4 Klassen und Objekte

Nachdem in C# alles ein Objekt ist, ist es an der Zeit selber Klassen zu schreiben und sie zu verwenden. Vielleicht haben Sie sich durch die Verwendung von Frameworkklassen und Objekten schon ein wenig an die Denkweise der OOP gewöhnt.

Zur Wiederholung: Die OOP zerlegt eine Programmieraufgabe in eine Menge von Objekten. Objekte kapseln zusammengehörige Daten und die Manipulation dieser Daten. Objekte haben zunächst einmal Eigenschaften und Methoden. Im C# Sprachgebrauch erlauben die Eigenschaften oder *properties* den kontrollierten Zugriff auf die Daten. Die Datenwerte (*fields*) werden in der Klasse vereinbart, sie sind privat, d.h. nur innerhalb der Klasse kann man auf die Daten direkt zugreifen. Methoden bestimmen das Verhalten des Objektes, beim Aufruf einer Methode passiert etwas, es wird eine Aufgabe ausgeführt, der Name einer Methode ist daher oft ein Verb, ein "Tun-Wort".

In einer "graphischen" Darstellung ist eine Klasse folgendermaßen aufgebaut:

Calculator	← Name der Klasse
a: double b: double c: double	← Datenwerte
set, get OpA set, get OpB get Result	← Properties
Add() Subtract() Divide() Multiply() PrintResult();	← Operationen, Methoden

Es ist dies eine Klasse, die einen einfachen Rechner implementiert. Ein Rechner ist ein Objekt mit Eigenschaften (den Operanden und dem Resultat) und mit Methoden (den Rechenoperationen).

4.1 Teile einer Klasse

4.1.1 Datenwerte

Datenwerte benötigt man um den aktuellen Zustand des Objektes zu erfassen. Der Rechner benötigt z.B. zwei Operanden und eine Variable für das Resultat der letzten Rechenoperation. In der Regel erlaubt man auf die Datenwerte (*fields*) keinen direkten Zugriff von außen, d.h. sie sind privat. Dies kann man durch das Schlüsselwort `private` noch hervorheben. Macht man keine Angabe zum Gültigkeitsbereich, so ist private die Voreinstellung. Die für ein Objekt charakteristischen Datenwerte werden über Eigenschaften nach außen sichtbar gemacht. Manche Datenwerte sind nur interne Hilfsvariablen. Die Vereinbarung der Datenwerte erfolgt üblicherweise gleich am Beginn einer Klasse. Die Variablen sind in der ganzen Klasse gültig. Außerhalb einer Klasse kann man keine Datenwerte vereinbaren.

Die Bezeichner der Daten beginnen mit einem Kleinbuchstaben. Ist die Programmieraufgabe aus dem Bereich der Mathematik oder Physik, so verwendet man gerne die gewohnten Formelzeichen, z.B. F für Kraft, W für Arbeit. In solchen Fällen greife ich auf eine Konvention aus der in C++ verbreitete ungarische Namenskonvention zurück und entscheide mich für Bezeichner wie mF und mW. Der Buchstabe m steht in der ungarischen Notation für Mem-

bervariable. Die Begriffe Datenwert, Feld, Instanzvariable, Membervariable werden synonym verwendet.

4.1.2 Eigenschaften, *Properties*

Eigenschaften ermöglichen den Zugriff auf die Datenwerte eines Objekts. Sie haben im einfachsten Fall die Form:

```
public Datentyp Name
{
  get { return datenwert; }
  set { datenwert = value; }
}
```

Dabei kann die get- oder die set-Operation auch fehlen. Im get- und set-Block können weitere Anweisungen stehen. Properties erlauben einen kontrollierten Zugriff auf ausgewählte Daten des Objektes. Man kann aber auch Properties schreiben, die einen Wert aus anderen Datenwerten berechnen. Man wird z.B. für ein Rechteck nur Datenwerte für die Länge und Breite definieren und zum Beispiel für die Fläche und den Umfang Properties verwenden:

```
double a, b;
public double Area { get { return a*b; }}
```

Eigenschaften beginnt man mit einem Großbuchstaben. Vielfach nennt man eine Property so wie den zugehörigen Datenwert, verwendet also z.B. name - Name, text - Text.

4.1.3 Methoden

```
public void Add()
{
  c = a + b;
}
```

ist ein Beispiel für die einfachste Form einer Methode. Methode dieser Form benötigten keine zusätzlichen Daten um den Zustand eines Objektes zu ändern oder eine sonstige Operation mit dem Objekt durchzuführen. "public" bedeutet, dass diese Methode der Außenwelt zur Verfügung gestellt wird, also öffentlich ist.

4.1.4 Klassen und Quellcodedateien

Üblicherweise verwendet man pro Klasse eine Datei mit dem Quellcode für die Definition der Klasse. Mit der Version 2 von C# hat man die Möglichkeit von partiellen Klassen eingeführt. Damit kann man den Code von großen Klassen auf mehrere Dateien aufteilen. Visual Studio verwendet dieses Konzept um den vom Designer erzeugten Code vom Code des Programmierers zu trennen.

