Universität Trier

Bernhard Baltes-Götz

Einführung in das Programmieren mit C# 6

2017 (Rev. 170515)

Herausgeber: Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier

Universitätsring 15
D-54286 Trier

WWW: zimk.uni-trier.de

E-Mail: zimk@uni-trier.de

Autor: Bernhard Baltes-Götz

WWW: https://www.uni-trier.de/~baltes

E-Mail: baltes@uni-trier.de

Vorwort

Dieses Manuskript entstand als Begleitlektüre zum C# - Einführungskurs, den das Zentrum für Informations-, Medien- und Kommunikationstechnologie (ZIMK) an der Universität Trier im Wintersemester 2016/2017 angeboten hat, sollte aber auch für das Selbststudium geeignet sein.

Lerninhalte und -ziele

C# ist eine von der Firma Microsoft für das .NET - Framework entwickelte und von der ECMA[1] standardisierte Programmiersprache, die auf den Vorbildern C++ (siehe z.B. Baltes-Götz 2003) und Java (siehe z.B. Baltes-Götz & Götz 2016) aufbaut, aber auch etliche Weiterentwicklungen bietet.

Das .NET - Framework hat sich als Standard für die Softwareentwicklung unter Windows etabliert und ist durch das Open Source - Projekt *Mono* erfolgreich auf andere Betriebssysteme (Linux, MacOS X) portiert worden.[2] Neben Mono entwickelt gerade das von Microsoft betriebene Open Source - Projekt *.NET Core* eine Plattform-unabhängige, unter Linux, Mac OS X und Windows verfügbare .NET - Variante. Derzeit (April 2017) werden das .NET - Framework (aktuelle Version: 4.6) und .NET Core (aktuelle Version: 1.1) parallel entwickelt. Im Kurs wird dem ausgereifteren .NET Framework, das für alle aktuellen Windows-Versionen inklusive graphischer Bedienoberfläche verfügbar ist, der Vorzug gegeben.

Von der Firma Xamarin, die einst die Mono-Entwicklung initiiert hat und mittlerweile von Microsoft übernommen worden ist, stammt ein attraktiver Ansatz zur Entwicklung mobiler Apps für Android, iOS und Windows Phone in C#.[3]

Ein Vorteil des .NET - Frameworks ist die freie Wahl zwischen verschiedenen Programmiersprachen, doch kann C# trotz der großen Konkurrenz als die bevorzugte .NET - Programmiersprache gelten. Schließlich wurde das Framework selbst überwiegend in C# entwickelt.

Wenngleich die Netzorientierung im Namen des neuen Software-Frameworks stark betont wird, ist C# wie jede andere .NET - Programmiersprache universell einsetzbar. Das Manuskript behandelt wesentliche Konzepte und Methoden der objektorientierten Softwareentwicklung (z.B. elementare Sprachelemente, Klassen, Vererbung, Polymorphie, Schnittstellen) und berücksichtigt viele Standardthemen der Programmierpraxis (z.B. grafische Benutzeroberflächen, Ausnahmebehandlung, Dateizugriff, Multithreading).

Voraussetzungen bei den Kursteilnehmern

- **Programmierkenntnisse werden *nicht* vorausgesetzt.**
  Teilnehmer *mit* Programmiererfahrung werden sich in den ersten Kapiteln eventuell etwas langweilen. Für diesen Personenkreis enthält das Manuskript einige Vertiefungen (mit entsprechender Kennzeichnung in der Überschrift), die von Einsteigern gefahrlos übersprungen werden können.
- **Motivation**
  Es ist mit einem erheblichen Zeitaufwand bei der Lektüre und bei der aktiven Auseinandersetzung mit dem Stoff (z.B. durch das Lösen von Übungsaufgaben) zu rechnen. Als Gegenleistung kann man hochrelevante Techniken erlernen und viel Spaß erleben.

---

[1] Ursprünglich stand EMCA für ***E**uropean **C**omputer **M**anufacturers **A**ssociation*, doch wurde diese Bedeutung abgelegt, um den nunmehr globalen Anspruch der Organisation zu dokumentieren (siehe z.B. http://www.ecma-international.org/).

[2] Webseite des Projekts: http://www.mono-project.com/
Zu Details der Entstehungsgeschichte von Mono siehe https://en.wikipedia.org/wiki/Mono_(software)

[3] https://www.xamarin.com/

Software zum Üben

Für die unverzichtbaren Übungen sollte ein Rechner mit einer aktuellen C# - Entwicklungsumgebung zur Verfügung stehen, z.B. die kostenlose Community-Variante von Microsofts Visual Studio 2015. Das ebenfalls erforderliche .NET - Framework ist auf jedem Rechner unter Windows Vista, 7, 8.x oder 10 als Bestandteil des Betriebssystems vorhanden und wird vom Visual Studio 2015 bei Bedarf auf die Version 4.6 aktualisiert.

Dateien zum Manuskript

Die aktuelle Version dieses Manuskripts ist zusammen mit den behandelten Beispielen und Lösungsvorschläge zu vielen Übungsaufgaben auf dem Webserver der Universität Trier von der Startseite (http://www.uni-trier.de/) ausgehend folgendermaßen zu finden:

Rechenzentrum > Studierende > EDV-Dokumentationen >
Programmierung > Einführung in das Programmieren mit C#

Leider blieb zu wenig Zeit für eine sorgfältige Kontrolle des Textes, so dass einige Fehler und Mängel verblieben sein dürften. Entsprechende Hinweise an die E-Mail-Adresse

baltes@uni-trier.de

werden dankbar entgegen genommen.

Trier, im April 2017 Bernhard Baltes-Götz

Inhaltsverzeichnis

1 Einleitung

Im ersten Kapitel geht es zunächst um die Denk- und Arbeitsweise der objektorientierten Programmierung. Danach werden die .NET - Plattform und ihre wohl wichtigste Programmiersprache C# vorgestellt.

1.1 Beispiel für die objektorientierte Softwareentwicklung mit C#

In diesem Abschnitt soll eine Vorstellung davon vermittelt werden, was ein Computerprogramm (in C#) ist. Dabei kommen einige Grundbegriffe der Informatik zur Sprache, wobei wir uns aber nicht unnötig lange von der Praxis fernhalten wollen.

Ein Computerprogramm besteht im Wesentlichen (von Medien und anderen Ressourcen einmal abgesehen) aus einer Menge von wohlgeformten und wohlgeordneten *Definitionen* und *Anweisungen* zur Bewältigung einer bestimmten Aufgabe. Ein Programm muss ...

- den betroffenen Anwendungsbereich **modellieren**
  Beispiel: In einem Programm zur Verwaltung einer Spedition sind z.B. Kunden, Aufträge, Mitarbeiter, Fahrzeuge, Einsatzfahrten, (Ent-)ladestationen und kommunikative Prozesse (Nachrichten zwischen beteiligten Akteuren) zu repräsentieren.
- **Algorithmen** realisieren, die in endlich vielen Schritten und unter Verwendung von endlich vielen Betriebsmitteln (z.B. Speicher) bestimmte Ausgangszustände in akzeptable Zielzustände überführen.
  Beispiel: Im Speditionsprogramm muss u.a. für jede Tour zu den meist mehreren (Ent-)ladestationen eine optimale Route ermittelt werden (hinsichtlich Entfernung, Fahrtzeit, Mautkosten etc.).

Wir wollen präzisere und komplettere Definitionen zum komplexen Begriff eines Computerprogramms den Lehrbüchern überlassen (siehe z.B. Goll et al. 2000) und stattdessen ein Beispiel im Detail betrachten, um einen Einstieg in die Materie zu finden.

Bei der Suche nach einem geeigneten C# - Einstiegsbeispiel tritt ein Dilemma auf:

- Einfache Beispiele sind für das Programmieren mit C# nicht besonders repräsentativ, z.B. ist von Objektorientierung außer einem gewissen Formalismus nichts vorhanden.
- Repräsentative C# - Programme eignen sich in der Regel wegen ihrer Länge und Komplexität (aus der Sicht des Anfängers) nicht für eine Detailanalyse. Insbesondere können wir das eben zur Illustration einer realen Aufgabenstellung verwendete, aber potentiell sehr aufwendige, Speditionsverwaltungsprogramm jetzt nicht vorstellen.

Wir analysieren stattdessen ein Beispielprogramm, das trotz angestrebter Einfachheit nicht auf objektorientiertes Programmieren (OOP) verzichtet. Seine Aufgabe besteht darin, elementare Operationen mit Brüchen auszuführen (Kürzen, Addieren), womit es etwa einem Schüler beim Anfertigen der Hausaufgaben (zur Kontrolle der eigenen Lösungen) nützlich sein kann.

1.1.1 Objektorientierte Analyse und Modellierung

Einer objektorientierten Programmentwicklung geht die **objektorientierte Analyse** der Aufgabenstellung voran mit dem Ziel einer Modellierung durch kooperierende **Klassen**. Man identifiziert per **Abstraktion** die beteiligten **Objektsorten** und definiert für sie jeweils eine **Klasse**. Eine solche Klasse ist gekennzeichnet durch:

- **Merkmale** (**Instanz- bzw. Klassenvariablen, Felder**)
  Viele Merkmal gehören zu den *Objekten* bzw. *Instanzen* der Klasse (z.B. Zähler und Nenner eines Bruchs), manche gehören zur Klasse selbst (z.B. Anzahl der bei einem Programmeinsatz bisher erzeugten Brüche). Im letztlich entstehenden Programm landet jede Merkmalsausprägung in einer so genannten **Variablen**. Dies ist ein benannter Speicherplatz, der Werte eines bestimmten Typs (z.B. Zahlen, Zeichen) aufnehmen kann. Variablen zur Repräsentation der Merkmale von Objekten oder Klassen werden oft als **Felder** bezeichnet.
- **Handlungskompetenzen** (**Methoden**)
  Analog zu den Merkmalen sind auch die Handlungskompetenzen entweder individuellen Objekten bzw. Instanzen zugeordnet (z.B. Kürzen bei Brüchen) oder der Klasse selbst (z.B. Informieren über die Anzahl der erzeugten Brüche). Im letztlich entstehenden Programm sind die Handlungskompetenzen durch so genannte *Methoden* repräsentiert. Diese ausführbaren Programmbestandteile realisieren die oben angesprochenen Algorithmen. Die Kommunikation zwischen Klassen bzw. Objekten besteht darin, ein anderes Objekt oder eine andere Klasse aufzufordern, eine bestimmte Methode auszuführen.

Eine Klasse …

- beinhaltet meist einen **Bauplan für konkrete Objekte**, die im Programmablauf nach Bedarf erzeugt und mit der Ausführung bestimmter Methoden beauftragt werden,
- kann andererseits aber auch **Akteur** sein (Methoden ausführen und aufrufen).

Weil der Begriff *Klasse* gegenüber dem Begriff *Objekt* dominiert, hätte man eigentlich die Bezeichnung *klassenorientierte Programmierung* wählen sollen. Allerdings gibt es keinen ernsthaften Grund, die eingeführte Bezeichnung *objektorientierte Programmierung* zu ändern.

Dass jedes Objekt gleich in eine Klasse („Schublade“) gesteckt wird, mögen die Anhänger einer ausgeprägt individualistischen Weltanschauung bedauern. Auf einem geeigneten Abstraktionsniveau betrachtet lassen sich jedoch die meisten Objekte der realen Welt ohne großen Informationsverlust in Klassen einteilen. Bei einer definitiv nur *einfach* zu besetzenden Rolle kann eine Klasse zum Einsatz kommen, die ausnahmsweise *nicht* zum Instanzieren (Erzeugen von Objekten) gedacht ist, sondern als Akteur.

In unserem Bruchrechnungsbeispiel ergibt sich bei der objektorientierten Analyse, dass vorläufig nur eine Klasse zum Modellieren von Brüchen benötigt wird. Beim möglichen Ausbau des Programms zu einem Bruchrechnungstrainer kommen jedoch weitere Klassen hinzu (z.B. Aufgabe, Schüler).

Dass Zähler und Nenner die zentralen **Merkmale** eines Bruchs sind, bedarf keiner Begründung. Sie werden in der Klassendefinition durch ganzzahlige Felder (C# - Datentyp **int**) repräsentiert, die folgende Namen erhalten sollen:

- `zaehler`
- `nenner`

Auf das oben angedeutete klassenbezogene Merkmal mit der Anzahl bereits erzeugter Brüche wird vorläufig verzichtet.

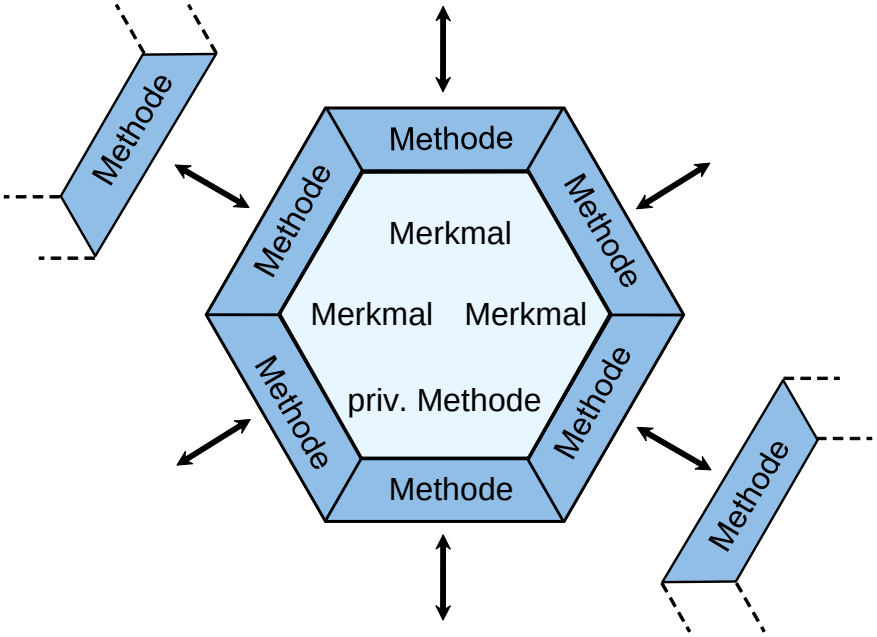
Im objektorientierten Paradigma ist jede Klasse für die Manipulation ihrer Merkmalsausprägungen selbst verantwortlich. Diese sollen **eingekapselt** und vor direktem Zugriff durch fremde Klassen geschützt sein. So kann sichergestellt werden, dass nur sinnvolle Änderungen der Merkmalsausprägungen möglich sind. Außerdem wird aus später zu erläuternden Gründen die Produktivität der Softwareentwicklung durch die Datenkapselung gefördert.

Demgegenüber sind die **Handlungskompetenzen** (Methoden) einer Klasse in der Regel von anderen Klassen und ansprechbar, wobei es aber auch *private* Methoden für den ausschließlich internen

Gebrauch gibt. Die *öffentlichen* Methoden einer Klasse bilden ihre **Schnittstelle** zur Kommunikation mit anderen Klassen.

Die folgende, an Goll et al. (2000) angelehnte Abbildung zeigt für eine Klasse ...

- im gekapselten Bereich ihre Merkmale sowie eine private Methode
- die Kommunikationsschnittstelle mit den öffentlichen Methoden



Die **Objekte** (Exemplare, Instanzen) einer Klasse, d.h. die nach diesem Bauplan erzeugten Individuen, sollen in der Lage sein, auf eine Reihe von **Nachrichten** mit einem bestimmten Verhalten zu reagieren. In unserem Beispiel sollte die Klasse `Bruch` z.B. eine Instanzmethode zum Kürzen besitzen. Dann kann einem konkreten `Bruch`-Objekt durch Aufrufen dieser Methode die Nachricht zugestellt werden, dass es Zähler und Nenner kürzen soll.

Sich unter einem Bruch ein Objekt vorzustellen, das Nachrichten empfängt und mit einem passenden Verhalten beantwortet, ist etwas gewöhnungsbedürftig. In der realen Welt sind Brüche, die sich selbst auf ein Signal hin kürzen, nicht unbedingt alltäglich, wenngleich möglich (z.B. als didaktisches Spielzeug). Das objektorientierte Modellieren eines Anwendungsbereichs ist nicht unbedingt eine direkte Abbildung, sondern eine *Rekonstruktion*. Einerseits soll der Anwendungsbereich im Modell gut repräsentiert sein, andererseits soll eine möglichst stabile, gut erweiterbare und wieder verwendbare Software entstehen.

Um fremden Klassen trotz Datenkapselung die Veränderung einer Merkmalsausprägung zu erlauben, müssen entsprechende Methoden (mit geeigneten Kontrollmechanismen) angeboten werden. Unsere Klasse `Bruch` sollte also über Methoden zum Verändern von Zähler und Nenner verfügen. Bei einem geschützten Merkmal ist auch der direkte *Lese*zugriff ausgeschlossen, so dass im `Bruch`-Beispiel auch noch Methoden zum Ermitteln von Zähler und Nenner erforderlich sind. Eine konsequente Umsetzung der Datenkapselung erzwingt also eventuell eine ganze Serie von Methoden zum Lesen und Ändern von Merkmalsausprägungen.

Mit diesem Aufwand werden aber erhebliche Vorteile realisiert:

- **Stabilität**
  Die Merkmalsausprägungen sind vor unsinnigen und gefährlichen Zugriffen geschützt, wenn Veränderungen nur über die vom Klassendesigner entworfenen Methoden möglich sind. Treten doch Fehler auf, sind diese leichter zu identifizieren, weil nur wenige Methoden verantwortlich sein können.
- **Produktivität**
  Durch Datenkapselung wird die **Modularisierung** unterstützt, so dass bei der Entwicklung großer Softwaresysteme zahlreiche Programmierer reibungslos zusammenarbeiten können. Der Klassendesigner trägt die Verantwortung dafür, dass die von ihm entworfenen Methoden korrekt arbeiten. Andere Programmierer müssen beim Verwenden einer Klasse lediglich die Methoden der Schnittstelle kennen. Das Innenleben einer Klasse kann vom Designer nach Bedarf geändert werden, ohne dass andere Programmbestandteile angepasst werden müssen. Bei einer sorgfältig entworfenen Klasse stehen die Chancen gut, dass sie in mehreren Software-Projekten genutzt werden kann (**Wiederverwendbarkeit**). Besonders günstig ist die Recycling-Quote bei den Klassen der .NET - Standardbibliothek (***F**ramework **C**lass **L**ibrary*, FCL) (siehe Abschnitt 1.2.7), von denen alle C# - Programmierer regen Gebrauch machen. Auch die Klasse `Bruch` aus dem Beispielprojekt besitzt einiges Potential zur Wiederverwendung. Wir werden unser Beispiel noch um die Klasse `Bruchaddition` erweitern, welche `Bruch`-Objekte benutzt, um ein Programm zur Addition von Brüchen zu realisieren.

Im Vergleich zu anderen objektorientierten Programmiersprachen wie z.B. Java und C++ bietet C# mit den so genannten **Eigenschaften** (engl.: **Properties)** eine Möglichkeit, den Aufwand mit den Methoden zum Lesen oder Verändern von Merkmalsausprägungen für den Designer und den Nutzer einer Klasse zu reduzieren. In der Klasse `Bruch` werden wir z.B. zum Feld `zaehler` die *Eigenschaft* `Zaehler` (großer Anfangsbuchstabe!) definieren, welche dem Nutzer einer Klasse *Methoden* zum Lesen und Setzen des Merkmals bietet, wobei dieselbe Syntax wie beim direkten Zugriff auf ein Feld verwenden werden darf. Um dieses Argument zu illustrieren, greifen wir der Beschäftigung mit elementaren C# - Sprachelementen vor. In der folgenden Anweisung wird der `zaehler` eines `Bruch`-Objekts mit dem Namen `b1` auf den Wert 4 gesetzt:

```
b1.Zaehler = 4;
```

Während der Entwickler der Klasse `Bruch` Zugriffs*methoden* bereitzustellen hat (siehe unten), sehen die Nutzer (das sind die Entwickler anderer Klassen) ein öffentliches *Merkmal*. Langfristig werden Sie diese Ergänzung des objektorientierten Sprachumfangs zu schätzen lernen. Momentan ist sie eher eine Belastung, da Sie vielleicht erstmals mit der Grundarchitektur einer Klasse konfrontiert werden, und die fundamentale Unterscheidung zwischen Merkmalen und Methoden einer Klasse durch die C# - Eigenschaften unscharf zu werden scheint. Letztlich erspart eine C# - Eigenschaft wie `Zaehler` dem Nutzer lediglich die Verwendung von Zugriffs*methoden*, z.B.

```
b1.SetzeZaehler(4);
```

Damit kann man die C# - Eigenschaften in die Kategorie *syntactic sugar* (Mössenböck, 2016, S. 3) einordnen, was aber keine Abwertung bedeuten soll. Wegen ihrer intensiven Nutzung in C# - Programmen ist ein Auftritt im ersten Kursbeispiel wohl trotz den angesprochenen didaktischen Probleme gerechtfertigt.

In realen (komplexeren) Programmen wird keinesfalls *jedes* gekapselte Feld über eine Eigenschaft zum Lesen und geschützten Schreiben durch die Außenwelt erschlossen.

Insgesamt sollen die Objekte unserer `Bruch`-Klasse folgende Methoden beherrschen bzw. Eigenschaften besitzen:

- `Nenner` (Eigenschaft zum Feld `nenner`)
  Das Objekt wird beauftragt, seinen `nenner`-Zustand mitzuteilen bzw. zu verändern. Ein direkter Zugriff auf das Merkmal soll fremden Klassen nicht erlaubt sein (Datenkapselung). Bei dieser Vorgehensweise kann ein `Bruch`-Objekt z.B. verhindern, dass sein `nenner` auf 0 gesetzt wird. Wie und wo die Kontrolle stattfindet, ist bald zu sehen.
- `Zaehler` (Eigenschaft zum Feld `zaehler`)
  Das Objekt wird beauftragt, seinen `zaehler`-Zustand mitzuteilen bzw. zu verändern. Die Eigenschaft `Zaehler` bringt im Gegensatz zur Eigenschaft `Nenner` keinen großen Gewinn an Sicherheit. Sie ist aber der Einheitlichkeit und damit der Einfachheit halber angebracht und hält die Möglichkeit offen, das Merkmal `zaehler` einmal anders zu realisieren.
- `Kuerze()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` zu kürzen. Welcher Algorithmus dazu benutzt wird, bleibt dem Klassendesigner überlassen.
- `Addiere(Bruch b)`
  Das Objekt wird beauftragt, den als Parameter (siehe unten) übergebenen Bruch zum eigenen Wert zu addieren.
- `Frage()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` beim Anwender via Konsole (Eingabeaufforderung) zu erfragen.
- `Zeige()`
  Das Objekt wird beauftragt, `zaehler` und `nenner` auf der Konsole anzuzeigen.

Man verwendet für die in einer Klasse definierten Bestandteile oft die Bezeichnung **Member**, gelegentlich auch die deutsche Übersetzung **Mitglieder**. Unsere `Bruch`-Klasse enthält folgende Member:

- Felder
  `zaehler`, `nenner`
- Eigenschaften
  `Zaehler`, `Nenner`
  Hier handelt es sich letztlich um Paare von Methoden.
- Methoden
  `Kuerze()`, `Addiere()`, `Frage()` und `Zeige()`

Durch die in C# signifikante (!) Groß-/Kleinschreibung der Namen kann man leichter unterscheiden zwischen:

- privaten Merkmalen (per Konvention mit kleinem Anfangsbuchstaben)
- Eigenschaften und Methoden (per Konvention mit großem Anfangsbuchstaben)

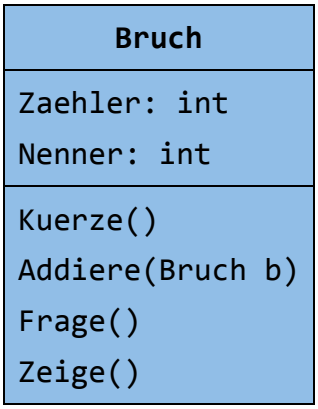
Später folgen detaillierte Empfehlungen zur Verwendung von Bezeichnern in C#.

Von kommunizierenden Objekten und Klassen mit Handlungskompetenzen zu sprechen, mag als übertriebener Anthropomorphismus (als Vermenschlichung) erscheinen. Bei der Ausführung von Methoden sind Objekte und Klassen selbstverständlich streng determiniert, während Menschen bei Kommunikation und Handlungsplanung ihren freien Willen einbringen. Fußball spielende Roboter (als besonders anschauliche Objekte aufgefasst) zeigen allerdings mittlerweile schon recht weitsichtige und auch überraschende Spielzüge. Was sie noch zu lernen haben, sind vielleicht Strafraumschwalben, absichtliches Handspiel etc. Nach diesen Randbemerkungen kehren wir zum Programmierkurs zurück, um möglichst bald freundliche und kluge Objekte erstellen zu können.

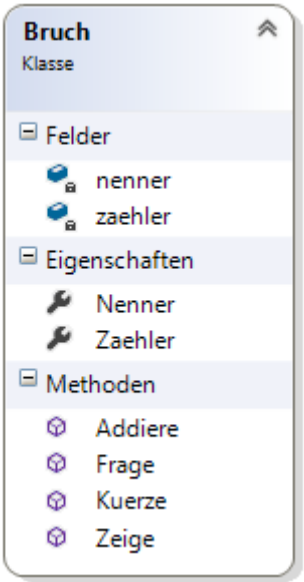
Um die durch objektorientierte Analyse gewonnene Modellierung eines Anwendungsbereichs standardisiert und übersichtlich zu beschreiben, wurde die **Unified Modeling Language** (UML) entwickelt. Hier wird eine Klasse durch ein Rechteck mit drei Bereichen dargestellt:

- Oben steht der **Name** der Klasse.
- In der Mitte stehen die **Merkmale**.
  Hinter dem Namen eines Merkmals gibt man seinen Datentyp (siehe unten) an. Bei den Eigenschaften von C# handelt es sich nach obigen Erläuterungen eigentlich um Zugriffs*methoden*, die aber syntaktisch wie (öffentlich verfügbare) Merkmale angesprochen werden. Es hängt vom Adressaten eines Klassendiagramms (z.B. Entwickler-Teamkollege oder Anwender der Klasse) ab, ob man die Felder, die Eigenschaften oder beides angibt.
- Unten stehen die **Handlungskompetenzen** (Methoden).
  In Anlehnung an eine in vielen Programmiersprachen (z.B. in C#) übliche und noch ausführlich zu behandelnde Syntax zur Methodendefinition gibt man für die Argumente eines Methodenaufrufs (mit Spezifikationen der gewünschten Ausführungsart bzw. mit Details der gesendeten Nachricht) den Datentyp an.

Bei der `Bruch`-Klasse erhält man folgende Darstellung, wenn die Eigenschaften aus der Anwenderperspektive betrachtet werden (als Merkmale und nicht als Methodenpaare):



Die im Kurs bevorzugte Entwicklungsumgebung Visual Studio 2015 Community (siehe Abschnitt 2.2) erstellt auf Wunsch zur Klasse `Bruch` das folgende Diagramm:



Sind bei einer Anwendung *mehrere* Klassen beteiligt, dann sind auch die *Beziehungen* zwischen den Klassen wesentliche Bestandteile des UML-Modells.

Nach der sorgfältigen Modellierung per UML muss übrigens die Kodierung eines Softwaresystems nicht am Punkt Null beginnen, weil professionelle UML-Werkzeuge Teile des Quellcodes automatisch aus dem Modell erzeugen können. Auch die im Kurs bevorzugte Entwicklungsumgebung *Visual Studio 2015 Community* enthält ein solches Werkzeug.

Das relativ einfache Einstiegsbeispiel sollte Sie nicht dazu verleiten, den Begriff *Objekt* auf *Gegenstände* zu beschränken. Auch *Ereignisse* wie z.B. die Fehler eines Schülers in einem entsprechend ausgebauten Bruchrechnungsprogramm kommen als *Objekte* in Frage.

Weiterführende Informationen zur objektorientierten Analyse und Modellierung bieten z.B. Balzert (2011) und Boch et al. (2007).

1.1.2 Objektorientierte Programmierung

In unserem Beispielprojekt soll nun die Klasse `Bruch` in der Programmiersprache C# kodiert werden, wobei die Felder (Instanzvariablen) zu deklarieren, sowie Eigenschaften und Methoden zu implementieren sind. Es resultiert der so genannte **Quellcode**, der am besten in einer Textdatei namens **Bruch.cs** untergebracht wird.

Zwar sind Ihnen die meisten Details der folgenden Klassendefinition selbstverständlich jetzt noch fremd, doch sind die Variablendeklarationen sowie die Eigenschafts- und Methodenimplementationen als zentrale Bestandteile leicht zu erkennen:

```
using System;

public class Bruch {
   int zaehler,  // wird automatisch mit 0 initialisiert
       nenner = 1;

   public int Zaehler {
      get {
         return zaehler;
      }
      set {
         zaehler = value;
      }
   }

   public int Nenner {
      get {
         return nenner;
      }
      set {
         if (value != 0)
            nenner = value;
      }
   }

   public void Zeige() {
      Console.WriteLine("   {0}\n  -----\n   {1}\n", zaehler, nenner);
   }
```

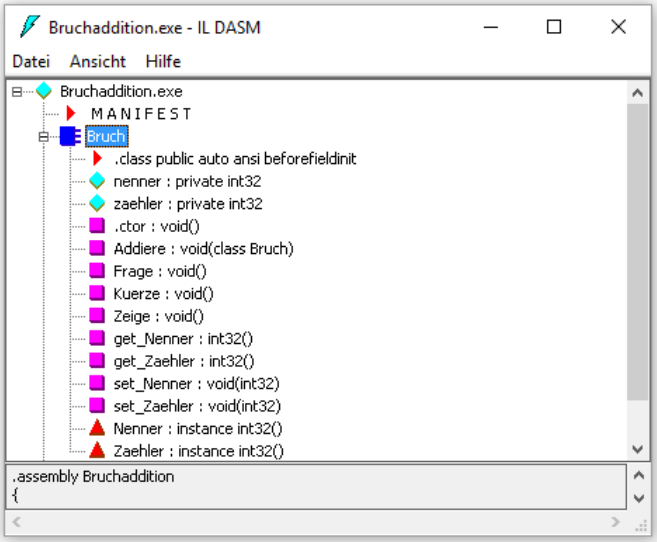
```
    public void Kuerze() {
       // größten gemeinsamen Teiler mit dem Euklidischen Algorithmus bestimmen
       if (zaehler != 0) {
          int ggt = 0;
          int az = Math.Abs(zaehler);
          int an = Math.Abs(nenner);
          do {
             if (az == an)
                ggt = az;
             else
                if (az > an)
                   az = az - an;
                else
                   an = an - az;
          } while (ggt == 0);

          zaehler /= ggt;
          nenner  /= ggt;
       }
    }

    public void Addiere(Bruch b) {
       zaehler = zaehler*b.nenner + b.zaehler*nenner;
       nenner = nenner*b.nenner;
       Kuerze();
    }

    public void Frage() {
       Console.Write("Zähler: ");
       zaehler = Convert.ToInt32(Console.ReadLine());
       Console.Write("Nenner: ");
       Nenner = Convert.ToInt32(Console.ReadLine());
    }

 }
```

Weil für die beiden Felder (`zaehler`, `nenner`) die voreingestellte **private**-Deklaration unverändert gilt, ist im Beispielprogramm das Prinzip der Datenkapselung realisiert. Eigenschaften und Methoden werden durch die Verwendung des Modifikators **public** für die Verwendung in klassenfremden Methoden frei gegeben. Außerdem wird für die Klasse selbst mit dem Modifikator **public** die Verwendung in beliebigen .NET - Programmen erlaubt.

Das zusammen mit der im Kurs bevorzugten Entwicklungsumgebung *Visual Studio 2015 Community* (siehe Abschnitt 2.2) installierte Hilfsprogramm **ILDasm** liefert die folgende Beschreibung der Klasse `Bruch`:[1]

[1] Nach einer Standardinstallation von Visual Studio 2015 Community (vgl. Abschnitt 2.2) ist das Programm **ildasm.exe** im folgenden Ordner zu finden:

C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools



Felder werden durch eine türkis gefärbte Raute (), Methoden durch ein Magenta-farbiges Quadrat () und Eigenschaften durch ein rotes, mit der Spitze nach oben zeigendes Dreieck () dargestellt.

Hier bestätigt sich übrigens die Andeutung von Abschnitt 1.1.1, dass hinter den C# - Eigenschaften letztlich Methoden für Lese- und Schreibzugriffe stehen (siehe z.B. `get_Nenner()`, `set_Nenner()`).

Wie Sie bei späteren Beispielen erfahren werden, dienen in einem objektorientierten Programm beileibe nicht alle Klassen zur Modellierung des Aufgabenbereichs. Es sind auch Objekte aus der Welt des Computers zu repräsentieren (z.B. Fenster der Benutzeroberfläche, Netzwerkverbindungen, Störungen des normalen Programmablaufs).

1.1.3 Algorithmen

Am Anfang von Abschnitt 1.1 wurden mit der *Modellierung des Anwendungsbereichs* und der *Realisierung von Algorithmen* zwei wichtige Aufgaben der Softwareentwicklung genannt, von denen die letztgenannte bisher kaum zur Sprache kam. Auch im weiteren Verlauf des Kurses wird die explizite Diskussion von Algorithmen (z.B. hinsichtlich Voraussetzungen, Korrektheit, Terminierung und Aufwand) keinen großen Raum einnehmen. Wir werden uns intensiv mit der Programmiersprache C# sowie der .NET - Klassenbibliothek beschäftigen und dabei mit möglichst einfachen Beispielprogrammen (Algorithmen) arbeiten.

Unser Einführungsbeispiel verwendet in der Methode `Kuerze()` den bekannten und nicht gänzlich trivialen **euklidischen Algorithmus**, um den größten gemeinsamen Teiler (GGT) von Zähler und Nenner eines Bruchs zu bestimmen, durch den zum optimalen Kürzen beide Zahlen zu dividieren sind. Beim euklidischen Algorithmus wird die leicht zu beweisende Aussage genutzt, dass für zwei natürliche Zahlen (1, 2, 3, …) u und v ($u > v$) der GGT gleich dem GGT von v und (u - v) ist:

Ist t ein Teiler von u und v, dann gibt es natürliche Zahlen t_u und t_v mit $t_u > t_v$ und

$$u = t_u \cdot t \quad \text{sowie} \quad v = t_v \cdot t$$

Folglich ist t auch ein Teiler von (u - v), denn:

$$u - v = (t_u \cdot - t_v \cdot) \cdot t$$

Ist andererseits t ein Teiler von u und (u – v), dann gibt es natürliche Zahlen t_u und t_d mit $t_u > t_d$ und

$$u = t_u \cdot t \quad \text{sowie} \quad (u - v) = t_d \cdot t$$

Folglich ist *t* auch ein Teiler von *v*:

$$u - (u - v) = v = (t_u \cdot - t_d) \cdot t$$

Weil die Paare (*u*, *v*) und (*v*, *u* - *v*) dieselben Mengen gemeinsamer Teiler besitzen, sind auch die größten gemeinsamen Teiler identisch. Weil die Zahl 1 als trivialer Teiler zugelassen ist, existiert übrigens zu zwei natürlichen Zahlen immer ein größter gemeinsamer Teiler, der eventuell gleich 1 ist.

Dieses Ergebnis wird in der Methode `Kuerze()` folgendermaßen ausgenutzt:

Es wird geprüft, ob Zähler und Nenner identisch sind. Trifft dies zu, ist der GGT gefunden (identisch mit Zähler und Nenner). Anderenfalls wird die größere der beiden Zahlen durch deren Differenz ersetzt, und mit diesem verkleinerten Problem startet das Verfahren neu.

Man erhält auf jeden Fall in endlich vielen Schritten zwei identische Zahlen und damit den GGT.

Der beschriebene Algorithmus eignet sich dank seiner Einfachheit gut für das Einführungsbeispiel, ist aber in Bezug auf den erforderlichen Berechnungsaufwand nicht überzeugend. In einer Übungsaufgabe zu Abschnitt 3.7 werden Sie eine erheblich effizientere Variante implementieren.

1.1.4 Startklasse und Main() - Methode

Bislang wurde im Anwendungsbeispiel aufgrund einer objektorientierten Analyse des Aufgabenbereichs die Klasse `Bruch` entworfen und in C# realisiert. Wir verwenden nun die Klasse `Bruch` in einer Konsolenanwendung zur Addition von zwei Brüchen. Dabei bringen wir einen Akteur ins Spiel, der in einem einfachen sequentiellen Handlungsplan `Bruch`-Objekte erzeugt und ihnen Nachrichten zustellt, die (zusammen mit dem Verhalten des Anwenders) den Programmablauf voranbringen.

In diesem Zusammenhang ist von Bedeutung, dass es in *jedem* C# - Programm eine besondere Klasse geben muss, die eine Methode mit dem Namen **Main()** in ihren klassenbezogenen Handlungsrepertoire besitzt. Beim Start eines Programms wird seine Startklasse ausfindig gemacht und aufgefordert, ihre **Main()** - Methode auszuführen.

Es bietet sich an, die oben angedachte Handlungssequenz des Bruchadditionsprogramms in der obligatorischen Startmethode unterzubringen.

Obwohl prinzipiell möglich, erscheint es nicht sinnvoll, die auf Wiederverwendbarkeit hin konzipierte Klasse `Bruch` mit der Startmethode für eine sehr spezielle Anwendung zu belasten. Daher definieren wir eine zusätzliche Klasse namens `Bruchaddition`, die nicht als Bauplan für Objekte dienen soll und auch kaum Recycling-Chancen besitzt. Ihr Handlungsrepertoire kann sich auf die *Klassenmethode* **Main()** zur Ablaufsteuerung im Bruchadditionsprogramm beschränken. Indem wir eine *neue* Klasse definieren und dort `Bruch`-Objekte verwenden, wird u.a. gleich demonstriert, wie leicht das Hauptergebnis unserer bisherigen Arbeit (die Klasse `Bruch`) für verschiedene Projekte genutzt werden kann.

In der `Bruchaddition`-Methode **Main()** werden zwei Objekte (Instanzen) aus der Klasse `Bruch` per **new**-Operator erzeugt und mit der Ausführung verschiedener Methoden beauftragt:[1]

[1] Mit dem Erzeugen von Objekten einer Klasse per **new**-Operator werden wie uns später noch ausführlich beschäftigen. Dabei ist eine spezielle Instanzmethode beteiligt, ein so genannter **Konstruktor**.

Quellcode in **Bruchaddition.cs**	Ein- und Ausgabe

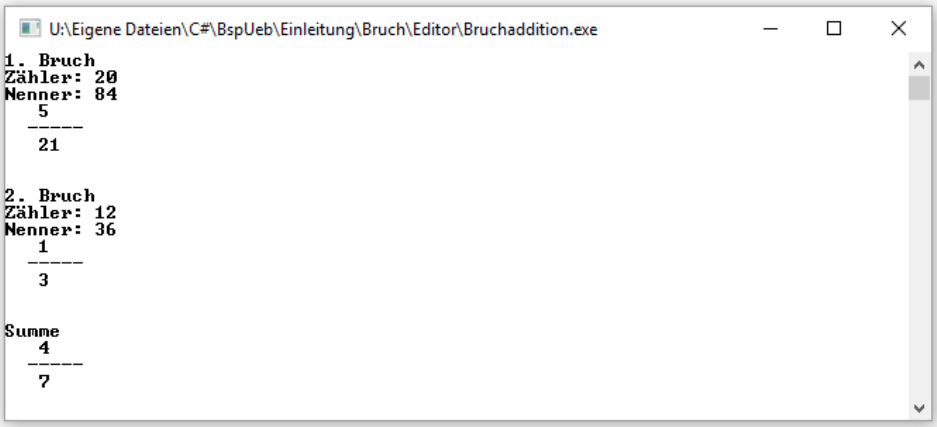
```
using System;

class Bruchaddition {
   static void Main() {
      Bruch b1 = new Bruch(), b2 = new Bruch();

      Console.WriteLine("1. Bruch");
      b1.Frage();
      b1.Kuerze();
      b1.Zeige();

      Console.WriteLine("\n2. Bruch");
      b2.Frage();
      b2.Kuerze();
      b2.Zeige();

      Console.WriteLine("\nSumme");
      b1.Addiere(b2);
      b1.Zeige();
      Console.ReadLine();
   }
}
```

```
1. Bruch
Zähler: 20
Nenner: 84
    5
  -----
    21


2. Bruch
Zähler: 12
Nenner: 36
    1
  -----
    3


Summe
    4
  -----
    7
```

Zum Ausprobieren startet man aus dem Ordner

...\BspUeb\Einleitung\Bruch\Editor

(zu finden an der im Vorwort vereinbarten Stelle) das Programm **Bruchaddition.exe**, z.B.:



In der **Main()** - Methode der Klasse `Bruchaddition` kommen nicht nur `Bruch`-Objekte zum Einsatz. Wir nutzen dort auch die Kompetenzen der Klasse **Console** aus der Standardbibliothek und rufen ihre Klassenmethoden **WriteLine()** und **ReadLine()** auf.

Wir haben zur Lösung der Aufgabe, ein Programm für die Addition von Brüchen zu erstellen, zwei Klassen mit folgender Rollenverteilung definiert:

- Die Klasse `Bruch` enthält den Bauplan für die wesentlichen Akteure im Aufgabenbereich. Dort alle Merkmale und Handlungskompetenzen von Brüchen zu konzentrieren, hat folgende Vorteile:
  - Die Klasse kann in verschiedenen Programmen eingesetzt werden (Wiederverwendbarkeit). Dies fällt vor allem deshalb so leicht, weil die Objekte Handlungskompetenzen (Methoden) besitzen **und** alle erforderlichen Instanzvariablen mitbringen.
  - Beim Umgang mit den `Bruch`-Objekten sind wenige Probleme zu erwarten, weil nur klasseneigene Methoden Zugang zu kritischen Merkmalen haben (Datenkapselung). Sollten doch Fehler auftreten, sind die Ursachen in der Regel schnell identifiziert.

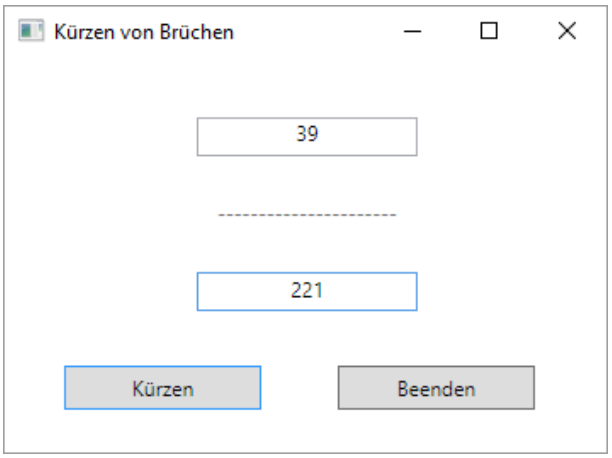
  Wir müssen bei der Definition der Klasse `Bruch` ihre allgemeine Verfügbarkeit explizit mit dem Zugriffsmodifikator **public** genehmigen. Per Voreinstellung ist eine Klasse nur intern (in der eigenen Übersetzungseinheit) verfügbar, was im Kurs noch ausführlich zu behandeln ist.
- Die Klasse `Bruchaddition` dient *nicht* als Bauplan für Objekte, sondern enthält die Klassenmethode **Main()**, die beim Programmstart automatisch aufgerufen wird und dann für einen speziellen Einsatz von `Bruch`-Objekten sorgt. Mit einer Wiederverwendung des `Bruchaddition`-Quellcodes in anderen Projekten ist kaum zu rechnen.

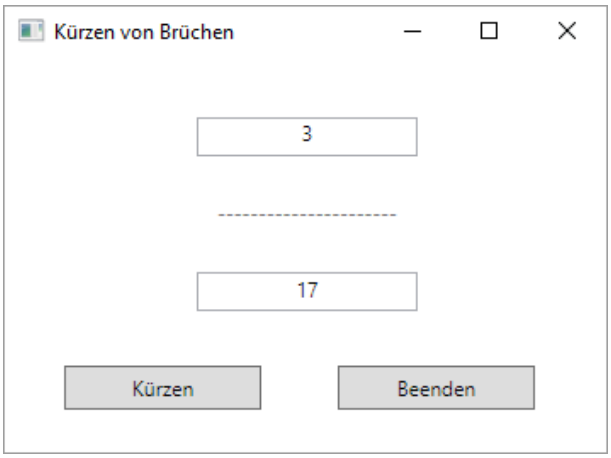
In der Regel bringt man den Quellcode jeder Klasse in einer eigenen Textdatei unter, die den Namen der Klasse trägt, ergänzt um die Namenserweiterung **.cs**. In C# sind aber Quellcodedateien mit mehreren Klassen und einem beliebigen Dateinamen erlaubt.

Während bei den Namen von C# - Klassen die Groß-/Kleinschreibung signifikant ist, spielt sie bekanntlich bei Dateinamen unter Windows keine Rolle. Wenn eine C# - Quellcodedatei der üblichen Praxis folgende genau *eine* Klassendefinition enthält, sollte der Dateiname m.E. *inkl. Groß-/Kleinschreibung* von der Klasse übernommen werden. Die Namen der von Microsoft erhältlichen Dateien mit dem .NET - Quellcode verwenden allerdings nur Kleinbuchstaben, z.B. **string.cs**.[1]

1.1.5 Ausblick auf Anwendungen mit grafischer Benutzerschnittstelle

Das obige Beispielprogramm arbeitet der Einfachheit halber mit einer Konsolen-orientierten Ein- und Ausgabe. Nachdem wir im Kurs in dieser übersichtlichen Umgebung grundlegende Sprachelemente kennengelernt haben, werden wir uns selbstverständlich auch mit der Programmierung von grafischen *Bedienoberflächen* beschäftigen. In folgendem Programm zum Kürzen von Brüchen wird die oben definierte Klasse `Bruch` verwendet, wobei an Stelle ihrer Methoden `Frage()` und `Zeige()` jedoch grafikorientierte Techniken zum Einsatz kommen:





[1] https://referencesource.microsoft.com/download.html

Mit dem Quellcode zur Gestaltung der grafischen Oberfläche könnten Sie im Moment noch nicht allzu viel anfangen. Am Ende des Kurses werden Sie derartige Anwendungen aber mit Leichtigkeit erstellen.

1.1.6 Zusammenfassung zu Abschnitt 1.1

Im Abschnitt 1.1 sollten Sie einen ersten Eindruck von der Softwareentwicklung mit C# gewinnen. Alle dabei erwähnten Konzepte der objektorientierter Programmierung und technischen Details der Realisierung in C# werden bald systematisch behandelt und sollten Ihnen daher im Moment noch keine Kopfschmerzen bereiten. Trotzdem kann es nicht schaden, an dieser Stelle einige Kernaussagen von Abschnitt 1.1 zu wiederholen:

- Vor der Programmentwicklung findet die **objektorientierte Analyse** der Aufgabenstellung statt. Dabei werden per Abstraktion die beteiligten Klassen und ihre Beziehungen identifiziert.
- Ein Programm besteht aus **Klassen**. Unsere Beispielprogramme zum Erlernen elementarer Sprachelemente werden oft mit einer einzigen Klasse auskommen. Praxisgerechte Programme bestehen in der Regel aus zahlreichen Klassen.
- Eine Klasse ist charakterisiert durch **Merkmale und Methoden**.
- Eine Klasse in der Regel als **Bauplan für Objekte**, kann aber auch selbst aktiv werden (Methoden ausführen und aufrufen).
- Ein Merkmal bzw. eine Methode wird entweder den Objekten einer Klasse oder der Klasse selbst zugeordnet.
- In den Methodendefinitionen werden Algorithmen realisiert. Dabei kommen selbst erstellte Klassen zum Einsatz, aber auch vordefinierte Klassen aus diversen Bibliotheken.
- Im Programmablauf kommunizieren die Akteure (Objekte und Klassen) durch den Aufruf von Methoden miteinander, wobei aber in der Regel noch „externe Kommunikationspartner“ (z.B. Benutzer, andere Programme) beteiligt sind.
- Beim Programmstart wird die Startklasse vom Laufzeitsystem aufgefordert, die Methode **Main()** auszuführen. Ein Hauptzweck dieser Methode besteht oft darin, Objekte zu erzeugen und somit „Leben auf die objektorientierte Bühne zu bringen“.

1.2 Das .NET - Framework

Eben haben Sie C# als eine Programmiersprache kennen gelernt, die Ausdrucksmittel zur Modellierung von Anwendungsbereichen und zur Formulierung von Algorithmen bereitstellt. Unter einem *Programm* wurde dabei der vom Entwickler zu formulierende *Quellcode* verstanden. Während *Sie* derartige Texte bald ohne große Mühe lesen und begreifen werden, kann die CPU (*Central Processing Unit*) eines Rechners nur einen maschinenspezifischen Satz von Befehlen verstehen, die als Folge von Nullen und Einsen kodiert werden müssen (*Maschinencode*). Die ebenfalls CPU-spezifische Assembler-Sprache stellt eine für Menschen lesbare Form des Maschinencodes dar. Mit dem Assembler- bzw. Maschinenbefehl

```
mov eax, 4
```

einer CPU aus der x86-Familie wird z.B. der Wert 4 in das EAX-Register (ein Speicherort im Prozessor) geschrieben. Die CPU holt sich einen Maschinenbefehl nach dem anderen aus dem Hauptspeicher und führt ihn aus, wobei heutzutage (2016) die CPU eines handelsüblichen Arbeitsplatzrechners bis zu 350 Milliarden Befehle pro Sekunde (*Instructions Per Second*, IPS) schafft.[1]

[1] Siehe: https://de.wikipedia.org/wiki/Instruktionen_pro_Sekunde

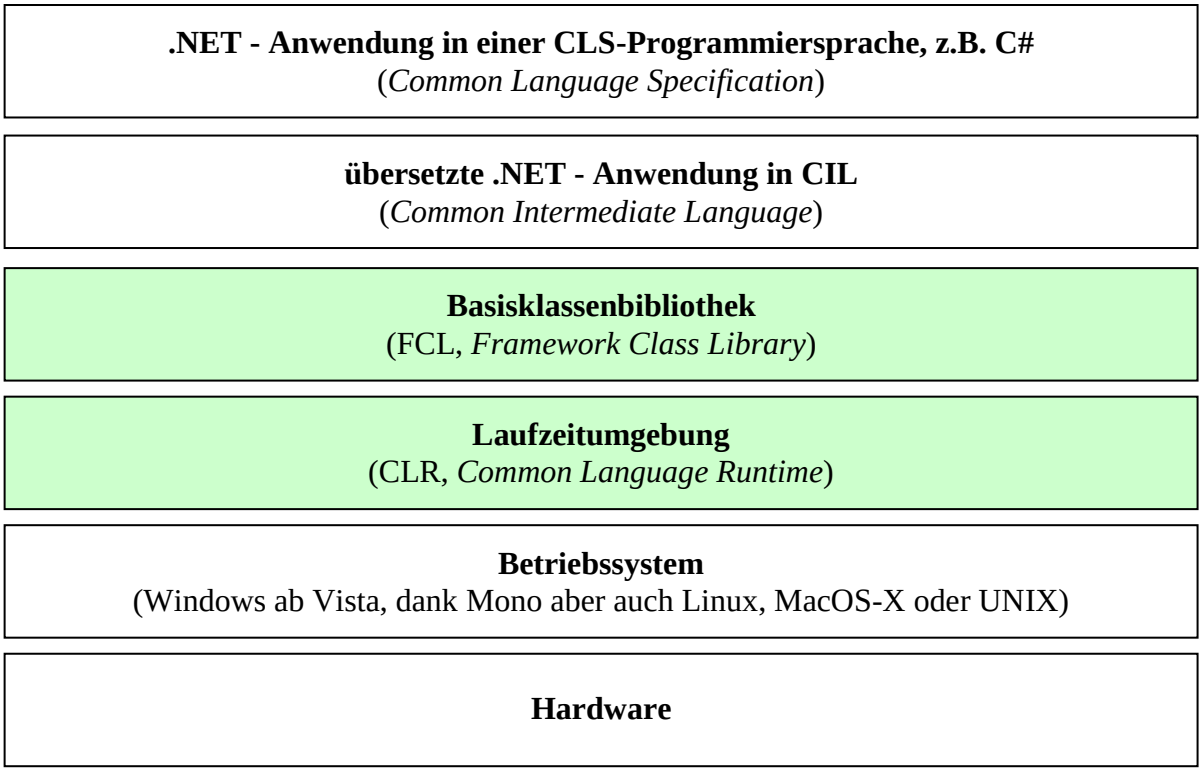
Ein Quellcode-Programm muss also erst in Maschinencode übersetzt werden, damit es von einem Rechner ausgeführt werden kann. Wie dies bei C# und anderen .NET - Programmiersprachen geschieht, sehen wir uns nun näher an. Wir behandeln nicht die komplette .NET - Softwarearchitektur, sondern beschränken uns auf die für Programmierer wichtigen Hintergrundinformationen.

1.2.1 Überblick

Beim .NET - Framework handelt es sich um eine von der Firma Microsoft entwickelte Plattform (Sammlung von Technologien) zur Modernisierung und Vereinfachung der Anwendungsentwicklung unter Window, die als Grundlage zur Entwicklung von Anwendungs-Software deutlich über das traditionelle Windows-API hinausgeht (Mössenböck, 2016, S. 4).

Grundsätzlich und nachweislich lässt sich das .NET - Framework auch auf anderen Betriebssystemen aufsetzen. Dank Mono-Projekt sind neben Windows auch die beiden anderen großen Desktop-Betriebssysteme „versorgt“.

Als Orientierung für die nächsten Abschnitte kann die folgende, stark vereinfachte Darstellung des .NET - Frameworks dienen:



Seit *Vista* ist das .NET - Framework fester Bestandteil des Betriebssystems Windows. Um die mit einem Rechner ausgelieferte .NET - Version durch eine neuere Version zu ersetzen, kann ein bei Microsoft kostenlos verfügbares .NET Framework - Paket nachinstalliert werden (zur Beschaffung und Installation siehe Abschnitt 1.2.2). Dieses Paket darf auch von Entwicklern zusammen mit eigenen .NET - Anwendungen ausgeliefert werden. Über Installationspakete aus dem Open Source - Projekt *Mono*[1] ist ein .NET - kompatibles Framework auch für Linux und MacOS-X verfügbar.

Das .NET - Framework hat seit 2002 zahlreiche Aktualisierungen erlebt und ist aktuell (2016) bei der Version 4.6.2 angekommen. Zwar besitzen das .NET-Framework und die Programmiersprache

[1] http://www.mono-project.com/

C# jeweils eine eigenständige Versionierung, doch sind die beiden Zeitreihen mit den Versionsständen stark korreliert, wie die folgende Tabelle zeigt:[1]

.NET - Framework	CLR-Version	Erscheinungsjahr	C#
1.0	1.0	2002	1.0
1.1	1.1	2003	1.2
2.0	2.0	2005	2.0
3.0	2.0	2006	2.0
3.5	2.0	2007	3.0
4.0	4	2010	4.0
4.5	4	2012	5.0
4.6	4	2015	6.0

1.2.2 Installation

Das zum Ausführen von .NET - Programmen erforderliche Framework (mit der CLR und der Klassenbibliothek FCL) ist in allen aktuell von Microsoft unterstützten Windows-Versionen enthalten:[2]

Windows-Version	Ausgelieferte .NET - Version
Vista	3.5
7	3.5
8	4.5
8.1	4.5.1
10	4.6
10, Update 1511	4.6.1
10, Update 1607	4.6.2

Durch kostenlos bei Microsoft verfügbare Installationspakete kann die mit einem Betriebssystem ausgelieferte .NET - Version aktualisiert werden. Während für Windows Vista bei .NET 4.6 Schluss ist, werden die anderen Windows-Versionen auch noch von der neuesten .NET - Version unterstützt.

Es kann erforderlich werden, eine ältere Framework-Version zu installieren oder via Windows-Systemsteuerung (**Programme und Funktionen > Windows-Features aktivieren**) freizuschalten, weil ältere .NET - Programme zur Vermeidung von Versionskonflikten stets von einer CLR mit einem bestimmten Versionsstand ausgeführt werden wollen. Die .NET - Version 3.5 bringt freundlicherweise die älteren Versionen 3.0 und 2.0 gleich mit.

Bei der Installation von Microsofts *Visual Studio 2015* (siehe Abschnitt 2.2) landet nötigenfalls das .NET - Framework in der Version 4.6.1 automatisch auf der Festplatte. Wenn auf den Rechnern Ihrer Kunden das Framework in der benötigten Version fehlt, können Sie es kostenlos von Microsoft beziehen und mit den eigenen Anwendungen ausliefern.[3]

Als .NET - Installationsordner werden (ohne Änderungsmöglichkeit) verwendet:

---

1 http://stackoverflow.com/questions/247621/what-are-the-correct-version-numbers-for-c
https://en.wikipedia.org/wiki/.NET_Framework_version_history

2 https://msdn.microsoft.com/de-de/library/5a4x27ek(v=vs.110).aspx

3 Eine Quelle zum Herunterladen via Internet ist (z.B. per Suchmaschine) leicht zu finden, so dass auf die Angabe von länglichen, versionsabhängigen und oft nicht sehr zeitstabilen Internet-Adressen verzichtet wird.

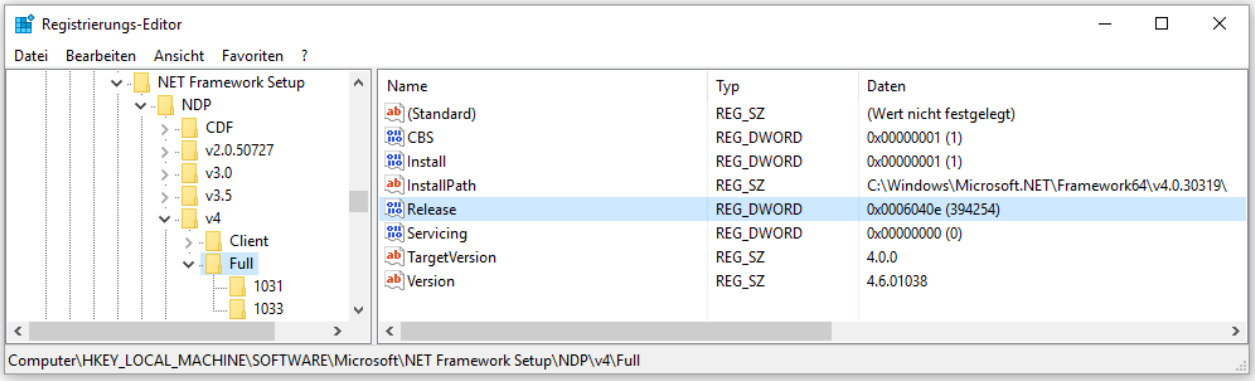
.NET - Version	Installationsordner
2.0	x86: %SystemRoot%\Microsoft.NET\Framework\v2.0.50727 x64: %SystemRoot%\Microsoft.NET\Framework64\v2.0.50727
3.0	x86: %SystemRoot%\Microsoft.NET\Framework\v3.0 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.0
3.5	x86: %SystemRoot%\Microsoft.NET\Framework\v3.5 x64: %SystemRoot%\Microsoft.NET\Framework64\v3.5
ab 4.0	x86: %SystemRoot%\Microsoft.NET\Framework\v4.0.30319 x64: %SystemRoot%\Microsoft.NET\Framework64\v4.0.30319

Damit auf einem PC mit 64-bittiger Windows-Installation auch x86-abhängige Assemblies (siehe Abschnitt 1.2.5.5) laufen, werden dort *zwei* .NET - Laufzeitumgebungen (mit 32 bzw. 64 Bit) installiert.

Zur Beschreibung des Verfahrens, wie die auf einem Rechner installierte .NET - Version festzustellen ist, benötigt Microsoft (im November 2016) einen sehr langen „Gewusst wie“ - Artikel.[1] Ab .NET 4.5 ist dem Registry-Key

HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\NET Framework Setup\NDP\v4\Full

die **Release**-Angabe



zu entnehmen und mit einer Tabelle zu vergleichen:

DWORD-Eintrag	Version
378389	.NET Framework 4.5
378675	.NET Framework 4.5.1 installiert mit Windows 8.1 oder Windows Server 2012 R2
378758	.NET Framework 4.5.1 installiert unter Windows 8, Windows 7 SP1 oder Windows Vista SP2
379893	.NET Framework 4.5.2
Auf Systemen unter Windows 10: 393295 Auf allen anderen Betriebssystemversionen: 393297	.NET Framework 4.6
Auf Systemen mit Windows 10, November-Update: 394254 Auf allen anderen Betriebssystemversionen: 394271	.NET Framework 4.6.1

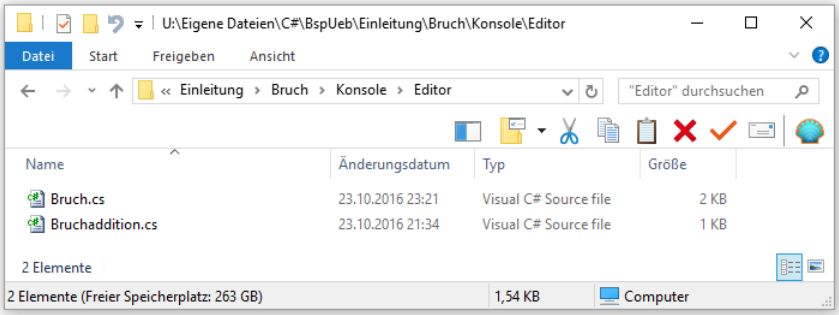
Auf dem Beispielrechner mit Windows 10 Professional (Update 1511) und installiertem Visual Studio 2015 ergibt sich die .NET - Version 4.6.1.

