Objektorientiertes Programmieren

- Kleine Einführung in OOP
- Eigenschaften, Methoden und Ereignisse
- Konstruktor und Destruktor
- Vererbung und Polymorphie
- Auflistungsklassen

C# erlaubt Ihnen, bereits ohne fundierte OOP-Kenntnisse objektorientiert zu programmieren! Davon haben Sie bereits vor der Lektüre dieses Kapitels, mehr oder weniger unbewusst, Gebrauch gemacht: Sie haben Ereignisbehandlungsroutinen (Event-Handler) geschrieben und den Objekten der visuellen Benutzerschnittstelle (Form, Steuerelemente) Eigenschaften zugewiesen bzw. deren Methoden aufgerufen. Wer sich noch einmal die grundlegende Vorgehensweise bei der objekt- und ereignisorientierten Windows-Programmierung ins Gedächtnis zurückrufen will, der sei auf den Abschnitt 1.3 des Einführungskapitels verwiesen.

Die Entwicklungsumgebung VS.NET erlaubt objektorientiertes Programmieren bereits mit einem Minimum an Vorkenntnissen. Das vorliegende Kapitel will etwas tiefer in die OOP-Problematik eindringen und präsentiert Ihnen neben einigen grundlegenden Ausführungen die für den Einstieg wichtigsten objektspezifischen Features von C# im Überblick.

Hinweis: In C# haben Sie grundsätzlich die Möglichkeit, zwischen Klassen und Strukturen zu wählen. Letztere wurden bereits im Sprachkapitel (Abschnitt 3.6) einführend behandelt, bieten allerdings noch weitaus mehr Möglichkeiten, die fast an die von Klassen heranreichen. Wir aber wollen uns im vorliegenden Kapitel ausschließlich mit Klassen beschäftigen.

4.1 Eine kleine Einführung in die OOP

In .NET ist alles ein Objekt! Viele Entwickler – insbesondere wenn sie mit "altem" Code zu kämpfen haben – tun sich noch ziemlich schwer mit OOP, weil ihnen die Komplexität einer vollständigen Anwendung noch zu hoch erscheint.

4.1.1 Wozu überhaupt objektorientiert programmieren?

Im Unterschied zur objektorientierten ist die strukturierte Programmierung ziemlich sprachunabhängig und hatte Zeit genug, um auch in den letzten Winkel der Programmierwelt vorzudringen.

Demgegenüber stand es um die Akzeptanz der objektorientierten Programmierung bis Anbruch des .NET-Zeitalters noch nicht zum Besten, das aber dürfte sich jetzt und in Zukunft gewaltig ändern.

Strukturierte Programmierung

Gern bezeichnet man die strukturierte Programmierung auch als Vorläufer der objektorientierten Programmierung, obwohl dieser Vergleich hinkt. Richtig ist, dass sowohl strukturierte als auch objektorientierte Programmierung fundamentale Denkmuster[1] sind, die gleichberechtigt nebeneinander existieren.

---

[1] Im abschreckenden Fachjargon heißt das "Paradigma".

Die Grundkonzepte der strukturierten Programmierung wurden beginnend mit dem Ende der Sechzigerjahre entwickelt und lassen sich mit folgenden Stichwörtern charakterisieren: hierarchische Programmorganisation, logische Programmeinheiten, zentrale Programmsteuerung, beschränkte Datenverfügbarkeit.

Ziel der strukturierten Programmierung ist es, Algorithmen so darzustellen, dass ihr Ablauf einfach zu erfassen und zu verändern ist.

Gegenstand der strukturierten Programmierung ist also die bestmögliche Anordnung von Code, um dessen Transparenz, Testbarkeit und Wiederverwendbarkeit zu maximieren[1].

Dass C# eine konsequent objektorientierte Sprache ist, bedeutet noch lange nicht, dass man damit nicht auch strukturiert programmieren könnte, im Gegenteil. Im Kapitel 3, wo sich alles um die grundlegenden sprachlichen Elemente von C# drehte, haben wir uns fast ausschließlich auf dem Boden der traditionellen strukturierten Programmierung bewegt und versucht, die OOP noch weitestgehend auszuklammern. So haben wir es größtenteils ignoriert, dass selbst die einfachen Datentypen Objekte sind, und haben z.B. anstatt mit Methoden mit Funktionen und Prozeduren und anstatt mit Klassen mit strukturierten Datentypen (*struct*) gearbeitet. Tatsächlich können Sie aber mit OOP alles machen, was auch die strukturierte Programmierung erlaubt.

Beispiel: Anstatt globale Variablen in einem Modul zu deklarieren, können Sie statische Klasseneigenschaften verwenden.

Um fit für die Herausforderungen der Zukunft zu sein, sollten Sie deshalb – wo immer es vertretbar ist – nach objektorientierten Lösungen streben.

Objektorientierte Programmierung

Die objektorientierte Programmierung entfaltete auf breiter Basis erst seit Ende der 80er-Jahre mit dem Beginn des Windows-Zeitalters ihre Wirkung. Sehr bekannte Vertreter objektorientierter Sprachen sind C++, Java, Smalltalk und Borland Delphi – aber auch das alte Visual Basic war bereits in vielen wesentlichen Zügen objektorientiert aufgebaut[2].

Objektorientierte Programmierung ist ein Denkmuster, bei dem Programme als Menge von über Nachrichten kooperierenden Objekten organisiert werden und jedes Objekt Instanz einer Klasse ist.

Im Unterschied zur strukturierten Programmierung bedeutet "objektorientiert" also, dass Daten und Algorithmen nicht mehr nebeneinander existieren, sondern in Objekten zusammengefasst sind.

Während Module in der strukturierten Programmierung zwar auch Daten und Code zusammenfassen, stellen Klassen jetzt Vorlagen dar, von denen immer neue Kopien

---

[1] Bahnbrechendes auf diesem Gebiet leistete Prof. Niklaus Wirth mit seinen Sprachen Pascal und Modula.

[2] Mit VB.NET wurde auch Visual Basic endlich (nahezu) vollwertiges Mitglied der OOP-Welt.

(Instanzen) angefertigt werden können. Diese Instanzen, d.h. die Objekte, kapseln den Zugriff auf die enthaltenen Daten hinter Schnittstellen (Interfaces).

Der große Vorteil der OOP ist ihre Ähnlichkeit mit den menschlichen Denkstrukturen. Dadurch wird vor allem dem Einsteiger, der bisher über keine bzw. wenig Programmiererfahrung verfügt, das Verständnis der OOP erleichtert.

Hinweis: Die OOP verlangt eine Anpassung des Software-Entwicklungsprozesses und der eingesetzten Methoden an den Denkstil des Programmierers – und nicht umgekehrt!

Die OOP ist eine der wenigen Fälle, in denen der Einsteiger gegenüber dem Profi zumindest einen kleinen Vorteil besitzt: Er ist noch nicht in der Denkweise klassischer Programmiersprachen gefangen, die dazu erziehen, in Abläufen zu denken, bei denen die in der realen Welt zu beobachtenden Abläufe Schritt für Schritt in Algorithmen umgesetzt werden, etwa um betriebliche Prozesse per Programm zu automatisieren.

Die OOP entspricht hingegen der üblichen menschlichen Denkweise, indem sie z.B. reale Objekte aus der abzubildenden Umwelt identifiziert und in ihrer Art beschreibt.

4.1.2 Grundbegriffe der OOP

Bereits im Kapitel 1 (Abschnitt 1.3) hatten Sie gesehen, dass Objekte durch Eigenschaften, Methoden und Ereignisse charakterisiert sind. Auf diese und andere Weise überwindet das Konzept der objektorientierten Programmierung (OOP) den prozeduralen Ansatz der klassischen strukturellen Programmierung zugunsten einer realitätsnahen Modellierung.

Bevor wir uns den Details zuwenden, sollen die wichtigsten Begriffe der objektorientierten Programmierung (OOP) zunächst allgemein, d.h. ohne Bezug auf eine konkrete Programmiersprache, erörtert werden.

Objekt

Der Programmierer versteht unter einem *Objekt* die Zusammenfassung (Kapselung) von Daten und zugehörigen Funktionalitäten. Ein solches Softwareobjekt wird auch oft benutzt, um Dinge des täglichen Lebens für Zwecke der Datenverarbeitung abzubilden. Aber das ist nur ein Aspekt, denn Objekte sind ganz allgemein Dinge, die Sie in Ihrem Code beschreiben wollen, es sind Gruppen von Eigenschaften, Methoden und Ereignissen, die logisch zusammengehören. Als Programmierer arbeiten Sie mit einem Objekt, indem Sie dessen Eigenschaften und Methoden manipulieren und auf seine Ereignisse reagieren.

Klasse

Eine *Klasse* ist nicht mehr und nicht weniger als ein "Bauplan", auf dessen Grundlage die Objekte zur Programmlaufzeit erzeugt werden. Gewissermaßen als Vorlage (Prägestempel) für das Objekt legt sie fest, wie das Objekt auszusehen hat und wie es sich verhalten soll. Es

handelt sich bei einer Klasse also um eine reine Softwarekonstruktion, die Eigenschaften, Methoden und Ereignisse eines Objekts definiert.

Hinweis: Oft wird anstatt "Klasse" mit völlig gleichwertiger Bedeutung auch der Begriff "Objekttyp" verwendet.

Instanz

Man erhält erst dann ein konkretes Objekt, wenn man eine *Instanz* einer Klasse bildet. Es lassen sich viele Objekte mit einer einzigen Klassendefinition erzeugen.

Beispiel: Auf dem Montageband werden zahlreiche Auto-Objekte nach ein und denselben Konstruktionsvorschriften für die Klasse "Auto" gebaut. Diesen Vorgang könnte man auch als Bildung von Instanzen der Klasse "Auto" bezeichnen.

Kapselung

Klassen realisieren das Prinzip der *Kapselung* von Objekten, das es ermöglicht, die Implementierung der Klasse (der Code im Inneren) von deren Schnittstelle bzw. Interface (die öffentlichen Eigenschaften, Methoden und Ereignisse) sauber zu trennen. Durch das Verbergen der inneren Struktur werden die internen Daten und einige verborgene Methoden geschützt, sind also von außen nicht zugänglich. Die Manipulation des Objekts kann lediglich über streng definierte, über die Schnittstelle zur Verfügung gestellte öffentliche Methoden erfolgen.

Wiederverwendbarkeit

Klassen ermöglichen die *Wiederverwendbarkeit* von Code. Nachdem eine Klasse geschrieben wurde, können Sie diese an verschiedenen Stellen innerhalb einer Applikation verwenden. Klassen reduzieren somit den redundanten Code einer Anwendung, sie erleichtern außerdem die Wartung des Codes.

Vererbung

Echte Vererbung (*Implementierungsvererbung*) ermöglicht es Klassen zu definieren, die von anderen Klassen abgeleitet werden, wobei nicht nur die Schnittstelle, sondern auch der dahinter liegende Code (die Implementierung) vom Nachkommen übernommen wird.

Da es nun möglich ist, die Implementierung einer Klasse für weitere Klassen als Grundlage zu verwenden, kann man Unterklassen bilden, die alle Eigenschaften und Methoden ihrer Oberklasse (auch oft als Superklasse bezeichnet) erben. Diese Unterklassen können zu den geerbten Eigenschaften neue hinzufügen oder Eigenschaften der Oberklasse verstecken, indem sie diese überschreiben.

Wird von einer solchen Unterklasse ein Objekt erzeugt (also eine Instanz der Unterklasse gebildet), dann dient für dieses Objekt sowohl die Ober- als auch die Unterklasse als "Bauplan".

C# unterstützt das Überschreiben (*Overriding*) von Methoden[1] der Oberklasse mit alternativen Methoden der Unterklasse (siehe Abschnitt 4.6.2).

Polymorphie

OOP macht es möglich, ein und dieselbe Methode für ganz verschiedene Objekte zu verwenden, man nennt dies dann *Polymorphie* (Vielgestaltigkeit). Jedes dieser Objekte kann die Ausführung unterschiedlich realisieren. Für das aufrufende Objekt bleibt der Vorgang trotzdem derselbe.

Beispiel: Die Methode "Beschleunigen" ist in einer "Fahrzeug"-Klasse definiert, welche an die Unterklassen "Auto" und "Fahrrad" vererbt. Es ist klar, dass diese Methoden in beiden Unterklassen überschrieben, d.h. völlig unterschiedlich implementiert werden müssen.

Als Polymorphie, die aufs Engste mit der Vererbung verknüpft ist, kann man also die Fähigkeit von Unterklassen bezeichnen, Eigenschaften und Methoden mit dem gleichen Namen, aber mit unterschiedlichen Implementierungen aufzurufen (siehe Abschnitt 4.6.3).

4.1.3 Sichtbarkeit von Klassen und ihren Mitgliedern

Um die Klasse bzw. ihre Mitglieder (Member, Elemente) gezielt zu verbergen oder offen zu legen, sollten Sie von den Zugriffsmodifizierern Gebrauch machen, die den Gültigkeitsbereich (bzw. die *Sichtbarkeit*) einschränken.