Realisierung der Calculator-Klasse in C#:

```
using System;

namespace CalculatorClass
{
   public class Calculator
   {
      // private members (fields) - Datenwerte

      double a;              // operand a
      double b;              // operand b
      double c;              // result
      char opCode = '+';     // operator used for last calculation

      // Operations to get and/or set the data values --> properties

      public double OpA
      {
         get { return a; }
         set { a = value; }
      }

      public double OpB
      {
         get { return b; }
         set { b = value; }
      }

      public double Result
      {
         get { return c; }
      }

      // Constructor

      public Calculator()
      {
         // add code for necessary initialization
      }

      // methods

      public void Add()
      {
         c = a + b;
         opCode = '+';
      }

      public void Subtract()
      {
         c = a - b;
         opCode = '-';
      }

.......

      public void PrintResult()
      {
         Console.WriteLine("{0} {1} {2} = {3}", a, opCode, b, c);
         //                                      0     1    2  3
      }
```

```
    }
```

4.1.5 Erzeugung und Verwendung von Objekten

Erzeugt wird ein Objekt mit

```
ObjektType ObjektName = new ObjektType();
```

oder auch auf diese Art:

```
ObjektType o1, o2;
o1 =  new ObjektType();
o2 =  new ObjektType();
```

Properties werden in der Form

```
ObjektName.PropertyName
```

Methoden in der Form

```
ObjektName.MethodeName( parameter-list_opt )
```

verwendet. Während eine Property ohne Klammern verwendet wird, werden Methoden immer mit einer Klammer aufgerufen, auch dann, wenn sie keine Parameter definiert hat.

Verwendung der Klasse:

```
namespace CalculatorClass
{
    class ConsolApplication
    {
       static void Main(string[] args)
       {
          // create a calculator object
          Calculator calculator = new Calculator();

          // use the properties to set the operands
          calculator.OpA = 2.3;
          calculator.OpB = 1.2;

          // use and test the methods
          calculator.Add();
          calculator.PrintResult();
          calculator.Subtract();
          calculator.PrintResult();
          calculator.Divide();
          calculator.PrintResult();
          calculator.Multiply();
          calculator.PrintResult();
          Console.WriteLine();
       }
    }
}
```

4.2 Methoden

Vereinfachte Syntax für die Definition einer Methode:

```
method-modifiers_opt return-type method-name( formal-
  parameter-list_opt )
{
  declarations
  statements
}
```

Die *method-modifiers* legen fest, wer die Funktion verwenden kann (`private`, `public`, …). Wird keine Angabe gemacht, so gilt die Methode als private Methode. Eine private Methode kann nur innerhalb der Klasse selber verwendet werden. Öffentliche (`public`) Methoden können in jedem Programmteil verwendet werden. Für die Zugriffsmodifizierer der Eigenschaften gelten dieselben Regeln.

Wichtig ist noch der Modifizierer `static`. Methoden dieses Typs sind Teil der Klasse und nicht des Objektes. Statische Methoden kann man auch verwenden, ohne eine Objekt erzeugt zu haben. Alle Methoden `Console.`*`MethodName`* oder die Methoden `Math.`*`MethodName`* sind statische Methoden.

4.2.1 Rückgabewert und Argumente

Return-type ist entweder `void` oder ein Datentyp. Liefert eine Methode ein Resultat, so ist "void" durch den Datentyp des Resultates zu ersetzen. Die Methode wird dann mit

```
return wert;
```

beendet und legt damit "wert" als Resultat fest. Die Syntax verlangt für eine Methode mit einem Rückgabewert nach dem `return` einen Ausdruck, dessen Resultat den richtigen Datentyp hat. Methoden mit einem Rückgabewert benötigen für jeden möglichen Programmablauf eine return-Anweisung. Methoden ohne Rückgabewert benötigen nicht unbedingt eine return-Anweisung. Fehlt die Anweisung, dann endet die Methode mit der letzten ausführbaren Programmzeile.

```
public bool CheckOverload()
{
  if (current > maxCurrent)
    return true;
  else
    return false;
}
```

Beispiele für Methoden:

```
(1) void PrintStatus() { … }
(2) public void Add() { … }
(3) public double Add(double a, double c) { … }
(4) public static int Parse(string s) { … }
```

(1) Eine private Methode mit dem Namen PrintStatus. Die Parameterliste ist leer und die Methode liefert keinen Rückgabewert.

(2) Eine öffentliche Methode mit einer leeren Parameterliste und ohne Rückgabewert.

(3) Eine öffentliche Methode mit zwei Parametern vom Typ double und einem Rückgabewert vom Typ double.

(4) Eine öffentliche statische Methode. Verlangt eine Zeichenkette als Argument und liefert einen int Wert als Resultat.

Beim Aufruf, d.h. der Verwendung einer Methode muss für jeden formalen Parameter ein aktueller Parameter übergeben werden. Der Typ des aktuellen Parameters muss stimmen.