[1] https://msdn.microsoft.com/de-de/library/hh925568(v=vs.110).aspx

1.2.3 C# - Compiler und CIL

Den (z.B. mit einem beliebigen Texteditor verfassten) C# - Quellcode übersetzt ein C# - **Compiler** in die **Common Intermediate Language** (**CIL**), die ursprünglich als **Microsoft Intermediate Language** (**MSIL**) bezeichnet wurde oft kurz als **IL** (**Intermediate Language**) genannt wird. Wenngleich dieser Zwischencode von den heute üblichen Prozessoren nicht direkt ausgeführt werden kann, hat er doch bereits viele Verarbeitungsschritte auf dem Weg vom Quell- zum Maschinencode durchlaufen. Weil kompakter als Maschinencode, eignet sind der CIL-Code gut für die Übertragung über Netzwerke. Die Übersetzung des Zwischencodes in die Maschinensprache einer konkreten CPU geschieht *Just-In-Time* bei der Ausführung des Programms durch die CLR (siehe Abschnitt 1.2.6).[1]

Befinden sich die beiden Quellcodedateien **Bruch.cs** und **Bruchaddition.cs** im aktuellen Verzeichnis



eines Konsolenfensters, dann kann ihre Übersetzung durch den C# - Compiler **csc.exe** des 64-bittigen .NET 4.x - Frameworks mit dem folgenden Kommando

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc *.cs
```

veranlasst werden:

Weil sich der Ordner

C:\Windows\Microsoft.NET\Framework64\v4.0.30319

mit dem C# - Compiler **csc.exe** per Voreinstellung nicht im Suchpfad für ausführbare Programme befindet, muss er im Aufruf angegeben werden. Der eigentliche Auftrag an den Compiler, sämtliche Dateien im aktuellen Verzeichnis mit der Namenserweiterung **.cs** zu übersetzen, ist sehr übersichtlich:

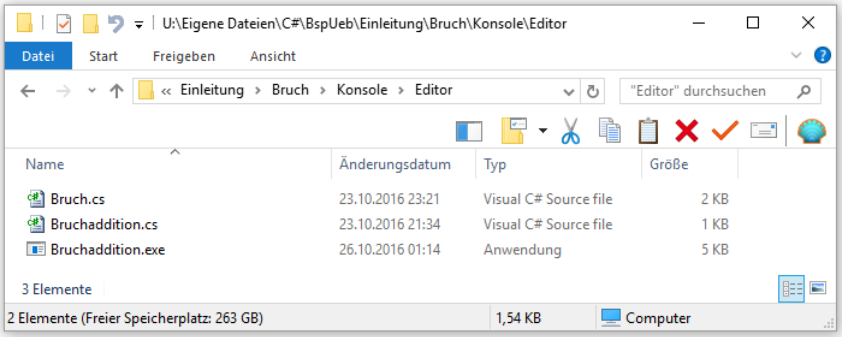
```
csc *.cs
```

Wir werden uns in Abschnitt 2.1.2 noch mit einigen Details des Compiler-Aufrufs beschäftigen.

---

[1] Die Übersetzung vom CIL-Code in Maschinencode sollte auf jeder Plattform gelingen, für die eine CLR verfügbar ist. Z.B. sollten sich die auf einem Rechner mit 64-bittiger Windows-Version erstellten .NET-Programme problemlos in einer 32-Bit - Umgebung nutzen lassen. Grundsätzlich ist diese Portabilität tatsächlich gegeben, jedoch wird in Abschnitt 1.2.5.5 von Ausnahmen zu berichten sein.

Im Übersetzungsergebnis **Bruchaddition.exe**



ist u.a. der CIL-Code der beiden Klassen `Bruch` und `Bruchaddition` enthalten. Besonders kurz sind die implizit zu einer C# - Eigenschaft (vgl. Abschnitt 1.1.1) vom Compiler erstellten **get**- und **set**-Methoden, z.B.:[1]

```
public int Nenner {
  get {
    return nenner;
  }
  set {
    if (value != 0)
      nenner = value;
  }
}
```

C# - Compiler ⇒

```
Bruch::get_Nenner : int32()
Suchen  Weitersuchen
.method public hidebysig specialname instance int32
        get_Nenner() cil managed
{
  // Code size       12 (0xc)
  .maxstack  1
  .locals init (int32 V_0)
  IL_0000:  nop
  IL_0001:  ldarg.0
  IL_0002:  ldfld      int32 Bruch::nenner
  IL_0007:  stloc.0
  IL_0008:  br.s       IL_000a
  IL_000a:  ldloc.0
  IL_000b:  ret
} // end of method Bruch::get_Nenner
```

```
Bruch::set_Nenner : void(int32)
Suchen  Weitersuchen
.method public hidebysig specialname instance void
        set_Nenner(int32 'value') cil managed
{
  // Code size       17 (0x11)
  .maxstack  2
  .locals init (bool V_0)
  IL_0000:  nop
  IL_0001:  ldarg.1
  IL_0002:  ldc.i4.0
  IL_0003:  ceq
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  brtrue.s   IL_0010
  IL_0009:  ldarg.0
  IL_000a:  ldarg.1
  IL_000b:  stfld      int32 Bruch::nenner
  IL_0010:  ret
} // end of method Bruch::set_Nenner
```

Offenbar resultiert aus der `Bruch`-Eigenschaft `Nenner` u.a. die Methode `get_Nenner()`, deren IL-Code aus 7 Anweisungen besteht.

Die konzeptionelle Verwandtschaft der CIL mit dem Bytecode der Java-Plattform ist unverkennbar.

Weil alle .NET - Quellcodedateien unabhängig von der verwendeten Programmiersprache in denselben Zwischencode übersetzt werden, ist die Bezeichnung *Common Intermediate Language* offenbar treffend gewählt. Bei Beachtung gewisser Regeln (siehe nächsten Abschnitt) können verschiedene Programmierer bei der Erstellung von Klassen für ein gemeinsames Projekt jeweils die individuell bevorzugte Programmiersprache verwenden.

---

[1] Diese Ausgabe liefert das in Abschnitt 1.1.2 angesprochene Werkzeug **ILDasm** nach einem Doppelklick auf die interessierende Methode (siehe Seite 7).

1.2.4 Common Language Specification

Mittlerweile sind für viele Programmiersprachen CIL-Compiler verfügbar, einige werden sogar mit dem .NET - Framework ausgeliefert (z.B. **csc.exe** für C#, **vbc.exe** für Visual Basic .NET). Allerdings unterstützen die .NET-Compiler in der Regel nicht den gesamten CIL-Sprachumfang, so dass sich mit den Compilern zu verschiedenen .NET-Sprachen durchaus Klassen produzieren lassen, die *nicht* zusammenarbeiten können (siehe Richter, 2006, S.50). Beispielweise erstellt der C# - Compiler bedenkenlos eine öffentliche Klasse mit zwei öffentlichen Methoden, deren Namen sich nur durch die Groß-/Kleinschreibung unterscheiden (z.B. `TuWas()` und `tuWas()`). Mit einer solchen Klasse können aber die Produktionen von Visual Basic .NET *nicht* kooperieren.

Microsoft hat unter dem Namen **Common Language Specification** (CLS) einen Sprachumfang definiert, den *jede* .NET-Programmiersprache erfüllen muss.[1] Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt. Man kann einen .NET - Compiler auffordern, die CLS-Kompatibilität zu überwachen, so dass z.B. der C# - Compiler im eben beschriebenen Beispiel warnt:

```
ClsIncomplient.cs(12,13): warning CS3005: Der Bezeichner
        "ClsIncomplient.TuWas()", der sich nur hinsichtlich der Groß- und
        Kleinschreibung unterscheidet, ist nicht CLS-kompatibel.
```

1.2.5 Assemblies und Metadaten

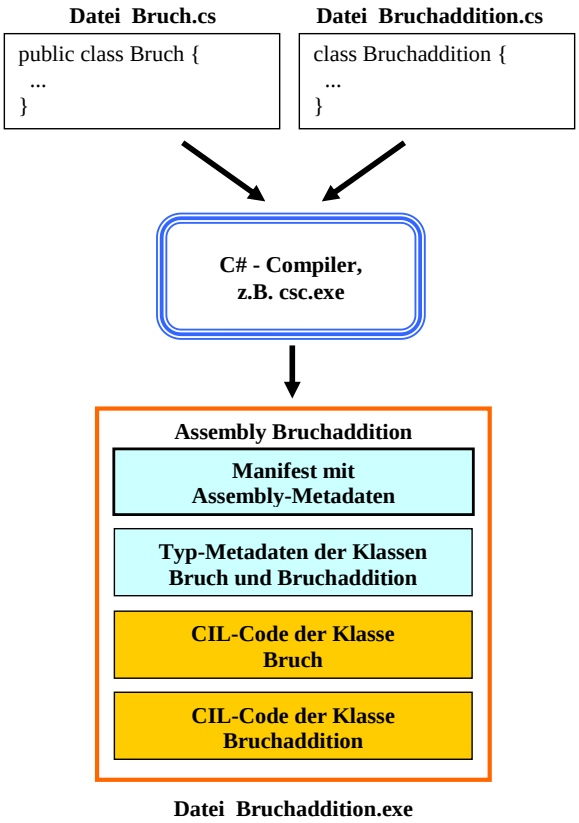
Die von einem .NET - Compiler erzeugten Binärdateien (mit CIL-Code) werden als **Assemblies** bezeichnet und haben die Namenserweiterung:

- **.exe** (bei .NET - Anwendungen) oder
- **.dll** (bei .NET - Bibliotheken)

Man übergibt dem Compiler im Allgemeinen *mehrere* Quellcodedateien mit jeweils einer Klassendefinition (siehe Beispiel in Abschnitt 1.2.3) und erhält als Ergebnis *ein* Assembly. In der Konsolen-Variante unseres Beispiels lassen wir vom Compiler ein Exe-Assembly mit dem Zwischencode der Klassen `Bruch` und `Bruchaddition` erzeugen.

Die folgende Abbildung (nach Mössenböck 2016, S. 7) fasst wesentliche Informationen über Quellcode, C# - Compiler, CIL-Code und Assemblies anhand des Bruchadditionsbeispiels zusammen und zeigt mit den anschließend zu beschreibenden Metadaten weitere wichtige Bestandteile eines .NET - Assemblies:

[1] Siehe z.B. http://msdn.microsoft.com/de-de/library/12a7a7h3.aspx



Von der Option, Ressourcen (z.B. Medien) in ein Assembly aufzunehmen, wird im Beispiel kein Gebrauch gemacht.

1.2.5.1 Typ-Metadaten

Ein Assembly enthält **Typ-Metadaten**, die alle enthaltenen Typen (Klassen und sonstige Datentypen) beschreiben und vom Laufzeitsystem (*Common Language Runtime*, siehe unten) für die Verwaltung der Typen genutzt werden. Zu jeder Klasse sind z.B. Informationen über ihre Methoden und Merkmale vorhanden.

Im `Bruchadditions`-Assembly sind z.B. über die `get_Zaehler` - Methode zur `Zaehler`-Eigenschaft der Klasse `Bruch` folgende Typ-Metadaten verfügbar: [1]

[1] Die Typ-Metadaten eines Assemblies erhält man in **ILDasm** über die Tastenkombination **Strg+M**.

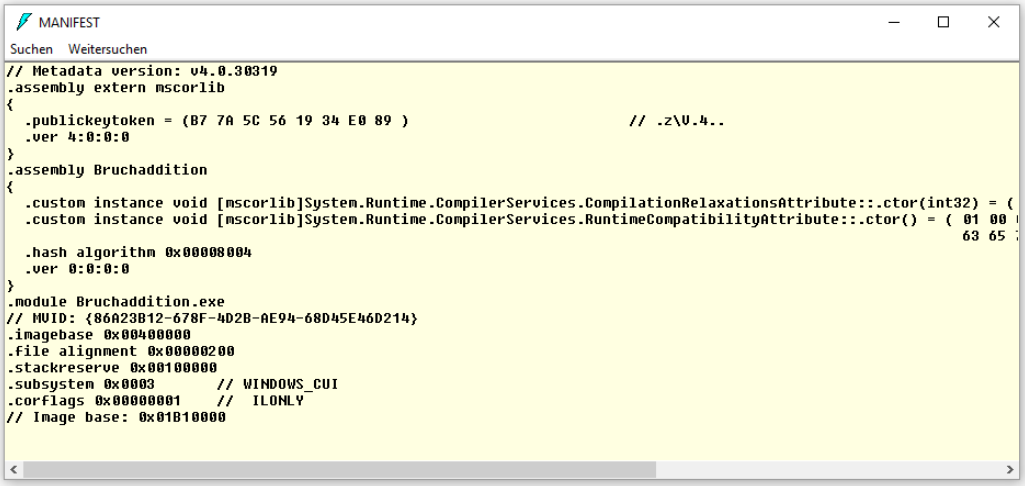
Neben **Definitionstabellen** mit Angaben zu den *eigenen* Klassen enthalten die Metadaten auch **Referenztabellen** mit Informationen zu den *fremden* Klassen, die im Assembly benutzt (referenziert) werden.

1.2.5.2 Manifest mit Assembly-Metadaten

Außerdem gehört zu einem Assembly das so genannte **Manifest** mit den **Assembly-Metadaten**. Dazu gehören:

- Name und Version des Assemblies
  Durch eine systematische Versionsverwaltung sollen im .NET - Framework Probleme mit Versionsunverträglichkeiten vermieden werden, die Windows-Anwender und -Entwickler unter der Bezeichnung *DLL-Hölle* kennen.
- Sicherheitsmerkmale bei signierten Assemblies
  Dazu gehört insbesondere der öffentliche Schlüssel des Herausgebers. Das Signieren ist erforderlich bei Assemblies, die im GAC (***G**lobal **A**ssembly **C**ache*) abgelegt werden sollen (siehe Abschnitt 1.2.5.4).
- Informationen über die Abhängigkeit von anderen Assemblies (z.B. aus der FCL)
  Auch hier sorgen exakte Versionsangaben für die Vermeidung von Versionsunverträglichkeiten.

Das Assembly **Bruchaddition.exe** hat noch keine große Versionshistorie, ist abhängig vom Assembly **mscorlib.dll** (in der Version 4:0:0:0), und hat keine Sicherheitsmerkmale:[1]



Besteht ein Assembly aus *mehreren* Dateien (siehe unten), dann ist das Manifest nur in *einer* Datei vorhanden und enthält die Namen der weiteren zum Assembly gehörenden Dateien. In diesem Fall kann das Manifest auch in einer eigenen Datei untergebracht werden.

1.2.5.3 Multidatei-Assemblies

Neben dem bisher beschriebenen *Einzeldatei*-Assembly, das in *einer* Binärdatei den Zwischencode und die Metadaten inklusive Manifest enthält, kennt das .NET - Framework auch das **Multidatei-Assembly**, das Einsteiger gefahrlos ignorieren dürfen. Es besteht aus mehreren **Moduldateien** mit Zwischencode und zugehörigen Typ-Metadaten. Das Manifest des gesamten Assemblies steckt entweder in einer Moduldatei oder in einer separaten Datei. Diese Architektur hat z.B. dann Vortei-

[1] Diese Ausgabe liefert das in Abschnitt 1.1.2 angesprochene Werkzeug **ILDasm** nach einem Doppelklick auf den Knoten **MANIFEST** (siehe Seite 7).

le, wenn ein Klient über eine langsame Netzverbindung auf ein Assembly zugreift, weil sich der Transport auf die tatsächlich benötigten Module beschränken kann. Ohne die Aufteilung in Module müsste das gesamte Assembly transportiert werden. Für Moduldateien ohne eigenes Manifest wird meist die Namenserweiterung **.netmodule** verwendet.

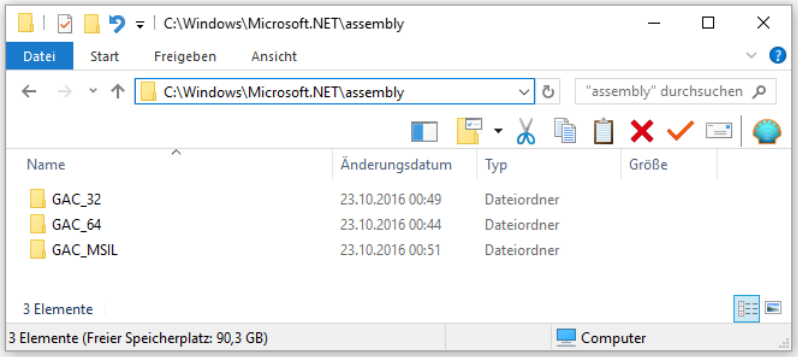
1.2.5.4 Private und globale Assemblies

Ein Assembly benötigt in der Regel weitere Assemblies, deren Typen (z.B. Klassen) verwendet werden. Dabei kommen private und systemweit verfügbare Assemblies in Frage. Private Assemblies werden meist im Ordner der Anwendung untergebracht, doch können mit Hilfe der (später zu behandelnden) Anwendungskonfigurationsdatei auch andere Ordner verwendet werden.

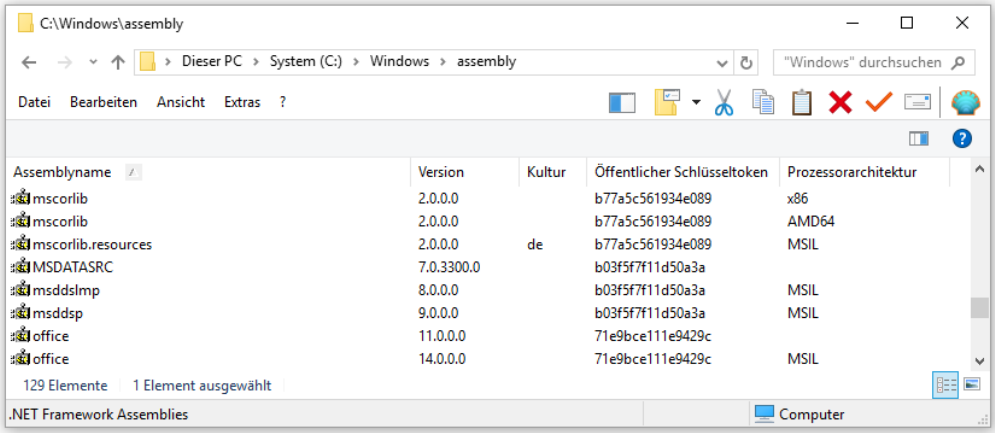
Für systemweit verfügbare Assemblies haben die .NET - Architekten den **Global Assembly Cache** (GAC) vorgesehen. Hier ist vor allen die Framework Class Library untergebracht, doch können auch andere Assemblies aufgenommen werden, wobei Microsoft eine zurückhaltende GAC-Verwendung empfiehlt.[1]

Die Assemblies im GAC werden von der CLR bei der Ausführung eines Programms verwendet, während der Compiler die von ihm benötigten FCL-Metadaten aus Kopien bezieht, die anderer Stelle abgelegt werden (Richter, 2012, Kap. 3).[2]

Seit der Framework-Version 4 befindet sich der GAC im Ordner **C:\Windows\Microsoft.NET\assembly**:



Für ältere Framework-Versionen ist der GAC im Ordner **C:\Windows\assembly** untergebracht, der vom Windows Explorer aufgrund einer Shell Extension auf spezielle Weise angezeigt wird:



---

1 https://msdn.microsoft.com/en-us/library/yf1d93sz(v=vs.110).aspx

2 Wo sich die vom Compiler ausgewerteten Bibliotheks-Assemblies befinden, ist im Abschnitt 2.1.2 zu erfahren.

Eine Besonderheit bei GAC-Assemblies ist die erforderliche Signatur (siehe Spalte **Öffentlicher Schlüsseltoken**), die für eine global eindeutige Identifikation sorgt, so dass man von der Verpflichtung zu *starken Namen* spricht.

Aufgrund der obigen Ausführungen über den Plattform- und CPU-unabhängigen MSIL- bzw. CIL-Code in .NET - Assemblies wundern Sie sich vermutlich über die Spalte **Prozessorarchitektur**, die im Ordnerfenster zum GAC für Framework-Versionen kleiner 4 zu sehen ist und neben MSIL noch andere Einträge enthält. Offenbar sind manche Assemblies von einer Prozessor-Architektur (z.B. x86, AMD64) abhängig, was im folgenden Abschnitt näher beleuchtet werden soll.

1.2.5.5 Plattformspezifische Assemblies

Manche Assemblies müssen mit herkömmlicher Windows-Software (*unmanaged code*) kooperieren, z.B. durch die Nutzung von Funktionen in den traditionellen Maschinencode-DLLs mit dem Windows-API (***A**pplication **P**rogramming **I**nterface*). Daraus resultiert eine Abhängigkeit von der Prozessorarchitektur, für die solche Software erstellt wurde. Ein besonders wichtiges Beispiel ist das Bibliotheks-Assembly **mscorlib.dll**, das elementare und oft benötigte FCL-Klassen implementiert und dabei Dienste des Betriebssystems nutzt. Auf einem Rechner mit 64-bittiger Windowsinstallation werden daher zwei Versionen dieses Assemblies benötigt (für die Plattformen x86 und x64 (alias AMD64), siehe Bildschirmfoto im letzten Abschnitt). Den Rest dieses Abschnitts sollte nur lesen, wer sich (jetzt schon) für weitere Details beim Einsatz von .NET - Software in einer 64-Bit - Umgebung interessiert.

Eine 64-bittige Windows-Version enthält ein 32-Bit - Subsystem namens **WOW64** (*Windows on Windows 64*), so dass 32-Bit-Software in der Regel problemlos in der gewohnten Umgebung ablaufen kann. Dabei herrscht eine strenge Trennung, so dass z.B. ein 64-bittig ausgeführtes Assembly keine DLL oder Komponente mit 32-Bit-Architektur benutzen kann. Stützt sich ein Assembly auf eine solche Software, muss es 32-bittig ausgeführt werden. Die Missachtung dieser Regel wird mit einem Laufzeitfehler bestraft.

Weil sich eine Abhängigkeit von der Systemumgebung nicht immer vermeiden lässt, besitzen Assemblies das Attribut **ProcessorArchitecture** mit den folgenden möglichen Werten (in Klammern die Bezeichnung für den C# - Compiler):

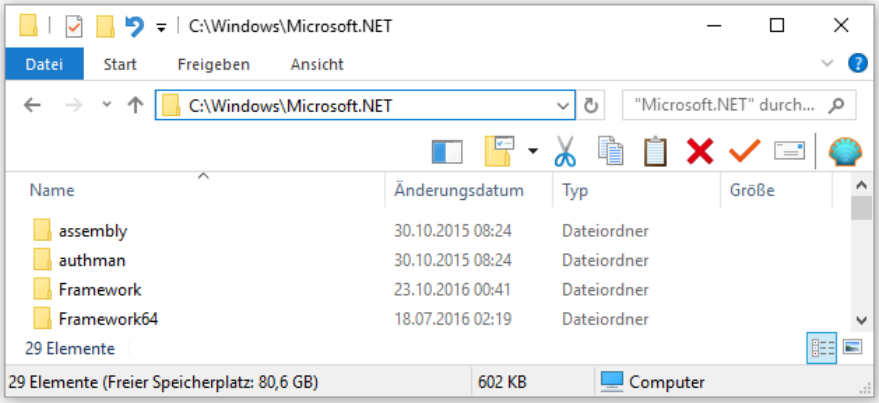
- Amd64 (x64)
- Arm (arm)
- IA64 (Itanium)
- CIL (anycpu)
- X86 (x86)

Beim Übersetzen wird das Attribut über eine Compiler-Option gesetzt, die beim C# - Compiler **platform** heißt und die Voreinstellung **CIL** (**anycpu**) hat, wie der folgende Auszug aus einer (mit `csc /?` angeforderten) Hilfe-Ausgabe des C# - Compilers zeigt:

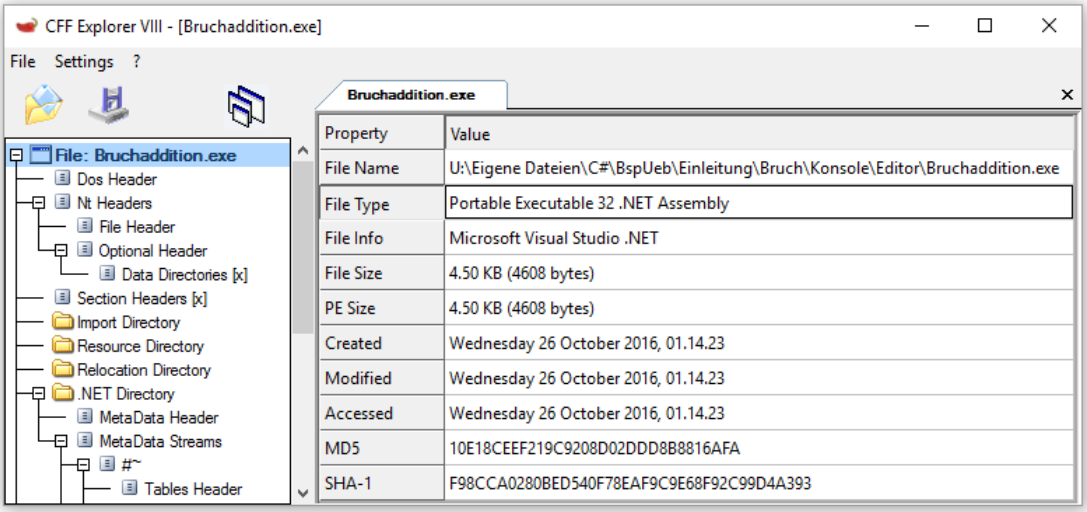
```
                        Visual C# Compiler Optionen
   . . .
/platform:<Zeichenfolge>       Legen Sie eine Beschränkung dafür fest,
                               auf welchen Plattformen dieser Code
                               ausgeführt werden kann: 'x86',
                               'Itanium', 'x64', 'arm',
                               'anycpu32bitpreferred' oder 'anycpu'.
                               Der Standard ist 'anycpu'.
```

Plattform**un**abhängig sind nur die Assemblies mit der **ProcessorArchitecture CIL**.

Auf einem PC mit 64-bittiger Windows-Installation werden *zwei* .NET - Laufzeitumgebungen installiert, sodass Assemblies alternativ in einem 32- oder 64-Bit - Prozess ausgeführt werden können:



Unter einer Windows-Version mit 64 Bit produziert die C# - Compiler bei der voreingestellten Prozessorarchitektur **CIL** ein **Portable Executable 32 .NET Assembly**, wie der kostenlos verfügbare **CFF-Explorer** von Daniel Pistelli zeigt:[1]



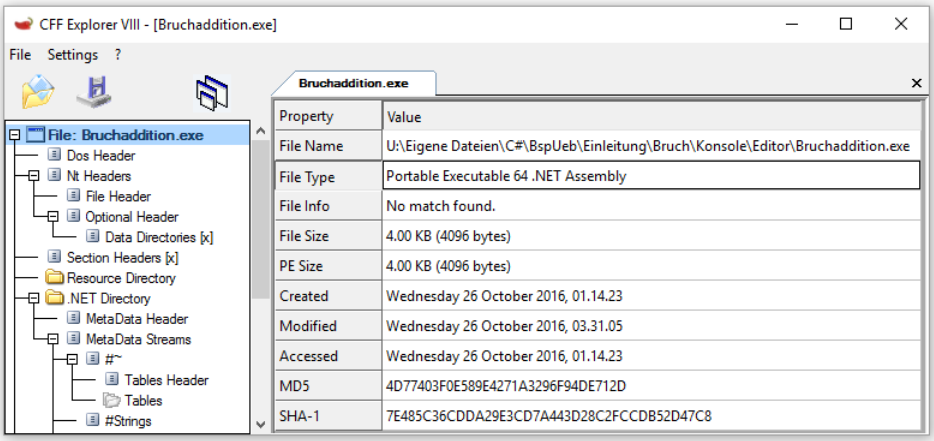
Ein solches Assembly wird unter Windows 64 in einem 64-Bit - Prozess ausgeführt und kann selbstverständlich auch unter Windows 32 genutzt werden.

Wird per Compiler-Option die **ProcessorArchitecture x64** gewählt,

```
csc *.cs /platform:x64
```

dann resultiert ein **Portable Executable 64 .NET Assembly:**

[1] Verfügbar über die Webseite: http://www.ntcore.com/exsuite.php



Es wird auf jeden Fall in einem 64-Bit - Prozess ausgeführt und kann daher unter Windows 32 *nicht* genutzt werden. Eine solche Wahl der Zielplattform ist erforderlich, wenn ein Assembly mit einer Maschinencode-DLL oder mit einer Komponente kooperieren muss, die nur mit 64-Bit-Architektur verfügbar ist.

Wird per Compiler-Option die **ProcessorArchitecture x86** gewählt,

```
csc *.cs /platform:x86
```

dann resultiert ein **Portable Executable 32 .NET Assembly**, das unter Windows 64 in einem 32-Bit - Prozess ausgeführt wird. Eine solche Wahl der Zielplattform ist erforderlich, wenn ein Assembly mit einer Maschinencode-DLL oder mit einer Komponente kooperieren muss, die nur mit 32-Bit-Architektur verfügbar ist.

1.2.5.6 Vergleich mit der COM-Technologie

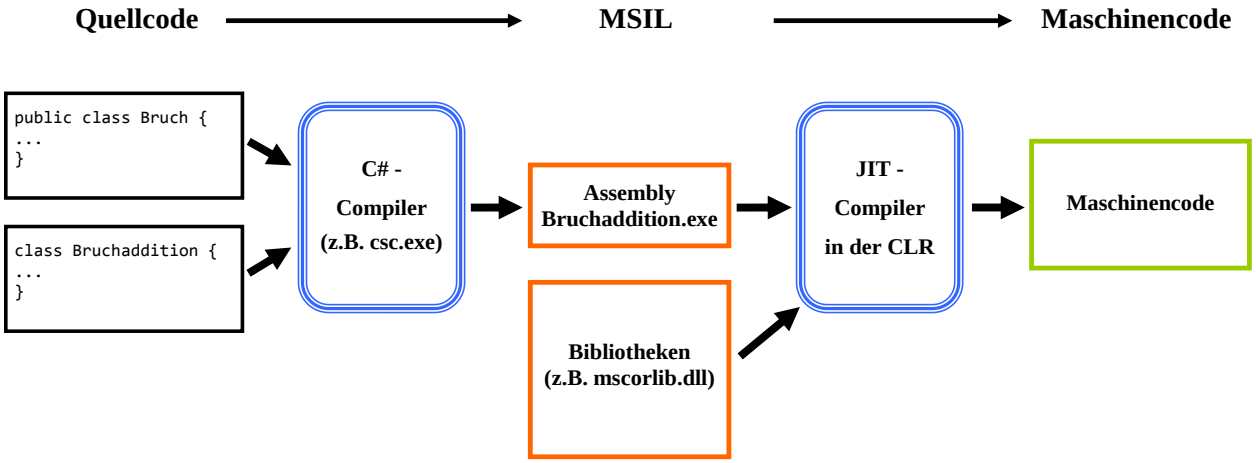
Die Metadaten der .NET -Technologie sorgen dafür, dass die Klassen eines Assemblies unproblematisch von beliebigen .NET - Programmen genutzt werden können. Durch das .NET - Framework soll die COM-Architektur (*Component Object Model*) abgelöst werden soll, die in den 90er Jahren des letzten Jahrhunderts geschaffen wurde, um sprachübergreifende Interoperabilität von Software-Komponenten zu ermöglichen. Hier werden ebenfalls Metadaten bereitgestellt, welche die Typen eines COM-Servers beschreiben. Diese Metadaten werden in der IDL (*Interface Definition Language*) erstellt, befinden sich in separaten Binärdateien (Typbibliotheken) und müssen in die Windows-Registry eingetragen werden. Es kann leicht zu Problemen kommen, weil die Metadaten zu einem COM-Server nicht auffindbar oder fehlerhaft sind. Die Metadaten eines .NET - Assemblies sind demgegenüber stets vorhanden und aktuell. Sie bieten außerdem wichtige Informationen (z.B. Version, Sprache, Abhängigkeiten), die in einer COM-Typbibliothek nur spärlich oder gar nicht vorhanden sind. Außerdem ist bei .NET - Metadaten kein Registry-Eintrag erforderlich.

1.2.6 CLR und JIT-Compiler

Bislang haben Sie erfahren, dass aus dem C# - Quellcode durch einen Compiler (z.B. **csc.exe**) ein Assembly mit Zwischencode (CIL) erzeugt wird. Beim Programmstart ist eine weitere Übersetzung in den Maschinencode der aktuellen CPU erforderlich. Diese Aufgabe wird von der Laufzeitumgebung für .NET - Anwendungen, der **CLR** (*Common Language Runtime*) erledigt. Dazu besitzt sie einen JIT-Compiler (***J**ust **I**n **T**ime*), der Leistungseinbußen aufgrund der zweistufigen Übersetzungsprozedur minimiert (z.B. durch das Speichern von mehrfach benötigtem Maschinencode).

.NET - Programme können auf jedem Windows-Rechner mit passendem Framework auf übliche Weise gestartet werden, z.B. per Doppelklick auf den Namen der Programmdatei.

In der folgenden Abbildung ist der Weg vom Quellcode bis zum ausführbaren Maschinencode für das Bruchadditionsbeispiel dargestellt:



Sorgen um mangelnde Performanz aufgrund der indirekten Übersetzung sind übrigens unbegründet. Eine Untersuchung von Schäpers & Huttary (2003) ergab, dass die Zwischencode-Sprachen C# und Java bei diversen Benchmarks sehr gut mit den „echten Compiler-Sprachen" C++ und Delphi mithalten können. Ein Grund ist wohl darin zu sehen, dass die meist sehr aktuelle CLR die aktuell vorhandene CPU besser kennt und z.B. Befehlserweiterungen besser ausnutzen kann als ein Maschinencode-Compiler, der ...

- älter ist als viele moderne Prozessoren,
- auf maximale Kompatibilität Wert gelegt und manche CPU-Spezifika nicht unterstützt hat, damit sein Produkt auf möglichst vielen CPUs ablauffähig ist.

Das im .NET - Framework enthaltene Hilfsprogramm **ngen.exe** (*Native Image Generator)* kann aus einem Assembly Maschinencode erzeugen und abspeichern, so dass bei der späteren Programmausführung kein JIT-Compiler mehr benötigt wird. Aus verschiedenen Gründen (siehe Richter 2006, S. 43ff) führt dies aber *nicht* unbedingt zu einer beschleunigten Programmausführung.

Die CLR hat bei der Verwaltung von .NET - Anwendungen neben der Übersetzungstätigkeit noch weitere Aufgaben zu erfüllen, z.B.:

- **Verifikation des IL-Codes**
  Irreguläre Aktionen einer .NET - Anwendung werden vom Verifier der CLR verhindert. Das macht .NET - Anwendungen sehr stabil.
- **Unterstützung der .NET - Anwendungen bei der Speicherverwaltung**
  Überflüssig gewordene Objekte werden vom Garbage Collector (Müllsammler) der CLR automatisch entsorgt. Mit diesem Thema werden wir uns noch ausführlich beschäftigen.
- **Überwachung von Code mit beschränkten Rechten**
  Bis zur Version 3.5 war im .NET - Framework die CAS-Technik (***C**ode **A**ccess **S**ecurity*) mit einer fein granulierten Rechteverwaltung orientiert an Merkmalen des jeweiligen Assemblies enthalten. So sollten die Sicherheitsmechanismen des Betriebssystems ergänzt werden, das jedem ausgeführten Programm ebenso viele Rechte einräumt wie dem angemeldeten *Benutzer*. Leider scheiterte die CAS-Technik an der zu hohen Komplexität. Seit .NET 4.0 haben Assemblies dieselben Rechte wie native Windows-Programme. Beschränkungen nach dem Sandkasten-Prinzip gelten jedoch für Code, der via Internet auf einen Rechner gelangt ist.[1]

[1] Siehe http://msdn.microsoft.com/en-us/magazine/ee677170.aspx

1.2.7 FCL und Namensräume

Damit Programmierer nicht das Rad (und ähnliche Dinge) neu erfinden müssen, bietet das .NET - Framework eine umfangreiche Bibliothek mit fertigen Klassen für nahezu alle Routineaufgaben. Dass die als **Framework Class Library** (FCL) bezeichnete Standardbibliothek in *allen* .NET - Programmsprachen zur Verfügung steht, ist für C# - Einsteiger noch wenig relevant. Bei der späteren Teamarbeit kann die sprachunabhängige .NET - Architektur jedoch sehr bedeutsam werden, wenn Anhänger verschiedener Programmiersprachen zusammen treffen.

Statt von der *Framework Class Library* wird oft von der ***B**ase **C**lass **L**ibrary* (BCL) gesprochen, gelegentlich synonym, oft mit einem Vorschlag zur Definition von zwei unterschiedlich breiten Bibliotheken. Wir plagen uns nicht mit Begriffsstreitereien oder überflüssigen Differenzierungen, sondern verwenden die Bezeichnung *FCL* für die Bibliothek mit den in jeder .NET - Installation enthaltenen Klassen.

Die Klassen und sonstigen Datentypen der .NET - Standardbibliothek sind nach funktionaler Verwandtschaft in so genannte **Namensräume** eingeteilt. Dieses Organisationsprinzip dient in erster Linie dazu, Namenskollisionen zu vermeiden. So dürfen z.B. zwei Klassen denselben Namen tragen, sofern sie sich in verschiedenen Namensräumen befinden. Ihre **voll qualifizierten Namen** (inkl. Namensraumbezeichnung) sind dann verschieden.

Namensräume sind keinesfalls auf die FCL beschränkt, und die von C# - Entwicklungsumgebungen angebotenen Vorlagen für neue Projekte (siehe unten) definieren meist einen eigenen Namensraum (**namespace**) für jedes Projekt, z.B.:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
namespace Bruchaddition
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

Bei kleinen Beispielprogrammen sind Namensräume jedoch überflüssig, und wir werden meist der Übersichtlichkeit halber darauf verzichten.

Bei professionellen Projekten sollte man Namensräume verwenden und dabei auch auf sinnvolle Namensraumbezeichnungen achten. Microsoft empfiehlt das folgende Schema für die Namensraumbezeichnungen:[1]

<Company>.{<Product>|<Technology>}[.<Feature>][.<Subnamespace>]

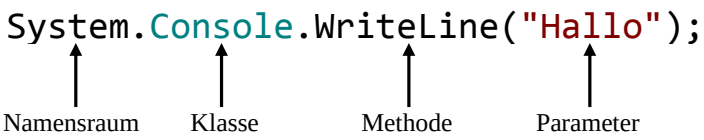
Beispiele:

- **Microsoft.Win32**
- **IBM.Data.DB2**

Eine .NET - Klasse muss grundsätzlich mit ihrem voll qualifizierten Namen angesprochen werden, z.B.:

---

[1] https://msdn.microsoft.com/en-us/library/ms229026(v=vs.110).aspx



Um in einem Programm die Klassen eines Namensraums vereinfacht (ohne Namensraumpräfix) ansprechen zu können, muss der Namensraum am Anfang des Quelltexts per **using**-Direktive importiert werden, z.B.:[1]

```
using System;
  .  .  .
Console.WriteLine("Hallo");
```

Durch diese **using**-Direktive wird dafür gesorgt, dass der Compiler dem Namen jeder Klasse, die nicht im projekteigenen Quellcode definiert ist, das Präfix **System** voranstellt und dann die referenzierten Assemblies (siehe unten) nach dem vervollständigten Namen durchsucht.

Bei Namenskollisionen gewinnt generell der lokalste Bezeichner. Im folgenden Beispiel werden der Namensraumbezeichner **System** und der Klassenname **Console** (im Namensraum **System**) durch lokale Bezeichner verdeckt:

```
using System;
class Prog {
   static int Console = 13;
   static int System = 1;
   static void Main() {
      Console.WriteLine("Hallo");
   }
}
```

Infolgedessen bewirkt die folgende Zeile

```
Console.WriteLine("Hallo");
```

keinen Methodenaufruf, weil der Compiler *`Console`* als **int**-Variablennamen betrachtet und meldet, dass der Datentyp **int** keine Definition für den Bezeichner *`WriteLine`* enthalte:

```
Fehler CS1061 "int" enthält keine Definition für "WriteLine"
```

In der folgenden Zeile

```
System.Console.WriteLine("Hallo");
```

betrachtet der Compiler *`System`* als **int**-Variablennamen und bemängelt, dass der Datentyp **int** keine Definition für den Bezeichner `Console` enthalte. Mit dem Schlüsselwort **global** und dem **::** - **Operator** kann man eine im globalen Namensraum beginnende Namensauflösung anordnen, und die folgende Zeile führt trotz der Verdeckungen zum erwünschten Methodenaufruf:

```
global::System.Console.WriteLine("Hallo");
```

Sie werden in eigenen Programmen das Verdecken von wichtigen Bezeichnern aus der FCL sicher unterlassen. Wenn komplexe Softwaresysteme unter Beteiligung vieler Programmierer entstehen, sind Namenskollisionen aber nicht auszuschließen. Der von Entwicklungsumgebungen automatisch erstellte Quellcode (siehe unten) enthält daher häufig den **::** - Operator in Verbindung mit dem Schlüsselwort **global**.

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

---

[1] Manche Code-Prüfungswerkzeuge empfehlen, bei einer Quellcodedatei mit Namensraumdeklaration die **using**-Direktiven *innerhalb* des Namensraums vorzunehmen, siehe z.B. http://stylecop.soyuz5.com/SA1200.html. Während diese Empfehlung für den (zu vermeidenden!) Fall einer Quellcodedatei mit *mehreren* Namensraumdeklarationen sinnvoll ist, sind die Argumente im Fall einer einzelnen Namensraumdeklaration sehr subtil. Wir machen uns daher keine Sorgen bei der Verwendung der Projektvorlagen unserer Entwicklungsumgebung.

- Klassen, die zum selben Namensraum gehören, können in verschiedenen Assemblies implementiert sein.
- In einem Assembly können Klassen aus verschiedenen Namensräumen implementiert werden, was aber nicht zu empfehlen ist.

Namensräume können hierarchisch untergliedert werden, was speziell bei großen Bibliotheken für Ordnung und entsprechend lange voll qualifizierte Namen mit Punkten zwischen den Unterraum-Bezeichnungen sorgt. Die .NET - Standardbibliothek (FCL) verwendet **System** als Wurzelnamensraum und enthält z.B. im Namensraum

System.Windows.Media.Imaging

Klassen und andere Datentypen zur Unterstützung von Grafiken im Bitmap-Format.

Einen ersten Eindruck vom Leistungsvermögen der FCL vermittelt die folgende *Auswahl* ihrer Namensräume. Diese Auflistung ist für Programmier*einsteiger* allerdings von begrenztem Wert und sollte bei diesem Leserkreis keine Verunsicherung durch die große Anzahl fremder Begriffe auslösen:

Namensraum	Inhalt
System	... enthält grundlegende Basisklassen sowie Klassen für Dienstleistungen wie Konsolenkommunikation oder mathematische Berechnungen. U.a. befindet sich hier die Klasse **Console**, die wir im Einführungsbeispiel für den Zugriff auf Bildschirm und Tastatur verwendet haben.
System.Collections	... enthält Container zum Verwalten von Listen, Warteschlangen, Bitarrays, Hashtabellen etc.
System.Data	... enthält zusammen mit diversen untergeordneten Namensräumen (z.B. **System.Data.SqlClient**) die Klassen zur Datenbankbearbeitung.
System.IO	... enthält Klassen für die Ein-/Ausgabebehandlung im Datenstrom-Paradigma.
System.Net	... enthält Klassen für die Netzwerk-Programmierung.
System.Reflection	... ermöglichst es u.a., zur Laufzeit Informationen über Klassen abzufragen oder neue Methoden zu erzeugen. Dabei werden die Metadaten in den .NET - Assemblies genutzt.
System.Seccurity	... enthält Klassen, die sich z.B. mit Verschlüsselungs-Techniken beschäftigen.
System.Threading	... unterstützt parallele Ausführungsfäden.
System.Web	... unterstützt die Entwicklung von Internet-Anwendungen (inkl. ASP.NET).
System.Windows.Controls	... enthält Klassen für die Steuerelemente einer Windows-Anwendung (z.B. Befehlsschalter, Textfelder, Menüs).
System.XML	... enthält Klassen für den Umgang mit XML-Dokumenten.

An dieser Stelle sollte vor allem geklärt werden, dass beim Einstieg in die .NET - Programmierung mit C# ...

- einerseits eine **Programmiersprache** mit bestimmter Syntax und Semantik zu erlernen
- und andererseits eine umfangreiche **Klassenbibliothek** zu studieren ist, die im Sinne der in Abschnitt 1.1.2 geschilderten Vorteile der objektorientierten Programmierung wesentlich an der Funktionalität eines Programms beteiligt ist.

1.2.8 Zusammenfassung zu Abschnitt 1.2

Als Vorteile der .NET - Technologie für die Softwareentwicklung sind u.a. zu nennen:

- **Sprachintegration**
  Mit C# erstellte Klassen können z.B. auch von VB.NET - Programmierern genutzt werden, sofern bei der Entwicklung auf CLS-Kompatibilität (*Common Language Specification*) geachtet wurde (vgl. Abschnitt 1.2.4).
- **Portabilität**
  Es ist möglich, die .NET-Plattform auf andere Betriebssysteme zu portieren. Das von der Firma Microsoft unterstützte Open Source - Projekt **Mono** ist auf diesem Weg schon weit voran gekommen.[1] Grundsätzlich enthalten ausführbare Programme und Klassenbibliotheken den portablen CIL-Code (*Common Intermediate Language*). Allerdings hat der Zwang zur Integration in historisch gewachsene Software-Landschaften dazu geführt, dass manche Assemblies von einer bestimmten Prozessor-Architektur (z.B. x86, x64) abhängig sind (vgl. Abschnitt 1.2.5.5).
- **Breites Anwendungsspektrum**
  Es kann Software für praktisch jeden Einsatzzweck zur Verwendung auf einem Arbeitsplatzrechner, auf einem Server oder auf einem Smartphone entstehen.

Wir haben im Abschnitt 1.2 u.a. folgende Begriffe kennen gelernt:

- **Common Intermediate Language (CIL)**
  .NET - Compiler (z.B. **csc.exe** bei C#) übersetzen den Quellcode in die *Common Intermediate Language*.
- **Common Language Specification (CLS)**
  Den von allen .NET - Programmiersprachen zu erfüllenden Sprachumfang bezeichnet man *Common Language Specification* (CLS). Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt.
- **Assembly**
  Beim Übersetzen von (im Allgemeinen mehreren Quellcodedateien) entsteht ein Assembly. Dies ist kleinste Einheit von .NET - Software bei der …
  - Weitergabe
  - Versionierung
  - Zuweisung von Sicherheitsbeweisen

  Ein Assembly kann beliebig viele Klassen implementieren. In einem ausführbaren Programm (Namenserweiterung **.exe**) ist eine Startklasse (mit Methode **Main()**) vorhanden. Bei einem Bibliotheks-Assembly endet der Dateiname mit der Erweiterung **.dll**.
- **Metadaten**
  Ein Assembly enthält neben dem CIL-Code auch Metadaten. Die *Typ-Metadaten* enthalten eine Beschreibung der implementierten und der referenzierten Typen. Im Manifest sind die *Assembly-Metadaten* mit Angaben zur Version, zur Sicherheit und zur Abhängigkeit von anderen Assemblies enthalten.
- **Common Language Runtime (CLR) mit Just-In-Time (JIT) - Compiler**
  Die Ausführungsumgebung für CIL-Code, der auch als *managed code* bezeichnet wird, besitzt einen Just-In-Time - Compiler zur Übersetzung von CIL-Code in nativen Maschinencode. Außerdem kümmert sich die CLR um Stabilität (Code-Verifikation), Überwachung von Code mit beschränkten Rechten (via Internet bezogen) und Speicherverwaltung (Garbage Collection).

---

[1] http://www.mono-project.com/

- **Namensräume**
  Indem man eine Klasse in einen Namensraum einfügt, ergänzt man ihren Namen um ein Präfix und vermeidet Namenskollisionen. Man fasst in der Regel funktional verwandte Klassen in einen gemeinsamen Namensraum zusammen, der nach Bedarf hierarchisch in Unterräume aufgeteilt werden kann.
- **Framework Class Library (FCL)**
  In der voluminösen und universellen Standardbibliothek der .NET - Plattform wird von Namensräumen reichlich Gebrauch gemacht.

1.3 .NET Core

Unter dem Namen *.NET Core* hat Microsoft im Juni 2016 eine modernisierte .NET - Variante in der Version 1.0 herausgegeben, die auf *mehreren* Plattformen nutzbar ist:

- Linux (64 Bit)
  Aktuell (Frühjahr 2017) unterstützte Linux-Distributionen:[1] RHEL, Ubuntu, Mint, Debian, Fedora, CentOS, Oracle Linux, openSUSE
- Mac OS X (ab Version 10.11, 64 Bit)
- Windows (ab Version 7)

Mittlerweile ist die Version 1.1 erschienen. Microsoft entwickelt .NET Core als **Open Source - Projekt** (verfügbar via GitHub), so dass externe Entwickler Beurteilungen vornehmen und Lösungen beisteuern können.[2]

Aktuell unterstützt .NET Core plattformübergreifend folgende Anwendungstypen:

- Konsolenanwendungen
- Webanwendungen mit ASP.NET Core
- Datenbankanwendungen mit Entity Framework Core als Brücke zwischen der objektorientierten Programmierung und relationalen Datenbanken

Nur unter Windows 10 sind auf .NET Core basierende Anwendungen mit *grafischer Bedienoberfläche* möglich, nämlich die so genannten *UWP-Anwendungen* (*Universelle Windows-Plattform*), die auf verschiedenen Gerätegattungen (z.B. Desktop-PCs, Mobil-Geräte, X-Box - Spielkonsole) laufen. Damit fehlt eine plattformübergreifende Technik für Desktop-Anwendungen mit grafischer Bedienoberfläche.

.NET Core wird parallel zum .NET Framework 4.x entwickelt und kann noch nicht als ausgereift gelten. Eine komplette Unterstützung durch das Visual Studio wird erst die Version 2017 der Entwicklungsumgebung bieten.

Im Unterschied zum .NET Framework ist .NET Core kein Bestandteil des Windows-Betriebssystems, sondern wird über *NuGet*-Pakete ausgeliefert.[3] Jede .NET Core - Anwendung kann ihre eigene .NET Core - Version mitbringen, während eine .NET Framework - Anwendung darauf angewiesen ist, dass auf dem Zielrechner ein .NET - Framework mit einer benötigten Version installiert ist. Mit den neuen Distributionsweg sollen schnellere Innovationen ermöglicht werden.

Vor allem bei UWP-Anwendungen wird die als *.NET Native* bezeichnete Möglichkeit genutzt, den CIL-Code bereits auf dem Entwicklungsrechner in Maschinencode zu übersetzen, so dass er auf

---

1 https://www.microsoft.com/net/core#linuxubuntu
2 https://github.com/dotnet/core
3 Über das NuGet-Ökosystem kann man Pakete für Microsofts Entwicklungsplattformen (z.B. .NET) beziehen und auch veröffentlichen (siehe z.B. https://docs.microsoft.com/de-de/nuget/).

dem Zielrechner direkt ausgeführt werden kann.[1] In deutlichem Widerspruch zu bisherigen .NET - Lehrmeinungen (siehe z.B. Abschnitt 1.2.6) wird für .NET Native ein Leistungsschub von 40 - 60 % versprochen.

Es ist damit zu rechnen, dass viele aktuell noch bestehende Lücken in der .NET Core - Klassenbibliothek geschlossen werden, um die Migration von bestehenden .NET - Anwendungen zu erleichtern. Die Verwendung von .NET Core ist zum jetzigen Zeitpunkt vor allem wegen der fehlenden GUI-Unterstützung für *alle* Windows-Versionen eher nicht zu empfehlen, und das Manuskript behandelt daher Anwendungen für das „vollständige“ .NET Framework 4.x.

Allerdings steht die Programmiersprache C#, das eigentliche Thema des Manuskripts, definitiv *nicht* zur Disposition. Von den Änderungen sind vor allem spezielle Teile der Standardbibliothek betroffen, so dass einige Kapitel des Manuskripts nach einem Wechsel vom .NET Framework auf .NET Core vermutlich nur noch eingeschränkt gültig sein werden:

- Kapitel 11 über die GUI-Programmierung mit der Windows Presentation Foundation
- Kapitel 14 über die Ein- und Ausgabe von Daten
- Kapitel 15 über Multithreading

Vorläufig weiter für das .NET Framework zu entwickeln, bringt u.a. die folgenden Vorteile:

- Entwicklung von Desktop-Anwendungen mit grafische Bedienoberfläche für *alle* Windows-Versionen
- Verwendung einer ausgereiften, vollständigen Standardbibliothek

Der Microsoft-Blogger Cesar de la Torre stellt im Juni 2016 zur zukünftigen Rolle des .NET - Frameworks (im Sinne von Abschnitt 1.2) fest:[2]

> If you want to build Desktop applications that run on Windows 7 through Windows 10, you need to use the .NET Framework.

Derselbe Autor informiert detaillierte darüber, für welche Anwendungstypen das .NET - Framework bzw. .NET Core verwendet werden sollten.

1.4 Übungsaufgaben zu Kapitel 1

1) Warum steigt die Produktivität der Softwareentwicklung durch objektorientiertes Programmieren?

2) Welche von den folgenden Aussagen sind richtig?

1. In C# kann man nur Software für Windows entwickeln.
2. Das .NET - Framework für Windows wurde in C# programmiert.
3. Unter den .NET - Programmiersprachen zeichnet sich C# durch eine besonders leistungsfähige Standardbibliothek aus.
4. Die Klassen in einem mit C# erstellten DLL-Assembly können auch in anderen .NET - Programmiersprachen (z.B. VB.NET) genutzt werden.

3) Welche Aufgaben erfüllt die Common Language Runtime (CLR)?

4) In welcher Beziehung stehen Assemblies und Namensräumen?

5) Was bedeuten die Abkürzungen CIL, FCL, CLS, COM?

---

[1] https://msdn.microsoft.com/en-us/library/dn584397(v=vs.110).aspx

[2] https://blogs.msdn.microsoft.com/cesardelatorre/2016/06/27/net-core-1-0-net-framework-xamarin-the-whatand-when-to-use-it/

2 Werkzeuge zum Entwickeln von C# - Programmen

In diesem Abschnitt werden kostenlos verfügbare Werkzeuge zum Entwickeln von .NET - Applikationen in der Programmiersprache C# beschrieben. Zunächst beschränken wir uns puristisch auf einen simplen Texteditor zum Erstellen des Quellcodes und ein Konsolenfenster für den direkten Aufruf des (im .NET - Framework enthaltenen) Compilers **csc.exe**, der durch Parameter des Startkommandos über Auftragsdetails informiert wird (z.B. über die Namen der zu übersetzenden Quellcode-Dateien). In dieser sehr übersichtlichen „Entwicklungsumgebung“ werden die grundsätzlichen Arbeitsschritte und einige Randbedingungen besonders deutlich. Im weiteren Verlauf des Kurses kommt dann aber mit Microsofts Visual Studio 2015 Community eine bequeme und leistungsfähige Entwicklungsumgebung zum Einsatz.

2.1 C# - Entwicklung mit Texteditor und Kommandozeilen-Compiler

2.1.1 Editieren

Grundsätzlich kann man zum Erstellen der Quellcode-Datei einen beliebigen Texteditor verwenden, z.B. das im Windows-Zubehör enthaltene Programm **Notepad** (alias **Editor**). Um das Erstellen, Compilieren und Ausführen von C# - Programmen ohne großen Aufwand üben zu können, erstellen wir das unvermeidliche **Hallo**-Programm:

```
using System;

class Hallo {
   static void Main() {
      Console.WriteLine("Hallo, echt .NET hier!");
   }
}
```

In unserem Einleitungsbeispiel (siehe Abschnitt 1.1) wurde einiger Aufwand in Kauf genommen, um einen halbwegs realistischen Eindruck von objektorientierter Programmierung (OOP) zu vermitteln. Das `Hallo`-Beispiel ist zwar angenehm einfach aufgebaut, kann aber als „pseudo-objektorientiert“ (POO) kritisiert werden. Es ist eine einzige Klasse namens `Hallo` mit der einzigen Methode namens **Main()** vorhanden. Beim Programmstart wird die Klasse `Hallo` von der CLR aufgefordert, ihre **Main()** - Methode auszuführen. Trotz Klassendefinition haben wir es praktisch mit einer Prozedur historischer Bauart zu tun, was für den einfachen Zweck des Programms durchaus angemessen ist. In den Kapiteln 2 und 3 werden wir solche pseudo-objektorientierten (POO-) Programme benutzen, um elementare Sprachelemente in möglichst einfacher Umgebung kennen zu lernen. Aus den letzten Ausführungen ergibt sich, dass C# zwar eine objektorientierte Programmierweise nahe legen und unterstützen, aber nicht erzwingen kann.

Das Hallo-Programm eignet sich aufgrund seiner Kürze zum Erläutern wichtiger Regeln, an die Sie sich so langsam gewöhnen müssen. Alle Themen werden aber später noch einmal systematischer und ausführlicher behandelt:

- In der ersten Zeile wird der Namensraum **System** importiert, damit die dort angesiedelte Klasse **Console** im Programm ohne Namensraumpräfix angesprochen werden kann (vgl. Abschnitt 1.2.7). Diese für C# - Programme typische Vorgehensweise soll auch im `Hallo`-Beispiel vorgeführt werden, obwohl sie hier den Schreibaufwand sogar vergrößert.
- Nach dem Schlüsselwort **class** folgt der frei wählbare Klassenname[1]. Hier ist wie bei allen Bezeichnern zu beachten, dass C# zwischen Groß- und Kleinbuchstaben unterscheidet.

[1] Ein paar Restriktionen gibt es beim Klassennamen schon. Z.B. sind die reservierten Wörter der Programmiersprache C# verboten. Nähere Informationen folgen später.

- Dem Kopf der Klassendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf.
- Weil die `Hallo`-Klasse startfähig sein soll, muss sie eine Methode namens **Main()** besitzen. Diese wird beim Programmstart ausgeführt und dient bei „echten“ OOP-Programmen oft dazu, Objekte zu erzeugen.
- Die Definition der Methode **Main()** wird von zwei Schlüsselwörtern eingeleitet, deren Bedeutung für Neugierige hier schon beschrieben wird:[1]
    - **static**
      Mit diesem Modifikator wird **Main()** als **Klassenmethode** gekennzeichnet. Im Unterschied zu den *Instanz*methoden der *Objekte* gehören die Klassenmethoden, oft auch als *statische Methoden* bezeichnet, zur *Klasse* und können ohne vorherige Objektkreation ausgeführt werden (vgl. Abschnitt 1.1.1). Die beim Programmstart automatisch auszuführende **Main()** - Methode der Startklasse muss auf jeden Fall durch den Modifikator **static** als Klassenmethode gekennzeichnet werden. In einem objektorientierten Programm hat sie insbesondere die Aufgabe, die ersten Objekte zu erzeugen (siehe unsere Klasse `Bruchaddition` auf Seite 10).
    - **void**
      Im Beispiel erhält die Methode **Main()** den Typ **void**, weil sie keinen Rückgabewert liefert. Mit Rückgabewerten von Methoden werden wir uns noch gründlich beschäftigen.

---

[1] Die folgende Mega-Fußnote sollte nur lesen, wer im Hallo-Beispielprogramm (z.B. aufgrund von Erfahrungen mit anderen C# - Beschreibungen) den Modifikator **public** vermisst:
Die Methode **Main()** wird beim Programmstart von der CLR aufgerufen. Weil es sich bei der CLR aus Sicht des Programms um einen *externen* Akteur handelt, liegt es nahe, die Methode **Main()** explizit über den Modifikator **public** für die Öffentlichkeit frei zu geben. Generell ist nämlich in C# eine Methode (oder ein Feld) **private** und folglich nur innerhalb der Klasse verfügbar. In der Tat findet man in der Literatur viele Hallo-Beispielprogramme (z.B. bei Gunnerson 2001, Louis et al. 2002, Troelsen 2002) mit dem Modifikator **public** im Kopf der **Main()** - Definition, z.B.:

```
using System;
class Hallo {
    public static void Main() {
        Console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

Allerdings wird **Main()** grundsätzlich *nur* von der CLR aufgerufen, die eben *nicht* wie eine fremde Klasse eingestuft wird. Laut C# - Sprachdefinition (ECMA 2006) ist für die **Main()** - Methode nur der Modifikator **static** vorgeschrieben, und demgemäß erweist sich der Modifikator **public** in der Praxis als überflüssig.
In den Hallo-Beispielprogrammen einiger Autoren (z.B. Eller & Kofler 2005) ist nicht nur die Methode **Main()**, sondern auch die *Klasse* als **public** definiert, z.B.:

```
using System;
public class Hallo {
    public static void Main() {
        Console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

Dies ist *nicht* erforderlich, weil in C# eine (nicht eingeschachtelte) Klasse per Voreinstellung die Schutzstufe **internal** (siehe unten) besitzt und folglich im gesamten Assembly bekannt ist, in das sie vom Compiler einbezogen wird (siehe unten). Außerdem wird die einzige Klasse der Hallo-Beispielprogramme von keiner einzigen anderen Klasse gesucht und benutzt.
Die von mir (aber z.B. auch von Albahari & Albahari (2015) sowie von Mössenböck (2016)) bevorzugte Variante mit dem kompletten Verzicht auf **public**-Modifikatoren für das Hallo-Beispiel und vergleichbare Programme kann folgendermaßen begründet werden:

- Sie hält sich an die ECMA-Sprachbeschreibung (siehe ECMA 2006).
- Es werden keine überflüssigen, unzureichend begründeten Forderungen aufgestellt.

- In der **Parameterliste** einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden. Hinter dem Methodennamen muss auf jeden Fall eine durch runde Klammern eingerahmte Parameterliste angegeben werden, gegebenenfalls - wie bei der Methode **Main()** - eben eine leere.
- Dem Kopf einer Methodendefinition folgt der mit geschweiften Klammern eingerahmte Rumpf mit Variablendeklarationen und sonstigen Anweisungen.
- In der **Main()** - Methode unserer `Hallo`-Klasse wird die (statische) **WriteLine()** - Methode der Klasse **Console** dazu benutzt, einen Text an die Standardausgabe zu senden. Zwischen dem Klassen- und dem Methodennamen steht ein Punkt.
- Während unsere **Main()** - Methodendefinition mit einer leeren Parameterliste auskommt, benötigt der im Methodenrumpf enthaltene **Aufruf** der Methode **Console.WriteLine()** einen Aktualparameter, damit der gewünschte Effekt auftritt. Wir geben eine durch doppelte Anführungszeichen begrenzte Zeichenfolge an.
- Bei einem Methodenaufruf handelt sich um eine **Anweisung**, die in C# mit einem Semikolon abzuschließen ist.

Es dient der Übersichtlichkeit, zusammengehörige Programmteile durch eine gemeinsame Einrücktiefe zu kennzeichnen. Man realisiert die Einrückungen am einfachsten mit der Tabulatortaste, aber auch Leerzeichen sind erlaubt. Für den Compiler sind die Einrückungen irrelevant.

Speichen Sie Ihr Quellprogramm unter dem Namen **Hallo.cs** in einem geeigneten Verzeichnis, z.B. in der Datei

U:\Eigene Dateien\C#\Kurs\Hallo\Editor\Hallo.cs

Im Unterschied zur Programmiersprache Java müssen in C# Klassen- und Dateiname *nicht* übereinstimmen, zwecks Übersichtlichkeit sollten sie es aber in der Regel doch tun.

2.1.2 Übersetzen in CIL

Unsere Entwicklungsumgebung Visual Studio 2015 verwendet einen top-aktuellen C# - Compiler aus dem **Roslyn-Projekt**.[1] Eine Kommandozeilenversion dieses Compilers mit dem Namen **csc.exe** findet sich nach der Installation der Entwicklungsumgebung auf einem Windows-Rechner mit 64-Bit - Architektur im folgenden Ordner:

C:\Program Files (x86)\MSBuild\14.0\Bin

Öffnen Sie ein Konsolenfenster, wechseln Sie in das Verzeichnis mit der eben erstellten Quellcodedatei **Hallo.cs**, und lassen Sie das Programm vom C# - Compiler **csc.exe** übersetzen:[2]

```
"C:\Program Files (x86)\MSBuild\14.0\Bin\csc" Hallo.cs
```

Wer längerfristig per Texteditor und Kommandozeilen-Compiler programmieren möchte, sollte den Ordner mit **csc.exe** in die Definition der Umgebungsvariablen PATH aufnehmen.

---

[1] https://github.com/dotnet/roslyn

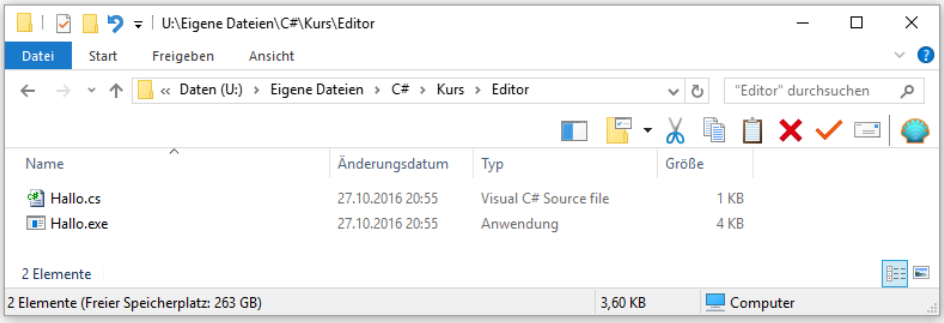
[2] Vor C# 6.0 verwendete man zur Übersetzung per Kommandozeile den C# - Compiler aus dem .NET - Installationsordner, z.B. (bei einer 64-bittigen Windows-Version mit dem .NET-Framework 4.x)

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc Hallo.cs
```

Dieser Compiler meldet nun:

```
This compiler is provided as part of the Microsoft (R) .NET Framework, but only supports
language versions up to C# 5, which is no longer the latest version. For compilers that
support newer versions of the C# programming language, see
http://go.microsoft.com/fwlink/?LinkID=533240
```

Falls beim Übersetzen keine Probleme auftreten (siehe Abschnitt 2.1.4), meldet sich der Rechner nach kurzer Tätigkeit mit einer neuen Kommando-Aufforderung zurück, und die Quellcodedatei **Hallo.cs** erhält Gesellschaft durch die Assembly-Datei **Hallo.exe**, z.B.:



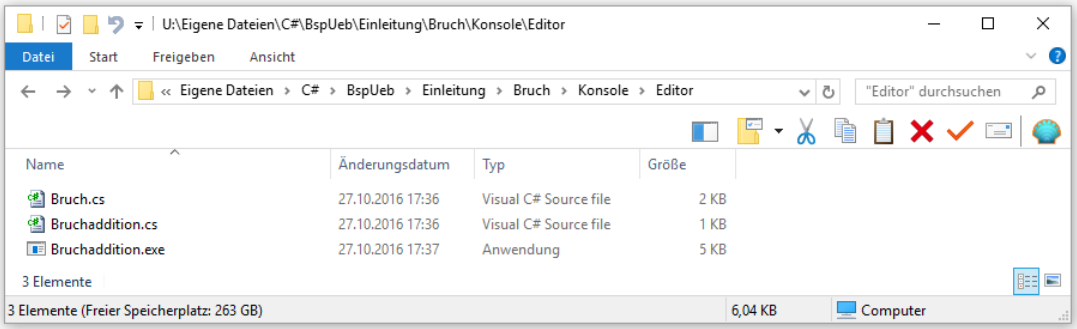
Sind *mehrere* Quellcodedateien in ein Assembly zu übersetzen, gibt man sie beim **csc**-Aufruf hintereinander an, z.B.

```
csc  Bruch.cs BruchAddition.cs
```

Dabei sind auch Jokerzeichen erlaubt, z.B.:

```
csc  *.cs
```

Das entstehende Assembly erbt seinen Namen von der Startklasse (mit der **Main()** - Methode):



Über die Befehlszeilenoption **out** kann der Assembly-Name aber auch frei gewählt werden, z.B.

```
csc  /out:ba.exe  *.cs
```

Diese und die im weiteren Verlauf des Abschnitts beschriebenen Optionen sind beim Compiler aus dem .NET - Framework - Installationsordner *und* bei dem mit unserer Entwicklungsumgebung Visual Studio 2015 installierten Compiler vorhanden.

Mit der Befehlszeilenoption **target** kann ein **Ausgabetyp** gewählt werden:

- **exe**: ausführbares Konsolenprogramm
  Dies ist die Voreinstellung und musste daher in obigen Beispielen nicht angegeben werden.
  Beispiel:
  ```
  csc  /target:exe  Hallo.cs
  ```
- **winexe**: ausführbares Windowsprogramm
  Im Unterschied zum Typ **exe** wird kein Konsolenfenster angezeigt, was im `Hallo`-Beispiel zu einem sinnlosen Programm ohne jeglichen Bildschirmauftritt führen würde.
  Beispiel:
  ```
  csc  /target:winexe  Bruch.cs  BruchGUI.cs
  ```

- **library**: DLL-Assembly
  Die resultierende Bibliothek kann analog zum Assembly **mscorlib.dll** der .NET - Standardbibliothek von anderen Assemblies genutzt werden.
  Beispiel:
  ```
  csc  /target:library  Simput.cs
  ```
- **module**: Teil eines Multidatei-Assemblys (vgl. Abschnitt 1.2.3)

Über die Befehlszeilenoption **reference** wird dem Compiler eine Liste von Assemblies bekannt gegeben, welche die im zu übersetzenden Quellcode verwendeten Klassen implementieren, z.B.:[1]

```
csc  /reference:WPF\PresentationFramework.dll  Prog.cs
```

Zwei Referenz-Assemblies sind durch ein Semikolon zu trennen. Bei Verwendung einer *relativen* Pfadangabe sucht der Compiler in der folgenden Reihenfolge an mehreren Orten nach den referenzierten Dateien:[2]

- im aktuellen Verzeichnis
- in der zum Compiler passenden Version des CLR-Ordners (z.B. in **C:\Windows\Microsoft.NET\Framework64\v4.0.30319**)[3]
- in einem per **lib**-Option benannten Ordner
- in einem per **LIB**-Umgebungsvariable benannten Ordner

Zusätzlich zu den explizit referenzierten Assemblies wird generell das Bibliotheks-Assembly **mscorlib.dll** durchsucht, das elementare und oft benötigte FCL-Klassen implementiert.

