier verwenden wir einen besonderen Listtyp, die sogenannte **BindingList**. Diese verfügt über alle Eigenschaften der üblichen typisierten Liste, und aktualisiert zusätzlich alle Änderungen an der Liste auch in der Ansicht. Dies gilt auch umgekehrt. Man kann in der `DataGridView` Änderungen vornehmen und diese werden (sofern zulässig) direkt in den Objekten der Liste gespeichert (diese Richtung funktioniert ebenfalls mit der ganz normalen typisierten Liste).

An dieser Stelle ist es zusätzlich notwendig die Using-Direktive `using ComponentModel` manuell hinzuzufügen (siehe Abbildung der Klasse **Kontenliste.cs**)

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Data;
using System.Collections;
using System.ComponentModel;

namespace Konto
{
    public class Kontenliste:BindingList<Konto>
    {

    }
}
```

Nach diesen Vorbereitungen ist es sehr einfach eine Tabellenanzeige zu erhalten. Man platziert aus der Toolbox das Steuerelement `DataGridView` auf dem Formular und benennt das Steuerelement entsprechend, z.B. `dgvKonten`.

In der Button-Click-Routine erfolgt nun die Bindung an unsere globale Kontenliste:

```
private void btnAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = eineKontenListe;
    dgvKonten.Columns[2].DefaultCellStyle.Format = "c";
    dgvKonten.Columns[3].DefaultCellStyle.Format = "c";
    dgvKonten.Refresh();
}
```

```

}

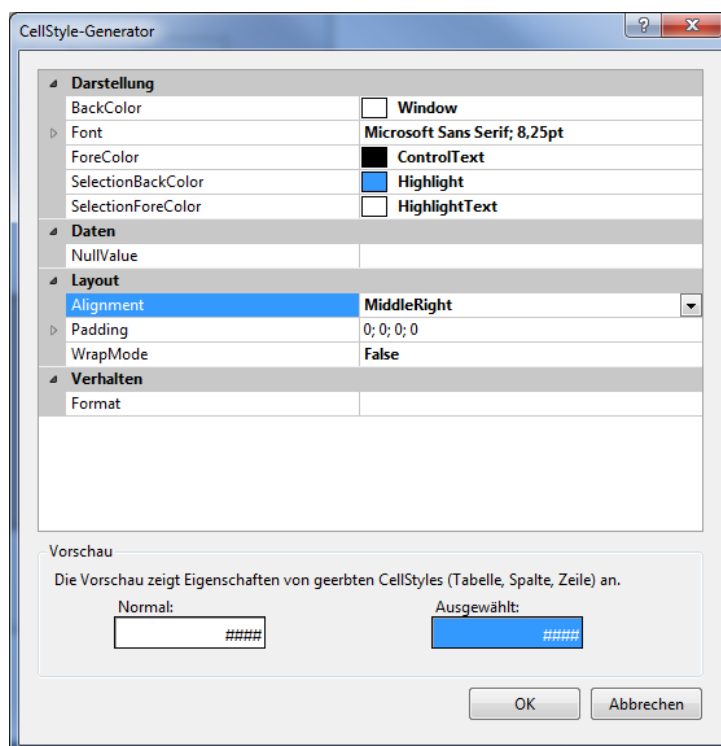
private void btnLeeren_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = null;
}

```

Zu Beginn werden die in der Liste vorhandenen Konten initialisiert und beim Klick auf “Alle Konten anzeigen” auch angezeigt. Nun legen wir ein neues Konto an und klicken hierzu die Schaltfläche „Konto anlegen“:

Dadurch, dass wir die DataGridView an die BindingList gebunden haben, wird ohne erneutes Klicken des Buttons „Alle Konten anzeigen“ das neue Konto ergänzt und mit angezeigt.

Das Erscheinungsbild des DataGridViews kann einfach über sein Eigenschaftsfenster geändert werden. Über den Menüpunkt DefaultCellStyle können Hintergrundfarbe, Schriftart, Farben und Textausrichtung eingestellt werden. Auch ein bestimmtes Zahlenformat kann unter „Format“ gewählt werden. Dieses würde allerdings auf die gesamte DataGridView angewendet werden, was meistens nicht erwünscht ist.



Möchte man nur bestimmten Spalten ein Zahlenformat zuweisen, wie z.B. den Feldern Saldo und Kreditlimit das Währungsformat und ein €-Symbol, so muss man dies manuell im Quellcode tun:

```

dgvKonten.Columns[2].DefaultCellStyle.Format = "C";
dgvKonten.Columns[3].DefaultCellStyle.Format = "C";

```

Auswirkung: Der jeweiligen Spalte des DataGridViews wird das Währungsformat ("c") zugewiesen.

	Kontonummer	Kontoinhaber	Saldo	Kreditlimit
▶	5005878	Müller, Hans	2.850,20 €	6.000,00 €
	5005612	Schmidt, Lolita	2.300,00 €	5.000,00 €
	7205822	Merck, Paul	2.850,20 €	0,00 €

Exkurs: Man kann DataGridViews auch ohne Datenbindung nutzen und im Code selbst aufbauen, wie die folgenden Codezeilen zeigen:

```
private void frmKonto_Load(object sender, EventArgs e)
{
    // Konstruktoraufwurf mit Spaltenbezeichnung und angezeigter Spaltenüberschrift (Header)
    dgvKonten.Columns.Add("kontonummer", "Kontonummer");
    dgvKonten.Columns.Add("kontoinhaber", "Kontoinhaber");
    dgvKonten.Columns.Add("saldo", "Saldo");
    dgvKonten.Columns.Add("kreditlimit", "Kreditlimit");

    // neue Zeilen werden angefügt
    dgvKonten.Rows.Add(3);
    // Fülle erste Zeile, Spaltenindex steht an erster Stelle
    dgvKonten[0, 0].Value = "5005878";
    dgvKonten[1, 0].Value = "Mueller, Hans";
    dgvKonten[2, 0].Value = "2850.20";
    dgvKonten[3, 0].Value = "6000.00";
    // Fülle zweite Zeile
    dgvKonten[0, 1].Value = "5005612";
    dgvKonten[1, 1].Value = "Schmidt, Lolita";
    dgvKonten[2, 1].Value = "2300.00";
    dgvKonten[3, 1].Value = "5000.00";
    // Fülle dritte Zeile
    dgvKonten[0, 2].Value = "7205822";
    dgvKonten[1, 2].Value = "Merck, Paul";
    dgvKonten[2, 2].Value = "2850.20";
    dgvKonten[3, 2].Value = "0.00";
}
```

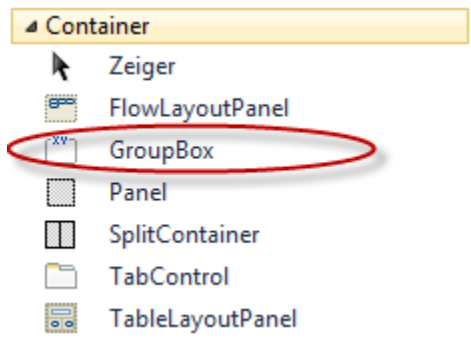
In einem solchen Fall muss man natürlich auch selbst immer auf die Konsistenz zwischen Listeninhalt und Anzeige achten.

6.2 Nutzung von RadioButtons und Checkboxes

Radiobuttons und Checkboxes sind Window-Steuerelemente, die man nutzen kann, wenn man dem Anwender mehrere Optionen zur Auswahl anbieten möchte. Plaziert man Radiobuttons gemeinsam in eine Groupbox, kann immer nur eine Option angewählt werden. Bei Checkboxes ist eine Mehrfachauswahl möglich. Das Einbetten der Auswahloptionen in einer Groupbox hat außerdem den Vorteil, dass das Formular übersichtlicher wird, und dass beim Verschieben der Groupbox, alle eingebetteten Steuerelemente mit verschoben werden.

Will man beispielsweise bei der Eingabe neuer Konten den Anwender wählen lassen, ob es sich um ein Giro- oder Sparkonto handelt, geht man wie folgt vor:

1. GroupBox aus der Toolbox auf dem Formular platzieren:



2. Unter Eigenschaften die Text-Eigenschaft füllen: z.B. Kontoart eintragen.
3. Zwei Radiobuttons auf die Groupbox ziehen und jeweils eine aussagekräftige Bezeichnung in der Eigenschaft Name eintragen, z. B. rbGiro und rbSpar sowie in die Eigenschaft Text (Girokonto und Sparkonto).
4. Testet man nun das Programm, so sieht man, dass man nur Girokonto oder Sparkonto anwählen kann, aber nicht beide.
5. Will man nun im Programm überprüfen, welche Auswahl der Anwender getroffen hat, kann man hierzu die Eigenschaft checked abfragen, ein Boolean-Wert, der true ist, falls die Option gewählt wurde und false sonst.

```
if (rbGiro.Checked)
{
    // hier soll ein Girokonto angelegt werden
}
```

6.3 Eine Lösung mit den bisher bekannten Mitteln

Im ersten Schritt setzen wir die Anforderungen mit den bisher bereitgestellten Programmiermethoden um. Hierbei wird zunächst die Methode Überweisen vernachlässigt. Diesen Aspekt nehmen wir erst im Kapitel „Umsetzung von Assoziationen“ auf. Genutzt werden aber die zuvor beschriebene Tabellendarstellung sowie Radiobuttons für die Auswahl der Kontoart.

Da bei unserem bisherigen Vorgehen Girokonten und Sparkonten in verschiedenen Listen verwaltet werden, werden im Formular auch zwei DataGridViews verwendet zur getrennten Darstellung von Giro- und Sparkonten. In der folgenden Abbildung sieht man den Zustand des Formulars direkt nach dem Laden des Programms.

Sparkasse Kleve

Kontoart
☒ Girokonto ☐ Sparkonto

Konto anlegen
 Kontonummer:
 Kontoinhaber:
 Saldo:
 Kreditlimit:
 Konto anlegen

Ein-/Auszahlungen
 Betrag:
 Einzahlen Auszahlen

Konten anzeigen

Kontonummer	Kontoinhaber	Saldo
3103901	Mueller, Hans	2000
3004813	Schmidt, Heinz	5000

Kontonummer	Kontoinhaber	Saldo	Kreditlimit
5005875	Mueller, Hans	1000	6000
5005612	Schmidt, Lola	2000	5000

Wir wollen im Folgenden die Programmiererteile jeweils für Giro- und Sparkonten gegenüberstellen.

Klasse Girokonto

```
public class Girokonto
{
    private long fKontonummer;

    public long Kontonummer
    {
        get { return fKontonummer; }
    }

    private string fKontoinhaber;

    public string Kontoinhaber
    {
        get { return fKontoinhaber; }
        set { fKontoinhaber = value; }
    }

    private double fSaldo;

    public double Saldo
    {
        get { return fSaldo; }
    }

    private double fKreditlimit;

    public double Kreditlimit
    {
        get { return fKreditlimit; }
        set { fKreditlimit = value; }
    }
}
```

Klasse Sparkonto

```
public class Sparkonto
{
    private long fKontonummer;

    public long Kontonummer
    {
        get { return fKontonummer; }
    }

    private string fKontoinhaber;

    public string Kontoinhaber
    {
        get { return fKontoinhaber; }
        set { fKontoinhaber = value; }
    }

    private double fSaldo;

    public double Saldo
    {
        get { return fSaldo; }
    }
}
```

```

    public Girokonto(long
pKontonummer, string pKontoinhaber, double
pSaldo, double pKreditlimit)
    {
        fKontonummer = pKontonummer;
        Kontoinhaber = pKontoinhaber;
        fSaldo = pSaldo;
        Kreditlimit = pKreditlimit;
    }

    public void Auszahlen(double
pBetrag)
    {
        if (pBetrag >
Math.Round((fSaldo + fKreditlimit),2))
        {
            throw new
Exception("Auszahlung in Höhe von " +
pBetrag + " € nicht möglich."
            + "\n" + "Maximal
möglicher Auszahlungsbetrag: " +
(Math.Round(Saldo + Kreditlimit,2)) + " €
");
        }
        else
        {
            fSaldo = fSaldo -
Math.Max(pBetrag, 0);
        }
    }

    public void Einzahlen(double
pBetrag)
    {
        fSaldo = fSaldo +
Math.Max(pBetrag, 0);
    }
}

```

```

public class
GiroKontenListe:BindingList<Girokonto>
{
    public int KontoSuche(long
pKontonummer)
    {
        for (int i = 0; i <
this.Count; i++)
        {
            if (this[i].Kontonummer ==
pKontonummer)
            {
                return i;
            }
        }
        return -1;
    }

    public void
AddEindeutigesKonto(Girokonto kto)
    {
        if
(this.KontoSuche(kto.Kontonummer) > -1)

```

```

    public Sparkonto(long
pKontonummer, string pKontoinhaber, double
pSaldo)
    {
        fKontonummer = pKontonummer;
        Kontoinhaber = pKontoinhaber;
        fSaldo = pSaldo;
    }

    public void Auszahlen(double
pBetrag)
    {
        if (pBetrag >
Math.Round(fSaldo+1, 2))
        {
            throw new
Exception("Auszahlung in Höhe von " +
pBetrag + " € nicht möglich."
            + "\n" + "Maximal
möglicher Auszahlungsbetrag: " +
(Math.Round(Saldo- 1, 2)) + " € ");
        }
        else
        {
            fSaldo = fSaldo -
Math.Max(pBetrag, 0);
        }
    }

    public void Einzahlen(double
pBetrag)
    {
        fSaldo = fSaldo +
Math.Max(pBetrag, 0);
    }
}

```