Aufruf der obigen Methoden:

```
(1) PrintStatus();
(2) Add();
(3) double a = 3.5;
    double c;
    c = Add (a, 2.0);
(3) int iValue;
    string input = Console.ReadLine();
    iValue = Parse(input);
```

Vielleicht ist der Vergleich mit Funktionen in der Mathematik oder in Excel hilfreich. Eine Funktion hat immer einen Namen, eine Parameterliste und ein Resultat. In Excel werden die Parameter durch Strichpunkt getrennt, in C# durch einen Beistrich.

Will man in der Mathematik ausdrücken, dass eine Funktion f von den drei Größen x, y und z abhängig ist, schreibt man auch f (x, y, z).

In älteren Programmiersprachen (PASCAL, FORTAN, VBA) gibt es Funktionen und Prozeduren. Die Funktionen haben wie in der Mathematik immer einen Rückgabewert (ein Resultat). Prozeduren haben keinen Rückgabewert. In der Parameterliste unterscheiden sich Funktionen und Prozeduren auch in PASCAL oder VBA nicht. Die OOP und C# verwendet den Begriff Methode. Eine Methode kann entsprechend dem alten Sprachgebrauch eine Funktion (mit Rückgabewert) oder eine Prozedur (ohne Rückgabewert → void) sein.

Typische Fehler:

Am Ende der Kopfzeile einer Methode einen Strichpunkt setzen.

```
public void Test();
{
  ...
}
```

4.2.2 Übergabemechanismus

***by value* - Werte-Parameter**

Methoden können also eine beliebige Anzahl von Argumenten übernehmen. Bei der Vereinbarung einer Methode sind die Argumente als Parameterliste anzugeben. Eine Parameterliste ist eine durch Beistrich getrennte Aufzählung von Datentyp und Name.

```
double Pow (double x, int n)
{
}
```

Im Rumpf der Methode sind die Parameter lokale Variablen, die mit den übergebenen Werten initialisiert werden.

Im Normalfall wird bei der Übergabe eines Arguments eine Kopie des Wertes erzeugt, diese Kopie kann als lokale Variable in der Methode verwendet werden. Von einer eventuelle Änderung an der Kopie wird das Original nicht berührt. Das ist auch gut so, denn die Aufgabe einer Methode ist es in der Regel nicht, Argumente zu ändern. Sie verwendet die Argumente und liefert dann ein Resultat.

Zur Demonstration ein Beispiel:

```
public void PassingByValue()
{
  int a = 2;
```

```
    Console.WriteLine("Caller: a = {0}", a);
    Console.WriteLine("Caller: Call ByValue(a)");
    ByValue(a);
    Console.WriteLine("Caller: a = {0}", a);

    int b = 4;
    Console.WriteLine("Caller: b = {0}", b);
    Console.WriteLine("Caller: Call ByValue(b)");
    ByValue(b);
    Console.WriteLine("Caller: b = {0}", b);
  }

  public void ByValue(int a)
  {
    Console.WriteLine("In method ByValue: a = {0}", a);
    Console.WriteLine("In method ByValue: a = 2*a;");
    a = 2 * a;
    Console.WriteLine("In method ByValue: a = {0}", a);
  }
```

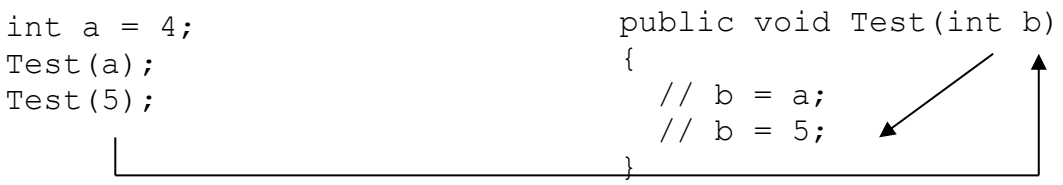
Eine Ausführung der Methode `PassingByValue` ergibt folgende Ausgaben:

Caller: a = 2
Caller: Call ByValue(a)
In method ByValue: a = 2
In method ByValue: a = 2*a;
In method ByValue: a = 4
Caller: a = 2

Caller: b = 4
Caller: Call ByValue(b)
In method ByValue: a = 4
In method ByValue: a = 2*a;
In method ByValue: a = 8
Caller: b = 4

Das demonstriert den verwendeten Mechanismus sehr deutlich, man bezeichnet diese Art der Übergabe von Argumenten an eine Methode "*passing by value*". Die so definierten Parameter nennt man Werteparameter (*value parameters*). Ich zeige das absichtlich einmal mit einer Variablen a und dann mit einer Variablen b. Die Bezeichner der Argumente sind völlig unabhängig von den Bezeichner der formalen Parameter.

Die Übergabe eines Arguments ist im Prinzip eine Zuweisung eines Wertes an die lokale Variable der Methode aus der Parameterliste.



***by ref* - Referenz-Parameter**

Will man auf die bisherige Art eine Methode schreiben, die zwei Zahlen tauschen soll, so ergibt sich ein Problem:

```
public void Swap1(double a, double b)
{
  double t = a;
  a = b;
  b = t;
}
```

Mit dieser Methode gelingt es nicht, zwei Zahlen, die außerhalb der Methode vereinbart sind, zu tauschen.

```
Swap1(a, b);
```

Das liegt an dem Übergabemechanismus, der Kopien erzeugt und dann mit diesen Kopien arbeitet.