Ein C# - Kommandozeilen-Compiler liest per Voreinstellung Referenzen und sonstige Optionen aus einer **Response-Datei**, wenn dies nicht per `noconfig`-Option verhindert wird. Der mit dem Visual Studio 2015 installierte C# - Compiler **csc.exe** (z.B. im Ordner **C:\Program Files (x86)\MSBuild\14.0\Bin**) verwendet eine Response-Datei namens **csc.rsp** mit dem folgenden Inhalt:

```
# Copyright (c)  Microsoft.  All Rights Reserved.  Licensed under the Apache License, Version 2.0.  See
License.txt in the project root for license information.

# This file contains command-line options that the C#
# command line compiler (CSC) will process as part
# of every compilation, unless the "/noconfig" option
# is specified.

# Reference the common Framework libraries
/r:Accessibility.dll
/r:Microsoft.CSharp.dll
/r:System.Configuration.dll
/r:System.Configuration.Install.dll
/r:System.Core.dll
/r:System.Data.dll
/r:System.Data.DataSetExtensions.dll
/r:System.Data.Linq.dll
/r:System.Data.OracleClient.dll
/r:System.Deployment.dll
/r:System.Design.dll
/r:System.DirectoryServices.dll
/r:System.dll
/r:System.Drawing.Design.dll
/r:System.Drawing.dll
/r:System.EnterpriseServices.dll
/r:System.Management.dll
```

---

[1] Das im Beispiel auftauchende Assembly **PresentationFramework.dll** liegt im CLR-Unterordner **WPF**, der explizit angegeben werden muss.

[2] https://msdn.microsoft.com/en-us/library/s5bac5fx(v=vs.80).aspx

[3] In Abschnitt 1.2.5.4 wurde mitgeteilt, dass sich die DLL-Assemblies der FCL im GAC befänden. Nun wird der CLR-Ordner als Stationierungsort angegeben. Beide Ortsangeben stimmen, denn mit dem .NET - Framework werden *zwei* Kopien der FCL-DLLs installiert (siehe Richter, 2012, Kap. 3).

```
/r:System.Messaging.dll
/r:System.Runtime.Remoting.dll
/r:System.Runtime.Serialization.dll
/r:System.Runtime.Serialization.Formatters.Soap.dll
/r:System.Security.dll
/r:System.ServiceModel.dll
/r:System.ServiceModel.Web.dll
/r:System.ServiceProcess.dll
/r:System.Transactions.dll
/r:System.Web.dll
/r:System.Web.Extensions.Design.dll
/r:System.Web.Extensions.dll
/r:System.Web.Mobile.dll
/r:System.Web.RegularExpressions.dll
/r:System.Web.Services.dll
/r:System.Windows.Forms.dll
/r:System.Workflow.Activities.dll
/r:System.Workflow.ComponentModel.dll
/r:System.Workflow.Runtime.dll
/r:System.Xml.dll
/r:System.Xml.Linq.dll
```

Wie an den **csc.rsp** - Einträgen zu sehen ist, kann die Befehlszeilenoption **reference** durch ihren Anfangsbuchstaben abgekürzt werden.

Mit der Befehlszeilenoption

```
/noconfig
```

verhindert man die Auswertung der voreingestellten Antwortdatei. Folglich wird (zeitsparend!) nur noch das zentrale Bibliotheks-Assembly **mscorlib.dll** automatisch durchsucht. Entwicklungsumgebungen bieten zur Verwaltung der in einem Projekt benötigten Referenzen bequeme Bedienelemente (siehe unten).

Die Compiler-Option **platform**, mit der die Abhängigkeit eines Assemblies von einer Prozessor-Architektur deklariert werden kann, wurde schon in Abschnitt 1.2.5.5 erwähnt. Durch Verzicht auf diese Option verwendet man die voreingestellte Platform **CIL** (bzw. **anycpu**). Unter Windows 64 produziert der C# - Compiler daraufhin ein **Portable Executable 32 .NET Assembly**. Es wird unter Windows 64 in einem 64-Bit - Prozess ausgeführt und läuft selbstverständlich auch unter Windows 32.

Über weitere Befehlszeilenoptionen informiert der Compiler beim folgenden Aufruf

```
csc /?
```

2.1.3 Ausführen

.NET - Programme können auf jedem Windows-Rechner mit passendem Framework auf übliche Weise gestartet werden, z.B. per Doppelklick auf den im Windows-Explorer angezeigten Dateinamen. Das Hallo-Programm startet man am besten im Konsolenfenster durch Abschicken seines Namens, z.B.:

Trotz der strengen Unterscheidung zwischen Groß- und Kleinbuchstaben im C# - Quellcode und trotz unserer Entscheidung für einen *großen* Anfangsbuchstaben im Klassennamen `Hallo`, ist auf der Ebene des Windows-Dateisystems, also z.B. beim Starten eines C# - Programms, die Groß/Kleinschreibung natürlich irrelevant.

2.1.4 Programmfehler beheben

Die vielfältigen Fehler, die wir mit naturgesetzlicher Unvermeidlichkeit beim Programmieren machen, kann man einteilen in:

- **Syntaxfehler**
  Diese verstoßen gegen eine Syntaxregel der verwendeten Programmiersprache, werden vom Compiler gemeldet und sind daher schon vor Starten eines Programms relativ leicht zu beseitigen.
- **Semantikfehler**
  Hier liegt kein Syntaxfehler vor, aber das Programm verhält sich anders als erwartet, wiederholt z.B. ständig eine nutzlose Aktion („Endlosschleife“).

Die C# - Designer haben dafür gesorgt, dass möglichst viele Fehler vom Compiler aufgedeckt werden können (z.B. durch strenge Typisierung, Beschränkung der impliziten Typanpassung).[1]

Wir wollen am Beispiel eines provozierten Syntaxfehlers überprüfen, ob der C# - Compiler im .NET - Framework hilfreiche Fehlermeldungen produziert. Wenn im `Hallo`-Programm der Bezeichner **Console** fälschlicherweise mit kleinem Anfangsbuchstaben geschrieben wird,

```
using System;
class Hallo {
    static void Main() {
        console.WriteLine("Hallo, echt .NET hier!");
    }
}
```

meldet der Compiler:

```
Hallo.cs(4,3): error CS0103: Der Name 'console' ist im aktuellen Kontext nicht
         vorhanden.
```

Der Compiler hat die fehlerhafte Stelle sehr gut lokalisiert: Datei **Hallo.cs**, Zeile 4, Spalte 3 (vor dem kleinen *c* stehen zwei Tabulatorzeichen). Auch die Fehlerbeschreibung fällt ziemlich eindeutig aus. Wer Erfahrungen mit Programmiersprachen wie Visual Basic oder Delphi hat, muss sich eventuell noch daran gewöhnen, dass in C# die Groß-/Kleinschreibung signifikant ist.

Im äußerst simplen `Hallo`-Beispiel einen *semantischen* Fehler unterzubringen, den der Compiler nicht bemerkt, ist sehr schwer, vielleicht sogar unmöglich. Im Bruchadditionsbeispiel aus Abschnitt 1.1 stehen unsere Chancen weit besser, Fehler am Compiler vorbei zu schmuggeln. Wird z.B. in der `Nenner` - Eigenschafts-Implementierung bei der Absicherung gegen Nullwerte der Ungleich-Operator (!=) durch sein Gegenteil (==) ersetzt, ist keine C# - Syntaxregel verletzt:

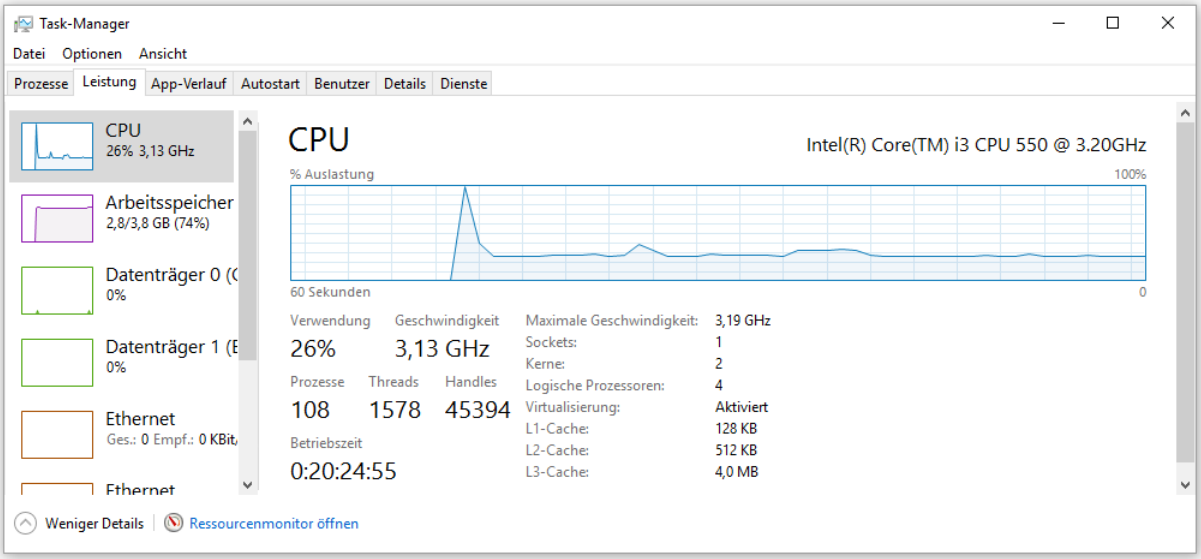
```
public int Nenner {
    get {
        return nenner;
    }
    set {
        if (value == 0)    // semantischer Fehler!
            nenner = value;
    }
}
```

Bei Eingabe kritischer „Brüche“ (wie z.B. $\frac{1}{0}$) zeigt das Programm `BruchAddition` aber ein unerwünschtes Verhalten: Die Methode `Kuerze()` (vgl. Abschnitt 1.1.2) gerät in eine Endlosschleife,

---

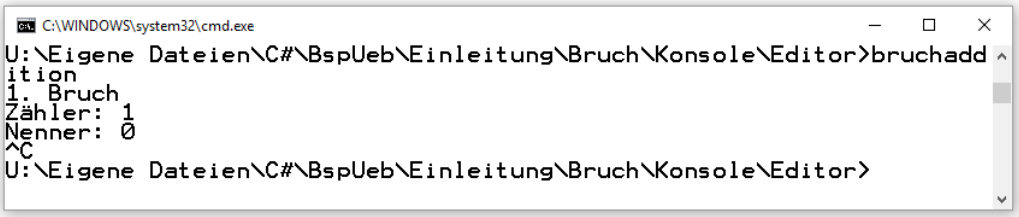
[1] In C# 4.0 taucht mit dem Datentyp **dynamic** eine praktischen Zwängen (z.B. bei der Kooperation mit typfreien Skriptsprachen) geschuldete Ausnahme auf. An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was die Wahrscheinlichkeit von Laufzeitfehlern erhöht.

und das Programm und verbraucht dabei reichlich Rechenzeit, wie der Windows-Taskmanager auf einem Rechner mit der Intel-CPU Core i3 (mit 4 logischen Kernen) zeigt:



Das sinnlos rotierende Programm belegt einen logischer Kern, also 25% der CPU-Zeit.

Ein derart außer Kontrolle geratenes Konsolenprogramm beendet man z.B. mit der Tastenkombination **Strg+C**:



2.2 Microsoft Visual Studio 2015 Community

Auf die Dauer ist das Hantieren mit Notepad und Kommandozeile beim Entwickeln von C# - Software keine ernsthafte Option. Im weiteren Verlauf des Kurses kommt mit Microsofts Visual Studio 2015 Community Edition eine bequeme und leistungsfähige Entwicklungsumgebung zum Einsatz. Das Visual Studio hat mit seinen verschiedenen Varianten seit vielen Jahren einen sehr großen Marktanteil bei der Softwareentwicklung für Windows, unterstützt mittlerweile auch andere Plattformen (z.B. Linux, MacOS-X, Android, iOS) und wird von Fremdfirmen sehr gut durch Erweiterungen für diverse Zwecke unterstützt.

Das Visual Studio bietet u.a. folgende Leistungen:

- Intelligenter Editor (z.B. mit kontextsensitiver Auflistung von Optionen zur Syntaxvervollständigung)
- Grafischer Designer für die Bedienoberfläche eines Programms
- Verschiedene Assistenten, z.B. zur Datenbankanbindung

Die Firma Microsoft stellt mit der Visual Studio 2015 Community Edition eine kostenlose Einstiegsversion ihrer Entwicklungsumgebung Visual Studio 2015 zur Verfügung, die für unsere Kurszwecke sehr gut geeignet ist. Es sind weitere Varianten von Microsofts Entwicklungsumgebung verfügbar, über die Troelsen & Japikse (2015, S. 35ff) ausführlich informieren. Im Vergleich zu den

ebenfalls kostenlosen Express-Editionen des Visual Studios hat die Community-Edition u.a. folgende Vorteile:

- Neben Windows-Programmen kann man auch .NET - Anwendungen erstellen, die unter alternativen Betriebssystemen laufen (z.B. Android, iOS).
- Während man zur Erstellung von Desktop- *oder* Web-Anwendungen jeweils eine spezielle Express-Edition benötigt, beherrscht die Community-Edition *beide* Einsatzfelder und kann außerdem die neuen universellen Anwendungen für Windows 10 erstellen.
- Die Community-Edition unterstützt neben C#, VB.NET und C++ noch weitere .NET - Programmiersprachen (z.B. F#, Python).
- Weitere Projekt-Typen (z.B. für Android, Low-Level C++)
  Beim Umfang der unterstützten Projektvorlagen steht die Community-Edition dem kommerziellen Visual Studio Professional *nicht* nach.
- Wie bei den kommerziellen Varianten steht die Plugin-Technik zur Erweiterung der Entwicklungsumgebung zur Verfügung.
- Visueller Klassen-Designer
  Ein vom Designer erstelltes Diagramm zur Klasse `Bruch` aus dem Einstiegsbeispiel war schon auf der Seite 6 zu sehen.

Viele dieser Leistungen gehören allerdings nicht zur Standardinstallation, sondern erhöhen nach Auswahl im Installationsprogramm den Bedarf an Massenspeicher sehr erheblich.

Weil alle Visual Studio - Varianten eine ähnliche Bedienoberfläche besitzen, ist der Wechsel zu einer anderen Variante kein großes Problem.

2.2.1 Installation

Im Oktober 2016 ist die Visual Studio 2015 Community Edition mit dem Updatestand 3 verfügbar.

2.2.1.1 Voraussetzungen

Offiziell nennt Microsoft sehr bescheidene Systemvoraussetzungen:[1]

- Windows 7 SP 1, Windows 8.x, Windows 10 sowie die korrespondierenden Server-Versionen
- Prozessor mit 1,6 GHz
- 1 GB RAM
- 4 GB Festplattenspeicher

Beim voreingestellten Installationsumfang werden allerdings auf einem Rechner unter Windows 10 ca. 8 GB Massenspeicher (auf Festplatte oder SSD) benötigt.

2.2.1.2 Bezugsquelle

Auf der Webseite

https://www.visualstudio.com/de/downloads/

kann man den **kostenlosen Download** eines Web-Installers zur Visual Studio Community Edition in deutscher Sprache anfordern, der die benötigten Dateien während der Installation aus dem Internet bezieht, oder etwas weiter blättern und das Angebot **Visual Studio 2015** expandieren, so dass u.a. die folgenden Optionen verfügbar werden:

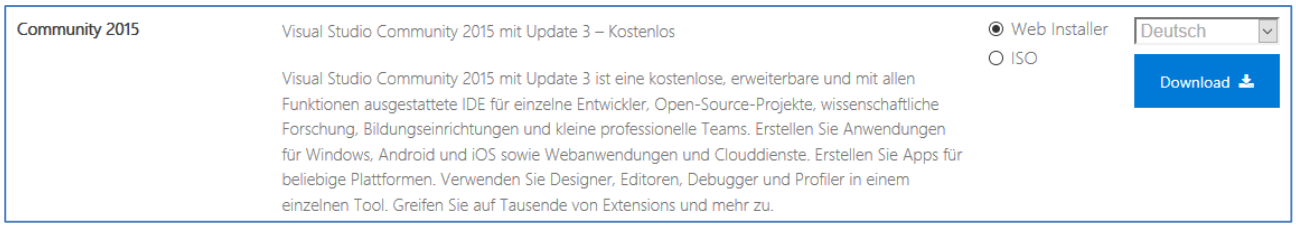
- Web-Installer Community 2015 mit wählbarer Sprache

---

[1] https://www.visualstudio.com/de-de/productinfo/vs2015-sysrequirements-vs

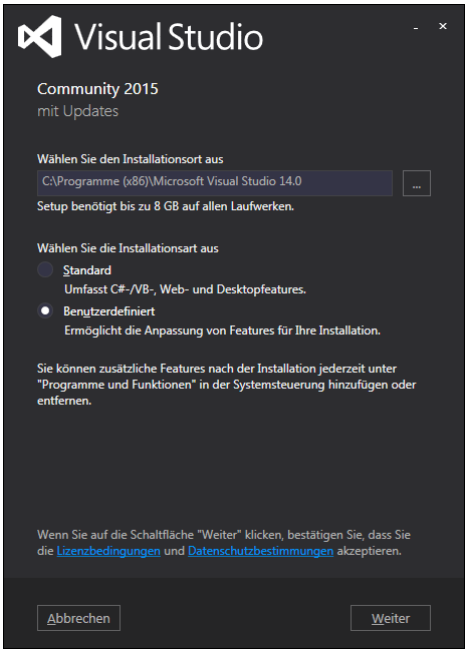
- ISO-Image Community 2015 mit wählbarer Sprache
  Man erhält ein ISO-Image mit ca. 7 GB, das eine Offline-Installation erlaubt.
- Visual Studio 2015 Sprachpaket
  Diese Zusatzinstallation ermöglicht ein Umschalten der Bediensprache, was im Zusammenhang mit der Benutzung von Anleitungen in verschiedenen Sprachen nützlich sein kann.

Anschließend wird der Installation mit dem folgenden Einstieg beschrieben:

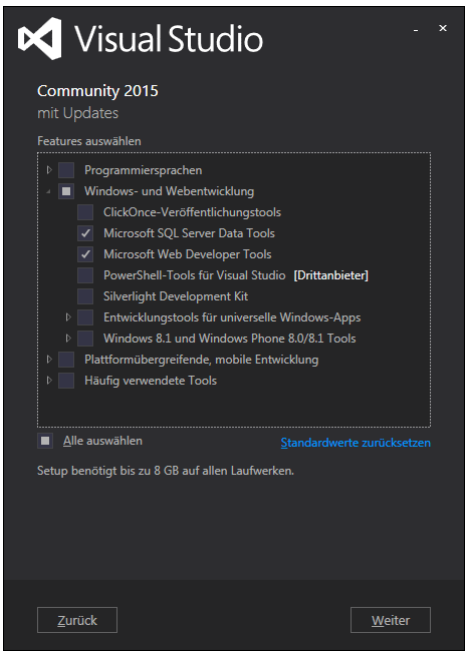


2.2.1.3 Web-Installer

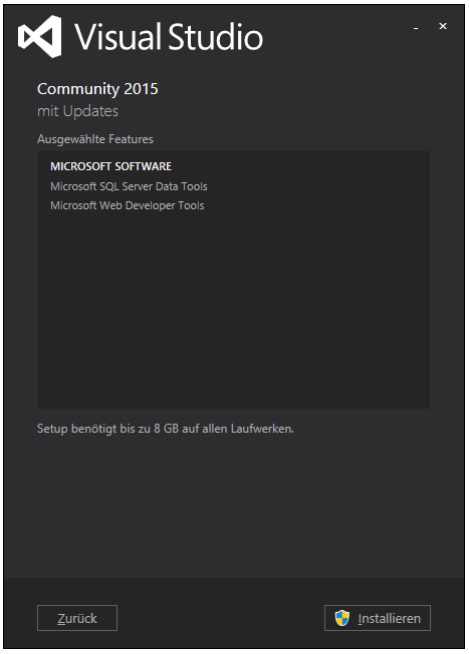
Der Web-Installer (z.B. **vs_community__f7b11a73f7594f5ca7cb8791a75eaf75.exe**) führt nach dem Start einige Initialisierungen durch und präsentiert dann den folgenden Dialog:



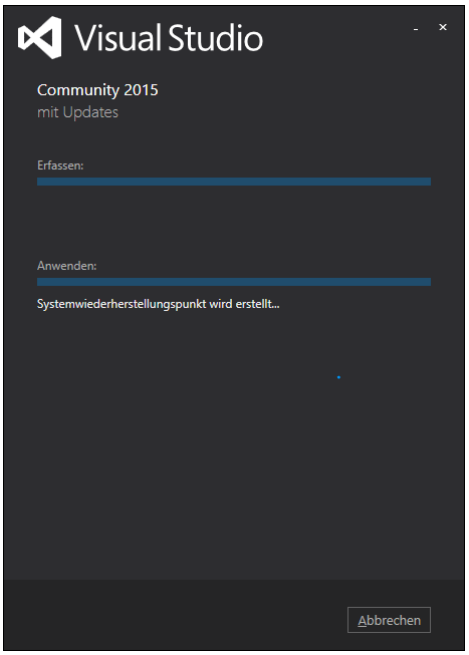
Wir wählen die **benutzerdefinierte** Installation und erweitert nach einem Klick auf **Weiter** den voreingestellten Installationsumfang um die **Microsoft SQL Server Data Tools**, die Werkzeuge zum Erstellen und Verwalten von Datenbanken in die Entwicklungsumgebung integrieren:



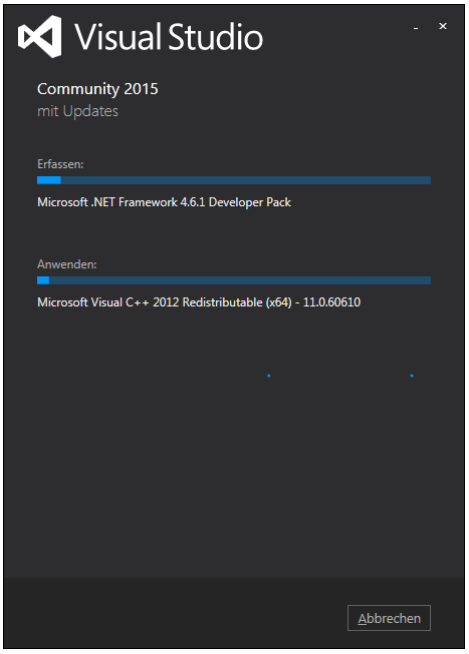
Mit diesem Plan starten wir die Installation:



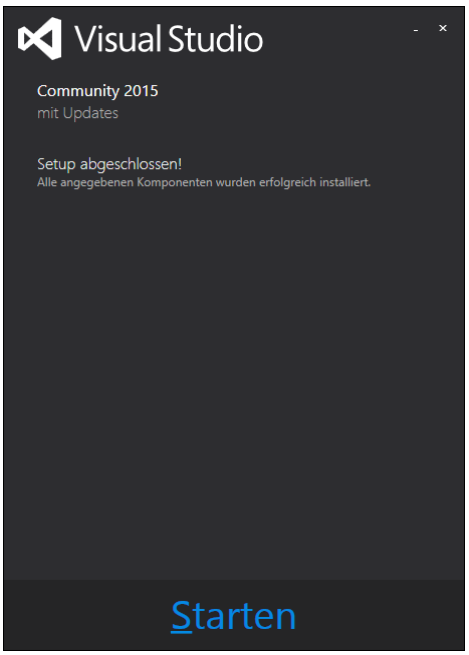
Nachdem wir der Windows-Benutzerkontensteuerung unser Vertrauen gegenüber dem Installationsprogramm bestätigt haben, wird zunächst ein Systemwiederherstellungspunkt angelegt:



U.a. in Abhängigkeit von der Internet-, CPU- und Festplattengeschwindigkeit kann die Installation recht lange dauern:



Auf einem Desktop-Rechner unter Windows 10 mit Intel-CPU Core i3, SSD und flotter Internetanbindung mit DSL 50 sind ca. 30 Minuten vergangen bis zur Erfolgsmeldung:

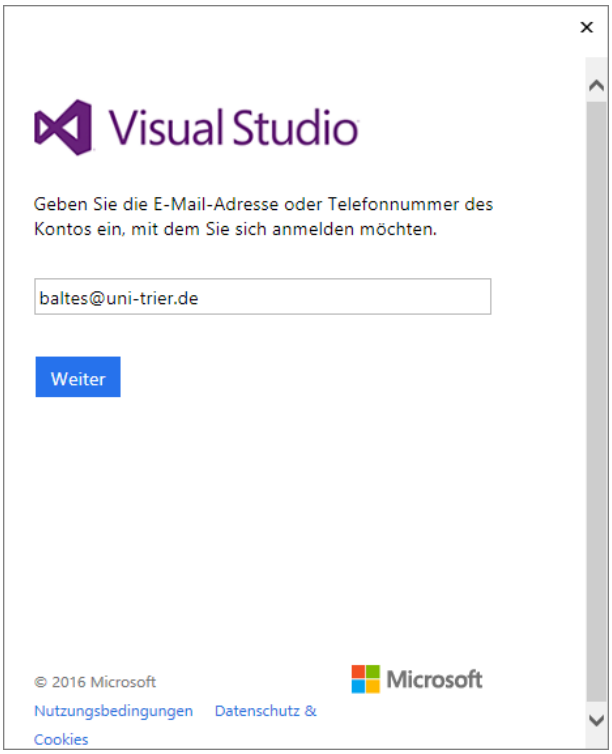


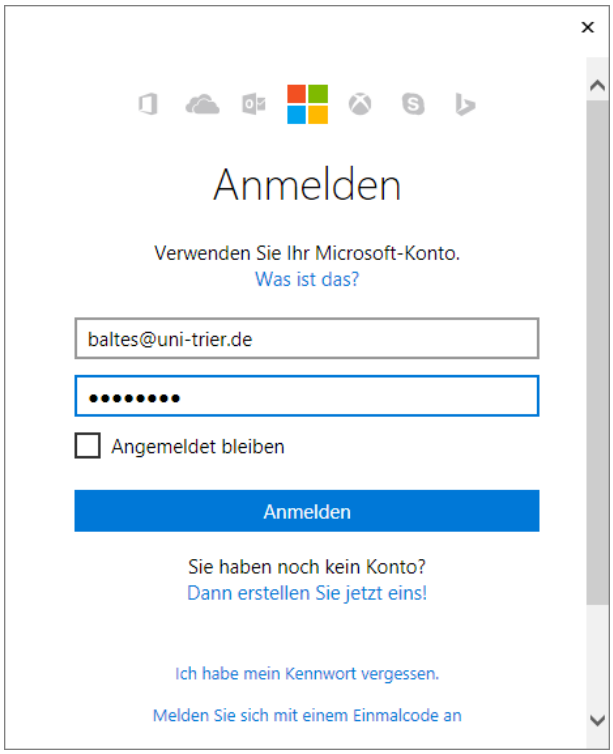
Der vom Installer angemeldete Massenspeicherbedarf (8 GB) wurde ziemlich genau eingehalten. Wenn Softwarevoraussetzungen für das Visual Studio fehlen (speziell das .NET - Framework in der Version 4.6), fällt der Massenspeicherbedarf deutlich höher aus, und es wird ein Neustart nach oder während der Installation fällig.

2.2.2 Initialisierung und Registrierung

Microsoft verlangt für die kostenlosen Varianten seiner Entwicklungsumgebung eine Registrierung durch die Anmeldung mit einem Microsoft-Konto, die sich in den ersten 30 Tagen aufschieben lässt:

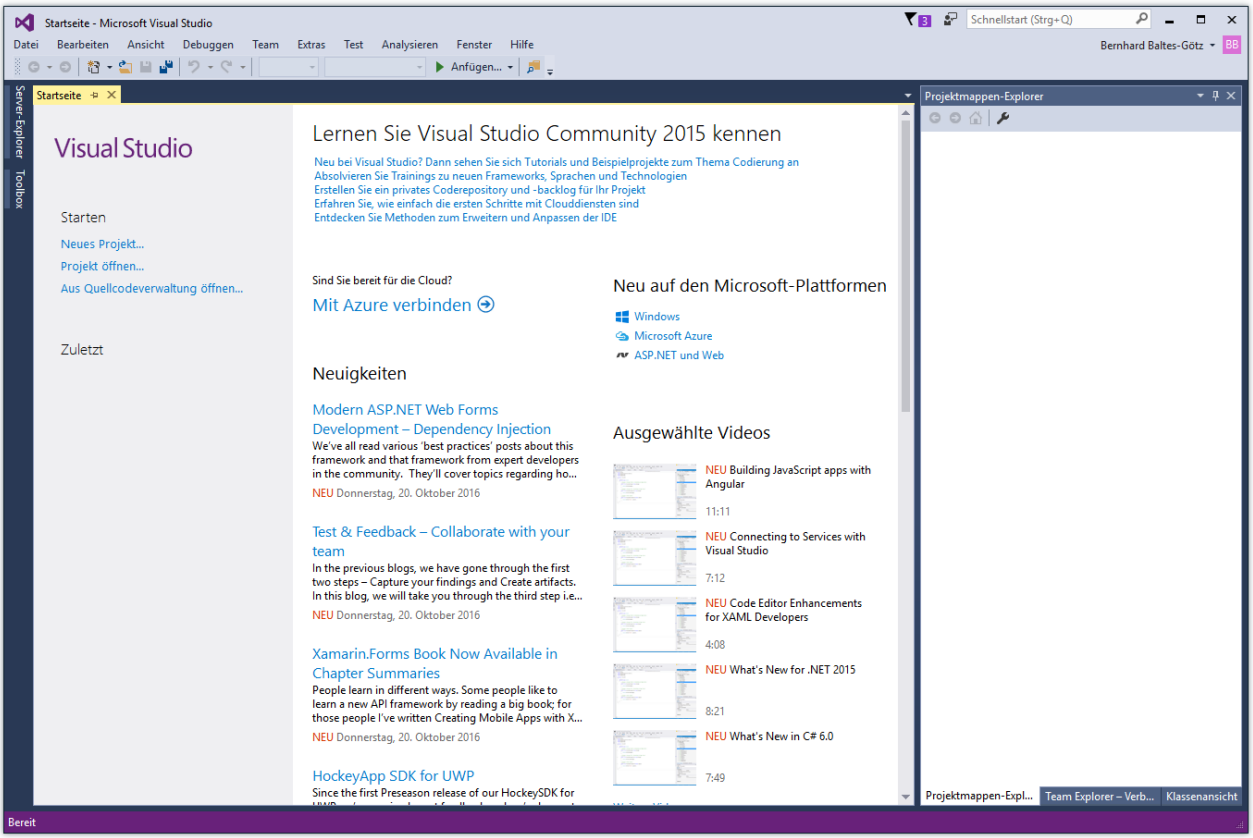
Mit einem vorhandenen Microsoft-Konto ist die Registrierung nach dem **Anmelden** schnell erledigt:





Beim ersten Start der Entwicklungsumgebung sind einmalige Vorbereitungen erforderlich, die je nach Rechner Sekunden oder Minuten dauern:

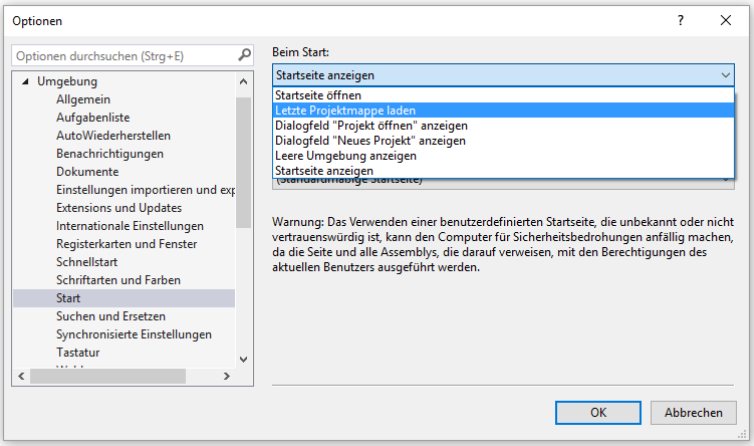
Anschließend unterbreitet die **Startseite** ein erschlagendes Informationsangebot:



Das Startverhalten der Entwicklungsumgebung lässt sich nach

Extras > Optionen > Umgebung > Start

im folgenden Dialog beeinflussen:

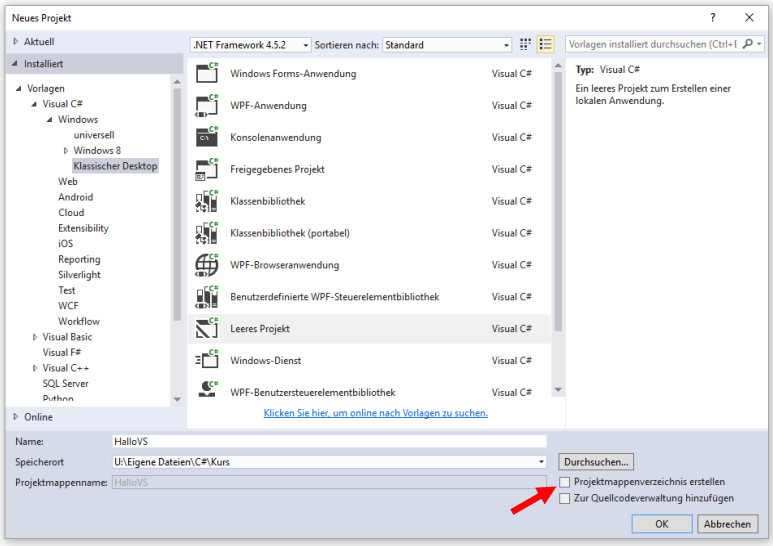


2.2.3 Ein erstes Konsolen-Projekt

Von der Startseite ausgehend öffnen wir mit dem Schalter oder dem Menübefehl

Datei > Neu > Projekt

den Dialog für **neue Projekte**:



Wählen Sie aus der Abteilung

Visual C# > Windows > Klassischer Desktop

die Vorlage **Leeres Projekt** sowie den **Namen** `HalloVS` und einen **Speicherort** (ein übergeordnetes Verzeichnis zum neu anzulegenden Projektordner). Auf einem ZIMK-Pool-PC an der Universität Trier eignet sich als Speicherort z.B.:

U:\Eigene Dateien\C#\Kurs

Jedes Projekt gehört zu einer **Projektmappe**, die eine *Familie* von zusammengehörigen Projekten (z.B. Klient- und Server-Anwendung für einen Dienst) verwaltet und von der Entwicklungsumgebung automatisch angelegt wird. Von der englischen Version der Entwicklungsumgebung wird eine Projektmappe als *Solution* bezeichnet, und gelegentlich taucht die deutsche Übersetzung *Lösung* auf. Bei unseren Kursbeispielen werden die Projektmappen jeweils nur ein einziges Projekt enthalten. In dieser Situation ist es überflüssig, im Ordner einer Projektmappe einen Unterordner für das einzige enthaltene Projekt anzulegen. Daher entfernen wir die Markierung beim Kontrollkästchen **Projektmappenverzeichnis erstellen** und wählen damit eine flachere Projektdateiverwaltung.

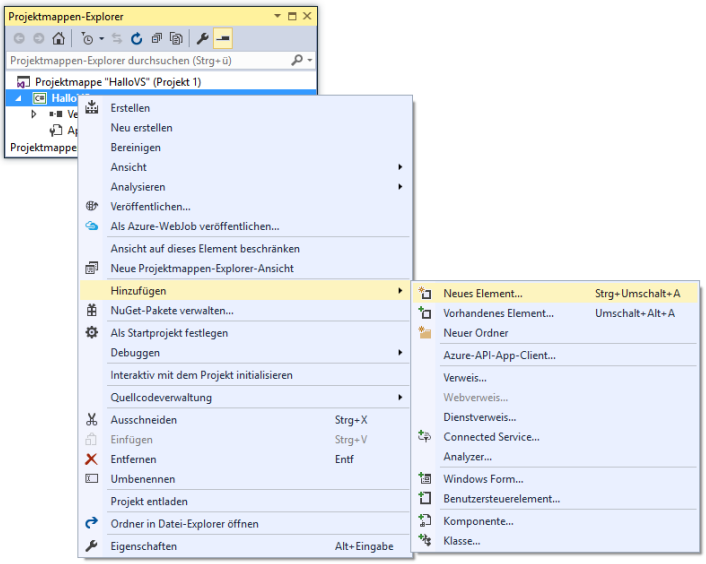
Nach einem Mausklick auf **OK** entsteht im Projekt(mappen)ordner

U:\Eigene Dateien\C#\Kurs\HalloVS

das neue Projekt mit den folgenden Dateien, über die sich das Projekt später öffnen lässt:

- **HalloVS.sln** (Projektmappe)
- **HalloVS.csproj** (Projekt)

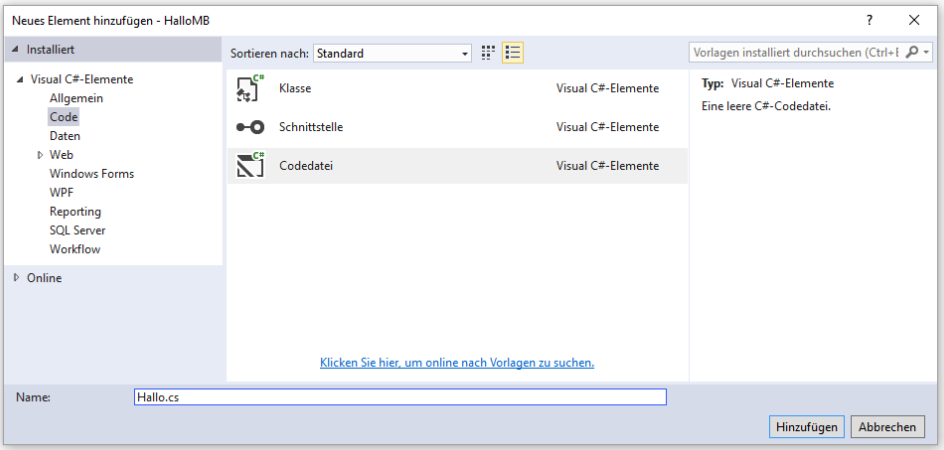
Öffnen Sie im **Projektmappen-Explorer** (am rechten Fensterrand) per Maus-Rechtsklick das Kontextmenü zum *Projekt* (nicht zur *Projektmappe*), und fügen Sie ein **neues Element** hinzu:



Zum selben Zweck taugt bei markiertem Projekt auch der Menübefehl

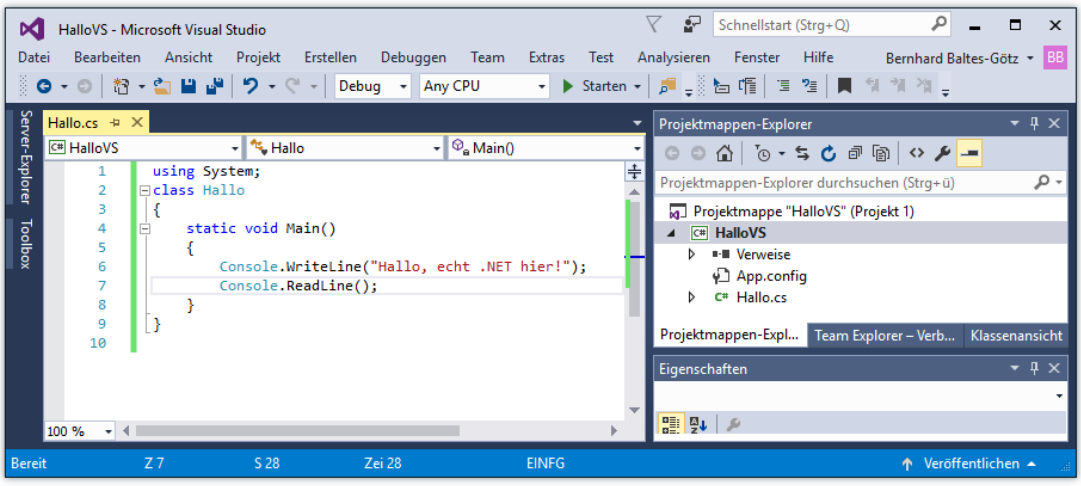
Projekt > Neues Element hinzufügen

Entscheiden Sie sich im Dialog für **neue Elemente** für eine **Codedatei** mit dem **Namen** `Hallo.cs`:



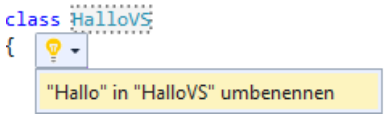
Nach dem **Hinzufügen** ist die Datei im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Wir übernehmen den Quellcode vom `Hallo`-Beispielprogramm aus Abschnitt 2.1.1 und bitten am Ende der **Main()** - Methodendefinition die Klasse **Console**, ihre Methode **ReadLine()** auszuführen:



Diese Methode wartet auf die **Enter**-Taste und verhindert so, dass die im Rahmen der Entwicklungsumgebung mit dem Schalter ▶ Starten oder der Funktionstaste **F5** gestartete Konsolenanwendung nach ihrer Bildschirmausgabe sofort verschwindet.

Außerdem passen wir den Namen der Startklasse an den Projektnamen an. Das Umbenennen einer Klasse, Methode oder Variablen kann in einem komplexen Projekt diverse Quellcodeänderungen erfordern, ist also aufwendig und fehleranfällig. Zum Glück kann unsere Entwicklungsumgebung solche Umgestaltungen, die man zu den *Refaktorierungen* rechnet, sehr gut unterstützen. Wenn Sie mit der Maus auf den geänderten Klassennamen zeigen, erscheint eine Glühbirne,

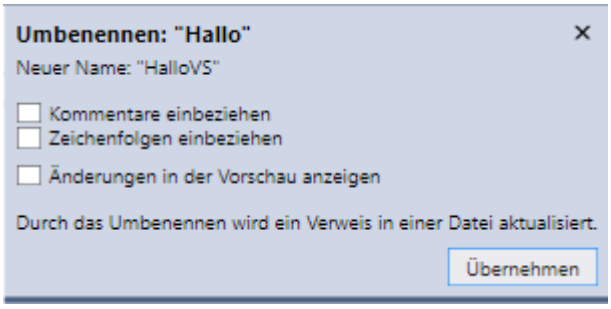


deren Drop-Down - Menü das Projekt-globale Umbenennen der Klasse anbietet. Bei unserem extrem primitiven Programm war kein weiteres Auftreten des Klassennamens zu aktualisieren. Sie werden aber bald die Möglichkeit schätzen lernen, in einem komplexen Programm einen Bezeichner zu ändern, ohne über die damit erzwungenen Aktualisierungen nachdenken zu müssen.

Statt durch Umbenennen eine schwierige Lage zu provozieren, auf die das Visual Studio mit einem Geistesblitz reagiert, sollte man das Umbenennen mit Ansage über die Bühne bringen:

- Setzen Sie die Schreibmarke auf den zu ändernden Bezeichner.
- Fordern Sie das Umbenennen an mit der Tastenkombination **Strg+R**, **Strg+R** oder mit dem folgenden Menübefehl:

  Bearbeiten > Umgestalten > Umbenennen
- Ändern Sie den Bezeichner und klicken Sie auf **Übernehmen**:

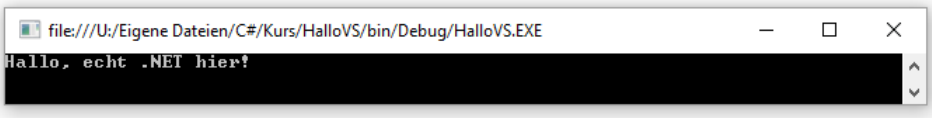


Das Visual Studio sieht übrigens nach dem Umbenennen einer Klasse keinen Anlass zu einer korrespondierenden Umbenennung der zugehörigen Quellcodedatei, weil C# für den Klassen- und den Dateinamen keine Verwandtschaft verlangt.

Speichern Sie das Projekt über den Schalter oder den Menübefehl

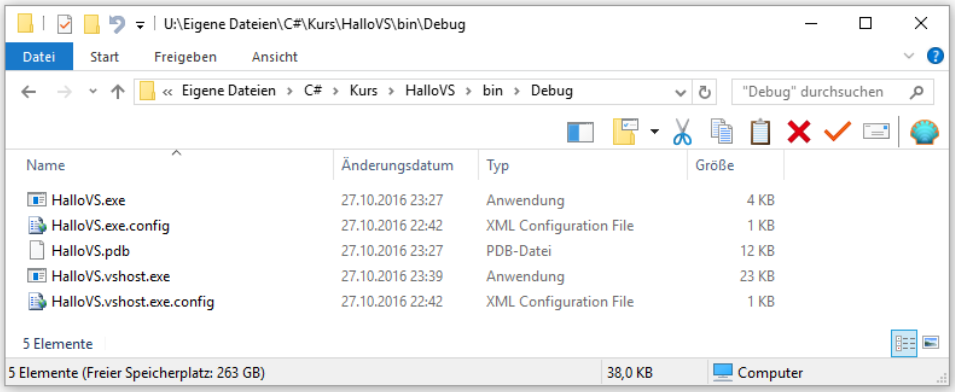
Datei > Alles Speichern

Über den Schalter ▶ Starten oder die Funktionstaste **F5** veranlasst man das Übersetzen und das anschließende Starten der Anwendung:



Sobald Sie die **Enter**-Taste drücken, endet die von der Klasse **Console** ausgeführte **ReadLine()** - Methode. Damit ist auch die **Main()** - Methode unserer Klasse fertig. Das Programm ist somit beendet, und das Konsolenfenster verschwindet.

Beim Programmstart mit **F5**, ▶ Starten oder mit dem Menübefehl **Debuggen > Debugging starten** wird die Debug-Konfiguration verwendet, so dass der Compiler ein gut testbares, aber nicht leistungsoptimiertes Assembly erzeugt und im Projekt-Unterordner **…\bin\Debug** ablegt, z.B.:



Außerdem zeigt das Visual Studio während der Programmaktivität diverse diagnostische Informationen an, mit denen wir momentan noch nicht viel anfangen können.

Beim Programmstart mit **Strg+F5** verwendet die Entwicklungsumgebung ebenfalls die Debug-Konfiguration (mit dem Ausgabeordner **…\bin\Debug**), verzichtet aber während des Programmlaufs auf die Anzeige diagnostischer Informationen. Außerdem wird hinter den Programmaufruf ein **Pause**-Kommando gesetzt, sodass kein **ReadLine()** - Methodenaufruf benötigt wird, um die letzte Ausgabe des Programms auf dem Bildschirm zu halten, z.B.:

Mit der bei auslieferungsbereiten Programmen sinnvollen Release-Konfiguration werden wir uns später beschäftigen. In jedem Fall ist das Assembly **HalloVS.exe** einsatzfähig und kann auf jedem Windows-Rechner mit .NET - Framework ausgeführt werden.

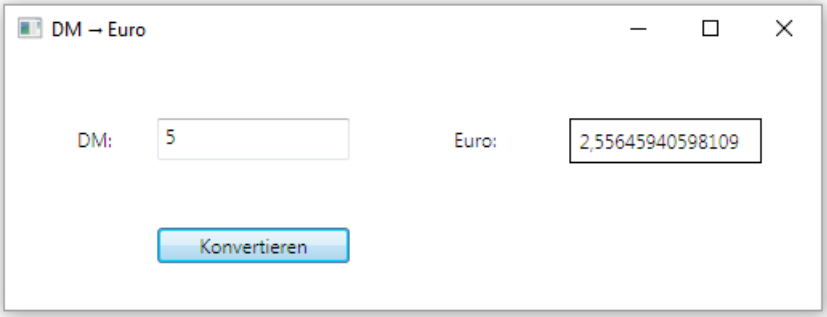
2.2.4 Eine erste GUI-Anwendung

Dieser Abschnitt bietet zwecks Steigerung Ihrer Motivation einen Vorausblick auf die Erstellung von Programmen mit GUI-Bedienung (*Graphical User Interface*) unter Verwendung von RAD-Werkzeugen (***R**apid **A**pplication **D**evelopment*).

2.2.4.1 Projekt anlegen

Wir erstellen einen Währungskonverter mit grafischer Bedienoberfläche, der DM-Beträge in Euro-Beträge wandeln kann:[1]

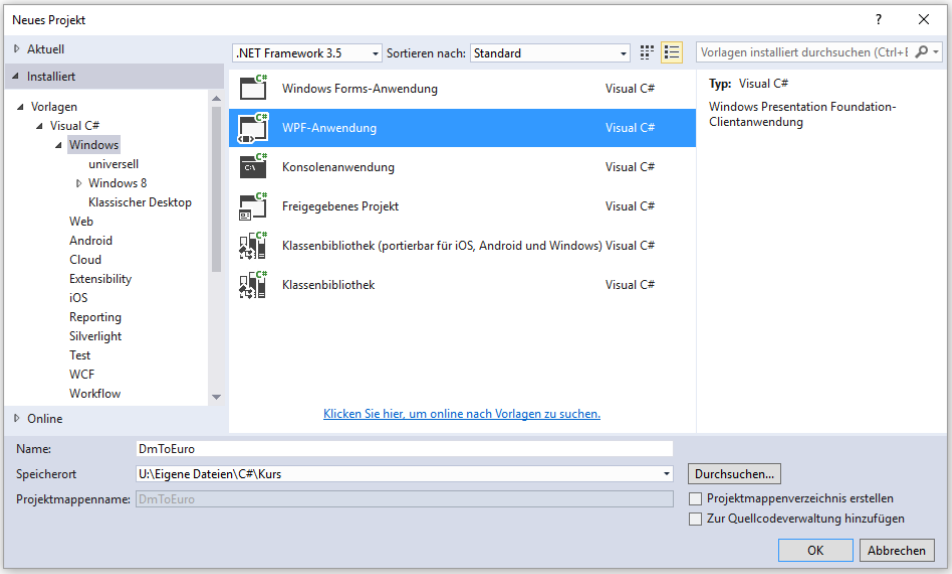
[1] Benötigt wird ein solches Programm z.B. noch bei der Auszahlung von Erbschaftsbeträgen zu Testamenten, welche in der DM-Ära verfasst wurden.



Nach dem Menübefehl

Datei > Neu > Projekt

wählen wir im folgenden Dialog



aus der Abteilung

Visual C# > Windows

die Projektvorlage **WPF-Anwendung**, so dass unsere Anwendung die mit .NET 3.0 eingeführte GUI-Bibliothek *Windows Presentation Foundation* (WPF) verwenden wird.

Wir werden im Kurs die WPF - GUI-Technik aus später noch im Detail zu diskutierenden Gründen gegenüber den Alternativen *WinForms* (veraltet) und *Modern UI* (basierend auf WinRT, keine Unterstützung für Windows 7) bevorzugen.

Als **Name** für das Projekt eignet sich z.B. `DmToEuro`, und als **Speicherort** (übergeordnetes Verzeichnis) zum neu anzulegenden Projektordner können Sie auf einem ZIMK-Pool-PC der Universität Trier analog zu Abschnitt 2.2.3 z.B.

U:\Eigene Dateien\C#\Kurs

verwenden.

Indem wir lediglich ein .NET - Framework in der Version 3.5 voraussetzen (siehe Drop-Down - Liste über den Projektvorlagen), wird das entstehende Programm auf jedem Windows-Rechner ab Vista verwendbar sein, ohne das .NET - Framework aktualisieren zu müssen. Wenn auf einem Rechner das in Visual Studio angegebenen Zielframework *nicht* vorhanden ist, muss die Ausfüh-

rung des Programms dort nicht notwendigerweise scheitern. Microsoft sagt zu dieser Kompatibilitätsfrage:[1]

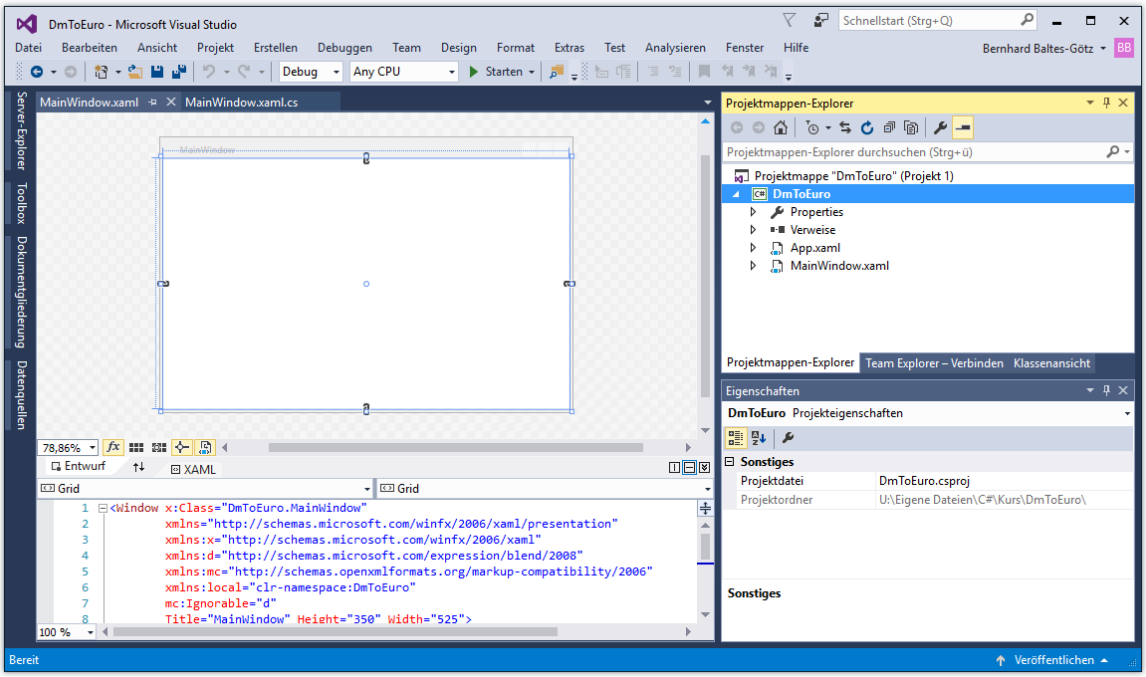
> By default, an app runs on the version of the .NET Framework that it was built for. If that version is not present and the app configuration file does not define supported versions, a .NET Framework initialization error may occur. In this case, the attempt to run the app will fail.

Bei einem Test konnte eine für das Zielframework 4.6.1 entwickelte Anwendung auf einem Vista-PC mit .NET 4.5 problemlos ausgeführt werden.

Wie schon in Abschnitt 2.2.3 begründet wurde, ist ein **Projektmappenverzeichnis** für unsere Übungsprojekte meist überflüssig.

Nach einem Mausklick auf **OK** präsentiert das Visual Studio im **Projektmappen-Explorer** am rechten Fensterrand eine Baumansicht zur Projektmappenverwaltung. Hier erscheint ein Eintrag für jedes Projekt in der potentiell aus mehreren Projekten bestehenden Projektmappe. Im aktuellen, für den Kurs typischen Beispiel befindet sich nur *ein* Projekt in der Mappe, und beide tragen denselben Namen. Zu jedem Projekt werden u.a. die Quellcode-Dateien zu den Klassendefinitionen sowie die Verweise auf benötigte Bibliotheks-Assemblies (siehe Abschnitt 2.2.6.1) aufgelistet. Wir werden die Bestandteile bei passender Gelegenheit behandeln.

Auf der linken Seite präsentiert der Fenster- bzw. WPF-Designer einen Rohling für das Hauptfenster der Anwendung:



Falls Sie unter dem **Projektmappen-Explorer** kein **Eigenschaften**-Fenster sehen sollen, schalten Sie es bitte mit dem Menübefehl

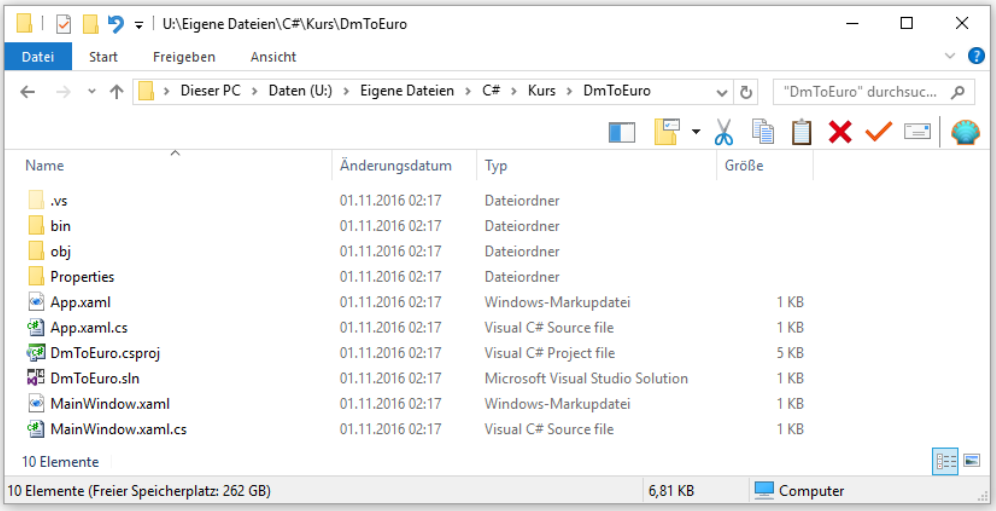
Ansicht > Eigenschaftenfenster

oder mit der Funktionstaste **F4** ein.

Weil wir uns *gegen* Projektordner im Projektmappenordner entschieden haben, resultieren im Beispiel folgende Ordner und Dateien:

---

[1] https://msdn.microsoft.com/en-us/library/ff602939(v=vs.110).aspx



Bei Verwendung der Programmiersprache C# besitzt eine Projektdatei die Namenserweiterung **csproj**, im Beispiel:

U:\Eigene Dateien\C#\Kurs\DmToEuro\DmToEuro.csproj

Bei einer Projektmappendatei wird die Namenserweiterung **sln** verwendet, im Beispiel:

U:\Eigene Dateien\C#\Kurs\DmToEuro\DmToEuro.sln

Um ein Projekt über den Windows-Explorer zu öffnen, setzt man einen Doppelklick auf die Projekt- oder auf die Projektmappendatei.

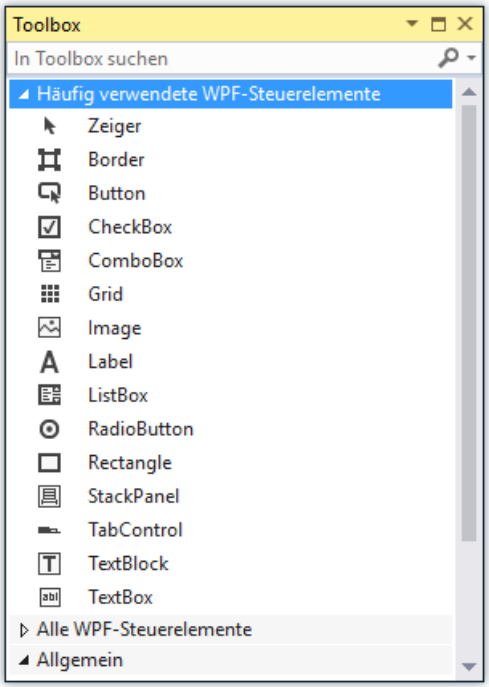
2.2.4.2 Bedienoberfläche entwerfen

Wir machen uns nun daran, das Hauptfenster unseres Währungskonverters mit den benötigten Bedienelementen (Steuerelementen, Controls) auszustatten. Während wir mit dem WPF-Designer fast wie mit einem Grafikprogramm arbeiten, erstellt und pflegt dieser Assistent eine Deklaration der Benutzeroberfläche in der ***E**xtensible **A**pplication **M**arkup **L**anguage* (XAML). Mit zunehmendem Wissen über die XAML-Beschreibungsprache werden wir später unsere Abhängigkeit vom Assistenten reduzieren. In der aktuellen Lernphase ändern wir den XAML-Code nur indirekt mit Hilfe des WPF-Designers.

Die Bedienelemente können aus dem **Toolbox**-Fenster per Drag & Drop (Ziehen & Ablegen) übernommen werden. Öffnen Sie bitte dieses Fenster mit dem Menübefehl

Ansicht > Toolbox

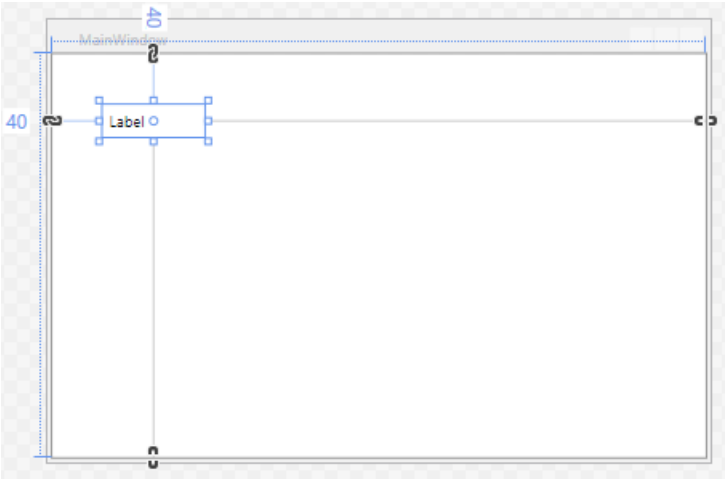
oder durch einen Mausklick auf die **Toolbox**-Schaltfläche am linken Fensterrand, und erweitern Sie nötigenfalls die Liste mit den **Häufig verwendeten WPF-Steuerelementen**:



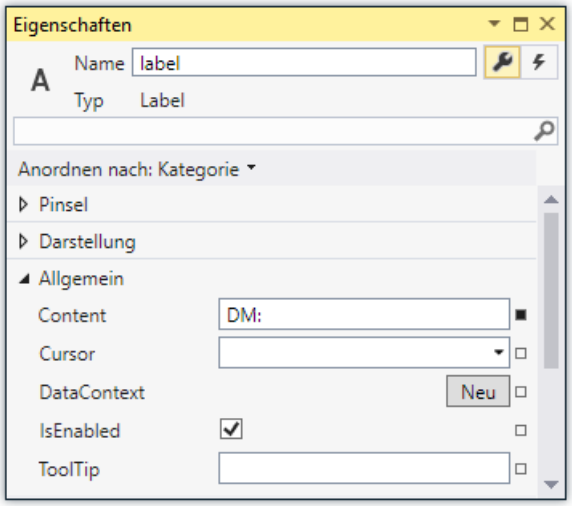
Erstellen Sie ein **Label**-Objekt (die Bezeichnung *Objekt* ist durchaus im Sinn von Abschnitt 1.1 gemeint) auf dem Formular, ...

- entweder per Doppelklick auf den **Toolbox**-Eintrag **Label**
- oder per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den **Toolbox**-Eintrag **Label** setzen, die Maus dann mit gedrückter Taste zum Ziel bewegen und dort die Taste wieder loslassen.

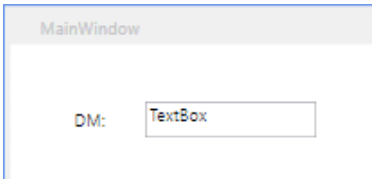
Durch die Wahl einer passenden Mauszeigerposition erhält man das Werkzeug zum Bewegen von Objekten oder das Werkzeug ⟷ zur horizontalen Größenveränderung. Damit lässt sich der folgende Zustand herstellen:



Ändern Sie die Beschriftung des **Label**-Objekts über einen passenden Wert für die Eigenschaft **Content**:

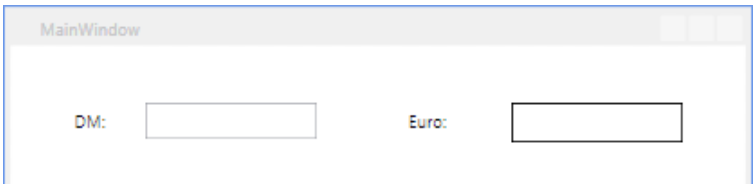


Setzen Sie ein Objekt der Klasse **TextBox** neben das **Label**-Objekt, wobei die Entwicklungsumgebung bei der Anpassung von Position und Größe mit Hilfslinien unterstützt. In das **TextBox**-Steuerelement sollen die Benutzer unseres Programms den zu konvertierenden DM-Betrag eingeben:



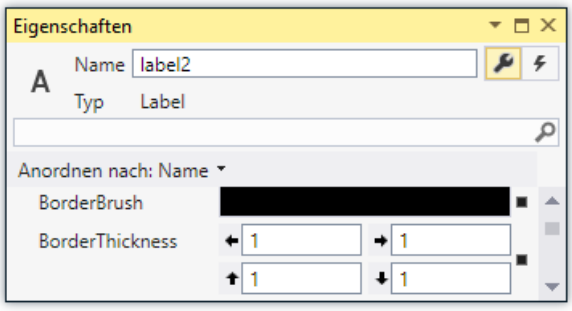
Der initial vorhandene Text stört und sollte gelöscht werden (Eigenschaft **Text**).

Setzen Sie zur Ausgabe des Euro-Betrags zwei weitere **Label**-Objekte auf das Fenster:



Während das erste Label wie das DM-Pendant zur Beschriftung dient, soll das zweite den Ergebnisbetrag anzeigen, also beim Programmstart leer sein. Passen Sie also die **Content**-Eigenschaftsausprägungen der Objekte passend an.

Um dem zweiten Label auch initial einen optischen Auftritt zu verschaffen, sollten Sie einen Rand anzeigen lassen. Dazu ist über die Eigenschaft **BorderBrush** eine Randfarbe und über die Eigenschaft **BorderThickness** eine Randstärke geeignet festzulegen, z.B.:

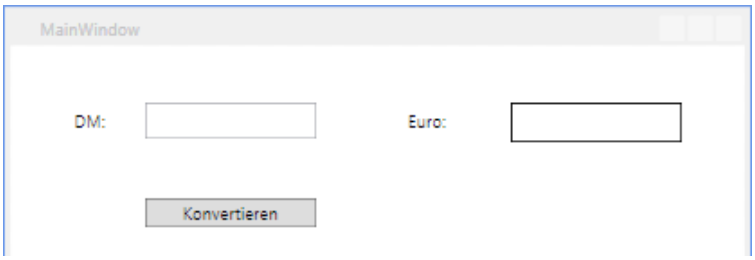


Um eine Eigenschaft zu lokalisieren, können Sie ....