```

public class
SparkontenListe:BindingList<Sparkonto>
{
    public int KontoSuche(long
pKontonummer)
    {
        for (int i = 0; i <
this.Count; i++)
        {
            if (this[i].Kontonummer ==
pKontonummer)
            {
                return i;
            }
        }
        return -1;
    }

    public void
AddEindeutigesKonto(Sparkonto kto)
    {
        if
(this.KontoSuche(kto.Kontonummer) >=0)

```

<pre> { throw new Exception("Konto kann nicht eingefügt werden, da Kontonummer bereits vorhanden!"); } else { this.Add(kto); } } } </pre>	<pre> { throw new Exception("Konto kann nicht eingefügt werden, da Kontonummer bereits vorhanden!"); } else { this.Add(kto); } } } </pre>
---	---

Beim Vergleich der Klassen müssen wir feststellen, dass das Qualitätsmerkmal „/Q20/ Das Programm sollte gut wartbar sein. Daher ist redundanter Code zu vermeiden.“ sehr stark verletzt ist. Gelb hinterlegt sind die unterschiedlichen Codestellen. D.h. der ganze übrige Teil ist redundant. Auch im zugehörigen Formularcode finden sich doppelte Codezeilen, die sich jeweils auf Giro- oder Sparkonten beziehen:

```

using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();
        }

        private void label4_Click(object sender, EventArgs e)
        {
        }

        private void textBox3_TextChanged(object sender, EventArgs e)
        {
        }

        GiroKontenListe klevGirokonten;
        SparkontenListe klevSparkonten;
        private void Form1_Load(object sender, EventArgs e)
        {
            //
            klevGirokonten = new GiroKontenListe();
            klevSparkonten = new SparkontenListe();
            klevSparkonten.AddEindeutigesKonto(new Sparkonto(3103901, "Mueller,
Hans", 2000.00));
            klevSparkonten.AddEindeutigesKonto(new Sparkonto(3004813, "Schmidt,
Heinz", 5000.00));
            // klevSparkonten.Add(new Sparkonto(3103901, "Meier, Sabine", 0.00));
            klevGirokonten.AddEindeutigesKonto(new Girokonto(5005875, "Mueller,
Hans", 1000.00, 6000.0));
            klevGirokonten.AddEindeutigesKonto(new Girokonto(5005612, "Schmidt,
Lola", 2000.00, 5000.0));
        }
    }
}

```

```

        //kleveGirokonten.AddEindeutigesKonto(new Girokonto(7205820, "Jansen,
Gregor", 3000.00, 0.0));
        dgvGirokonten.DataSource = kleveGirokonten;
        dgvSparkonten.DataSource = kleveSparkonten;
    }

    private void btnAnlegen_Click(object sender, EventArgs e)
    {
        try
        {
            long kontonummer = Convert.ToInt64(tbxKontonummer.Text);
            string kontoinhaber = tbxKontoinhaber.Text;
            double saldo = Convert.ToDouble(tbxSaldo.Text);
            if (rbGiro.Checked)
            {
                double kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
                kleveGirokonten.AddEindeutigesKonto(new Girokonto(kontonummer,
kontoinhaber, saldo, kreditlimit));
            }
            else
            {
                kleveSparkonten.AddEindeutigesKonto(new Sparkonto(kontonummer,
kontoinhaber, saldo));
            }
        }
        catch (Exception ex)
        {
            MessageBox.Show("Eingabefehler: " + ex.ToString());
        }
    }

    private void rbGiro_CheckedChanged(object sender, EventArgs e)
    {
        if (rbGiro.Checked)
        {
            tbxKreditlimit.Enabled = true;
            dgvGirokonten.Refresh();
        }
        else
        {
            tbxKreditlimit.Enabled = false;
            tbxKreditlimit.Text = 0.ToString();
            dgvSparkonten.Refresh();
        }
    }

    private void btnEinzahlen_Click(object sender, EventArgs e)
    {
        try
        {
            long ktonr = Convert.ToInt64(tbxKontonummer.Text);
            double betrag = Convert.ToDouble(tbxBetrag.Text);
            int index = kleveGirokonten.KontoSuche(ktonr);
            if (index > -1)
            {
                kleveGirokonten[index].Einzahlen(betrag);
                dgvGirokonten.Refresh();
            }
            else
            {
                index = kleveSparkonten.KontoSuche(ktonr);
                if (index > -1)

```

```

        {
            kleveSparkonten[index].Einzahlen(betrag);
            dgvSparkonten.Refresh();
        }
        else
        {
            throw new Exception("Kontonummer nicht im System");
        }
    }
}
catch (Exception ex)
{
    MessageBox.Show("Eingabefehler: " + ex.ToString());
}
}

private void btnAnzeigen_Click(object sender, EventArgs e)
{
    dgvGirokonten.Refresh();
    dgvSparkonten.Refresh();
}

private void btnAuszahlen_Click(object sender, EventArgs e)
{
    try
    {
        long ktonr = Convert.ToInt64(tbxKontonummer.Text);
        double betrag = Convert.ToDouble(tbxBetrag.Text);
        int index = kleveGirokonten.KontoSuche(ktonr);
        if (index > -1)
        {
            kleveGirokonten[index].Auszahlen(betrag);
            dgvGirokonten.Refresh();
        }
        else
        {
            index = kleveSparkonten.KontoSuche(ktonr);
            if (index > -1)
            {
                kleveSparkonten[index].Auszahlen(betrag);
                dgvSparkonten.Refresh();
            }
            else
            {
                throw new Exception("Kontonummer nicht im System");
            }
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Eingabefehler: " + ex.ToString());
    }
}
}
}

```

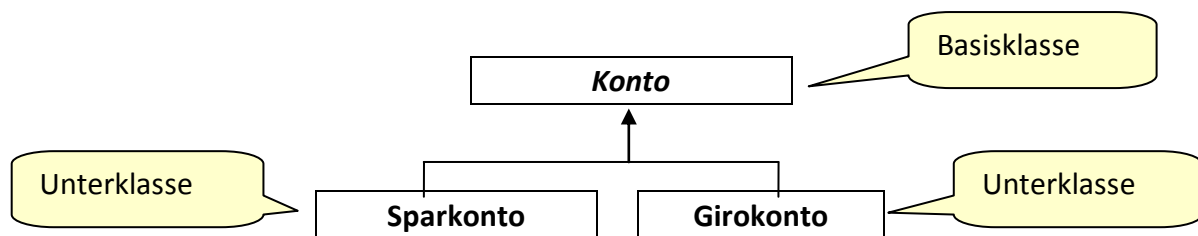
Die Vererbung ist nun ein Konzept Gemeinsamkeiten zweier Klassen in einer gemeinsamen Oberklasse zu verwalten und nur die Unterschiede in den sogenannten Kindklassen zu belassen. Im Folgenden werden die Grundzüge der Vererbung sowie ihre programmtechnische Umsetzung näher erläutert.

6.4 Allgemeine Hinweise zur Vererbung

Bei der Vererbung werden Codebestandteile, die in mehreren Klassen gleichzeitig auftreten, in eine neue allgemeine Klasse (die sog. Basisklasse) gepackt. Die Basisklasse vererbt diesen allgemeinen Code (Instanzvariablen und Methoden) an Unterklassen. Voraussetzung dafür ist, dass die Unterklassen angeben, dass die allgemeine Klasse ihre Basisklasse ist bzw. dass sie die Basisklasse erweitern. Eine Unterklasse kann eigene neue Methoden und Instanzvariablen hinzufügen und Methoden überschreiben, die sie von der Basisklasse erbt.

Der Vorgang, aus vorhandenen Klassen Gemeinsamkeiten zu isolieren und in einer Basisklasse zusammenzufassen, wird als **Generalisierung** bezeichnet. Der Vorgang, dass eine Klasse alle Merkmale einer anderen Klasse besitzt und noch einige andere hinzufügt, wird als **Spezialisierung** bezeichnet.

Klassendiagramm (vereinfacht):



In einem Klassendiagramm wird die Vererbungsbeziehung durch einen Pfeil dargestellt mit unausgefüllter Spitze. Hierbei zeigt eine Unterklasse stets auf die Basisklasse. Die obige Darstellung bedeutet, dass die Klassen **Sparkonto** und **Girokonto** von der Klasse **Konto** erben und diese durch eigene Attribute und Methoden erweitern.

Bei einer Vererbung wird auch von einer „IST-EIN-Beziehung“ gesprochen, weil jedes Objekt der Unterklasse ein Objekt der Oberklasse ist.

Beispiele für mögliche Vererbungsbeziehungen:

- Ein Girokonto **ist ein** Konto.
- Ein PKW **ist ein** Auto.
- Ein Angestellter **ist ein** Mitarbeiter.

Das Vererbungs-IST-EIN-Verhältnis funktioniert nur in eine Richtung! Ein Girokonto ist ein Konto, aber ein Konto ist nicht zwingend ein Girokonto.

Beispiele für nicht mögliche Vererbungsbeziehungen (Keine IST-EIN-Beziehung):

- Ein PKW ist kein Motor, sondern **hat** einen Motor.

6.5 Methoden überschreiben

Die Methoden einer Basisklasse können in den Unterklassen überschrieben, d. h. mit eigenem Quelltext überschrieben bzw. erweitert, werden. Dabei muss die Signatur der überschreibenden Methode genau mit der Signatur der überschriebenen Methode übereinstimmen. Die überschreibenden Methoden bestätigen sozusagen, dass Sie den Methoden-Vertrag einhalten werden. Methoden, die die gleiche Signatur haben, aber in verschiedenen Klassen unterschiedlichen implementiert sind, bezeichnet man als polymorphe Methoden (vielgestaltig).

Damit in C# eine Methode der Oberklasse in einer Unterklasse überschrieben werden darf, muss die Oberklasse dieses zunächst „genehmigen“. Dies geschieht mit dem Schlüsselwort **virtual**.

Also: Methoden einer Basisklasse können nur dann überschrieben werden, wenn Sie in der Basisklasse als **virtual** gekennzeichnet sind! Soll die in der Basisklasse angegebene Methode in der Unterklasse modifiziert werden, so macht man dies mit dem Schlüsselwort **override** in der Methodensignatur der Kindklasse kenntlich. Dies ist notwendig, damit nicht unbeabsichtigt gleiche Methodenbezeichner in Unter- und Oberklasse verwandt werden.

Wenn Sie eine Methode auf einer Instanz einer Unterklasse aufrufen, die von der Unterklasse überschrieben wurde, so wird die überschriebene Version der Methode aufgerufen. Man kann sagen, dass die **niedrigste Methode in der Vererbung gewinnt**.

Exkurs1: Eine Methode überladen

Wenn in einer Basis- und einer Unterklasse zwei Methoden mit dem gleichen Namen, aber unterschiedlichen Signaturen vorliegen, so werden die Methoden nicht überschrieben, sondern überladen. Eine überladene Methode ist einfach eine Methode, die zufällig den gleichen Methodennamen hat. Das hat nichts mit Polymorphie zu tun. Eine **überladene** Methode ist somit NICHT das Gleiche wie eine **überschriebene** Methode.

Exkurs2: Eine Methode verdecken

Man kann auch bewusst in der Unterklasse eine gleichnamige Methode zur Oberklasse einführen, die die Methode der Basisklasse verdeckt. Dies kann gemacht werden, wenn eine Basismethode gar nicht zu den Belangen der Unterklasse passt. Damit auch hier kein unbeabsichtigtes Überdecken auftritt, muss man in der Unterklasse das Schlüsselwort **new** verwenden!

6.6 Die Basisklassenversion einer Methode aufrufen

Wenn eine Unterklasse eine Methode der Basisklasse erweitern will, überschreibt sie diese Methode und fügt ihr den restlichen Code hinzu. Der Aufruf der Basisklassenmethode erfolgt mit Hilfe des Schlüsselwortes **base**, welches eine Referenz auf den Basisklassenteil eines Objektes liefert.

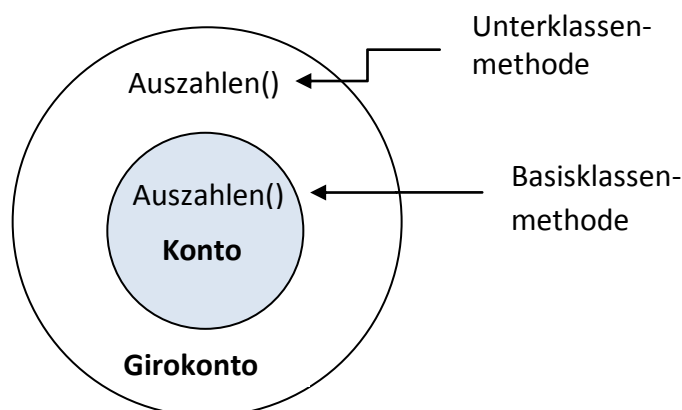
```
base.oberklassenMethode()
```

Basisklasse Konto

```
public virtual void Auszahlen(double pBetrag)
{
    fSaldo = fSaldo - Math.Max(pBetrag, 0);
}
```

Unterklasse Girokonto