Klassen

Die folgende Tabelle zeigt die möglichen Einschränkungen bei der Sichtbarkeit von Klassen:

Modifizierer	Sichtbarkeit
public	Unbeschränkt. Auch von anderen Assemblierungen aus können Objekte der Klasse erstellt werden.
internal	Nur innerhalb des aktuellen Projekts. Außerhalb des Projekts ist kein Objekt dieser Klasse erstellbar. Gilt als Standard, falls kein Modifizierer vorangestellt wird.
private	Nur innerhalb einer anderen Klasse.

Klassenmitglieder

Die folgende Tabelle zeigt die Zugriffsmöglichkeiten auf die Klassenmitglieder (Member).

[1] Nicht zu verwechseln mit dem Überladen (Overloading) von Methoden (siehe 3.11.7).

Modifizierer	Sichtbarkeit
public	Unbeschränkt.
protected	Innerhalb der Klasse und der daraus abgeleiteten Klassen.
internal	Innerhalb des aktuellen Projekts.
internal protected	Innerhalb des aktuellen Projekts oder der abgeleiteten Klassen.
private	Nur innerhalb der Klasse.

Die Schlüsselwörter *private* und *public* definieren immer die beiden Extreme des Zugriffs. Betrachtet man die jeweiligen Klassen als allein stehend, so reichen diese beiden Zugriffsarten völlig aus. Mit solchen, quasi isolierten, Klassen lassen sich allerdings keine komplexeren Probleme lösen.

Um einzelne Klassen miteinander zu verbinden, benutzen Sie den mächtigen Mechanismus der Vererbung. In diesem Zusammenhang gewinnt die *protected*-Deklaration wie folgt an Bedeutung:

- Da eine abgeleitete Klasse auf die *protected*-Member zugreifen kann, sind diese Member für die abgeleitete Klasse quasi *public*.
- Ist eine Klasse nicht von einer anderen abgeleitet, kann sie nicht auf deren *protected*-Member zugreifen, da diese dann quasi *private* sind.

Mehr zu diesem Thema finden Sie im Abschnitt 4.6 (Vererbung).

4.1.4 Allgemeiner Aufbau einer Klasse

Bevor der Einsteiger seine erste Klasse schreibt, sollte er sich zunächst im Sprachkapitel 3 mit den Strukturen (*struct*, siehe Abschnitt 3.6.2) etwas anfreunden, die in Aufbau und Anwendung starke Ähnlichkeiten zu Klassen aufweisen[1]. Auch im Aufbau von Methoden sollte er sich auskennen (Abschnitt 3.11).

Im Unterschied zu einer Struktur (Schlüsselwort *struct*) wird eine Klasse mit dem Schlüsselwort *class* deklariert. Die (stark vereinfachte) Syntax:

Syntax:
```
Modifizierer class Bezeichner
{
  // ... Felder
  // ... Konstruktoren
  // ... Eigenschaften
  // ... Methoden
  // ... Ereignisse
 }
```

[1] Der wesentliche Unterschied ist, dass Strukturen Wertetypen, Klassen hingegen Referenztypen sind.

Zur Bedeutung der (Zugriffs-)*Modifizierer* wird auf obige Tabellen verwiesen.

Im Klassenkörper haben es wir es mit "Klassenmitgliedern" (Member) wie Feldern, Konstruktoren, Eigenschaften, Methoden und Ereignissen zu tun, auf die wir noch detailliert zu sprechen kommen.

Die Definition der Klassenmitglieder bezeichnet man auch als *Implementation* der Klasse.

Beispiel: Eine einfache Klasse *CKunde* wird deklariert und implementiert.

```
public class CKunde
{
         private string _anrede;          //     Feld
         private string _name;            //     dto.

         public CKunde(string anr, string nam)         // Konstruktor
         {
                  _anrede = anr;
                  _name = nam;
         }

         public string name             // Eigenschaft
         {
                  get {return(_name); }
                  set {_name = value; }
         }

         public string adresse()               // Methode
         {
                  string s = _anrede + " " + _name;
                  return(s);
         }
 }
```

Unsere Klasse verfügt damit über zwei Felder, eine Eigenschaft und eine Methode. Da die beiden Felder mit dem *private*-Modifizierer deklariert wurden, sind sie von außen nicht sichtbar.

4.1.5 Der Lebenszyklus eines Objekts

Existiert eine Klasse, so steht dem Erzeugen von Objektvariablen nichts mehr im Weg. Eine Objektvariable ist ein Verweistyp, sie enthält also nicht das Objekt selbst, sondern stellt lediglich einen Zeiger (Adresse) auf den Speicherbereich des Objekts bereit. Es können sich also durchaus mehrere Objektvariablen auf ein und dasselbe Objekt beziehen. Wenn eine

Objektvariable den Wert *null* enthält, bedeutet das, dass sie momentan "ins Leere" zeigt, also kein Objekt referenziert.

Unter der Voraussetzung, dass eine gültige Klasse existiert, verläuft der Lebenszyklus eines Objekts in Ihrem Programm in folgenden Etappen:

- Referenzierung (eine Objektvariable wird deklariert, sie verweist momentan noch auf *null*)
- Instanziierung (die Objektvariable zeigt jetzt auf einen konkreten Speicherplatzbereich)
- Initialisierung (die Datenfelder der Objektvariablen werden mit Anfangswerten gefüllt)
- Arbeiten mit dem Objekt (es wird auf Eigenschaften und Methoden des Objekts zugegriffen, Ereignisse werden ausgelöst)
- Zerstören des Objekts (das Objekt wird dereferenziert, der belegte Speicherplatz wird wieder freigegeben)

Werfen wir nun einen genaueren Blick auf die einzelnen Etappen.

Referenzieren und Instanziieren

Es stehen zwei Varianten zur Verfügung.

In *Variante 1* sind ausnahmsweise beide Schritte in einer Anweisung zusammengefasst:

Syntax:
```
Modifizierer Klasse Object;
Object = new Klasse(Parameter);
```

Beispiel: Ein Objekt *kunde1* wird referenziert und erzeugt.

```
private CKunde kunde1;                    // Referenzieren
kunde1 = new CKunde1();                   // Erzeugen
```

In *Variante2*, der Kurzform, sind beide Schritte in einer Anweisung zusammengefasst, d.h., das Objekt wird zusammen mit seiner Deklaration erzeugt.

Syntax:
```
Klasse Object = new Klasse();
```

Beispiel: Das Äquivalent zum Vorgängerbeispiel.

```
private CKunde kunde1 = new CKunde();
```

Dem Klassenbezeichner (*Klasse*) müsste genauer genommen noch der Name der Klassenbibliothek (bzw. Name des Projekts) vorangestellt werden, doch dies wird unter VS.NET nicht erforderlich sein, da der entsprechende Namensraum (*Namespace*) bereits automatisch eingebunden wurde (*using*-Anweisung).

Obwohl die Kurzform sehr eindrucksvoll ist, können Sie hier keine Fehlerbehandlung (*try...catch*-Block) durchführen. Diese Einschränkung macht diese Art von Deklaration weniger nützlich.

Empfehlenswert ist also fast immer das getrennte Deklarieren und Erzeugen[1].

Beispiel: Eine mögliche Fehlerbehandlung

```
private CKunde kunde1;
try
{
         kunde1 = new CKunde();
}
catch(Exception ex)
{
         MessageBox.Show(ex.Message);
}
```

Initialisierung

Anstatt die Anfangswerte einzeln zuzuweisen, können Sie diese zusammen mit einem Konstruktor übergeben.

Beispiel: Das Objekt *kunde1* wird erzeugt (Standardkonstruktor), zwei Eigenschaften werden einzeln zugewiesen.

```
CKunde kunde1 = new CKunde();
kunde1.anrede = "Frau"; kunde1.name = "Müller";
```

Beispiel: Das Objekt *kunde1* wird erzeugt und mit einem Konstruktor initialisiert.

```
CKunde kunde1 = new CKunde("Frau", "Müller");
```

Weitere Einzelheiten entnehmen Sie dem Abschnitt 4.5.1 (Konstruktor).

Arbeiten mit dem Objekt

Wie Sie bereits wissen, erfolgt der Zugriff auf Eigenschaften und Methoden eines Objekts, indem der Name des Objekts mit einem Punkt (.) vom Namen der Eigenschaft/Methode getrennt wird.

Syntax: *Objekt.Eigenschaft|Methode()*

[1] Aus Platzgründen halten sich die Autoren nicht immer an diese Empfehlung.

Beispiel: Die Eigenschaft *guthaben* des Objekts *kunde1* wird zugewiesen und die Methode *adresse* aufgerufen.

```
kunde1.guthaben = 10;
label1.Text = kunde1.adresse();
```

Zerstören des Objekts

Wenn Sie das Objekt nicht mehr brauchen, können Sie die Objektvariable auf *null* setzen.

Beispiel: Der *kunde1* wird in die ewigen Jagdgründe befördert.

```
kunde1 = null;
```

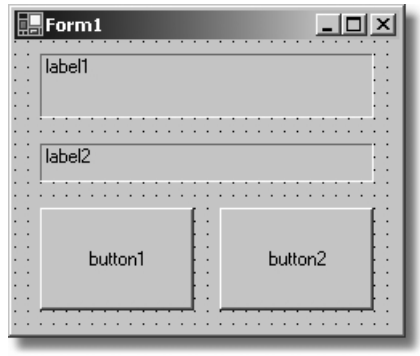
Das Objekt wird allerdings erst dann zerstört, wenn der Garbage Collector festgestellt hat, dass es nicht länger benötigt wird (siehe 4.5.2).

4.1.6 Einführungsbeispiel

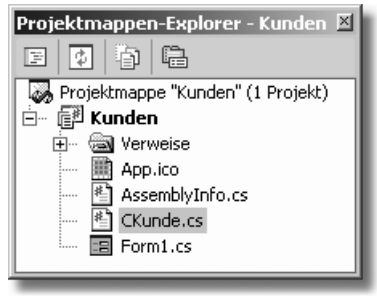
Raus aus dem muffigen Hörsaal, lasst uns mit Visual Studio.NET endlich einmal selbst eine einfache Klasse erstellen und beschnuppern!

Vorbereitungen

- Öffnen Sie ein neues Projekt (z.B. mit dem Namen "Kunden") als Windows-Anwendung.
- Auf das Startformular (*Form1*) platzieren Sie zwei Labels und zwei Buttons.



- Nachdem Sie den Menüpunkt *Projekt|Klasse hinzufügen...* gewählt haben, geben Sie im Dialogfenster den Namen *CKunde.cs* ein und klicken "Öffnen". Der Projektmappen-Explorer zeigt jetzt die neue Klasse:



Hinweis: Sie müssen eine Klasse nicht unbedingt in einem eigenen Klassenmodul definieren, Sie könnten die Klasse z.B. auch zum bereits vorhandenen Code-Fenster (*Form1.cs*) hinzufügen. Das Verwenden eigener Klassenmodule (idealerweise eins pro Klasse) steigert aber die Übersichtlichkeit des Programmcodes und erleichtert dessen Wiederverwendbarkeit.

Klasse definieren

Im Code-Fenster *Kunde.cs* ist bereits der Rahmencode für unsere Klasse vorbereitet:

```
using System;

namespace Kunden
{
        /// <summary>
        /// Zusammenfassende Beschreibung für CKunde.
        /// </summary>
        public class CKunde
        {
                public CKunde()
                {
                        //
                        // TODO: Fügen Sie hier die Konstruktorlogik hinzu
                        //
                }
        }
}
```

Sie sehen, dass die Klasse *CKunde* im gleichen Namensraum (*Kunden*) angelegt wurde, wie die bereits vorhandene Klasse *Form1*. Standardmäßig wurde die *System*-Klassenbibliothek importiert.

Löschen Sie alle Zeilen, die wir im obigen Code **nicht** fett hervorgehoben haben, denn sie sind momentan nicht wichtig (auf den Konstruktor kommen wir erst später zu sprechen, siehe 4.5.1).

Tragen Sie dann in den Klassenkörper die Implementierung der Klasse ein, so dass der komplette Code schließlich folgendermaßen ausschaut:

```
using System;
namespace Kunden
{
         public class CKunde
         {
                  private const char LF = (char) 10;  //  private Konstante (Zeilenumbruch)
                  public string anrede;          //    öffentliches Feld
                  public string name;            //      dto.
                  public int plz;                //      dto.
                  public string ort;             //      dto.
                  public bool stammkunde;        //      dto.
                  public decimal guthaben;       //      dto.
            public string adresse()                 //    öffentliche Methode
            {
                     string s = anrede + " " + name + LF + plz.ToString() + " " + ort;
                     return(s);
            }
            public void addGuthaben(decimal betrag)   // öffentliche Methode
            {
                     if (stammkunde) guthaben += betrag;
            }
         }
}
```

Bemerkungen

- Die Klasse verfügt über sechs "einfache" Eigenschaften, und zwar sind das alle als *public* deklarierten Variablen, die man auch als "öffentliche Felder" bezeichnet. Die Betonung liegt hier auf "einfach", da wir später noch lernen werden, wie man "richtige" Eigenschaften programmiert.
- Weiterhin verfügt die Klasse über zwei *Methoden*. Die *string*-Methode *adresse()* liefert einen Rückgabewert, nämlich die komplette Anschrift.
- Die *void*-Methode *addGuthaben* hingegen liefert keinen Wert zurück, sie erhöht den Wert des *guthaben*-Felds bei jedem Aufruf um 50 €.
- Die private Konstante *LF* wird von der Methode *adresse()* für das Einfügen des Zeilenumbruchs benötigt.