- die zugehörige Kategorie raten und aufklappen,

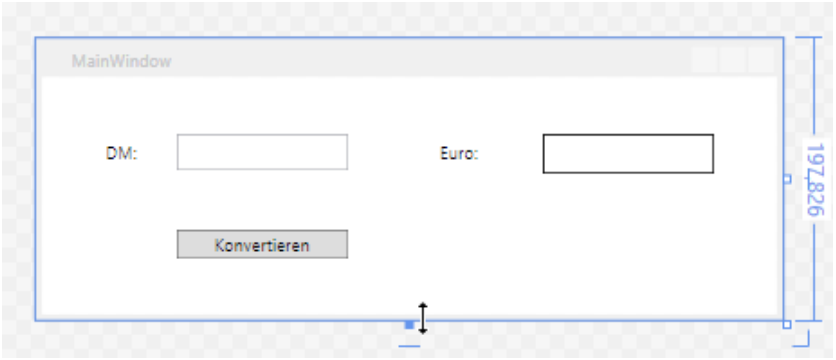
- eine **Anordnung** der Eigenschaften nach dem **Namen** veranlassen,
- nach der Eigenschaft suchen.

Setzen Sie nun noch ein **Button**-Objekt auf das Fenster, damit die Benutzer per Mausklick die Konvertierung des zuvor angeforderten DM-Betrags anfordern können:



Auch beim **Button**-Objekt sorgt man über die **Content**-Eigenschaft für eine passende Beschriftung.

Mittlerweile hat sich gezeigt, dass die vorgegebene Fensterfläche in vertikaler Richtung überdimensioniert ist. Packen Sie mit der Maus den unteren Fensterrand, und wählen Sie eine neue Höhe:

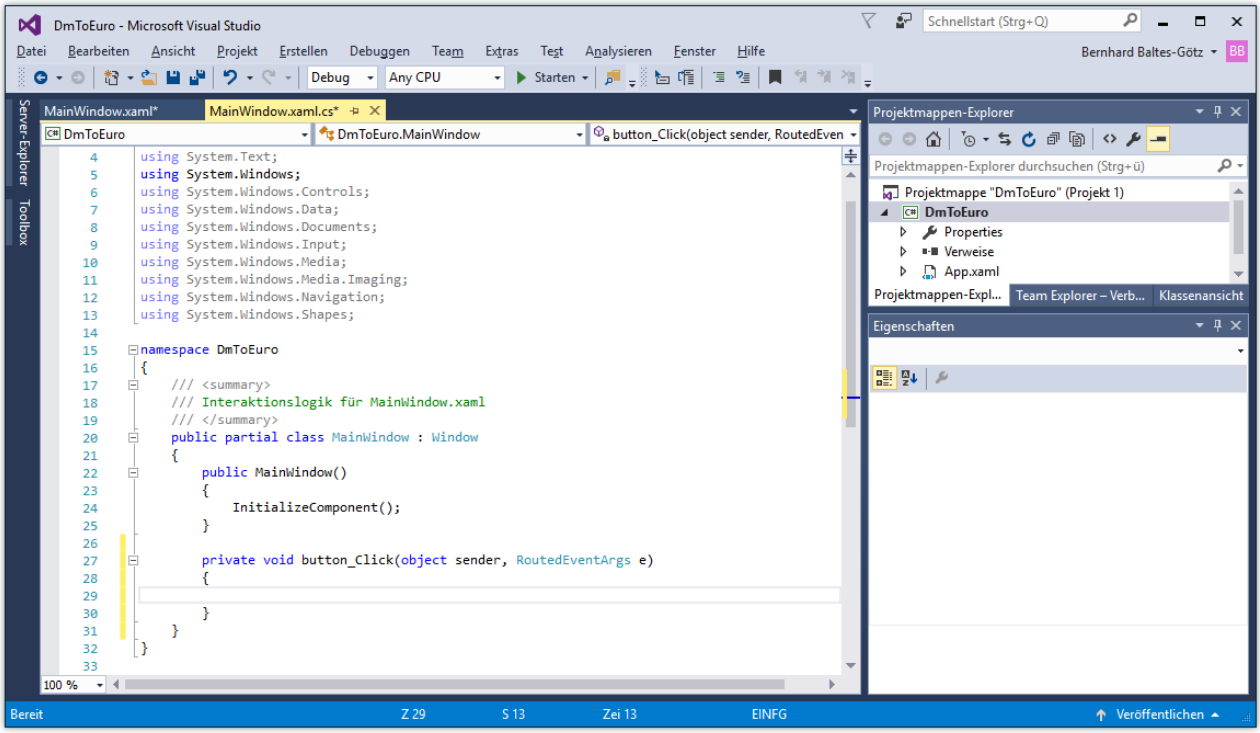


2.2.4.3 Behandlungsmethode zum Click-Ereignis des Befehlsschalters erstellen

Nun ist die Bedienoberfläche des entstehenden Programms in einem akzeptablen Zustand, und wir können uns um die Funktionalität kümmern. Dazu ist eine Methode zu erstellen, die bei einem Mausklick auf den Befehlsschalter ausgeführt werden soll. Sie hat folgende Aufgaben:

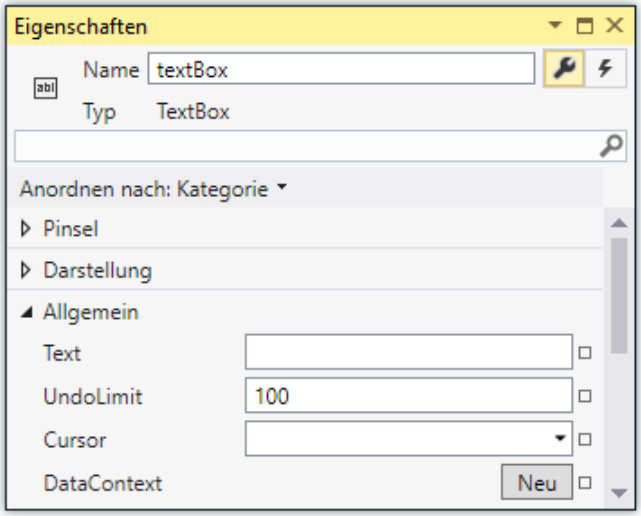
- beim **TextBox**-Objekt die aktuell eingetragene Zeichenfolge erfragen
- diese Zeichenfolge nach Möglichkeit in eine Zahl wandeln
- das Ergebnis durch den DM-Euro - Umrechnungsfaktor 1,95583 dividieren
- den Euro-Betrag in eine Zeichenfolge wandeln und das umrahmte **Label**-Objekt auffordern, diese Zeichenfolge anzuzeigen.

Sobald Sie einen Doppelklick auf das **Button**-Objekt setzen, öffnet das Visual Studio den Quellcode-Editor und fügt dort eine Methode namens `button_Click()` ein, die im fertigen Programm nach jedem Mausklick auf den Schalter ausgeführt wird:

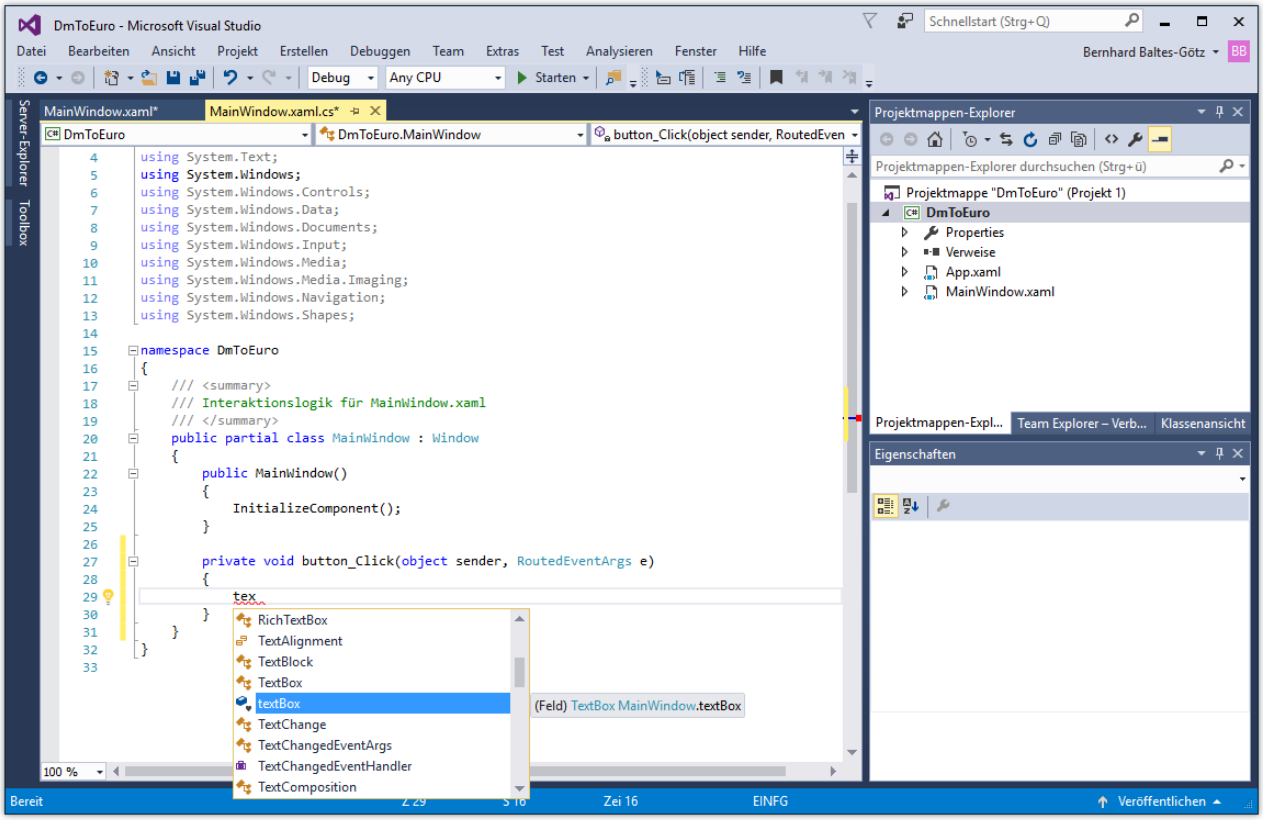


Nun wird auch erkennbar, dass wir gerade dabei sind, mit Assistentenhilfen eine Klasse namens `MainWindow` zu definieren. Für das Projekt hat die Entwicklungsumgebung den Namensraum `DmToEuro` definiert, wobei die Bezeichnung mit dem Projektnamen übereinstimmt.

Das **TextBox**-Memberobjekt der Klasse `MainWindow` hat beim Fenster-Design automatisch den Namen `textBox` erhalten, den Sie per **Eigenschaften**-Fenster ändern könnten:

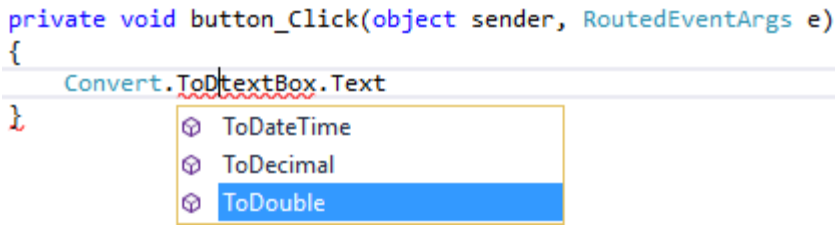


Sobald Sie damit beginnen, im Rumpf der Methode `button_Click` den Namen des **TextBox**-Objekts einzutippen, um sich über seine **Text**-Eigenschaft nach der vom Benutzer eingetippten Zeichenfolge zu erkundigen, erahnt das Visual Studio Ihre Absicht und bietet mögliche Fortsetzungen Ihrer Anweisung an:



Akzeptieren Sie den Vorschlag `textBox` der sogenannten *IntelliSense*-Technik per Tabulatortaste, und setzen Sie einen Punkt hinter den Objektnamen. Nun erscheint eine Liste mit den Methoden und Eigenschaften des Objekts. Das **TextBox**-Objekt soll seine **Text**-Eigenschaft abliefern. Wählen Sie diese Eigenschaft aus der Liste (z.B. per Doppelklick).

Wir verwenden die erfragte Zeichenfolge als Parameter (Argument) in einem Aufruf der Methode **ToDouble()**, welche die Klasse **Convert** beherrscht. Bei der erforderlichen Erweiterung der gerade entstehenden Anweisung C# - Anweisung bewährt sich wiederum die IntelliSense-Technik unserer Entwicklungsumgebung:



Nach einem Doppelklick auf **ToDouble** müssen wir lediglich die runden Klammern um den Parameter ergänzen, um den Methodenaufruf zu komplettieren.

Der Quellcode-Editor unserer Entwicklungsumgebung bietet außerdem …

- farbliche Unterscheidung verschiedener Sprachestandteile
- automatische Quellcode-Formatierung (z.B. bei Einrückungen)

- automatische Syntaxprüfung, z.B.:

```
private void button_Click(object sender, RoutedEventArgs e)
{
    Convert.ToDouble(textBox.Text)
}
```

  Wir haben mittlerweile einen Methodenaufruf erstellt, der aber keine vollständige Anweisung ist, was unsere Entwicklungsumgebung durch rotes Unterschlängeln der mutmaßlichen Fehlerstelle reklamiert. Wir ergänzen das an dieser Stelle erforderliche Semikolon.

Um bei den weiteren Verarbeitungsschritten einen überlangen Ausdruck zu vermeiden, benutzen wir zur lokalen Zwischenspeicherung des DM-Betrags in der Methode `button_Click()` eine lokale Variable namens `betrag` vom Typ **double** (siehe Abschnitt 3.3.4):

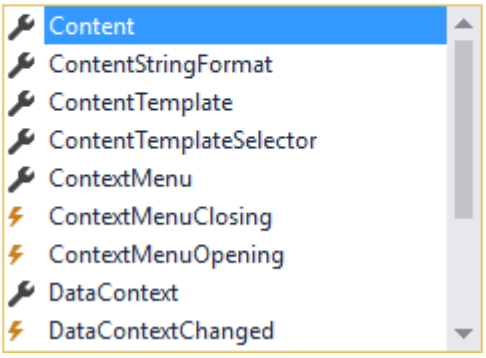
```
private void button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(textBox.Text);
}
```

Die von **ToDouble()** als Rückgabe gelieferte und mittlerweile in der Variablen `betrag` gespeicherte Zahl muss durch 1,95583 dividiert werden. Den resultierenden Euro-Betrag lassen wir durch die Methode **ToSring()** der Klasse **Convert** in eine Zeichenfolge (ein Objekt der Klasse **String**) wandeln:

```
private void button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(textBox.Text);
    Convert.ToString(betrag / 1.95583);
}
```

Das Endergebnis muss dem **Label**-Objekt `label2` als neuer Wert der Eigenschaft **Content** übergegeben werden:

```
private void button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(textBox.Text);
    label2.ConContConvert.ToString(betrag / 1.95583);
}
```

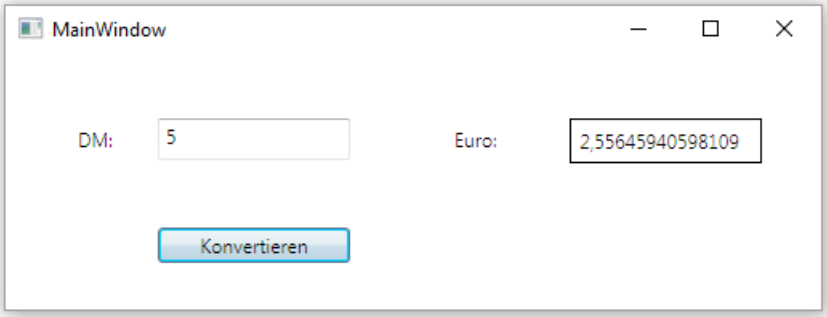


So sieht die fertige Methode `button_Click` aus:

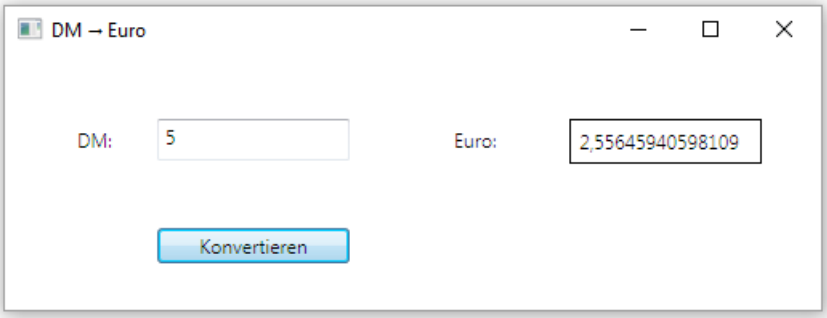
```
private void button_Click(object sender, RoutedEventArgs e)
{
    double betrag = Convert.ToDouble(textBox.Text);
    label2.Content = Convert.ToString(betrag / 1.95583);
}
```

2.2.4.4 Testen und verbessern

Wir lassen das Programm über die Tastenkombination **Strg+F5** übersetzen und ausführen:

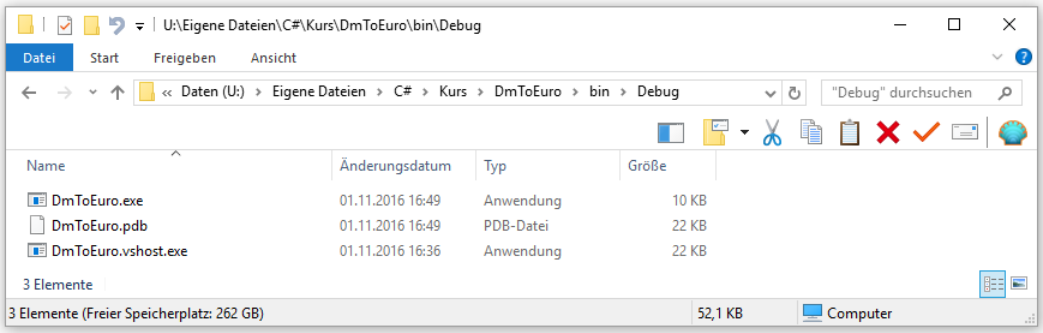


Spontan fällt als einziger Mangel der wenig informative Fenstertitel auf. Markieren Sie bei aktivem WPF-Editor das Hauptfenster und ändern Sie per Eigenschaftenfenster seine **Title**-Eigenschaft:[1]

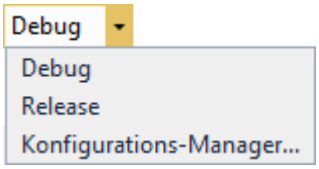


In der Endversion werden wir einen benutzerfreundlich gerundeten Euro-Betrag präsentieren.

Ist die **Debug**-Konfiguration aktiv, erstellt der Compiler ein gut testbares, aber nicht optimiertes Assembly und legt es Projekt-Unterordner **…\bin\Debug** ab:



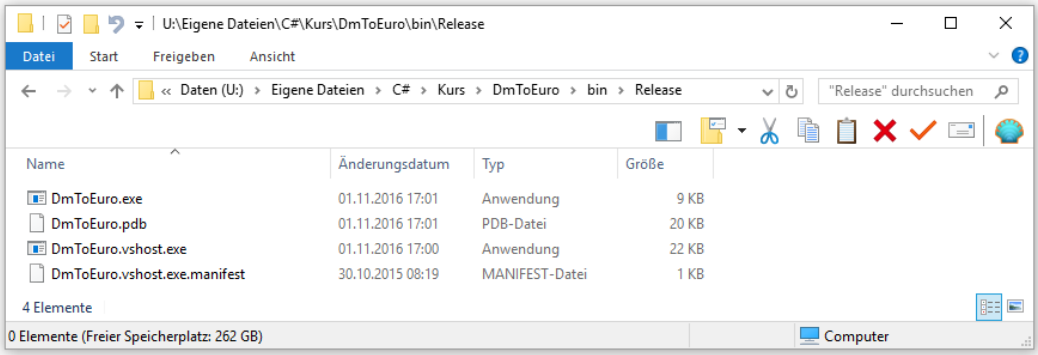
Zur Änderung der Konfiguration steht in der Symbolleiste **Standard** ein Bedienelement bereit:



Ist die **Release**-Konfiguration aktiv, erstellt der Compiler ein optimiertes, aber weniger gut testbares Assembly und legt es im Projekt-Unterordner **…\bin\Release** ab:

---

[1] Falls jemand wissen möchte, wie man den rechtsgerichteten Pfeil eintippt: Tastenkombination **Alt+26** auf dem Ziffernblock der Tastatur.



In jedem Fall ist das Assembly **DmToEuro.exe** einsatzfähig und kann auf jedem Windows-Rechner mit .NET - Framework (ab Version 3.5) ausgeführt werden. Weil sich die benötigten Bibliotheks-Assemblies im GAC (Global Assembly Cache) befinden und keine Hilfsdateien erforderlich sind, muss zur „Installation“ des Programms lediglich die EXE-Datei an ihren Einsatzort transportiert werden.

Soll eine Anwendung nur erstellt (aber nicht ausgeführt) werden, wählt man die Tastenkombination **Strg-Umschalt+B** oder den Menübefehl

Erstellen > Projektmappe erstellen

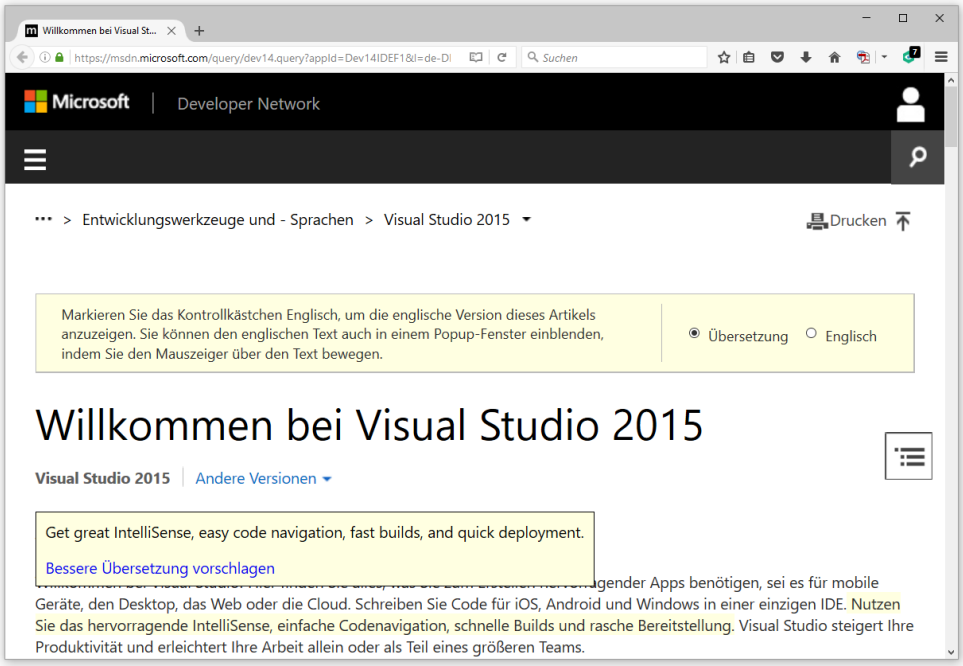
Trotz der guten Erfahrungen mit der GUI-Programmierung werden wir uns die Grundbegriffe der Programmierung im Rahmen von möglichst einfachen Konsolenprojekten erarbeiten.

2.2.5 FCL-Dokumentation und andere Hilfeinhalte

Die über den Menübefehl

Hilfe > Hilfe anzeigen

angezeigten Hilfeinhalte werden per Voreinstellung aus dem Internet bezogen und von einem Browser angezeigt, z.B.:



Wer lokal abgelegt Hilfeinformationen verwenden und mit dem **Microsoft Help Viewer** inspizieren möchte, kann diesen Wunsch über den folgenden Menübefehl vortragen:

Hilfe > Hilfeeinstellungen festlegen > In Help Viewer starten

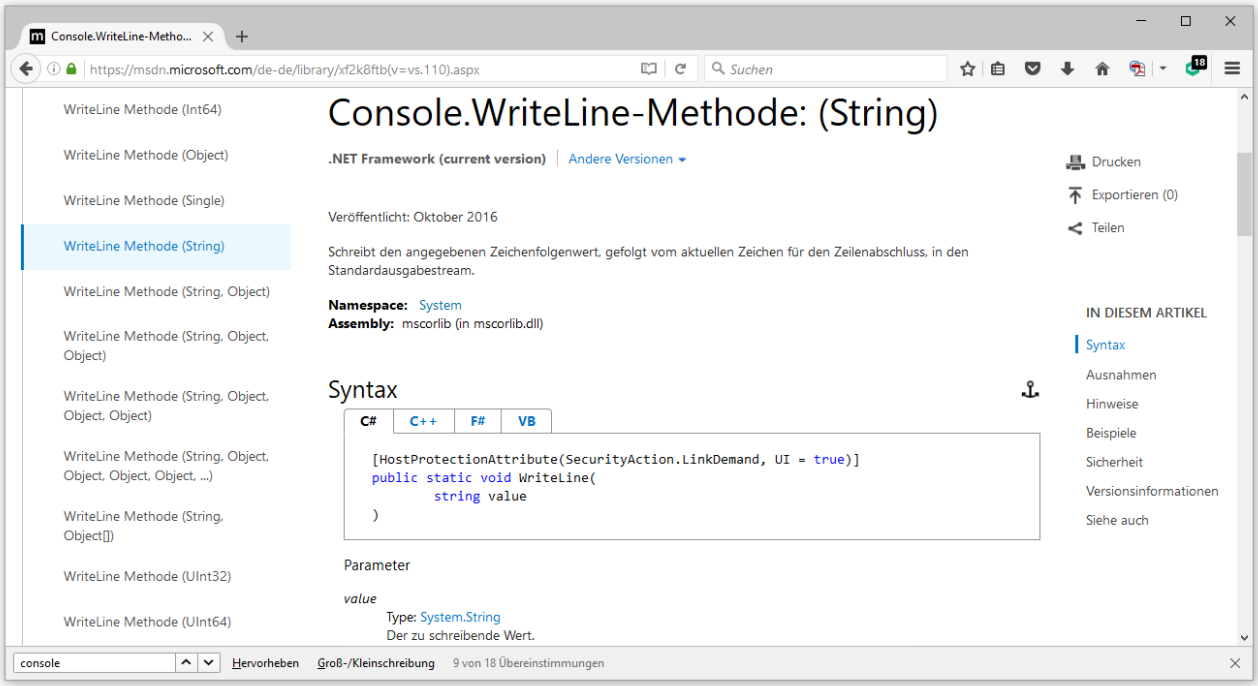
Wir benötigen die Hilfefunktion vor allem zur Präsentation der FCL-Dokumentation. Wer sich z.B. über die in unseren Beispielen oft benutzte Methode **WriteLine()** der Klasse **Console** aus dem Namensraum **System** informieren möchte, kann in der Navigationszone (am oberen Fensterrand) den Einstieg

MSDN Library > Entwicklungswerkzeuge und Sprachen > .NET - Entwicklung > .NET Framework 4.6 > .NET Framework-Klassenbibliothek

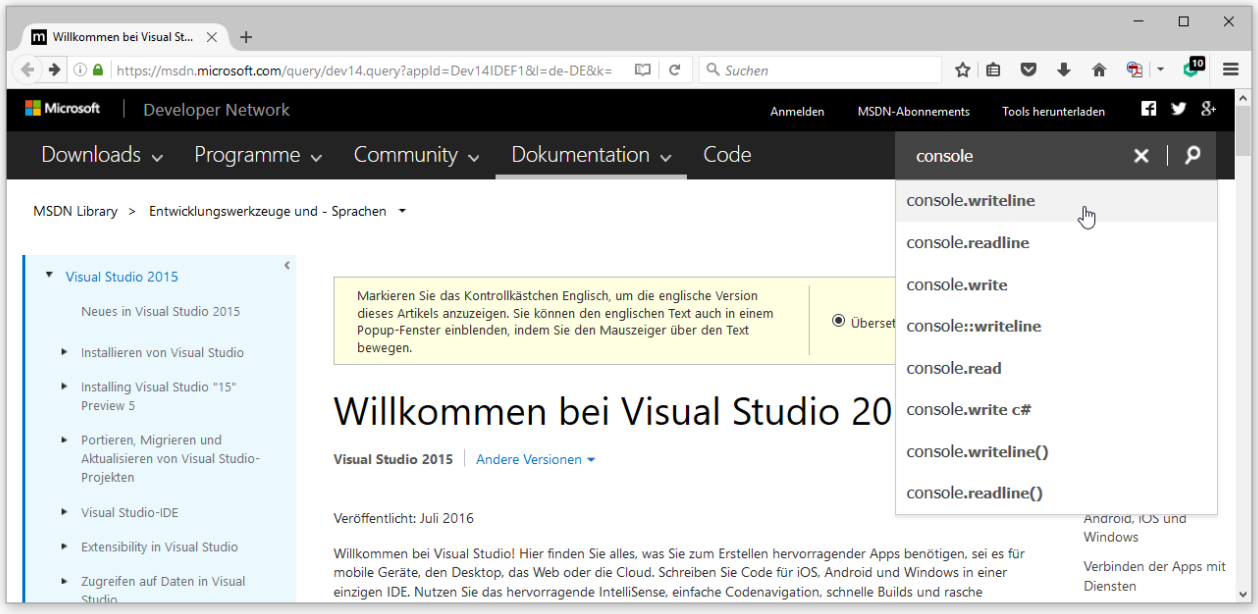
wählen und sich dann (etwas mühsam) mit den folgenden Stationen weiter bewegen:

System-Namespace > Console Klasse > Console Methoden > WriteLine Methode

Jetzt ist noch zwischen vielen Spezialisierungen (Überladungen, siehe unten) der Methode **WriteLine()** zu wählen, damit im Inhaltsbereich passende Informationen angezeigt werden:



Meist führt die Suchfunktion des Hilfefensters schneller zum Ziel, z.B.:



Zu einem markierten (die Einfügemarke enthaltenden) C# - Schlüsselwort oder FCL-Bezeichner im Quellcode-Editor der Entwicklungsumgebungen erreicht man die zugehörige Dokumentation besonders bequem über die Funktionstaste **F1**.

Um die gut gemeinten, aber auf Dauer störenden PopUp-Fenster mit Übersetzungen abzustellen, kann man so vorgehen:

- Ans Ende des Browser-Fensters rollen und auf das Bedienelement mit der automatisch erkannten Sprache klicken:

- Anschließend ein Land mit englischer Standardsprache wählen (z.B. United States)

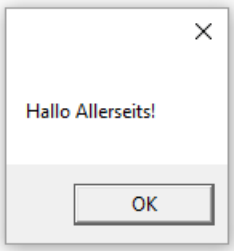
2.2.6 Compiler-Optionen in der Entwicklungsumgebung setzen

In Abschnitt 2.1.2 zur Übersetzung von Quellcode in die Microsoft Intermediate Language (MSIL) wurden wichtige Optionen des Compiler-Aufrufs behandelt, u.a.:

- Ausgabetyp des resultierenden Assemblies
- Referenzen auf Assemblies, die der Compiler nach Klassen durchsuchen soll
- Zielplattform (MSIL, x86, x64, Itanium)

Beim Einsatz einer Entwicklungsumgebung werden die Compileraufrufe automatisch erstellt und im Hintergrund abgesetzt. Die gewünschten Compiler-Optionen wählt man in bequemen Dialogfenstern.

Um dies üben zu können, erstellen wir nun mit dem Visual Studio ein Hallo-Projekt gemäß Abschnitt 2.2.3, ersetzen aber die Textausgabe durch eine Messagebox:



Während die generelle GUI-Programmierung relativ anspruchsvoll ist, gelingt die Präsentation einer Messagebox mit Leichtigkeit. Es ist lediglich ein Aufruf der statischen Methode **Show()** der Klasse **MessageBox** an Stelle des **Console.WriteLine()** - Aufrufs erforderlich, z.B.:

```
MessageBox.Show("Hallo Allerseits!");
```

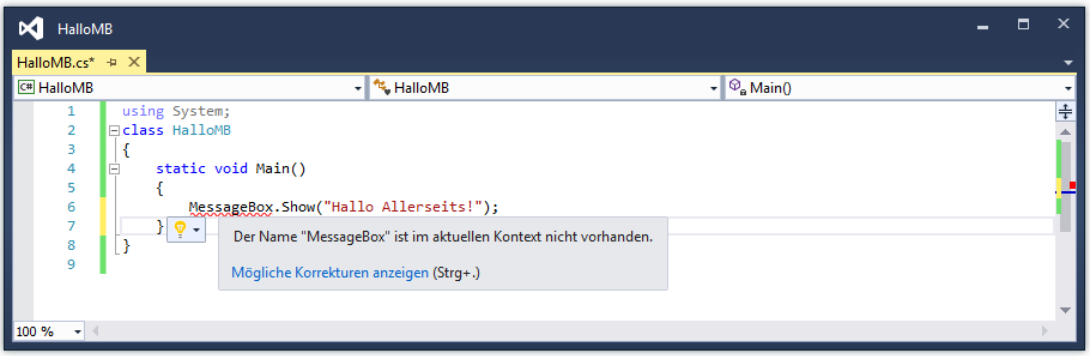
Außerdem kann der Methodenaufruf

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

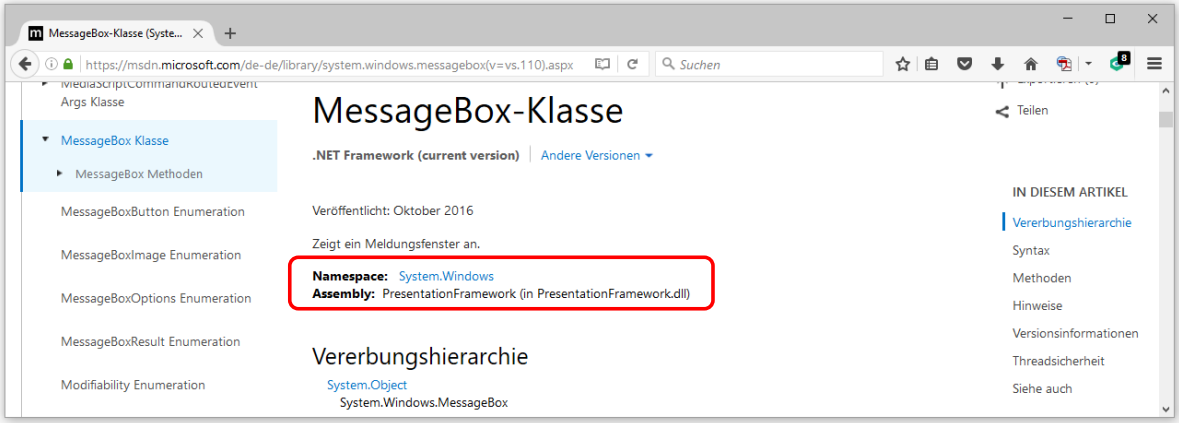
entfallen, weil kein Konsolenfenster offen gehalten werden muss (vgl. Abschnitt 2.2.6.2).

2.2.6.1 Referenzen

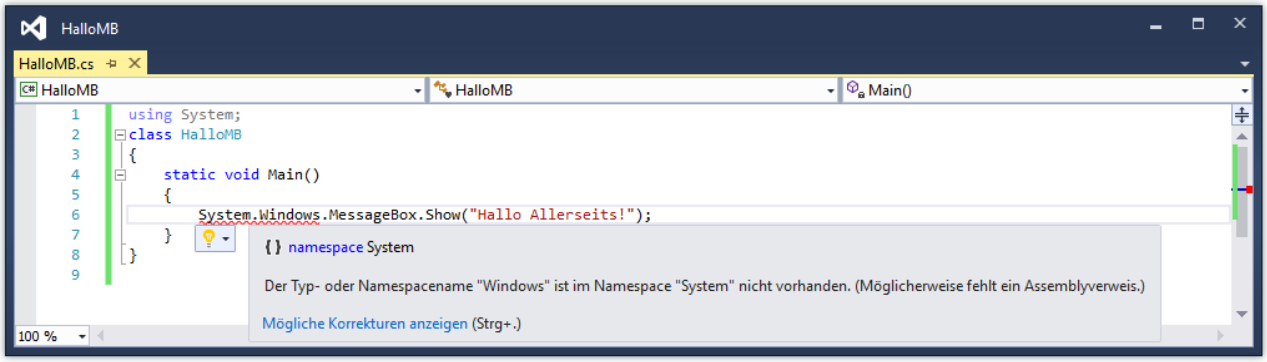
Ohne weitere Ergänzungen des Quellcodes kann die Übersetzung allerdings nicht gelingen, wie die Entwicklungsumgebung vorausblickend mit genauer Fehlerdiagnose mitteilt:



Mit Hilfe der FCL-Dokumentation lässt sich ermitteln, welcher Namensraum anzugeben ist (als Präfix zum Klassennamen oder in einer **using**-Direktive), um dem C# - Compiler die Klasse **MessageBox** bekannt zu machen. Ganz einfach ist die folgende Information allerdings nicht zu erreichen, weil die FCL-Bibliothek *zwei* Klassen mit dem Namen **MessageBox** kennt. Wir entscheiden uns für die Variante aus dem Assembly **PresentationFramework.dll**:



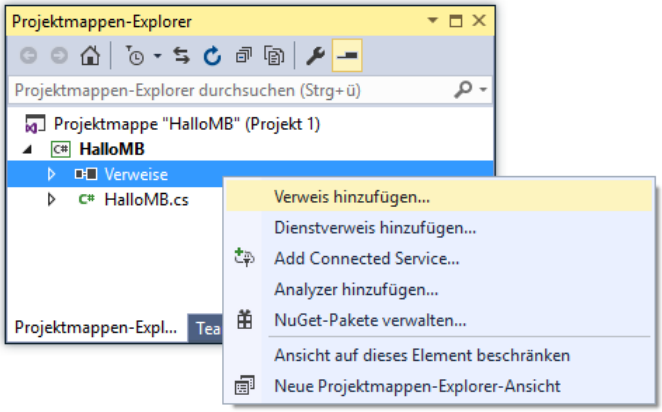
Mit dem folgenden funktionstüchtigen (!) Quellcode wird das Problem scheinbar nicht gelöst, sondern nur verlagert:



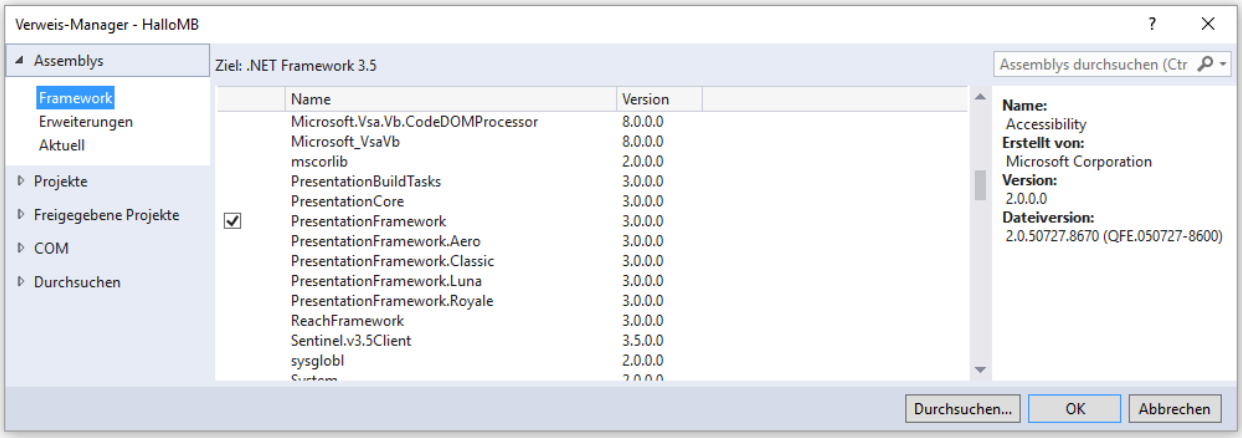
Das Visual Studio liefert aber mit dem Hinweis auf einen eventuell fehlenden Assembly-Verweis praktisch die Lösung des Problems: Weil sich das die Klasse **MessageBox** implementierende Assembly **PresentationFramework.dll** nicht in der beim Übersetzen zu berücksichtigenden Referenzenliste des Projekts befindet, wird die Klasse nicht gefunden. Folglich benötigt unser Projekt eine Referenz auf das Assembly **PresentationFramework.dll**.[1] Wählen Sie dazu im **Projektmappen-**

[1] Bei Verwendung der Projektvorlage **WPF-Anwendung** legt die Entwicklungsumgebung automatisch eine Referenz auf das Assembly **PresentationFramework.dll** an, nicht jedoch bei der Vorlage **Leeres Projekt**, die wir aus Gründen der Einfachheit und Transparenz derzeit bevorzugen.

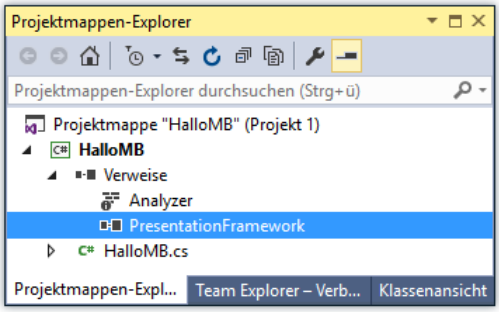
Explorer aus dem Kontextmenü zum Projekt-Knoten **Verweise** die Option **Verweis hinzufügen**:



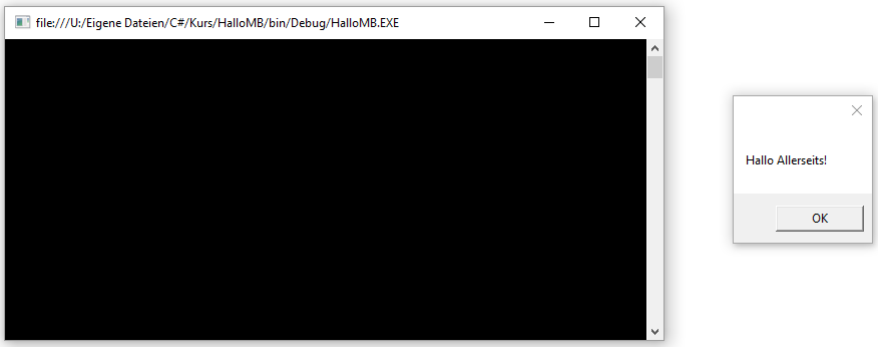
Nun kann in folgender Dialogbox das zu durchsuchende Assembly lokalisiert und im markierten Zustand mit **OK** in die Verweisliste aufgenommen werden:



Anschließend taucht der Verweis im Projektmappen-Explorer auf,



und die Reklamation im Quellcode-Editor verschwindet. Wenn Sie das Programm starten, erscheint die gewünschte Messagebox:

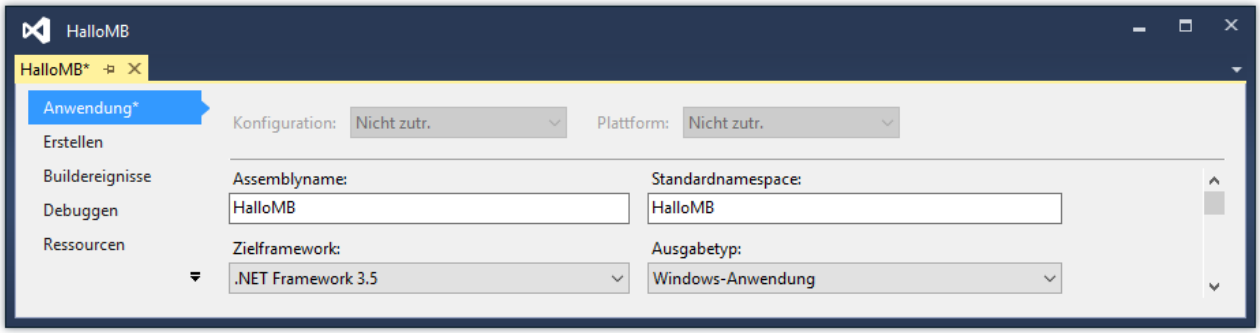


2.2.6.2 Ausgabetyp

Außerdem erscheint aber auch ein leeres Konsolenfenster. Um seinen Auftritt zu verhindern, ersetzt man per Compiler-Option den voreingestellten Ausgabetyp **exe** durch die Alternative **winexe** (vgl. Abschnitt 2.1.2). Das kann im Visual Studio nach

Projekt > Eigenschaften > Anwendung

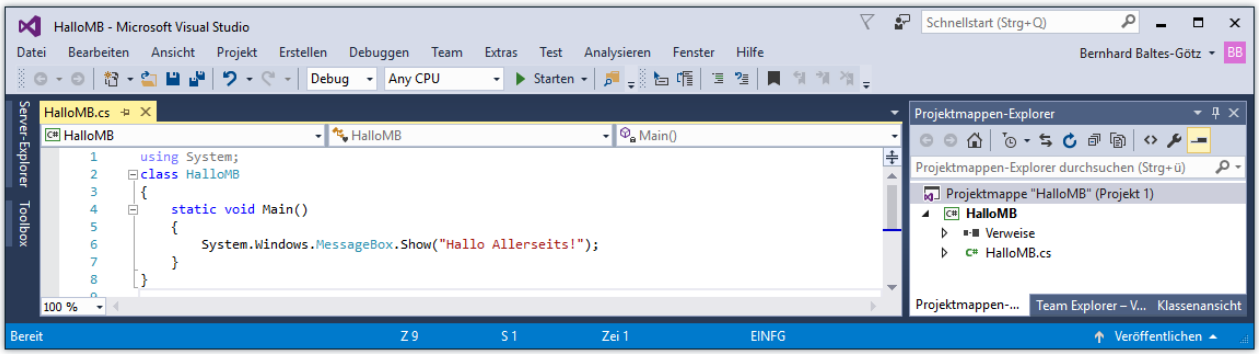
in folgendem Dialogfenster geschehen:



Bei einem Programm mit dem Ausgabetyp **winexe** (**Windows-Anwendung**) gehen alle Konsolenausgaben verloren, so dass man diesen Ausgabetyp mit Bedacht wählen sollte.

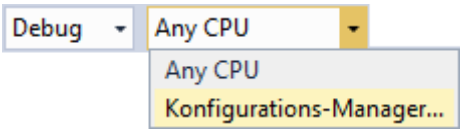
2.2.6.3 Zielplattform

In Visual Studio 2015 ist (zumindest bei einer Windows-Version mit 64-Bit-Architektur) für den C# - Compiler die Zielplattform **Any CPU** voreingestellt, z.B.:



Dies ist konsistent mit der Beobachtung aus Abschnitt 1.2.5.5, dass der C# - Kommandozeilen - Compiler die voreingestellte Zielplattform **MSIL** verwendet. Der Compiler produziert daraufhin ein **Portable Executable 32 .NET Assembly**, das unter Windows 64 in einem 64-Bit - Prozess ausgeführt wird und selbstverständlich auch unter Windows 32 läuft.

Obwohl es nur selten erforderlich ist, eine alternative Zielplattform einzustellen (siehe Diskussion in Abschnitt 1.2.5.5), werden die erforderlichen Schritte anschließend beschrieben. Soll für eine Projektmappe eine alternative Zielplattform wählbar sein, muss man den Konfigurations-Manager bemühen, was über die Plattformliste in der Symbolleiste **Standard**

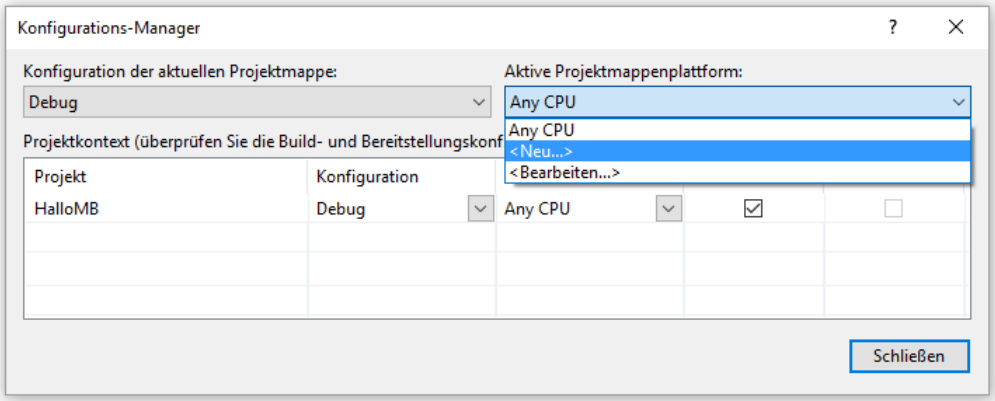


oder den Menübefehl

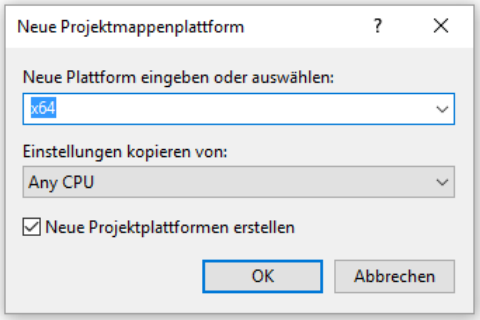
Erstellen > Konfigurations-Manager

möglich ist. Nun kann man z.B. so vorgehen:

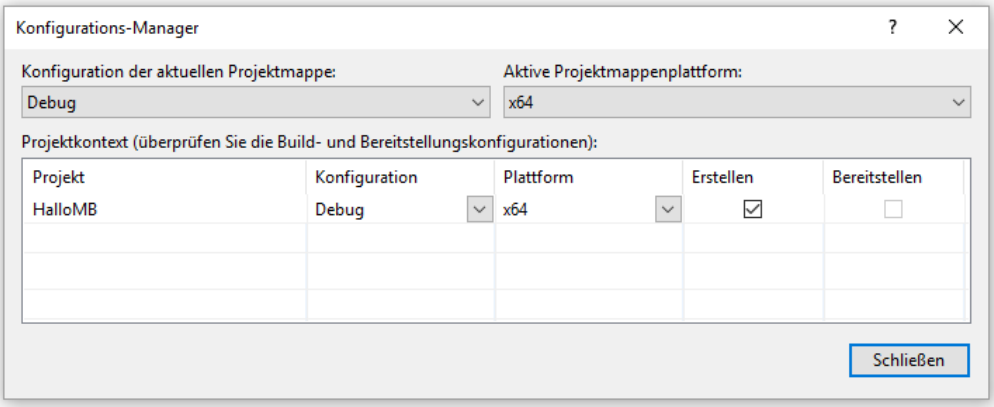
- Eine neue **Projektmappenplattform** anlegen:



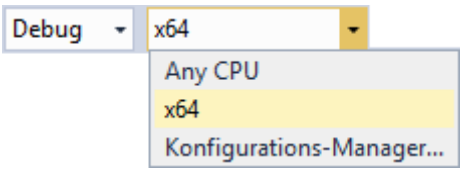
- Im folgenden Dialog die gewünschte Plattform wählen, z.B.:



- Die Markierung beim Kontrollkästchen **Neue Projektplattformen erstellen** belassen, so dass nach dem Quittieren mit **OK** bei allen Projekten in der Mappe die neue Zielplattform voreingestellt ist:



Anschließend kann ein Assembly mit der gewünschten Zielplattform erstellt werden



Bei den Beispielprojekten im Kurs werden wir wohl kaum einen Grund finden, die vom Visual Studio vorgeschlagene Zielplattform **Any CPU** zu ändern.

2.3 Übungsaufgaben zu Kapitel 2

1) Installieren Sie nach Möglichkeit auf Ihrem privaten PC das Visual Studio 2015 Community (gemäß Abschnitt 2.2.1).

2) Experimentieren Sie mit dem `Hallo`-Beispielprogramm in Abschnitt 2.1.1:

- Ergänzen Sie weitere Ausgabeanweisungen.
- Erstellen Sie eine Variante ohne **using**-Direktive (vgl. Abschnitt 1.2.7).

3) Beseitigen Sie die Fehler in folgender Variante des `Hallo`-Programms:

```
Using System;
class Hallo {
  static void Main() {
    Console.WriteLn("Hallo, echt .NET hier!);
  }
```

3 Elementare Sprachelemente

In Kapitel 1 wurde anhand eines halbwegs realistischen Beispiels versucht, einen ersten Eindruck von der objektorientierten Softwareentwicklung mit C# zu vermitteln. Nun erarbeiten wir uns die Details der Programmiersprache C# und beginnen dabei mit elementaren Sprachelementen. Diese dienen zur Realisation von Algorithmen innerhalb von Methoden und sehen bei C# nicht wesentlich anders aus als bei älteren, *nicht* objektorientierten Sprachen (z.B. C).

3.1 Einstieg

3.1.1 Aufbau von einfachen C# - Programmen

Sie haben schon einiges über den Aufbau von C# - Programmen erfahren:

- Ein C# - Programm besteht aus **Klassen**. Unser Bruchadditionsbeispiel in Abschnitt 1.1 besteht aus den beiden Klassen `Bruch` und `Bruchaddition`.
- In einer **Klassendefinition** werden **Felder** deklariert sowie **Eigenschaften** und **Methoden** definiert. Meist verwendet man für den Quellcode einer Klasse jeweils eine eigene Datei mit demselben Namen wie die Klasse und **.cs** als Namenserweiterung.
- Von den Klassen eines Programms muss eine **startfähig** sein. Dazu benötigt sie eine Methode mit dem Namen **Main()** und folgenden Besonderheiten:
  - Modifikator **static**
  - Rückgabetyp **int** oder **void**

  Diese Methode wird beim Programmstart vom Laufzeitsystem (von der CLR) aufgerufen. Wenn die **Main()** - Methode ihrem Aufrufer (also der CLR bzw. dem Betriebssystem) per **return**-Anweisung (siehe Abschnitt 4.3.1.2) eine ganze Zahl als Information über den (Miss)erfolg ihrer Tätigkeit liefern möchte (z.B. 0: alles gut gegangen, 1: Fehler), dann ist in der Methodendefinition der Rückgabetyp **int** anzugeben.[1] Fehlt eine solche Rückgabe, ist der Pseudorückgabetyp **void** anzugeben. Beim Bruchadditions-Beispiel in Abschnitt 1.1 ist die Klasse `Bruchaddition` startfähig. Ihre **Main()** - Methode verwendet den Pseudorückgabetyp **void**.
- Die zu einem Programm gehörigen Quellcodedateien werden gemeinsam vom **Compiler** in die Common Intermediate Language (CIL) übersetzt, z.B.:

  ```
  csc  Bruch.cs  Bruchaddition.cs
  ```

  Das resultierende Assembly **Bruchaddition.exe** übernimmt seinen Namen von der Startklasse und enthält neben dem **CIL-Code** auch Typ- und Assembly-**Metadaten**.
- Die ersten Zeilen einer C# - Quellcodedatei enthalten meist **using**-Direktiven zum Import von **Namensräumen**, damit die dortigen Klassen später ohne Namensraumpräfix vor den Klassennamen angesprochen werden können.
- Die Definition einer Klasse, Eigenschaft oder Methode besteht aus:
  - **Kopf**
    Bei einer Klassendefinition folgt auf das Schlüsselwort **class** der relativ frei wählbare Klassenname.

[1] Nach Beendigung des Programms findet sich der **return**-Code in der Windows-Umgebungsvariablen **errorlevel**.

  - **Rumpf**
    Im Rumpf einer Klassendefinition werden Felder deklariert sowie Eigenschaften und Methoden definiert. Im Rumpf einer Methode finden sich die Anweisungen zur Bewältigung des Auftrags.

  Details folgen gleich in Abschnitt 3.1.2.

- Eine **Anweisung** ist die kleinste ausführbare Einheit eines Programms. In C# sind bis auf wenige Ausnahmen alle Anweisungen mit einem **Semikolon** abzuschließen.

Während der Beschäftigung mit elementaren C# - Sprachelementen werden wir mit einer extrem einfachen und nicht sonderlich objektorientierten Programmstruktur arbeiten, die Sie schon aus dem `Hallo`-Beispiel kennen. Es wird nur *eine* Klasse definiert, und diese erhält nur eine einzige Methodendefinition. Weil die Klasse startfähig sein muss, liegt **Main** als Name der Methode fest, und wir erhalten die folgende Programmstruktur:

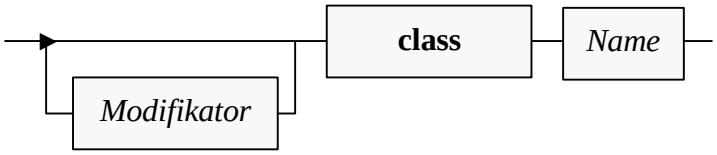
```
using System;
class Prog {
   static void Main() {
      // Platz zum Üben elementarer Sprachelemente
   }
}
```

Damit die wenig objektorientierten Beispiele Ihrem Programmierstil nicht prägen, wurde zu Beginn des Kurses (in Abschnitt 1.1) eine Anwendung vorgestellt, die bereits etliche OOP-Prinzipien realisiert.

3.1.2 Syntaxdiagramm

Um für C# - Sprachbestandteile (z.B. Definitionen oder Anweisungen) die Bildungsvorschriften kompakt und genau zu beschreiben, werden wir im Kurs u.a. sogenannte **Syntaxdiagramme** einsetzen, für die folgende Vereinbarungen gelten:

- Man bewegt sich vorwärts in Pfeilrichtung durch das Syntaxdiagramm und gelangt dabei zu Rechtecken, welche die an der jeweiligen Stelle zulässigen Sprachbestandteile angeben.
  z.B.:



- Bei Abzweigungen kann man sich für eine Richtung entscheiden, wenn nicht durch Pfeilspitzen eine Bewegungsrichtung vorgeschrieben ist. Zulässige Realisationen zum obigen Segment sind also z.B.:
  - `class Bruchaddition`
  - `public class Bruch`

  Verboten sind hingegen z.B. folgende Formulierungen:
  - `class public Bruchaddition`
  - `Bruchaddition public class`
- Für **konstante** (**terminale**) Sprachbestandteile, die aus einem Rechteck exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet. Im konkreten Quellcode muss anstelle des Platzhalters eine zulässige Realisation stehen, und die zugehörigen Regeln sind an anderer Stelle (z.B. in einem anderen Syntaxdiagramm) erklärt.

- Bisher kennen Sie nur den Klassenmodifikator **public**, welcher die allgemeine Verfügbarkeit einer Klasse anordnet. Später werden Sie weitere Klassenmodifikatoren kennen lernen. Sicher kommt niemand auf die Idee, z.B. den Modifikator **public** mehrfach zu vergeben und damit gegen eine Syntaxregel zu verstoßen. Das obige (möglichst einfach gehaltene) Syntaxdiagrammsegment lässt diese offenbar sinnlose Praxis zu. Es bieten sich zwei Lösungen an:
  - Das Syntaxdiagramm mit einem gesteigerten Aufwand an Formalismus präzisieren.
  - Durch eine generelle Zusatzregel die Mehrfachverwendung eines Modifikators verbieten.

  Im Manuskript wird die zweite Lösung verwendet.

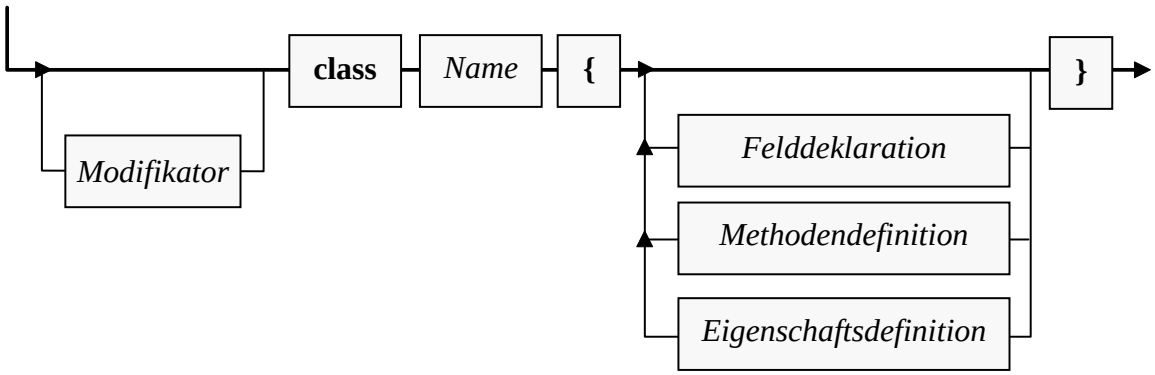
Als Beispiele betrachten wir anschließend die Syntaxdiagramme zur Definition von Klassen, Methoden- und Eigenschaften. Aus didaktischen Gründen zeigen die Diagramme nur solche Sprachbestandteile, die bisher in einem Beispiel verwendet oder im Text beschrieben wurden, so dass sie langfristig keinesfalls als Referenz taugen. Trotz der Vereinfachung sind die Syntaxdiagramme für die meisten Leser vermutlich nicht voll verständlich, weil etliche Bestandteile noch nicht systematisch beschrieben wurden (z.B. Modifikator, Feld- und Parameterdeklaration).

Im aktuellen Abschnitt 3.1.2 geht es primär darum, Syntaxdiagramme als metasprachliche Hilfsmittel einzuführen. Vielleicht tragen aber die vorgestellten Beispiele trotz der gerade angesprochenen Kompromisse auch zur allmählichen Festigung der wichtigen Begriffe *Klasse*, *Methode* und *Eigenschaft* bei. Auf keinen Fall handelt es sich bei den nächsten drei Abschnitten um die „offizielle“ Behandlung dieser Begriffe.

3.1.2.1 Klassendefinition

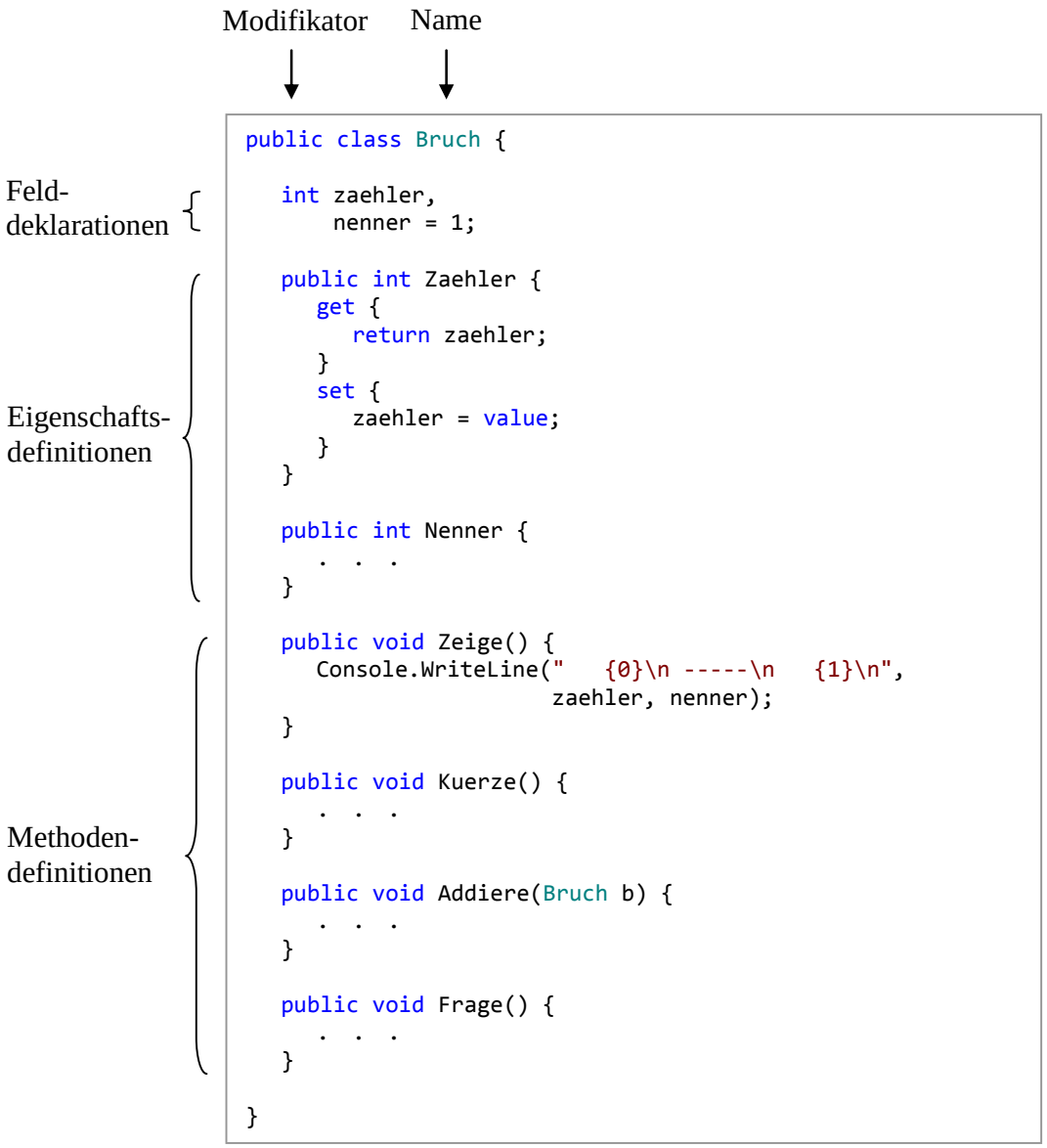
Wir arbeiten vorerst mit dem folgenden, leicht vereinfachten Klassenbegriff:

Klassendefinition



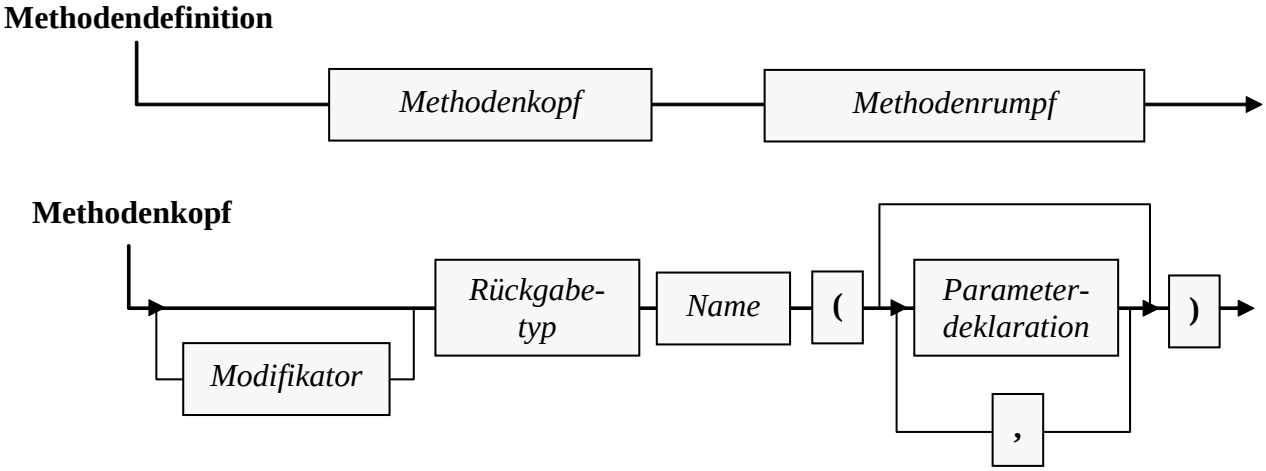
Solange man sich auf zulässigen Pfaden bewegt (immer in Pfeilrichtung, eventuell auch in Schleifen), an den Stationen (Rechtecken) entweder den konstanten Sprachbestandteil exakt übernimmt oder den Platzhalter auf zulässige (an anderer Stelle erläuterte) Weise ersetzt, sollte eine syntaktisch korrekte Klassendefinition entstehen.