```
public override void Auszahlen(double pBetrag)
{
    if (pBetrag > fSaldo + Kreditlimit)
    {
        throw new Exception("Auszahlung in Höhe von " + pBetrag + " € nicht möglich."
            + "\n" + "Maximal möglicher Auszahlungsbetrag: " + (Saldo + Kreditlimit) + " € ");
    }
    else
    {
        base.Auszahlen(pBetrag);
    }
}
```

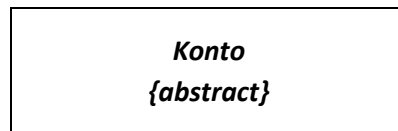


(äußerer Ring = niedrigste Methode, gewinnt, falls nicht das Schlüsselwort **base** verwendet wird.)

6.7 Abstrakte Klassen

Klassen, von denen niemals Objekte erzeugt werden (Beispiel **Konto**), sondern die lediglich als Basisklassen zur Verfügung stehen sollen, werden als abstrakte Klassen bezeichnet. Sie werden mit dem Schlüsselwort **abstract** gekennzeichnet. Wenn eine Klasse abstrakt ist, garantiert der Compiler, dass niemand Objekte dieses Typs erstellen kann.

Abstrakte Klassen werden in UML genau wie normale Klassen dargestellt. Lediglich der Klassenbezeichner wird kursiv geschrieben. Alternativ hierzu kann unterhalb des Klassenbezeichners der Zusatz **{abstract}** angegeben werden



Im Kontobeispiel ist die Klasse **Konto** abstrakt, weil von ihr nie Objekte erzeugt werden. Jedes Konto ist entweder ein **Girokonto** oder ein **Sparkonto**. Die abstrakte Klasse **Konto** bietet somit die Sicherheit, dass nur Instanzen von den Klassen **Sparkonto** und **Girokonto** erzeugt werden können. Abstrakte Klassen sind nur dann sinnvoll, wenn sie mindestens eine konkrete Unterklasse besitzen. Abstrakte Klassen können auch Unterklassen abstrakter Klassen sein.

Abstrakte Klassen können noch einen weiteren Zweck erfüllen: das Vorschreiben von abstrakten Methoden, die in einer Unterklasse zwingend zu implementieren sind. (hier: Zinszahlung).

6.8 Abstrakte Methoden

Wenn die Basisklasse eine Operation benötigt, die in allen Unterklassen erforderlich ist, je nach Unterklasse aber unterschiedliche Ausprägungen hat (s. `Zinszahlung()`), so wird diese Operation in der Basisklasse als abstrakte Operation definiert, die von den Unterklassen zu spezifizieren bzw. zu überschreiben ist. Eine abstrakte Methode enthält lediglich die Deklaration des Methodenkopfes aber keine Implementierung des Methodenrumpfes. Hierum müssen sich die Unterklassen kümmern. Daher steht bei abstrakten Methoden anstelle der geschweiften Klammern mit den auszuführenden Anweisungen lediglich ein Semikolon. Zusätzlich wird die Definition mit dem Attribut **abstract** versehen.

Da es keinen Body (Methodenrumpf) gibt, gibt es auch keine geschweiften Klammern. Die Deklaration einer abstrakten Methode wird mit einem Semikolon abgeschlossen.

Syntax einer abstrakten Methode:

```
public abstract void methodenName();
```



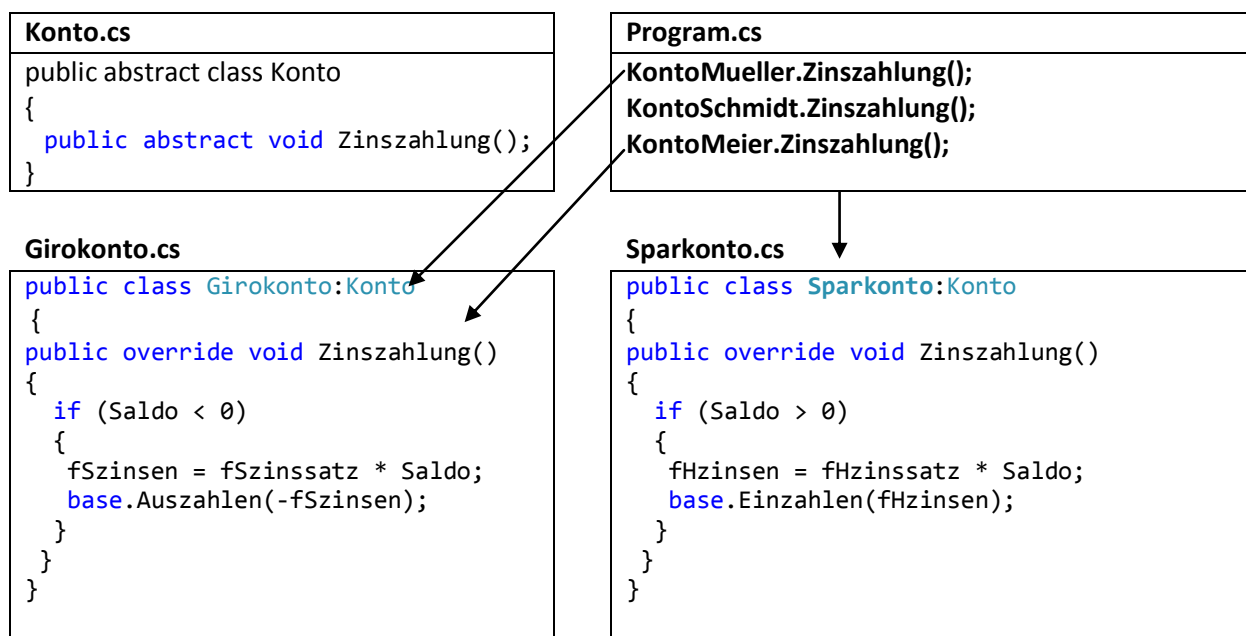
Eine abstrakte Methode
hat einen Kopf
aber keinen Body!!!

Wenn eine Klasse abstrakt ist, heißt das, dass die Klasse erweitert und von allen Unterklassen implementiert (d. h. überschrieben) werden muss. Stecken Sie in eine Klasse eine einzige abstrakte Methode, müssen Sie die Klasse abstrakt machen. Aber Sie können in einer Klasse abstrakte und nicht abstrakte Methoden mischen.

Der Sinn einer abstrakten Methode besteht darin, den Unterklassen vorzuschreiben (wie ein Protokoll), diese Methoden mit der gleichen Signatur (Name und Argumente) und einem kompatiblen Rückgabetypp zu implementieren.

Anwendung auf das Konto-Beispiel

Die Methode **Zinszahlung()** wird in den Unterklassen überschrieben. D. h. die polymorphe Methode **Zinszahlung()** funktioniert mit unterschiedlichen Objekten – egal ob Sparkonto oder Girokonto– und ruft die entsprechenden überschriebenen Methoden der Klassen **Girokonto** und **Sparkonto** auf.



6.9 Die Klasse Object

Die Klasse **Object** ist die Basisklasse aller Klassen („Ultra-Super-Mega-Klasse“). Wir haben schon von Anfang an Unterklassen der Klasse **Object** erstellt, ohne dass wir das explizit in der Klassendefinition hinzufügen mussten. Jede Klasse, die nicht explizit eine andere Klasse erweitert, erweitert implizit **Object**. Eine selbst erstellte Klasse sähe im Prinzip folgendermaßen aus:

```
public class Konto : Object { }
```

Die Klasse **Object** hat die folgenden Methoden:

<code>equals(Object o)</code>	findet heraus, ob ein Objekt gleich einem anderen Objekt ist
<code>getClass()</code>	gibt den Typ des Objekts zurück
<code>hashCode()</code>	gibt den Hashcode für das Objekt zurück
<code>toString()</code>	gibt eine String-Nachricht für das Objekt aus

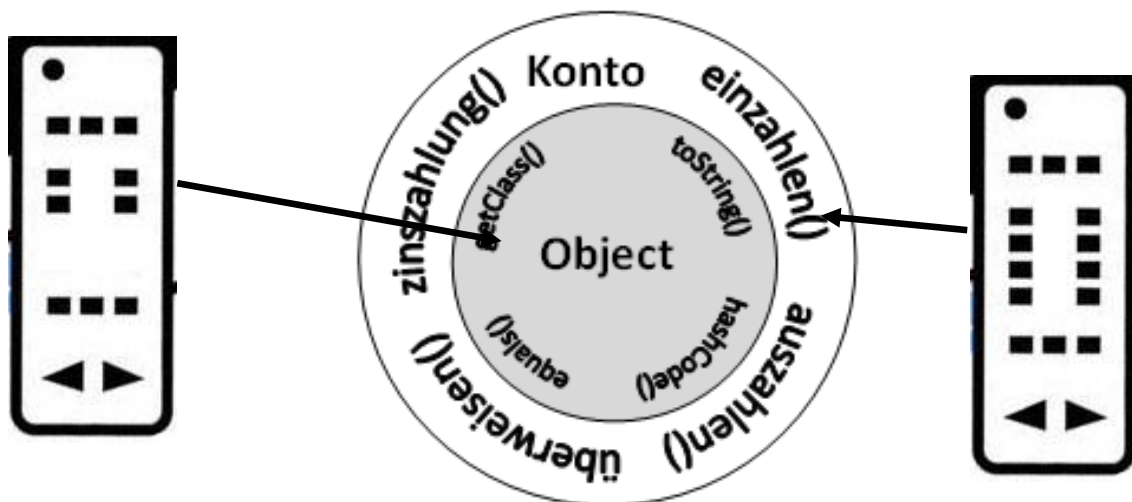
Einige Methoden der Klasse **Object** kann man nicht überschreiben (sie sind `final`), andere schon. `hashCode()`, `equals()` und `ToString()` bieten nur Standardimplementierungen, die in der Regel nicht ausreichen. Sie sollten daher in selbst erstellten Klassen überschrieben werden.

Ein Objekt enthält alles, was es von seinen Basisklassen erbt. Das bedeutet, dass jedes Objekt – unabhängig von seinem eigentlichen Typ – auch eine Instanz der Klasse **Object** ist. Jedes Objekt umhüllt im Grunde einen inneren Kern, der seinen **Object**-Teil enthält.

Sie können ein Konto als **Konto** oder als **Object** behandeln (s. „Polymorphie“ „Viele Formen“). Wenn man eine Referenz als Fernsteuerung betrachtet, so erhält diese Fernsteuerung desto mehr Tasten, je weiter Sie sich im Vererbungsbaum nach unten bewegen. Eine Fernsteuerung (Referenz) des Typs **Object** hat nur ein paar Tasten (die Buttons für die veröffentlichten Methoden der Klasse **Object**). Aber eine Fernsteuerung des Typs **Konto** schließt alle Tasten der Klasse **Object** plus alle neuen Tasten (für neue Methoden) der Klasse **Konto** ein. Je spezifischer die Klasse ist, desto mehr Tasten kann sie haben.

Object o = einKonto

Konto einKonto = new Konto();



6.10 Arraylists bzw. typisierte Listen und Vererbung

Die Vererbung ermöglicht auch, Listen für Objekte unterschiedlicher Klassen einer Vererbungshierarchie zu erstellen. So lässt sich für die Kontoverwaltung z. B. eine typisierte Liste vom Typ der abstrakten Basisklasse **Konto** erstellen. In diese Liste können sowohl Objekte der Klasse **Girokonto** als auch Objekte der Klasse **Sparkonto** aufgenommen werden.

Kontenliste.cs