Objekt erzeugen und initialisieren

Wechseln Sie nun in das Code-Fenster von *Form1*.

Auf Klassenebene deklarieren Sie eine Objektvariable *kunde1*:

```
private CKunde kunde1;                // Objekt referenzieren
```

Dem linken Button geben Sie die Beschriftung "Objekt erzeugen und initialisieren" und belegen das *Click*-Ereignis wie folgt:

```
private void button1_Click(object sender, System.EventArgs e)
{
         kunde1 = new CKunde();          // Objekt erzeugen
         // Objektfelder initialisieren:
         kunde1.anrede = "Herr";
         kunde1.name = "Müller";
         kunde1.plz = 12345;
         kunde1.ort = "Berlin";
         kunde1.stammkunde = true;
}
```

Objekt benutzen

Hinterlegen Sie nun den rechten Button mit der Beschriftung "Methoden und Eigenschaften benutzen" wie folgt:

```
private void button2_Click(object sender, System.EventArgs e)
{
         label1.Text = kunde1.adresse();  // erste Methode aufrufen
         kunde1.addGuthaben(50M);         // zweite Methode aufrufen
         label2.Text = "Guthaben ist " +
          kunde1.guthaben.ToString("C");  // Eigenschaft lesen
}
```

Unterstützung durch die Intellisense

Sie haben beim Eintippen des Quelltextes (insbesondere im Code-Fenster von *Form1*) bereits gemerkt, dass Sie durch die Intellisense von VS.NET eifrigst unterstützt werden.

Die Intellisense weist Sie z.B. auf die verfügbaren Klassenmitglieder (Eigenschaften und Methoden) hin und ergänzt den Quellcode automatisch, wenn Sie auf den gewünschten Eintrag doppelklicken.

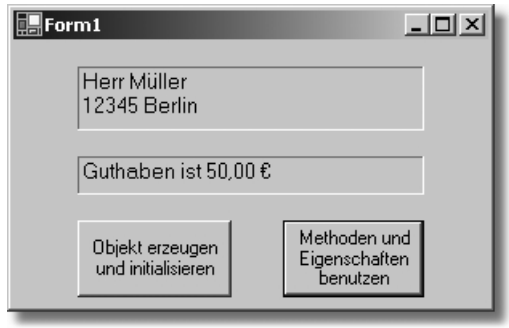
```
// Objekt erzeugen und initialisieren:
private void button1_Click(object sender, System.EventArgs e)
{
    kunde1 = new CKunde();
    kunde1.
    kund                                    er";
    kund
    kund                                    n";
    kund                                     true;
}
// Method                                   aften benutzen:
private                                     ick(object sender, System.EventArgs e)
{
    labe                                    l.Adresse();    // erste Methode aufrufen
    kund                                    50M);          // zweite Methode aufrufen
```

addGuthaben
Adresse
anrede
Equals
GetHashCode
GetType
guthaben
name
ort
plz

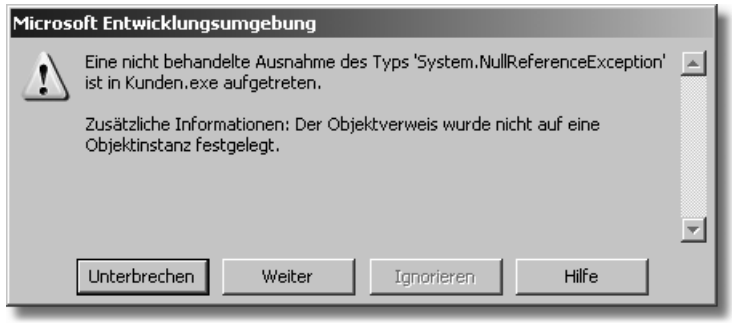
Falls das gewünschte Klassenmitglied nicht erscheint, müssen Sie sofort stutzig werden und es keinesfalls mit dem gewaltsamen Eintippen des Namens versuchen, denn dann gibt es wahrscheinlich einen Fehler beim Compilieren. Überprüfen Sie stattdessen lieber nochmals die Klassendeklaration, z.B. ob vielleicht nicht doch der *public*-Modifizierer vergessen wurde.

Objekt testen

Nun ist es endlich so weit, dass Sie Ihr erstes eigenes C#-Objekt vom Stapel lassen können. Unmittelbar nach Programmstart betätigen Sie den linken Button und danach den rechten. Durch mehrmaliges Klicken auf den zweiten Button wird sich das Guthaben des Kunden Müller in 50-€-Schritten erhöhen.



Falls Sie zu voreilig gewesen sind und unmittelbar nach Programmstart den zweiten statt den ersten Button gedrückt haben, stürzt Ihnen das Programm mit folgender Fehlermeldung ab:



Ein paar kritische Bemerkungen

Unsere Klasse funktioniert nach außen hin zwar ohne erkennbare Mängel, ist hinsichtlich ihrer inneren Konstruktion aber keinesfalls als optimal zu bezeichnen. Wir haben deshalb keinerlei Grund, uns zufrieden zurückzulehnen, denn das uns unter C# zur Verfügung stehende OOP-Instrumentarium wurde von uns bei weitem noch nicht ausgeschöpft.

- Beispielsweise haben wir nur "einfache" Eigenschaften, nämlich *public*-Felder verwendet, was eigentlich eine schwere Sünde in den Augen der OOP-Puristen ist (siehe Abschnitt 4.2.1).
- Weiterhin war das Initialisieren der Eigenschaften über mehrere Codezeilen ziemlich mühselig (von einem hilfreichen Konstruktor haben wir noch keinerlei Gebrauch gemacht, siehe Abschnitt 4.5.1).
- Außerdem wird eine Klasse erst so richtig effektiv, wenn davon nicht nur eine, sondern mehrere Instanzen (sprich Objekte) abgeleitet werden. Diese wiederum kann man ziemlich elegant in so genannten Auflistungen (Collections) verwalten (siehe Abschnitt 4.7).

Doch zur Beseitigung dieser und anderer Meckereien kommen wir erst später. Ein weiteres Problem, was uns unter den Nägeln brennt, können und wollen wir aber nicht weiter aufschieben und es gleich im folgenden Abschnitt behandeln.

4.2 Eigenschaften

Eigenschaften bestimmen die statischen Attribute eines Objekts, sie leiten sich von dessen *Zustand* ab, wie er in den Zustandsvariablen (Objektfeldern) gespeichert ist. Im Unterschied zu den Methoden, die von allen Instanzen der Klasse gemeinsam genutzt werden, sind die den Eigenschaften zugewiesenen Werte für alle Objekte einer Klasse meist unterschiedlich.

4.2.1 Eigenschaften mit Zugriffsmethoden kapseln

Von den im Objekt enthaltenen Feldern sind die *public*-Felder als "einfache" Eigenschaften zu betrachten.

Im Beispiel 4.1.4 hatten wir für die Klasse *CKunde* solche "einfachen" Eigenschaften als *public*-Variable deklariert. Das allerdings ist nicht die "feine Art" der objektorientierten Programmierung, denn das Veröffentlichen von Feldern widerspricht dem hochgelobten Prinzip der Kapselung und erlaubt keinerlei Zugriffskontrolle wie z.B. Wertebereichsüberprüfung oder die Vergabe von Lese- und Schreibrechten.

Idealerweise sind deshalb in einem Objekt nur private Felder enthalten, und der Zugriff auf diese wird durch Accessoren (Zugriffsmethoden) gesteuert.

In diesem Sinn ist eine *Eigenschaft* gewissermaßen ein Mittelding zwischen Feld und Methode. Sie verwenden die Eigenschaft wie ein öffentliches Feld. Vom Compiler aber wird der Feldzugriff in den Aufruf von Accessoren – das sind spezielle Zugriffsmethoden auf private Felder – übersetzt. Doch schauen wir uns das Ganze lieber in der Praxis an.

Deklaration von Eigenschaften

Eigenschaften werden ähnlich wie öffentliche Methoden deklariert. Innerhalb der Deklaration implementieren Sie für den Lesezugriff eine *set*- und für den Schreibzugriff eine g*et*-Zugriffsmethode. Während die *get*-Methode ihren Rückgabewert über *return* liefert, erhält die *set*-Methode den zu schreibenden Wert über *value*.

Syntax:
```
Modifizierer Datentyp Eigenschaftsname
{
  get
    {
      // hier Lesezugriff auf priv. Felder implementieren
      return(privatesFeld);
    }
  set
    {
      // hier Schreibzugriff auf priv. Felder implementieren
      privatesFeld = value;
    }
}
```

Lasst uns nun unser Beispiel 4.1.4 endlich mit "echten" Eigenschaften ausstatten! Dazu werden zunächst die public-Felder in *private* verwandelt und durch Voranstellen von "_" umbenannt, um Namenskonflikte mit den gleichnamigen Eigenschafts-Deklarationen zu vermeiden.

Der Schreibzugriff auf die Eigenschaft *anrede* wird so kontrolliert, dass nur die Werte "Herr" oder "Frau" zulässig sind.

```
using System;
using System.Windows.Forms;        // wegen MessageBox
namespace Kunden
{
```

```
public class CKunde
{
 // ...
        private string _anrede;          //    privates Feld
        private string _name;            //      dto.
        private int _plz;                //      dto.
        private string _ort;             //      dto.
        private bool _stammkunde;        //      dto.
        private decimal _guthaben;       //      dto.

       public string anrede
       {
          get {return(_anrede); }
          set
           {
             if (value == "Herr" || value == "Frau") _anrede = value;
                 else MessageBox.Show("Die Anrede '" + value + "' ist nicht zulässig!");
           }
       }

       public string name
       {
          get {return(_name); }
          set {_name = value; }
       }

  // .... hier folgen die weiteren Implementierungen    (siehe Buch-CD)
 }
}
```

Zugriff

Wenn Sie ein Objekt benutzen, merken Sie auf Anhieb natürlich nicht, ob es noch über "einfache" oder schon über "richtige" Eigenschaften verfügt, es sei denn, die in die *get*- bzw. *set*-Methoden eingebauten Zugriffsbeschränkungen werden verletzt und Sie erhalten entsprechende Fehlermeldungen.

Beispiel: Sie wollen die Anrede "Mister" zuweisen, was nicht zulässig ist.

```
kunde1 = new CKunde();
kunde1.anrede = "Mister";      // erzeugt Laufzeitfehler
```

Bemerkung

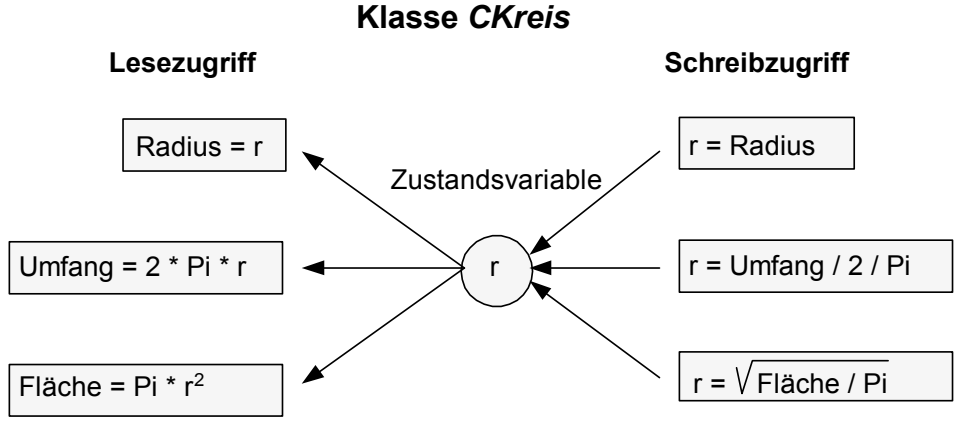
- Beim Schreiben des Quellcodes in der Entwicklungsumgebung VS.NET merken Sie den "feinen" Unterschied zwischen "einfachen" und "richtigen" Eigenschaften, denn die Intellisense zeigt dafür unterschiedliche Symbole.
- In unserem Beispiel verhält sich nur die Eigenschaft *anrede* "intelligent", d.h., sie unterliegt einer Zugriffskontrolle. Bei den übrigen Eigenschaften erfolgt lediglich eine 1:1-Zuordnung zu den privaten Feldern. Hier sollte man nicht "päpstlicher als der Papst" sein und es bei den ursprünglichen *public*-Feldern belassen. Wir aber haben diesen (eigentlich sinnlosen) Aufwand nur wegen des Lerneffekts betrieben.

.