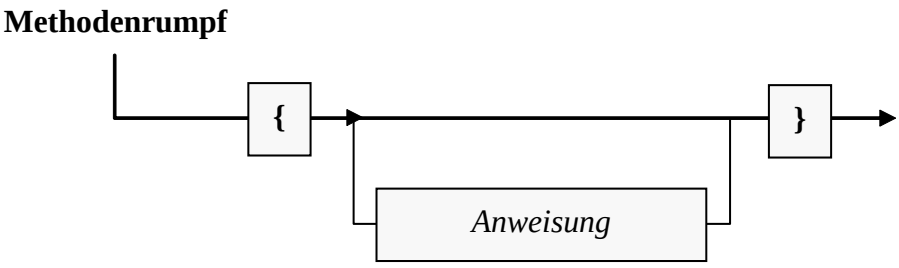
Als Beispiel betrachten wir die im Abschnitt 1.1 vorgestellte Klasse `Bruch`:



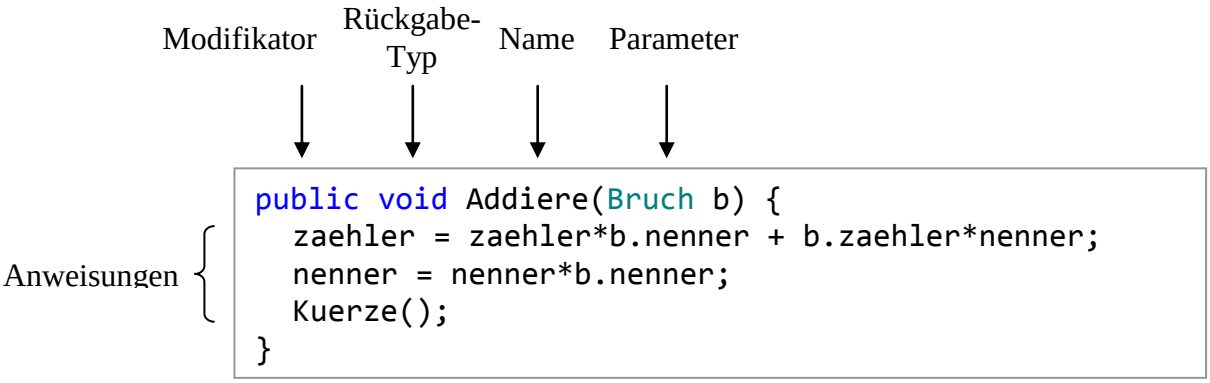
3.1.2.2 Methodendefinition

Weil *ein* Syntaxdiagramm für die komplette Methodendefinition etwas unübersichtlich wäre, betrachten wir separate Diagramme für die Begriffe *Methodenkopf* und *Methodenrumpf*:





Als Beispiel betrachten wir die Definition der `Bruch`-Methode `Addiere()`:



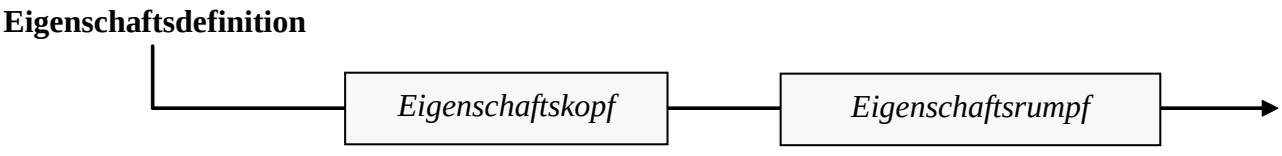
In vielen Methoden werden so genannte *lokale Variablen* (siehe unten) deklariert, z.B. in der `Bruch`-Methode `Kuerze()`:

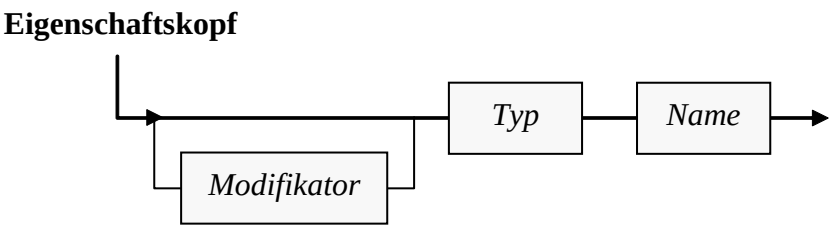
```
public void Kuerze() {
   if (zaehler != 0) {
      int ggt = 0;
      int az = Math.Abs(zaehler);
      int an = Math.Abs(nenner);
         . . .
   }
}
```

Weil wir bald u.a. von einer Variablendeklarations*anweisung* sprechen werden, benötigt das Syntaxdiagramm zum Methodenrumpf jedoch (im Unterschied zum Klassendefinitionsdiagramm) *kein* separates Rechteck für die Variablendeklaration.

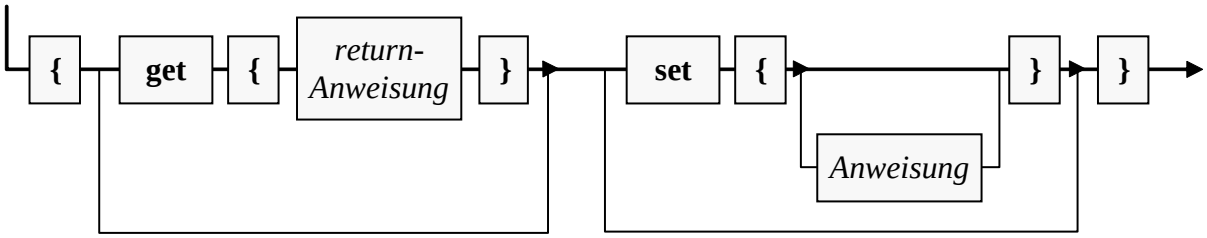
3.1.2.3 Eigenschaftsdefinition

Auch beim Syntaxdiagramm für den Eigenschaftsbegriff gehen wir schrittweise vor:

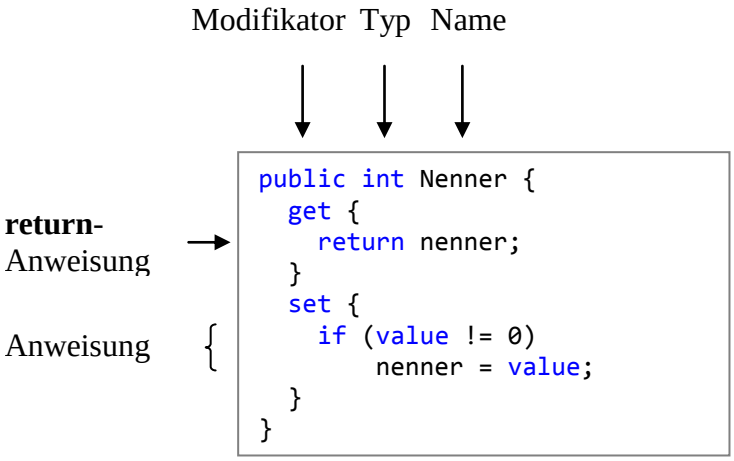




Eigenschaftsrumpf



Als Beispiel betrachten wir die `Bruch`-Eigenschaft `Nenner`:



3.1.3 Hinweise zur Gestaltung des Quellcodes

Zur Formatierung von C# - Programmen haben sich Konventionen entwickelt, die wir bei passender Gelegenheit besprechen werden. Der Compiler ist hinsichtlich der Formatierung sehr tolerant und beschränkt sich auf folgende Regeln:

- Die einzelnen Bestandteile einer Definition oder Anweisung müssen in der richtigen Reihenfolge stehen.
- Zwischen zwei Sprachbestandteilen muss ein **Trennzeichen** stehen, wobei das Leerzeichen, das Tabulatorzeichen und der Zeilenumbruch erlaubt sind. Diese Trennzeichen dürfen sogar in beliebigen Anzahlen und Kombinationen auftreten. *Innerhalb* eines Sprachbestandteils (z.B. Namens) sind Trennzeichen (z.B. Zeilenumbruch) natürlich verboten.
- Zeichen mit festgelegter Bedeutung wie z.B. ";", "(", "+", ">" sind *selbstisolierend*, d.h. vor und nach ihnen sind keine Trennzeichen nötig (aber erlaubt).

Wer dieses Manuskript am Bildschirm liest oder an einen Farbdrucker geschickt hat, profitiert hoffentlich von der farblichen Gestaltung der Code-Beispiele. Es handelt sich um die Syntaxhervorhebungen der Entwicklungsumgebung Visual Studio 2015, die via Windows-Zwischenablage in den Text übernommen wurden.

Ob man beim Rumpf einer Klassen- oder Methodendefinition die öffnende geschweifte Klammer an das Ende der Kopfzeile setzt oder an den Anfang der Folgezeile, ist Geschmacksache, z.B.:

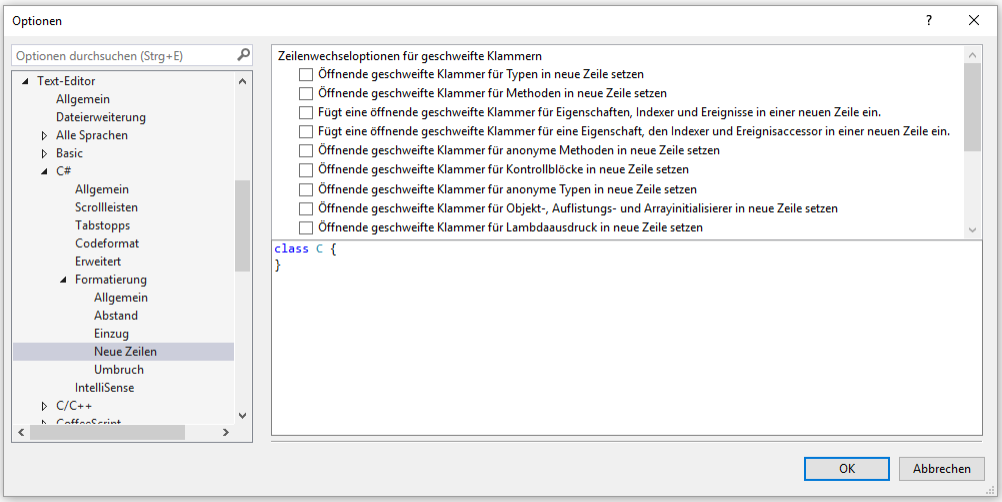
```
class Hallo {
  static void Main() {
    System.Console.WriteLine("Hallo");
  }
}
```

```
class Hallo
{
  static void Main()
  {
     System.Console.WriteLine("Hallo");
  }
}
```

Das Visual Studio bevorzugt die rechte Variante, kann aber nach

Extras > Optionen > Text-Editor > C# > Formatierung > Neue Zeilen

umgestimmt werden:



Weitere Hinweise zur übersichtlichen Gestaltung des C# - Quellcodes finden sich z.B. bei Krüger (2002) und im MSDN-Webangebot (*Microsoft Developer Network*) zu C#.[1]

3.1.4 Kommentar

C# bietet folgende Möglichkeiten, den Quelltext zu kommentieren:

- **Zeilenrestkommentar**
  Alle Zeichen von `//` bis zum Ende der Zeile gelten als Kommentar, wobei kein Kommentar-Terminierungszeichen erforderlich ist, z.B.:

  ```
  private int zaehler; // wird automatisch mit 0 initialisiert
  ```

  Hier wird eine Variablendeklarationsanweisung in derselben Zeile kommentiert.

- **Explizit terminierter Kommentar**
  Ein durch `/*` eingeleiteter Kommentar muss explizit durch `*/` terminiert werden. In der Regel wird diese Syntax für ausführliche Kommentar verwendet, die sich über mehrere Zeilen erstrecken, z.B.:

  ```
  /*
  Ein Bruch-Objekt verhindert, dass sein Nenner auf 0
  gesetzt wird, und hat daher stets einen definierten Wert.
  */
  public int Nenner {
     . . .
  }
  ```

[1] https://msdn.microsoft.com/de-de/library/ff926074.aspx

Ein mehrzeiliger Kommentar eignet sich u.a. auch dazu, ein Programmsegment (vorübergehend) zu deaktivieren, ohne es löschen zu müssen.

Speziell in Beispielen für Lehrtexte wird manchmal von der Möglichkeit Gebrauch gemacht, hinter einem explizit terminierten Kommentar in derselben Zeile eine Definition oder Anweisung fortzusetzen, z.B.:

```
void meth() {/* Anweisungen */}
```

Weil der explizit terminierte Kommentar (jedenfalls ohne farbliche Hervorhebung der auskommentierten Passage) etwas unübersichtlich ist, wird er selten verwendet.

- **Dokumentationskommentar**
  Neben den Kommentaren, welche ausschließlich das Lesen des Quelltexts unterstützen sollen, kennt C# noch den Dokumentationskommentar. Er wird vom Compiler bei einem Aufruf mit der Option **doc** in eine separate XML-Dokumentationsdatei umgesetzt, z.B.:

  ```
  csc  *.cs  /doc:Bruch.xml
  ```

  Daraus kann über meist kostenlos verfügbare Hilfsprogramme eine HTML-Dokumentation erstellt werden.[1] In Microsofts Entwicklungsumgebungen fehlt leider eine entsprechende Funktionalität.

  Ein Dokumentationskommentar darf vor einem benutzerdefinierten Typ (z.B. einer Klasse) oder vor einem Klassen-Member (z.B. Feld, Eigenschaft, Methode) stehen und wird in *jeder* Zeile durch drei Schrägstriche eingeleitet, z.B.:

  ```
  /// <summary>
  /// Ein Bruch-Objekt verhindert, dass sein Nenner auf Null
  /// gesetzt wird und hat daher stets einen sinnvollen Wert.
  /// </summary>
  public int Nenner {
     . . .
  }
  ```

  Durch `/**` eingeleitete und durch `*/` terminierte *mehrzeilige* Dokumentationskommentare erfordern die Beachtung spezieller Regeln und sind wegen des damit verbundenen Fehlerrisikos nicht empfehlenswert.

Wenn man im Editor unserer Entwicklungsumgebung das Auskommentieren eines markierten Blocks mit dem Menübefehl

Bearbeiten > Erweitert > Auswahl kommentieren

bzw. mit der Tastenkombination **Strg+K**, **Strg+C** veranlasst, werden doppelte Schrägstriche vor jede Zeile gesetzt.[2] Wendet man den Menübefehl

Bearbeiten > Erweitert > Auskommentierung der Auswahl aufheben

bzw. die Tastenkombination **Strg+K**, **Strg +U** auf einen zuvor mit Doppelschrägstrichen auskommentierten Block an, werden die Kommentarschrägstriche entfernt.

3.1.5 Namen

Für Klassen, Eigenschaften, Methoden, Felder, Parameter und sonstige Elemente eines C# - Programms benötigen wir Namen, wobei folgende Regeln gelten:

---

[1] Auf der folgenden Stack Overflow - Seite werden einige Hilfsprogramme erwähnt (z.B. *Sandcastle*):
http://stackoverflow.com/questions/3082044/how-to-turn-c-sharp-xml-doc-comments-into-something-useful

[2] Ist allerdings nur ein Teil *einer* Zeile markiert, bewirkt die Tastenkombination **Strg+K**, **Strg+C** zu einer Wandlung in einen explizit terminierten Kommentar.

- Die Länge eines Namens ist nicht begrenzt.
- Das erste Zeichen muss ein Buchstabe oder ein Unterstrich sein, danach dürfen außerdem auch Ziffern auftreten.
- C# - Programme werden intern im **Unicode**-Zeichensatz dargestellt. Daher erlaubt C# im Unterschied zu vielen anderen Programmiersprachen in Namen auch Umlaute oder sonstige nationale Sonderzeichen, die als Buchstaben gelten.
- Die Groß-/Kleinschreibung ist *signifikant*. Für den C# - Compiler sind also z.B.

  ```
  Anz   anz   ANZ
  ```

  grundverschiedene Namen.
- Die folgenden **reservierten Schlüsselwörter** dürfen nicht als Namen verwendet werden:

abstract	as	base	bool	break	byte	case	catch	char
checked	class	const	continue	decimal	default	delegate	do	double
else	enum	event	explicit	extern	false	finally	fixed	float
for	foreach	goto	if	implicit	in	int	interface	internal
is	lock	long	namespace	new	null	object	operator	out
override	params	private	protected	public	readonly	ref	return	sbyte
sealed	short	sizeof	stackalloc	static	string	struct	switch	this
throw	true	try	typeof	uint	ulong	unchecked	unsafe	ushort
using	virtual	void	volatile	while				

  Um ein reserviertes Wort doch als Name verwenden zu können, muss das @-Zeichen vorangestellt werden. Im folgenden Beispiel wird auf diese Weise der Variablenname `@object` realisiert:

  ```
  object @object = new object();
  Console.WriteLine(@object);
  ```

- Daneben gibt es **kontextbezogen-reservierte Schlüsselwörter**, die nur in bestimmten Kontexten als Namen verboten sind:[1, 2]

add	ascending	async	await	by	descending	dynamic	equals	from
get	global	group	into	join	let	on	orderby	partial
remove	select	set	value	var	where	yield	set	value

  Vorsichtshalber sollte man sie generell als Namen vermeiden.
- Namen müssen in ihrem Deklarationsbereich (siehe unten) eindeutig sein.

Während Sie obige Regeln einhalten *müssen*, ist die Beachtung der folgenden Konventionen freiwillig, aber empfehlenswert:

- Die Namen von *lokalen* (methodeninternen) Variablen (siehe Abschnitt 3.3.3), Methodenparametern und *privaten* (gekapselten, nur klassenintern ansprechbaren) Feldern werden *klein* geschrieben, z.B.:

`ggt`	lokale Variable in der `Bruch`-Methode `Kuerze()`
`b`	Parameter in der `Bruch`-Methode `Addiere()`
`nenner`	privates Feld in der Klasse `Bruch`

  Sonstige Namen (z.B. von Klassen, Methoden oder Eigenschaften) beginnen mit einem großen Anfangsbuchstaben, z.B.:

---

1 https://msdn.microsoft.com/en-us/library/the35c6y.aspx
2 https://msdn.microsoft.com/en-us/library/bb310804.aspx

`Bruch`	Name einer Klasse
`Kuerze()`	Name einer Methode
`Nenner`	Name einer Eigenschaft

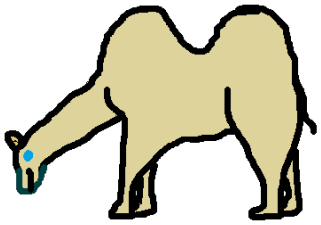
- Bei zusammengesetzten Namen beginnt jedes Wort mit einem Großbuchstaben (*Pascal Casing*), z.B.:

  ```
  WriteLine()
  ```

  Eine Ausnahme stellen die nach obiger Empfehlung mit einem Kleinbuchstaben zu beginnenden Namen dar (*Camel Casing*), z.B.:

  ```
  numberOfObjects
  ```

  Das zur Vermeidung von Urheberrechtsproblemen handgemalte Tier kann hoffentlich trotz ästhetischer Mängel zur Klärung des Begriffs *Camel Casing* beitragen:

  Alternativ kann man bei zusammengesetzter Namen auch den Unterstrich zur Verbesserung der Lesbarkeit verwenden, was der folgende Methodenname demonstriert, den das Visual Studio 2015 im `DmToEuro`-Projekt erstellt hat (siehe Abschnitt 2.2.4.3):

  ```
  private void button_Click(object sender, RoutedEventArgs e)
  ```

3.1.6 Übungsaufgaben zu Abschnitt 3.1

1) Welche **Main()** - Varianten sind zum Starten eines Programms geeignet?

```
static void main() { . . . }
public static void Main() { . . . }
static int Main() { . . . }
static double Main() { . . . }
static void Main() { . . . }
```

2) Welche von den folgenden Namen sind unzulässig?

```
4you   maiLink   else   Lösung   b_3
```

3.2 Ausgabe bei Konsolenanwendungen

Eine formatierte Konsolenausgabe lässt sich in C# recht bequem über die Methode **Console.WriteLine()** erzeugen, z.B.:

```
using System;
  . . .
Console.WriteLine("   {0}\n  -----\n   {1}\n", zaehler, nenner);
```

Es handelt es sich um eine statische Methode der Klasse **Console** aus dem Namensraum **System**, d.h.:

- Der Namensraum **System** muss am Beginn der Quelle per **using**-Direktive importiert werden, um die Klasse **Console** ohne Namensraumpräfix ansprechen zu können.
- Weil es sich um eine **statische** Methode handelt, richten wir den Methodenaufruf nicht an ein **Console**-*Objekt*, sondern an die Klasse selbst.
- Im Methodenaufruf sind Klassen- und Methodenname durch einen **Punkt** zu trennen.

WriteLine() schließt jede Ausgabe automatisch mit einer Zeilenschaltung ab. Wo dies unerwünscht ist, setzt man die ansonsten äquivalente **Console**-Methode **Write()** ein.

Sie kennen bereits zwei nützliche Spezialisierungen der **WriteLine()** - Methode (später werden wir von *Überladungen* sprechen):

- In obigem Beispiel ist die *formatierte* Ausgabe von zwei Werten zu sehen, wobei ein einleitender Zeichenfolgen-Parameter angibt, wie die Ausgabe der restlichen Parameter erfolgen soll. Auf diese Technik gehen wir in Abschnitt 3.2.2 näher ein.
- Oft reicht die im Abschnitt 3.2.1 behandelte Ausgabe einer (zusammengesetzten) Zeichenfolge.

3.2.1 Ausgabe einer (zusammengesetzten) Zeichenfolge

Im `Hallo`-Beispiel (vgl. Abschnitt 2.1.1) haben wir der **WriteLine()** - Methode als einzigen Parameter eine Zeichenkette zur Ausgabe auf dem Bildschirm übergeben:

```
Console.WriteLine("Hallo, echt .NET hier!");
```

Übergebene Argumente anderen Typs werden vor der Ausgabe automatisch in eine Zeichenfolge konvertiert, z.B. der Wert einer ganzzahligen Variablen (siehe unten):

```
int i = 4711;
Console.WriteLine(i);
```

Besonders angenehm ist die Möglichkeit, mehrere Teilausgaben mit dem „+“ - Operator zu verketten, z.B.:

```
int i = 4711;
Console.WriteLine("i hat den Wert: " + i);
```

Der Wert der ganzzahligen Variablen `i` wird in eine Zeichenfolge gewandelt, die anschließend an die Zeichenfolge `"i hat den Wert: "` angehängt und dann mit ihr zusammen ausgegeben wird:

Durch die bequeme Zeichenfolgenverkettung mit dem „+“ - Operator und die automatische Konvertierung von beliebigen Datentypen in eine Zeichenfolge ist die **WriteLine()** - Variante mit unformatierter Ausgabe schon recht flexibel. Außerdem erlauben die folgenden **Escape-Sequenzen** (vgl. Abschnitt 3.3.9.4), die wie gewöhnliche Zeichen in die Ausgabezeichenfolge geschrieben werden, eine Gestaltung der Ausgabe:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

Quellcode-Fragment	Ausgabe
`int i = 47, j = 1771;` `Console.WriteLine("Ausgabe:\n\t" + i + "\n\t" + j);`	`Ausgabe:` `47` `1771`

Noch mehr Gestaltungsmöglichkeiten bietet die im nächsten Abschnitt behandelte formatierte Ausgabe.

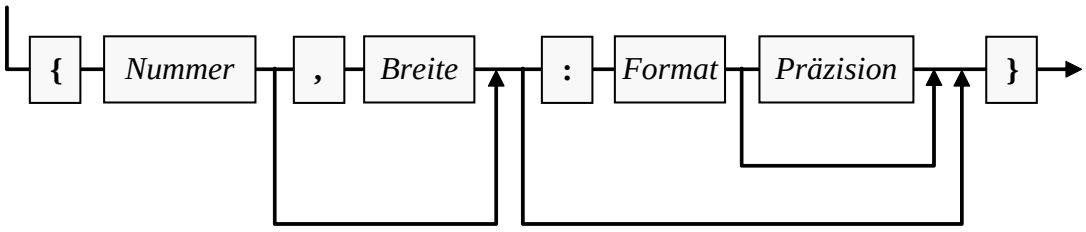
3.2.2 Formatierte Ausgabe

Die gleich zu beschreibenden Formatierungstechniken sind nicht nur bei Konsolenausgaben zu gebrauchen. Wir werden später auf analoge Weise mit der statischen Methode **Format()** der Klasse **String** kombinierte Zeichenfolgen bestehend aus festen und variablen Bestandteilen erzeugen, die z.B. im Rahmen einer grafischen Bedienoberfläche verwendet oder in eine Datei geschrieben werden sollen.

3.2.2.1 Traditionelle Variante mit Platzhaltern

Bei der traditionellen Variante formatierten Ausgabe per **Write()** oder **WriteLine()** wird als erster Parameter eine Zeichenfolge übergeben, die Platzhalter mit optionalen Formatierungsangaben für die restlichen, auf der Konsole auszugebenden Parameter enthält. Für einen Platzhalter ist folgende Syntax vorgeschrieben:

Platzhalter für die formatierte Ausgabe



Darin bedeuten:

Nummer	Fortlaufende Nummer des auszugebenden Arguments, bei 0 beginnend
Breite	Ausgabebreite für das zugehörige Argument Positive Werte bewirken eine *rechtsbündige*, negative Werte eine *linksbündige* Ausgabe.
Format	Formatspezifikation gemäß anschließender Tabelle
Präzision	Anzahl der Nachkommastellen oder sonstige Präzisionsangabe (abhängig vom Format), muss der Formatangabe unmittelbar folgen (ohne trennende Leerstellen)

Es werden u.a. folgende Formate unterstützt:

Format	**Beschreibung**	**Beispiele mit den Variablen** `int i = 41483; float f = 21.415926f;`	
		WriteLine-Parameterliste	Ausgabe
d, D	**Ganze Dezimalzahl**	`("{0,7:d}", i)`	`41483`
f, F	**Festformatierte Kommazahl** Präzision: Anzahl der Nachkommastellen	`("{0,7:f2}", f)`	`21,42`
e, E	**Exponentialnotation** Präzision: Anzahl Stellen in der Mantisse	`("{0:e}", f)` `("{0:e2}", f)`	`2,141593e+001` `2,14e+001`
ohne	Bei fehlender Formatangabe entscheidet der Compiler.	`("{0,10}", i)` `("{0,10}", f)`	`41483` `21,41593`

In der Formatierungszeichenfolge sind auch auszugebende gewöhnliche Zeichen und Escape-Sequenzen (vgl. Abschnitt 3.3.9.4) erlaubt:

\n	Zeilenwechsel (*new line*)
\t	Horizontaler Tabulator

Beispiel:

Quellcode	Ausgabe

```
using System;
class Prog {
  static void Main() {
    int i = 47, j = 1771;
    double d = 3.1415926;
    Console.WriteLine("Feste Breite:\n{0,8}\n{1,8}\n{2,8:f3}" +
                      "\nTabulatoren:\n\t{0}\n\t{1}\n\t{2}", i, j, d);
  }
}
```

```
Feste Breite:
      47
    1771
   3,142
Tabulatoren:
        47
        1771
        3,1415926
```

Auf eine Formatierungszeichenfolge mit *k* Platzhaltern müssen entsprechend viele Ausdrücke (z.B. Variablen) mit einem zum jeweiligen Platzhalterformat passenden Datentyp folgen.

Um die geschweiften Klammern als gewöhnliche Zeichen in eine Formatierungszeichenfolge aufzunehmen, müssen sie verdoppelt werden, z.B.:

```
Console.WriteLine("{{{0}, {1}}}", i, j);
```

3.2.2.2 Zeichenfolgeninterpolation

Seit der C# - Version 6.0 bietet die so genannte *Zeichenfolgeninterpolation* eine Vereinfachung bei der formatierten Ausgabe:

- Man schreibt ein **$**-Zeichen als Präfix vor die Formatierungszeichenfolge.
- Statt die formatiert auszugebenden Ausdrücke *hinter* die Formatierungszeichenfolge zu schreiben, setzt man sie *in* die Zeichenfolge an Stelle der bisher verwendeten Platzhalternummern.

Im Beispielprogramm aus dem letzten Abschnitt kann der **WriteLine()** - Aufruf mit der neuen Technik vereinfacht werden:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = 47, j = 1771;` `  double d = 3.1415926;` `  Console.WriteLine($"Feste Breite:\n{i,8}\n{j,8}\n{d,8:f3}" +` `  $"\nTabulatoren:\n\t{i}\n\t{j}\n\t{d}");` `  }` `}`	`Feste Breite:` `  47` `  1771` `  3,142` `Tabulatoren:` `  47` `  1771` `  3,1415926`

3.2.3 Übungsaufgaben zu Abschnitt 3.2

1) Wie ist das fehlerhafte „Rechenergebnis“ in folgendem Programm zu erklären?

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine("3,3 + 2 = " + 3.3 + 2);` `  }` `}`	`3,3 + 2 = 3,32`

Sorgen Sie mit einem Paar runder Klammern dafür, dass die folgende Ausgabe erscheint.

```
3,3 + 2 = 5,3
```

Verbringen Sie nicht zu viel Zeit mit der Aufgabe, weil wir die genauen technischen Hintergründe erst in Abschnitt 3.5.10 behandeln.

2) Schreiben Sie ein Programm, das aufgrund der folgenden Variablendeklaration und -initialisierung

```
int i = 4711, j = 471, k = 47, m = 4;
```

mit zwei **WriteLine()** - Aufrufen diese Ausgabe produziert:

```
Rechtsbündig:
        i = 4711
        j =  471
        k =   47
        m =    4

Linksbündig:
        4711 (i)
        471  (j)
        47   (k)
        4    (m)
```

3.3 Variablen und Datentypen

Während ein Programm läuft, müssen zahlreiche Informationen mehr oder weniger lange im Arbeitsspeicher des Rechners aufbewahrt und natürlich auch modifiziert werden, z.B.:

- Die Merkmalsausprägungen eines Objekts werden aufbewahrt, solange das Objekt existiert.
- Die zur Ausführung einer Methode benötigten Daten werden bis zum Ende des Methodenaufrufs gespeichert.

Zum Speichern eines Werts (z.B. einer Zahl) wird eine **Variable** verwendet, worunter Sie sich einen **benannten Speicherplatz von bestimmtem Datentyp** (z.B. Ganzzahl) vorstellen können.

Eine Variable erlaubt über ihren Namen den lesenden oder schreibenden Zugriff auf den zugeordneten Platz im Arbeitsspeicher, z.B.:

```
using System;
class Prog {
   static void Main() {
      int ivar;                        // Deklaration von ivar
      ivar = 4711;                     // schreibender Zugriff auf ivar
      Console.WriteLine(ivar);         // lesender Zugriff auf ivar
   }
}
```

Als wichtige Eigenschaften einer C# - Variablen halten wir fest:

- **Name**
  Es sind beliebige Bezeichner gemäß Abschnitt 3.1.5 erlaubt.
- **Datentyp**
  Damit sind festgelegt: Wertebereich, Speicherplatzbedarf und zulässige Operationen.
- **Aktueller Wert**
- **Ort im Hauptspeicher**
  Im Unterschied zu anderen Programmiersprachen (z.B. C++) spielt in C# die Verwaltung von Speicher*adressen* praktisch keine Rolle. Wir werden jedoch später zwei wichtige Speicherregionen unterscheiden (*Stack* und *Heap*).

3.3.1 Strenge Compiler-Überwachung bei C# - Variablen

Um die Details bei der Verwaltung der Variablen im Arbeitsspeicher müssen wir uns nicht kümmern, da wir schließlich mit einer problemorientierten, „höheren“ Programmiersprache arbeiten. Allerdings verlangt C# beim Umgang mit Variablen im Vergleich zu anderen Programmier- oder Skriptsprachen einige Sorgfalt, letztlich mit dem Ziel, Fehler zu vermeiden bzw. frühzeitig zu erkennen:

3.3.1.1 Explizite Deklaration

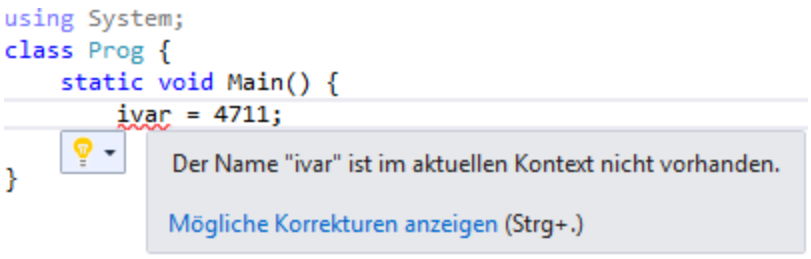
Variablen müssen **explizit deklariert** werden. In der folgenden Anweisung

```
int ivar;
```

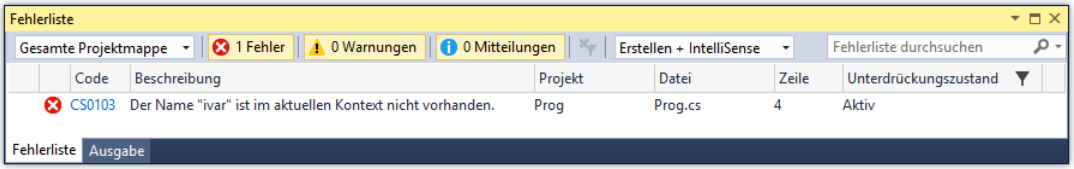
wird die Variable `ivar` vom Typ **int** (zum Speichern einer ganzen Zahl) deklariert. Wenn Sie versuchen, eine nicht deklarierte Variable zu verwenden, beschwert sich der Compiler, z.B.:

```
Prog.cs(5,3): error CS0103: Der Name "ivar" ist im aktuellen Kontext nicht
vorhanden.
```

Im Visual Studio informiert schon der Quellcode-Editor über das Problem, z.B.:



Versucht man trotzdem eine Übersetzung, dann erscheint die Compiler-Reklamation in der **Fehlerliste**:



Durch den Deklarationszwang werden z.B. Programmfehler wegen falsch geschriebener Variablennamen verhindert. Auch in Programmiersprache VB.NET (Visual Basic .NET) besteht per Voreinstellung Deklarationszwang. Hier lässt er sich jedoch mit der Compiler-Option **Option Explicit Off** aufheben. Diese für die Praxis nicht empfehlenswerte Option macht es möglich, das Verhalten vieler Skriptsprachen zu simulieren:

VB.NET - Quellcode	Ausgabe
`Option Explicit Off` `Module Module1` `  Sub Main()` `  ii = 12  ' Verwendung ohne Deklaration` `  ij = ii + 1  ' Tippfehler fällt nicht auf` `  Console.WriteLine(ii)` `  End Sub` `End Module`	`12`

3.3.1.2 Statische Typisierung

In C# ist für jede Variable bei der Deklaration ein fester (später nicht mehr änderbarer) **Datentyp** anzugeben. [1] Er legt fest, …

- welche Informationen (z.B. ganze Zahlen, Zeichen, Adressen von `Bruch`-Objekten) in der Variablen gespeichert werden können,
- welche Operationen auf die Variable angewendet werden dürfen.

Der Compiler kennt zu jeder Variablen den Datentyp und kann daher **Typsicherheit** garantieren, d.h. die Zuweisung von Werten mit ungeeignetem Datentyp verhindern. Außerdem kann auf (zeitaufwändige) Typprüfungen *zur Laufzeit* verzichtet werden. In der folgenden Anweisung

```
int ivar = 4711;
```

wird die Variable `ivar` vom Typ **int** deklariert, der für ganze Zahlen im Bereich von -2147483648 bis 2147483647 eignet ist.

Seit C# 4.0 ermöglicht der Datentyp **dynamic** eine Ausnahme von der statischen Typisierung, um einige sehr spezielle Aufgaben zu erleichtern:

- Kooperation mit typfreien Skriptsprachen wie z.B. *IronPython*
- Nutzung von COM (*Component Object Model*) - APIs (z.B. zur Office-Automatisierung)
- Zugriffe auf das HTML-Dokumentobjektmodell (DOM)

An Stelle des Compilers ist hier die CLR für die Typprüfung verantwortlich, was leicht zu Laufzeitfehlern führen kann. Dieser Datentyp sollte nur in begründeten Ausnahmefällen verwendet werden und kommt im Kurs nicht zum Einsatz.[2]

---

[1] Halten Sie die *statische Typisierung* (im Sinn von *unveränderlicher* Typfestlegung) in begrifflicher Distanz zu den bereits erwähnten *statischen Variablen* (im Sinn von *klassenbezogenen* Variablen). Das Wort *statisch* ist eingeführter Bestandteil bei beiden Begriffen, so dass es mir nicht sinnvoll erschien, eine andere Bezeichnung vorzunehmen, um die Doppelbedeutung zu vermeiden.

[2] Hier finden sich MSDN-Informationen zum Typ **dynamic**:
https://msdn.microsoft.com/de-de/library/dd264741.aspx

3.3.1.3 Initialisierung

In der folgenden Anweisung

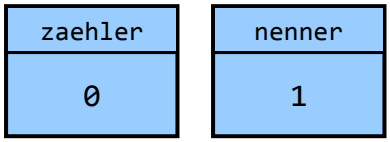
```
int ivar = 4711;
```

erhält die Variable `ivar` beim Deklarieren gleich den Initialisierungswert 4711. Auf diese oder andere Weise müssen Sie jeder *lokalen*, d.h. innerhalb einer Methode deklarierten, Variablen einen Wert zuweisen, bevor Sie zum ersten Mal lesend darauf zugreifen (vgl. Abschnitt 3.3.6). Weil die zu einem Objekt oder zu einer Klasse gehörigen Variablen (siehe unten) automatisch initialisiert werden, hat in C# *jede* Variable stets einen definierten Wert.

3.3.2 Wert- und Referenztypen

Bei der objektorientierten Programmierung werden neben den traditionellen Variablen zur Aufbewahrung von Zahlen, Zeichen oder Wahrheitswerten auch Variablen benötigt, welche die Adresse eines Objekts aufnehmen und so die Kommunikation mit dem Objekt ermöglichen. Wir unterscheiden also bei den Datentypen von Variablen zwei übergeordnete Kategorien:

- **Werttypen**
  Die traditionellen Datentypen werden in C# als *Werttypen* (engl.: *Value Types*) bezeichnet. Variablen mit einem Werttyp sind auch in C# unverzichtbar (z.B. als Felder von Klassen oder als lokale Variablen). In der `Bruch`-Klassendefinition (siehe Abschnitt 1.1) haben die Felder für Zähler und Nenner eines Objekts den Werttyp **int**, können also eine Ganzzahl im Bereich von –2147483648 bis 2147483647 aufnehmen. Sie werden in der folgenden Anweisung deklariert, wobei das Feld `nenner` auch noch einen Initialisierungswert erhält:
  ```
  int zaehler, nenner = 1;
  ```
  Beim Feld `zaehler` wird auf die explizite Initialisierung verzichtet, so dass die automatische Null-Initialisierung von **int**-Feldern greift. Für ein frisch erzeugtes `Bruch`-Objekt befinden sich im Arbeitsspeicher folgende Instanzvariablen (Felder):

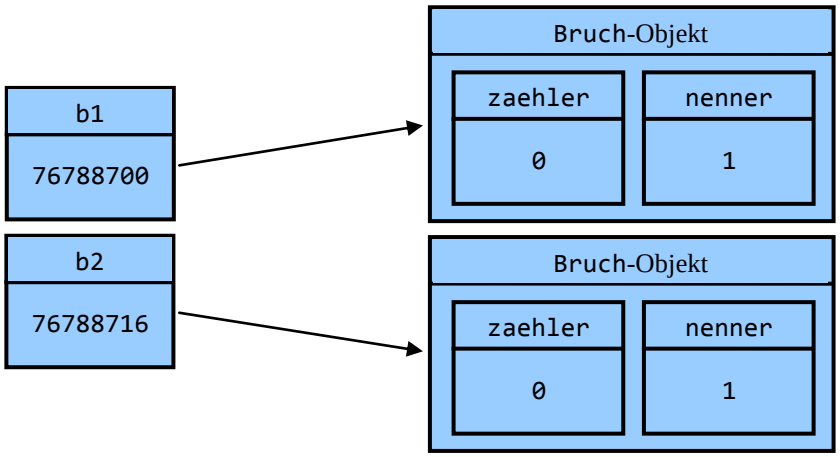
  

  In der `Bruch`-Methode `Kuerze()` tritt u.a. die lokale Variable `ggt` auf, die ebenfalls den Werttyp **int** besitzt:
  ```
  int ggt = 0;
  ```
  Wie Sie bereits wissen, ist bei *lokalen* (innerhalb einer Methode oder Eigenschaft deklarierten) Variablen vor dem ersten Lesezugriff eine Initialisierung erforderlich. Im Beispiel findet diese gleich bei der Deklaration statt (siehe Abschnitt 3.3.6).
- **Referenztypen**
  Eine Variable mit Referenztyp dient dazu, die **Speicheradresse eines Objekts** aus einer bestimmten Klasse aufnehmen. Sobald ein solches Objekt erzeugt und seine Speicheradresse der Referenzvariablen zugewiesen worden ist, kann das Objekt über die Referenzvariable angesprochen werden. Von den Variablen mit Werttyp unterscheidet sich eine Referenzvariable also …
  - durch ihren speziellen Inhalt (Objektadresse)
  - und durch ihre Rolle bei Kommunikation mit Objekten.

  Man kann jede Klasse (aus der FCL übernommen oder selbst definiert) als Datentyp verwenden, also Referenzvariablen von diesem Typ deklarieren. In der **Main()** - Methode der Klasse `Bruchaddition` werden z.B. die Referenzvariablen `b1` und `b2` aus der Klasse `Bruch` deklariert:

```
Bruch b1 = new Bruch(), b2 = new Bruch();
```

Sie erhalten als Initialisierungswert jeweils eine Referenz auf ein (per **new**-Operator, siehe unten) neu erzeugtes `Bruch`-Objekt. Daraus resultiert im Arbeitsspeicher die folgende Situation:



Das von `b1` referenzierte `Bruch`-Objekt wurde bei einem konkreten Programmlauf von der CLR an der Speicheradresse 76788700 (Hexadezimal: 0x0493b3dc) untergebracht. Wir plagen uns nicht mit solchen Adressen, sondern sprechen die dort abgelegten Objekte über Referenzvariablen an, z.B. in der folgenden Anweisung aus der **Main()** - Methode der Klasse `Bruchaddition`:

```
b1.Frage();
```

Jedes `Bruch`-Objekt enthält die Felder (Instanzvariablen) `zaehler` und `nenner` vom Werttyp **int**.

Zur Beziehung der Begriffe *Objekt* und *Variable* halten wir fest:

- Ein Objekt enthält im Allgemeinen mehrere Felder (Instanzvariablen) von beliebigem Typ. So enthält z.B. ein `Bruch`-Objekt die Felder `zaehler` und `nenner` vom Werttyp **int** (zur Aufnahme einer Ganzzahl). Bei einer späteren Erweiterung der `Bruch`-Klassendefinition werden ihre Objekte auch ein Feld mit Referenztyp erhalten.
- Eine Referenzvariable dient zur Aufnahme einer Objektadresse.
  So kann z.B. eine Variable vom Datentyp `Bruch` die Adresse eines `Bruch`-Objekts aufnehmen und die Kommunikation mit diesem Objekt unterstützen. Es ist ohne weiteres möglich und oft sinnvoll, dass mehrere Referenzvariablen die Adresse *desselben* Objekts enthalten. Das Objekt existiert unabhängig vom Schicksal einer konkreten Referenzvariablen, wird jedoch überflüssig, wenn im gesamten Programm keine einzige Referenz (Kommunikationsmöglichkeit) mehr vorhanden ist.

3.3.3 Klassifikation der Variablen nach Zuordnung

Nach der Zuordnung zu einer *Methode* oder *Eigenschaft*, zu einem *Objekt* oder zu einer *Klasse* unterscheidet man:

- **Lokale Variablen**
  Sie werden innerhalb einer Methode bzw. Eigenschaft deklariert. Ihre Gültigkeit beschränkt sich auf die Methode bzw. Eigenschaft, genauer: auf einen Anweisungsblock innerhalb der Methode bzw. Eigenschaft (siehe Abschnitt 3.3.7).

  Solange eine Methode bzw. Eigenschaft ausgeführt wird, befinden sich ihre Variablen in einem Speicherbereich, den man als *Stack* (dt.: *Stapel*) bezeichnet. Die obige Abbildung zeigt

die lokalen Variablen `b1` und `b2` aus der **Main()** - Methode der Klasse `Bruchaddition`, die als Referenzvariablen auf Objekte der Klasse `Bruch` zeigen.

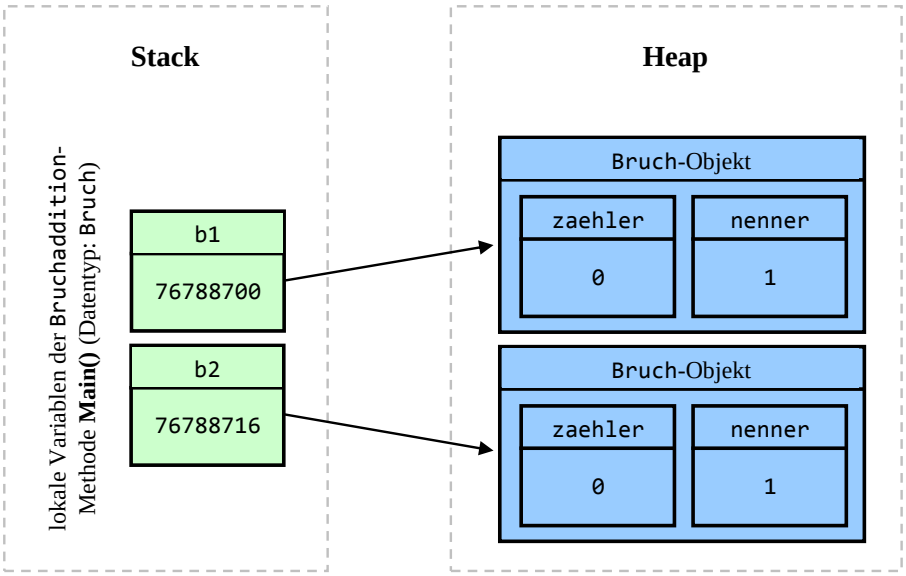
- **Instanzvariablen (nicht-statische Felder)**
  Jedes Objekt (man kann auch sagen: *jede Instanz*) einer Klasse verfügt über einen vollständigen Satz der Instanzvariablen dieser Klasse. So besitzt z.B. jedes Objekt der Klasse `Bruch` einen `zaehler` und einen `nenner`. Vielleicht wäre es terminlogisch klarer, von *Objektvariablen* zu sprechen. Wir bleiben aber bei der dominanten Bezeichnung, die z.B. auch von Microsoft (2012, S. 93) in der *C# 4.0 Language Specification* verwendet wird.

  Solange ein Objekt existiert, befinden es sich mit all seine Instanzvariablen in einem Speicherbereich, den man als *Heap* (dt.: *Haufen*) bezeichnet.

- **Klassenvariablen (statische Felder)**
  Diese Variablen beziehen sich auf eine Klasse insgesamt, nicht auf einzelne Instanzen der Klasse. Z.B. kann man in einer Klassenvariablen festhalten, wie viele Objekte der Klasse bereits bei einem Programmeinsatz erzeugt worden sind. In unserem Bruchrechnungs-Beispielprojekt haben wir der Einfachheit halber bisher auf statische Felder verzichtet.

  Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, die beim Erzeugen des Objekts auf dem Heap angelegt werden, existieren Klassenvariablen nur *einmal*. Sie werden beim Laden der Klasse zusammen mit anderen typbezogenen Informationen (z.B. Methodentabelle) auf dem Heap abgelegt.

Die im Wesentlichen schon aus Abschnitt 3.3.2 bekannte Abbildung zur Lage im Arbeitsspeicher bei Ausführung der **Main()** - Methode der Klasse `Bruchaddition` aus unserem OOP-Standardbeispiel (vgl. Abschnitt 1.1) wird anschließend ein wenig präzisiert. Durch Farben und Ortsangaben wird für die beteiligten lokalen Variablen bzw. Instanzvariablen die Zuordnung zu einer Methode bzw. zu einem Objekt und die damit verbundene Speicherablage verdeutlicht:



Die lokalen Referenzvariablen `b1` und `b2` der Methode **Main()** befinden sich im Stack-Bereich des Arbeitsspeichers und enthalten jeweils die Adresse eines `Bruch`-Objekts. Jedes `Bruch`-Objekt enthält die Felder (Instanzvariablen) `zaehler` und `nenner` vom primitiven Typ **int** und befindet sich im Heap-Bereich des Arbeitsspeichers.

Auf Instanz- und Klassenvariablen kann in allen Methoden der eigenen Klasse zugegriffen werden. Wenn (abweichend vom Prinzip der Datenkapselung) entsprechende Rechte eingeräumt werden, ist dies auch in Methoden fremder Klassen möglich.

In Kapitel 3 werden wir ausschließlich mit *lokalen* Variablen arbeiten. Im Zusammenhang mit der systematischen Behandlung der objektorientierten Programmierung werden die Instanz- und Klassenvariablen ausführlich erläutert.

Im Unterschied zu anderen Programmiersprachen (z.B. C++) ist es in C# *nicht* möglich, so genannte *globale* Variablen außerhalb von Klassen zu deklarieren.

3.3.4 Elementare Datentypen

Als *elementare Datentypen* sollen die in C# vordefinierten Werttypen zur Aufnahme von einzelnen Zahlen, Zeichen oder Wahrheitswerten bezeichnet werden. Speziell für Zahlen existieren diverse Datentypen, die sich hinsichtlich Wertebereich und Speicherplatzbedarf unterscheiden. Von der folgenden Tabelle sollte man sich vor allem merken, wo sie im Bedarfsfall zu finden ist. Eventuell sind Sie aber auch jetzt schon neugierig auf einige Details:

Typ	Beschreibung	Werte	Bits
sbyte	Diese Variablentypen speichern ganze Zahlen *mit* Vorzeichen. Beispiel: `int zaehler = -7`	–128 … 127	8
short		–32768 … 32767	16
int		–2147483648 ... 2147483647	32
long		–9223372036854775808 … 9223372036854775807	64
byte	Diese Variablentypen speichern ganze Zahlen ≥ 0. Beispiel: `byte alter = 31;`	0 … 255	8
ushort		0 … 65535	16
uint		0 ... 4294967295	32
ulong		0 … 18446744073709551615	64
float	Variablen vom Typ **float** speichern Gleitkommazahlen nach der Norm IEEE 754 (32 Bit) mit einer Genauigkeit von mind. 7 signifikanten Dezimalstellen. Beispiel: `float p = 1252.61f;` **float**-Literale (siehe unten) benötigen den Suffix **f** (oder **F**).	Minimum: $-3.4028235 \cdot 10^{38}$ Maximum: $3.4028235 \cdot 10^{38}$ Kleinster Betrag > 0: $1.4 \cdot 10^{-45}$	32 1 für das Vorz., 8 für den Expon., 23 für die Mantisse
double	Variablen vom Typ **double** speichern Gleitkommazahlen nach der Norm IEEE 754 (64 Bit) mit einer Genauigkeit von mind. 15 signifikanten Dezimalstellen. Beispiel: `double p=113445.626535891;`	Minimum: $-1,7976931348623157 \cdot 10^{308}$ Maximum: $1,7976931348623157 \cdot 10^{308}$ Kleinster Betrag > 0: $4,9 \cdot 10^{-324}$	64 1 für das Vorz., 11 für den Expon., 52 für die Mantisse

Typ	Beschreibung	Werte	Bits
decimal	Variablen vom Typ **decimal** speichern Gleitkommazahlen mit 28 bis 29 Dezimalstellen exakt und eignen sich besonders für die **Finanzmathematik**, wo Rundungsfehler zu vermeiden sind. Beispiel: `decimal p = 2344.2554634m;` **decimal**-Literale (siehe unten) benötigen den Suffix **m** (oder **M**).	Minimum: $-(2^{96}-1) \approx -7{,}9 \cdot 10^{28}$ Maximum: $2^{96}-1 \approx 7{,}9 \cdot 10^{28}$ Kleinster Betrag > 0: 10^{-28}	128 1 für das Vorz., 5 für den Expon., 96 für die Mantisse, restl. Bits ungenutzt Im Exponenten sind nur die Werte 0 bis 28 erlaubt, die negativ interpret. werden.
char	Variablen vom Typ **char** speichern ein Unicode-Zeichen. Im Speicher landet aber nicht die Gestalt eines Zeichens, sondern seine Nummer im Zeichensatz. Daher zählt **char** zu den ganzzahligen (integralen) Datentypen. Beispiel: `char zeichen = 'j';` **char** - Literale (s.u.) sind mit *einfachen* Anführungszeichen einzurahmen.	Unicode-Zeichen Tabellen mit allen Unicode-Zeichen sind z.B. auf der folgenden Webseite des Unicode-Konsortiums zu finden: http://www.unicode.org/charts/	16
bool	Variablen vom Typ **bool** speichern Wahrheitswerte. Beispiel: `bool cond = false;`	**true**, **false**	1

Eine *Gleitkommazahl* (synonym: *Gleitpunkt-* oder *Fließkommazahl*, englisch: *floating point number* dient zur approximativen Darstellung einer reellen Zahl in der EDV. Es werden drei Bestandteile separat gespeichert: Vorzeichen, Mantisse und Exponent. Diese ergeben nach folgender Formel den dargestellten Wert, wobei ***b*** für die Basis eines Zahlensystems steht (meist verwendet: 2 oder 10):

$$\textbf{Wert} = \textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot b^{\textbf{Exponent}}$$

Bei dieser von Konrad Zuse entwickelten Darstellungstechnik[1] resultiert im Vergleich zur Festkommadarstellung bei gleichem Speicherplatzbedarf ein erheblich größerer Wertebereich. Während die Mantisse für die Genauigkeit sorgt, speichert der Exponentialfaktor die Größenordnung, z.B.:

$$-0{,}0000001252612 = (-1) \cdot 1{,}252612 \cdot 10^{-7}$$
$$1252612000000000 = (1) \cdot 1{,}252612 \cdot 10^{15}$$

Durch eine Änderung des Exponenten könnte man das Dezimalkomma durch die Mantisse „gleiten“ lassen. Allerdings wird in der Regel durch eine Restriktion der Mantisse (z.B. auf das Intervall [1; 2)) für Eindeutigkeit gesorgt.

Weil der verfügbare Speicher für Mantisse und Exponent begrenzt ist (siehe obige Tabelle), bilden die Gleitkommazahlen nur eine endliche (aber für praktische Zwecke ausreichende) Teilmenge der reellen Zahlen. Zur Verarbeitung von Gleitkommazahlen wurde die *Gleitkommaarithmetik* entwickelt, normiert und zur Verbesserung der Verarbeitungsgeschwindigkeit teilweise sogar in Computer-Hardware realisiert. Nähere Informationen über die Darstellung von Gleitkommazahlen im Arbeitsspeicher eines Computers folgen für speziell interessierte Leser im Abschnitt 3.3.5.

---

[1] Quelle: http://de.wikipedia.org/wiki/Konrad_Zuse

3.3.5 Vertiefung: Darstellung von Gleitkommazahlen im Arbeitsspeicher des Computers

Die als *Vertiefung* bezeichneten Abschnitte können beim ersten Lesen des Manuskripts gefahrlos übersprungen werden. Sie enthalten interessante Details, über die man sich irgendwann im Verlauf der Programmierkarriere informieren sollte. Im Kurskontext dienen sie auch als Zeitvertreib für Teilnehmer mit Vorkenntnissen, die sich eventuell (z.B. im Abschnitt über elementare Sprachelemente) etwas langweilen.

3.3.5.1 Binäre Gleitkommadarstellung

Bei den binären Gleitkommatypen **float** und **double** werden auch „relativ glatte“ Zahlen in der Regel nicht genau gespeichert, wie das folgende Programm zeigt:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double df13 = 1.3f;` `  double df125 = 1.25f;` `  Console.WriteLine("{0,20:f16}", df13);` `  Console.WriteLine("{0,20:f16}", df125);` `  float ff13 = 1.3f;` `  Console.WriteLine("\n{0,20:f16}", ff13);` `  }` `}`	`1,2999999523162800` `1,2500000000000000` `1,3000000000000000`

Die Zahl 1,3 kann im **float**-Format (7 signifikante Dezimalstellen) *nicht* exakt gespeichert werden. Um dies zu demonstrieren, wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 16 Dezimalstellen ausgegeben.[1] Demgegenüber wird die Zahl 1,25 im **float**-Format fehlerfrei gespeichert. Mit einer **float**-Variablen lässt sich das Problem nicht demonstrieren, weil die Ungenauigkeit bei der Ausgabe von der CLR weggerundet wird.

Diese Ergebnisse sind durch das Speichern der Zahlen im **binären Gleitkommaformat** nach der vom ***I**nstitute of **E**lectrical and **E**lectronics **E**ngineers* (IEEE) veröffentlichten Norm **IEEE-754** zu erklären, wobei jede Zahl als Produkt aus drei getrennt zu speichernden Faktoren dargestellt wird:

$$\textbf{Vorzeichen} \cdot \textbf{Mantisse} \cdot 2^{\textbf{Exponent}}$$

Im ersten Bit einer **float**- oder **double** - Variable wird das Vorzeichen gespeichert (0: positiv, 1: negativ).

Für die Ablage des Exponenten (zur Basis 2) als Ganzzahl stehen 8 (**float**) bzw. 11 (**double**) Bits zur Verfügung. Allerdings sind im Exponenten die Werte 0 und 255 (**float**) bzw. 0 und 2047 (**double**) für Spezialfälle (z.B. denormalisierte Darstellung, +/-Unendlich) reserviert (siehe Abschnitt 3.6.2). Um auch die für Zahlen mit einem Betrag kleiner Eins benötigten *negativen* Exponenten darstellen zu können, werden Exponenten mit einer Verschiebung (*Bias*) um den Wert 127 (**float**) bzw. 1023 (**double**) abgespeichert und interpretiert. Besitzt z.B. eine **float**-Zahl den Exponenten 0, landet der Wert

$$01111111_2 = 127$$

im Speicher, und bei negativen Exponenten resultieren dort Werte kleiner als 127.

Abgesehen von betragsmäßig sehr kleinen Zahlen (siehe unten) werden die **float**- und **double**-Werte **normalisiert**, d.h. auf eine Mantisse im Intervall [1; 2) gebracht, z.B.:

---

[1] Man könnte einwenden, dass bei der Darstellung des wahren Wertes `1,3000000000000000` durch die Approximation `1,2999999523162800` nur 2 Stellen korrekt sind und damit die Zusicherung von 7 signifikanten Dezimalstellen in Zweifel ziehen. Allerdings ist die Zusicherung so gemeint, dass beim *Runden* auf bis zu 7 Stellen alle Ziffern stimmen, was im Beispiel der Fall ist.

$$24{,}48 = 1{,}53 \cdot 2^4$$

$$0{,}2448 = 1{,}9584 \cdot 2^{-3}$$

Zur Speicherung der Mantisse werden 23 (**float**) bzw. 52 (**double**) Bits verwendet. Weil die führende Eins der normalisierten Mantisse *nicht* abgespeichert wird (*hidden bit*), stehen alle Bits für die Restmantisse (die Nachkommastellen) zur Verfügung mit dem Effekt einer verbesserten Genauigkeit. Oft wird daher die Anzahl der Mantissen-Bits mit 24 (**float**) bzw. 53 (**double**) angegeben. Das *i*-te Mantissen-Bit (von links nach rechts mit *Eins* beginnend nummeriert) hat die Wertigkeit 2^{-i}, so dass sich der *dezimale* Mantissenwert folgendermaßen ergibt:

$$1+m \quad \text{mit} \quad m = \sum_{i=1}^{23\,\text{bzw.}\,52} b_i 2^{-i}, \;\; b_i \in \{0,1\}$$

Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), dem Exponenten *e* und dem dezimalen Mantissenwert (1+ *m*) speichert also bei normalisierter Darstellung den Wert:

$$\mathbf{(-1)^{\mathit{v}} \cdot (1 + \mathit{m}) \cdot 2^{\mathit{e}-127}} \quad \text{bzw.} \quad \mathbf{(-1)^{\mathit{v}} \cdot (1 + \mathit{m}) \cdot 2^{\mathit{e}-1023}}$$

In der folgenden Tabelle finden Sie einige normalisierte **float**-Werte:

Wert	**float-Darstellung (normalisiert)**		
	Vorz.	**Exponent**	**Mantisse**
0,75 = $(-1)^0 \cdot 2^{(126-127)} \cdot (1+0{,}5)$	0	01111110	10000000000000000000000
1,0 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}0)$	0	01111111	00000000000000000000000
1,25 = $(-1)^0 \cdot 2^{(127-127)} \cdot (1+0{,}25)$	0	01111111	01000000000000000000000
-2,0 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}0)$	1	10000000	00000000000000000000000
2,75 = $(-1)^0 \cdot 2^{(128-127)} \cdot (1+0{,}25+0{,}125)$	0	10000000	01100000000000000000000
-3,5 = $(-1)^1 \cdot 2^{(128-127)} \cdot (1+0{,}5+0{,}25)$	1	10000000	11000000000000000000000

Nun kommen wir endlich zur Erklärung der eingangs dargestellten Genauigkeitsunterschiede beim Speichern der Zahlen 1,25 und 1,3. Während die Restmantisse 0,25 perfekt dargestellt werden kann,

$$\begin{aligned} 0{,}25 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} \end{aligned}$$

gelingt dies bei der Restmantisse 0,3 nur approximativ:

$$\begin{aligned} 0{,}3 &= 0 \cdot 2^{-1} + 1 \cdot 2^{-2} + 0 \cdot 2^{-3} + 0 \cdot 2^{-4} + 1 \cdot 2^{-5} + \ldots \\ &= 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{4} + 0 \cdot \frac{1}{8} + 0 \cdot \frac{1}{16} + 1 \cdot \frac{1}{32} + \ldots \end{aligned}$$

Sehr aufmerksame Leser werden sich darüber wundern, wieso die Tabelle mit den elementaren Datentypen in Abschnitt 3.3.4 z.B.

$$1{,}4 \cdot 10^{-45}$$

als betragsmäßig kleinsten **float**-Wert nennt, obwohl der minimale Exponent nach obigen Überlegungen -126 beträgt, was zum (gerundeten) dezimalen Exponentialfaktor

$$1{,}2 \cdot 10^{-38}$$

führt. Dahinter steckt die *denormalisierte* Gleitkommadarstellung, die als Ergänzung zur bisher beschriebenen normalisierten Darstellung eingeführt wurde, um eine bessere Annäherung an die Zahl Null zu erreichen. Bei der denormalisierten Gleitkommadarstellung sind die Exponenten-Bits alle auf 0 gesetzt, und dem Exponentialfaktor wird der feste Wert 2^{-126} (**float**) bzw. 2^{-1022} (**double**) zugeordnet. Die Mantissen-Bits haben dieselbe Wertigkeiten (2^{-i}) wie bei der normalisierten Darstellung (siehe oben). Weil es kein *hidden bit* gibt, stellen sie aber nun einen dezimalen Wert im Intervall [0, 1) dar. Eine **float**- bzw. **double**-Variable mit dem Vorzeichen *v* (0 oder 1), mit kom-

plett auf 0 gesetzten Exponenten-Bits und dem dezimalen Mantissenwert *m* speichert also bei denormalisierter Darstellung die Zahl:

$$\mathbf{(-1)^v \cdot m \cdot 2^{-126}} \quad \text{bzw.} \quad \mathbf{(-1)^v \cdot m \cdot 2^{-1022}}$$

In der folgenden Tabelle finden Sie einige denormalisierte **float**-Werte:

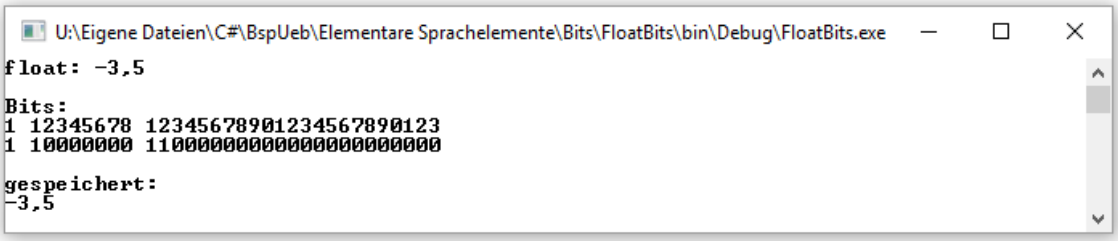
Wert	float-Darstellung (denormalisiert)		
	Vorz.	**Exponent**	**Mantisse**
$0{,}0 = (-1)^0 \cdot 2^{-126} \cdot 0$	0	00000000	00000000000000000000000
$-5{,}877472 \cdot 10^{-39} \approx (-1)^1 \cdot 2^{-126} \cdot 2^{-1}$	1	00000000	10000000000000000000000
$1{,}401298 \cdot 10^{-45} \approx (-1)^0 \cdot 2^{-126} \cdot 2^{-23}$	0	00000000	00000000000000000000001

Weil die Mantissen-Bits auch zur Darstellung der Größenordnung verwendet werden, schwindet die relative Genauigkeit mit der Annäherung an die Null.