```
public class Kontenliste : List<Konto>
{
}
}
```

Vorteile der Vererbung

- Man vermeidet doppelten Code, wenn man allgemeines Verhalten aus einer Gruppe von Klassen herauszieht und es in eine Basisklasse steckt.
- Überarbeitungen werden einfacher, weil Änderungen des Codes in der Basisklasse in allen Klassen wirksam werden, die die Basisklasse erweitern.
- Durch die Vererbung wird ein allgemeines Protokoll (Vertrag) für eine Gruppe von Klassen vereinbart. Hierdurch ist garantiert, dass die Unterklassen alle Methoden haben, die die Basisklasse hat.
- Ganz wichtig ist das sogenannte Prinzip der **Polymorphie** (Vielgestaltigkeit): Obwohl in einer Liste sowohl Giro- als auch Sparkonten verwaltet werden, wird zu jedem Objekt z.B. immer die richtige Methode `Auszahlen()` verwandt, da der zugrundeliegende Datentyp „gewinnt“.

6.11 Zugriffsmodifizierer

Bislang wurde davon ausgegangen, dass alle Unterklassen die Eigenschaften der Basisklasse übernehmen. Das ist aber nicht immer der Fall. Zwar besitzt eine abgeleitete Klasse immer alle Variablen und Methoden der Basisklasse, sie kann aber unter Umständen nicht darauf zugreifen, wenn deren Sichtbarkeit eingeschränkt wurde. Zugriffsmodifizierer können den Zugriff auf die Elemente einer Klasse (Variable oder Methode) beschränken. Sie werden der Deklaration vorangestellt. C# besitzt vier Zugriffsebenen und drei Zugriffsmodifizierer **public**, **protected** und **private**. Die vierte Zugriffsebene ist die **Default**-Ebene

Private

Der Modifizierer **private** schränkt den Zugriff auf die Elemente der eigenen Klasse ein. Er sollte für alle Instanzvariablen und für Methoden verwendet werden, die äußerer Code nicht aufrufen können soll.

Protected

Der Modifizierer **protected** ist speziell auf die Vererbung bezogen. Er ermöglicht, dass Unterklassen innerhalb und außerhalb des eigenen Paketes auf die erweiterte Basisklasse zugreifen können.

Public

Der Modifizierer **public** ermöglicht, dass auf Elemente einer Klasse von überall her zugegriffen wird, auch von außerhalb des Paketes. Er sollte für Klassen, Konstanten und Methoden verwendet

werden, die für anderen Code veröffentlicht werden muss (z. B. Getter und Setter), sowie die meisten Konstruktoren.

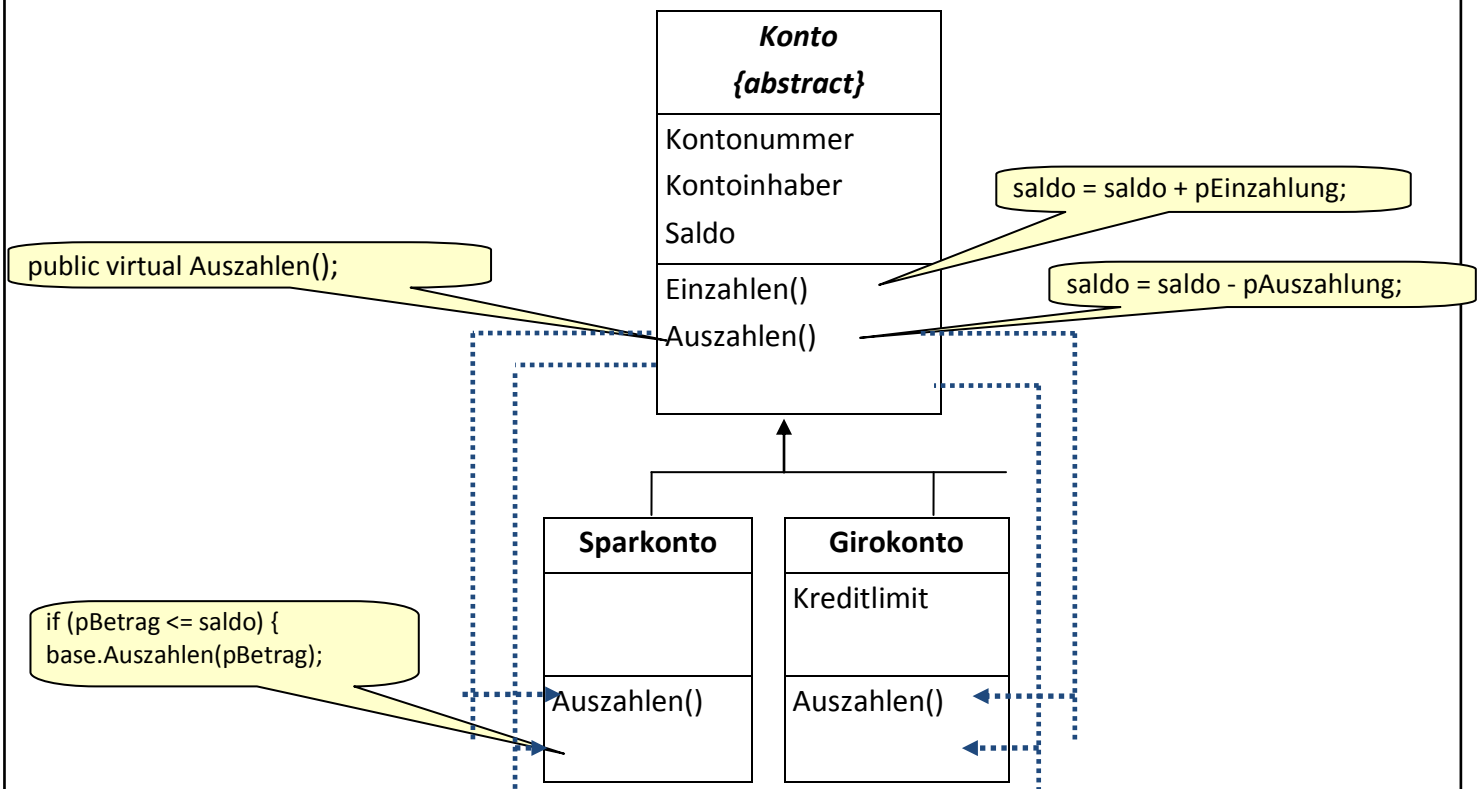
Deklaration	Zugriff erlaubt für
private	nur die Klasse selbst
protected	die Klassen des Paketes und erbende Unterklassen aus demselben und anderen Paketen
public	sämtliche Klassen, also auch außerhalb des Paketes

	UML-Notation	Die Klasse selbst, innere Klassen	Klassen im selben Assembly	Unterklassen	Sonstige Klassen
private	–	Ja	Nein	Nein	Nein
protected	#	Ja	Ja	Ja	Nein
public	+	Ja	Ja	Ja	Ja

6.12 Anwendung auf das Konto-Beispiel

1. In einem ersten Schritt wird festgelegt, welche Attribute und Methoden der Klassen **Sparkonto** und **Girokonto** gleich sind und in die übergeordnete Klasse **Konto** ausgelagert werden können.
2. In einem zweiten Schritt erfolgt die Bestimmung der zusätzlichen Attribute sowie der zusätzlichen Konstruktoren und Methoden der Klassen **Sparkonto** und **Girokonto**.
3. In einem dritten Schritt wird festgelegt, welche Methoden von allen Klassen benötigt werden und welche Methoden nur von einem Teil der Klassen benötigt werden.
 - Die Methode **Einzahlen()** ist für alle Konten gleich und kann daher allein in der Basisklasse deklariert werden.
 - Die Methoden **Auszahlen()** unterscheiden sich durch die Bedingungen: Beim Sparkonto kann nur ausgezahlt und überwiesen werden, wenn der Auszahlungsbetrag kleiner als der Saldo ist. Beim Girokonto kann nur ausgezahlt und überwiesen werden, wenn der Auszahlungsbetrag kleiner als der Saldo und dem Kreditlimit ist.

Als Ergebnis dieser Überlegungen ergibt sich das folgende vereinfachte Klassendiagramm gemäß UML:



Umsetzung in C#: Vollständiger Code: s. Anlage

1. Die Klasse **Konto** muss zur abstrakten Klasse gemacht werden, damit keine Objekte direkt von dieser Klasse angeleitet werden können.
2. Für die Ausgabe kann in der Klasse **Konto** die **ToString()**-Methode eingesetzt werden. Diese muss in der **Girokonto** überschrieben werden.
3. In den Klassen **Sparkonto** und **Girokonto** ist die Methode **Auszahlen()** zu überschreiben.
4. In der Klasse **Girokonto** wird zusätzlich die Property **Kreditlimit** ergänzt.

6.11 Glossar: Schlüsselworte in C# im Zusammenhang zur Vererbung

:-Operator	In C# wird die Vererbungsbeziehung zwischen zwei Klassen mit Hilfe des :-Operators ausgedrückt, der zwischen dem Namen der abgeleiteten Klasse und der Basisklasse in der gesetzt wird. z.B. <code>public Class Girokonto:Konto</code> { }
Explizite Vererbung	Die mit dem :-Operator beschriebenen Vererbungen bezeichnet man auch als explizite Vererbungen.
Implizite Vererbung	Erbt eine Klasse nicht explizit von einer Basisklasse, erbt sie implizit von der Klasse object (die „Wurzelklasse“). Dadurch gibt es z.B. einige Methoden wie die Methode ToString(), die implizit in allen Klassen vorhanden ist.
Klasse object	
override	Soll eine Methode der (impliziten oder expliziten) Basisklasse überschrieben werden, so ist das Schlüsselwort override zu verwenden, dass zwischen dem Zugriffsmodifizierer und der Typangabe des Rückgabewertes zu platzieren ist. (Das Überschreiben einer Methode soll bewusst vorgenommen werden.)
Zugriffsmodifizierer	Legen den Gültigkeitsbereich z.B. von Klassen, Feldern, Eigenschaften und Methoden fest. Zugriffsmodifizierer sind public, private, internal, protected
Internal	Klassen, die ohne Zugriffsmodifizierer angelegt werden, sind implizit internal. Die Sichtbarkeit einer solchen Klasse ist auf die sie enthaltende Assembly (=Untereinheit einer Komponente) begrenzt.
public	Soll die Klasse der gesamten Anwendung zur Verfügung stehen, ist der Zugriffsmodifizierer public zu verwenden.
private	Felder, Eigenschaften und Methoden, die mit dem Zugriffsmodifizierer private verwendet werden, können nur in der zugehörigen Klasse verwendet werden.
protected	Sichtbarkeit für die Klasse selbst und alle erbbenden Klassen.
Polymorphie	Gehört zu den Grundprinzipien der Objektorientierung. Es bezeichnet die Fähigkeit eines Objektes, je nach Kontext verschiedenen Typen zu entsprechen. Eine Spezialisierung kann also ebenfalls einer Methode übergeben werden, für deren Parameter der Typ der Generalisierung festgesetzt wurde.
Mehrfachvererbung	In C# nicht unterstützt!
virtual	Methoden, einer Basisklasse können nur dann überschrieben werden, wenn Sie in der Basisklasse als virtual gekennzeichnet sind. (private und virtual schließen sich aus!)
base	Soll in dem Code der überschreibenden Methode die Methode der Basisklasse aufgerufen werden, so verwendet man das Schlüsselwort base. Dieses verweist auf den Typ der Basisklasse. z.B.: <code>base.Auszahlen();</code>
sealed	Virtual in der Basisklasse erlaubt die Überschreibung gleich auch für alle tieferen Hierarchiestufen. Will man dies nicht zulassen, so kann man die Methode versiegeln: z.B. <code>public override sealed void Auszahlen();</code> Auch ganze Klassen lassen sich vollständig versiegeln. (Hierfür kann die Klasse Math als Beispiel dienen.)

abstract (abstrakte Klassen)	Häufig sind Generalisierungen nur Gedankenkonstrukte, um Gemeinsame Eigenschaften und Methoden zweier oder mehrerer Klassen zusammenzufassen. Real existieren hierzu oft keine Objekte. Dann ist es sinnvoll eine Klasse als abstract zu definieren. Dies bewirkt, dass von dieser Klasse keine Objekte erzeugt werden können. Abstrakte Klassen machen nur Sinn, wenn man von diesen erben kann. Deshalb ist die gleichzeitige Verwendung der Schlüsselworte sealed und abstract nicht zulässig.
abstract (abstrakte Methoden)	In einer abstrakten Klasse können auch abstrakte Methoden definiert werden. Für diese wird ebenfalls das Schlüsselwort abstract angegeben und der Methodenrumpf fehlt gänzlich. Der Methodenkopf wird mit einem Semikolon abgeschlossen. Abstrakte Methoden sind implizit virtual (überschreibbar). z.B. abstract void Auszahlen(double pBetrag); Abgeleitete Klassen müssen abstrakte Methoden der Basisklassen implementieren oder diese wieder als abstrakt kennzeichnen.
base()	Konstruktoren sind die einzigen Elemente einer Klasse, die grundsätzlich nicht vererbt werden können. Dies liegt daran, dass Konstruktoren in der Regel voll initialisierte Objekte zurückgeben sollen. Da in der Regel die „Erben“ mehr Felder besitzen als ihre Basisklasse ist eine Vererbung hier nicht sinnvoll. Allerdings ist es in der Regel sinnvoll den Konstruktor der Basisklasse bei der Definition des eigenen Konstruktors zu nutzen. Public Girokonto:base() {... } Dies bewirkt, dass zunächst der Konstruktor der Basisklasse aufgerufen wird, bevor die Anweisungen in den geschweiften Klammern ausgeführt werden.

7. Assoziationen in C# umsetzen

7.1 Ausgangsbeispiel: Kontoverwaltung für die Sparkasse Kleve

Die Mitarbeiter von Insoft haben in Zusammenarbeit mit der Fachabteilung für das Projekt „Kontenverwaltung“ folgende Zusammenhänge zusammengetragen:

- Ein Kunde kann ein Konto oder mehrere Konten besitzen,
- Ein Kunde erhält eine **eindeutige** Kundennummer, die Kundennummer beginnt bei 1000 und wird fortlaufend erzeugt (1001, 1002, 1003, usw.).
- Um Kontoauszüge zur Verfügung stellen zu können, müssen alle Kontenbewegungen (Transaktionen) mit Bewegungsart, Termin, Betrag etc. festgehalten werden.



Beispieldaten: Kontoeröffnung

KontoNr	Kontoart	KundenNr	Kontoinhaber	Wohnort	Saldo	Kreditlimit
8006130	Giro	1001	Mueller, Hans	Kalkar	2.850,20	6.000,00
5105875	Spar	1001	Mueller, Hans	Kalkar	2.000,00	-
8005612	Giro	1002	Schmidt, Lolita	Kleve	2.300,00	5.000,00
5015612	Spar	1002	Schmidt, Lolita	Kleve	5.000,00	-
8205820	Giro	1003	Meier, Sabine	Kleve	0	0,00
5200450	Spar	1003	Meier, Sabine	Kleve	0	
8206442	Giro	1004	Merck, Paul	Münster	0	0,00
7205820	Giro	1005	Jansen, Gregor	Kleve	2.850,20	0,00
5205820	Spar	1005	Jansen, Gregor	Kleve	5.000,00	-

Beispieldaten: Kontobewegungen (s. Vererbung)

Kontobewegung	Kontonummer	Betrag	Saldo	Fehlermeldung
Einzahlung	8006130	720,00 €	1720,00 €	
Auszahlung	7205820	1800,00€	1200,00 €	
Auszahlung	7205820	1950,00€	1200,00 €	ja
Auszahlung	8006130	3000,00€	1280,00 €	
Auszahlung	8006130	5000,00€	-1280,00 €	ja
Einzahlung	5200450	100,00€	100,00 €	
Auszahlung	5200450	100,00€	0,00 €	

Überweisungen:

/T30/Folgende Kontobewegungen sind mit Hilfe des Programms durchzuführen und zu überprüfen:

Art der Konto- bewegung	Konto- nummer	Betrag	ZielkontoNr	Ergebnis
Einzahlung	5005875	720,00 €	-	Saldo= 1720,00 €
Überweisung	5005612	250	5005875	A: 1750,00 € Z:-1030,00 €
Überweisung	7205820	1201	5005875	A: 1200,00 € Z:-1030,00 € Fehlermeldung!
Überweisung	3103901	1000	5005875	A: 1000,00 € Z:- 30,00 €
Überweisung	3103901	1000	5005875	A: 1000,00 € Z:- 30,00 € Fehlermeldung!

Aufgabenstellung

Es soll nun ein Klassendiagramm entwickelt werden, dass diese Vorgaben berücksichtigt und anschließend sollen die sich ergebenden Neuerungen im Kontoverwaltungsprogramm berücksichtigt werden.

7.2 (mögliche) Umsetzung der Situation in ein Klassendiagramm:

10

7.3 Informationsteil zur Umsetzung von Assoziationen in C#

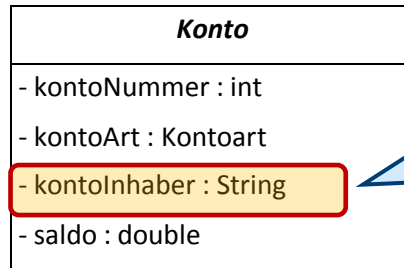
7.3.1 Wie werden Assoziationen in C# umgesetzt?

Im Programm müssen zwei verschiedene Assoziationen berücksichtigt werden

1. Assoziation zwischen Kunde und Konto
2. Komposition zwischen Konto und Kontobewegung

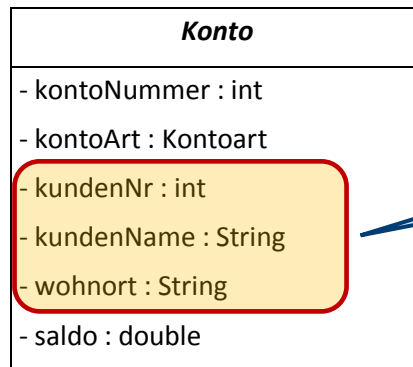
Zu 1. Umsetzung der Assoziation zwischen Kunde und Konto

Alternativ könnte das Kontoobjekt nur den Namen des Kontoinhabers beinhalten.



zu wenig Informationen
über den Kunden
bekannt

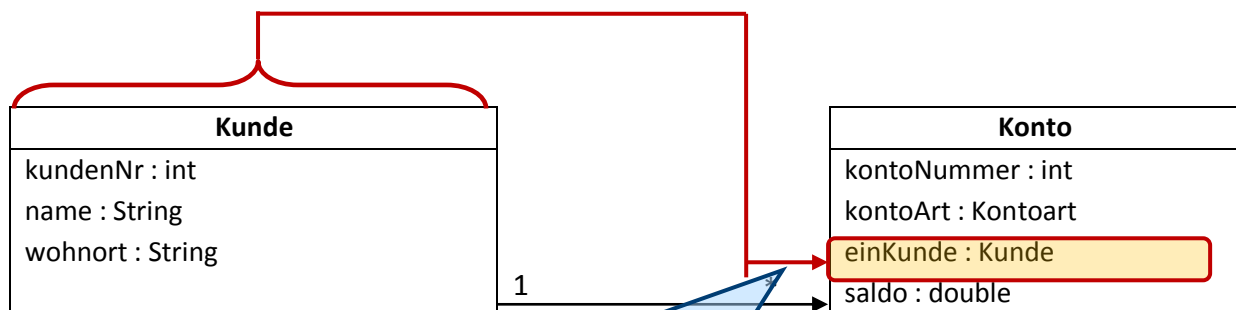
Dies stellt eine Verkürzung dar. In der Praxis werden über den Kunden natürlich weitere Informationen gespeichert wie z. B. eine Kundennummer und die Adressdaten des Kunden. Diese Zusatzinformationen könnte man im Kontoobjekt speichern. Dies wäre aber nicht praxisgerecht. Wenn ein Kunde mehrere Konten besitzt, müssen die Kundeninformationen in jedem Kontoobjekt des Kunden gespeichert werden und ggf. auch geändert werden.



Mögliche Ergänzung:
nicht praxistauglich!

Es empfiehlt sich stattdessen, für die Verwaltung der Kundendaten eine eigenständige Klasse Kunde anzulegen. Wenn für einen Kunden ein Konto angelegt wird, werden die Kundeninformationen dem Konto in Form eines Kundenobjektes übergeben.

In der Klasse Konto werden also keine Einzelangaben zum Kunden gespeichert. Stattdessen wird dem Konto ein Kundenobjekt übergeben, das alle Angaben zum Kunden enthält.



Lösung:

- Eigene Klasse Kunde anlegen
- im Kontenobjekt **Referenz** auf ein Kundenobjekt speichern, d.h. es wird das Kundenobjekt nicht doppelt erstellt, sondern dem Kontoobjekt wird nur mitgeteilt an welcher Stelle das Objekt seines Kontoinhabers gespeichert ist.

Wenn nun diese neue Klasse Kunde in das bestehende Projekt integriert werden soll – welche Änderungen sind zu beachten?

- a) Klasse Kunde wird neu angelegt und die zugehörige Konstruktormethode und ToString-Methode ausprogrammiert.

Kunde Class

Felder

- fAdresse : string
- fKundenId : int
- fLetzteKundenId : int
- fZusName : string

Eigenschaften

- Adresse { get; set; } : string
- KundenId { get; } : int
- LetzteKundenId { get; } : int
- ZusName { get; set; } : string

Methoden

- Kunde(string pZusName, string pAdresse)
- ToString() : string

Statisches Feld, vorbelegt mit 1000, um zu realisieren, dass die Kundennummern beginnend bei 1001 fortlaufend erzeugt werden sollen.