4.2.2 Berechnete Eigenschaften

Mit Zugriffsmethoden lässt sich weit mehr anstellen, als nur den Zugriff auf private Felder der Klasse zu kontrollieren. So können z.B. innerhalb der Methode komplexe Berechnungen mit den Feldern (die man auch *Zustandsvariablen* nennt) und den übergebenen Parametern ausgeführt werden.

Beispiel: Eine Klasse *CKreis* hat die Eigenschaften *radius*, *umfang* und *fläche*. In der einzigen Zustandsvariablen *r* braucht aber nur der Radius abgespeichert zu werden, da sich die übrigen Eigenschaften aus *r* berechnen lassen (*get* = Lesezugriff) bzw. umgekehrt (*set* = Schreibzugriff).



```
public class CKreis
{
        private double r;                   // das einzige Feld (Zustandsvariable)
```

Die Eigenschaft *radius*:

```
        public string radius
        {
                get {return (r.ToString("#,#0.00")); }
                set
                {
                        if (value != "")  r = Convert.ToDouble(value);
                        else r = 0;
                }
        }
```

Die Eigenschaft *umfang*:

```
        public string umfang
        {
                get {return (2 * Math.PI * r).ToString("#,#0.00"); }
                set
                {
                        if (value != "")  r = Convert.ToDouble(value) / 2 / Math.PI;
                        else r = 0;
                }
        }
```

Die Eigenschaft *fläche*:

```
        public string fläche
        {
                get {return (Math.PI * Math.Pow(r, 2)).ToString("#,#0.00");}
                set
                {
                if (value != "") r = Math.Sqrt(Convert.ToDouble(value) / Math.PI);
                                 else r = 0;
                }
        }
}
```

Das komplette Programm finden Sie unter

R9 ... Eigenschaften mit Zugriffsmethoden kapseln?

4.2.3 Lese-/Schreibschutz

Es kommt häufig vor, dass bestimmte Eigenschaften nur gelesen oder nur geschrieben werden dürfen (*ReadOnly* bzw. *WriteOnly*). Um diese Art der Zugriffsbeschränkung zu realisieren, ist keinerlei Aufwand erforderlich – im Gegenteil:

Hinweis: Um eine Eigenschaft allein für den Lese- bzw. Schreibzugriff zu deklarieren, lässt man einfach die *get*- bzw. die *set*-Zugriffsmethode weg.

Beispiel: In unserer *CKunde*-Klasse soll das Guthaben für den direkten Schreibzugriff gesperrt werden. Das klingt logisch, da zur Erhöhung des Guthabens bereits die Methode *addGuthaben* existiert.

```
public decimal guthaben
{
        get {return(_guthaben); }
}
```

In der Entwicklungsumgebung von VS.NET wird nun der Versuch abgewiesen, dieser Eigenschaft einen Wert zuzuweisen:

```
kunde1.stammkunde = true;
kunde1.guthaben = 10M;
```

Einer Eigenschaft oder einem Indexer 'Kunden.CKunde.guthaben' kann nicht zugewiesen werden -- sie sind schreibgeschützt

4.2.4 Statische Eigenschaften

Mitunter gibt es Eigenschaften, deren Werte für alle aus der Klasse instanziierten Objekte identisch sind und die deshalb nur einmal in der Klasse gespeichert zu werden brauchen.

Hinweis: Statische Eigenschaften (*Klasseneigenschaften*) werden mit dem Schlüsselwort *static* deklariert.

Außer dem *static*-Schlüsselwort gibt es beim Deklarieren keine Unterschiede zu den normalen Instanzeneigenschaften.

Statische Eigenschaften können benutzt werden, ohne dass dazu eine Objektvariable deklariert und ein Objekt instanziiert werden muss! Es genügt das Voranstellen des Klassenbezeichners.

Beispiel: Die Klasse *CKunde* soll zusätzlich eine "einfache" Eigenschaft *rabatt* bekommen, die für jedes Kundenobjekt immer den gleichen Wert hat.

```
public class CKunde
{
```

```
    ...
    public static double rabatt;
    ...
}
```

Der Zugriff ist sofort über den Klassenbezeichner möglich, ohne dass dazu eine Objektvariable erzeugt werden müsste.

Beispiel: Allen Kunden wird ein Rabatt von 15% zugewiesen.

```
CKunde.rabatt = 0.15;
```

Hinweis: Vielen Umsteigern, die aus der strukturierten Programmierung kommen, bereitet es Schwierigkeiten, auf ihre globalen Variablen zu verzichten, mit denen sie Werte zwischen verschiedenen Programmmodulen ausgetauscht haben. Genau hier bieten sich statische Eigenschaften an, die z.B. in einer extra für derlei Zwecke angelegten Klasse *CAllerlei* abgelegt werden könnten.

4.3 Methoden

Methoden bestimmen die dynamischen Attribute eines Objekts, also sein Verhalten. Eine Methode ist eine Funktion, die im Körper der Klasse implementiert ist.

4.3.1 Öffentliche und private Methoden

Bereits im Kapitel 3 (Abschnitt 3.11) haben wir gelernt, wie man Methoden programmiert. Jetzt wollen wir noch etwas nachhaken und den Fokus auf die Methoden richten, die in unseren selbst programmierten Klassen zum Einsatz kommen.

Genau wie das bei "richtigen" Eigenschaften der Fall ist, arbeiten in einer sauber programmierten Klasse alle Methoden ausschließlich mit privaten Feldern (Zustandsvariablen) zusammen.

Wenn Sie eine Methode als *private* deklarieren, ist sie nur innerhalb der Klasse sichtbar, und es handelt sich um keine Methode im eigentlichen Sinn.

Beispiel: Die beiden öffentlichen Methoden *adresse* und *addGuthaben* des erweiterten Beispiels 4.1.4 arbeiten mit sechs privaten Feldern zusammen.

```
public class CKunde
{
        private string _anrede;         //    privates Feld
        private string _name;           //      dto.
        private int _plz;               //      dto.
```

```
        private string _ort;             //      dto.
        private bool _stammkunde;        //      dto.
        private decimal _guthaben;       //      dto.
        public string adresse()          //    öffentliche Methode
        {
           string s = _anrede + " " + _name  + _plz.ToString() + " " + _ort;
           return(s);
        }
        public void addGuthaben(decimal betrag)   // öffentliche Methode
        {
          if (stammkunde) _guthaben += betrag;
        }
}
```

Deklaration und Aufruf:

```
CKunde kunde1 = new CKunde();
...
label1.Text = kunde1.adresse();   // erste Methode aufrufen
kunde1.addGuthaben(50M);          // zweite Methode aufrufen
```

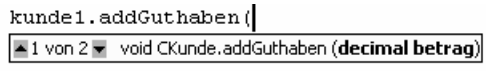
4.3.2 Überladene Methoden

Innerhalb des Klassenkörpers dürfen zwei und mehr gleichnamige Methoden konfliktfrei nebeneinander existieren, wenn sie eine unterschiedliche Signatur (Reihenfolge und Datentyp der Übergabeparameter) besitzen.

Beispiel: Zwei überladene Versionen einer Methode in der Klasse *CKunde*, die erste hat nur den Nettobetrag als Parameter die zweite den Bruttobetrag und die Mehrwertsteuer.

```
public void addGuthaben(decimal betrag)              // erste Überladung
{
          if (stammkunde) _guthaben += betrag;
}

public void addGuthaben(double brutto, double mwst)   // zweite Überladung
{
          if (stammkunde) _guthaben += Convert.ToDecimal(brutto/(1 + mwst));
}
```

Wenn Sie diese Methoden benutzen wollen, so fällt die Auswahl im Code-Fenster leicht:



4.3.3 Statische Methoden

Genauso wie die unter 4.2.4 erläuterten statischen Eigenschaften können *statische Methoden* (auch als *Klassenmethoden* bezeichnet) ohne Verwendung eines Objekts aufgerufen werden. Statische Methoden eignen sich z.B. gut für diverse Formelsammlungen (siehe *Math*-Klassenbibliothek in Abschnitt 3.10.1 des Sprachkapitels). Auch können Sie damit auf private statische Klassenmitglieder zugreifen.

Hinweis: Der Einsatz statischer Methoden für relativ einfache Aufgaben ist bequemer und ressourcenschonender als das Arbeiten mit Objekten, die Sie jedes Mal extra instanziieren müssten.

Beispiel: Wir bauen eine Klasse, in die wir wahllos einige von uns häufig benötigte Berechnungsformeln verpacken.

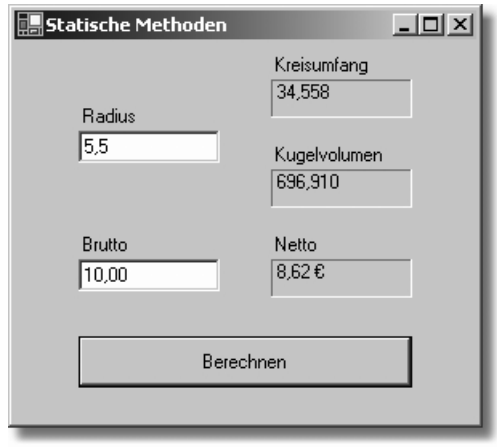
```
public class meineFormeln
{
    public static double kreisUmfang(double radius)
    {
       return (2 * Math.PI * radius);
    }

    public static double kugelVolumen(double radius)
    {
       return ( 4 / 3.0 * Math.PI * Math.Pow(radius, 3));
    }

    public static decimal netto(decimal brutto, double mwst)
    {
        return(brutto/ Convert.ToDecimal(1 + mwst));
    }
// .... weitere Methoden
}
```

Der Zugriff von außerhalb ist absolut problemlos, weil man sich nicht mehr um das lästige Instanziieren einer Objektvariablen kümmern muss.

Beispiel: Die statischen Methoden der Klasse *meineFormeln* werden in einer Eingabemaske aufgerufen.

```
private void button1_Click(object sender, System.EventArgs e)
{
         double r = Convert.ToDouble(textBox1.Text);       // Kreisradius konvertieren
         label1.Text = meineFormeln.kreisUmfang(r).ToString("0.000");
         label2.Text = meineFormeln.kugelVolumen(r).ToString("0.000");
```

```
        decimal b = Convert.ToDecimal(textBox2.Text);    // Brutto konvertieren
        label3.Text = meineFormeln.netto(b, 0.16).ToString("C");
}
```



Hinweis: Sie können mit *static* auch ein Feld deklarieren, dass von allen Instanzen der Klasse gemeinsam genutzt werden kann und nicht für jedes Objekt extra zugewiesen werden muss (siehe Beispiel im Abschnitt 4.5.2).

4.4 Ereignisse

Nachdem wir uns den Eigenschaften und Methoden von Objekten ausführlich gewidmet haben, wollen wir die Dritten im Bunde, die Ereignisse, nicht vergessen. Wie Sie bereits wissen, werden Ereignisse unter bestimmten Bedingungen vom Objekt ausgelöst und können dann in einer Ereignisbehandlungsroutine abgefangen und ausgewertet werden.

Allerdings bieten bei weitem nicht alle Klassen Ereignisse an, denn diese werden nur benötigt, wenn auf bestimmte Änderungen eines Objekts reagiert werden soll.

Nachdem wir mit dem Deklarieren von Eigenschaften und Methoden überhaupt keine Probleme hatten, hört aber bei Ereignissen der Spaß auf.

4.4.1 Ereignisse hinzufügen

Um einer Klasse ein Ereignis hinzuzufügen, sind drei Schritte erforderlich:

- 1. Die Deklaration des Ereignistyps (*delegate*-Schlüsselwort)
- 2. Die Instanziierung des Ereignisses (*event*-Schlüsselwort)
- 3. Das Auslösen des Ereignisses (innerhalb einer Methode oder Eigenschaft)

Um einer heillosen Verwirrung vorzubeugen, machen wir es diesmal umgekehrt und beginnen gleich mit einem Beispiel, ehe wir später die Syntax und weitere Einzelheiten erklären.

Beispiel: In unserer *Kunden*-Klasse wird ein Ereignis-Delegate mit dem Namen *GuthabenLeer* deklariert, davon wird ein Ereignis mit dem Namen *guthabenLeer1* erzeugt. Dieses Ereignis "feuert" innerhalb der Methode *addGuthaben* genau dann, wenn das Guthaben den Wert von 10 € unterschreitet.

```
public class CKunde
{
```

Um den Code für den Benutzer der Klasse etwas zu vereinfachen, werden den privaten Feldern Standardwerte zugewiesen[1]:

```
  private bool _stammkunde = true;        //      initialisiertes Feld
  private decimal _guthaben = 100M;       //      dto.
```

1.Schritt: den Ereignistyp definieren:

```
  public delegate void GuthabenLeer(object sender, string e);
```

2. Schritt: eine Ereignisinstanz *guthabenLeer1* deklarieren:

```
  public event GuthabenLeer guthabenLeer1;
```

Die Methode, in welcher das Ereignis ausgelöst wird:

```
  public void addGuthaben(decimal betrag)
  {
     if (stammkunde) _guthaben += betrag;
```

3. Schritt: Ereignis auslösen:

```
        if (_guthaben <= 10)
        {
            // das Ereignis feuert nur, wenn ...
            if (guthabenLeer1 != null)  // ... mindestens ein Event-Handler angemeldet ist
            {
                string msg = "Das Guthaben beträgt nur noch " +
                              _guthaben.ToString("C") + "!";
                guthabenLeer1(this, msg);
             }
        }
   }
  // ... weitere Implementierungen
}
```

[1] Später werden wir diese Aufgabe dem Konstruktor übertragen.