Visual Studio - Projekte zur Anzeige der Bits einer (de)normalisierten **float**- bzw. **double**-Zahl finden Sie in den Ordnern

…\BspUeb\Elementare Sprachelemente\Bits\FloatBits
…\BspUeb\Elementare Sprachelemente\Bits\DoubleBits

Diese Programme werden nicht beschrieben, weil die erforderlichen Techniken (speziell die Nachbildung von C++ - Unions in C# mit Hilfe von Attributen, siehe Kapitel 13) im Kurs ansonsten keine Rolle spielen. Eine Beispielausgabe des Programms `FloatBits`:



3.3.5.2 Dezimale Gleitkommadarstellung

Neben den eben beschriebenen *binären* Gleitkommatypen (mit der Basis Zwei für Mantisse und Exponent) bietet C# auch den *dezimalen* Gleitkommatyp **decimal**, dessen Speicherorganisation die Basis 10 verwendet. Dabei werden 102 Bits folgendermaßen eingesetzt:[1]

- 1 Bit für das Vorzeichen
- 96 Bits für die Mantisse
  Hier wird eine Ganzzahl im Bereich von 0 bis 2^{96}-1 gespeichert.
- 5 Bits für den Exponenten
  Hier wird die *Anzahl* der dezimalen Nachkommastellen als Ganzzahl gespeichert, wobei nur Werte von 0 bis 28 erlaubt sind.

Eine **decimal**-Variable mit dem Vorzeichen *v* (0 oder 1), der Mantisse *m* und dem Exponenten *e* speichert den Wert:

$$\mathbf{(-1)^v \cdot m \cdot 10^{-e}}$$

Durch die 96-Manitissen-Bits einer **decimal**-Variablen lassen sich alle natürlichen Zahlen mit max. 28 Stellen darstellen:

---

[1] Von den insgesamt belegten 128 Bit bleiben also einige ungenutzt.

$$\overbrace{9999999999999999999999999999}^{28\text{ Stellen}} < 2^{96}-1 < \overbrace{99999999999999999999999999999}^{29\text{ Stellen}}$$

Folglich kann jede Dezimalzahl mit maximal 28 Stellen (Vorkommastellen + Nachkommastellen) exakt in einer **decimal**-Variablen gespeichert werden.

Wie in Abschnitt 3.3.5.1 zu sehen war, wird z.B. die Zahl 1,3 im *binären* Gleitkommaformat durch eine prinzipiell unendliche Serie von Brüchen mit einer Zweierpotenz im Nenner dargestellt, so dass die Begrenzung des Speichers zu einer Ungenauigkeit führt:

$$1{,}3 = 1 + \frac{1}{4} + \frac{1}{32} + \frac{1}{64} + \frac{1}{512} + \frac{1}{1024} + \ldots$$

Im *dezimalen* Gleitkommaformat wird die Zahl exakt gespeichert:

$$1{,}3 = 13 \cdot 10^{-1}$$

Das folgende Programm demonstriert den Genauigkeitsvorteil des Datentyps **decimal** im finanzmathematisch relevanten Wertebereich gegenüber den binären Gleitkommatypen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine(10.0f - 9.9f);` `  Console.WriteLine(10.0 - 9.9);` `  Console.WriteLine(10.0m - 9.9m);` `  }` `}`	`0,1000004` `0,0999999999999996` `0,1`

Allerdings hat der Typ **decimal** auch Nachteile im Vergleich zu **float** bzw. **double**

- Kleiner Wertebereich
  Beispielweise kann die Zahl 10^{30} in einer **double**-Variablen (sogar exakt) gespeichert werden, während sie außerhalb des **decimal**-Wertebereichs liegt.
- Hoher Speicherbedarf
  Eine **double**-Variable belegt nur halb so viel Speicherplatz wie eine **decimal**-Variable.
- Hoher Zeitaufwand bei arithmetischen Operationen
  Bei der Aufgabe,

  $$1300000000 - \sum_{i=1}^{1000000000} 1{,}3$$

  zu berechnen, ergaben sich für die Datentypen **double** und **decimal** folgende Genauigkeits- und Laufzeitunterschiede:[1]

  ```
  double:
    Abweichung:   -24,5162951946259
    Benöt. Zeit:  990,8876 Millisek.

  decimal:
    Abweichung:   0,0
    Benöt. Zeit:  19463,5288 Millisek.
  ```

  Die gut bezahlten Verantwortlichen bei vielen Banken, die sich gerne als „Global Player“ betätigen und dabei den vollen Sinn der beiden Worte ausschöpfen (mit Niederlassungen in

[1] Gemessen auf einem PC mit Intel-CPU Core i3 550 unter Windows 10-64

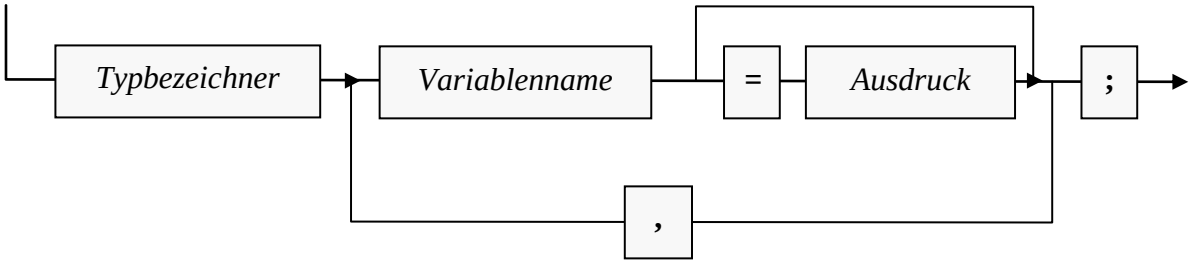
Schanghai, New York, Mumbai etc. und einem Verhalten wie im Spielcasino) wären heilfroh, wenn nach einem Spiel mit 1,3 Milliarden Euro Einsatz nur 24,52 Euro in der Kasse fehlen würden. Generell sind im Finanzsektor solche Fehlbeträge aber unerwünscht, so dass man bei finanzmathematischen Aufgaben trotz des erhöhten Zeitaufwands (im Beispiel: ca. Faktor 20) den Datentyp **decimal** verwenden sollte.

- Keine Unterstützung für Sonderfälle wie +/- Unendlich und NaN (***N**ot **a** **N**umber*) (vgl. Abschnitt 3.6)

3.3.6 Variablendeklaration, Initialisierung und Wertzuweisung

In C#-Programmen muss jede Variable vor ihrer ersten Verwendung deklariert werden. Dabei sind auf jeden Fall der Name und der Datentyp anzugeben, wie das Syntaxdiagramm zur **Variablendeklarationsanweisung** für lokale Variablen zeigt:

Deklaration von lokalen Variablen



Als Datentypen kommen in Frage (vgl. Abschnitt 3.3.2):

- Werttypen, z.B.

```
int i;
```

- Referenztypen, also Klassen (aus der FCL oder selbst definiert), z.B.

```
Bruch b;
```

Wir betrachten vorläufig nur *lokale* Variablen, die innerhalb einer Methode oder Eigenschaft existieren. Ihre Deklaration darf im Rumpf einer Methoden- bzw. Eigenschaftsdefinition an beliebiger Stelle *vor* der ersten Verwendung erscheinen. Es ist üblich, ihre Namen mit einem Kleinbuchstaben beginnen zu lassen (vgl. Abschnitt 3.1.5).

Neu deklarierte Variablen kann man optional auch gleich **initialisieren**, also auf einen gewünschten Wert bringen, z.B.:

```
int i = 4711;
Bruch b = new Bruch();
```

Im zweiten Beispiel wird per **new**-Operator ein `Bruch`-Objekt erzeugt und dessen Adresse in die neue Referenzvariable `b1` geschrieben. Mit der Objektkreation und auch mit der Konstruktion von gültigen *Ausdrücken*, die einen Wert von passendem Datentyp liefern müssen, werden wir uns noch ausführlich beschäftigen.

Wenn der Compiler den Datentyp erkennen kann, darf bei der Deklaration von lokalen Variablen über das Schlüsselwort **var** die **implizite Typisierung** verwendet werden:

```
var i = 4711;
var b = new Bruch();
```

Dabei wird das Prinzip der strengen und statischen Typisierung keinesfalls aufgehoben. Sobald der Mauszeiger über einem Variablennamen verharrt, zeigt sich die Entwicklungsumgebung perfekt über den Datentyp informiert:

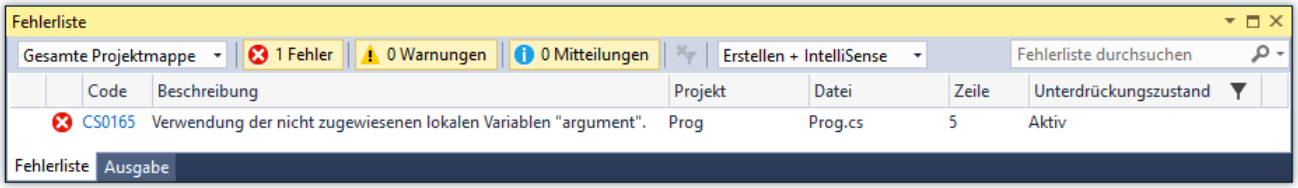
```
var it = 4711;
```
(lokale Variable) int it

Eventuell war im Beispiel für die Variable `it` aber der Datentyp **double** vorgesehen und versehentlich ein **int**-Literal zum Initialisieren verwenden worden. Dann kann der falsche Datentyp aufgrund der Unterschiede zwischen der Ganzzahl- und der Gleitkommaarithmetik (siehe Abschnitt 3.5.1) zum gravierenden Problem werden.

Weil *lokale* Variablen *nicht* automatisch initialisiert werden, muss man ihnen vor dem ersten lesenden Zugriff einen Wert zuweisen. Ein Verstoß gegen diese Regel wird vom C# - Compiler verhindert, wie das folgende Programm demonstriert:

```
using System;
class Prog {
  static void Main() {
    int argument;
    Console.WriteLine("Argument =  {0}", argument);
  }
}
```

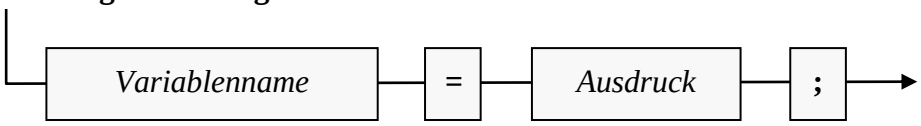
Der Compiler meint dazu:



Weil *Instanz- und Klassenvariablen* automatisch mit der typspezifischen Null initialisiert werden (siehe unten), ist in C# dafür gesorgt, dass alle Variablen beim Lesezugriff stets einen definierten Wert haben.

Um den Wert einer Variablen im weiteren Programmablauf zu verändern, verwendet man eine **Wertzuweisung**, die zu den einfachsten und am häufigsten benötigten Anweisungen gehört:

Wertzuweisungsanweisung



Beispiel: `ggt = az;`
Durch diese Wertzuweisungsanweisung aus der `Kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1.1) erhält die **int**-Variable `ggt` den Wert der **int**-Variablen `az`.

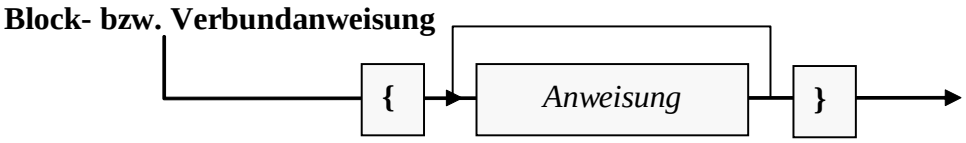
Es wird sich bald herausstellen, dass auch ein Ausdruck stets einen Datentyp hat. Bei der Wertzuweisung muss dieser Typ kompatibel zum Datentyp der Variablen sein.

Mittlerweile haben Sie zwei Sorten von C# - Anweisungen kennen gelernt:

- Deklaration von lokalen Variablen
- Wertzuweisung

3.3.7 Blöcke und Deklarationsbereiche für lokale Variablen

Wie Sie bereits wissen, besteht der Rumpf einer Methodendefinition aus einem Block mit beliebig vielen Anweisungen, der durch geschweifte Klammern begrenzt ist. Innerhalb des Methodenrumpfs können weitere Anweisungsblöcke gebildet werden, wiederum durch geschweifte Klammen begrenzt:



Man spricht hier auch von einer **Block- bzw. Verbundanweisung,** und diese kann überall stehen, wo eine einzelne Anweisung erlaubt ist.

Unter den Anweisungen eines Blocks dürfen sich selbstverständlich auch wiederum Blockanweisungen befinden. Einfacher ausgedrückt: Blöcke dürfen geschachtelt werden.

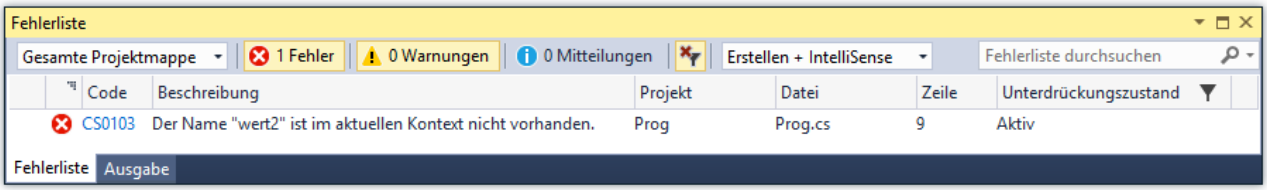
Oft treten Blöcke als Bestandteil von Bedingungen oder Schleifen (siehe Abschnitt 3.7) auf, z.B. in der Methode `Kuerze()` der Klasse `Bruch` (siehe Abschnitt 1.1):

```
public void Kuerze() {
 if (zaehler != 0) {
  int ggt = 0;
  int az = Math.Abs(zaehler);
  int an = Math.Abs(nenner);
    . . .
    . . .
  zaehler /= ggt;
  nenner  /= ggt;
 } else
  nenner = 1;
}
```

Anweisungsblöcke haben einen wichtigen Effekt auf die Gültigkeit der darin deklarierten Variablen: Eine lokale Variable ist verfügbar von der deklarierenden Zeile bis zur schließenden Klammer des innersten Blocks. Nur in diesem **Deklarations- bzw. Sichtbarkeitsbereich** kann sie über ihren Namen angesprochen werden. Beim Versuch, das folgende (weitgehend sinnfreie) Beispielprogramm

```
using System;
class Prog {
  static void Main() {
    int wert1 = 1;
    if (wert1 == 1) {
      int wert2 = 2;
      Console.WriteLine("Wert2 = " + wert2);
    }
    Console.WriteLine("Wert2 = " + wert2);
  }
}
```

zu übersetzen, erhält man vom Compiler die Fehlermeldung:



Bei hierarchisch geschachtelten Blöcken ist es in C# *nicht* erlaubt, auf mehreren Stufen Variablen mit identischem Namen zu deklarieren. Diese kaum sinnvolle Option ist in der Programmiersprache C++ vorhanden und erlaubt dort Fehler, die schwer aufzuspüren sind. In C# gehören die eingeschachtelten Blöcke zum Deklarationsbereich der umgebenden Blocks.

Zur übersichtlichen Gestaltung von C# - Programmen ist das Einrücken von Anweisungsblöcken sehr zu empfehlen, wobei Sie die Position der einleitenden Blockklammer und die Einrücktiefe nach persönlichem Geschmack wählen können, z.B.:

```
if (wert1 == 1)
{
    int wert2 = 2;
    Console.WriteLine("Wert2 = "+wert2);
}
```

```
if (wert1 == 1) {
  int wert2 = 2;
  Console.WriteLine("Wert2 = "+wert2);
}
```

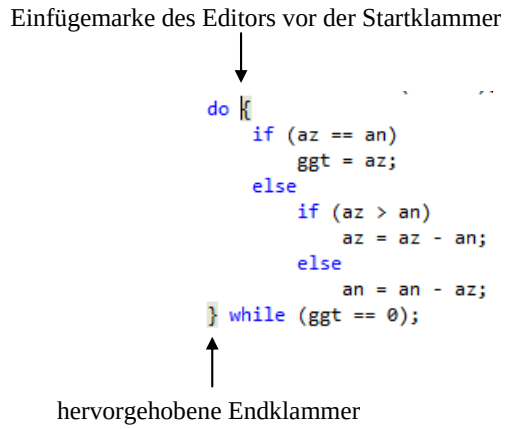
Bei den Quellcode-Editoren unsere Entwicklungsumgebungen kann ein markierter Block aus mehreren Zeilen mit

Tab komplett nach rechts eingerückt

und mit

Umschalt + Tab komplett nach links ausgerückt

werden. Außerdem kann man sich zu einer Blockklammer das Gegenstück anzeigen lassen:



3.3.8 Konstanten

Für die in einem Programm benötigten festen Werte (z.B. Mehrwertsteuersatz) sollte man in der Regel jeweils eine *Konstante* definieren, die dann im Quellcode über ihren Namen angesprochen werden kann, denn:

- Bei einer späteren Änderung des Wertes ist nur die Quellcodezeile mit der Konstantendeklaration betroffen.
- Der Quellcode ist leichter zu lesen.

Im folgenden Beispiel wird der Datentyp **decimal** verwendet, weil eine finanztechnische Anwendung angedeutet wird. Wie gleich im Abschnitt über Literale zu erfahren ist, benötigt ein fester Wert vom Typ **decimal** den Suffix m (z.B. `1.19m`):

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  const decimal MWST = 1.19m;` `  decimal netto = 28.10m, brutto;` `  brutto = netto * MWST;` `  Console.WriteLine("Brutto: {0:f2}", brutto);` `  }` `}`	`Brutto: 33,44`

Im Vergleich zu einer Variablen weist eine Konstante folgende Besonderheiten auf:

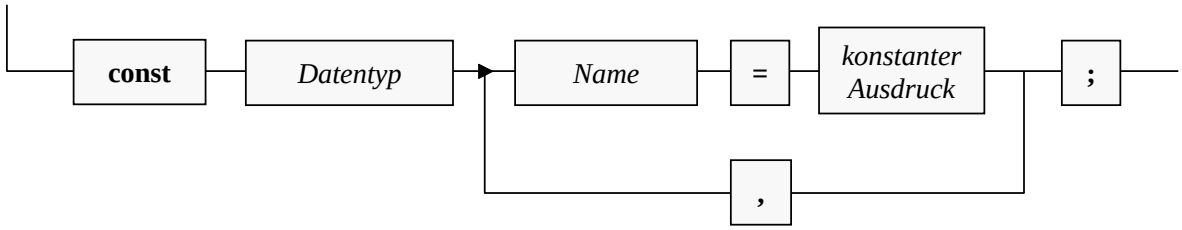
- Ihre Deklaration beginnt mit dem Modifikator **const** und *muss* eine Initialisierung enthalten, wobei ein *konstanter* Ausdruck zu verwenden ist, der nur Literale (siehe Abschnitt 3.3.9) und andere Konstanten enthält.
- Ihr Wert kann im Programm *nicht* geändert werden.

Programmierer verwenden traditionell im Namen einer Konstanten ausschließlich Großbuchstaben und verbessern bei einem Mehrwortnamen die Lesbarkeit durch trennende Unterstriche (z.B.: `MAXIMALE_QUOTE`). Dies ist aber nicht vorgeschrieben, und in der FCL scheint Microsoft folgende Konvention anzuwenden:

- Namen von Konstanten werden komplett groß geschrieben, wenn sie aus maximal zwei Zeichen bestehen, z.B. **Math.PI** (Kreiszahl π, siehe unten).
- Bei längeren Namen wird der Pascal-Stil verwendet (Anfangsbuchstaben aller Wörter groß), z.B. **Double.MaxValue** (größter Wert des Typs **double**).

Das Syntaxdiagramm zur Deklaration einer *konstanten* lokalen Variablen:

Deklaration von konstanten lokalen Variablen



Neben lokalen Variablen können auch Felder einer Klasse als konstant deklariert werden (siehe Abschnitt 4.2.5).

3.3.9 Literale

Die im Programmcode auftauchenden expliziten Werte bezeichnet man als *Literale*. Wie Sie aus dem Abschnitt 3.3.8 wissen, ist es oft sinnvoll, Literale innerhalb von Konstanten-Deklarationen zu verwenden, z.B.:

```
const decimal MWST = 1.19;
```

Aber auch andere Einsatzorte (z.B. Initialisierungen von Variablen, Parameterwerte) kommen in Frage.

Auch die Literale besitzen in C# stets einen **Datentyp**, wobei einige Regeln zu beachten sind, die gleich erläutert werden.

In diesem Abschnitt haben viele Passagen Nachschlage-Charakter, so dass man beim ersten Lesen nicht jedes Detail aufnehmen muss bzw. kann.

3.3.9.1 Ganzzahlliterale

Ganzzahlliterale können im dezimalen oder im hexadezimalen Zahlensystem (mit der Basis 16 und den Ziffern 0, 1, …, 9, A, B, C, D, E, F) geschrieben werden, wobei der hexadezimale Fall durch das Präfix **0x** oder **0X** zu kennzeichnen ist. Die Anweisungen:

```
int i = 11, j = 0x11;
Console.WriteLine("i = " + i + ", j = " + j);
```

liefern die Ausgabe:

```
i = 11, j = 17
```

Für das Ganzzahlliteral `0x11` ergibt sich der dezimale Wert 17 aufgrund der Stellenwertigkeiten im Hexadezimalsystem folgendermaßen:

$$11_{\text{Hex}} = 1 \cdot 16^1 + 1 \cdot 16^0 = 1 \cdot 16 + 1 \cdot 1 = 17$$

Eventuell fragen Sie sich, wozu man sich mit dem Hexadezimalsystem plagen sollte. Gelegentlich ist ein ganzzahliger Wert (z.B.als Methodenparameter) anzugeben, den man (z.B. aus einer Tabelle) nur in hexadezimler Darstellung kennt. In diesem Fall spart man sich durch Verwendung dieser Darstellung die Wandlung in das Dezimalsystem.

Der Datentyp eines Ganzzahlliterals hängt von seinem Wert und einem eventuell vorhandenen Suffix ab:

- Ist *kein* Suffix vorhanden, hat das Ganzzahlliteral den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

  int, **uint**, **long**, **ulong**

  Beispiele:

Literale	Typ
`2147483647, -21`	**int**
`2147483648`	**uint**
`-2147483649, 9223372036854775807`	**long**
`9223372036854775808`	**ulong**

- Ein positives Ganzzahlliteral mit Suffix **u** oder **U** (*unsigned*, ohne Vorzeichen) hat den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

  uint, **ulong**

  Beispiele:

Literale	Typ
`2147483647U`	**uint**
`9223372036854775808u`	**ulong**

- Ein Ganzzahlliteral mit Suffix **l** oder **L** (*Long*) hat den ersten Typ aus folgender Serie, der seinen Wert aufnehmen kann:

  long, **ulong**

  Beispiele:

Literale	Typ
`2147483647L`	**long**
`9223372036854775808L`	**ulong**

  Der Kleinbuchstabe **l** ist leicht mit der Ziffer **1** zu verwechseln und daher als Suffix ungeeignet. Aufgrund der in C# vorhandenen automatischen Typanpassungen wird das Suffix **L** sehr selten benötigt.

- Ein Ganzzahlliteral mit Suffix **ul**, **lu**, **UL**, **LU**, **uL**, **Lu**, **Ul**, oder **lU** hat den Typ **ulong**.

- Kann ein Wert von keinem Datentyp aufgenommen werden, warnt das Visual Studio, z.B.

```
using System;
class Prog {
    static void Main() {
        Console.WriteLine(18446744073709551616);
    }
}
```

struct System.Int32
Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar.

Die integrale Konstante ist zu groß.

  Obwohl die Fehlermeldung es nicht vermuten lässt, hat die Entwicklungsumgebung *alle* in Frage kommenden Datentypen (inkl. **ulong**) daraufhin untersucht, ob sie den Wert aufnehmen können.

Weil der Compiler ganze Zahlen gut kennt und einige Intelligenz bei der Verwendung von Ganzzahlliteralen zeigt, sind nach meinem Eindruck die eben erwähnten Suffixe überflüssig. Man kann z.B. einer vorzeichenlosen Ganzzahlvariablen ein Literal mit geeignetem Wert zuweisen, obwohl dieses Literal formal zu einem vorzeichenbehafteten Typ gehört:

```
uint i = 33;
```

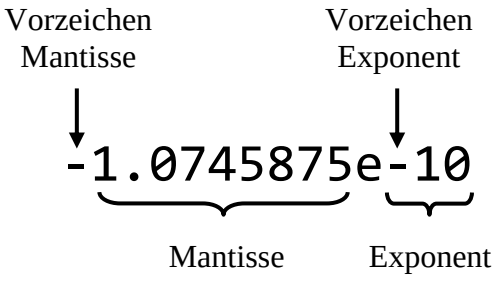
struct System.Int32
Stellt eine 32-Bit-Ganzzahl mit Vorzeichen dar.

3.3.9.2 Gleitkommaliterale

Zahlen mit Dezimalpunkt oder Exponent sind in C# vom Typ **double**, wenn nicht per Suffix ein alternativer Typ erzwungen wird:

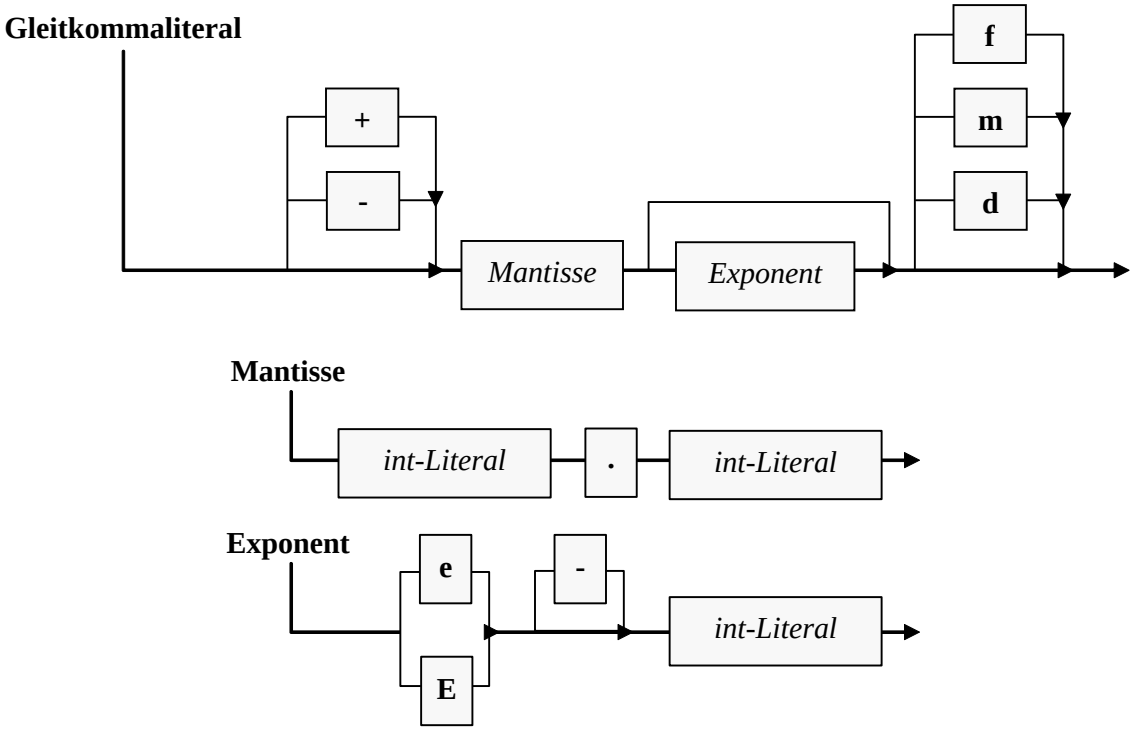
- Durch das Suffix **F** oder **f** wird der Datentyp **float** erzwungen, z.B.:
  `9.78f`
- Durch das Suffix **M** oder **m** wird der Datentyp **decimal** erzwungen, z.B.:
  `1.3m`
- Mit dem (kaum jemals erforderlichen) Suffix **D** oder **d** wird der Datentyp **double** „optisch betont“, z.B.:
  `1d`

Neben der alltagsüblichen Schreibeweise erlaubt C# bei Gleitkommaliteralen auch die Exponentialnotation (mit der Basis 10), z.B. bei der Zahl -0,00000000010745875):



Diese wissenschaftliche Notation erlaubt das Gleiten des Dezimaltrennzeichens, das die Bezeichnung *Gleitkommaliteral* begründet.

In den folgenden Syntaxdiagrammen werden die die wichtigsten Regeln für Gleitkommaliterale beschrieben:



Die in der Mantisse und im Exponenten auftretenden Ganzzahlliterale müssen das dezimale Zahlensystem verwenden und den Datentyp **int** besitzen, so dass die in Abschnitt 3.3.9.1 beschriebenen Präfixe (**0x**, **0X**) und Suffixe (z.B. **L**, **U**) verboten sind. Die Exponenten werden zur Basis Zehn verstanden.

Der Compiler achtet bei Wertzuweisungen streng auf die Typkompatibilität. Z.B. führt die folgende Deklarationsanweisung:

```
float p = 1.25;
```

zu der Fehlermeldung:

```
Literale des Typs "Double" können nicht implizit in den Typ "float" kon-
vertiert werden. Verwenden Sie ein F-Suffix, um ein Literal mit diesem
Typ zu erstellen.
```

Um das Problem zu lösen, muss ein Suffix an das Gleitkommaliteral angehängt werden:

```
float p = 1.25f;
```

Im Unterschied zu den Suffixen für Ganzzahlliterale (siehe Abschnitt 3.3.9.1) werden die Suffixe **f** und **m** für Gleitkommaliteralen also benötigt.

3.3.9.3 bool-Literale

Als Literale vom Typ **bool** sind nur die beiden reservierten Schlüsselwörter **true** und **false** erlaubt, z.B.:

```
bool cond = true;
```

Die **bool**-Literale sind mit *kleinem* Anfangsbuchstaben zu schreiben, obwohl sie in der Konsolenausgabe anders erscheinen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  bool b = false;` `  Console.WriteLine(b);` `  }` `}`	`False`

3.3.9.4 char-Literale

char-Literale werden in C# durch *einfache* Hochkommata begrenzt. Es sind erlaubt:

- **Einfache Zeichen**
  Beispiel:
  ```
  const char a = 'a';
  ```
  Das einfache Hochkomma kann allerdings auf diese Weise ebenso wenig zum **char**-Literal werden wie der Rückwärts-Schrägstrich (\). In diesen Fällen benötigt man eine so genannte *Escape-Sequenz*:
- **Escape-Sequenzen**
  Hier dürfen einem einleitenden Rückwärts-Schrägstrich u.a. folgen:
  - Ein Steuerzeichen, z.B.:

Neue Zeile	`\n`
Tabulator	`\t`
Signalton	`\a`

  - Einfaches oder doppeltes Hochkomma sowie der Rückwärts-Schrägstrich:
    `\'`
    `\"`
    `\\`

  Beispiel:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  char rs = '\\';` `  Console.WriteLine("Inhalt von rs: " + rs);` `  }` `}`	`Inhalt von rs: \`

- **Unicode-Escape-Sequenzen**
  Eine Unicode-Escape-Sequenz enthält eine Unicode-Zeichennummer (vorzeichenlose Ganzzahl mit 16 Bit, also im Bereich von 0 bis 2^{16}-1 = 65535) in hexadezimaler, vierstelliger Schreibweise (ggf. links mit Nullen aufgefüllt) nach der Einleitung durch **\u** oder **\x**. So lassen sich Zeichen ansprechen, die per Tastatur nicht einzugeben sind.
  Beispiel:
  ```
  const char alpha = '\u03b1';
  ```
  Im Konsolenfenster werden die Unicode-Zeichen oberhalb von \u00ff in der Regel als Fragezeichen dargestellt. In einem GUI-Fenster erscheinen alle Unicode-Zeichen in voller Pracht (siehe nächsten Abschnitt).

3.3.9.5 Zeichenkettenliterale

Zeichenkettenliterale werden (im Unterschied zu **char**-Literalen) durch *doppelte* Hochkommata begrenzt. Hinsichtlich der erlaubten Zeichen und der Escape-Sequenzen gelten die Regeln für **char**-Literale analog, wobei das einfache und das doppelte Hochkomma ihre Rollen tauschen, z.B.:

```
string name = "Otto's Welt";
```

Zeichenkettenliterale sind vom Datentyp **string**, und später wird sich herausstellen, dass es sich bei diesem Typ um eine Klasse aus dem Namensraum **System** handelt. Das (klein geschriebene!) Schlüsselwort **string** steht in C# als Aliasname für die Klassenbezeichnung **String** zur Verfügung. Wenn der Namensraum **System** (wie bei den meisten Quellcodedateien üblich) per **using**-Direktive importiert worden ist, kann die obige Variablendeklaration auch so geschrieben werden:

```
String name = "Otto's Welt";
```

Aus einem ganz speziellen Grund ist hier ausnahmsweise die Groß-/Kleinschreibung irrelevant.

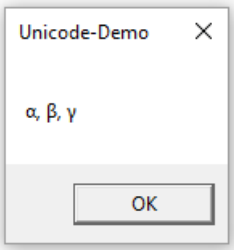
Während ein **char**-Literal stets *genau ein* Zeichen enthält, kann ein Zeichenkettenliteral aus beliebig vielen Zeichen bestehen oder auch leer sein, z.B.:

```
string name = "";
```

Das folgende Programm verwendet einen Aufruf der statischen **Show()** - Methode der Klasse **MessageBox** aus dem Namensraum **System.Windows** zur Anzeige eines Zeichenkettenliterals, das drei Unicode-Escape-Sequenzen enthält:[1]

```
class Prog {
   static void Main() {
      System.Windows.MessageBox.Show("\u03b1, \u03b2, \u03b3", "Unicode-Demo");
   }
}
```

Beim Programmstart erscheint das folgende Meldungsfenster:



Um die Bedeutung des Rückwärts-Schrägstrichs als Escape-Zeichen abzuschalten, stellt man das @-Zeichen voran, z.B.:

```
Console.WriteLine(@"Pfad: C:\Program Files\Map\bin");
```

Das Fehlen des Escape-Zeichens hat weitere Konsequenzen:

- Um ein doppeltes Hochkomma in die Zeichenfolge einzufügen, muss man es zweimal schreiben, z.B.:

  ```
  string s = @"Auftretenshäufigkeit für das Wort""Resonanz"":";
  ```

- Die Escape-Sequenz **\n** wird durch einen Zeilenumbruch im Quellcode ersetzt, was sich allerdings nachteilig auf die Formatierung des Quellcodes auswirkt, z.B.:

---

[1] Für eine erfolgreiche Übersetzung des Programms ist ein Verweis auf das Assembly **PresentationFramework.dll** erforderlich. Wie man im Visual Studio Verweise setzt, wird in Abschnitt 2.2.6.1 beschrieben.

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  string s = @"Literal` `mit Zeilenwechsel";` `  System.Console.WriteLine(s);` `  }` `}`	`Literal` `mit Zeilenwechsel`

Das @-Zeichen darf mit dem in Abschnitt 3.2.2.2 beschriebenen Interpolationspräfix ($) kombiniert werden, z.B.:

```
string s = "Map";
Console.WriteLine($@"Pfad: C:\Program Files\{s}\bin");
```

3.3.9.6 Referenzliteral null

Einer Referenzvariablen kann das Referenzliteral **null** zugewiesen werden, z.B.:

```
Bruch b1 = null;
```

Damit ist sie nicht undefiniert, sondern zeigt explizit auf nichts.

Zeigt eine Referenzvariable aktuell auf ein existentes Objekt, kann man diese Referenz per **null**-Zuweisung aufheben. Sofern im Programm keine andere Referenz auf dasselbe Objekt vorliegt, ist es zum Abräumen durch den Garbage Collector freigegeben.

Da C# eine streng typisierte Programmiersprache ist, und das Literal **null** einen Ausdruck darstellt (vgl. Abschnitt 3.5), muss es einen Datentyp besitzen. Es ist der **Nulltyp** (engl.: *null type*). Weil es in C# keinen Bezeichner für den Nulltyp gibt, kann man keine Variable von diesem Typ deklarieren. Außer dem **null**-Literal gibt es keinen weiteren Ausdruck mit Nulltyp, so dass man ihn im Programmieralltag getrost vergessen kann.

3.3.10 Übungsaufgaben zu Abschnitt 3.3

1) Wieso klagt der Compiler über ein unbekanntes Symbol, obwohl die Variable `i` deklariert worden ist?

Quellcode	Fehlermeldung
`class Prog {` `  static void Main() {` `  {` `  int i = 2;` `  }` `  System.Console.WriteLine(i);` `  }` `}`	`Prog.cs(6,28): error CS0103: Der Name i ist im aktuellen Kontext nicht vorhanden.`

2) Beseitigen Sie bitte alle Fehler in folgendem Programm:

```
class Prog {
  static void Main() {
    float pi = 3,141593;
    double radius = 2,0;
    System.Console.WriteLine('Der Flächeninhalt beträgt: {0:f3}',
                             pi * radius * radius);
  }
}
```

3) Schreiben Sie bitte ein Programm, das die folgende Ausgabe produziert:

```
Dies ist ein Zeichenkettenliteral:
        "Hallo"
```

4) In folgendem Programm erhält eine **char**-Variable das Zeichen 'c' als Wert. Anschließend wird dieser Inhalt auf eine **int**-Variable übertragen, und bei der Konsolenausgabe erscheint schließlich die Zahl 99:

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  char zeichen = 'c';` `  int nummer = zeichen;` `  System.Console.WriteLine("zeichen = " + zeichen +` `  "\nnummer = " + nummer);` `  }` `}`	`zeichen = c` `nummer = 99`

Warum hat der ansonsten sehr pingelige C# - Compiler nichts dagegen, einer **int**-Variablen den Wert einer **char**-Variaben zu übergeben? Wie kann man das Zeichen 'c' über eine Unicode-Escape-Sequenz ansprechen?

3.4 Einfache Techniken für Benutzereingaben

Für unsere Übungsprogramme brauchen wir gelegentlich eine einfache Möglichkeit, Benutzereingaben entgegen zu nehmen.

3.4.1 Via Konsole

In der `Frage()` - Methode der Klasse `Bruch` aus dem Einleitungsbeispiel (siehe Abschnitt 1.1) wird folgende Anweisung genutzt, um einen **int**-Wert von der Konsole entgegen zu nehmen:

```
zaehler = Convert.ToInt32(Console.ReadLine());
```

Von der statischen Methode **ReadLine()** der Klasse **Console** wird eine vom Benutzer per **Enter**-Taste abgeschickte Zeile von der Konsole gelesen. Diese Zeichenfolge dient anschließend als Argument für die statische Methode **ToInt32()** aus der Klasse **Convert**, die wie **Console** zum Namensraum **System** gehört. Sofern sich die übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich interpretieren lässt, liefert der **ToInt32()** - Aufruf diese Zahl zurück, und sie landet schließlich im **int**-Feld `zaehler`.

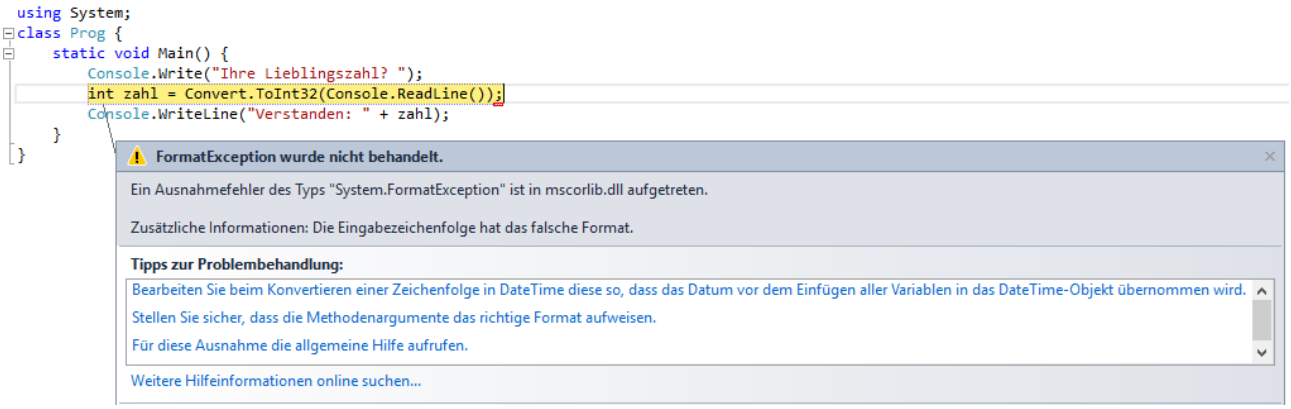
Hier liegt hier eine Verschachtelung zweier Methodenaufrufe vor, die bei Programmierern der kompakten Schreibweise wegen sehr beliebt ist. Ein etwas umständliches, aber für Anfänger leichter verständliches Äquivalent zur obigen Anweisung könnte lauten:

```
string eingabe;
eingabe = Console.ReadLine();
zaehler = Convert.ToInt32(eingabe);
```

Die vorgestellte Datenerfassungstechnik hat ein Problem mit weniger kooperativen Benutzern: Wird eine nicht konvertierbare Zeichenfolge abgeschickt, endet ein betroffenes Programm mit einem unbehandelten Ausnahmefehler, z.B.:

Quellcode	Ein- und Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.Write("Ihre Lieblingszahl? ");` `  int zahl = Convert.ToInt32(Console.ReadLine());` `  Console.WriteLine("Verstanden: " + zahl);` `  }` `}`	`Ihre Lieblingszahl? drei` `Unbehandelte Ausnahme:` `System.FormatException: Die Eingabezeichenfolge hat das falsche Format.`

Geschieht dies nach einem Start aus dem Visual Studio heraus mit der Taste **F5** oder mit dem Schalter ▶ (also im so genannten Debug-Modus), dann führt der Ausnahmefehler zu folgender Anzeige:



In dieser Situation lässt sich das havarierte und noch „baumelnde“ Programm mit dem Menübefehl

Debuggen > Debugging beenden

oder mit der Tastenkombination **Umschalt+F5** stoppen.

Um derartige Probleme zu verhindern, sind Programmiertechniken erforderlich, mit denen wir uns momentan noch nicht beschäftigen wollen, z.B. die folgende Ausnahmebehandlung:

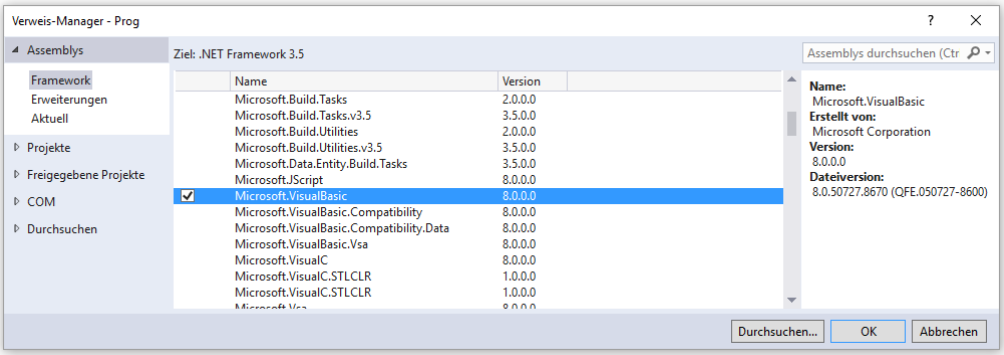
Quellcode	Ein- und Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int zahl;` `  Console.Write("Ihre Lieblingszahl? ");` `  try {` `  zahl = Convert.ToInt32(Console.ReadLine());` `  Console.WriteLine("Verstanden: " + zahl);` `  }` `  catch {` `  Console.WriteLine("Falsche Eingabe!");` `  }` `  }` `}`	`Ihre Lieblingszahl? drei` `Falsche Eingabe!`

In den Übungs- bzw. Demoprogrammen verwenden wir der Einfachheit halber ungesicherte **ToInt32** - Aufrufe bzw. analoge Varianten für andere Datentypen.

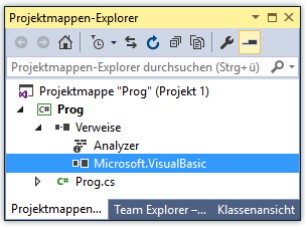
3.4.2 Via InputBox

Wer in einer simplen Anwendung mit grafischer Bedienoberfläche Benutzereingaben entgegen nehmen möchte, ohne die Komplexität einer WPF-Anwendung (vgl. Abschnitt 2.2.4) in Kauf nehmen zu müssen, kann statt der **Console**-Methode **ReadLine()** z.B. die statische Methode **InputBox()** der Klasse **Interaction** aus dem Namensraum **Microsoft.VisualBasic** benutzen. Die Klasse

Interaction ist als Migrationshilfe für Visual Basic 6 - Programmierer gedacht, kann aber auch in C# - Programmen genutzt werden. Dazu muss dem Compiler das implementierende und im **GAC** (Global Assembly Cache) installierte Assembly **microsoft.visualbasic.dll** bekannt gemacht werden. Wie Sie aus Abschnitt 2.2.6 bereits wissen, kann man unserer Entwicklungsumgebung sehr bequem erklären, welche Assembly-Referenzen beim Übersetzen eines Projekts erforderlich sind. Man wählt im Projektmappen-Explorer aus dem Kontextmenü zum Projektnamen die Option **Verweis hinzufügen** und kann dann in folgendem Dialog das gesuchte Assembly lokalisieren und im markierten Zustand per **OK** in die Verweisliste des Projekts aufnehmen:



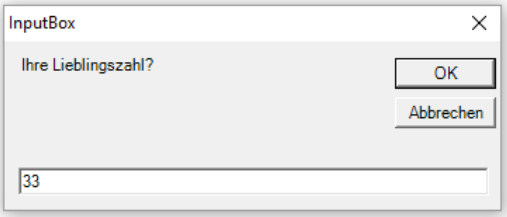
Anschließend wird die ergänzte Referenz im Projektmappen-Explorer angezeigt:



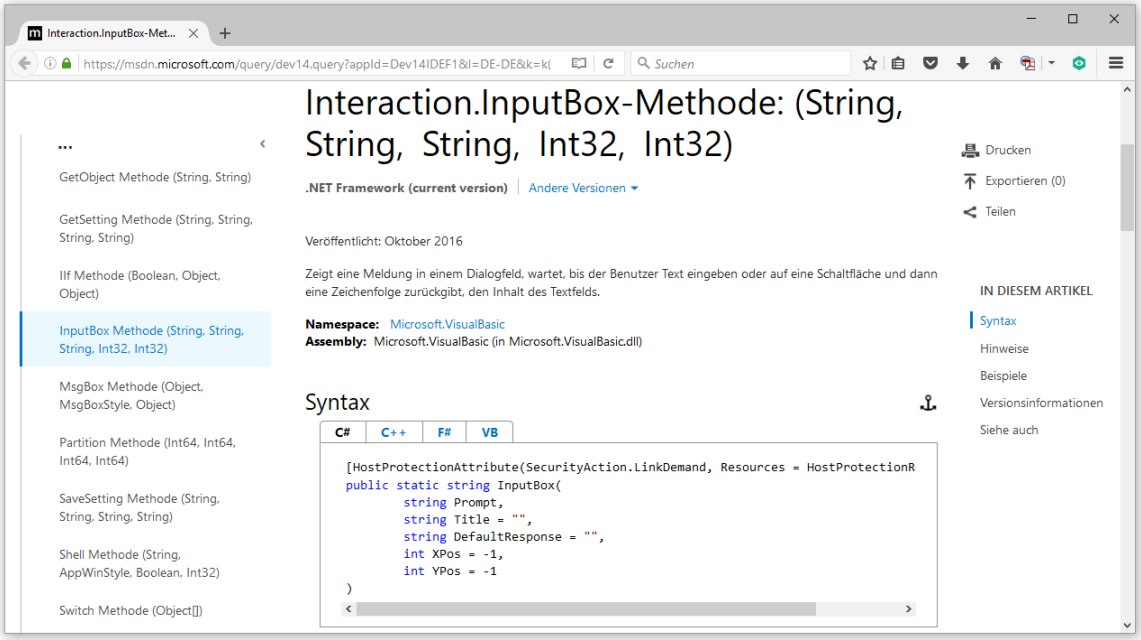
Für die folgende GUI-Alternative zum Beispielprogramm in Abschnitt 3.4.1 wird außerdem eine Referenz auf Assembly **PresentationFramework.dll** gesetzt, weil die Klasse **MessageBox** zum Einsatz kommt:

```
using System;
using System.Windows;
using Microsoft.VisualBasic;

class InputBox {
    static void Main() {
        string eingabe = Interaction.InputBox("Ihre Lieblingszahl?", "InputBox", "", -1, -1);
        int zahl = Convert.ToInt32(eingabe);
        MessageBox.Show("Verstanden: " + zahl);
    }
}
```

Ein- und Ausgabe werden per Dialogbox erledigt:



Über die Parameter (Argumente) und den Rückgabewert des **Inputbox()** - Methodenaufrufs kann man sich z.B. über die Hilfefunktion der Entwicklungsumgebung informieren. Bei markiertem Methodennamen ruft ein Druck auf die Taste **F1** die folgende Webseite auf:



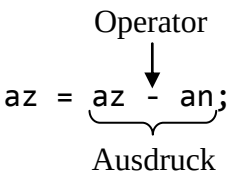
Zwar sieht die Ein- bzw. Ausgabe per GUI attraktiver aus, jedoch lohnt sich der erhöhte Aufwand bei Demo- bzw. Übungsprogrammen zu elementaren Sprachelementen kaum, so dass wir in der Regel darauf verzichten werden.

Die **Convert**-Methode **ToInt32()** reagiert natürlich auch im optisch aufgewerteten Programm auf ungeschickte Benutzereingaben mit einer Ausnahme (siehe Abschnitt 3.4.1). Später werden wir das Problem mit einer professionellen Ausnahmebehandlung lösen.

3.5 Operatoren und Ausdrücke

Im Zusammenhang mit der Variablendeklaration und der Wertzuweisung haben wir das Sprachelement *Ausdruck* ohne Erklärung benutzt, und diese soll nun nachgeliefert werden. Im aktuellen Abschnitt 3.5 werden wir Ausdrücke als wichtige Bestandteile von C# - Anweisungen recht detailliert betrachten. Dabei lernen Sie elementare Datenverarbeitungs-Möglichkeiten kennen, die von so genannten Operatoren mit ihren Argumenten veranstaltet werden, z.B. von den arithmetischen Operatoren (**+**, **-**, *****, **/**) für die Grundrechenarten. Am Ende des Abschnitts kann immerhin schon das Programmieren eines Währungskonverters als Übungsaufgabe gestellt werden. Allzu große Begeisterung wird wohl trotzdem nicht aufkommen, doch ein sicherer Umgang mit Operatoren und Ausdrücken ist unabdingbare Voraussetzung für das erfolgreiche Implementieren von Methoden und Eigenschaften. Hier werden die Algorithmen bzw. Handlungskompetenzen von Klassen oder Objekten realisiert.

Während die Variablen zur *Speicherung* von Werten dienen, geht es bei den **Operatoren** darum, aus vorhandenen Variableninhalten und/oder anderen Argumenten neue Werte zu berechnen. Den zur Berechnung eines Wertes geeigneten, aus Operatoren und zugehörigen Argumenten aufgebauten Teil einer Anweisung, bezeichnet man als **Ausdruck**, z.B. in der folgenden Wertzuweisung:



Durch diese Anweisung aus der `Kuerze()` - Methode unserer Klasse `Bruch` (siehe Abschnitt 1.1) wird der lokalen **int**-Variablen `az` der Wert des Ausdrucks `az - an` zugewiesen. Wie in diesem Beispiel landen die Werte von Ausdrücken oft in Variablen, wobei Ausdruck und Variable typkom-

patibel sein müssen. Aus den Operatoren eines Ausdrucks und den zugehörigen Argumenten ergibt sich nicht nur ein **Wert**, sondern auch ein **Datentyp**.

Schon bei einem Literal, einer Variablen oder einem Methodenaufruf haben wir es mit einem Ausdruck zu tun.[1]

Beispiel: `1.5`
Dies ist ein Ausdruck mit dem Typ **double** und dem Wert 1,5.

Mit Hilfe diverser Operatoren entsteht ein komplexerer Ausdruck, wobei sein Typ und sein Wert von den Argumenten und den Operatoren abhängen.

Beispiele: `2 * 1.5`
Hier resultiert der **double**-Wert 3,0.

`2 > 1.5`
Hier resultiert der **bool**-Wert **true**.

In der Regel beschränken sich die Operatoren darauf, aus ihren Argumenten (Operanden) einen Wert zu ermitteln und diesen für die weitere Verarbeitung zur Verfügung zu stellen. Einige Operatoren haben jedoch zusätzlich einen **Nebeneffekt** auf eine als Argument fungierende *Variable*.

Beispiel:
```
int i = 12;
int j = i++;
```
In der zweiten Anweisung des Beispiels tritt der **Postinkrementoperator** `++` mit der **int**-Variablen `i` als Argument auf. Der Ausdruck `i++` hat den Typ **int** und den Wert 12, welcher in der Zielvariable `j` landet. Außerdem wird die Argumentvariable `i` beim Auswerten des Ausdrucks durch den Postinkrementoperator auf den neuen Wert 13 gesetzt.

Die meisten Operatoren verarbeiten *zwei* Operanden (Argumente) und heißen daher **zweistellig** bzw. **binär**. Im folgenden Beispiel ist der **Additionsoperator** zu sehen, der zwei numerische Argumente erwartet:

```
a + b
```

Manche Operatoren verarbeiten nur *ein* Argument und heißen daher **einstellig** bzw. **unär**. Ein Beispiel ist der **Negationsoperator**, der mit einem „**!**“ bezeichnet wird, *ein* Argument vom Typ **bool** erwartet und dessen Wahrheitswert umdreht (**true** und **false** vertauscht):

```
!cond
```

Wir werden auch noch einen *dreistelligen* Operator kennen lernen.

Weil Ausdrücke von passendem Ergebnistyp als Argumente einer Operation erlaubt sind, können beliebig komplexe Ausdrücke aufgebaut werden. Unübersichtliche Exemplare sollten jedoch als potentielle Fehlerquellen vermieden werden.

3.5.1 Arithmetische Operatoren

Weil die arithmetischen Operatoren für die vertrauten Grundrechenarten der Schulmathematik zuständig sind, müssen ihre Operanden (Argumente) einen numerischen Typ haben (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float, double** oder **decimal**).

Die resultierenden **arithmetischen Ausdrücke** übernehmen ihren Ergebnistyp von den beiden Argumenten. Am Beispiel des Additionsoperators soll eine genauere Beschreibung geliefert werden.

---

[1] Besteht ein Ausdruck aus einem Methodenaufruf mit dem Pseudorückgabetyp **void**, dann liegt allerdings *kein* Wert vor.

C# kennt 7 vordefinierte Additionsoperatoren (siehe C# - Sprachspezifikation in Microsoft 2012, Abschnitt 7.8.4). In der folgenden Auflistung tritt eine ungewohnte Funktions- bzw. Methodennotation für den überladenen Additionsoperator auf, doch sind die Datentypen von Argumenten und Funktionswerten gut zu erkennen:

- Ganzzahladdition
  - **int operator +(int x, int y)**
  - **uint operator +(uint x, uint y)**
  - **long operator +(long x, long y)**
  - **ulong operator +(ulong x, ulong y)**
- Binäre Fließkommaaddition
  - **float operator +(float x, float y)**
  - **double operator +(double x, double y)**
- Dezimale Fließkommaaddition
  - **decimal operator +(decimal x, decimal y)**

Wenn bei einem Argument oder bei beiden Argumenten der Datentyp ungeeignet ist, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt (vgl. Abschnitt 3.5.7). Bevor z.B. ein **int**-Argument zu einem **double**-Wert addiert werden kann, muss es in den Typ **double** konvertiert werden. Sind z.B. beide Argumente einer Additionsoperation vom Typ **byte**, werden sie vor der Addition in den Typ **int** gewandelt, den auch die Summe erhält. Der Compiler lehnt es ab, diesen **int**-Wert in eine **byte**-Variable zu schreiben:

```
byte b1 = 1, b2 = 2;
b1 = b1 + b2;
```

(lokale Variable) byte b1

Der Typ "int" kann nicht implizit in "byte" konvertiert werden. Es ist bereits eine explizite Konvertierung vorhanden (möglicherweise fehlt eine Umwandlung).

Ist keine automatische Typanpassung möglich, beschwert sich der Compiler, z.B.:

```
double d = 2.0 + 1.19m;
```

Der +-Operator kann nicht auf Operanden vom Typ "double" und "decimal" angewendet werden.

Es hängt von den Datentypen der Argumente ab, ob die **Ganzzahl**-, oder die **Gleitkommaarithmetik** zum Einsatz kommt. Besonders auffällig sind die Unterschiede im Verhalten des Divisionsoperators, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = 2, j = 3;` `  double a = 2.0, b = 3.0;` `  Console.WriteLine(i / j);` `  Console.WriteLine(a / b);` `  }` `}`	`0` `0,666666666666667`

Bei der Ganzzahldivision werden die Stellen nach dem Dezimaltrennzeichen abgeschnitten, was gelegentlich durchaus erwünscht ist. Im Zusammenhang mit dem Über- bzw. Unterlauf (siehe Abschnitt 3.6) werden Sie noch weitere Unterschiede zwischen Ganzzahl- und Gleitkommaarithmetik kennenlernen.

In der nächsten Tabelle sind alle arithmetischen Operatoren beschrieben, wobei die Platzhalter *Num*, *Num1* und *Num2* für Ausdrücke mit einem numerischen Typ stehen, und *Var* eine numerische *Variable* vertritt:

Operator	Bedeutung	Beispiel	
		Quellcode-Fragment	Ausgabe
-Var	**Vorzeichenumkehr**	`int i = 2 , j = -3;` `Console.WriteLine("{0} {1}",-i,-j);`	`-2 3`
Num1 + Num2	**Addition**	`Console.WriteLine(2 + 3);`	`5`
Num1 – Num2	**Subtraktion**	`Console.WriteLine(2.6 - 1.1);`	`1,5`
*Num1 * Num2*	**Multiplikation**	`Console.WriteLine(4 * 5);`	`20`
Num1 / Num2	**Division**	`Console.WriteLine(8.0 / 5);` `Console.WriteLine(8 / 5);`	`1,6` `1`
Num1 % Num2	**Modulo (Divisionsrest)** Sei *GAD* der ganzzahlige Anteil aus dem Ergebnis der Division (*Num1* / *Num2*). Dann ist *Num1* % *Num2* def. durch *Num1* - *GAD* · *Num2*	`Console.WriteLine(19 % 5);` `Console.WriteLine(-19 % 5.25);`	`4` `-3,25`
++Var *−−Var*	**Präinkrement bzw. -dekrement** Als Argument ist nur eine Variable erlaubt. *++Var* liefert *Var* + 1 erhöht *Var* um 1 *−−Var* liefert *Var* − 1 reduziert *Var* um 1	`int i = 4;` `double a = 1.2;` `Console.WriteLine(++i + "\n" +` `--a);`	`5` `0,2`
Var++ *Var−−*	**Postinkrement bzw. -dekrement** Als Argument ist nur eine Variable erlaubt. *Var++* liefert *Var* erhöht *Var* um 1 *Var−−* liefert *Var* reduziert *Var* um 1	`int i = 4;` `Console.WriteLine(i++ + "\n" +` `i);`	`4` `5`

Bei den Inkrement- und den Dekrementoperatoren ist zu beachten, dass sie *zwei* Effekte haben:

- Das Argument wird ausgelesen, um den Wert des Ausdrucks zu ermitteln.
- Die als Argument fungierende numerische Variable wird verändert (vor oder nach dem Auslesen). Wegen dieses **Nebeneffekts** sind Prä- und Postinkrement- bzw. -dekrementausdrücke im Unterschied zu sonstigen arithmetischen Ausdrücken bereits vollständige *Anweisungen* (vgl. Abschnitt 3.7.1), wenn man ein Semikolon dahinter setzt, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = 12;` `  i++;` `  Console.WriteLine(i);` `  }` `}`	`13`

Ein (De)inkrementoperator bietet keine eigenständige mathematische Funktion, sondern eine vereinfachte Schreibweise. So ist z.B. die folgende Anweisung

```
j = ++i;
```

mit den beiden **int**-Variablen `i` und `j` äquivalent zu

```
i = i + 1;
j = i;
```

Für den eventuell bei manchen Lesern noch wenig bekannten Modulo-Operator gibt es viele sinnvolle Anwendungen, z.B.:

- Man kann für eine ganze Zahl bequem feststellen, ob sie gerade (durch Zwei teilbar) ist. Dazu prüft man, ob der Rest aus der Division durch Zwei gleich Null ist:

Quellcode-Fragment	Ausgabe
`int i = 19;` `Console.WriteLine(i % 2 == 0);`	`False`

- Man kann bei einer Gleitkommazahl den gebrochenen Anteil ermitteln bzw. abspalten:[1]

Quellcode-Fragment	Ausgabe
`double a = 7.1248239;` `double rest = a % 1.0;` `double ganz = a - rest;` `Console.WriteLine("{0} = {1,1:f0} + {2}", a, ganz, rest);`	`7,1248239 = 7 + 0,1248239`

3.5.2 Methodenaufruf

Obwohl Ihnen eine gründliche Behandlung der Methoden noch bevorsteht, haben Sie doch schon einige Erfahrung mit diesen Handlungskompetenzen von Klassen bzw. Objekten gewonnen:

- Die Arbeitsweise einer Methode kann von Argumenten (Parametern) abhängen.
- Viele Methoden liefern ein Ergebnis an den Aufrufer. Die in Abschnitt 3.4.1 vorgestellte Methode **Convert.ToInt32()** liefert z.B. einen **int**-Wert, sofern die als Parameter übergebene Zeichenfolge als ganze Zahl im **int**-Wertebereich (siehe Tabelle in Abschnitt 3.3.4) interpretierbar ist. Bei der Methodendefinition ist der Datentyp der Rückgabe anzugeben (siehe Syntaxdiagramm in Abschnitt 3.1.2.2).
- Liefert eine Methode dem Aufrufer *kein* Ergebnis, ist in der Definition der Pseudo-Rückgabetyp **void** anzugeben.
- Neben der Wertrückgabe hat ein Methodenaufruf oft weitere Effekte, z.B. auf die Merkmalsausprägungen des handelnden Objekts oder auf die Konsolenausgabe.

In syntaktischer Hinsicht halten wir fest, dass ein Methodenaufruf einen **Ausdruck** darstellt, wobei seine Rückgabe den Datentyp und den Wert des Ausdrucks bestimmt. Wir nehmen auch zur Kenntnis, dass es sich bei dem die Parameterliste begrenzenden Paar runder Klammern um den **() - Operator** handelt.[2] Jedenfalls sollten Sie sich nicht darüber wundern, dass der Methodenaufruf in Tabellen mit den C# - Operatoren auftaucht und dort eine (ziemlich hohe) Auswertungspriorität besitzt (vgl. Abschnitt 3.5.10).

Bei passendem Rückgabetyp darf ein Methodenaufruf auch als Argument für komplexere Ausdrücke oder für übergeordnete Methodenaufrufe verwendet werden (siehe Abschnitt 4.3.1.2). Bei einer Methode *ohne* Rückgabewert resultiert ein Ausdruck vom Typ **void**, der nicht als Argument für Operatoren oder andere Methoden taugt.

---

[1] Der (gerundete) ganzzahlige Anteil eines **double**- oder **decimal**-Wertes lässt sich auch über die statische Methode **Round()** bzw. **Truncate()** aus der Klasse **Math** bzw. aus der Struktur **Decimal** ermitteln.

[2] Diese Bezeichnung wird auch in Microsofts C# - Referenz verwendet, siehe z.B. http://msdn.microsoft.com/en-us/library/0z4503sa.aspx

Ein Methodenaufruf mit angehängtem Semikolon stellt eine **Anweisung** dar (vgl. Abschnitt 3.7), was Sie z.B. bei den zahlreichen Einsätzen der Methode **Console.WriteLine()** in unseren Beispielprogrammen bereits beobachten konnten.

Mit den arithmetischen Operatoren lassen sich nur elementare mathematische Probleme lösen. Darüber hinaus stellt das .NET - Framework eine große Zahl mathematischer Standardfunktionen (z.B. Potenzfunktion, Logarithmus, Wurzel, trigonometrische Funktionen) über statischen Methoden der Klasse **Math** im Namensraum **System** zur Verfügung (siehe FCL-Dokumentation). Im folgenden Programm wird die Methode **Pow()** zur Berechnung der allgemeinen Potenzfunktion (b^e) genutzt:

Quellcode	Ausgabe
`using System;` `class Prog{` `  static void Main(){` `  Console.WriteLine(Math.Pow(2.0, 3.0));` `  }` `}`	`8`

Alle **Math**-Methoden sind als **static** definiert, werden also von der Klasse selbst ausgeführt.

Im Beispielprogramm liefert die Methode **Math.Pow()** einen Rückgabewert vom Typ **double**, der gleich als Argument der Methode **Console.WriteLine()** Verwendung findet. Solche Verschachtelungen sind bei Programmierern wegen ihrer Kompaktheit ähnlich beliebt wie die Inkrement- bzw. Dekrementoperatoren. Ein etwas umständliches, aber für Anfänger leichter nachvollziehbares Äquivalent zum obigen **WriteLine()** - Aufruf könnte z.B. so aussehen:

```
double d;
d = Math.Pow(2.0, 3.0);
Console.WriteLine(d);
```

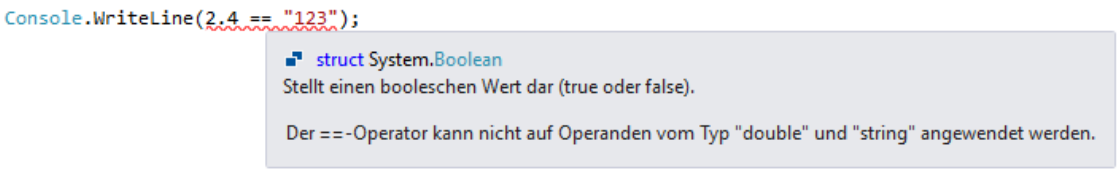
3.5.3 Vergleichsoperatoren

Durch Anwendung eines *Vergleichsoperators* auf zwei komparable (miteinander vergleichbare) Argumentausdrücke entsteht ein **Vergleich**. Dies ist ein einfacher **logischer Ausdruck** (vgl. Abschnitt 3.5.5), kann dementsprechend die booleschen Werte **true** (wahr) und **false** (falsch) annehmen und eignet sich dazu, eine *Bedingung* zu formulieren. Das folgende Beispiel dürfte verständlich sein, obwohl die **if**-Anweisung noch nicht behandelt wurde:

```
if (arg > 0)
  Console.WriteLine(Math.Log(arg));
```

In der folgenden Tabelle mit den von C# unterstützten Vergleichsoperatoren stehen ...