```
public Kunde(string pZusName, string pAdresse)
{
    fKundenId = fLetzteKundenId++;
    //Letzte KundenId wird um eins erhöht und als neue KundenId vergeben;
    fZusName = pZusName;
    fAdresse = pAdresse;
}
```

Vorbereitung zur Nutzung in z.B: Comboboxen.

```
public override string ToString()
{
    return
    fKundenId.ToString()+" : " + fZusName;
}
```

- b) Klasse Kundeliste wird zur Verwaltung mehrere Kunden angelegt und von der Klasse BindingList abgeleitet.

```
public class KundenListe:BindingList<Kunde>
{
}
```

- c) In der Klasse Konto müssen bezüglich des Attributs Kontoinhaber das zugehörige Feld, die zugehörige Property sowie der Konstruktor und die ToString()-Methode angepasst werden:

```
protected Kunde fKontoinhaber;

public Kunde Kontoinhaber
{
    get { return fKontoinhaber; }
    set { fKontoinhaber = value; }
}

public Konto(long pKontonummer, Kunde pKontoinhaber, double
pSaldo)
{
    fKontonummer = pKontonummer;
    Kontoinhaber = pKontoinhaber;
}
```

```

        fSaldo = pSaldo;
    }
    public override string ToString()
    {
        string ausgabeText = "";
        string neueZeile = "\n";
        ausgabeText += String.Format("{0, -13} {1, -15}
{2,10:F2}€ ",
        Kontonummer, Kontoinhaber.ToString(), Saldo) +
        neueZeile;
        return ausgabeText;
    }
}

```

- d) In den erbbenden Klassen Girokonto und Sparkonto muss analog zu c) ebenfalls der Konstruktor angepasst werden sowie die Methode ToString() der Klasse Girokonto.
- e) Weitere Änderungen ergeben sich in der Präsentationsschicht (Anpassung des Formulars und der zugehörigen Codierung)

Die Wesentlichen Änderungen für den Formularcode beziehen sich auf die Form_Load-Methode, Neuerungen sind wieder hervorgehoben:

```

KontenListe klevKonten;
KundenListe klevKunden;

private void Form1_Load(object sender, EventArgs e)
{
    //
    klevKonten = new KontenListe();
    klevKunden = new KundenListe();
    // Kunden erzeugen und anfügen
    klevKunden.Add(new Kunde("Mueller, Hans", "Kalkar"));
    klevKunden.Add(new Kunde("Schmidt, Lolita", "Kleve"));
    klevKunden.Add(new Kunde("Schmidt, Heinz", "Kleve"));
    klevKunden.Add(new Kunde("Merk, Paul", "Münster"));
    klevKunden.Add(new Kunde("Jansen, Gregor", "Kleve"));
}

```

```

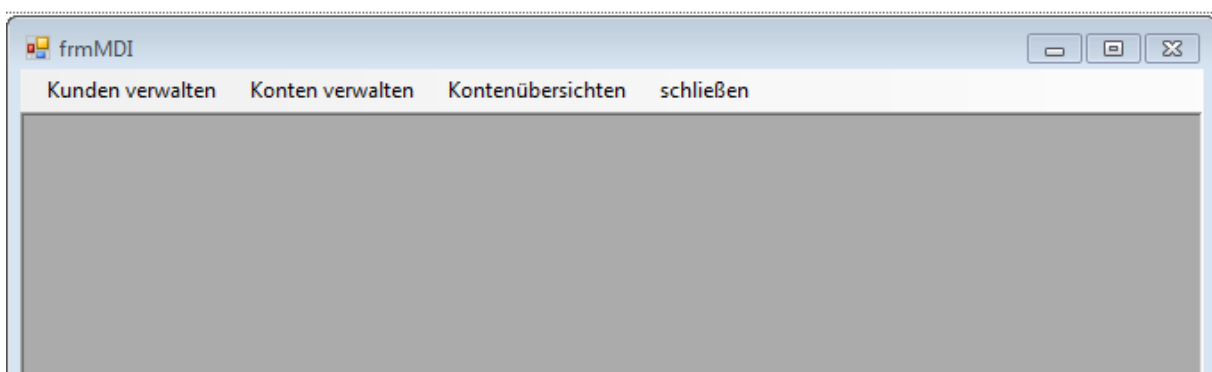
        kleveKonten.AddEindeutigesKonto(new Sparkonto(3103901,
kleveKunden[0], 2000.00));
        kleveKonten.AddEindeutigesKonto(new Sparkonto(3004813,
kleveKunden[2], 5000.00));
        // kleveKonten.Add(new Sparkonto(3103901,"Meier, Sabine",0.00));
        kleveKonten.AddEindeutigesKonto(new Girokonto(5005875,
kleveKunden[0], 1000.00, 6000.0));
        kleveKonten.AddEindeutigesKonto(new Girokonto(5005612,
kleveKunden[1], 2000.00, 5000.0));
        kleveKonten.AddEindeutigesKonto(new Girokonto(7205820,
kleveKunden[4], 3000.00, 0.0));
        dgvKonten.DataSource = kleveKonten;
        cbxKunden.DataSource = kleveKunden;
        //

```

Dies ist nicht die einzige Alternative der Umsetzungsmöglichkeit, aber die einfachste. Denn das neu eingefügte Attribut vom Typ Kunde wird nur auf einer Seite der Beziehung ergänzt. Würde ein Konto mehrere Kontoinhaber besitzen können (in unserer Situation bewusst ausgeschlossen) müsste man als Attribut bereits eine Kundenliste ergänzen. Im folgenden Exkurs finden Sie weitere Möglichkeiten dargestellt, für den Fall, dass Sie die Leserichtung der Assoziation in beide Richtungen erweitern möchten.

Nun fehlt allerdings die Möglichkeit neue Kunden hinzuzufügen. Die Möglichkeit hierzu könnte man im selben Formular anbieten. Praxisgerechter ist es jedoch hierfür ein eigenes Formular anzulegen. Hier im Beispiel werden nur sehr wenige Kundeninformationen erhoben, um die Komplexität und die Schreibarbeit einzugrenzen. Üblicherweise kommt ein Vielfaches hinzu. Deshalb wird im Folgenden gezeigt wie die „Ein-Formular-Anwendung“ auf eine MDI-Anwendung mit mehreren Formularen ausgedehnt werden kann. Besondere Schwierigkeit hierbei ist, dass wir das Objekt kleveKunden dann in zwei Formularen benötigen.

Zur Vorbereitung der MDI-Anwendung wird zunächst ein neues Formular mit MenuStrip erstellt und der Eigenschaft `isMdiContainer = true`.



Im Program.cs wird dieses Formular als Startformular verknüpft und dann das bestehende Formular als Kindfenster verwendet. Im Folgenden ist der Code zu dem zugehörigen Menu-Item und zu dem Schließen-Item angegeben:

```

private void kontenverwaltenToolStripMenuItem_Click(object sender, EventArgs e)
{
    Form1 frm = new Form1();
    frm.MdiParent = this;
    frm.WindowState = FormWindowState.Maximized;
}

```

```

        frm.Show();
    }

    private void schliesßenToolStripMenuItem_Click(object sender, EventArgs e)
    {
        this.Close();
    }

```

Bevor nun ein neues Formular zur Kundenverwaltung angegeben wird, wird die Verwendung globaler Variablen, die in allen Formularen Gültigkeit haben sollen, näher beschrieben.

7.3.2 Listobjekte sollen in mehreren Formularen verfügbar sein, wie geht das?

Das Vorgehen wird hier am Beispiel der Kunden aufgezeigt, und kann analog auf Konten und Kontenbewegungen übertragen werden.

Die Idee ist eine statische Klasse mit Klassenvariablen zu verwenden. Die Klassenvariable existiert dann nur einmal und kann überall im Projekt verwendet werden.

Was macht man mit Daten, die bereits beim Starten des Projektes zur Verfügung stehen sollen (z.B. bereits bestehende Kunden, die nicht über die Oberfläche neu erfasst werden sollen)?

Hierfür schreibt man eine statische Methode, die unsere statische Kundenliste KLeveKunden instanziiert und mit Werten belegt. Diese Methode wird bei Programmstart in program.cs aufgerufen. Die zuvor im Formload erstellte Kundenliste wird dann überflüssig.

Diese Methode muss dann direkt bei Programmstart ausgeführt werden!

```

public static class Global
{
    private static KundenListe fKLeveKunden;

    public static KundenListe KLeveKunden
    {
        get { return Global.fKLeveKunden; }
    }

    public static void InitKLeveKunden()
    {
        fKLeveKunden=new KundenListe();
        fKLeveKunden.Add(new Kunde("Mueller, Hans", "Kalkar"));
        fKLeveKunden.Add(new Kunde("Schmidt, Lolita", "Kleve"));
        fKLeveKunden.Add(new Kunde("Schmidt, Heinz", "Kleve"));
        fKLeveKunden.Add(new Kunde("Merk, Paul", "Münster"));
        fKLeveKunden.Add(new Kunde("Jansen, Gregor", "Kleve"));
    }
}

static class Program
{
    /// <summary>
    /// Der Haupteinstiegspunkt für die Anwendung.
    /// </summary>
    [STAThread]
    static void Main()
    {
        Global.InitKLeveKunden();
    }
}

```

```

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
        Application.Run(new frmMDI());
    }
}

```

Worauf muss jetzt noch geachtet werden?

Überall wo bereits schon die Liste `kleveKunden` verwendet wird, muss diese durch die statische Property `KleveKunden` der Klasse `Global` ersetzt werden. Da die Property statisch ist, erfolgt der Aufruf mit Klassennamen plus Punktselektor plus Propertynamen.

z. B.:

```

KleveKonten.AddEindeutigesKonto(new Girokonto(5005875, Global.KleveKunden[0],
1000.00, 6000.0));

```

7.3.3 Kundenverwaltung ergänzen

An dem Beispiel der Kundenverwaltung soll zum einen gezeigt werden, dass sich nun die globale Eigenschaft `Global.KleveKunden` aus beiden Formularen nutzen lässt, und zum anderen, dass das `DataGridView` auch direkt die Möglichkeit gibt Kundendaten zu ändern, ohne das zusätzlicher Code geschrieben werden muss! Zunächst wird ein Formular mit den unten abgebildeten Steuerelementen erstellt und analog zu oben mit der MDI-Anwendung verknüpft.

	KundenId	ZusName	Adresse
▶	1000	Mueller, Hans	Kalkar
	1001	Schmidt, Lolita	Kleve
	1002	Schmidt, Heinz	Kleve
	1003	Merk, Paul	Münster
	1004	Jansen, Gregor	Kleve
	1005	Meier, Claudia	Kleve, Meisenstr...

Dann wird die Ereignisroutine zum Button ausprogrammiert, und im `Form_Load` das `DataGridView` an die `Global.KleveKunden` gebunden.