Nun kommen wir zu den sicherlich dringend notwendigen Erklärungen:

Ereignis deklarieren

Ereignisse werden von so genannten Delegaten abgeleitet. Ein Delegate ist ein Ereignistyp, er sieht – bis auf das *delegate*-Schlüsselwort – wie eine Methode aus und verhält sich auch ähnlich. Delegaten ermöglichen es, ein Framework von Rückruf- und Benachrichtigungsmethoden für miteinander kooperierende Klassen zu implementieren.

Syntax: *Modifizierer* **delegate** *Datentyp delegateName* (*Datentyp Parameter*);

Falls der Delegate keinen Rückgabewert liefert, wird dieser – wie bei einer Methode – als *void* angegeben.

Beispiel: Die Deklaration des Delegaten aus dem Vorgängerbeispiel

```
public delegate void GuthabenLeer(object sender, string e);
```

Ereignis instanziieren

Nachdem Sie mittels Delegaten den Ereignistyp deklariert haben, steht dem Erzeugen einer Delegatinstanz, also eines spezifischen Ereignisses, nichts mehr im Wege. Sie benutzen dazu das *event*-Schlüsselwort.

Syntax: *Modifizierer* **event** *delegateName ereignisName*;

Ähnlich wie bei Objekten (diese sind bekanntlich Instanzen einer Klasse) handelt es sich bei einem Ereignis um eine Instanz des Delegaten. Da der Aufbau bereits feststeht, genügen die Angabe des Namens der Ereignisdeklaration (*delegateName*) und der spezielle Name des Ereignisses (*ereignisName*).

Beispiel: Von im Vorgängerbeispiel deklarierten Delegaten wird ein Ereignis mit dem Namen *guthabenLeer1* instanziiert.

```
public event GuthabenLeer guthabenLeer1;
```

Es ist durchaus möglich und üblich, auch mehrere Ereignisse vom gleichen Delegaten abzuleiten.

Ereignis auslösen

Ein Ereignis wird immer innerhalb der Klasse ausgelöst, in der es deklariert und erzeugt wurde. Das kann an verschiedenen Stellen innerhalb von Methoden oder Eigenschaften geschehen.

Das Auslösen erfolgt wie ein normaler Methodenaufruf:

Syntax: *ereignisName*(*Parameter*);

Die Signatur der Parameter (Reihenfolge und Datentyp) muss der im entsprechenden Delegate festgelegten Parameterliste entsprechen.

Hinweis: Ereignisse sind Verweistypen und können demzufolge auch auf *null* abgefragt werden.

Beispiel: Das im Vorgängerbeispiel deklarierte Ereignis wird innerhalb der Methode *addGuthaben* ausgelöst[1]. Vor dem Aufruf wird getestet, ob zumindest ein Event-Handler für dieses Ereignis angemeldet ist (was genau unter "Anmelden" zu verstehen ist, erfahren Sie im nächsten Abschnitt).

```
public void addGuthaben(decimal betrag)
{
   if (stammkunde) _guthaben += betrag;
       if (_guthaben <= 10)
       {
           if (guthabenLeer1 != null)    // mindestens ein Event-Handler angemeldet?
           {
               string msg = "Das Guthaben beträgt nur noch " +
                            _guthaben.ToString("C") + "!";
               guthabenLeer1(this, msg);
            }
       }
 }
```

4.4.2 Ereignisse benutzen

In der Klasse, in welcher wir mit dem Ereignis arbeiten wollen, sind – zusätzlich zur Erzeugung der Objektvariablen – zwei Schritte durchzuführen:

- 1. Ereignisbehandlung (Event-Handler) schreiben
- 2. Event-Handler anmelden

Lassen Sie uns auch hier mit einem Beispiel beginnen.

Beispiel: Wir benutzen die im Vorgängerabschnitt definierte Klasse *CKunde*, welche von uns gerade mit dem Ereignis *guthabenLeer1* nachgerüstet wurde.

[1] Im Programmiererjargon sagt man auch "Das Ereignis feuert"!

Auf Klassenebene referenzieren wir zunächst die übliche Objektvariable.

```
private CKunde kunde1;                          // Objekt referenzieren
```

1. Schritt: Wir schreiben nun eine Ereignisbehandlung (Event-Handler) für das Ereignis:

```
private void guthabenKontrolle(object o, string s)
{
        MessageBox.Show(s, "Warnung");
}
```

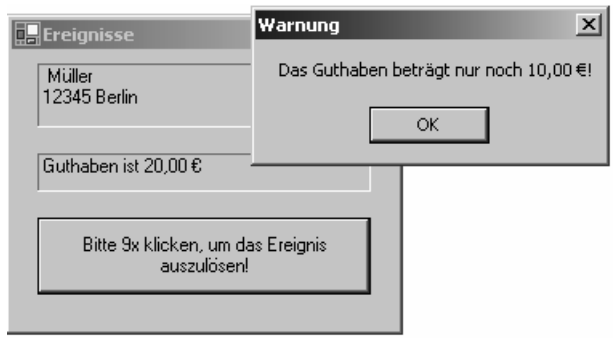
2. Schritt: Um das Ereignis mit dem Event-Handler zu verbinden, ist eine Anmeldung erforderlich, die wir zum Beispiel beim Laden des Formulars mit erledigen:

```
private void Form1_Load(object sender, System.EventArgs e)
{
        kunde1 = new CKunde();      // Objekt instanziieren
// .....
// Event-Handler anmelden:
        kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
}
```

Das Ereignis ist jetzt eingebunden, und einem Funktionstest steht nichts mehr im Weg. Dazu rufen wir wiederholt die Methode *addGuthaben* auf, die das Guthaben jedes Mal um 10 € verringert:

```
private void button2_Click(object sender, System.EventArgs e)
{
        label1.Text = kunde1.adresse();
        kunde1.addGuthaben(-10M);         // Guthaben verringern
        label2.Text = "Guthaben ist " + kunde1.guthaben.ToString("C");
}
```

Nachdem Sie den *Button* neunmal geklickt haben, feuert das Ereignis:



Den kompletten Quellcode entnehmen Sie der Buch-CD.

Nun zu den Details.

Ereignisbehandlung schreiben

Die Frage "Was soll passieren, wenn das Ereignis ausgelöst wurde?" wird in einer Ereignisbehandlungsmethode (Event-Handler) beantwortet.

Syntax: *Modifizierer Datentyp methodenName* (*Datentyp Parameter*)

Den Namen der Methode können Sie frei wählen. Den konkreten Namen des Ereignisses finden Sie hier nicht, d.h., eine Ereignisbehandlung lässt sich auch von mehreren Ereignissen gemeinsam benutzen. Lediglich die Methodensignatur (*Datentyp Parameter*) muss der des Delegates entsprechen, nach dessen Muster das Ereignis erzeugt wurde.

Beispiel: Beim Auftreten des Ereignisses wird nicht nur der Parameter *s* angezeigt, in der Titelleiste erscheint zusätzlich noch der Name des Kunden, den wir durch explizite Typkonvertierung aus dem ebenfalls übergebenen *object*-Parameter "herausziehen".

```
private void guthabenKontrolle(object o, string s)
{
        CKunde k = (CKunde) o;                // Typecasting
        MessageBox.Show(s, k.name);
}
```

Ereignisbehandlung anmelden

Um dem Compiler mitzuteilen, welcher Event-Handler bei Auftreten des Ereignisses denn nun aufzurufen ist, müssen Sie die gewünschte Ereignisbehandlung beim Objekt (der Klasseninstanz) anmelden.

Syntax: *Objekt.ereignisName* += new *delegateName*(*eventHandlerName*);

Beispiel: Dem Objekt *kunde1* wird mitgeteilt, dass bei Auftreten des Ereignisses *guthabenLeer1* der Event-Handler *guthabenKontrolle* aufzurufen ist.

```
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(guthabenKontrolle);
```

Wichtig ist dabei die Verwendung des Operators += (siehe Sprachkapitel, Abschnitt 3.3), denn pro Ereignis sind durchaus mehrere Event-Handler möglich. In diesem Fall erfolgt deren Abarbeitung in der Reihenfolge der Anmeldung.

Beispiel: Zum Ereignis *guthabenLeer1* wird ein zweiter Event-Handler hinzugefügt. Beim Eintreten des Ereignisses erscheint zunächst das vom ersten Eventhandler produzierte Meldungsfenster (siehe oben) und anschließend der Name des Kunden in einem Label.

```
// zweiten Event-Handler implementieren
private void kundenAnschrift(object o, string s)
{
         CKunde k = (CKunde) o;
         label3.Text = k.name;
}
....
// zweiten Event-Handler hinzufügen:
kunde1.guthabenLeer1 += new CKunde.GuthabenLeer(kundenAnschrift);
```

Falls ein Event-Handler nicht mehr benötigt wird, sollten Sie ihn wieder abmelden.

Beispiel: Abmelden des Event-Handlers *kundenAnschrift*.

```
kunde1.guthabenLeer1 -= new CKunde.GuthabenLeer(kundenAnschrift);
```

Bemerkungen

Wenn Sie im Eigenschaften-Fenster der VS.NET-Entwicklungsumgebung auf bekannte Weise Event-Handler für die Objekte der Bedienoberfläche erzeugen, so hat die IDE nicht nur den Rahmencode des Event-Handlers für Sie generiert, sondern – quasi im Verborgenen – auch die benutzten Ereignisse angemeldet[1]. Üblicherweise übergeben diese Ereignisse zwei Parameter an die aufrufende Instanz: eine Referenz auf das Objekt, welches das Ereignis ausgelöst hat, und ein Objekt der *EvenArgs*- oder einer davon abgeleiteten Klasse.

Beispiel: Rahmencode des automatisch generierten Event-Handlers für das *Click*-Ereignis eines *Button*.

```
private void button1_Click(object sender, System.EventArgs e)
{


}
```

Klappen Sie die Region *Windows Form Designer generated code* auf, so finden Sie die entsprechende Befehlszeile für die Anmeldung des Event-Handlers:

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

[1] Wer sich intensiver mit der Thematik beschäftigen will, sollte sich unter den Stichwörtern Delegates und Schnittstellen bzw. Interfaces in der Online-Hilfe kundig machen.

Die folgende Abbildung veranschaulicht nochmals die Syntax dieser Zeile.

```
this.button1.Click += new System.EventHandler(this.button1_Click);
```

Objekt Ereignis Delegat Event-Handler

Und zum Schluss noch ein Hinweis auf einen ziemlich häufigen Unterlassungsfehler:

Hinweis: Wenn Sie die Codezeilen eines Event-Handlers komplett per Hand löschen (man sollte das eigentlich nicht tun), so müssen Sie auch die entsprechende Anmeldungszeile (siehe oben) löschen, ansonsten gibt es einen Compilerfehler!

4.5 Arbeiten mit Konstruktor und Destruktor

Eine "richtige" objektorientierte Sprache wie C# realisiert das Erzeugen und Entfernen von Objekten mit Hilfe von Konstruktoren und Destruktoren.

"Bis jetzt sind wir doch glänzend ohne Konstruktor ausgekommen!", werden Sie jetzt vielleicht einwenden. Ganz stimmt das nicht, denn wenn Sie sich um keinen eigenen Konstruktor kümmern, wird der von *System.Object* geerbte parameterlose *new*-Standardkonstruktor verwendet.

4.5.1 Konstruktor

Der Konstruktor ist gewissermaßen die Standardmethode der Klasse.

Hinweis: Der Name des Konstruktors ist immer identisch mit dem Namen der Klasse.

Der Konstruktor wird automatisch bei der Instanziierung eines Objekts (*new*) aufgerufen und dient vor allem dazu, den Feldern des neu erzeugten Objekts Anfangswerte zuzuweisen.

Deklaration

Einen Konstruktor fügen Sie dem Klassenkörper ähnlich wie eine *public void*-Methode hinzu, nur dass Sie der Methode den Namen der Klasse geben und auf das *void*-Schlüsselwort verzichten (ein Konstruktor hat keinen Rückgabewert). Als Parameter übergeben Sie die Werte für die Felder, die initialisiert werden sollen.

Syntax:
```
public KlassenName(Datentyp Parameter)
        {
            // Initialisierung der Klasse
        }
```

Wie bei jeder anderen Methode können Sie auch hier mehrere überladene Konstruktoren implementieren.