- *Expr1* und *Expr2* für miteinander vergleichbare Ausdrücke
  Hier ein Beispiel für *fehlende* Vergleichbarkeit:



- *Num1* und *Num2* für numerische Ausdrücke

Operator	Bedeutung	Beispiel	
		Programmfragment	Ausgabe
Expr1 `==` *Expr2*	**gleich**	`Console.WriteLine(2 == 3);`	`False`
Expr1 `!=` *Expr2*	**ungleich**	`Console.WriteLine(2 != 3);`	`True`
Num1 `>` *Num2*	**größer**	`Console.WriteLine(3 > 2);`	`True`
Num1 `<` *Num2*	**kleiner**	`Console.WriteLine(3 < 2);`	`False`
Num1 `>=` *Num2*	**größer oder gleich**	`Console.WriteLine(3 >= 3);`	`True`
Num1 `<=` *Num2*	**kleiner oder gleich**	`Console.WriteLine(3 <= 2);`	`False`

Achten Sie unbedingt darauf, dass der Identitätsoperator durch **zwei** „="-Zeichen ausgedrückt wird. Ein nicht ganz seltener C# - Programmierfehler besteht darin, beim Identitätsoperator das zweite Gleichheitszeichen zu vergessen. Dabei muss nicht unbedingt ein harmloser Syntaxfehler entstehen, der nach dem Studium einer Compiler-Fehlermeldung leicht zu beseitigen ist, sondern es kann auch ein mehr oder weniger unangenehmer Semantikfehler resultieren, also ein irreguläres Verhalten des Programms (vgl. Abschnitt 2.1.4 zur Unterscheidung von Syntax- und Semantikfehlern). Im ersten **WriteLine()** - Aufruf des folgenden Programms wird das Ergebnis eines Vergleichs auf die Konsole geschrieben:[1]

Quellcode	Ausgabe
`using System;` `class Prog{` `  static void Main(){` `  int i = 1;` `  Console.WriteLine(i == 2);` `  Console.WriteLine(i);` `  }` `}`	`False` `1`

Durch Weglassen eines Gleichheitszeichens wird aus dem Vergleich jedoch ein *Wertzuweisungsausdruck* (siehe Abschnitt 3.5.8) mit dem Typ **int** und dem Wert 2:

Quellcode	Ausgabe
`using System;` `class Prog{` `  static void Main(){` `  int i = 1;` `  Console.WriteLine(i = 2);` `  Console.WriteLine(i);` `  }` `}`	`2` `2`

Die versehentlich entstandene Zuweisung sorgt nicht nur für eine unerwartete Ausgabe, sondern verändert auch den Wert der Variablen `i`, was im weiteren Verlauf eines größeren Programms recht unangenehm werden kann.

[1] Wir wissen schon aus Abschnitt 3.2.1, dass **WriteLine()** einen Ausdruck mit beliebigem Typ verarbeiten kann, wobei automatisch eine Zeichenfolgen-Repräsentation erstellt wird.

3.5.4 Vertiefung: Gleitkommawerte vergleichen

Bei den *binären* Gleitkommatypen (**float** und **double**) muss man beim Identitätstest unbedingt technisch bedingte Abweichungen von der reinen Mathematik berücksichtigen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  const double USD = 1.0e-14; // Unterschieds-Schwelle Double` `  double d1 = 10.0 - 9.9;` `  double d2 = 0.1;` `  Console.WriteLine(d1 == d2);` `  Console.WriteLine(10.0m - 9.9m == 0.1m);` `  Console.WriteLine(Math.Abs((d1 - d2)/d1) < USD);` `  }` `}`	`False` `True` `True`

Der Vergleich

```
10.0 - 9.9 == 0.1
```

führt trotz Datentyp **double** (mit mindestens 15 signifikanten Dezimalstellen) zum Ergebnis `False`. Wenn man die in Abschnitt 3.3.5.1 beschriebenen Genauigkeitsprobleme bei der Speicherung von binären Gleitkommazahlen berücksichtigt, ist das Vergleichsergebnis *nicht* überraschend. Im Kern besteht das Problem darin, dass mit der binären Gleitkommatechnik auch relativ „glatte“ rationale Zahlen (wie z.B. 9,9) nicht exakt gespeichert werden können:[1]

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double d = 9.9f;` `  Console.WriteLine("{0,20:f16}", d);` `  }` `}`	`9,8999996185302700`

Folglich steckt im zwischengespeicherten Berechnungsergebnis 10,0 - 9,9 ein anderer Fehler als im Speicherabbild der Zahl 0,1. Weil die Vergleichspartner nicht Bit für Bit identisch sind, meldet der Identitätsoperator ein **false**.

Für Anwendungen im Bereich der Finanzmathematik wurde schon in Abschnitt 3.3.5 der dezimale Gleitkommadatentyp **decimal** empfohlen. Mit der *dezimalen* Gleitkommaarithmetik und -speichertechnik resultiert beim Vergleich

```
10.0m - 9.9m == 0.1m
```

das korrekte Ergebnis **true**. Allerdings eignen sich **decimal**-Variablen wegen ihres relativ kleinen Wertebereichs und des hohen Rechenzeitaufwands nicht für alle Anwendungen.

Um eine praxistaugliche Identitätsbeurteilung von **double**-Werten zu erhalten, sollte eine an der Rechen- bzw. Speichergenauigkeit orientierte **Unterschiedlichkeitsschwelle** verwendet werden. Nach diesem Vorschlag werden zwei **normalisierte** (also insbesondere von Null verschiedene) **double**-Werte d_1 und d_2 (vgl. Abschnitt 3.3.5.1) dann als numerisch identisch betrachtet, wenn der relative Abweichungsbetrag kleiner als $1{,}0 \cdot 10^{-14}$ ist:

---

[1] Um dies zu demonstrieren, wird ein **float**-Wert (erzwungen per Literal-Suffix **f**) in einer **double**-Variablen abgelegt und dann mit 16 Dezimalstellen ausgegeben. Mit einer **float**-Variablen lässt sich das Problem nicht demonstrieren, weil die Ungenauigkeit bei der Ausgabe von der CLR weggerundet wird.

$$\left|\frac{d_1 - d_2}{d_1}\right| < 1{,}0 \cdot 10^{-14}$$

Die Wahl der Bezugsgröße d_1 oder d_2 für den Nenner ist beliebig. Um das Verfahren vollständig festzulegen, wird die Verwendung der betragsmäßig größeren Zahl vorgeschlagen.

Ein Vorschlag zur Definition der *numerischen Identität* von zwei **double**-Werten muss die *relative* Differenz zugrunde legen, weil die technisch bedingten Mantissenfehler bei zwei **double**-Variablen mit eigentlich identischem Wert in Abhängigkeit vom Exponenten zu sehr unterschiedlichen Gesamtfehlern in der Differenz führen können. Vom häufig anzutreffenden Vorschlag,

$$|d_1 - d_2|$$

mit einer Schwelle zu vergleichen, ist daher abzuraten. Dieses Verfahren ist (bei geeignet gewählter Schwelle) nur tauglich für Zahlen in einem engen Größenbereich. Bei einer Änderung der Größenordnung muss die Schwelle angepasst werden.

Zu einer Schwelle für die relative Abweichung $\left|\frac{d_1 - d_2}{d_1}\right|$ gelangt man durch Betrachtung von zwei normalisierten **double**-Variablen d_1 und d_2, die bis auf ihre durch begrenzte Speicher- und Rechengenauigkeit bedingten Mantissenfehler e_1 bzw. e_2 denselben Wert $(1 + m)\ 2^k$ enthalten:

$$d_1 = (1 + m + e_1)\ 2^k \quad \text{und} \quad d_2 = (1 + m + e_2)\ 2^k$$

Für den Betrag des technisch bedingten relativen Fehlers gilt bei normalisierten Werten (mit einer Mantisse im Intervall [1, 2)) mit der oberen Schranke ε für den absoluten Mantissenfehler einer einzelnen **double**-Zahl die Abschätzung:

$$\left|\frac{d_1 - d_2}{d_1}\right| = \left|\frac{e_1 - e_2}{1 + m + e_1}\right| \le \frac{|e_1| + |e_2|}{|1 + m + e_1|} \le \frac{2 \cdot \varepsilon}{|1 + m + e_1|} \le 2 \cdot \varepsilon \quad (\text{wegen } (1 + m + e_1) \in [1, 2))$$

Bei normalisierten **double**-Werten (mit 52 Mantissen-Bits) ist aufgrund der begrenzten Speichergenauigkeit mit Fehlern im Bereich des halben Abstands zwischen zwei benachbarten Mantissenwerten zu rechnen:

$$2^{-53} \approx 1{,}1 \cdot 10^{-16}$$

Die vorgeschlagene Schwelle $1{,}0 \cdot 10^{-14}$ berücksichtigt über den Speicherfehler hinaus auch eingeflossene Rechnungsungenauigkeiten. Mit welcher Fehlerkumulation bzw. -verstärkung zu rechnen ist, hängt vom konkreten Algorithmus ab, so dass die Unterschiedlichkeitsschwelle eventuell angehoben werden muss. Immerhin hängt sie (anders als bei einem Kriterium auf Basis der einfachen Differenz $|d_1 - d_2|$) nicht von der Größenordnung der Zahlen ab.

An der vorgeschlagenen Identitätsbeurteilung mit Hilfe einer Schwelle für den relativen Abweichungsbetrag ist u.a. zu bemängeln, dass eine Verallgemeinerung für die mit geringerer Genauigkeit gespeicherten *denormalisierten* Werte (Betrag kleiner als 2^{-1022} beim Typ **double**, siehe Abschnitt 3.3.5.1) benötigt wird.

Dass die definierte Indifferenzrelation nicht transitiv ist, muss hingenommen werden. Für drei **double**-Werte *a*, *b* und *c* kann also das folgende Ergebnismuster auftreten:

- *a* numerisch identisch mit *b*
- *b* numerisch identisch mit *c*
- *a* **nicht** numerisch identisch mit *c*

Für den Vergleich einer **double**-Zahl *a* mit dem Wert Null ist eine Schwelle für die *absolute* Abweichung (statt der relativen) sinnvoll, z.B.:

$$|a| < 1{,}0 \cdot 10^{-14}$$

Die besprochenen Genauigkeitsprobleme sind auch bei den Grenzfällen von *einseitigen* Vergleichen (<, <=, >, >=) relevant.

Bei vielen naturwissenschaftlichen oder technischen Problemen ist es generell wenig sinnvoll, zwei Größen auf exakte Übereinstimmung zu testen, weil z.B. schon aufgrund von Messungenauigkeiten eine Abweichung von der theoretischen Identität zu erwarten ist. Bei Verwendung einer anwendungslogisch gebotenen Unterschiedsschwelle dürften die technischen Beschränkungen der Gleitkommatypen keine große Rolle mehr spielen. Präzisere Aussagen zur Computer-Arithmetik finden sich z.B. bei Müller (2004) oder Strey (2003).

3.5.5 Logische Operatoren

Aus dem Abschnitt 3.5.3 wissen wir, dass jeder Vergleich (z.B. `arg > 0`) bereits ein logischer Ausdruck, also die Werte **true** und **false** annehmen kann. Durch Anwendung der logischen Operatoren auf bereits vorhandene logische Ausdrücke kann man neue, komplexere logische Ausdrücke erstellen. Die Wirkungsweise der logischen Operatoren wird in **Wahrheitstafeln** beschrieben (*La1* und *La2* seien logische Ausdrücke):

Argument *La1*	**Negation** **!***La1*
true	**false**
false	**true**

Argument 1 *La1*	**Argument 2** *La2*	**Logisches UND** *La1* **&&** *La2* *La1* **&** *La2*	**Logisches ODER** *La1* **||** *La2* *La1* **|** *La2*	**Exklusives ODER** *La1* **^** *La2*
true	**true**	**true**	**true**	**false**
true	**false**	**false**	**true**	**true**
false	**true**	**false**	**true**	**true**
false	**false**	**false**	**false**	**false**

Es folgt eine Tabelle mit wichtigen Erläuterungen und Beispielen zu den logischen Operatoren:

Operator	Bedeutung	Beispiel	
		Programmfragment	**Ausgabe**
!*La1*	**Negation** Der Wahrheitswert wird umgekehrt.	`bool erg = true;` `Console.WriteLine(!erg);`	`False`
La1 && *La2*	**Logisches UND (mit bedingter Auswertung)** *La1* && *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Ist *La1* falsch, wird *La2* **nicht** ausgewertet.	`int i = 3;` `bool erg = false && i++ > 3;` `Console.WriteLine(erg + "\n" + i);` `erg = true && i++ > 3;` `Console.WriteLine(erg + "\n" + i);`	`False` `3` `False` `4`
La1 & *La2*	**Logisches UND (mit unbedingter Auswertung)** *La1* & *La2* ist genau dann wahr, wenn beide Argumente wahr sind. Es werden auf jeden Fall **beide** Ausdrücke ausgewertet.	`int i = 3;` `bool erg = false & i++ > 3;` `Console.WriteLine(erg + "\n" + i);`	`False` `4`
La1 || *La2*	**Logisches ODER (mit bedingter Auswertung)** *La1* || *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Ist *La1* wahr, wird *La2* **nicht** ausgewertet.	`int i = 3;` `bool erg = true || i++ == 3;` `Console.WriteLine(erg + "\n" + i);` `erg = false || i++ == 3;` `Console.WriteLine(erg + "\n" + i);`	`True` `3` `True` `4`
La1 | *La2*	**Logisches ODER (mit unbedingter Auswertung)** *La1* | *La2* ist genau dann wahr, wenn mindestens ein Argument wahr ist. Es werden auf jeden Fall **beide** Ausdrücke ausgewertet.	`int i = 3;` `bool erg = true | i++ > 3;` `Console.WriteLine(erg + "\n" + i);`	`True` `4`
La1 ^ *La2*	**Exklusives logisches ODER** *La1* ^ *La2* ist genau dann wahr, wenn genau *ein* Argument wahr ist, wenn also die Argumente verschiedene Wahrheitswerte haben.	`Console.WriteLine(true ^ true);`	`False`

Der Unterschied zwischen den beiden UND-Operatoren **&&** und **&** bzw. zwischen den beiden ODER-Operatoren **||** und **|** ist für Einsteiger vielleicht etwas unklar, weil man spontan den nicht ausgewerteten logischen Ausdrücken keine Bedeutung beimisst. Allerdings ist es in C# nicht ungewöhnlich, „Nebeneffekte“ in einen logischen Ausdruck einzubauen, z.B.:

```
b & (i++ > 3)
```

Hier erhöht der Postinkrementoperator beim Auswerten des rechten **&**-Arguments den Wert der Variablen `i`. Eine solche Auswertung wird jedoch in der folgenden Variante des Beispiels (mit **&&**-Operator) unterlassen, wenn bereits nach Auswertung des linken UND-Arguments das Gesamtergebnis **false** feststeht:

```
b && (i++ > 3)
```

Man spricht hier von einer *Kurzschlussauswertung*. Das vom Programmierer nicht erwartete Ausbleiben einer Auswertung (z.B. bei „`i++`“) kann erhebliche Auswertungen auf ein Programm haben.

Mit der Entscheidung, grundsätzlich die unbedingte Operatorvariante zu verwenden, nimmt man (mehr oder weniger relevante) Leistungseinbußen in Kauf. Eher empfehlenswert ist der Verzicht auf Nebeneffekt-Konstruktionen im Zusammenhang mit bedingt arbeitenden Operatoren.

Dank der *bedingten* Auswertung des Operators **&&** kann man sich im rechten Operanden darauf verlassen, dass der linke Ausdruck den Wert **true** besitzt, was im folgenden Beispiel ausgenutzt wird. Dort prüft der linke Operand die Existenz und der rechte Operand die Länge einer Zeichenfolge:

```
str != null && str.Length < 10
```

Wenn die Referenzvariable `str` vom Typ der Klasse **String** keine Objektadresse enthält, darf der rechte Ausdruck nicht ausgewertet werden, weil eine Längenanfrage (per Eigenschaftszugriff) an ein nicht existentes Objekt zu einem Laufzeitfehler führen würde.

Wie der Tabelle auf Seite 132 zu entnehmen ist, unterscheiden sich die beiden UND-Operatoren **&&** und **&** bzw. die beiden ODER-Operatoren || und | auch hinsichtlich der Bindungskraft auf Operanden (Auswertungspriorität).

Um die Verwirrung noch ein wenig zu steigern, werden die Zeichen **&** und | auch für *bitorientierte* Operatoren verwendet (siehe Abschnitt 3.5.6). Diese Operatoren erwarten zwei *integrale* Argumente (z.B. Datentyp **int**), während die logischen Operatoren den Datentyp **bool** voraussetzen. Folglich kann der Compiler erkennen, ob ein logischer oder ein bitorientierter Operator gemeint ist.

3.5.6 Vertiefung: Bitorientierte Operatoren

Über unseren momentanen Bedarf hinausgehend bietet C# einige Operatoren zur bitweisen Analyse und Manipulation von Variableninhalten. Statt einer systematischen Darstellung der verschiedenen Operatoren (siehe C# - Sprachspezifikation in Microsoft 2012) beschränken wir uns auf ein Beispielprogramm, das zudem nützliche Einblicke in die Speicherung von **char**-Daten im Arbeitsspeicher eines Computers erlaubt. Allerdings sind Beispiel und zugehörige Erläuterungen mit einigen technischen Details belastet. Wenn Ihnen der Sinn momentan nicht danach steht, können Sie den aktuellen Abschnitt ohne Sorge um den weiteren Kurserfolg an dieser Stelle verlassen.

Das Programm `CharBits` liefert die Unicode-Kodierung zu einem vom Benutzer erfragten Zeichen. Dabei kommt die statische Methode **ToChar()** der Klasse **Convert** aus dem Namensraum **System** zum Einsatz. Außerdem wird mit der **for**-Schleife eine Wiederholungsanweisung verwendet, die erst in Abschnitt 3.7.3.1 offiziell vorgestellt wird. Im Beispiel startet die **int**-wertige Indexvariable `i` mit dem Wert 15, der am Ende jedes Schleifendurchgangs um Eins dekrementiert wird (`i--`). Ob es zum nächsten Schleifendurchgang kommt, hängt von der Fortsetzungsbedingung ab (`i >= 0`):

Quellcode	Ausgabe
`using System;` `class CharBits {` `  static void Main() {` `  char cbit;` `  Console.Write("Zeichen: ");` `  cbit = Convert.ToChar(Console.ReadLine());` `  Console.Write("Unicode: ");` `  for (int i = 15; i >= 0; i--) {` `  if ((1 << i & cbit) != 0)` `  Console.Write('1');` `  else` `  Console.Write('0');` `  }` `  }` `}`	`Zeichen: x` `Unicode: 0000000001111000`

Der **Links-Shift-Operator** << im Ausdruck:

```
1 << i
```

verschiebt die Bits in der binären Repräsentation der Ganzzahl Eins um *i* Stellen nach links, wobei am linken Rand `i` Stellen verworfen werden und auf der rechten Seite `i` Nullen nachrücken. Von den 32 Bits, die ein **int**-Wert insgesamt belegt (siehe Abschnitt 3.3.4), interessieren im Augenblick nur die rechten 16. Bei der Eins erhalten wir:

```
0000000000000001
```

Im 10. Schleifendurchgang (`i = 6`) geht dieses Muster z.B. über in:

```
0000000001000000
```

Nach dem Links-Shift- kommt der **bitweise UND-Operator** zum Einsatz:

```
1 << i & cbit
```

Das Operatorzeichen **&** wird leider in doppelter Bedeutung verwendet: Wenn beide Argumente vom Typ **bool** sind, wird **&** als *logischer* Operator interpretiert (siehe Abschnitt 3.5.5). Sind jedoch (wie im vorliegenden Fall) beide Argumente von integralem Typ, was auch für den Typ **char** zutrifft, dann wird **&** als UND-Operator für Bits aufgefasst. Er erzeugt dann ein Bitmuster, das genau dann an der Stelle *k* eine Eins enthält, wenn *beide* Argumentmuster an dieser Stelle eine Eins besitzen. Hat bei einem Programmablauf die **char**-Variable `cbit` vom Benutzer den Wert `'x'` erhalten, ist das Unicode-Bitmuster dieses Zeichens

```
0000000001111000
```

beteiligt, und `1 << i & cbit` liefert z.B. bei `i = 6` das Muster:

```
0000000001000000
```

Das von `1 << i & cbit` erzeugte Bitmuster hat den Typ **int** und kann daher mit der Null verglichen werden:

```
(1 << i & cbit) != 0
```

Dieser logische Ausdruck wird bei einem Schleifendurchgang genau dann wahr, wenn das zum aktuellen `i`-Wert korrespondierende Bit in der Binärdarstellung des untersuchten Zeichens den Wert Eins hat.

3.5.7 Typumwandlung (Casting) bei elementaren Datentypen

3.5.7.1 Implizite Typanpassung

Bei der Auswertung des Ausdrucks

```
2.3 / 7
```

trifft der Divisionsoperator auf ein **double**- und ein **int**-Argument. Der C# - Compiler kennt 7 vordefinierte Divisionsoperatoren (siehe C# - Sprachspezifikation in Microsoft 2012, Abschnitt 7.8.2):

- Ganzzahldivision
  - **int operator /(int x, int y)**
  - **uint operator /(uint x, uint y)**
  - **long operator /(long x, long y)**
  - **ulong operator /(ulong x, ulong y)**
- Binäre Fließkommadivision
  - **float operator /(float x, float y)**
  - **double operator /(double x, double y)**
- Dezimale Fließkommadivision
  - **decimal operator /(decimal x, decimal y)**

Wenn bei einem Argument oder bei beiden Argumenten kein unterstützter Datentyp vorliegt, findet nach Möglichkeit eine automatische (implizite) Typanpassung „nach oben“ statt. Im Beispiel wird das **int**-Argument in den Datentyp **double** gewandelt und eine binäre Fließkommadivision durchgeführt.

In vergleichbaren Situationen (z.B. bei Wertzuweisungen) kommt es automatisch zu den folgenden **erweiternden Typanpassungen**:

Der Typ …	**wird nach Bedarf automatisch konvertiert in:**
sbyte	short, int, long, float, double, decimal
byte	short, ushort, int, uint, long, ulong, float, double, decimal
short	int, long, float, double, decimal
ushort	int, uint, long, ulong, float, double, decimal
int	long, float, double, decimal
uint	long, ulong, float, double, decimal
long	float, double, decimal
char	ushort, int, uint, long, ulong, float, double, decimal
float	double
ulong	float, double, decimal

Bei den Konvertierungen von **int**, **uint** oder **long** in **float** sowie von **long** in **double** kann es zu einem Verlust an Genauigkeit kommen, z.B.:

Quellcode

```
using System;
class Prog {
   static void Main() {
      long i = 9223372036854775313;
      double d = i;
      Console.WriteLine(i);
      Console.WriteLine("{0,20:f2}", d);
   }
}
```

Ausgabe

```
9223372036854775313
9223372036854780000,00
```

Eine Abweichung von 4687 (z.B. Euro oder Meter) kann durchaus unerfreuliche Konsequenzen haben.

Weil eine **char**-Variable die Unicode-Nummer eines Zeichens speichert, macht ihre Konvertierung in numerische Typen kein Problem, z.B.:

Quellcode

```
class Prog {
  static void Main() {
    System.Console.WriteLine("x/2 \t= " + 'x' / 2);
    System.Console.WriteLine("x*0,27 \t= " + 'x' * 0.27);
  }
}
```

Ausgabe

```
x/2     = 60
x*0,27  = 32,4
```

3.5.7.2 Explizite Typkonvertierung

Gelegentlich gibt es gute Gründe, über den so genannten **Casting-Operator** eine *explizite* Typumwandlung zu erzwingen. Im nächsten Beispielprogramm wird mit

```
(int)'x'
```

die **int**-erpretation des (aus Abschnitt 3.5.6 bekannten) Bitmusters zum kleinen „x“ ausgegeben, damit Sie nachvollziehen können, warum das Beispielprogramm im vorigen Abschnitt beim „Halbieren“ dieses Zeichens auf den Wert 60 kam:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine((int)'x');` `  double a = 3.7615926;` `  int i = (int)a;` `  Console.WriteLine(i);` `  Console.WriteLine((int)(a + 0.5));` `  a = 214748364852.13;` `  Console.WriteLine((int)a);` `  }` `}`	`120` `3` `4` `-2147483648`

Manchmal ist es erforderlich, einen Gleitkommawert in eine Ganzzahl zu wandeln, z.B. weil bei einem Methodenaufruf ein ganzzahliger Datentyp benötigt wird. Dabei werden die Nachkommastellen abgeschnitten. Soll stattdessen ein Runden stattfinden, addiert man vor der Typkonvertierung 0,5 zum Gleitkommawert.

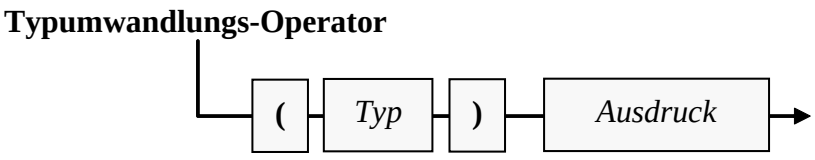
Es ist auf jeden Fall zu beachten, dass eine **einschränkende Konvertierung** stattfindet, so dass die zu erwartenden Gleitkommazahlen im Wertebereich des Ganzzahltyps liegen müssen. Wie die letzte Ausgabe zeigt, sind kapitale Programmierfehler möglich, wenn die Wertebereiche der beteiligten Variablen bzw. Datentypen nicht beachtet werden und bei der Zielvariablen ein Überlauf auftritt (vgl. Abschnitt 3.6.1). So soll die Explosion der europäischen Weltraumrakete Ariane-5 am 4. Juni 1996 (Schaden: ca. 500 Millionen Dollar)

durch die Konvertierung eines **double**-Werts (mögliches Maximum: $1{,}7976931348623157 \cdot 10^{308}$) in einen **short**-Wert (mögliches Maximum: $2^{15}-1 = 32767$) verursacht worden sein. Es zeigt sich, dass profunde Kenntnisse über elementare Sprachelemente unverzichtbar sind für eine erfolgreiche Raketenforschung und -entwicklung.

Später wird sich zeigen, dass auch zwischen Referenztypen gelegentlich eine explizite Wandlung erforderlich ist.

Welche expliziten Typkonvertierungen in C# erlaubt sind, ist der C# - Sprachspezifikation zu entnehmen (Microsoft 2012, Abschnitt 6.2).

Die C# - Syntax zur expliziten Typumwandlung:



Am Rand soll noch erwähnt werden, dass die Wandlung in einen Ganzzahltyp keine sinnvolle Technik ist, um die Nachkommastellen in einem Gleitkommawert zu entfernen oder zu extrahieren.

Dazu kann man z.B. den Modulo-Operator verwenden (vgl. Abschnitt 3.5.1), ohne ein Wertebereichsproblem befürchten zu müssen, z.B.:[1]

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double a = 2147483648.13, b;` `  int i = (int)a;` `  b = a - a % 1;` `  Console.WriteLine("{0}\n{1}\n{2}", a, i, b);` `  }` `}`	`2147483648,13` `-2147483648` `2147483648`

3.5.8 Zuweisungsoperatoren

Bei den ersten Erläuterungen zur Wertzuweisung (vgl. Abschnitt 3.3.6) blieb aus didaktischen Gründen unerwähnt, dass eine Wertzuweisung einen *Ausdruck* darstellt, dass wir es also mit dem binären (zweistelligen) Operator „=“ zu tun haben, für den folgende Regeln gelten:

- Auf der linken Seite muss eine Variable oder eine Eigenschaft stehen.
- Auf der rechten Seite muss ein Ausdruck mit kompatiblem Typ stehen.
- Der zugewiesene Wert stellt auch den Ergebniswert des Ausdrucks dar.

Wie beim Inkrement- bzw. Dekrementoperator sind auch beim Zuweisungsoperator zwei Effekte zu unterscheiden:

- Die als linkes Argument fungierende Variable oder Eigenschaft erhält einen neuen Wert.
- Es wird ein Wert für den Ausdruck produziert.

In folgendem Beispiel fungiert ein Zuweisungsausdruck als Parameter für einen **WriteLine()** - Methodenaufruf:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int ivar = 13;` `  Console.WriteLine(ivar = 4711);` `  Console.WriteLine(ivar);` `  }` `}`	`4711` `4711`

Beim Auswerten des Ausdrucks `ivar = 4711` entsteht der an **WriteLine()** zu übergebende Wert, *und* die Variable `ivar` wird verändert.

Selbstverständlich kann eine Zuweisung auch als Operand in einen übergeordneten Ausdruck integriert werden, z.B.:

[1] Der (gerundete) ganzzahlige Anteil eines **double**-Wertes lässt sich auch über die statische Methode **Round()** bzw. **Truncate()** aus der Klasse **Math** ermitteln.

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = 2, j = 4;` `  i = j = j * i;` `  Console.WriteLine(i + "\n" + j);` `  }` `}`	`8` `8`

Beim mehrfachen Auftreten des Zuweisungsoperators ist seine **Rechts-Assoziativität** relevant (vgl. Tabelle in Abschnitt 3.5.10). Sie bewirkt, dass die Anweisung

```
i = j = j * i;
```

folgendermaßen ausgeführt wird:

- Weil der Multiplikationsoperator eine höhere Bindungskraft besitzt als der Zuweisungsoperator, wird zuerst der Ausdruck `j * i` ausgewertet, was zum Zwischenergebnis 8 (mit Datentyp **int**) führt.
- Die Rechts-Assoziativität des Zuweisungsoperators führt zu der anschließend durch runde Klammern hervorgehobenen Zuordnung der Operanden:

  ```
  i = (j = 8)
  ```

  Folglich wird nach der Multiplikation die *rechte* Zuweisung ausgeführt. Der folgende Ausdruck mit Wert 8 und Typ **int**

  ```
  j = 8
  ```

  verschafft der Variablen `j` einen neuen Wert.
- In der zweiten Zuweisung (bei Betrachtung von rechts nach links) wird der Wert des Ausdrucks `j = 8` an die Variable `i` übergeben.

Ausdrücke der Art

```
i = j = k;
```

stammen übrigens nicht aus einem Kuriositätenkabinett, sondern sind in C# - Programmen oft anzutreffen, weil im Vergleich zur Alternative

```
j = k;
i = k;
```

Schreibaufwand gespart wird.

Wie wir seit Abschnitt 3.3.6 wissen, stellt ein Zuweisungsausdruck bereits eine vollständige **Anweisung** dar, sobald man ein Semikolon dahinter setzt. Dies gilt auch für die die Prä- und Postinkrementausdrücke (vgl. Abschnitt 3.5.1) sowie für Methodenaufrufe, jedoch *nicht* für die anderen Ausdrücke, die in Abschnitt 3.5 vorgestellt werden.

Für die häufig benötigten Zuweisungen nach dem Muster

```
j = j * i;
```

(eine Variable erhält einen neuen Wert, an dessen Konstruktion sie selbst mitwirkt), bietet C# spezielle Zuweisungsoperatoren für Schreibfaule, die gelegentlich auch als **Aktualisierungsoperatoren** bezeichnet werden. In der folgenden Tabelle steht *Var* für eine numerische Variable und *Expr* für einen typkompatiblen Ausdruck:

Operator	Bedeutung	Beispiel	
		Programmfragment	Neuer Wert von `i`
Var `+=` *Expr*	*Var* erhält den neuen Wert *Var* + *Expr*.	`int i = 2;` `i += 3;`	`5`
Var `-=` *Expr*	*Var* erhält den neuen Wert *Var* − *Expr*.	`int i = 10, j = 3;` `i -= j * j;`	`1`
Var `*=` *Expr*	*Var* erhält den neuen Wert *Var* * *Expr*.	`int i = 2;` `i *= 5;`	`10`
Var `/=` *Expr*	*Var* erhält den neuen Wert *Var* / *Expr*.	`int i = 10;` `i /= 5;`	`2`
Var `%=` *Expr*	*Var* erhält den neuen Wert *Var* % *Expr*.	`int i = 10;` `i %= 5;`	`0`

Eine Marginalie: Während für zwei **byte**-Variablen

```
byte b1 = 1, b2 = 2;
```

die folgende Zuweisung

```
b1 = b1 + b2;
```

verboten ist, weil der Ausdruck `(b1 + b2)` den Typ **int** besitzt (vgl. Abschnitt 3.5.1), akzeptiert der Compiler den äquivalenten Ausdruck mit Aktualisierungsoperator:

```
b1 += b2;
```

Allerdings soll diese Randbemerkung nicht als Geheimtipp für cleveres Programmieren verstanden werden, weil sich das Überlaufrisiko (vgl. Abschnitt 3.6.1) erhöht:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  byte b1 = 200, b2 = 200;` `  b1 += b2;` `  Console.WriteLine(b1);` `  }` `}`	`144`

Insgesamt ist es keine schlechte Idee, auf die Aktualisierungsoperatoren zu verzichten. In fremden Programmen (erstellt von schreibfaulen Kollegen) muss man aber mit ihnen rechnen.

3.5.9 Konditionaloperator

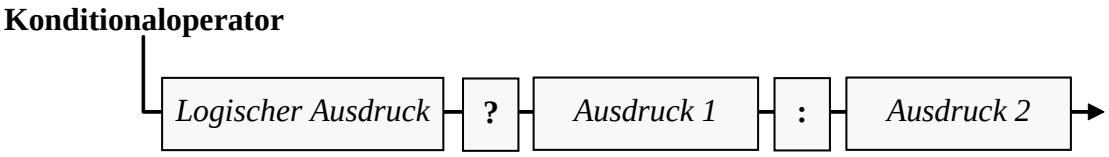
Der **Konditionaloperator** erlaubt eine sehr kompakte Schreibweise, wenn beim neuen Wert einer Zielvariablen bedingungsabhängig zwischen zwei Ausdrücken zu entscheiden ist, z.B.

$$i = \begin{cases} i + j & \text{falls } k > 0 \\ i - j & \text{sonst} \end{cases}$$

In C# ist für diese Zuweisung mit Fallunterscheidung nur eine einzige Zeile erforderlich:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = 2, j = 1, k = 7;` `  i = k>0 ? i+j : i-j;` `  Console.WriteLine(i);` `  }` `}`	`3`

Eine Besonderheit des Konditionaloperators besteht darin, dass er *drei* Argumente verarbeitet, welche durch die Zeichen **?** und **:** getrennt werden:



Ist der logische Ausdruck *wahr*, liefert der Konditionaloperator den Wert von *Ausdruck 1*, andernfalls den Wert von *Ausdruck 2*.

Die Frage nach dem Typ eines Konditionalausdrucks ist etwas knifflig, und in der C# - Sprachspezifikation werden etliche Fälle unterschieden (Microsoft 2012, Abschnitt 7.14). Es liegt an Ihnen, sich auf den einfachsten und wichtigsten Fall zu beschränken: Wenn der zweite und der dritte Operand denselben Typ haben, ist dies auch der Typ des Konditionalausdrucks.

3.5.10 Auswertungsreihenfolge

Bisher haben wir zusammengesetzte Ausdrücke mit *mehreren* Operatoren und das damit verbundene Problem der Auswertungs*reihenfolge* nach Möglichkeit gemieden. Wie gleich deutlich wird, sind für Schwierigkeiten und Fehlergefahren hauptverantwortlich:

- Komplexität des Ausdrucks (Anzahl der Operatoren, Schachtelungstiefe)
- Operatoren mit Nebeneffekten

Um Problemen aus dem Weg zu gehen, sollte man also übertriebene Komplexität vermeiden und auf Nebeneffekte weitgehend verzichten.

3.5.10.1 Regeln

Nun werden die Regeln vorgestellt, nach denen der C# - Compiler einen Ausdruck mit mehreren Operatoren auswertet.

1) Runde Klammern

Wenn aus den anschließend erläuterten Regeln zur Bindungskraft und Assoziativität der beteiligten Operatoren nicht die gewünschte Operandenzuordnung resultiert, greift man mit runden Klammern steuernd ein, wobei auch eine Schachtelung erlaubt ist. Durch Klammern werden Terme zu *einem* Operanden zusammengefasst, so dass die *internen* Operationen ausgeführt sind, bevor der Klammerausdruck von einem *externen* Operator verarbeitet wird.

2) Bindungskraft

Bei konkurrierenden Operatoren entscheidet die Bindungskraft (siehe Tabelle in Abschnitt 3.5.10.2) darüber, wie die Operanden den Operatoren zugeordnet werden. Mit `a`, `b` und `c` als Platzhaltern für Operanden (z.B. Zahlen oder Variablen) wird

```
a + b * c
```

nach der Regel „*Punktrechnung geht vor Strichrechnung*“ interpretiert als

```
a + (b * c)
```

Die höhere Bindungskraft des Postinkrementoperators führt im folgenden Beispiel

```
b * b++
```

zur Operandenzuordnung

```
b * (b++)
```

Der Postinkrementoperator hat seinen linken Nachbarn als Operanden an sich gebunden, und der resultierende Teilausdruck wird zum rechten Operanden der Multiplikation. Nach einer gleich vorzustellenden Regel wird in C# der linke Operand eines binären Operators stets vor dem rechten ausgewertet. Damit bleibt der Nebeneffekt des rechten Multiplikationsoperanden ohne Einfluss auf den linken Operanden, und wir erhalten als Wert des Ausdrucks b^2. Außerdem wird die Variable `b` inkrementiert.

3) Assoziativität (Orientierung)

Stehen mehrere Operatoren gleicher Bindungskraft zur Auswertung an, dann entscheidet deren Assoziativität (Orientierung) über die Zuordnung der Operanden:

- Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links-assoziativ*. Z.B. wird

  ```
  x - y - z
  ```

  ausgewertet als

  ```
  (x - y) - z
  ```

- Die Zuweisungsoperatoren sind *rechts-assoziativ*. Z.B. wird

  ```
  a += b -= c = d
  ```

  ausgewertet als

  ```
  a += (b -= (c = d))
  ```

In C# ist dafür gesorgt, dass Operatoren mit gleicher Bindungskraft stets auch die gleiche Assoziativität besitzen, z.B. die im letzten Beispiel enthaltenen Operatoren `+=`, `-=` und `=`.

Für manche Operationen gilt das mathematische Assoziativitätsgesetz, so dass die Reihenfolge der Auswertung irrelevant ist, z.B.:

$$(3 + 2) + 1 = 6 \quad = \quad 3 + (2 + 1)$$

Anderen Operationen fehlt diese Eigenschaft, z.B.:

$$(3 - 2) - 1 = 0 \quad \neq \quad 3 - (2 - 1) = 2$$

Während sich die Addition und die Multiplikation von *Ganzzahltypen* in C# tatsächlich assoziativ verhalten, gilt das aus EDV-technischen Gründen *nicht* für die die Addition und die Multiplikation von *Gleitkommatypen*.

4) Links vor Rechts bei der Auswertung der Argumente eines binären Operators

Bevor ein Operator ausgeführt werden kann, müssen erst seine Argumente (Operanden) ausgewertet werden. Bei jedem binären Operator kann man sich in C# darauf verlassen, dass erst der linke Operand ausgewertet wird, dann der rechte. Kommt es bei der Auswertung des linken Operanden zu einem Ausnahmefehler (siehe unten), dann unterbleibt die Auswertung des rechten Operanden. Bei den logischen Operatoren mit bedingter Ausführung (**&&**, ||) verhindert ein bestimmter Wert des linken Operanden die Auswertung des rechten Operanden.

Das folgende, schon im Zusammenhang mit Regel 2 verwendete Beispiel zeigt, dass die hohe Bindungskraft des Postinkrementoperators (siehe Tabelle in Abschnitt 3.5.10.2) *nicht* dazu führt, dass sich der Nebeneffekt des Ausdrucks `ivar++` auf den linken Operanden der Multiplikation auswirkt:

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  int ivar = 2;` `  int erg = ivar * ivar++;` `  System.Console.WriteLine("{0} {1}", erg, ivar);` `  }` `}`	`4 3`

Die Auswertung des Ausdrucks `ivar * ivar++` verläuft so:

- Zuerst wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet:
  - Es resultiert das Ergebnis 2.
  - Die Postinkrementoperation hat einen Nebeneffekt auf die Variable `ivar`.
- Die Ausführung der Multiplikationsoperation liefert schließlich das Endergebnis 4.

Wie eine leichte Variation des letzten Beispiels zeigt, kann sich ein Nebeneffekt im *linken* Operanden einer binären Operation auf den rechten Operanden auswirken:

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  int ivar = 2;` `  int erg = ivar++ * ivar;` `  System.Console.WriteLine("{0} {1}", erg, ivar);` `  }` `}`	`6 3`

Im folgenden Beispiel mit drei Operanden (`a`, `b`, `c`) und zwei Operatoren (*, +)

```
a + b * c
```

resultiert aus der Bindungskraftregel die folgende Zuordnung der Operanden:

```
a + (b * c)
```

Zusammen mit der Links-vor-Rechts - Regel ergibt sich für die Auswertung der Operanden bzw. Ausführung der Operatoren die folgende Reihenfolge:

```
a, b, c, *, +
```

Wenn die Operanden-Platzhalter (z.B. `a`, `b`, `c`) für Zahlen oder numerische Variablen stehen, wird bei der „Auswertung“ eines Operanden lediglich sein Wert ermittelt, und die Reihenfolge der Operandenauswertung ist belanglos. Im letzten Beispiel eine falsche Auswertungsreihenfolge zu unter-

stellen (z.B. `b`, `c`, *, `a`, `+`), bleibt ungestraft. Wenn Operanden *Nebeneffekte* enthalten (Zuweisungen, In- bzw. Deinkrementoperationen oder Methodenaufrufe), ist die Reihenfolge der Auswertung jedoch relevant, und eine falsche Vermutung kann gravierende Fehler verursachen. Der Übersichtlichkeit halber sollte ein Ausdruck maximal *einen* Nebeneffekt enthalten.

Auch bei Beteiligung von rechts-assoziativen Operatoren erfolgt die Auswertung *der Operanden* von links nach rechts, so dass im folgenden Beispiel

```
a += ++a
```

diese Auswertungs- bzw. Ausführungsreihenfolge resultiert:

```
a, ++a, +=
```

Als neuer Wert von `a` resultiert `a + (a+1)`.

Die oft anzutreffende Behauptung, Klammerausdrücke würden generell zuerst ausgewertet, ist falsch, wie das folgende Beispiel zeigt:

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  int ivar = 2;` `  int erg = ivar * (++ivar + 5);` `  System.Console.WriteLine(erg);` `  }` `}`	`16`

Die Auswertung des Ausdrucks `ivar * (++ivar + 5)` verläuft so:

- Wegen der Links-vor-Rechts - Regel wird zuerst wird der linke Operand der Multiplikation ausgewertet (Ergebnis: 2)
- Dann wird der rechte Operand der Multiplikation ausgewertet (also der Klammerausdruck).
- Hier ist mit der Addition eine weitere binäre Operation vorhanden, und nach der Links-vor-Rechts - Regel wird zunächst deren linker Operand ausgewertet (Ergebnis: 3, Nebeneffekt auf die Variable ivar). Dann wird der rechte Operand der Addition ausgewertet (Ergebnis: 5). Die Ausführung der Additionsoperation liefert für den Klammerausdruck den Wert 8.
- Schließlich führt die Multiplikation zum Endergebnis 16.

3.5.10.2 Operatorentabelle

In der folgenden Tabelle sind die bisher behandelten Operatoren in absteigender Bindungskraft (Priorität) aufgelistet. Gruppen von Operatoren mit gleicher Priorität sind durch horizontale Linien voneinander abgegrenzt. In der **Operanden**-Spalte werden die zulässigen Datentypen der Argumentausdrücke mit Hilfe der folgenden Platzhalter beschrieben:

N	Ausdruck mit numerischem Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float**, **double**, **decimal**)
I	Ausdruck mit ganzzahligem (integralem) Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**)
L	logischer Ausdruck (Typ **bool**)
K	Ausdruck mit kompatiblem Datentyp
S	**String** (Zeichenfolge)
V	Variable mit kompatiblem Datentyp
V_n	Variable mit kompatiblem, numerischem Datentyp (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**, **float**, **double**, **decimal**)

Operator	Bedeutung	Operanden
Methode(*Parameter*)	Methodenaufruf	
x++, *x*--	Postinkrement bzw. -dekrement	V_n
-*x*	Vorzeichenumkehr	*N*
!	Negation	*L*
++*x*, --*x*	Präinkrement bzw. -dekrement	V_n
(*Typ*)*x*	Typumwandlung	*K*
*, /	Multiplikation, Division	*N, N*
%	Modulo (Divisionsrest)	*N, N*
+, -	Addition, Subtraktion	*N, N*
+	String-Verkettung	*S, K* oder *K, S*
<<, >>	Links- bzw. Rechts-Shift	*I, I*
>, <, >=, <=	Vergleichsoperatoren	*N, N*
==, !=	Gleichheit, Ungleichheit	*K, K*
&	Bitweises UND	*I, I*
&	Logisches UND (mit unbedingter Auswertung)	*L, L*
^	Exklusives logisches ODER	*L, L*
|	Bitweises ODER	*I, I*
|	Logisches ODER (mit unbedingter Auswertung)	*L, L*
&&	Logisches UND (mit bedingter Auswertung)	*L, L*
||	Logisches ODER (mit bedingter Auswertung)	*L, L*
? :	Konditionaloperator	*L, K, K*
=	Wertzuweisung	*V, K*
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung	V_n, *N*

Im Anhang A finden Sie eine erweiterte Version dieser Tabelle, die zusätzlich alle Operatoren enthält, die im weiteren Verlauf des Kurses noch behandelt werden.

3.5.11 Übungsaufgaben zu Abschnitt 3.5

1) Welche Werte und Datentypen besitzen die folgenden Ausdrücke?

```
6/4*2.0
(int)6/4.0*3
(int)(6/4.0*3)
3*5+8/3%4*5
```

2) Welche Werte haben die Variablen `erg1` und `erg2` am Ende des folgenden Programms?

```
using System;
class Prog {
  static void Main() {
    int i = 2, j = 3, erg1, erg2;
    erg1 = (i++ == j ? 7 : 8) % 3;
    erg2 = (++i == j ? 7 : 8) % 2;
    Console.WriteLine("erg1 = {0}\nerg2 = {1}", erg1, erg2);
  }
}
```

3) Welche Wahrheitsweite erhalten in folgendem Programm die booleschen Variablen `la1` bis `la3`?

```
using System;
class Prog {
  static void Main() {
    bool la1, la2, la3;
    int i = 3;
    char c = 'n';

    la1 = 3 > 2 && 2 == 2 ^ 1 == 1;
    Console.WriteLine(la1);

    la2 = ((2 > 3) && (2 == 2)) ^ (1 == 1);
    Console.WriteLine(la2);

    la3 = !(i > 0 || c == 'j');
    Console.WriteLine(la3);
  }
}
```

Tipp: Die Negation von zusammengesetzten Ausdrücken ist etwas unangenehm. Mit Hilfe der Regeln von **DeMorgan** kommt man zu äquivalenten Ausdrücken, die leichter zu interpretieren sind:

!(*la1* && *la2*) = !*la1* || !*la2*
!(*la1* || *la2*) = !*la1* && !*la2*

4) Erstellen Sie ein Programm, das den Exponentialfunktionswert e^x (mit e als der Eulerschen Zahl) zu einer vom Benutzer eingegebenen Zahl *x* bestimmt und ausgibt, z.B.:

Eingabe: `Argument: 1`
Ausgabe: `exp(1) = 2,71828182845905`

Hinweise:

- Suchen Sie mit Hilfe der FCL-Dokumentation zur Klasse **Math** im Namensraum **System** eine passende Methode.
- Verwenden Sie zum Einlesen des Arguments eine Variante der in Abschnitt 3.4 beschriebenen Technik, wobei die **Convert**-Methode **ToInt32()** geeignet zu ersetzen ist.

5) Erstellen Sie ein Programm, das einen DM-Betrag entgegen nimmt und diesen in Euro konvertiert. In der Ausgabe sollen ganzzahlige, korrekt gerundete Werte für Euro und Cent erscheinen, z.B.:

Eingabe: `DM-Betrag: 321`
Ausgabe: `164 Euro und 12 Cent`

Umrechnungsfaktor: 1 Euro = 1,95583 DM

3.6 Über- und Unterlauf bei numerischen Datentypen

Wie Sie inzwischen wissen, haben die numerischen Datentypen jeweils einen bestimmten Wertebereich (siehe Tabelle in Abschnitt 3.3.4). Dank strenger Typisierung kann der Compiler verhindern, dass einer Variablen ein Ausdruck mit „zu großem Typ“ zugewiesen wird. So kann z.B. einer **int**-Variablen kein Wert vom Typ **long** zugewiesen werden. Bei der Auswertung eines Ausdrucks kann jedoch „unterwegs“ ein Wertebereichsproblem (z.B. ein Überlauf) auftreten. Im betroffenen Programm ist mit einem mehr oder weniger gravierenden Fehlverhalten zu rechnen, so dass Wertebereichsprobleme unbedingt vermieden bzw. rechtzeitig diagnostiziert werden müssen.

3.6.1 Überlauf bei Ganzzahltypen

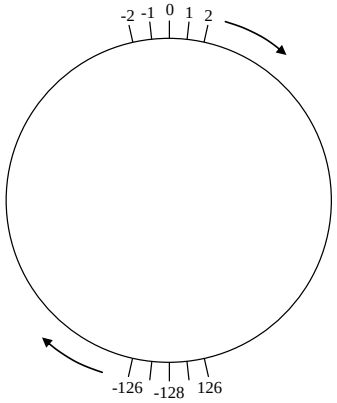
Wird z.B. zu einer ganzzahligen Variablen, die bereits den maximalen Wert ihres Typs besitzt, eine positive Zahl addiert, kann das Ergebnis nicht mehr korrekt abgespeichert werden. Ohne besondere Vorkehrungen stellt ein C# - Programm im Fall eines solchen Ganzzahlüberlaufs keinesfalls seine Tätigkeit ein (z.B. mit einem Ausnahmefehler), sondern arbeitet munter weiter. Dieses Verhalten ist beim Programmieren von Pseudozufallszahlgeneratoren willkommen, ansonsten aber eher bedenklich. Das folgende Programm

```
using System;
class Prog {
  static void Main() {
    int i = 2147483647, j = 5, k;
    k = i + j;    // Überlauf!
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

liefert ohne jede Warnung das fragwürdige Ergebnis:

```
2147483647 + 5 = -2147483644
```

Um das Auftreten eines negativen „Ergebniswerts“ zu verstehen, machen wir einen kurzen Ausflug in die Informatik. Die Werte der vorzeichenbehafteten Ganzzahltypen (mit positiven und negativen Werten) sind nach dem **Zweierkomplementprinzip** auf einem Zahlenkreis angeordnet, und nach der größten positiven Zahl beginnt der Bereich der negativen Zahlen (mit abnehmendem Betrag), z.B. beim Typ **sbyte**:



Speziell bei der Steuerung von Raketenmotoren (vgl. Abschnitt 3.5.7) ist also Vorsicht geboten, weil ansonsten das Kommando „Mr. Spock, please push the engine.“ zum heftigen Rückwärtsschub führen könnte.[1] Es zeigt sich erneut, dass eine erfolgreiche Raketenforschung und -entwicklung ohne die sichere Beherrschung der elementaren Sprachelemente kaum möglich ist.

Oft kann ein Überlauf durch Wahl eines **geeigneten Datentyps** verhindert werden. Mit den Deklarationen

```
long i = 2147483647, j = 5, k;
```

erhält man das korrekte Ergebnis, weil neben `i`, `j` und `k` nun auch der Ausdruck `i+j` den Typ **long** hat:

```
2147483647 + 5 = 2147483652
```

Im Beispiel genügt es *nicht*, für die Zielvariable `k` den beschränkten Typ **int** durch **long** zu ersetzen, weil der Überlauf beim Berechnen des Ausdrucks („unterwegs“) auftritt. Mit den Deklarationen

```
int i = 2147483647, j = 5;
long k;
```

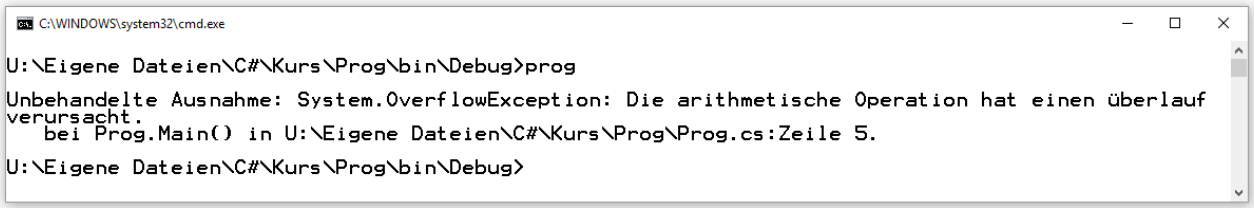
bleibt das Ergebnis falsch, denn …

- In der Anweisung
  ```
  k = i + j;
  ```
  wird zunächst der Ausdruck `i + j` berechnet.
- Weil beide Operanden vom Typ **int** sind, erhält auch der Ausdruck diesen Typ, und die Summe kann nicht korrekt berechnet bzw. zwischenspeichert werden.
- Schließlich wird der **long**-Variablen `k` das falsche Ergebnis zugewiesen.

In C# steht im Unterschied zu vielen anderen Programmiersprachen mit dem **checked**-Operator eine Möglichkeit bereit, den Überlauf bei Ganzzahlvariablen abzufangen. Eine gesicherte Variante des ursprünglichen Beispielprogramms

```
using System;
class Prog {
  static void Main() {
    int i = 2147483647, j = 5, k;
    k = checked(i + j);
    Console.WriteLine(i + " + " + j + " = " + k);
  }
}
```

rechnet nach einem Überlauf nicht mit „Zufallszahlen“ weiter, sondern bricht mit einem Ausnahmefehler ab:[2]



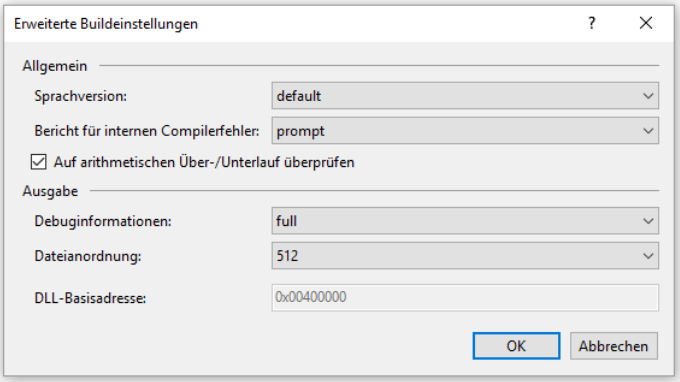
[1] Mr. Spock arbeitete jahrelang als erster Offizier auf dem Raumschiff Enterprise.

[2] Im Kapitel über Ausnahmebehandlung werden Sie lernen, Ausnahmefehler abzufangen, damit sie nicht mehr zum Abbruch des Programms führen.

Angewandt auf die Raketenforschung lässt sich sagen, dass ein abgestürztes Steuerungsprogramm und ein infolgedessen mit unveränderter Geschwindigkeit weiterfliegendes Raumschiff das kleinere Übel sind im Vergleich zu einem Antrieb, der unerwartet auf Rückwärtsschub schaltet.

An Stelle des **checked**-Operators bietet C# noch weitere Möglichkeiten, die Überlaufdiagnose für Ganzzahltypen einzuschalten:

- **checked**-Anweisung
  Man kann die Überwachung für einen kompletten Anweisungsblock einschalten.
  Beispiel:
  ```
  checked {
    . . .
      k = i + j;
    . . .
  }
  ```
- **checked** - Compiler-Option
  Man kann die Überwachung per Compiler-Option für das gesamte erstellte Assembly einschalten. Die Option wirkt sich auf alle Ganzzahlarithmetik-Operationen aus, die sich nicht im Gültigkeitsbereich eines **unchecked**-Operators (siehe unten) befinden, z.B.:
  ```
  csc /checked Prog.cs
  ```
  Im Visual Studio vereinbart man diese Compiler-Option für ein Projekt folgendermaßen:
  - Menübefehl **Projekt > Eigenschaften**
  - Registerkarte **Erstellen**
  - Schalter **Erweitert**
  - Kontrollkästchen **Auf arithmetischen Über-/Unterlauf überprüfen**



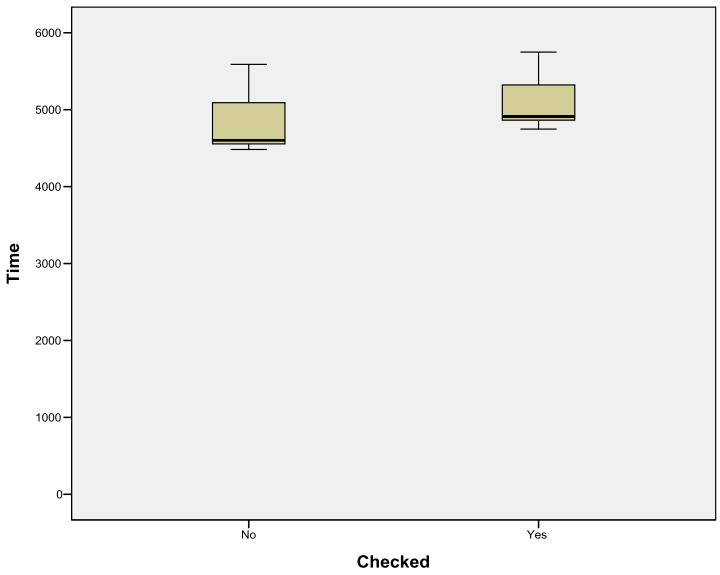
Gegen die nahe liegende Entscheidung, die Überlaufdiagnose für Ganzzahltypen *generell* einzuschalten, spricht nur der zeitliche Mehraufwand. Bei der Aufgabe, mit 0 beginnend 2147483647 mal den **int**-Wert 1 zu addieren,

$$\sum_{i=1}^{2147483647} 1$$

ergaben sich basierend auf jeweils 40 Tests mit schwankenden Einzelergebnissen die folgenden mittleren Laufzeiten in Millisekunden:

- Ohne Überlaufkontrolle: 4818,80
- Mit Überlaufkontrolle: 5087,47

In der folgenden Abbildung sind die Verteilungen der Laufzeiten für die beiden Bedingungen durch Boxplots dargestellt:



Aus diesem Ergebnisbild ergibt sich die Empfehlung, die Überlaufkontrolle für Ganzzahloperationen einzuschalten. Das Ergebnis kann allerdings nur eingeschränkt auf andere Programme generalisiert werden.

Welches Verhalten ein Programm beim Überschreiten eines Ganzzahlwertebereichs zeigt, liegt jedoch eindeutig fest:

- *Mit* Überlaufkontrolle kommt es zum Laufzeitfehler, der entweder abgefangen wird oder das Programm zum Stillstand bringt.
- *Ohne* Überlaufkontrolle agiert das Programm chaotisch, gerät z.B. in eine Endlosschleife oder bringt ein Fluggerät zum Absturz.

Der Vollständigkeit halber soll an dieser Stelle noch der **unchecked**-Operator erwähnt werden, mit dem sich Kontrollfunktionen des Compilers abschalten lassen, was für sehr spezielle Zwecke (z.B. zum Programmieren von Pseudozufallszahlengeneratoren) sinnvoll sein kann. Auch ohne **checked**-Instruktion verhindert der Compiler z.B., dass eine Ganzzahl > 2147483647 in den Typ **int** gewandelt wird. Unsere Entwicklungsumgebung verrät, wie man diese Überwachung abschalten kann:

Wird der „Tipp“ umgesetzt, resultiert ein fehlerhafter Wert:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int i = unchecked((int) 2147483652);` `  Console.WriteLine(i);` `  }` `}`	`-2147483644`

3.6.2 Unendliche und undefinierte Werte bei den Typen float und double

Auch bei den binären Gleitkommatypen **float** und **double** kann ein Überlauf auftreten, obwohl die unterstützten Wertebereiche hier weit größer sind. Dabei kommt es aber weder zu einem sinnlosen

Zufallswert noch zu einem Ausnahmefehler, sondern zu den speziellen Gleitkommawerten **+/- Unendlich**, mit denen anschließend sogar weitergerechnet werden kann. Das folgende Programm:

```
using System;
class Prog {
  static void Main() {
    double bigd = Double.MaxValue;
    Console.WriteLine("Double.MaxValue =\t" + bigd);
    bigd = Double.MaxValue * 10.0;
    Console.WriteLine("Double.MaxValue * 10 =\t" + bigd);
    Console.WriteLine("Unendl. + 10 =\t\t" + (bigd + 10));
    Console.WriteLine("Unendl. * -13 =\t\t" + (bigd * -13));
    Console.WriteLine("13.0/0.0 =\t\t" + (13.0 / 0.0));
  }
}
```

liefert im .NET-Framework 3.5 die Ausgabe:[1]

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  +unendlich
Unendl. + 10 =          +unendlich
Unendl. * -13 =         -unendlich
13.0/0.0 =              +unendlich
```

Mit Hilfe der Unendlich-Werte „gelingt“ offenbar bei der Gleitkommaarithmetik sogar die Division durch Null, während bei der Ganzzahlarithmetik ein solcher Versuch zu einem Laufzeitfehler (aus der Klasse **DevideByZeroException**) führt.

Bei den folgenden „Berechnungen“

$$\text{Unendlich} - \text{Unendlich}$$

$$\frac{\text{Unendlich}}{\text{Unendlich}}$$

$$\text{Unendlich} \cdot 0$$

$$\frac{0}{0}$$

resultiert der spezielle Gleitkommawert **NaN** (*Not a Number*), wie das folgende Programm zeigt:

---

[1] In den.NET - Versionen ab 4.0 sieht die Ausgabe desselben Programms leider anders aus:

```
Double.MaxValue =       1,79769313486232E+308
Double.MaxValue * 10 =  8
Unendl. + 10 =          8
Unendl. * -13 =         -8
13.0/0.0 =              8
```

Vermutlich wird der Fehler durch die Methode **ToString()** der Type **Double** verursacht. Eventuell soll die 8 das mathematische Symbol für Unendlich (∞) vertreten, was fast gelingt (bis auf eine Drehung um 90°). Wer den Effekt der .NET-Version nachprüfen möchte, muss lediglich nach

Projekt > Eigenschaften > Anwendung

das **Zielframework** ändern. Der Fehler ist eher ästhetischer Natur, weil die im weiteren Verlauf des Abschnitts vorgestellten statischen **Double**-Methoden **IsPositiveInfinity()** und **IsPositiveInfinity()** korrekt arbeiten.

```
using System;
class Prog {
  static void Main() {
    double bigd;
    bigd = Double.MaxValue * 10.0;
    Console.WriteLine("Unendlich – Unendlich =\t" + (bigd - bigd));
     Console.WriteLine("Unendlich / Unendlich =\t" + (bigd / bigd));
     Console.WriteLine("Unendlich * 0.0 =\t" + (bigd * 0.0));
    Console.WriteLine("0.0 / 0.0 =\t\t" + (0.0 / 0.0));
  }
}
```

Es liefert im .NET-Framework 3.5 die Ausgabe:[1]

```
Unendlich - Unendlich = n. def.
Unendlich / Unendlich = n. def.
Unendlich * 0.0 =       n. def.
0.0 / 0.0 =             n. def.
```

Zu den letzten Beispielprogrammen ist noch anzumerken, dass man über das öffentliche Feld **MaxValue** der Struktur[2] **Double** aus dem Namensraum **System** den größten Wert in Erfahrung bringt, der in einer **double**-Variablen gespeichert werden kann.

Über die statischen **Double**-Methoden

- **public static bool IsPositiveInfinity(double** *d***)**
- **public static bool IsNegativeInfinity(double** *d***)**
- **public static bool IsNaN(double** *d***)**

mit einem Parameter vom Typ **double** und einer Rückgabe vom Typ **bool** lässt sich für einen **double**-wertigen Ausdruck prüfen, ob er einen unendlichen oder undefinierten Wert besitzt, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double d = 0.0 / 0.0;` `  Console.WriteLine(Double.IsNaN(d));` `  }` `}`	`True`

Eben wurde eine Technik zur Beschreibung von *vorhandenen Bibliotheksmethoden* (im Unterschied zur Beschreibung von C# - Syntaxregeln) im Manuskript erstmals benutzt, die auch in der FCL-Dokumentation in ähnlicher Form Verwendung findet, z.B.:

Dabei wird die Benutzung einer vorhandenen Methode erläutert durch Angabe von:

---

[1] In den.NET - Versionen ab 4.0 sieht die Ausgabe des Programms unwesentlich anders aus:

```
Unendlich - Unendlich = NaN
Unendlich / Unendlich = NaN
Unendlich * 0.0 =       NaN
0.0 / 0.0 =             NaN
```

[2] Bei den später noch ausführlich zu behandelnden *Strukturen* handelt es sich um Werttypen mit starker Verwandtschaft zu den Klassen. Insbesondere wird sich zeigen, dass die elementaren Datentypen (z.B. **double**) auf Strukturtypen aus dem Namensraum **System** abgebildet werden (z.B. **Double**).