```
private void btnAnlegen_Click(object sender, EventArgs e)
{
    if(tbxAdresse.Text==" " || tbxNachname.Text==" ")
    {
        MessageBox.Show("Bitte Daten erst vollständig eingeben!");
        return;
    }
    Global.KleveKunden.Add(new Kunde(tbxNachname.Text + " ",
    "+tbxVorname.Text,tbxAdresse.Text));
}
```

```
private void frmKundenverwaltung_Load(object sender, EventArgs e)
{
    dgvKunden.DataSource = Global.KleveKunden;
}
```

Nun ist die Anwendung schon funktionstüchtig. Hier kann man auch die Leistungsfähigkeit des DataGridViews erkennen. In das Feld KundenId kann nichts geschrieben werden, da in der zugehörigen Klasse nur ein Lesezugriff gestattet wird. Der Name und die Adresse lassen sich aber überschreiben und sie werden auch direkt in der gebundenen Liste geändert. Dies kann man daran erkennen, dass wenn man das Fenster zur Kontenverwaltung aufruft die Änderung bereits in der Combobox angezeigt wird.

Interessant ist, dass man bei der Voreinstellung auch direkt einen Kunden löschen kann. **Alle abhängigen Konten, in denen dieser Kunde als Kontoinhaber eingetragen ist, werden dann auch gelöscht!** Will man dies nicht zulassen, so kann man diese Funktion ausschließen. Dann muss die Property AllowUsersToDeleteRows auf false eingestellt werden.

Nun soll noch im Folgenden das Formular für die Kontenverwaltung erweitert werden, um ebenfalls Überweisungen tätigen zu können. Außerdem sollen alle Kontobewegungen der Klasse in einer Liste zu jedem Konto verwaltet werden (Komposition). Hierzu ist es aber notwendig, dass auch die Kontenliste global für alle Formulare bekannt ist. Daher wird die Klasse Global nun auch um die Property Klevekunden und einer entsprechenden Init-Methode erweitert. Auch die Berücksichtigung im weiteren Code ist analog zum Kundenbeispiel. (Vollständiger Code s. Anhang II).

7.3.4 Implementierung von Aggregationen und Kompositionen

Technisch gibt es bei der Umsetzung von Aggregationen in C# keinen nennenswerten Unterschied zur Umsetzung von Kompositionen, darum wird hierauf nicht näher eingegangen. Hingegen ist bei der Komposition zu beachten, dass die Lebensdauer der Teile an das Ganze gekoppelt ist.

Zu 2. Umsetzung der Komposition zwischen Konto und Kundenbewegung

OK

Bei Kompositionen ist zu beachten, dass die abhängigen Objekte in der zugeordneten Klasse gebildet werden. Die Löschung des Gesamtobjekts hat die Löschung aller abhängigen Objekte zu Folge. D. h. die Lebensdauer des Gesamtobjekts begrenzt ebenfalls die Lebensdauer der abhängigen Objekte.

Dies wird realisiert, in dem die Objekte der abhängigen Klasse im Konstruktor des Gesamtobjekts erzeugt werden.

(Die Alternative mit inneren Klassen zu arbeiten wird in der Praxis nicht oder nur selten verwendet.

Golo Roden, Autor des Buches „Auf der Fährte von C#“ schreibt hierzu: „Verschachtelte Klassen hingegen ermöglichen, innerhalb einer Klasse eine weitere Klasse zu definieren, genau so, wie auch innerhalb eines Namensraumes ein weiterer Namensraum angelegt werden kann. Im Gegensatz zu Namensräumen ist dies bei Klassen in der Praxis jedoch unüblich, zudem gibt es so gut wie keine Anwendungsfälle, in denen ein solches Verfahren notwendig wäre, weshalb darauf nicht näher eingegangen wird.“)

Im Folgenden ist der Code der Klasse Kontobewegung sowie der Ausschnitt der Klasse Konto, der sich auf die Kontobewegungen bezieht, angegeben.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KontoVererbung
{
    public class Kontobewegung
    {
        private static int sTransaktionsNr = 10000;
        private int fTransaktionsNr;

        public int TransaktionsNr
        {
            get { return fTransaktionsNr; }
            set { fTransaktionsNr = value; }
        }
        private double fBetrag;

        public double Betrag
        {
            get { return fBetrag; }
            set { fBetrag = value; }
        }
        private BewegungsArt fArt;
```

Bewegungsart ist eine Aufzählung.
Alternativ kann hier auch der Datentyp String verwendet werden.

```
public enum BewegungsArt
{
    Einzahlen = 0,
    Auszahlen = 1,
    Ueberweisen = 2,
}
```

```

public BewegungsArt Art
{
    get { return fArt; }
    set { fArt = value; }
}
private DateTime fDatum;

public DateTime Datum
{
    get { return fDatum; }
    set { fDatum = value; }
}
private string fBemerkung;

public string Bemerkung
{
    get { return fBemerkung; }
    set { fBemerkung = value; }
}

public Kontobewegung(BewegungsArt pArt, double pBetrag, DateTime pDatum, string
pBemerkung )
{
    fTransaktionsNr = sTransaktionsNr++;
    fArt = pArt;
    fBetrag = pBetrag;
    fDatum = pDatum;
    fBemerkung = pBemerkung;
}
}
}

```

```

public abstract class Konto
{
    protected KontobewegungsListe fKontobewegungen;

    public KontobewegungsListe Kontobewegungen
    {
        get { return fKontobewegungen; }
    }

    public Konto(long pKontonummer, Kunde pKontoinhaber, double pSaldo)
    {
        fKontonummer = pKontonummer;
        Kontoinhaber = pKontoinhaber;
        fSaldo = pSaldo;
        fKontobewegungen = new KontobewegungsListe();
        fEroeffnungsSaldo = pSaldo;
    }

    public abstract void Auszahlen(double pBetrag);

    public void Einzahlen(double pBetrag)
    {
        fSaldo = fSaldo + Math.Max(pBetrag, 0);
    }
}

```

Die Ergänzungen der Kontobewegungen werden in der Methode Auszahlen() der Klassen Girokonto und Sparkonto berücksichtigt

```

        fKontobewegungen.Add(new Kontobewegung(BewegungsArt.Einzahlen,
pBetrag, DateTime.Now, ""));
    }
    public void Ueberweisen(double pBetrag, ref Konto pzielKonto)
    {
        try
        {
            this.Auszahlen(pBetrag);
            pzielKonto.Einzahlen(pBetrag);
            fKontobewegungen.Last().Art=BewegungsArt.Ueberweisen;
            fKontobewegungen.Last().Bemerkung="Überweisung nach
"+pzielKonto.fKontonummer.ToString();
            //
            pzielKonto.fKontobewegungen.Last().Art =
BewegungsArt.Ueberweisen;
            pzielKonto.fKontobewegungen.Last().Bemerkung = "Überweisung von "
+ this.fKontonummer.ToString();

        }
        catch
        {
            throw new Exception("Überweisung fehlgeschlagen, da Betrag nicht
gedeckt");
        }
    }
}

```

Die drei Methoden Einzahlen, Auszahlen und Überweisen wurden so erweitert, dass jede Kontobewegung auch in der Liste Kontobewegungen dokumentiert wird. Da Überweisen im Wesentlichen eine Hintereinander-Ausführung der Methoden Einzahlen() und Auszahlen() ist, wird nach der Durchführung die Dokumentation noch einmal überarbeitet. Jeweils bei der letzten Bewegung des Kontos bzw. Zielkontos wird die Bewegungsart auf „Überweisen“ umgestellt und die Bemerkung um das Ausgangs- bzw. Zielkonto ergänzt.

Neu ist hier auch die Verwendung des Schlüsselwortes **ref** bei dem Aufrufparameter pZielkonto der Ueberweisen()-Methode. Werden Parameter mit diesem Schlüsselwort übergeben, so kann der Parameter in der Methode geändert werden. Während normalerweise Änderungen von Parametern nach Beenden der Methode wirkungslos sind, bleiben bei ref-Parametern die Änderungen auch nach Beenden der Methode bestehen.

```

public class KontobewegungsListe:BindingList<Kontobewegung>
{
}

```

Um nun Alles testen zu können wird die Oberfläche um ein einfaches Formular frmKontobewegungen erweitert. Hier kann ein Konto ausgewählt werden und alle hierzu erstellten Kontobewegungen werden in einem DataGridView angezeigt.

In dem Formular der Kontenverwaltung werden die Kontobewegungen um das Überweisen erweitert.

Formular Kontenverwalten (vor Betätigung des Buttons „Ausführen“)

frmMDI - [Form1]

Kunden verwalten Konten verwalten Kontenübersichten schließen

Sparkasse Kleve

Kontoart
☒ Girokonto ☐ Sparkonto

Konto anlegen
Kontonummer:
Kontoinhaber: 1000: Mueller, Hans
Saldo:
Kreditlimit:

Transaktion
Kontonummer: 5005875
Betrag: 175
Transaktionsart: überweisen
Zielkonto: 3004813

	Kontonummer	Kontoinhaber	Saldo
	3103901	1000: Mueller, H...	2000
	3004813	1002: Schmidt, H...	5000
▶	5005875	1000: Mueller, H...	1000
	5005612	1001: Schmidt, L...	2000
	7205820	1004: Jansen, Gr...	3000

Formular Kontenverwalten (nach Betätigung des Buttons „Ausführen“)

frmMDI - [Form1]

Kunden verwalten Konten verwalten Kontenübersichten schließen

Sparkasse Kleve

Kontoart
☒ Girokonto ☐ Sparkonto

Konto anlegen
Kontonummer:
Kontoinhaber: 1000: Mueller, Hans
Saldo:
Kreditlimit:

Transaktion
Kontonummer: 5005875
Betrag: 175
Transaktionsart: überweisen
Zielkonto: 3004813

	Kontonummer	Kontoinhaber	Saldo
	3103901	1000: Mueller, H...	2000
	3004813	1002: Schmidt, H...	5175
▶	5005875	1000: Mueller, H...	825
	5005612	1001: Schmidt, L...	2000
	7205820	1004: Jansen, Gr...	3000

Formular Kontobewegungen (nach mehrmaligem Ausführen von Kontobewegungen)

frmMDI - [frmKontoBewegungen]

Kunden verwalten Konten verwalten Kontenübersichten schließen

Konto wählen

	TransaktionsNr	Betrag	Art	Datum	Bemerkung
▶	10000	-175	Ueberweisen	21.03.2012 00:33	Überweisung nach 3004813
	10002	113	Einzahlen	21.03.2012 00:36	
	10003	-222	Auszahlen	21.03.2012 00:36	
	10004	-222	Auszahlen	21.03.2012 00:36	
	10005	-335,27	Ueberweisen	21.03.2012 00:36	Überweisung nach 7205820
	10008	1200	Ueberweisen	21.03.2012 00:38	Überweisung von 3103901

7.3.5 Exkurs: Weitere Alternativen zur Berücksichtigung verschiedener Leserichtungen am Beispiel des Kontoinhabers

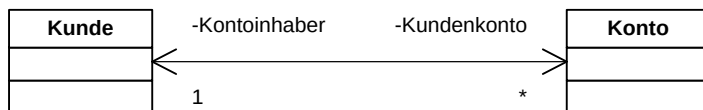


Das oben angegebene Klassendiagramm sagt aus, dass jedes Objekt obj der KlasseA mit beliebig vielen (Multiplizität *) Objekten der KlasseB in Beziehung steht, die die angegebene Rolle für obj spielen.¹⁶

Grundsätzlich gilt für die Realisierung in C#:

1. Assoziationen werden durch Attribute dargestellt.
2. In der ersten Abbildung ist Rolle der KlasseB Attribut von KlasseA.
3. Ist die zugehörige Multiplizität nicht größer als 1, ist der Typ des Attributs KlasseB.
4. Ist die Multiplizität größer 1, ist der Typ des Attributs eine typisierte Liste¹⁷ von der KlasseB.
5. Insbesondere, wenn eine Assoziation eigene Attribute aufweist ist die Umsetzung der Assoziation über eine eigene Klasse angezeigt (Beispiel: Kontobewegung).

Konkretes Beispiel: Ein Kunde besitzt mehrere Konten (Rolle: Kundenkonto).



Welche Informationen sind nun alle in dieser Beziehung enthalten?

- Ein Kunde kann kein, ein oder mehrere Konten haben.
- Ein Konto hat immer genau einen Kontoinhaber.
- Navigierbarkeit geht in beide Richtungen, d.h. der Kunde kennt seine Konten und das Konto kennt seinen Kontoinhaber.
- Stimmigkeit: Die Konten, die zum Kunden gehören haben alle diesen Kunden als Kontoinhaber eingetragen und umgekehrt.

Prinzipiell können wir in C# für die Umsetzungen von Beziehungen Attribute, Methoden und neue Klassen nutzen (C++ mit der Möglichkeit der Mehrfachvererbung stehen hier noch andere Konzepte zur Verfügung).