Um den gewünschten Effekt zu erreichen, muss man die Parameter jetzt als Referenzen definieren:

```
public void Swap2(ref double a, ref double b)
{
  double t = a;
  a = b;
  b = t;
}
```

und die Methode folgendermaßen aufrufen:

```
double a = 4.2;
double b = 5.8;
Swap2 (ref a, ref b);
```

Das `ref` Schlüsselwort ändert den Übergabemechanismus ziemlich dramatisch. Anstatt einer Kopie des Wertes erhält die Methode eine Referenz auf das Argument, das ist im Prinzip die Adresse des Speicherplatzes des Arguments. Damit hat die Methode Vollzugriff auf das Argument, kann es also ändern. Während man beim *by value* Mechanismus auch einen Wert übergeben kann, kann man beim *by ref* Mechanismus nur eine bereits initialisierte Variable übergeben.

***out* - Ausgabe-Parameter**

Der Übergabemechanismus für diese Art von Argumenten ist wie bei den Referenz-Parametern. Man kennzeichnet die Parameter mit dem Schlüsselwort out und verwendet diesen Mechanismus dann, wenn eine Methode mehr als ein Resultat liefern soll. Ein Beispiel ist die Berechnung von Polarkoordinaten aus kartesischen Koordinaten:

```
double x = 4;
double y = 2;
double r, phi;
ToPolar(x, y, out r, out phi);
Console.WriteLine("x = {0:f2} ; b = {1:f2} <-> r = {2:f2} ;
phi = {3:f2}",    x, y, r, phi);

public void ToPolar(double x, double y,
  out double r, out double phi)
{
  r = Math.Sqrt(x * x + y * y);
```

```
    phi = Math.Atan2(y, x);
}
```

Einen Unterschied gibt es jedoch: Während `ref` Argumente vorher initialisiert werden müssen, müssen `out` Argumente nicht initialisiert sein, sie sind ja nur als Ausgabewerte (Schreibzugriff) und nicht als Ein- und Ausgabewerte (Lese- und Schreibzugriff) konzipiert. Der Compiler überprüft auch, ob ein Ausgabeparameter tatsächlich in der Methode einen Wert zugewiesen bekommt.

Zwei wichtige Methoden aus dem Framework, die Ausgabeparameter verwenden:

```
bool int.TryParse (string text, out int value)
```

und

```
bool double.TryParse (string text, out double value)
```

Referenzen

Die vorige Behauptung, dass ein Parameter in einer Methode geändert werden kann, ohne dass sich das Original ändert, ist nicht ganz richtig. Die Behauptung stimmt nur für Wertetypen, welche mit ihrem Standardübergabemechanismus übergeben werden.

C# unterscheidet zwischen Werte- und Referenztypen (*value types and reference types*). Alle einfachen Datentypen sind Wertetypen. Zu den Referenztypen gehören Arrays und Objekte. Es ist wichtig den Unterschied zwischen Werte- und Referenztypen zu verstehen. Dazu ein typischer Programmteil. Zwei Werte des Typs double und ein Taschenrechnerobjekt sind vereinbart.

```
double a = 3.4;
double b;
Calculator calculator;
```

Dann schaut das im Speicher folgendermaßen aus:

Adresse	Inhalt	Kommentar
2012	3.4	Speicher für Variable a
2004	0.0	Speicher für Variable b
2000	0 (null)	Speicher für Objektreferenz `calculator`

Die Variable a ist mit 3.4 initialisiert, b ist automatisch mit 0.0 initialisiert und `calculator` hat den Wert null, was soviel wie eine nicht definierte Referenz bedeutet. Nach den beiden weiteren Anweisungen

```
b = 5.6;
calculator = new Calculator();
```

schaut die Sache dann so aus: Das `Calculator` Objekt benötigt natürlich auch Speicherplatz, dieser Platz wird in einem Speicherbereich, dem so genannten Heap reserviert, sagen wird ab der Adresse 20200.

Adresse	Inhalt	Kommentar	Speicher
20200	Calculator	Speicher für das Calculator-Objekt	Heap

2012	3.4	Speicher für Variable a	Stack
2004	5.6	Speicher für Variable b	
2000	20200	Speicher für Objektreferenz `calculator`	

Während bei Wertetypen die Werte selber gespeichert (ein double-Wert benötigt 8 Byte), speichert eine Referenz nur, wo denn das zugehörige Objekt tatsächlich im Speicher zu finden ist, eben eine Referenz auf das Objekt. In C oder C++ nennt man das einen Zeiger (*pointer*). Die neueren Prozessoren verwenden für Adressen 64 Bit und können damit einen unvorstellbaren Adressraum von $1{,}84 \cdot 10^{19}$ Byte adressieren. Prozessoren mit 32-Bit Adressen können ca. 4 GByte adressieren.