Beispiel: Unserer Klasse *CKunde* werden zwei überladene Konstruktoren hinzugefügt.

```
public class CKunde
{
```

Die Felder:

```
        private string _anrede;
        private string _name;
        private int _plz;
        private string _ort;
        private bool _stammKunde;
        private decimal _guthaben;
```

Der erste Konstruktor initialisiert nur zwei Felder:

```
public CKunde(string anr, string nam)
{
        _anrede = anr;    _name = nam;
}
```

Der zweite Konstruktor initialisiert **alle** Felder der Klasse:

```
public CKunde(string a, string n, int p, string o, bool s, decimal g)
{
        _anrede = a;      _name = n;        _plz = p;
        _ort = o;         _stammKunde = s;  _guthaben = g;
}
```

Aufruf

Nachdem Sie einer Klasse einen oder mehrere Konstruktoren hinzugefügt haben, sind Sie auch zur Benutzung von mindestens einem davon verpflichtet. Die bisher gewohnte einfache Instanziierung von Objekten ist nicht mehr möglich!

Beispiel: Zwei Objekte der Klasse *CKunde* werden erzeugt und mit Anfangswerten initialisiert. Für jedes Objekt wird ein anderer überladener Konstruktor verwendet.

```
CKunde kunde1, kunde2, kunde3;       // drei Objekte referenzieren
```

Sicherheitshalber bauen wir das Erzeugen der Objekte in einen Exception-Handler ein (siehe dazu Kapitel 11).

```
try
{
```

```
        kunde1 = new CKunde("Herr", "Müller");
        kunde2 = new CKunde("Frau", "Hummel", 12345, "Berlin", true, 100);
    //  kunde3 = new CKunde();    // erzeugt Compilerfehler!!!
        MessageBox.Show("Objekte erfolgreich erzeugt!");
}
catch(Exception ex)
{
        MessageBox.Show(ex.Message + " Sch... Konstruktor!");
}
```

Wenn Sie den Code mit dem Anfangsbeispiel im Abschnitt 4.1.6 vergleichen, so sehen Sie, dass das Initialisieren der Objekte viel übersichtlicher geworden ist. Anstatt umständlich eine Eigenschaft nach der anderen zuzuweisen, geht das jetzt in einer einzigen Befehlszeile.

4.5.2 Destruktor und Garbage Collector

Das Pendant zum Konstruktor ist aus objektorientierter Sicht der Destruktor. Da der Lebenszyklus eines Objektes bekanntlich mit dessen Zerstörung und der Freigabe der belegten Speicherplatzressourcen endet, ist der Destruktor für das Erledigen von "Aufräumarbeiten" zuständig, kurz bevor das Objekt sein Leben aushaucht.

In .NET haben wir allerdings keine echten Destruktoren, da hier die endgültige Zerstörung eines Objekts nicht per Code, sondern automatisch vom Garbage Collector vorgenommen wird. Dieser durchstöbert willkürlich und in unregelmäßigen Zeitabständen den Heap nach Objekten, um diejenigen zu suchen, die nicht mehr referenziert werden.

An die Stelle eines echten Destruktors tritt ein Quasi-Destruktor. Das ist eine Finalisierungsmethode, die zu einem unbestimmbaren Zeitpunkt vom Garbage Collector aufgerufen wird, kurz bevor dieser das Objekt vernichtet.

Ähnlich wie beim Konstruktor wird auch hier der Name der Klasse als Methodenbezeichner verwendet, allerdings mit einer Tilde (~) als Präfix. Der *public*-Zugriffsmodifizierer entfällt, da Sie selbst den Destruktor nicht aufrufen dürfen, auch Parameter dürfen nicht übergeben werden.

Syntax: *~KlassenName*()

```
        {
            // hier Code für Aufräumarbeiten implementieren
        }
```

Beispiel: Unsere Klasse *CKunde* erhält ein öffentliches statisches Feld, welches durch den Konstruktor inkrementiert und durch den Quasi-Destruktor dekrementiert werden soll. Wir beabsichtigen damit, die Anzahl der momentan instanziierten Klassen (sprich Anzahl der Kunden) abzufragen.

Der auf das Wesentliche reduzierte Code von *CKunde*:

```
public class CKunde
{
         public static int anzahl = 0;

// Konstruktor:
         public CKunde()
         {
                  anzahl++;
         }

// Destruktor:
                  ~CKunde()
         {
                  anzahl--;
         }
}
```

Wir verwenden zum Testen der Klasse ein Windows-Formular mit zwei Buttons, einer Timer-Komponente (*Interval* = 1000, *Enabled* = *True*) und einem Label.

Zum Code der Klasse *Form1* fügen Sie hinzu:

```
CKunde kunde1;  // Objekt referenzieren

// Objekt hinzufügen:
private void button1_Click(object sender, System.EventArgs e)
{
         kunde1 = new CKunde();
}

// Objekt entfernen:
private void button2_Click(object sender, System.EventArgs e)
{
         kunde1 = null;     // dereferenzieren
}

// Anzeige der im Speicher befindlichen Instanzen im Sekundentakt:
private void timer1_Tick(object sender, System.EventArgs e)
{
         label1.Text = CKunde.anzahl.ToString();
}
```

Beim Programmtest müssen Sie etwas Geduld aufbringen.

Nach Programmstart fügen Sie durch Klicken auf den linken Button ein Objekt *kunde1* hinzu, wonach sich die Anzeige von 0 auf 1 ändert. Anschließend klicken Sie auf den rechten Button, um das Objekt wieder zu entfernen.

Es kann fünf bis zehn Minuten dauern, bis die Anzeige wieder auf 0 wechselt, nämlich dann, wenn den Garbage Collector gerade einmal wieder die Lust zum Aufräumen überkommt und er den Quasi-Destruktor aufruft[1].

Übrigens können Sie auch den linken Button mehrmals hintereinander klicken. Die Anzeige zählt zwar hoch, das aber täuscht, denn es bleibt bei nur einer Objektvariablen (*kunde1*). Allerdings wird Ressourcenverschwendung betrieben, denn dem Objekt wird immer wieder ein neuer Speicherbereich zugewiesen. Der vorher belegte Speicher liegt brach und wartet auf die Freigabe durch den Garbage Collector.

Hinweis: Obiges Beispiel sollten Sie aufgrund seiner Unberechenbarkeit keinesfalls als Vorbild für ähnliche Zählaufgaben verwenden!

Da wegen der Unberechenbarkeit der Objektvernichtung der Umgang mit dem Quasi-Destruktor ziemlich problematisch ist, sollten Sie für das definierte Freigeben von Objekten besser eine separate Methode verwenden (siehe *Close*- bzw. *Dispose*-Methode in Online-Dokumentation).

[1] Der Garbage Collector läuft in einem eigenen Thread, er wird nur dann aufgerufen, wenn sich die anderen Threads in einem sicheren Zustand befinden.

4.6 Vererbung und Polymorphie

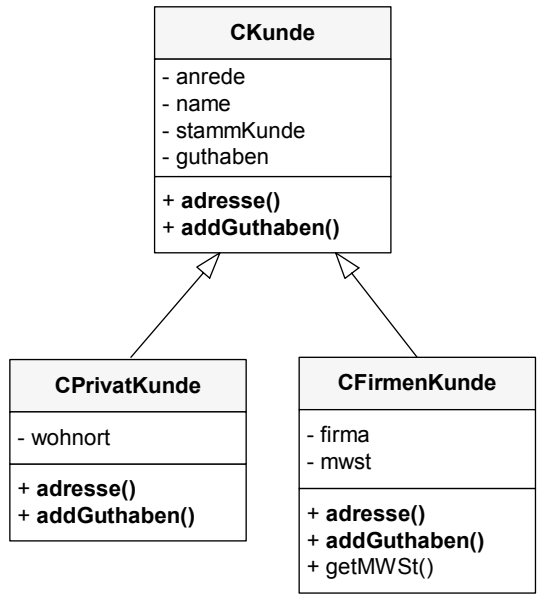
Ein zentrales OOP-Thema ist die *Vererbung*, die es ermöglicht, Klassen zu definieren, die von anderen Klassen abhängen. Eng mit der Vererbung verknüpft ist die *Polymorphie* (Vielgestaltigkeit). Man versteht darunter die Fähigkeit von Subklassen, die Methoden der Basisklasse mit unterschiedlichen Implementierungen zu verwenden. C# unterstützt sowohl Vererbung als auch polymorphes Verhalten, da das Überschreiben (*Overriding*) der Basisklassenmethoden mit alternativen Implementierungen erlaubt ist.

4.6.1 Vererbung verstehen

Durch Vererbung können Sie sich die Programmierarbeit wesentlich erleichtern, indem Sie spezialisierte Subklassen benutzen, die den Code zum großen Teil von einer allgemeinen Basisklasse erben. Die Subklassen heißen auch *abgeleitete Klassen, Kind-* oder *Unterklassen*, die Basisklasse wird auch als *Super-* oder *Elternklasse* bezeichnet. In den Subklassen können Sie bestimmte Funktionalitäten überschreiben, um spezielle Prozesse auszuführen.

Mittels *Unified Modeling Language* (UML) lassen sich Vererbungsbeziehungen zwischen verschiedenen Klassen grafisch darstellen.

Beispiel: Die folgende Abbildung zeigt die Basisklasse *CKunde*, von der die Klassen *CPrivatKunde* und *CFirmenKunde* "erben". Die Basisklasse hat die Eigenschaften *anrede*, *name*, *stammKunde* (ja/nein) und *guthaben* und die Methoden *adresse()* und *addGuthaben()* (das Guthaben ist hier als Bonus zu verstehen, der den Kunden in prozentualer Abhängigkeit von den getätigten Einkäufen gewährt wird).



Die Subklassen *CPrivatKunde* und *CFirmenKunde* können auf sämtliche Eigenschaften und Methoden der Basisklasse zugreifen und fügen selbst eigene Eigenschaften und Methoden hinzu. Die "geerbten" Methoden *adresse* und *addGuthaben* sind allerdings *überschriebene Methoden* (*Method-Overriding*), d.h., Adresse und Guthaben sollen für Privatkunden auf andere Weise als für Firmenkunden ermittelt werden.

4.6.2 Die wichtigsten Regeln

Für die Programmierung von Vererbungsbeziehungen gelten in C# folgende Regeln:

- Alle öffentlichen Eigenschaften und Methoden der Basisklasse sind auch über die abgeleiteten Subklassen verfügbar.
- Methoden der Basisklasse, die von den abgeleiteten Subklassen überschrieben werden dürfen (so genannte *virtuelle Methoden*), müssen mit dem Schlüsselwort *virtual* deklariert werden.
- Fehlt das Schlüsselwort *virtual* bei der Methodendeklaration, so bedeutet das, dass dies die einzige Implementierung der Methode ist.
- Methoden der Subklassen, welche die gleichnamige Methode der Basisklasse überschreiben, müssen mit dem Schlüsselwort *override* deklariert werden.
- Wenn Sie das *override*-Schlüsselwort in der Subklasse vergessen, wird angenommen, dass es sich um eine "Schattenfunktion" der originalen Funktion handelt. Eine solche Funktion hat denselben Namen wie das Original, überschreibt dieses aber nicht.
- Private Felder der Basisklasse, auf die die Subklassen zugreifen dürfen, müssen mit *protected* deklariert werden.
- Die Basisklasse wird der Subklasse durch einen der Klassendeklaration nachgestellten Doppelpunkt bekannt gemacht:

  Syntax:
  ```
  class SubKlasse : BasisKlasse
  {
     // ... Implementierungscode
  }
  ```

- Eine Subklasse kann immer nur von einer einzigen Basisklasse abgeleitet werden (keine multiple Vererbung möglich).
- Mit dem *base*-Objekt kann von den Subklassen auf die Basisklasse zugegriffen werden, mit dem *this*-Objekt auf die eigene Klasse.
- Wenn die Basisklasse einen eigenen Konstruktor verwendet, so müssen in den Subklassen ebenfalls eigene Konstruktoren definiert werden (Konstruktoren können nicht vererbt werden).
- Der Konstruktor einer Subklasse muss den Konstruktor seiner Basisklasse aufrufen (*base*-Schlüsselwort).

- Falls aber die Basisklasse über keinen eigenen Konstruktor verfügt, wird der Standardkonstruktor automatisch aufgerufen, wenn ein Objekt aus einer Subklasse erzeugt wird.

Schluss jetzt mit dem Schattenboxen, lassen Sie uns anhand eines kurzen und dennoch ausführlichen Beispiels endlich die wichtigsten Vererbungstechniken demonstrieren!

4.6.3 Implementieren der Klassen

Vorbild für die drei zu implementierenden Klassen ist obiges Klassendiagramm.