Die Informationen, die die Beziehung enthält, werden über geeignete Methoden abgebildet. Schwierig ist die Einhaltung der Stimmigkeit (referentielle Integrität). Änderungen an dem Beziehungsattribut der einen Klasse bedingen Änderungen am Beziehungsattribut der anderen Klasse und umgekehrt.

- Will ich ein Konto in der Liste Kundenkonten ergänzen, so muss dieses Objekt vorhanden sein und als Kontoinhaber eben diesen Kunden eingetragen haben. Wird umgekehrt ein Kontoinhaber geändert, dann muss er aus den Kundenkonten des alten Besitzers gelöscht und in der des neuen Besitzers eingefügt werden.
- Man erkennt, dass diese Art der Implementierung aufwändig ist. Daher sollte man die beidseitig navigierbaren Assoziationen nur da verwenden, wo man auf eine schnelle Navigation auf beiden Seiten angewiesen ist.

¹⁶ Details zu Klassendiagrammen mit ausführlichen Erläuterungen finden Sie im UML-Skript.

¹⁷ Hier wird nur der Weg über typisierte Listen gewählt. Je nach Anwendungsfall sind andere Datenstrukturen noch besser geeignet wie queue, stack, set, ...).

- Pro: schnelle Navigation in beiden Richtungen.
- Contra: Änderungen sind aufwändig zu implementieren, Informationen werden redundant gespeichert.

Die beiden Möglichkeiten zur Änderung einer Assoziation werden in den folgenden Sequenzdiagrammen verdeutlicht:

Hinzufügen eines Kontos (analog: Entfernen eines Kontos)

UML

Ändern des Kontoinhabers eines Kontos:



Problem:

An den Sequenzdiagrammen erkennt man gut, dass bei der Umsetzung die Gefahr einer Endlosschleife droht. SetKontoinhaber und KontoAdd rufen sich wechselseitig auf!
Im Folgenden findet sich eine mögliche Umsetzung für das gegebene Beispiel mit beidseitiger Navigation:

```
public abstract class Konto
{
    protected double fKontostand;

    public double Kontostand
    {
        get { return fKontostand; }
        set { fKontostand = value; }
    }
    protected int fKontonummer;

    public int Kontonummer
```

```

    {
        get { return fKontonummer; }
    }
    protected DateTime fEroeffnungsdatum;

    public DateTime Eroeffnungsdatum
    {
        get { return fEroeffnungsdatum; }
    }
    protected Kunde fKontoinhaber;

    public Kunde Kontoinhaber
    {
        get
        {
            return fKontoinhaber;
        }
        set
        {
            if (this.Kontoinhaber != value)
                // neuer Kontoinhaber verschieden vom bisherigen Kontoinhaber?
            {
                if (this.Kontoinhaber != null)
                {
                    Kunde InhaberVorher = this.Kontoinhaber; //
                    // Zwischenspeichern
                    this.Kontoinhaber = null;
                    // Verhinderung der Endlosschleife
                    InhaberVorher.Kundenkonten.Remove(this);
                    // Abmelden aus bisheriger Kontenliste
                }
                this.Kontoinhaber = value; // neuer Kunde zugewiesen
            }
            value.Kundenkonten.Add(this);
        }
    }

    public abstract Boolean Auszahlen(double pBetrag);

    public void Einzahlen(double pBetrag)
    {
        Kontoinhaber.Kundenkonten.Remove(this);
        Kontostand += Math.Max(pBetrag, 0);
        Kontoinhaber.Kundenkonten.Add(this);
    }
}

public class Kunde
{
    private int fKundennummer;
    private static int fLetzteKundennummer;
    private string fNachname;
    private string fVorname;

    private List<Konto> fKundenkonten;

    public List<Konto> Kundenkonten
    {
        get { return fKundenkonten; }
        set { fKundenkonten = value; }
    }
}

```

```

        public void KontoAdd(ref Konto neuesKonto)
        {
            //ref kennzeichnet einen Referenzparameter. Dies sind solche Werte,
            //die an die Methode übergeben werden, aber andererseits bleiben Änderungen an
            //diesen Werten über die Methode hinaus erhalten.
            neuesKonto.Kontoinhaber = this;
            Kundenkonten.Add(neuesKonto);
        }
    }
}

```

Wie zu Beginn dieses Infoteils bereits beschrieben wird die Implementierung einfacher, wenn man sich auf eine einseitige direkte Navigierung beschränkt und die andere Richtung indirekt nutzt. Im Beispiel hieße das, dass jedes Konto seinen Kontoinhaber als Attribut hat. Außerdem gibt es die Verwaltungsklasse Kontenliste, die alle Konten enthält. Von einem Kunden aus geht dann eine Anfrage an alle Konten (Kontenliste), welche Elemente ihn selbst als Kontoinhaber eingetragen haben.

this.SucheMeineKonten(Kontenliste obj)

Diese Art der Implementierung lässt sich gut anwenden, falls eine Navigationsrichtung sehr viel häufiger benötigt wird als die andere. Sie hat nicht das Redundanzproblem der zuerst vorgestellten Lösung. Änderungen sind nun schneller und unkompliziert. Navigationen entgegen der Hauptrichtung sind allerdings sehr langsam.

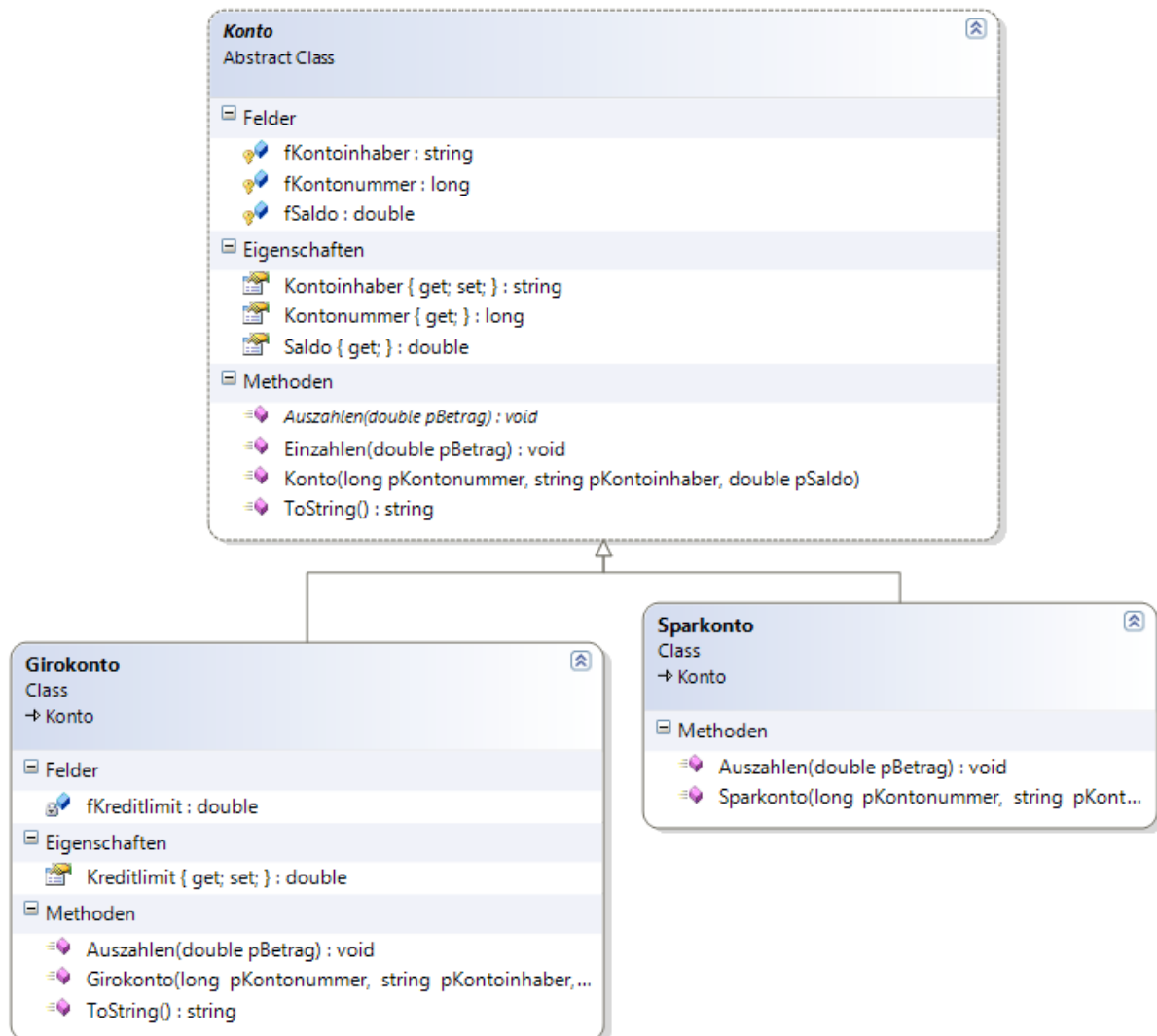
Als letzte Möglichkeit soll noch auf die Realisierung der Assoziation als eigene Assoziationsklasse eingegangen werden.

Für jede Verbindung zwischen den Objekten beider Klassen Kunde und Konto wird ein Objekt der Assoziationsklasse angelegt. Navigiert wird durch Nachfrage des aufrufenden Objektes durch alle angelegten Assoziationsobjekte und Selektion der jeweils zugehörigen.

Vorteilhaft an dieser Umsetzung ist, dass die Datenhaltung automatisch konsistent ist und dass nur tatsächlich existierende Verbindungen gespeichert werden. Die Struktur der beteiligten Klassen muss nicht verändert werden. Nachteilig ist, dass hier die Navigation ausgesprochen langwierig ist.

Anhang I (Vollständige Lösung des Konto-Beispiels mit Vererbung)

Klassendiagramm



Grafische Benutzeroberfläche (GUI)

Form1

Sparkasse Kleve

Kontoart

☒ Girokonto ☐ Sparkonto

Konto anlegen

Kontonummer

Kontoinhaber

Saldo

Kreditlimit

Ein-/Auszahlungen

Betrag

	Kontonummer	Kontoinhaber	Saldo
▶	3103901	Mueller, Hans	2000
	3004813	Schmidt, Heinz	5000
	5005875	Mueller, Hans	1000
	5005612	Schmidt, Lola	2000

Form1

Sparkasse Kleve

Kontoart

☒ Girokonto ☐ Sparkonto

Konto anlegen

Kontonummer

Kontoinhaber

Saldo

Kreditlimit

Ein-/Auszahlungen

Betrag

	Kontonummer	Kontoinhaber	Saldo	Kreditlimit
▶	5005875	Mueller, Hans	1000	6000
	5005612	Schmidt, Lola	2000	5000

Konto.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KontoVererbung
{
    public abstract class Konto
    {
        protected long fKontonummer;

        public long Kontonummer
        {
            get { return fKontonummer; }
        }

        protected string fKontoinhaber;

        public string Kontoinhaber
        {
            get { return fKontoinhaber; }
            set { fKontoinhaber = value; }
        }

        protected double fSaldo;

        public double Saldo
        {
            get { return fSaldo; }
        }

        public Konto(long pKontonummer, string pKontoinhaber, double pSaldo)
        {
            fKontonummer = pKontonummer;
            Kontoinhaber = pKontoinhaber;
            fSaldo = pSaldo;
        }

        public override string ToString()
        {
            string ausgabeText = "";
            string neueZeile = "\n";
            ausgabeText += String.Format("{0, -13} {1, -15} {2,10:F2}€ ",
                Kontonummer, Kontoinhaber, Saldo) + neueZeile;
            return ausgabeText;
        }

        public abstract void Auszahlen(double pBetrag);

        public void Einzahlen(double pBetrag)
        {
            fSaldo = fSaldo + Math.Max(pBetrag, 0);
        }
    }
}
```

Sparkonto.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KontoVererbung
{
    public class Sparkonto:Konto
    {

        public Sparkonto(long pKontonummer, string pKontoinhaber, double
pSaldo):base(pKontonummer,pKontoinhaber, pSaldo)
        {

        }

        public override void Auszahlen(double pBetrag)
        {
            if (pBetrag > Math.Round(fSaldo+1, 2))
            {
                throw new Exception("Auszahlung in Höhe von " + pBetrag + " €
nicht möglich."
                + "\n" + "Maximal möglicher Auszahlungsbetrag: " +
(Math.Round(Saldo- 1, 2)) + " € ");
            }
            else
            {
                fSaldo = fSaldo - Math.Max(pBetrag, 0);
            }
        }
    }
}
```

Girokonto.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace KontoVererbung
{
    public class Girokonto:Konto
    {

        private double fKreditlimit;

        public double Kreditlimit
        {
            get { return fKreditlimit; }
            set { fKreditlimit = value; }
        }
    }
}
```

```

        public Girokonto(long pKontonummer, string pKontoinhaber, double pSaldo,
double pKreditlimit):base(pKontonummer,pKontoinhaber,pSaldo)
        {

            Kreditlimit = pKreditlimit;
        }

        public override string ToString()
        {
            string ausgabeText = "";
            string neueZeile = "\n";
            ausgabeText += String.Format("{0, -13} {1, -15} {2,10:F2}€ {3,10:F2}€",
                Kontonummer, Kontoinhaber, Saldo, Kreditlimit) + neueZeile;
            return ausgabeText;
        }
        /// <summary>
        ///
        /// </summary>
        /// <param name="pBetrag"></param>
        public override void Auszahlen(double pBetrag)
        {
            if (pBetrag > Math.Round((fSaldo + fKreditlimit),2))
            {
                throw new Exception("Auszahlung in Höhe von " + pBetrag + " € nicht
möglich."
                + "\n" + "Maximal möglicher Auszahlungsbetrag: " + (Math.Round(Saldo +
Kreditlimit,2)) + " € ");
            }
            else
            {
                fSaldo = fSaldo - Math.Max(pBetrag, 0);
            }
        }
    }
}

```

Kontenliste.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.ComponentModel;

namespace KontoVererbung
{
    class KontenListe:BindingList<Konto>
    {
        public int KontoSuche(long pKontonummer)
        {
            for (int i = 0; i < this.Count; i++)
            {
                if (this[i].Kontonummer == pKontonummer)
                {
                    return i;
                }
            }
            return -1;
        }
    }
}