Die Adressen für die Daten, Properties und Methoden eines Objektes werden dann aus dieser einen Objektreferenz berechnet. Darum muss man sich als Programmierer zum Glück nicht kümmern.

Bei der Übergabe einer Referenz nach dem standardmäßigen *by value* Mechanismus wird ebenfalls eine Kopie der Referenz erstellt, es gibt dann eben eine zweite Referenz auf das zugehörige Objekt. Es gibt jedoch deswegen keine Kopie des Objektes, d.h. die Methode erreicht das Objekt zwar über eine Kopie der Referenz, sie arbeitet jedoch mit dem originalen Objekt. Dazu noch ein Beispiel mit einem Array:

```
public void SortNumbers(int[] numbers)
{
  bool sorted;
  int n = numbers.Length;
  do
  {
    sorted = true;
    for (int i = 0; i < n - 1; i++)
    {
      if (numbers[i] > numbers[i + 1])
      {
        int t = numbers[i];
        numbers[i] = numbers[i + 1];
        numbers[i + 1] = t;
        sorted = false;
      }
    }
  } while (!sorted);
  // to show that numbers is a copy of the passed array
reference
  numbers = null;
}
```

Aufruf der Methode:

```
Random r = new Random();
int[] ai = new int[10];
for (int i = 0; i < 10; i++)
{
  ai[i] = r.Next(100);
  Console.Write("{0,3:d} ", ai[i]);
}
Console.WriteLine();
p.SortNumbers(ai);
for (int i = 0; i < 10; i++)
{
  Console.Write("{0,3:d}", ai[i]);    // ai is still alive
}
```

Ausgabe des Programms:

```
 53   1  54  80  42  48  26  71  24  87
  1  24  26  42  48  53  54  71  80  87
```

4.2.3 Methoden überladen

Bei der Verwendung von Methoden aus dem Framework wird oft eine ganze Liste von Methoden angezeigt, die denselben Namen haben, aber eine unterschiedliche Argumentliste. Das ist in C# eine prinzipielle Eigenschaft von Methoden. Die Signatur einer Methode ergibt sich aus dem Namen (incl. Namespace) und der Argumentliste.

4.3 Konstruktoren und Destruktoren

Der Konstruktor einer Klasse hat den Zweck, bei der Erzeugung eines neuen Objektes Standardeinstellungen vorzunehmen. Dies kann man zwar teilweise auch mit der Initialisierung von Objektvariablen erreichen, in einem Konstruktor hat man jedoch mehr Möglichkeiten, vor allem kann man wie bei einer Methode Argumente übernehmen.

Konstruktor für eine Klasse `Student`:

```
public Student(string name, float grade, float absenceHours)
{
  this.name = name;
  this.grade = grade;
  this.absenceHours = absenceHours;
}
```

Der Konstruktor hat immer den gleichen Namen wie die Klasse, er hat keinen Rückgabewert und ist auch nicht mit `void` gekennzeichnet. Die hier gezeigte Form ist übliche Programmierpraxis. Um die Argumente von den Klassenvariablen zu unterscheiden, muss man die Klassenvariablen mit `this`, das ist der Bezug auf die Klasse selber, verwenden.

Ausschnitt aus der Klasse `Student`:

```
public class Student
{
  float grade;          // average grade
  float absenceHours;   // number of absences from school
  string name;
  ...
}
```

Verwendung des Konstruktors:

```
Student st1 = new Student("Max Motiviert", 1.8F, 12);
Student st2 = new Student("Sigi Schulschwänzer", 3.8F, 86);
```

Anmerkung: 1.8F ist eine `float` Konstante, 1.8 wäre eine `double` Konstante, und double kann nicht automatisch nach `float` konvertiert werden. 12 ist eine `int` Konstante, sie kann automatisch nach `float` konvertiert werden.

Genauso wie Methoden kann man auch Konstruktoren überladen. Jede Klasse hat automatisch einen leeren parameterlosen Konstruktor. Dieser verschwindet, wenn man explizit einen weiteren Konstruktor schreibt. Will man den Standardkonstruktor weiterhin verwenden, so muss man ihn auch explizit deklarieren.

```
public Student()
{
}
```

Destruktor

Der Destruktor wird aufgerufen, wenn ein Objekt gelöscht, also schlussendlich aus dem Speicher entfernt wird. Dieses Thema steht in engem Zusammenhang mit dem sehr fundamentalen Konzept der "*Garbage Collection*" in C# bzw. .NET. Daraus ergibt sich auch ein wichtiger Unterschied zu C++. In C++ ist der Zeitpunkt, zu dem ein Destruktor zum Einsatz kommt genau bekannt. Vereinbarte Objekte werden entfernt, wenn die Methode zu Ende ist, dynamisch mit `new` erzeugte Objekte musste der Programmierer auch explizit wieder mit `delete` löschen. In C# kümmert sich um das Entfernen der Objekte der Garbage Collector. Im wesentlichen prüft er periodisch, ob es auf Objekte noch gültige Referenzen gibt. Gibt es keine gültigen Referenzen mehr, so löscht er das Objekt, d.h. er gibt den vom Objekt verwendeten Speicherplatz wieder frei. Dabei wird der Code in einem allenfalls explizit deklarierter Destruktor durchlaufen. Destruktoren muss man in C# nur in Ausnahmefällen schreiben.