Basisklasse CKunde

Die Deklaration entspricht (fast) der einer normalen Klasse. Dass es sich um eine Basisklasse handelt, erkennt man eigentlich nur an dem *protected*-Feld und an den *virtual*-Methoden-Deklarationen[1].

```
public class CKunde               // Basisklasse
{
```

Die privaten Felder:

```
        private string _anrede;
        private string _name;
        private bool _stammKunde;
```

Durch den *protected*-Modifizierer für das *guthaben*-Feld wird es möglich, dass auch die beiden Subklassen auf dieses Feld zugreifen können:

```
        protected decimal _guthaben = 0;      // Feld ist in Subklassen sichtbar!
```

Ein eigener Konstruktor ersetzt den Standardkonstruktor::

```
        public CKunde(string anrede, string nachName)     // Konstruktor
        {
                _anrede = anrede; _name = nachName;
        }
```

Die Eigenschaften:

```
        public bool stammKunde
        {
                get {return(_stammKunde); }
                set {_stammKunde = value; }
        }
```

[1] Eigentlich hätten wir die Klasse auch noch als *abstract* deklarieren müssen (siehe dazu 4.6.6).

```
        public decimal guthaben        // ReadOnly
        {
                get {return(_guthaben); }
        }
```

Um den Quellcode übersichtlich zu halten, verzichten wir hier auf die Implementierung der Eigenschaften *anrede* und *name* (siehe Abschnitt 4.2), da ein Direktzugriff in unserem Beispiel nicht erforderlich ist (das Zuweisen erledigt der Konstruktor, das Lesen geschieht indirekt über die *adresse*()-Methode).

Die beiden Methoden können durch die Subklassen überschrieben werden:

Der Rückgabewert setzt sich aus Anrede und Namen des Kunden zusammen:

```
        public virtual string adresse()                    // virtuelle Methode
        {
                string s = _anrede + " " + _name;
                return(s);
        }
```

Nur Stammkunden erhalten Bonusguthaben:

```
        public virtual void addGuthaben(decimal betrag)   // virtuelle Methode
        {
                if (stammKunde) _guthaben += betrag;
        }
}
```

Subklasse CPrivatKunde

Die Klasse erbt alle Eigenschaften und Methoden der Basisklasse. Das *override*-Schlüsselwort der beiden Methoden bedeutet, dass hier die in der Basisklasse als *virtual* definierten Funktionen überschrieben werden. Das erlaubt der Subklasse, eine eigene Implementierung der Funktionen zu realisieren.

```
public class CPrivatKunde : CKunde             // erbt von der Basisklasse CKunde!
{
        private string _wohnOrt;
```

Der Konstruktor ist unbedingt notwendig, weil auch die Basisklasse einen eigenen Konstruktor verwendet. Es wird das *base*-Schlüsselwort benutzt, um den Konstruktor der Basisklasse aufzurufen.

```
        public CPrivatKunde(string anrede, string name, string ort): base(anrede, name)
        {
                _wohnOrt = ort;            // klassenspezifische Ergänzung
        }
```

Die erste Methode wird so überschrieben, dass zusätzlich zu Anrede und Name (von der Basisklasse geerbt) noch der Wohnort des Privatkunden angezeigt wird.

```
        public override string adresse()
        {
                const char LF = (char) 10;       // Zeilenvorschub
                return(base.adresse() + LF + _wohnOrt);
        }
```

Die zweite Methode wird komplett neu überschrieben. Ohne Rücksicht auf die Zugehörigkeit zur Stammkundschaft werden jedem Privatkunden (ach wie großzügig!) 5% vom Rechnungsbetrag als Bonusguthaben angerechnet:

```
        public override void addGuthaben(decimal geld)
        {
                // Zugriff auf protected-Variable in Basisklasse:
                _guthaben += 0.05M * geld;
        }
}
```

Subklasse CFirmenKunde

Der Code für die Subklasse *CFirmenKunde* unterscheidet sich in folgenden Details von der Klasse *CPrivatKunde*:

- Die *adresse()*-Methode liefert statt des Wohnorts den Namen der Firma des Kunden.
- Die *guthaben*()-Methode berechnet zunächst den Nettobetrag und addiert davon 1% zum Bonusguthaben. Damit nur Stammkunden in den Genuss dieser Vergünstigung kommen, wird dazu die gleichnamige Methode der Basisklasse aufgerufen.
- Die neu hinzugekommene "stinknormale" Methode *getMWSt*() erlaubt einen Lesezugriff auf die Mehrwertsteuer-Konstante.

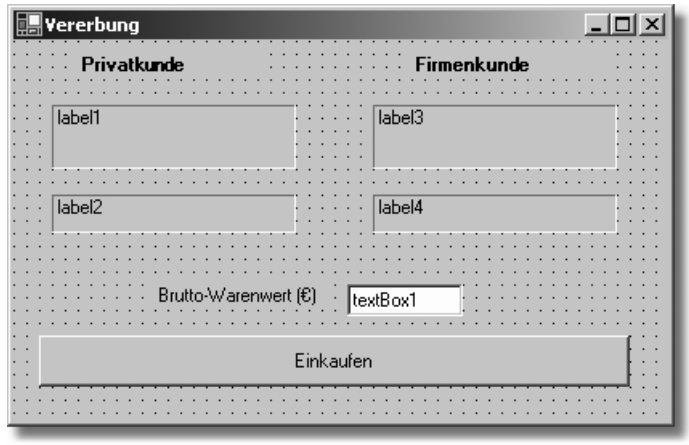
```
public class CFirmenKunde : CKunde
{
        private string _firma;
        private  const float _mwst = 0.16F;     // Mehrwertsteuer

        public CFirmenKunde(string anrede, string name, string frm): base(anrede, name)
        {
                _firma = frm;
        }
        public override string adresse()
        {
                const char LF = (char) 10;      // Zeilenvorschub
```

```
                return(base.adresse() + LF + _firma);
        }

        public override void addGuthaben(decimal brutto)
        {
                decimal netto = brutto / Convert.ToDecimal(1 + _mwst);
                base.addGuthaben(netto * 0.01M);   // Aufruf der Methode der Basisklasse
        }

        public double getMWSt()    // eine ganz normale Methode
        {
                return(_mwst);
        }
}
```

Die Implementierung der drei Klassen ist geschafft!

Testoberfläche

Um die Funktionsfähigkeit der implementierten Klassen zu testen, gestalten Sie die folgende Benutzerschnittstelle:



4.6.4 Implementieren der Objekte

Für einen kleinen Test genügt es, wenn wir mit nur zwei Objekten (einem Privat- und einem Firmenkunden) arbeiten.

```
CPrivatKunde kunde1;
CFirmenKunde kunde2;
```

Beim Laden des Formulars werden die Konstruktoren aufgerufen. Die Ja-/Nein-Eigenschaft *stammKunde* muss extra zugewiesen werden:

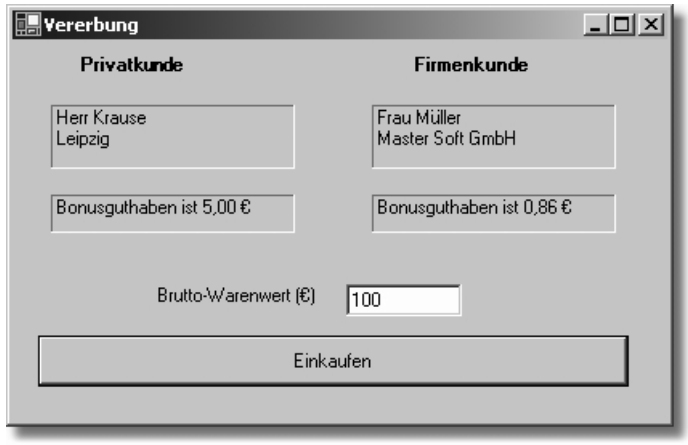
```
private void Form1_Load(object sender, System.EventArgs e)
{
        kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
        kunde1.stammKunde = false;
        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.stammKunde = true;
        textBox1.Text = "100";
}
```

Bei Klick auf den Button werden für jedes Objekt Eigenschaften abgefragt und Methoden aufgerufen:

```
private void button1_Click(object sender, System.EventArgs e)
{
        decimal brutto = Convert.ToDecimal(textBox1.Text);
        label1.Text = kunde1.adresse();
        kunde1.addGuthaben(brutto);
        label2.Text = "Bonusguthaben ist " + kunde1.guthaben.ToString("C");
        label3.Text = kunde2.adresse();
        kunde2.addGuthaben(brutto);
        label4.Text = "Bonusguthaben ist " + kunde2.guthaben.ToString("C");
}
```

Praxistest

Überzeugen Sie sich nun davon, dass die Klassen wie gewünscht zusammenarbeiten und dass Vererbung tatsächlich funktioniert.



Dem Privatkunden Krause wurde ein Guthaben von 5 € (5% aus 100 €) zugebilligt (Stammkundschaft spielt bei Privatkunden keine Rolle, da die Methode *addGuthaben()* komplett überschrieben ist).

Frau Müller ist eine Firmenkundin und erhält – nur weil sie Stammkundin ist – ein mickriges Guthaben von 0,86 € (1% auf den Nettowert).

Durch wiederholtes Klicken auf "Einkaufen" kumulieren die Bonusguthaben.

4.6.5 Polymorphes Verhalten

Polymorphes Verhalten bedeutet, dass erst zur Laufzeit einer Anwendung entschieden wird, welche der möglichen Methodenimplementierungen aufgerufen wird, da dies zum Zeitpunkt des Compilierens noch unbekannt ist.

Bislang haben wir von den Vorzügen der Polymorphie allerdings noch keinen Gebrauch gemacht, denn Privat- und Firmenkunde wurden in einzelnen Objektvariablen gespeichert und bereits per Programmcode fest mit ihren Methoden *adresse()* und *addGuthaben()* verbunden.

Um Polymorphie sichtbar zu machen, müssen wir das bei der Implementierung der Objekte zielgerichtet ausnutzen. Wie wir gleich sehen werden, treten die Vorzüge von Polymorphie besonders augenscheinlich zutage, wenn Objekte unterschiedlicher Klassenzugehörigkeit nacheinander in Arrays oder Auflistungen abgespeichert werden.

Fortsetzung des Beispiels

An den Implementierungen der drei Klassen des Vorgängerbeispiels brauchen wir keinerlei Veränderungen vorzunehmen, denn polymorphes Verhalten ergibt sich als logische Konsequenz aus der Vererbung von Klassen. Änderungen müssen wir lediglich bei der Abspeicherung der Objektvariablen vornehmen.

Aus den Subklassen *CPrivatKunde* und *CFirmenKunde* wollen wir insgesamt drei Objekte (*kunde1*, *kunde2*, *kunde3*) instanziieren (ein Privatkunde, zwei Firmenkunden).

Innerhalb des Klassencodes von *Form1* deklarieren Sie:

```
private CPrivatKunde kunde1;
private CFirmenKunde kunde2, kunde3;

private CKunde[] kunden = new CKunde[3];       // Array für 3 Objekte!

private const char LF = (char) 10;             // für Zeilenvorschub
```

Beim Laden von *Form1* werden die notwendigen Initialisierungen vorgenommen:

```
private void Form1_Load(object sender, System.EventArgs e)
{
         kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
```

```
        kunde1.stammKunde = false;

        kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
        kunde2.stammKunde = true;

        kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");
        kunde3.stammKunde = false;
```

Da das Array vom Typ der Basisklasse ist, kann es auch Objekte der Subklassen (Privat- und Firmenkunden) in wahlloser Reihenfolge aufnehmen:

```
        kunden[0] = kunde1;
        kunden[1] = kunde2;
        kunden[2] = kunde3;
        textBox1.Text = "100";
}
```

Das Array wird in einer *for*-Schleife durchlaufen und ausgelesen. Dabei werden die polymorphen Methoden (das sind die mit *virtual* bzw. *override* deklarierten) für alle Objekte aufgerufen:

```
private void button1_Click(object sender, System.EventArgs e)
{
        decimal brutto = Convert.ToDecimal(textBox1.Text); label1.Text = "";

        for (int i = 0; i < kunden.Length; i++)
        {
                kunden[i].addGuthaben(brutto);
                label1.Text = label1.Text + LF + kunden[i].adresse() + LF
                        + kunden[i].guthaben.ToString("C");
        }
}
```

Obwohl im Array die Objekte bunt durcheinander gewürfelt sein können, "weiß" das Programm zur Laufzeit genau, welche Implementierung der polymorphen Methoden *adresse()* und *addGuthaben()* jeweils für Privat- und für Firmenkunden die richtige ist.

Hinweis: Genau hier liegt der springende Punkt zum Verständnis der Polymorphie!

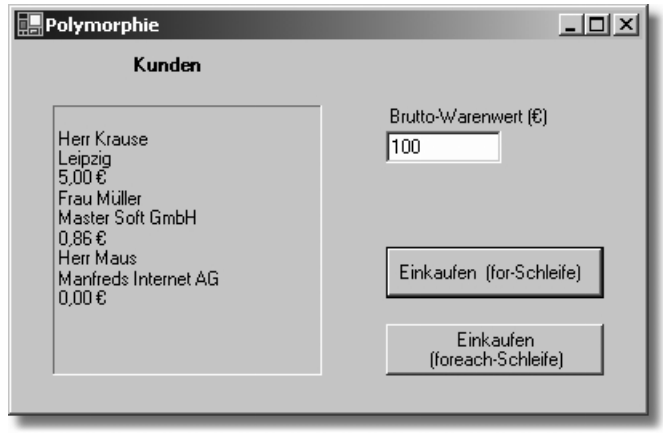
Eine alternative Implementierung mittels *foreach*-Schleife bringt die Polymorphie noch deutlicher ans Tageslicht, da die Methodenaufrufe nicht mit Objekten der Subklassen *CPrivatKunde*/*CFirmenkunde*, sondern mit Objekten der Basisklasse *CKunde* verknüpft sind:

```
private void button2_Click(object sender, System.EventArgs e)
{
```

```
        decimal brutto = Convert.ToDecimal(textBox1.Text);  label1.Text = "";

        foreach (CKunde ku in kunden)
        {
                ku.addGuthaben(brutto);
                label1.Text = label1.Text + LF + ku.adresse() + LF
                        + ku.guthaben.ToString("C");
        }
}
```

Praxistest

Das Ergebnis beweist, dass Vererbung und Polymorphie tatsächlich untrennbar miteinander verbunden sind. Egal ob Privat- oder Firmenkunde – es werden immer die passenden Methodenimplementierungen aufgerufen[1]:



Hinweis: Das tiefere Verständnis der Polymorphie ist mit Sicherheit der schwierigste Part der OOP, deshalb wurde unser Beispiel bewusst einfach gehalten, damit Sie zunächst zu einem Grundverständnis gelangen, welches Sie später weiter ausbauen können.