```

```

    }
    public void AddEindeutigesKonto(Konto kto)
    {
        if (this.KontoSuche(kto.Kontonummer) >=0)
        {
            throw new Exception("Konto kann nicht eingefügt werden, da Kontonummer  
bereits vorhanden!");
        }
        else
        {
            this.Add(kto);
        }
    }
    public GiroKontenListe GirokontenSelektieren()
    {
        GiroKontenListe liste = new GiroKontenListe();
        foreach (Konto kto in this)
        {
            if (kto is Girokonto)
            {
                liste.Add((Girokonto) kto);
            }
        }
        return liste;
    }
    public SparkontenListe SparkontenSelektieren()
    {
        SparkontenListe liste = new SparkontenListe();
        foreach (Konto kto in this)
        {
            if (kto is Sparkonto)
            {
                liste.Add((Sparkonto) kto);
            }
        }
        return liste;
    }
}
}

```

GirokontenListe.cs

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
// der Namensraum System.ComponentModel enthält die BindingList
using System.ComponentModel;

namespace KontoVererbung
{
    public class GiroKontenListe:BindingList<Girokonto>
    {

    }
}

```

SparkontenListe.cs

Analog!

frmKonto.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class Form1 : Form
    {
        public Form1()
        {
            InitializeComponent();

            KontenListe klevKonten;

            private void Form1_Load(object sender, EventArgs e)
            {
                //
                klevKonten = new KontenListe();

                klevKonten.AddEindeutigesKonto(new Sparkonto(3103901, "Mueller, Hans",
2000.00));
                klevKonten.AddEindeutigesKonto(new Sparkonto(3004813, "Schmidt, Heinz",
5000.00));
                // klevKonten.Add(new Sparkonto(3103901,"Meier, Sabine",0.00));
                klevKonten.AddEindeutigesKonto(new Girokonto(5005875, "Mueller, Hans",
1000.00, 6000.00));
                klevKonten.AddEindeutigesKonto(new Girokonto(5005612, "Schmidt, Lola",
2000.00, 5000.00));
                //klevKonten.AddEindeutigesKonto(new Girokonto(7205820, "Jansen, Gregor",
3000.00, 0.0));
                dgvKonten.DataSource = klevKonten;
            }

            private void btnAnlegen_Click(object sender, EventArgs e)
            {
                try
                {
                    long kontonummer = Convert.ToInt64(tbxKontonummer.Text);
                    string kontoinhaber = tbxKontoinhaber.Text;
                    double saldo = Convert.ToDouble(tbxSaldo.Text);
                    if (rbGiro.Checked)
                    {
                        double kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
                        klevKonten.AddEindeutigesKonto(new Girokonto(kontonummer,
kontoinhaber, saldo, kreditlimit));
                    }
                    else
                    {
                        klevKonten.AddEindeutigesKonto(new Sparkonto(kontonummer,
kontoinhaber, saldo));
                    }
                }
                catch (Exception ex)
            }
        }
    }
}
```

```

        {
            MessageBox.Show("Eingabefehler: " + ex.ToString());
        }
    }

private void rbGiro_CheckedChanged(object sender, EventArgs e)
{
    if (rbGiro.Checked)
    {
        tbxKreditlimit.Enabled = true;
        dgvKonten.Refresh();
    }
    else
    {
        tbxKreditlimit.Enabled = false;
        tbxKreditlimit.Text = 0.ToString();
        dgvKonten.Refresh();
    }
}

private void btnEinzahlen_Click(object sender, EventArgs e)
{
    try
    {
        long ktonr = Convert.ToInt64(tbxKontonummer.Text);
        double betrag = Convert.ToDouble(tbxBetrag.Text);
        int index = kleveKonten.KontoSuche(ktonr);
        if (index > -1)
        {
            kleveKonten[index].Einzahlen(betrag);
            dgvKonten.Refresh();
        }

        else
        {
            throw new Exception("Kontonummer nicht im System");
        }
    }

    catch (Exception ex)
    {
        MessageBox.Show("Eingabefehler: " + ex.ToString());
    }
}

private void btnAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = kleveKonten;
    dgvKonten.Refresh();
}

private void btnAuszahlen_Click(object sender, EventArgs e)
{
    try
    {
        long ktonr = Convert.ToInt64(tbxKontonummer.Text);
        double betrag = Convert.ToDouble(tbxBetrag.Text);
        int index = kleveKonten.KontoSuche(ktonr);
        if (index > -1)
        {
            kleveKonten[index].Auszahlen(betrag);
        }
    }
    catch (Exception ex)
    {
        MessageBox.Show("Eingabefehler: " + ex.ToString());
    }
}

```

```

        dgvKonten.Refresh();
    }

    else
    {
        throw new Exception("Kontonummer nicht im System");
    }
}

catch (Exception ex)
{
    MessageBox.Show("Eingabefehler: " + ex.ToString());
}
}

private void btnGiroAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = kleveKonten.GirokontenSelektieren();
}

private void btnSparAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = kleveKonten.SparkontenSelektieren();
}
}
}

```

Anhang II (Lösung des Konto-Beispiels mit Assoziationen)

Angegeben wird hier der Formularcode, Die wesentlichen Klassen wurden bereits im Infoteil angegeben.

Formular frmMDI

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class frmMDI : Form
    {
        public frmMDI()
        {
            InitializeComponent();
        }

        private void kontenverwaltenToolStripMenuItem_Click(object sender, EventArgs e)
        {
            frmKontenVerwalten frm = new frmKontenVerwalten();
            frm.MdiParent = this;
            frm.WindowState = FormWindowState.Maximized;
            frm.Show();
        }

        private void schliesßenToolStripMenuItem_Click(object sender, EventArgs e)
        {
            this.Close();
        }

        private void kundenVerwaltenToolStripMenuItem_Click(object sender, EventArgs e)
        {
            frmKundenverwaltung frm = new frmKundenverwaltung();
            frm.MdiParent = this;
            frm.WindowState = FormWindowState.Maximized;
            frm.Show();
        }

        private void kontenübersichtenToolStripMenuItem_Click(object sender, EventArgs e)
        {
            frmKontoBewegungen frm = new frmKontoBewegungen();
            frm.MdiParent = this;
            frm.WindowState = FormWindowState.Maximized;
            frm.Show();
        }
    }
}
```

Datei Program.cs

```
namespace KontoVererbung
{
    static class Program
    {
        /// <summary>
        /// Der Haupteinstiegspunkt für die Anwendung.
        /// </summary>
        [STAThread]
        static void Main()
        {
            Application.EnableVisualStyles();
            Application.SetCompatibleTextRenderingDefault(false);
            Global.InitKleveKunden();
            Global.InitKleveKonten();
            Application.Run(new frmMDI());
        }
    }
}
```

Formular frmKundenverwaltung

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class frmKundenverwaltung : Form
    {
        public frmKundenverwaltung()
        {
            InitializeComponent();
        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
        }

        private void btnAnlegen_Click(object sender, EventArgs e)
        {
            if (tbxAdresse.Text == "" | tbxNachname.Text == "")
            {
                MessageBox.Show("Bitte Daten erst vollständig eingeben!");
                return;
            }
            Global.KleveKunden.Add(new Kunde(tbxNachname.Text + ", " +
            tbxVorname.Text, tbxAdresse.Text));
        }

        private void frmKundenverwaltung_Load(object sender, EventArgs e)
        {
            dgvKunden.DataSource = Global.KleveKunden;
        }
    }
}
```

Formular frmKontenverwaltung

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class frmKontenVerwalten : Form
    {
        public frmKontenVerwalten()
        {
            InitializeComponent();

            private void label4_Click(object sender, EventArgs e)
            {
            }

            private void textBox3_TextChanged(object sender, EventArgs e)
            {
            }

            private void frmKontenVerwalten_Load(object sender, EventArgs e)
            {
                cbxTransaktionsart.SelectedIndex = 0;

                dgvKonten.DataSource = Global.KleveKonten;
                cbxKunden.DataSource = Global.KleveKunden;
                cbxKonten.DataSource = Global.KleveKonten;
                cbxKonten.DisplayMember = "Kontonummer";

                //
            }

            private void btnAnlegen_Click(object sender, EventArgs e)
            {
                try
                {
                    long kontonummer = Convert.ToInt64(tbxKontonummer.Text);
                    Kunde kontoinhaber = (Kunde) cbxKunden.SelectedItem;
                    double saldo = Convert.ToDouble(tbxSaldo.Text);
                    if (rbGiro.Checked)
                    {
                        double kreditlimit = Convert.ToDouble(tbxKreditlimit.Text);
                        Global.KleveKonten.AddEindeutigesKonto(new Girokonto(kontonummer,
                        kontoinhaber, saldo, kreditlimit));
                    }
                    else
                    {
                        Global.KleveKonten.AddEindeutigesKonto(new Sparkonto(kontonummer,
                        kontoinhaber, saldo));
                    }
                }
                catch { }
            }
        }
    }
}
```

```

    }
}
catch (Exception ex)
{
    MessageBox.Show("Eingabefehler: " + ex.ToString());
}
}

private void rbGiro_CheckedChanged(object sender, EventArgs e)
{
    if (rbGiro.Checked)
    {
        tbxKreditlimit.Enabled = true;
        dgvKonten.Refresh();
    }
    else
    {
        tbxKreditlimit.Enabled = false;
        tbxKreditlimit.Text = 0.ToString();
        dgvKonten.Refresh();
    }
}

private void btnAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = Global.KleveKonten;
    dgvKonten.Refresh();
}

private void btnGiroAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = Global.KleveKonten.GirokontenSelektieren();
}

private void btnSparAnzeigen_Click(object sender, EventArgs e)
{
    dgvKonten.DataSource = Global.KleveKonten.SparkontenSelektieren();
}

private void cbxTransaktionsart_SelectedIndexChanged(object sender, EventArgs
e)
{
    if (cbxTransaktionsart.SelectedIndex <= 1)
    {
        lblZielkonto.Visible = false;
        cbxZielkonto.Visible = false;
    }
    else
    {
        lblZielkonto.Visible = true;
        cbxZielkonto.Visible = true;
        cbxZielkonto.Items.Clear();
        foreach (Konto item in Global.KleveKonten)
        {
            cbxZielkonto.Items.Add(item.Kontonummer);
        }
    }
}
}

```

```

private void btnAusfuehren_Click(object sender, EventArgs e)
{
    try
    {
        double betrag = Convert.ToDouble(tbxBetrag.Text);
        if (cbxTransaktionsart.SelectedIndex == 0)
        {
            ((Konto)cbxKonten.SelectedItem).Einzahlen(betrag);
        }
        else if (cbxTransaktionsart.SelectedIndex == 1)
        {
            ((Konto)cbxKonten.SelectedItem).Auszahlen(betrag);
        }
        else if (cbxTransaktionsart.SelectedIndex == 2)
        {
            int index = Global.KleveKonten.KontoSuche((long)
cbxZielkonto.SelectedItem);
            if (index < 0)
            {
                MessageBox.Show("Zielkonto nicht vorhanden");
                return;
            }
            else
            {
                Konto zielkonto = Global.KleveKonten[index];
                ((Konto)cbxKonten.SelectedItem).Ueberweisen(betrag, ref
zielkonto);
            }
        }
        dgvKonten.Refresh();
    }
    catch (Exception ex)
    {
        MessageBox.Show("Fehler: " + ex.ToString());
    }
}
}
}

```

Formular frmKundenverwaltung

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class frmKundenverwaltung : Form
    {
        public frmKundenverwaltung()
        {
            InitializeComponent();
        }

        private void textBox1_TextChanged(object sender, EventArgs e)
        {
        }

        private void btnAnlegen_Click(object sender, EventArgs e)
        {
            if (tbxAdresse.Text == "" || tbxNachname.Text == "")
            {
                MessageBox.Show("Bitte Daten erst vollständig eingeben!");
                return;
            }
            Global.KleveKunden.Add(new Kunde(tbxNachname.Text + ", " +
            tbxVorname.Text, tbxAdresse.Text));
        }

        private void frmKundenverwaltung_Load(object sender, EventArgs e)
        {
            dgvKunden.DataSource = Global.KleveKunden;
        }
    }
}
```

Formular frmKontenBewegungen

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Windows.Forms;

namespace KontoVererbung
{
    public partial class frmKontoBewegungen : Form
    {
        public frmKontoBewegungen()
        {
            InitializeComponent();

            private void frmKontoBewegungen_Load(object sender, EventArgs e)
            {
                cbxKonten.DataSource = Global.KleveKonten;
                cbxKonten.DisplayMember = "Kontonummer";
            }

            private void cbxKonten_SelectedIndexChanged(object sender, EventArgs e)
            {
                dgvKontoBewegungen.DataSource =
                ((Konto)cbxKonten.SelectedItem).Kontobewegungen;
            }
        }
    }
}
```

8. Anhang I

Steuerelement	Beispiel
frm = Form (Formular)	frmBuchung
btn = Button (Schaltfläche)	btnAnzeigen
lbl = Label (Beschriftungsfeld)	lblKontonummer
tbx = TextBox (Textfeld)	tbxKontonummer
pbx = PictureBox (Bildfeld)	pbxLogo
chk = CheckBox (Kontrollkästchen)	chkKontotyp
tmr = Timer (Zeitgeber)	tmrUhr
lbx = Listbox (Listenfeld)	lbxKunde
cbx = ComboBox (Kombinationsfeld)	cbxKonto
dgv = DataGridView (Datentabelle)	dgvKonten