Um eine Objektreferenz explizit als ungültig zu erklären, genügt es, sie auf `null` zu setzen:

```
st1 = null;
```

Der *Garbage Collector* kümmert sich dann beim nächsten Durchgang um das Entfernen aus dem Speicher.

Student Destruktor:

```
~Student()
{
  Debug.WriteLine("Student destructor called");
}
```

Die Idee, den Destruktor mit Tilde (~) und folgendem Klassennamen zu schreiben hängt mit der Verwendung der Tilde als Einerkomplement Operator zusammen, die Tilde symbolisiert die Umkehrung des Konstruktors.

Man kann den Destruktor nicht aufrufen. Er wird vom GC aufgerufen. Schreiben Sie nicht gedankenlos Destruktoren, was im Hintergrund passiert, ist nicht gerade trivial. Will man selber explizit Objekte löschen, so macht man das über die Methode Dispose(), die man dann implementieren muss. Das ist aber ein Thema für sehr fortgeschrittene Programmierer. Die Methode `Dispose` ist wichtig, wenn ein Objekt so genannten "*Unmanaged Code*" verwendet. Unmanaged Code ist Code außerhalb des .NET Frameworks und wird intern öfters verwendet, als man glaubt, z.B. im Zusammenhang mit den ganzen Zeichenoperationen oder beim Zugriff auf Files.

4.4 Collections

Collections oder Listen sind eine der häufigsten Strukturen zur Organisation von Daten. Ein Objekt kapselt eine Reihe von zusammengehörigen Daten, die oft vom Typ eines Datensatzes sind, wie man ihn zum Beispiel in einer Tabelle verwendet. Eine Zeile ist die Zusammenfassung von Daten, die in Spalten organisiert sind. Die Menge der Zeilen ist eine Sammlung oder Liste von Datensätzen. Im Prinzip kann man so eine Datenstruktur mit einem Array realisieren. Das Array hat jedoch eine Reihe von Nachteilen. Die Größe eines einmal erzeugten Arrays kann man nachträglich nur mehr schwer ändern, das Einfügen und Löschen von Elementen muss selber programmiert werden.

Wesentlich mehr Komfort bietet eine Collection oder dynamische Liste. Typischerweise Implementiert eine Collection mindestens folgende Methoden und Eigenschaften.

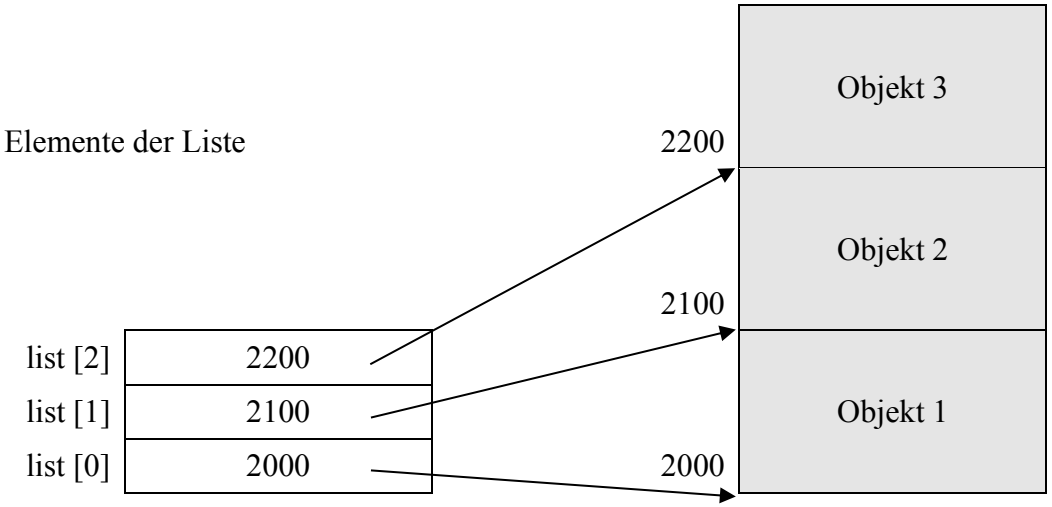
Methoden:

`Add`	neues Objekt hinzufügen
`Clear`	Entfernt alle Objekte aus der Liste
`Contains`	prüft ob ein bestimmtes Objekt in der Liste enthalten ist
`IndexOf`	Liefert den 0 basierten Index eines Objektes
`Insert`	Fügt ein Objekt an einer bestimmten Stelle ein
`Remove`	entfernt ein Objekt
`RemoveAt`	entfernt das Objekt an einer bestimmten Position

Properties:

`Count`	Anzahl der Objekte in der Liste
`Indexer`	erlaubt indizierten Zugriff

Es lohnt sich, im Zusammenhang mit einer Liste nochmals die Situation im Speicher zu studieren. Wir ein Objekt erzeugt, so wird dafür im Speicher natürlich Platz benötigt. Um das Objekt im Speicher zu adressieren, d.h. auch wieder zu finden, muss man sich eine Referenz auf das Objekt merken. Die Liste merkt sich die Referenzen der "gesammelten" Objekte.