4.6.6 Abstrakte Klassen

Klassen, die lediglich ihr "Erbmaterial" an andere Klassen weitergeben und von denen selbst keine Instanzen gebildet werden, bezeichnet man als *abstrakt*. Typische Beispiele für abstrakte Klassen wären *Fahrzeug*, *Tier* oder *Nahrung*[1].

[1] Tja, der arme Herr Maus. Weil er kein Stammkunde ist, bekommt er auch kein Bonusguthaben.

Um zu verhindern, dass von abstrakten Klassen Instanzen gebildet werden, müssen diese mit dem Modifikator *abstract* gekennzeichnet werden.

Beispiel: In unserem Vorgängerbeispiel werden von der Klasse *CKunde* keine Instanzen gebildet, sie kann deshalb als abstrakt deklariert werden.

```
public abstract class CKunde
{
...
}
```

Während die Referenzierung nach wie vor möglich ist

```
CKunde kunde;
```

schlägt der Versuch einer Instanziierung fehl:

```
kunde = new CKunde("Herr", "Krause");              // Fehler
```

Abstrakte Methoden

In Verbindung mit polymorphem Verhalten finden sich innerhalb abstrakter Klassen oft auch *abstrakte Methoden*, diese enthalten grundsätzlich keinen Code, da sie in den abgeleiteten Klassen komplett mit *override* überschrieben werden. Zur Kennzeichnung abstrakter Methoden verwenden Sie das Schlüsselwort *abstract.* Die Deklaration erfolgt in einer Zeile, also ohne Rumpf.

Beispiel: Die Funktion *adresse()* der abstrakten *CKunde*-Klasse wird in der Subklasse *CPrivatKunde* komplett überschrieben.

```
public abstract class CKunde
{
...
 public abstract string adresse();             // abstrakte Methode
...
}
```

Da eine abstrakte Methode implizit eine virtuelle Methode darstellt, muss sie in der Subklasse mit *override* deklariert werden.

```
public class CPrivatKunde : CKunde;
{
...
   public override string adresse()           // überschreibt abstrakte Methode
   {
```

---

[1] Können Sie sich vorstellen, wie eine Instanz *Nahrung* konkret aussieht?

```
    return(_anrede + "  " + _name + "  " & _wohnort)
  }
...
}
```

4.6.7 Versiegelte Klassen

Wenn Sie unbedingt verhindern möchten, dass andere Programmierer von einer von Ihnen entwickelten Komponente weitere Subklassen ableiten, so müssen Sie Ihre Klasse mit Hilfe des Modifikators *sealed* schützen.

Beispiel: Die Klasse *CPrivatKunde* wird versiegelt und darf deshalb keine Nachkommen haben.

```
public sealed class CPrivatKunde : CKunde
{
...
}
```

Beim Versuch, davon eine Subklasse abzuleiten, schlägt Ihnen der Compiler erbarmungslos auf die Pfoten:

```
public class CStudent : CPrivatKunde          // Fehler!!!
{
...
}
```

Hinweis: Vererbungsmodifikatoren wie *abstract* und *virtual* führen in einer versiegelten Klasse zum Compilerfehler, da sie keinen Sinn ergeben!

Übrigens: Ein bekanntes Beispiel für eine versiegelte Klasse ist der *string*-Datentyp, was jedweden Begehrlichkeiten einen Riegel vorschiebt.

4.6.8 Einige Ratschläge zur Vererbung

Wenn Sie mit Vererbung arbeiten, sollten Sie Folgendes beachten:

- Es gibt keinerlei Beschränkung bezüglich der Stufenanzahl der Vererbungshierarchie. Sie können die Hierarchie so tief wie nötig staffeln, die Eigenschaften/Methoden werden trotzdem durch alle Vererbungsstufen hindurchgereicht. Allgemein gilt, je weiter unten sich eine Klasse in der Hierarchie befindet, umso spezialisierter ist ihr Verhalten. Zum Beispiel eine *CHochschulKunden*-Klasse, die von einer *CSchulKunden* erbt und diese wiederum von der *CKunden*-Klasse.

- Um die Komplexität zu minimieren und die Wartbarkeit des Codes zu vereinfachen, sollten Sie die Vererbungshierarchie nicht tiefer als ca. vier Stufen staffeln.
- Jede Subklasse kann nur von einer Basisklasse erben! So kann z.B. eine *CHochSchul-Kunden*-Klasse nicht sowohl von der *CKunden*-Klasse und einer *CSchulKunden*-Klasse erben. Das ist in Ordnung so, denn eine solche multiple Vererbung könnte sehr schnell zu einem komplexen, unübersichtlichen und nicht mehr beherrschbaren Ungetüm entarten.

Es gibt zwei primäre Anwendungsfälle für Vererbung in Anwendungen:

- Sie benutzen Objekte unterschiedlichen Typs mit ähnlicher Funktionalität. So erben z.B. *CSchulKunden*-Klasse und *CStaatsKunden*-Klasse von der *CKunden*-Klasse.
- Sie haben gleiche Prozesse mit einer Menge von Objekten auszuführen. So erbt z.B. jeder Typ eines Geschäftsobjekts von einer Business Object(BO)-Klasse.

Sie sollten in folgenden Fällen auf Vererbung verzichten:

- Sie brauchen nur eine einzige Funktion von der Basisklasse. In diesem Fall sollten Sie die Funktion in die eigene Klasse delegieren, anstatt von einer anderen zu erben.
- Sie möchten alle Funktionen überschreiben. In einem solchen Fall sollten Sie eine Schnittstelle (*Interface*) anstatt Vererbung verwenden.

4.6.9 Die Rolle von System.Object

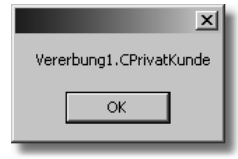
Jedes Objekt in .NET ist von der Basisklasse *System.Object* abgeleitet. Diese Klasse ist Teil des Microsoft .NET Frameworks und beinhaltet die Basiseigenschaften und -methoden, wie sie für ein .NET-Objekt erforderlich sind.

Alle öffentlichen Eigenschaften und Methoden von *System.Object* stehen automatisch auch in jedem Objekt zur Verfügung, welches Sie erzeugt haben. Beispielsweise ist in *System.Object* bereits ein Standardkonstruktor enthalten. Wenn Sie in Ihrem Objekt keinen eigenen Konstruktor definiert haben, wird es mit diesem Konstruktor erzeugt.

Viele der öffentlichen Eigenschaften und Methoden von *System.Object* haben eine Standardimplementation. Das heißt, Sie brauchen selbst keinerlei Code zu schreiben, um sie zu benutzen.

Beispiel: Die *ToString*-Methode liefert den Namen der Anwendungskomponente (die Windows-Anwendung heißt hier *Vererbung1*) und die Klassenzugehörigkeit von *kunde1*.

```
MessageBox.Show(kunde1.ToString());
```



Sie können das standardmäßige Verhalten von *ToString()* mittels *override*-Schlüsselwort verändern. Dies erlaubt Ihnen eine individuelle Implementation einiger Eigenschaften bzw. Methoden von *System.Object*.

Beispiel: Die gleiche *ToString()*-Methode des Vorgängerbeispiels liefert nun den Namen des Kunden, wenn Sie die folgende Methode zum Klassenkörper von *CKunde* hinzufügen.

```
public override string ToString()
{
   return (_name);
}
```

4.7 Objekte in Auflistungen speichern

Wenn Sie mehrere Objekte speichern wollen, so ist es mitunter günstiger, dies anstatt in einem Array (siehe Sprachkapitel, Abschnitt 3.7) in einer Auflistung (Collection) zu tun.

Das .NET-Framework stellt zu diesem Zweck über den Namensraum *System.Collections* zahlreiche Auflistungsklassen zur Verfügung, von denen wir nur auf zwei wichtige Typen näher eingehen wollen.

4.7.1 ArrayList

Die *ArrayList* ist eine nullbasierte Collection, der mittels *Add*-Methode beliebige Elemente hinzugefügt werden. Beim Erzeugen einer *ArrayList* können Sie eine Startkapazität angeben.

Überschreitet die Anzahl der Elemente die Startkapazität, so wird die *ArrayList* automatisch vergrößert. Erzeugen Sie eine *ArrayList* ohne Startkapazität, so wird automatisch der Wert 16 eingestellt.

Beispiel: Drei Nachkommen unterschiedlichen Typs der Klasse *CKunde* werden in einer *ArrayList* (Startkapazität = 3) gespeichert:

```
CPrivatKunde kunde1 = new CPrivatKunde("Herr", "Krause", "Leipzig");
CFirmenKunde kunde2 = new CFirmenKunde("Frau", "Müller", "Master Soft GmbH");
CFirmenKunde kunde3 = new CFirmenKunde("Herr", "Maus", "Manfreds Internet AG");

ArrayList kunden = new ArrayList(3);        // Startkapazität 3
kunden.Add(kunde1);
```

```
kunden.Add(kunde2);
kunden.Add(kunde3);
```

Für alle Kunden-Objekte wird die (polymorphe) *addGuthaben*-Methode aufgerufen:

```
foreach (CKunde ku in kundenA)
{
        ku.addGuthaben(brutto);
}
```

Hinweis: Im Vergleich mit einem normalen Array, das Sie nur ziemlich umständlich dynamisch vergrößern können, schneidet eine *ArrayList* deutlich schneller ab (zum Teil mehr als Faktor 1000!).

Weitere Eigenschaften und Methoden

Ein Direktzugriff auf die einzelnen Elemente entspricht dem bei einem normalen Array, allerdings ist dazu explizite Typkonvertierung erforderlich.

Beispiel: Das Guthaben des ersten Kunden wird angezeigt

```
CKunde kd = (CKunde) kunden[0];                 // Typecasting
label1.Text = kd.guthaben.ToString("C");
```

Mit *AddRange* fügen Sie Elemente aus einer anderen Auflistung hinzu.

Beispiel: Zwei Personen werden der Kundenliste hinzugefügt.

```
ArrayList personen = new ArrayList[2];
...
kunden.AddRange(personen);
```

Interessant dürften weiterhin die Methoden *Clear* (entfernt alle Elemente), *RemoveAt* (entfernt Element am angegebenen Index), *Sort* (Sortieren) und *BinarySearch* (Suchen in sortierter Auflistung) sein, die so wie bei einem normalen Array funktionieren, siehe

☞ R5 ... in einer ArrayList suchen und sortieren?

4.7.2 Hashtable

Die *Hashtable* ist eine Auflistung, die das schnelle Auffinden von Objekten erlaubt. Beim Hinzufügen von Elementen muss allerdings von Ihnen ein eindeutiger (also nicht mehrfach vorkommender) Schlüsselwert angegeben werden. Für den Schlüssel sind beliebige Datentypen möglich, meistens nimmt man dafür *int*- oder *string*-Werte. Allerdings wird in der

Hashtable nicht der Schlüssel gespeichert, sondern dessen so genannter *Hashcode* (eine automatisch ermittelte Integer-Zahl zur eindeutigen Identifizierung des Schlüsselwerts).

Beispiel: Die drei im Vorgängerbeispiel erzeugten Instanzen der Klassen *CPrivatKunde* bzw. *CFirmenKunde* werden mit den Integer-Schlüsseln 101, 102, 103 in einem *Hashtable*-Objekt gespeichert:

```
Hashtable kunden = new Hashtable();

kunden.Add(101, kunde1);
kunden.Add(102, kunde2);
kunden.Add(103, kunde3);
```

Für alle Kunden-Objekte wird die *addGuthaben*-Methode aufgerufen.

```
foreach (CKunde ku in kunden.Values)
{
         ku.addGuthaben(brutto);
}
```

Der direkte Zugriff auf das Element mit dem Schlüssel 102:

```
CKunde kd = (CKunde) kunden[102];
label1.Text = kd.guthaben.ToString("C");
```

Hinweis: Weil Sie die Felder einer *Hashtable* nicht normal indizieren können, ist das Durchlaufen mittels *for*-Schleife nicht möglich, Sie müssen also immer die *foreach*-Schleife verwenden.