Die Objekte enthalten ihre Datenfelder und eine Tabelle mit Zeigern auf die Methoden. Um innerhalb eines Objektes die einzelnen Datenfelder usw. zu finden, muss man wissen, von welchem Typ das Objekt ist, d.h. man muss die zugehörige Klasse kennen.

Das Framework bieten eine Fülle von Collections an:

4.4.1 List<T>

Mit der Version 2 von C# wurden einige wichtige Verbesserungen eingeführt. Dazu gehört auch generische Klassen en und Methoden. Dieses Konzept erlaubt es, Operationen in einer Form zu implementieren, die unabhängig von einem bestimmten Datentyp ist. Ein Beispiel für eine solche Klasse ist die List<T>. T steht für den austauschbaren Datentyp.

Beispiel:

Klasse Student

```
using System;
using System.Collections.Generic;
using System.Diagnostics;

namespace StudentCollection
{
   public class Student
   {
      string name;
      float grade;
      float absenceHours;

      public float Grade { get { return grade; } set { grade = value; } }
      public float AbsenceHours
      { get { return absenceHours; } set { absenceHours = value; } }
      public string Name { get { return name; } set { name = value; } }

      public Student() {}

      public Student(string name, float grade, float absenceHours)
      {
         this.name = name;
         this.grade = grade;
         this.absenceHours = absenceHours;
      }
   }
```

Oft ist es eine gute Lösung, eine Sammlung von Objekten wieder in einer Klasse zu kapseln. Man kann dann die Funktionalität auf das reduzieren, was man braucht, man kann weitere Methoden hinzufügen und diese Vorgangsweise erleichtert auch die saubere Trennung von Daten und Ansicht (Benutzerschnittstelle). Das zeigt die Klasse `Students`. Beachte auch die Namen `Student` (Einzahl, für einen Studenten) und `Students` (für die Liste, Sammlung der Studenten).

Klasse Students:

```
...

namespace StudentCollection
{
   class Students
   {
      List<Student> list;

      public Students()
      {
         list = new List<Student>();
      }

      public void Add(string name, float grade, float absenceHours)
```

```
      {
         Student s = new Student(name, grade, absenceHours);
         list.Add(s);
      }

      public int Count { get { return list.Count; } }

      public Student this[int index] { get { return list[index]; } }

      public Student GetStudent(int i)
      {
         if (i < list.Count)  // mit Index-Überprüfung
            return list[i];
         else
            return null;
      }

      public float AverageGrade()
      {
         int n = list.Count;
         if (n == 0) return 0.0F;
         Student s;
         float sum = 0;
         for (int i = 0; i < n; i++)
         {
            sum += list[i].Grade;
         }
         return sum / n;
      }

      public string DumpToString()
      {
         string text = string.Empty;
         foreach (Student s in list)
         {
            text += string.Format("{0,-25}: {1,6:f1} {2,6:f0}\n",
               s.Name, s.Grade, s.AbsenceHours);
         }
         return text;
      }

   } // end Students
}
```

Indexer

Das Beispiel verwendet auch einen Indexer. Ein Indexer ist eine Sonderform einer Property und ist für Collections in der Regel implementiert.

```
public Student this[int index]
{
  get { return list[index]; }
}
```

Er erlaubt die Verwendung der Indexschreibweise zur Auswahl eines Elementes. Man kann nur einen Indexer pro Klasse schreiben, im Unterschied zu einer Eigenschaft hat der Indexer ja keinen Namen. Bei der Deklaration wird statt des Namens `this` verwendet.

Man kann `get` und /oder `set` implementieren.

4.4.2 ArrayList

Das ist eine Collection, die ohne die Techniken der generischen Klassen auskommt. Sie sammelt Objekte als Objekte des allgemeinen Basis-Typs aller Objekte (object). Man muss diese Objekte dann mit einem Type-Cast versehen, um sie als Originalobjekte verwenden zu können. Typs Dies erreicht man durch folgenden typischen Code.

Namespace verwenden:

```
using System.Collections;
```

`ArrayList` vereinbaren und erzeugen:

```
ArrayList students;
students = new ArrayList();
```

oder

```
ArrayList students = new ArrayList();
```

Objekte erzeugen und in die Liste einfügen:

```
Student s = new Student(name, grade, absenceHours);
students.Add(s);
s = new Student("Fritz Meier", 2.2F, 25);
students.Add(s);
students.Add(new Student("Otto Lustig", 2.8F, 45));
```

s ist hier nur eine Hilfsreferenz. Solange eine Objektreferenz in der Liste enthalten ist, bleibt das Objekt auf jeden Fall erhalten.

Die ArrayList selber merkt sich nicht, welchen Typ die Objekte haben. Der Indexer liefert nur eine typlose Objektreferenz. Will man mit dem Objekt dann arbeiten, dann muss man das Resultat des Indexers in den gewünschten Objekttyp umwandeln.

Ausdruck	Typ
`students[i]`	object
`(Student) students[i]`	Student
`students[i] as Student`	Student

(Students) students[i]

ist der von C herrührende Stil des unären Cast-Operators wie man ihn auch verwendet um einen einfachen Datentyp in einen anderen einfachen Datentyp umzuwandeln:

```
int code = 65;
char c = (char) code;
```

Mehr im Stil von Visual Basic ist diese Möglichkeit:

```
students[i] as Student
```

Um z.B. die Liste der Studenten im Debug/Output Fenster anzuzeigen benötigt man folgenden Code:

```
for (int i = 0; i < students.Count; i++)
{
  Student s = (Student) students[i];              // so
  // Student s = students[i] as Student;          // oder so
  Debug.WriteLine(string.Format("{0,-25}: {1}",
    s.Name, s.Grade, s.AbsenceHours));
}
```

Eine weitere, sehr elegante Möglichkeit ist:

```
foreach (Student s in students)
```

```
{
  Debug.WriteLine(
    string.Format("{0,-25}: {1:f1}", s.Name, s.Grade));
}
```

4.5 Strukturen

Strukturen sind in C, C++ und auch in C# eine Möglichkeit, zusammengehörige Daten in einem benutzerdefinierten Datentyp zusammenzufassen. Strukturen haben Datenfelder, Properties, Methoden und Konstruktoren und auch die Syntax für die Definition unterscheidet sich nur in wenigen Details von der Definition einer Klasse.

```
namespace Structures
{
   struct Time
   {
     int hours;
     int minutes;  // = 0 -> cannot have instance field
     int seconds;  //         initializers

     public int Hours
     { get { return hours; }
       set { hours = value % 24; }
     }

     public int Minutes
     { get { return minutes; }
       set
       {
         Hours += value / 60;
         minutes = value % 60;
       }
     }

     public int Seconds
     { get { return seconds; }
       set
       {
         Minutes += value / 60;
         seconds = value % 60;
       }
     }
```

```
    public override string ToString()
    {
      return string.Format("{0,2:d2}:{1,2:d2}:{2,2:d2}",
                   hours, minutes, seconds);
    }

    public Time(int hh, int mm, int ss)
    {
      hours = 0;
      minutes = 0;
      seconds = 0;
      // cannot use properties before initialization
      Hours = hh;
      Minutes = mm;
      Seconds = ss;
    }

    /*
    Structs cannot contain explizit parameterless
    constructors

    public Time()
    {
    }

    */
  }

  class Program
  {
    static void Main(string[] args)
    {
      Time t1 = new Time();
      Time t2 = new Time(12, 34, 30);
      Time t3;
      Console.WriteLine(t1.ToString());
      Console.WriteLine(t2.ToString());

      // Console.WriteLine(t3.ToString());  -> Use of
         unassigned local variable t3;

      t3 = t1;
      t3.Hours = 6;
      Console.WriteLine(t1.ToString());
      Console.WriteLine(t3.ToString());

      // struct arrays

      Console.WriteLine("\nArray of Time\n");

      Time [] times = new Time[3];
      times[1] = new Time(1, 12, 3);
      int i;
      for (i = 0; i < 3; i++)
      {
        Console.WriteLine("{0:d} {1}",
```

```
                i, times[i].ToString());
            }

            Console.ReadLine();
        }
    }
}
```

Ausgaben des Programms:

```
00:00:00
12:34:30
00:00:00
06:00:00

Array of Time

0 00:00:00
1 01:12:03
2 00:00:00
```

Unterschiede zwischen Klassen und Strukturen:

- Strukturen sind Wertetypen und werden am Stack erzeugt, Klassen sind Referenztypen.
- Für Strukturen kann man keinen parameterlosen Standardkonstruktor schreiben.
- Der Compiler generiert immer selber einen Standardkonstruktor, der alle Datenfelder mit 0, `false` oder `null` initialisiert.
- In einem Konstruktor mit Parametern muss man jedoch alle Felder selber initialisieren.
- In einer Struktur kann man Felder nicht bei der Deklaration initialisieren.
- Da Strukturen Wertetypen sind, muss man sie nicht mit `new` erzeugen. Man kann auf die Felder einer Struktur dann jedoch zunächst nicht zugreifen.
- Bei einer Struktur sorgt ein `new` nur für den Aufruf des "Konstruktors".
- `Time [] times = new Time[3];`

sorgt auch für alle drei Zeiten für den Aufruf des Konstruktors, d.h. die Felder der Zeiten sind (mit 0) initialisiert.

- `t3 = t1;` kopiert alle Felder der Zeit `t1, t3` ist eine unabhängige Kopie von `t1`. Wäre Time eine Klasse, dann wäre `t3` nur eine zweite Referenz auf dasselbe Objekt.

Diese Besonderheiten werden in obigem Programmbeispiel demonstriert.

Verwende Strukturen nur dann, wenn es darum geht hauptsächlich einige wenige Daten zusammenzufassen. Beispiele für Strukturen in Framework: `Point, PointF, Rectangle`