- Modifikatoren (z.B. für den Zugriffsschutz)
- Rückgabetyp
- Methodenname
- Parameterliste (mit Angabe der Parametertypen)

Für besonders neugierige Leser sollen abschließend noch die **float**-Darstellungen der speziellen Gleitkommawerte angegeben werden (vgl. Abschnitt 3.3.5.1):

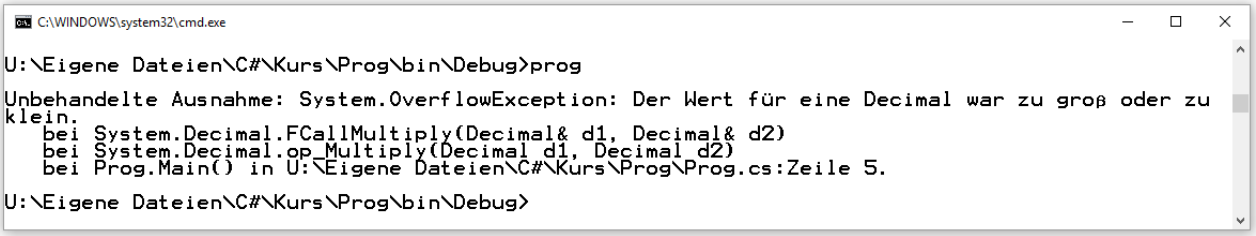
Wert	float-Darstellung		
	Vorz.	Exponent	Mantisse
+unendlich	0	11111111	00000000000000000000000
-unendlich	1	11111111	00000000000000000000000
NaN	0	11111111	10000000000000000000000

3.6.3 Überlauf beim Typ decimal

Beim Typ **decimal** wird im Fall eines Überlaufs *nicht* mit dem speziellen Wert Unendlich weitergearbeitet. Stattdessen wird ein Ausnahmefehler gemeldet, der unbehandelt zur Beendigung des Programms führt. Im Unterschied zu den Ganzzahltypen (vgl. Abschnitt 3.6.1) muss die Überlaufdiagnose *nicht* per **checked**-Operator, -Anweisung oder -Compileroption angeordnet werden. Das Beispielprogramm

```
using System;
class Prog {
  static void Main() {
    decimal d = Decimal.MaxValue;
    d = d * 10;
    Console.WriteLine(d);
  }
}
```

wird mit folgender Meldung abgebrochen:



3.6.4 Unterlauf bei den Gleitkommatypen

Bei den Gleitkommatypen **float**, **double** und **decimal** ist auch ein **Unterlauf** möglich, wobei eine Zahl mit sehr kleinem Betrag (bei **double**: $< 4{,}94065645841247 \cdot 10^{-324}$) nicht mehr dargestellt werden kann. In diesem Fall rechnet ein C# - Programm mit dem Wert 0 weiter, was in der Regel akzeptabel ist, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double smalld = Double.Epsilon;` `  Console.WriteLine(smalld);` `  smalld /= 10.0;` `  Console.WriteLine(smalld);` `  }` `}`	`4,94065645841247E-324` `0`

Das öffentliche Feld **Double.Epsilon** enthält den kleinsten Betrag, der in einer **double**-Variablen gespeichert werden kann (vgl. Abschnitt 3.3.5.1 zu denormalisierten Werten bei den binären Gleitkommatypen **float** und **double**).

Kommt es bei der Berechnung eines Ausdrucks *unterwegs* zu einem Unterlauf, ist allerdings ein falsches Endergebnis zu erwarten. Das folgende Programm kommt bei der Rechnung

$$10^{-323} * 10^{308} * 10^{16}$$

dem korrekten Ergebnis 10 recht nahe. Wird aber der Gesamtfaktor Eins (1,0 = 0,1·10,0) unglücklich in den Rechenweg eingebaut, führt ein irreversibler Unterlauf zum falschen Ergebnis Null:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double a = 1e-323;` `  double b = 1e308;` `  double c = 1e16;` `  Console.WriteLine(a * b * c);` `  Console.WriteLine(a * 0.1 * b * 10.0 * c);` `  }` `}`	`9,88131291682493` `0`

3.7 Anweisungen (zur Ablaufsteuerung)

Wir haben uns in Kapitel 3 zunächst mit (lokalen) **Variablen** und elementaren **Datentypen** vertraut gemacht. Dann haben wir gelernt, aus Variablen, Literalen und Methodenaufrufen mit Hilfe von **Operatoren** mehr oder weniger komplexe **Ausdrücke** zu bilden. Diese wurden meist mit der **Console**-Methode **WriteLine()** auf dem Bildschirm ausgegeben oder in Wertzuweisungen verwendet.

In den meisten Beispielprogrammen traten nur wenige Sorten von Anweisungen auf (Variablendeklarationen, Wertzuweisungen und Methodenaufrufe). Nun werden wir uns systematisch mit dem allgemeinen Begriff einer C# - Anweisung befassen und vor allem die wichtigen Anweisungen zur Ablaufsteuerung (Verzweigungen und Schleifen) kennen lernen.

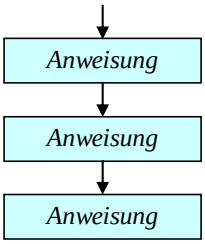
3.7.1 Überblick

Ausführbare Programmteile, die in C# nach unserem bisherigen Kenntnistand als Methoden oder Eigenschaften von Klassen zu realisieren sind, bestehen aus Anweisungen (engl. *statements*).

Am Ende von Abschnitt 3.7 werden Sie die folgenden Sorten von Anweisungen kennen:

- **Deklarationsanweisung für lokale Variablen**
  Die Deklarationsanweisung für lokale Variablen wurde schon in Abschnitt 3.3.6 eingeführt.
  Beispiel: `int i = 1, k;`

- **Ausdrucksanweisungen**
  Folgende Ausdrücke werden zu Anweisungen, sobald man ein Semikolon dahinter setzt:
  - **Wertzuweisung** (vgl. Abschnitte 3.3.6 und 3.5.8)
    Beispiel: `k = i + j;`
  - **Prä- bzw. Postinkrement- oder -dekrementoperation**
    Beispiel: `i++;`
    Im Beispiel ist nur der „Nebeneffekt“ des Ausdrucks `i++` von Bedeutung (vgl. Abschnitt 3.5.1). Sein Wert bleibt ungenutzt.
  - **Methodenaufruf**
    Beispiel: `Console.WriteLine(cond);`
    Besitzt die aufgerufene Methode einen Rückgabewert (siehe unten), wird dieser ignoriert.
- **Leere Anweisung**
  Beispiel: `;`
  Die durch ein einsames (nicht anderweitig eingebundenes) Semikolon ausgedrückte *leere* Anweisung hat keinerlei Effekte und kommt gelegentlich zum Einsatz, wenn die Syntax eine Anweisung verlangt, aber nichts geschehen soll.
- **Blockanweisung**
  Eine Folge von Anweisungen, die durch ein Paar geschweifter Klammern zusammengefasst bzw. abgegrenzt werden, bildet eine **Verbund- bzw. Blockanweisung.** Wir haben uns bereits in Abschnitt 3.3.7 im Zusammenhang mit dem Deklarationsbereich von lokalen Variablen mit Anweisungsblöcken beschäftigt. Wie gleich näher erläutert wird, fasst man z.B. *dann* mehrere Abweisungen zu einem Block zusammen, wenn diese Anweisungen unter einer gemeinsamen Bedingung ausgeführt werden sollen. Es wäre sehr unpraktisch, dieselbe Bedingung für jede betroffene Anweisung wiederholen zu müssen.
- **Anweisungen zur Ablaufsteuerung**
  Die Methoden der bisherigen Beispielprogramme in Kapitel 3 bestanden meist aus einer *Sequenz* von Anweisungen, die bei jedem Programmlauf komplett und linear durchlaufen wurde:

  Oft möchte man jedoch z.B.
  - die Ausführung einer Anweisung (eines Anweisungsblocks) von einer *Bedingung* abhängig machen
  - oder eine Anweisung (einen Anweisungsblock) *wiederholt* ausführen lassen.

  Für solche Zwecke stellt C# etliche Anweisungen zur Ablaufsteuerung zur Verfügung, die bald ausführlich behandelt werden (**bedingte Anweisung**, **Fallunterscheidung**, **Schleifen**).

Blockanweisungen sowie Anweisungen zur Ablaufsteuerung enthalten andere Anweisungen und werden daher auch als **zusammengesetzte Anweisungen** bezeichnet.

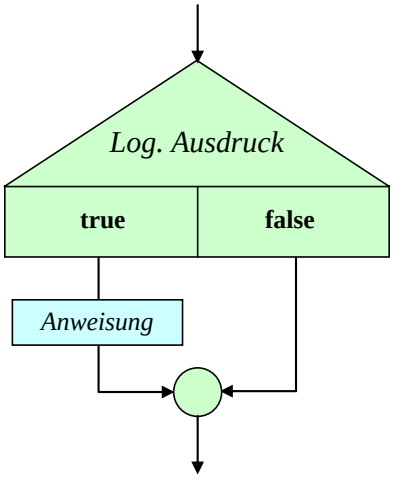
Anweisungen werden durch ein **Semikolon** abgeschlossen, sofern sie nicht mit einer schließenden Blockklammer enden.

3.7.2 Bedingte Anweisung und Fallunterscheidung

Oft ist es erforderlich, dass eine Anweisung nur unter einer bestimmten Bedingung ausgeführt wird. Etwas allgemeiner formuliert geht es darum, dass viele Algorithmen *Fallunterscheidungen* benötigen, also an bestimmten Stellen in Abhängigkeit vom Wert eines steuernden Ausdrucks in unterschiedliche Pfade verzweigen.

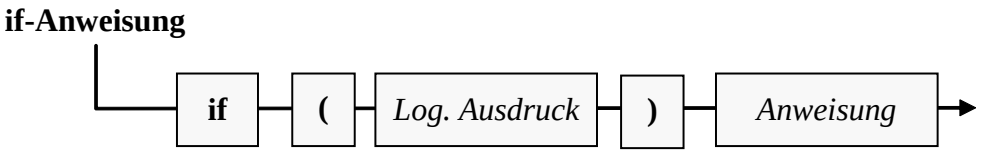
3.7.2.1 if-Anweisung

Nach dem folgenden **Programmablaufplan** (**PAP**) bzw. **Flussdiagramm** soll eine (Block-)Anweisung nur dann ausgeführt werden, wenn ein logischer Ausdruck den Wert **true** besitzt:



Wir werden diese Darstellungstechnik ab jetzt verwenden, um einen Algorithmus oder einen Programmablauf zu beschreiben. Die verwendeten Symbole sind hoffentlich anschaulich, entsprechen aber keiner strengen Normierung.

Während der Programmablaufplan den Zweck (die Semantik) eines Sprachbestandteils erläutert, beschreibt das vertraute Syntaxdiagramm recht präzise, wie zulässige Exemplare des Sprachbestandteils zu bilden sind. Das folgende Syntaxdiagramm beschreibt die zur Realisation einer bedingten Ausführung geeignete **if**-Anweisung:



Um genau zu sein, muss zu diesem Syntaxdiagramm noch angemerkt werden, dass als bedingt auszuführende Anweisung keine Variablendeklaration erlaubt ist. Es ist übrigens nicht vergessen worden, ein Semikolon ans Ende des **if**-Syntaxdiagramms zu setzen. Dort wird eine Anweisung verlangt, wobei konkrete Beispiele oft mit einem Semikolon enden, manchmal aber auch mit einer schließenden geschweiften Klammer.

Im folgenden Beispiel wird eine Meldung ausgegeben, wenn die Variable `anz` einen Wert kleiner oder gleich Null besitzt:

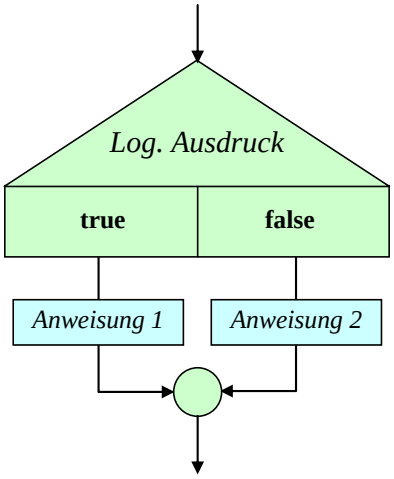
```
if (anz <= 0)
  Console.WriteLine("Die Anzahl muss > 0 sein!");
```

Der Zeilenumbruch zwischen dem logischen Ausdruck und der (Unter-)Anweisung dient nur der Übersichtlichkeit und ist für den Compiler irrelevant.

Selbstverständlich ist als Anweisung auch ein Block erlaubt.

3.7.2.2 if-else - Anweisung

Soll auch etwas passieren, wenn der steuernde logische Ausdruck den Wert **false** besitzt,



erweitert man die **if**-Anweisung um eine **else**-Klausel.

Zur Beschreibung der **if**-**else** - Anweisung wird an Stelle eines Syntaxdiagramms eine alternative Darstellungsform gewählt, die sich am typischen C# - Quellcode-Layout orientiert:

if (*Logischer Ausdruck***)**
    Anweisung 1
else
    Anweisung 2

Wie bei den Syntaxdiagrammen gilt auch für diese Form der Syntaxbeschreibung:

- Für **terminale Sprachbestandteile**, die exakt in der angegebenen Form in konkreten Quellcode zu übernehmen sind, wird **fette** Schrift verwendet.
- *Platzhalter* sind durch *kursive* Schrift gekennzeichnet.

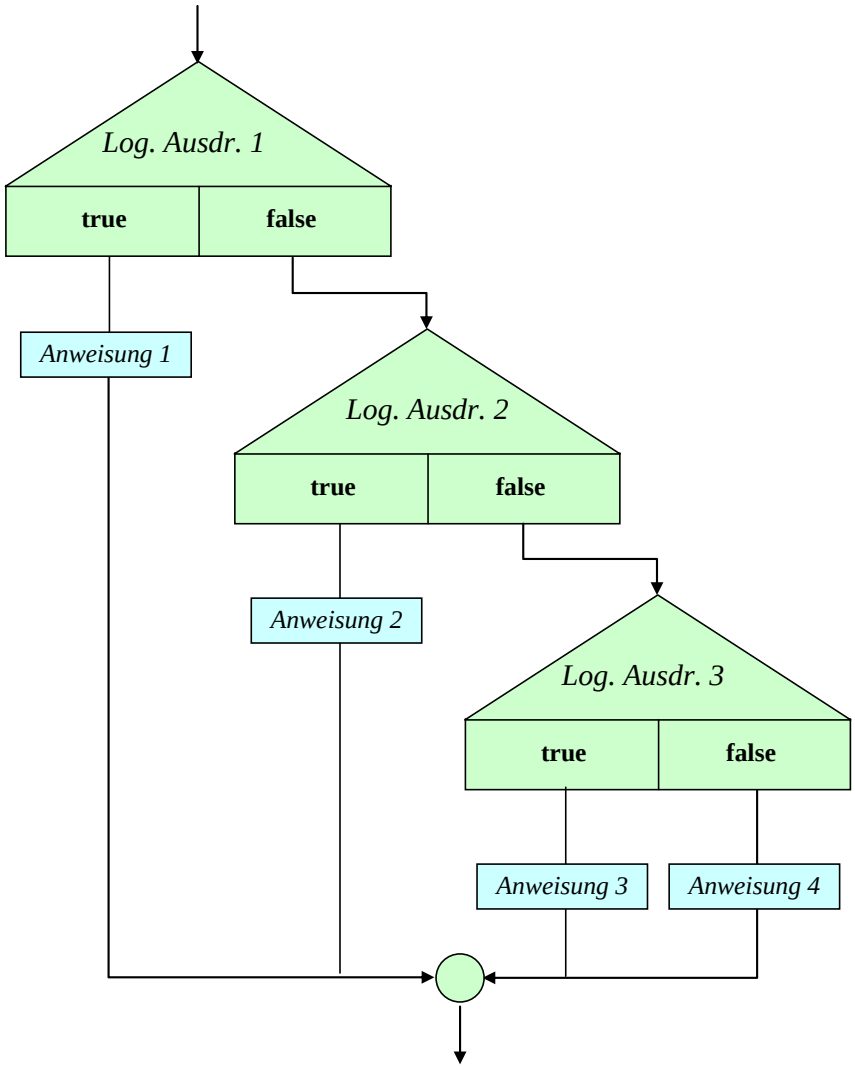
Während die Syntaxbeschreibung im Quellcode-Layout sehr übersichtlich ist, bietet das Syntaxdiagramm den Vorteil, bei komplizierter, variantenreicher Syntax alle zulässigen Formulierungen kompakt und präzise als Pfade durch das Diagramm zu beschreiben.

Wie schon bei der einfachen **if**-Anweisung gilt auch bei der **if**-**else** - Anweisung, dass Variablendeklarationen nicht als eingebettete Anweisungen erlaubt sind.

Im folgenden **if**-**else** - Beispiel wird der natürliche Logarithmus zu einer Zahl geliefert, falls diese positiv ist. Anderenfalls erscheint eine Fehlermeldung mit Signalton (Escape-Sequenz **\a**). Das Argument wird vom Benutzer über eine **ToDouble()** - **ReadLine()** - Konstruktion erfragt (vgl. Abschnitt 3.4.1).

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.Write("Argument: ");` `  double arg = Convert.ToDouble(Console.ReadLine());` `  if (arg > 0)` `  Console.WriteLine("ln(" + arg + ") = "` `  + Math.Log(arg));` `  else` `  Console.WriteLine("\aArgument <= 0!");` `  }` `}`	`Argument: 2` `ln(2) = 0,693147180559945`

Eine bedingt auszuführende Anweisung darf durchaus wiederum vom **if**- bzw. **if-else** - Typ sein, so dass sich mehrere, *hierarchisch geschachtelte* Fälle unterscheiden lassen. Den folgenden Programmablauf mit „sukzessiver Restaufspaltung“



realisiert z.B. eine **if-else** - Konstruktion nach diesem Muster:

if (*Logischer Ausdruck 1***)**
Anweisung 1
else if (*Logischer Ausdruck 2***)**
Anweisung 2
. . .
. . .
else if (*Logischer Ausdruck k***)**
Anweisung k
else
Default-Anweisung

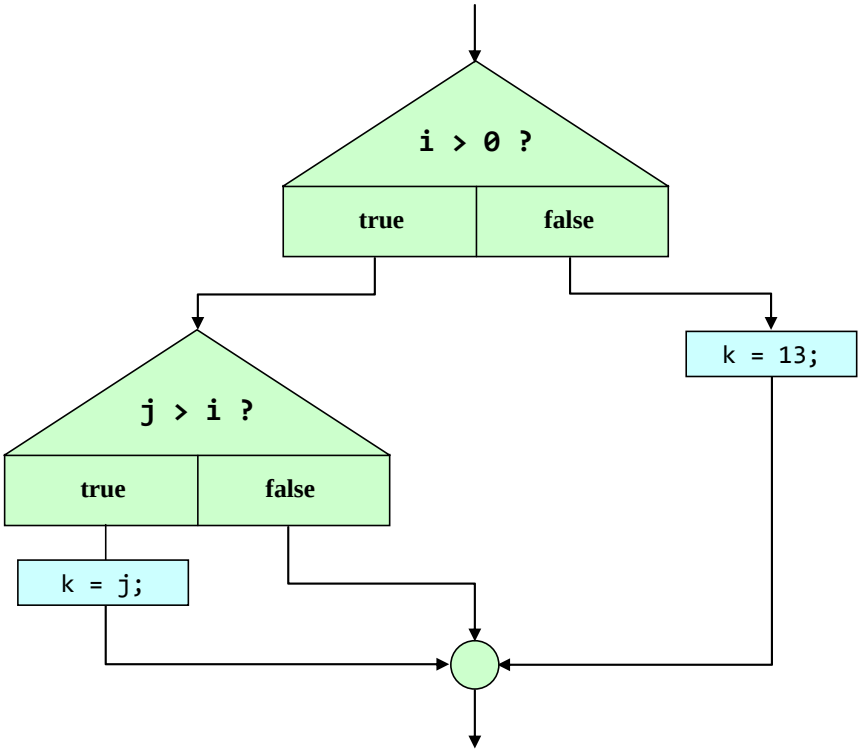
Wenn alle logischen Ausdrücke den Wert **false** annehmen, wird die **else**-Klausel zur letzten **if**-Anweisung ausgeführt.

Bei einer Mehrfallunterscheidung ist die in Abschnitt 3.7.2.3 vorzustellende **switch**-Anweisung gegenüber einer verschachtelten **if-else** - Konstruktion zu bevorzugen, wenn die Fallzuordnung über die verschiedenen Werte *eines* Ausdrucks (z.B. vom Typ **int**) erfolgen kann.

Beim Schachteln von bedingten Anweisungen kann es zum so genannten **dangling-else** - Problem[1] kommen, wobei ein Missverständnis zwischen Compiler und Programmierer hinsichtlich der Zuordnung einer **else**-Klausel besteht. Im folgenden Codefragment

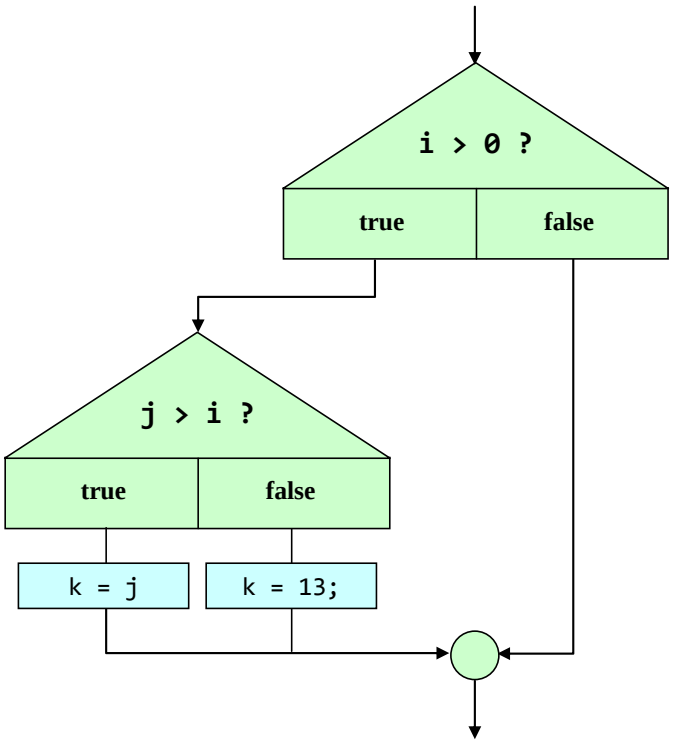
```
if (i > 0)
    if (j > i)
        k = j;
else
    k = 13;
```

lassen die Einrücktiefen vermuten, dass der Programmierer die **else**-Klausel auf die *erste* **if**-Anweisung bezogen zu haben glaubt:



[1] Deutsche Übersetzung von *dangling*: *baumelnd*.

Der Compiler ordnet eine **else**-Klausel jedoch dem in Aufwärtsrichtung nächstgelegenen **if** zu, das nicht durch Blockklammern abgeschottet ist und noch keine **else**-Klausel besitzt. Im Beispiel bezieht er die **else**-Klausel also auf die *zweite* **if**-Anweisung, so dass de facto folgender Programmablauf resultiert:



Bei `i` ≤ 0 geht der Programmierer vom neuen `k`-Wert 13 aus, der beim tatsächliche Programmablauf nicht unbedingt zu erwarten ist.

Mit Hilfe von Blockklammern kann die gewünschte Zuordnung erzwungen werden:

```
if (i > 0)
    {if (j > i)
        k = j;}
else
    k = 13;
```

Alternativ kann man auch dem zweiten **if** eine **else**-Klausel spendieren und dabei eine leere Anweisung verwenden:

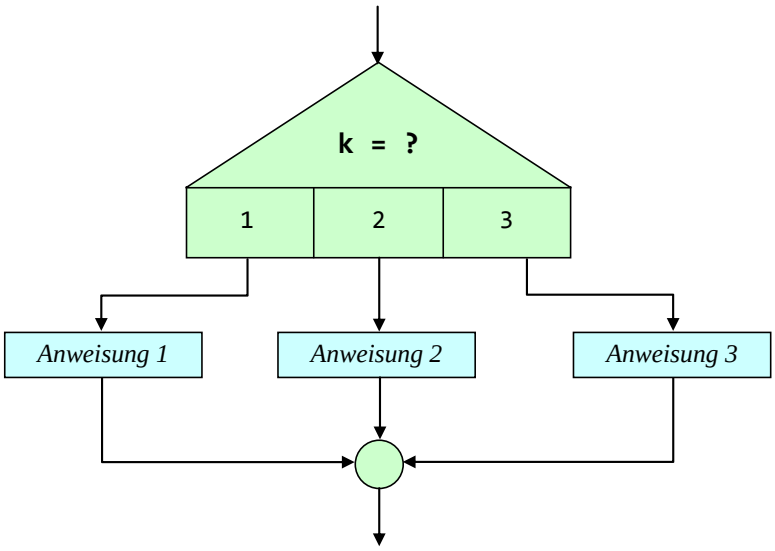
```
if (i > 0)
    if (j > i)
        k = j;
    else
        ;
else
    k = 13;
```

Gelegentlich kommt als Alternative zu einer simplen **if-else** - Anweisung, die zur Berechnung eines Wertes bedingungsabhängig zwei unterschiedliche Ausdrücke benutzt, der Konditionaloperator (vgl. Abschnitt 3.5.9) in Frage, z.B.:

if-else - Anweisung	Konditionaloperator
`double arg = 3.0, d;` `if (arg > 1)` `  d = arg * arg;` `else` `  d = arg;`	`double arg = 3.0, d;` `d = arg > 1 ? arg * arg : arg;`

3.7.2.3 switch-Anweisung

Wenn eine Fallunterscheidung mit mehr als zwei Alternativen in Abhängigkeit vom Wert *eines* Ausdrucks vorgenommen werden soll,

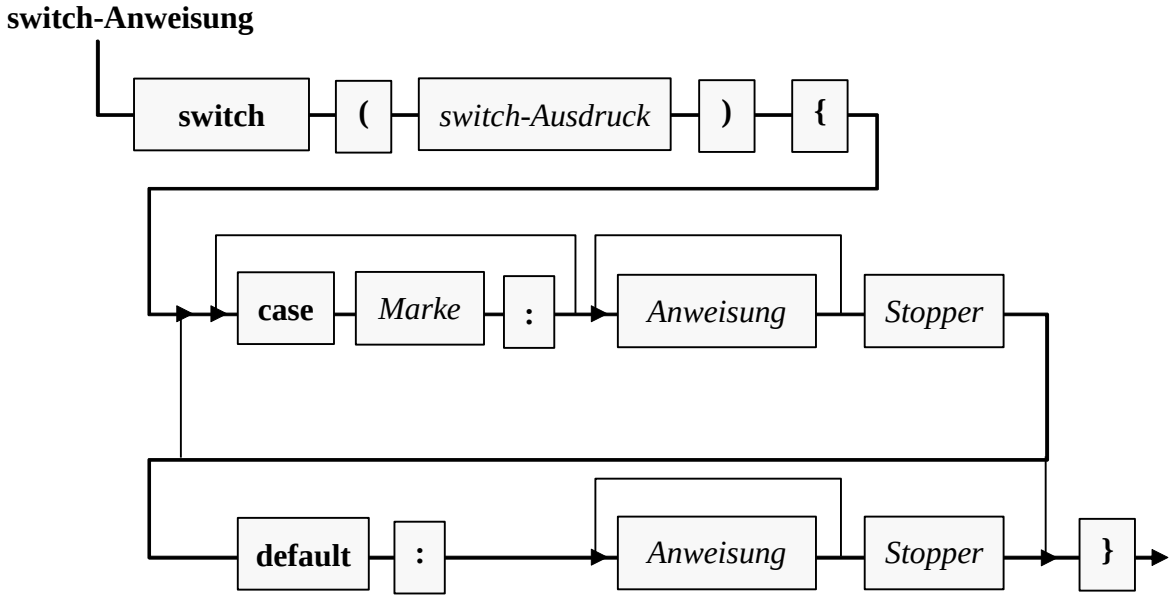


dann ist eine **switch**-Anweisung weitaus handlicher als eine verschachtelte **if-else** - Konstruktion.

In Bezug auf den Datentyp des steuernden Ausdrucks ist C# sehr flexibel und erlaubt:

- Ganzzahlige (integrale) numerische Datentypen (**sbyte**, **short**, **int**, **long**, **byte**, **ushort**, **uint**, **ulong**, **char**). Dazu gehört auch der Datentyp **char** (vgl. Abschnitt 3.3.4).
- **bool**
- Aufzählungstypen (siehe unten)
- Zeichenfolgen (Datentyp **string**)

Der Genauigkeit halber wird die **switch**-Anweisung mit einem Syntaxdiagramm beschrieben. Wer die Syntaxbeschreibung im Quellcode-Layout bevorzugt, kann ersatzweise einen Blick auf die gleich folgenden Beispiele werfen.



Weil später noch ein praxisnahes (und damit auch etwas kompliziertes) Beispiel folgt, ist hier ein ebenso einfaches wie sinnfreies Exemplar zur Erläuterung der Syntax angemessen:

Quellcode	Ausgabe

```
using System;
class Prog {
 static void Main() {
  int zahl = 2;
  switch (zahl) {
   case 1:
    Console.WriteLine("Fall 1 (mit break-Stopper)");
    break;
   case 2:
    Console.WriteLine("Fall 2 (mit Durchfall per goto)");
    goto case 3;
   case 3:
   case 4:
    Console.WriteLine("Fälle 3, 4 (mit break-Stopper)");
    break;
   default:
    Console.WriteLine("Restkategorie");
    break;
  }
 }
}
```

```
Fall 2 (mit Durchfall per goto)
Fälle 3, 4 (mit break-Stopper)
```

Als **case**-Marken sind *konstante* Ausdrücke erlaubt, deren Ergebnis schon der Compiler ermitteln kann (z.B. Literale, Konstanten oder mit konstanten Argumenten gebildete Ausdrücke).

Stimmt bei der Ausführung einer Methode der Wert des **switch**-Ausdrucks mit einer **case**-Marke überein, dann wird die zugehörige Anweisung ausgeführt, ansonsten (falls vorhanden) die **default**-Anweisung.

Soll für mehrere Werte des **switch**-Ausdrucks dieselbe Anweisung ausgeführt werden, setzt man die zugehörigen **case**-Marken hintereinander und lässt die Anweisung auf die letzte Marke folgen. Leider gibt es keine Möglichkeit, eine *Serie* von Fällen durch Angabe der Randwerte (z.B. von *a* bis *z*) festzulegen.

Jeder Fall *muss* mit einem **Stopper** abgeschlossen werden, sogar der terminale **default** - Fall:

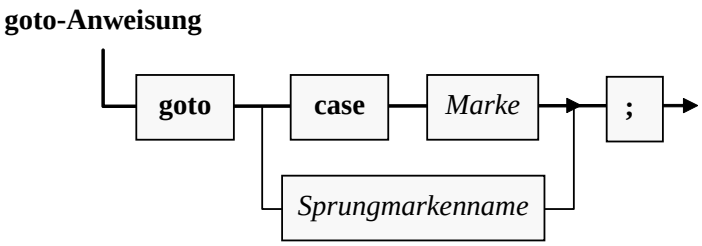
```
    default:
        Console.WriteLine("Restkategorie"); //break;
}
```

Die Steuerung kann nicht von der abschließenden case-Bezeichnung ("default:") aus dem switch-Ausdruck übergeben werden.

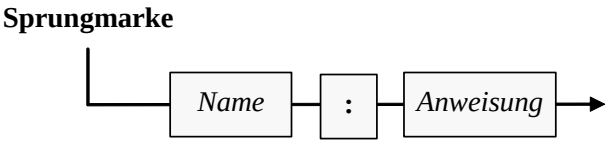
Der aus anderen Programmiersprachen (z.B. C, C++, Java) bekannte automatische „Durchfall“ zu den Anweisungen „tieferer“ Fälle ist verboten, womit viele Programmierfehler vermieden werden.

Meist stoppt man mit der **break**-Anweisung, wobei die Methode hinter der **switch**-Anweisung fortgesetzt wird.

Alternativ kann per **goto**-Anweisung



eine **case**-Marke innerhalb der **switch**-Anweisung oder eine Sprungmarke



an anderer Stelle innerhalb der Methode angesteuert werden. Das gute (böse) alte **goto**, als Inbegriff rückständiger Programmierung („Spaghetti-Code“) aus vielen modernen Programmiersprachen verbannt, ist also in C# erlaubt. Es ist in manchen Situationen durchaus brauchbar, z.B. um den aus anderen Programmiersprachen bekannten **switch**-Durchfall in C# trotz Stopper-Vorschrift zu realisieren.

Im folgenden Beispielprogramm wird die Persönlichkeit des Benutzers mit Hilfe seiner Farb- und Zahlpräferenzen analysiert. Während bei einer Vorliebe für Rot oder Schwarz die Diagnose sofort feststeht, wird bei den restlichen Farben auch die Lieblingszahl berücksichtigt:

```
using System;
class PerST {
   static void Main(String[] args) {
      if (args.Length < 2) {
         Console.WriteLine("Bitte Lieblingsfarbe und -zahl angeben!");
         return;
      }
      String farbe = args[0].ToLower();
      int zahl = Convert.ToInt32(args[1]);
      switch (farbe) {
         case "rot":
            Console.WriteLine("Sie sind durchsetzungsfreudig und impulsiv.");
            break;
         case "schwarz":
            Console.WriteLine("Nehmen Sie nicht Alles so tragisch.");
            break;
         default:
            Console.WriteLine("Ihre Emotionalität ist unauffällig.");
            if (zahl % 2 == 0)
               Console.WriteLine("Sie haben einen geradlinigen Charakter.");
            else
               Console.WriteLine("Sie machen wohl gerne krumme Touren.");
            break;
      }
   }
}
```

Das Programm `PerST` demonstriert nicht nur die **switch**-Anweisung, sondern auch die Verwendung von **Befehlszeilenargumenten**. Benutzer des Programms sollen ihre bevorzugte Farbe sowie ihre Lieblingszahl über Befehlszeilenargumente (Kommandozeilenparameter) angeben. Wer z.B. die Farbe Blau und die Zahl 17 bevorzugt, sollte das Programm (bis auf die beliebige Groß-/Kleinschreibung) folgendermaßen starten:

```
perst Blau 17
```

Im Quellcode wird jeweils nur *eine* Anweisung benötigt, um die (durch Leerzeichen getrennten) Befehlszeilenargumente auszuwerten und das Ergebnis in eine **String**- bzw. **int**-Variable zu befördern. Solche Anweisungen werden Sie mit Leichtigkeit selbst formulieren, sobald Methoden-Parameter sowie Arrays und Zeichenketten behandelt worden sind. An dieser Stelle greifen wir späteren Erläuterungen mal wieder etwas vor (hoffentlich mit motivierendem Effekt):

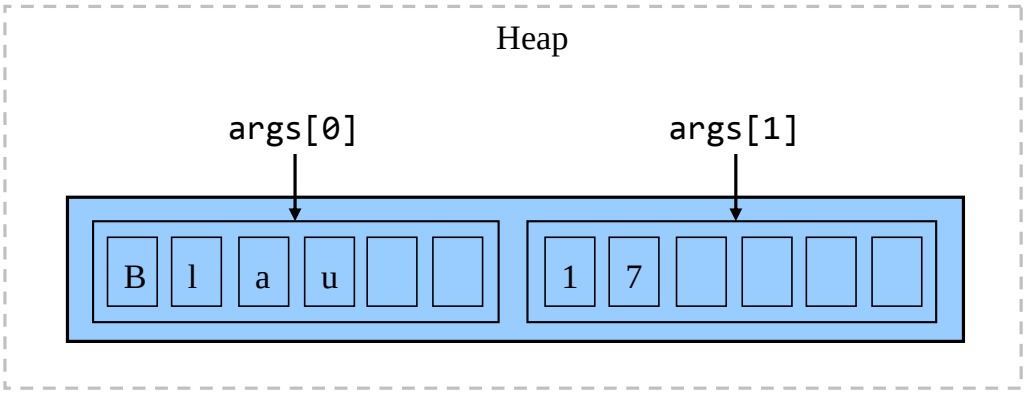
- Bei einem **Array** handelt es sich um ein Objekt, das eine Serie von Elementen desselben Typs aufnimmt, auf die man per Index, d.h. durch die mit eckigen Klammern begrenzte Elementnummer, zugreifen kann.

- In unserem Beispiel kommt ein Array mit Elementen vom Datentyp **String** zum Einsatz. Dies sind Objekte der Klasse **String**, die jeweils eine Zeichenfolge repräsentieren. Literale mit diesem Datentyp sind uns schon öfter begegnet (z.B. `"Hallo"`).
- In der Parameterliste einer Methode kann die gewünschte Arbeitsweise näher spezifiziert werden.
- Besitzt die **Main()** - Methode einer Startklasse einen einzigen Parameter vom Datentyp **String[]** (Array mit **String**-Elementen), dann übergibt das .NET - Laufzeitsystem dieser Methode als **String**-Objekte die Spezifikationen, die der Anwender beim Start hinter den Programmnamen in die Kommandozeile, jeweils durch Leerzeichen getrennt, geschrieben hat. Der Datentyp des Parameters ist fest vorgegeben, sein Name jedoch frei wählbar (im Beispiel: `args`). In der Methode **Main()** kann man auf den Array `args` lesend genauso zugreifen wie auf eine lokale Variable vom selben Typ.
- Das erste Befehlszeilenargument landet im ersten Element des **String**-Arrays `args` und wird mit `args[0]` angesprochen, weil Array-Elemente mit 0 beginnend nummeriert sind. Als Objekt der Klasse **String** wird `args[0]` aufgefordert, die Methode **ToLower()** auszuführen. Diese Methode erstellt ein neues **String**-Objekt, das im Unterschied zum angesprochenen Original auf Kleinschreibung normiert ist, was die spätere Verwendung im Rahmen der **switch**-Anweisung erleichtert. Die Adresse dieses Objekts landet als **ToLower()** - Rückgabewert in der lokalen **String**-Referenzvariablen `farbe`.
- Das zweite Element des Zeichenketten-Arrays `args` (mit der Nummer 1) enthält das zweite Befehlszeilenargument. Zumindest bei kooperativen Benutzern des Beispielprogramms kann man aus dieser Zeichenfolge mit der Klassenmethode **ToInt32()** der Klasse **Convert** eine Zahl vom Datentyp **int** gewinnen und anschließend der Variablen `zahl` zuweisen.

Nach einem Programmstart mit dem Aufruf

```
perst Blau 17
```

kann man sich den **String**-Array `args`, der als Objekt im Heap-Bereich des programmeigenen Speichers abgelegt wird, ungefähr so vorstellen:



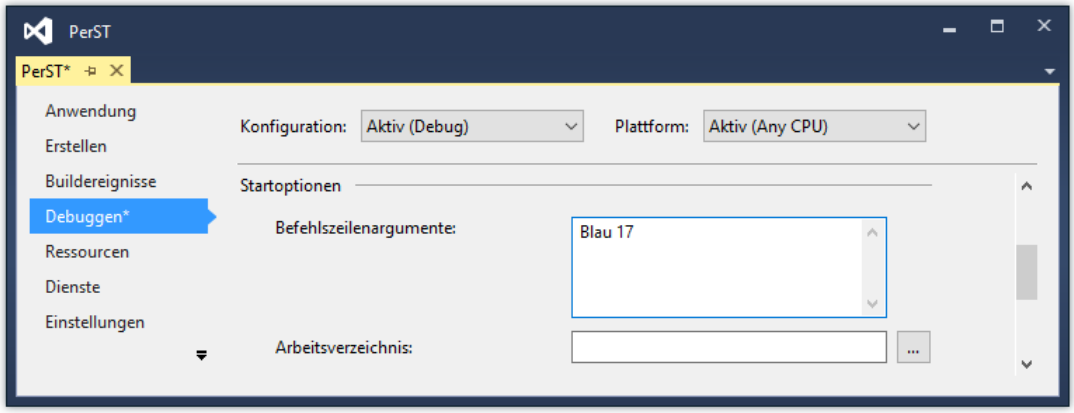
Ansonsten ist im Beispielprogramm noch die **return**-Anweisung von Interesse, welche die **Main()** - Methode und damit das Programm in Abhängigkeit von einer Bedingung sofort beendet:

```
return;
```

Die „offizielle“ Behandlung der **return**-Anweisung erfolgt später im Zusammenhang mit den Methoden.

Für den Programmstart aus dem Visual Studio kann man die Befehlszeilenargumente folgendermaßen vereinbaren:

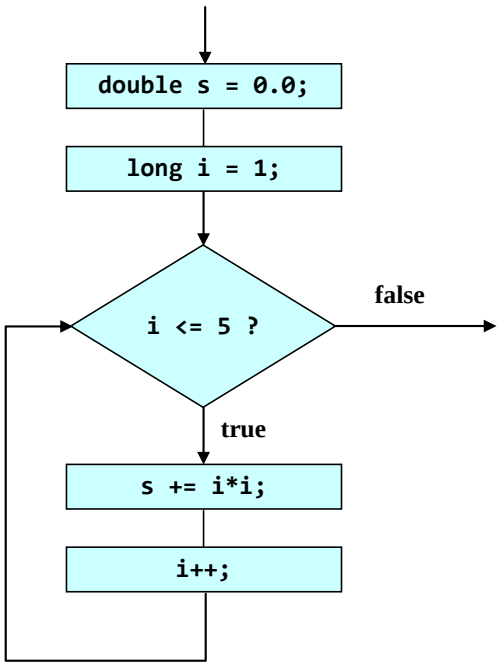
- Menübefehl **Projekt > Eigenschaften**
- Registerkarte **Debuggen**
- **Befehlszeilenargumente** eintragen, z.B.:



3.7.3 Wiederholungsanweisungen

Eine Wiederholungsanweisung (oder schlicht: *Schleife*) kommt zum Einsatz, wenn eine (Verbund-)-Anweisung *mehrfach* ausgeführt werden soll, wobei sich in der Regel schon der Gedanke daran verbietet, die Anweisung entsprechend oft in den Quelltext zu schreiben.

Im folgenden Flussdiagramm ist ein iterativer Algorithmus zu sehen, der die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet: [1]



C# bietet verschiedene Wiederholungsanweisungen, die sich bei der Ablaufsteuerung unterscheiden. Wir werden sie gleich im Detail betrachten und als Beispiel jeweils den Algorithmus aus dem

[1] Das Verzweigungssymbol sieht aus darstellungstechnischen Gründen etwas anders aus als in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**, was aber keine Verwirrung stiften sollte. Obwohl im Beispiel eine Steigerung der Laufgrenze für die Variable `i` kaum in Frage kommt, soll an dieser Stelle das Thema *Ganzzahlüberlauf* (vgl. Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**) kurz in Erinnerung gerufen werden. Weil die Variable `i` vom Typ **long** ist, kann der Algorithmus bis zur Laufgrenze 3037000499 verwendet werden. Für größere `i`-Werte tritt beim Ausdruck `i*i` ein Überlauf auf, und das Ergebnis ist unbrauchbar.

obigen Flussdiagramm implementieren. Zunächst sollen die Optionen zur Schleifensteuerung bei leicht vereinfachender Beschreibung im Überblick präsentiert werden:

- **Zählergesteuerte Schleife** (**for**)
  Die Anzahl der Wiederholungen steht typischerweise schon vor Schleifenbeginn fest. Bei der Ablaufsteuerung kommt eine Zählvariable zum Einsatz, die *vor dem ersten* Schleifendurchgang initialisiert und *am Ende jedes* Durchlaufs aktualisiert (z.B. inkrementiert) wird. Die zur Schleife gehörige (Verbund-)Anweisung wird ausgeführt, solange die Zählvariable einen festgelegten Grenzwert nicht überschritten hat.
- **Iterieren über die Elemente einer Kollektion** (**foreach**)
  Mit der von Visual Basic übernommenen **foreach**-Schleife bietet C# die Möglichkeit, eine Anweisung für jedes Element eines Arrays, einer Zeichenfolge oder einer anderen Kollektion (siehe unten) auszuführen.
- **Bedingungsabhängige Schleife** (**while**, **do**)
  Bei jedem Schleifendurchgang wird eine Bedingung überprüft, und das Ergebnis entscheidet über das weitere Vorgehen:
  - **true**: Die zur Schleife gehörige Anweisung wird ein weiteres Mal ausgeführt.
  - **false**: Die Schleife wird beendet.

  Bei der *kopf*gesteuerten **while**-Schleife wird die Bedingung *vor Beginn* eines Durchgangs geprüft, bei der *fuß*gesteuerten **do**-Schleife hingegen *am Ende*. Weil man z.B. *nach* dem 3. Schleifendurchgang in keiner anderen Lage ist wie *vor* dem 4. Schleifendurchgang, geht es bei der Entscheidung zwischen Kopf- und Fußsteuerung lediglich darum, ob auf jeden Fall ein *erster* Schleifendurchgang stattfinden soll oder nicht.

Die gesamte Konstruktion aus Schleifensteuerung und (Verbund-)anweisung stellt in syntaktischer Hinsicht *eine* zusammengesetzte Anweisung dar.

3.7.3.1 Zählergesteuerte Schleife (for)

Die Anweisung einer **for**-Schleife wird ausgeführt, solange eine Bedingung erfüllt ist, die normalerweise auf eine ganzzahlige Laufvariable Bezug nimmt.

Auf das Schlüsselwort **for** folgt die von runden Klammern umgebene Schleifensteuerung, wo die Vorbereitung der Laufvariablen (nötigenfalls samt Deklaration), die Fortsetzungsbedingung und die Aktualisierungsvorschrift untergebracht werden. Danach folgt die wiederholt auszuführende (Block-)Anweisung:

for (*Vorbereitung***;** *Bedingung***;** *Aktualisierung***)**
    Anweisung

Zu den drei Bestandteilen der Schleifensteuerung sind einige Erläuterungen erforderlich, wobei anschließend etliche weniger typische bzw. sinnvolle Möglichkeiten weggelassen werden:

- ***Vorbereitung***
  In der Regel wird man sich auf *eine* Laufvariable beschränken und dabei einen Ganzzahl-Typ wählen. Somit kommen im Vorbereitungsteil der **for**-Schleifensteuerung in Frage:
  - eine Wertzuweisung, z.B.:
    ```
    i = 1
    ```
  - eine Variablendeklaration mit Initialisierung, z.B.
    ```
    long i = 1
    ```

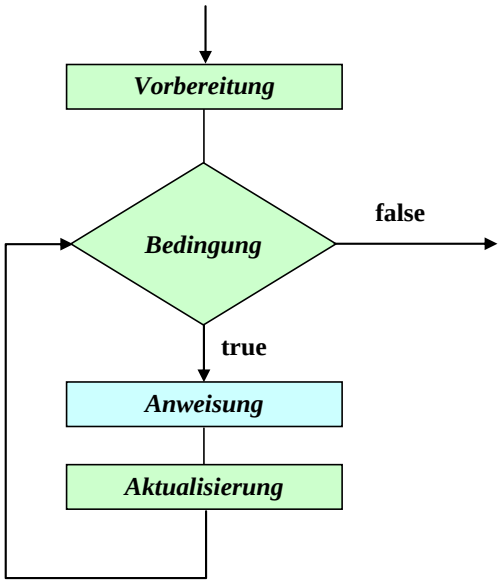
  Im folgenden Programm, das die Summe der quadrierten natürlichen Zahlen von 1 bis 5 berechnet, findet sich die zweite Variante:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double s = 0.0;` `  for (long i = 1; i <= 5; i++)` `  s += i*i;` `  Console.WriteLine("Quadratsumme = " + s);` `  }` `}`	`Quadratsumme = 55`

Der Vorbereitungsteil wird *vor dem ersten Durchlauf* ausgeführt. Eine hier deklarierte Variable ist *lokal* bzgl. der **for**-Schleife, steht also nur in deren Anweisung(sblock) zur Verfügung.

- ***Bedingung***
  Üblicherweise wird eine Ober- oder Untergrenze für die Laufvariable gesetzt, doch erlaubt C# beliebige logische Ausdrücke. Die Bedingung wird *vor jedem Schleifendurchgang* geprüft. Resultiert der Wert **true**, wird der Anweisungsteil ausgeführt, anderenfalls wird die **for**-Schleife verlassen. Folglich kann es auch passieren, dass überhaupt kein Schleifendurchgang zustande kommt.
- ***Aktualisierung***
  Am Ende jedes Schleifendurchgangs (nach Ausführung der Anweisung) wird der Aktualisierungsteil ausgeführt. Hier wird meist die Laufvariable in- oder dekrementiert.

Im folgenden Flussdiagramm sind die eben geschriebenen semantischen Regeln zur **for**-Schleife dargestellt, wobei die Bestandteile der Schleifensteuerung an der grünen Farbe zu erkennen sind:



Zu den (zumindest stilistisch) bedenklichen Konstruktionen, die der Compiler klaglos umsetzt, gehören **for**-Schleifenköpfe ohne Vorbereitung oder ohne Aktualisierung, wobei die trennenden Strichpunkte trotzdem zu setzen sind. In solchen Fällen ist die Umlaufzahl einer **for**-Schleife natürlich nicht mehr aus dem Schleifenkopf abzulesen. Dies gelingt auch dann nicht, wenn eine Indexvariable in der Schleifenanweisung modifiziert wird.

3.7.3.2 Iterieren über die Elemente einer Kollektion (foreach)

Obwohl wir uns bisher nur anhand von Beispielen mit *Kollektionen* wie Arrays oder Zeichenfolgen beschäftigt haben, soll die einfach aufgebaute **foreach**-Schleife doch hier im Kontext mit den übrigen Schleifen behandelt werden. Konzentrieren Sie sich also auf das gleich präsentierte, leicht

nachvollziehbare Beispiel, und lassen Sie sich durch die Begriffe *Array*, *Kollektion* und *Interface*, die zu später behandelten Themen gehören, nicht beunruhigen.

Die Steuerungslogik der **foreach**-Schleife:

- Es ist eine Kollektion mit einer festen Anzahl von gleichartigen Elementen vorhanden, z.B. eine Zeichenfolge mit acht Zeichen.
- Die Anweisung der **foreach**-Schleife wird nacheinander für jedes Element der Kollektion ausgeführt.
- Im Schleifenkopf wird eine Iterationsvariable vom Datentyp der Kollektionselemente deklariert, über die in der Schleifenanweisung das aktuelle Element angesprochen werden kann.

Die Syntax der **foreach**-Schleife:

foreach (*Elementtyp Iterationsvariable* **in** *Kollektion***)**
    Anweisung

Als Kollektion erlaubt der Compiler:[1]

- einen Array
- eine Instanz eines Typs, der das Interface **IEnumerable** im Namensraum **System.Collections** implementiert

Das Beispielprogramm `PerST` in Abschnitt 3.7.2.3 hat demonstriert, wie man über einen Parameter der Methode **Main()** auf die Zeichenfolgen zugreifen kann, welche der Benutzer beim Start eines Konsolenprogramms hinter den Assembly-Namen geschrieben hat. Im folgenden Programm wird durch zwei geschachtelte **foreach**-Schleifen für jedes Element im **string**-Array `args` mit den Befehlszeilenargumenten folgendes getan:

- In der äußeren **foreach**-Schleife wird die aktuelle Zeichenfolge komplett ausgegeben.
- In der inneren **foreach**-Schleife wird jedes Zeichen der aktuellen Zeichenfolge separat ausgegeben.

---

[1] Genaugenommen ...

- muss das Interface **IEnumerable** nicht *explizit* implementiert werden (siehe Kapitel 8). Es genügt, die Methode **GetEnumerator()** mit einer Rückgabe vom Typ **IEnumerator** oder **IEnumerator<T>** zu implementieren (siehe Abschnitt 8.6), z.B.:

```
class FET {
  public System.Collections.Generic.IEnumerator<int> GetEnumerator() {
    for (int i = 0; i < 5; i++) yield return i;
  }
}
class Prog {
  public static void Main() {
    var fet = new FET();
    foreach (int i in fet) System.Console.WriteLine(i);
  }
}
```

- hätte die **foreach**-Tauglichkeit von Arrays nicht explizit genannt werden müssen, weil jeder Array ein Objekt aus einer Klasse ist, welche die Schnittstelle **IEnumerable** implementiert.

Auch ein Typ, welche das generische Interface **IEnumerable<T>** im Namensraum **System.Collections.Generic** implementiert, ist **foreach**-tauglich. Allerdings stammt dieses Interface vom älteren, nichtgenerischen Interface **IEnumerable** ab, so dass auch die **GetEnumerator()** - Überladung der nichtgenerischen Schnittstelle (mit einer Rückgabe vom Typ **IEnumerator**) implementiert werden muss.

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main(string[] args) {` `  foreach (string s in args) {` `  Console.WriteLine(s);` `  foreach (char c in s)` `  Console.WriteLine(" " + c);` `  Console.WriteLine();` `  }` `  }` `}`	`eins` ` e` ` i` ` n` ` s` `zwei` ` z` ` w` ` e` ` i`

Beim Array mit den Befehlszeilenargumenten haben wir es mit einer Kollektion zu tun, die als Elemente wiederum Kollektionen enthält (nämlich **String**-Objekte).

Eine wichtige Einschränkung der **foreach**-Schleife besteht darin, dass man in ihrer Anweisung über die Iterationsvariable nur *lesend* auf die Kollektionselemente zugreifen kann, so dass z.B. der folgende Versuch zum „Löschen“ der Elemente im **string**-Array `args` misslingt:

```
foreach (string s in args)
    s = "erased";
```

Der Grund für dieses Verhalten besteht darin, dass die Iterationsvariable eine *Kopie* des aktuellen Kollektionselements enthält, sodass eine Änderung sinnlos wäre.[1]

Über eine **for**-Schleife ist der Plan aus dem letzten Beispiel durchaus zu realisieren, z.B.:

```
for (int i = 0; i < args.Length; i++)
    args[i] = "erased";
```

3.7.3.3 Bedingungsabhängige Schleifen

Wie die Erläuterungen zur **for**-Schleife gezeigt haben, ist die Überschrift zu diesem Abschnitt nicht sehr trennscharf, weil bei der **for**-Schleife ebenfalls eine beliebige Terminierungsbedingung angegeben werden darf. In vielen Fällen ist es eine Frage des persönlichen Geschmacks, welche Wiederholungsanweisung man zur Lösung eines konkreten Iterationsproblems benutzt. Unter der aktuellen Abschnittsüberschrift diskutiert man traditionsgemäß die **while**- und die **do**-Schleife.

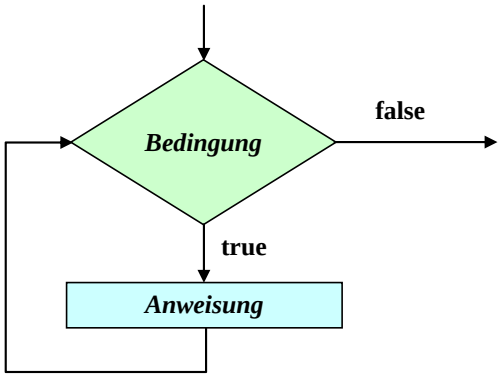
3.7.3.3.1 while-Schleife

Die **while**-Anweisung kann als vereinfachte **for**-Anweisung beschreiben kann: Wer im Kopf einer **for**-Schleife auf Vorbereitung und Aktualisierung verzichten möchte, ersetzt besser das Schlüsselwort **for** durch **while** und erhält dann folgende Syntax:

while (*Bedingung***)**
    Anweisung

[1] Weitere Details sind hier zu finden:
http://stackoverflow.com/questions/4004755/why-is-foreach-loop-read-only-in-c-sharp

Wie bei der **for**-Anweisung wird die Bedingung *vor Beginn* eines Schleifendurchgangs geprüft. Resultiert der Wert **true**, so wird die Anweisung ausgeführt, anderenfalls wird die **while**-Schleife verlassen, eventuell noch vor dem ersten Durchgang:

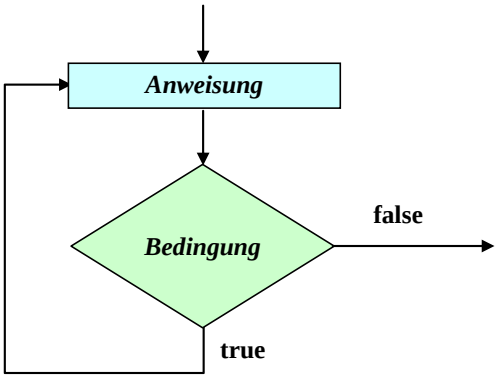


Das in Abschnitt 3.7.3.1 vorgestellte Beispielprogramm zur Quadratsummenberechnung mit Hilfe einer **for**-Schleife kann leicht auf die Verwendung einer **while**-Schleife umgestellt werden:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  double s = 0.0;` `  long i = 1;` `  while (i <= 5) {` `  s += i * i;` `  i++;` `  }` `  Console.WriteLine("Quadratsumme = " + s);` `  }` `}`	`Quadratsumme = 55`

3.7.3.3.2 do-Schleife

Bei der **do**-Schleife wird die Fortsetzungsbedingung *am Ende* der Schleifendurchläufe geprüft, so dass wenigstens *ein* Durchlauf stattfindet:



Das Schlüsselwort **while** tritt auch in der Syntax zur **do**-Schleife auf:

do
  Anweisung
while (*Bedingung***);**

do-Schleifen werden seltener benötigt als **while**-Schleifen, sind aber z.B. dann von Vorteil, wenn man vom Benutzer eine Eingabe mit bestimmten Eigenschaften einfordern möchte. In folgendem Programm wird wie gewohnt mit der statischen Methode **Console.ReadLine()** eine per **Enter** -

Taste quittierte Zeile von der Konsole gelesen. Weil dieser Methodenaufruf ein Ausdruck vom Typ **String** ist, kann das erste Zeichen (mit der Nummer 0) per Indexzugriff (mit Hilfe der eckigen Klammern) angesprochen werden:

Quellcode	Ein-/Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  char antwort;` `  do {` `  Console.Write("Einverstanden? (j/n)? ");` `  antwort = Console.ReadLine()[0];` `  } while (antwort != 'j' && antwort != 'n');` `  }` `}`	`Einverstanden? (j/n)? r` `Einverstanden? (j/n)? 4` `Einverstanden? (j/n)? j`

Bei einer **do**-Schleife mit Anweisungs*block* sollte man die **while**-Klausel unmittelbar hinter die schließende Blockklammer setzen (in dieselbe Zeile), um sie optisch von einer selbständigen **while**-Anweisung abzuheben (siehe Beispiel).

3.7.3.4 Endlosschleifen

Bei einer **for**-, **while**- oder **do**-Schleife kann es in Abhängigkeit von der Fortsetzungsbedingung passieren, dass der Anweisungsteil so lange wiederholt wird, bis das Programm von außen abgebrochen wird. Solche „Endlosschleifen“ sind als gravierende Programmierfehler unbedingt zu vermeiden. Befindet sich ein Programm in diesem Zustand muss es mit Hilfe des Betriebssystems abgebrochen werden, bei unseren Konsolenanwendungen z.B. über die Tastenkombination **Strg+C**.

In folgendem Beispiel resultiert eine Endlosschleife aus einer ungeschickten Identitätsprüfung bei **double**-Werten (vgl. Abschnitt 3.5.3):

```
using System;
class Prog {
  static void Main() {
    long i = 0;
    double d = 1.0;
    // besser: while (d > 1.0e-14) {
    while (d != 0.0) {
      i++;
      d = d - 0.1;
      Console.WriteLine("i = {0}, d = {1}", i, d);
    }
    Console.WriteLine("Fertig!");
  }
}
```

3.7.3.5 Schleifen(durchgänge) vorzeitig beenden

Mit der **break**-Anweisung, die uns schon in Abschnitt 3.7.2.3 als Bestandteil der **switch**-Anweisung begegnet ist, kann eine Schleife vorzeitig verlassen werden, wobei die Methode hinter der Schleife fortgesetzt wird.[1]

Mit der **continue**-Anweisung veranlasst man C#, den aktuellen Schleifendurchgang zu beenden und sofort mit dem nächsten zu beginnen (bei **for** und **while** nach Prüfung der Fortsetzungsbedingung).

---

[1] Eine eher selten genutzte Alternative zur **break**-Anweisung ist die ebenfalls in Abschnitt 3.7.2.3 vorgestellte **goto**-Anweisung. Damit lässt sich die Methode bei der angegebenen Sprungmarke fortsetzen.

In folgendem Beispielprogramm zur (relativ primitiven) Primzahlendiagnose wird die schon in Abschnitt 3.4.1 erwähnte Ausnahmebehandlung per **try-catch** - Anweisung eingesetzt, die später in einem eigenen Kapitel ausführlich behandelt wird. Während wir in der Regel der Einfachheit halber auf eine Prüfung der Eingabedaten verzichten, findet im aktuellen Programm vor allem deshalb eine praxisnahe Validierung statt, weil sich dabei ein sinnvoller **continue**-Einsatz ergibt:

```
using System;
class Primitiv {
  static void Main() {
    bool tg;
    ulong i, mtk, zahl;
    Console.WriteLine("Einfacher Primzahlendetektor\n");
    while (true) {
      Console.Write("Zu untersuchende positive Ganzzahl oder 0 zum Beenden: ");
        try {
          zahl = Convert.ToUInt64(Console.ReadLine());
        } catch {
          Console.WriteLine("Keine Zahl (im zulässigen Bereich)!\n");
          continue;
        }
      if (zahl == 0)
        break;
      tg = false;
      mtk = (ulong) Math.Sqrt(zahl);  //Maximaler Teiler-Kandidat
      for (i = 2; i <= mtk; i++)
        if (zahl % i == 0) {
          tg = true;
          break;
        }
      if (tg)
        Console.WriteLine(zahl+" ist keine Primzahl (Teiler: "+i+").\n");
      else
         Console.WriteLine(zahl+" ist eine Primzahl.\n");
    }
    Console.WriteLine("\nVielen Dank für den Einsatz dieser Software!");
  }
}
```

Bei einer irregulären, nicht als ganze Zahl im **ulong**-Wertebereich interpretierbaren Eingabe „wirft“ die **Convert**-Methode **ToUInt64()** eine Ausnahme. Weil sich der Methodenaufruf in einem **try**-Block befindet, wird im Ausnahmefall der zugehörige **catch**-Block ausgeführt. Im Beispielprogramm erscheint dann eine Fehlermeldung auf dem Bildschirm, und der aktuelle Durchgang der **while**-Schleife wird per **continue** verlassen.

Durch Eingabe der Zahl 0 kann das Beispielprogramm beendet werden, wobei die absichtlich konstruierte **while** - „Endlosschleife“ per **break** verlassen wird.

Man hätte die **continue**- und die **break**-Anweisung zwar vermeiden können (siehe Übungsaufgabe in Abschnitt 3.7.4), doch werden bei dem vorgeschlagenen Verfahren lästige Sonderfälle (unzulässige Werte, 0 als Terminierungssignal) auf besonders übersichtliche Weise abgehakt, bevor der Kernalgorithmus startet.

Zum Kernalgorithmus der Primzahlendiagnose sollte vielleicht noch erläutert werden, warum die Suche nach einem Teiler des Primzahlkandidaten bei seiner Wurzel enden kann (genauer: bei der größten ganzen Zahl ≤ Wurzel):

Sei d (≥ 2) ein echter Teiler der positiven, ganzen Zahl z, d.h. es gebe eine Zahl k (≥ 2) mit

$$z = k \cdot d$$

Dann ist auch k ein echter Teiler von z, und es gilt:

$$d \leq \sqrt{z} \quad \text{oder} \quad k \leq \sqrt{z}$$

Anderenfalls wäre das Produkt $k \cdot d$ größer als z. Wir haben also folgendes Ergebnis: Wenn eine Zahl z keinen echten Teiler kleiner oder gleich $\sqrt{z}$ hat, kann man auch jenseits dieser Grenze keinen finden, und z ist eine Primzahl.

Zur Berechnung der Quadratwurzel verwendet das Beispielprogramm die Methode **Sqrt()** aus der Klasse **Math**, über die man sich bei Bedarf in der FCL-Dokumentation informieren kann.

3.7.4 Übungsaufgaben zu Abschnitt 3.7

1) Bei einer Lotterie soll der folgende Gewinnplan gelten:

- Durch 13 teilbare Losnummern gewinnen 100 Euro.
- Losnummern, die nicht durch 13 teilbar sind, gewinnen immerhin noch einen Euro, wenn sie durch 7 teilbar sind.

Wird in folgendem Codesegment für Losnummern in der **int**-Variablen `losNr` der richtige Gewinn ermittelt?

```
if (losNr % 13 != 0)
   if (losNr % 7 == 0)
      Console.WriteLine("Das Los gewinnt einen Euro!");
else
   Console.WriteLine("Das Los gewinnt 100 Euro!");
```

2) Warum liefert dieses Programm widersprüchliche Auskünfte über die boolesche Variable `b`?

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  bool b = false;` `  if (b = false)` `  Console.WriteLine("b ist False");` `  else` `  Console.WriteLine("b ist True");` `  Console.WriteLine("Kontr.ausg. von b: " + b);` `  }` `}`	`b ist True` `Kontr.ausg. von b: False`

3) Erstellen Sie eine Variante des Primzahlen-Diagnoseprogramms aus Abschnitt 3.7.3.5, die ohne **break** und **continue** auskommt.

4) Wie oft wird die folgende **while**-Schleife ausgeführt?

```
using System;
class Prog {
    static void Main() {
     int i = 0;
     while (i < 100);
     {
        i++;
        Console.WriteLine(i);
     }
  }
}
```

5) Verbessern Sie das im Übungsaufgaben-Abschnitt 3.5.11 in Auftrag gegebene Programm zur DM-Euro - Konvertierung so, dass es nicht für jeden Betrag neu gestartet werden muss. Vereinba-

ren Sie mit dem Benutzer ein geeignetes Verfahren für den Fall, dass er das Programm doch irgendwann einmal beenden möchte.

6) Bei einem **double**-Wert sind maximal 16 signifikante Dezimalstellen garantiert (siehe Abschnitt 3.3.4). Folglich kann ein Rechner die **double**-Werte 1,0 und $1{,}0 + 2^{-i}$ ab einem bestimmten Exponenten *i* nicht mehr voneinander unterscheiden. Bestimmen Sie mit einem Testprogramm den größten ganzzahligen Exponenten *i*, für den man noch erhält:

$$1{,}0 + 2^{-i} > 1{,}0$$

In dem (zur freiwilligen Lektüre empfohlenen) Vertiefungsabschnitt 3.3.5.1 findet sich eine Erklärung für das Ergebnis.

7) In dieser Aufgabe sollen Sie verschiedene Varianten von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers (GGT) zu zwei natürlichen Zahlen *u* und *v* implementieren und die Laufzeitunterschiede messen. Verwenden Sie als ersten Kandidaten den im Einführungsbeispiel zum Kürzen von Brüchen (Methode `Kuerze()` der Klasse `Burch`) benutzten Algorithmus (siehe Abschnitt 1.1.2). Sein Problem besteht darin, dass bei stark unterschiedlichen Zahlen *u* und *v* sehr viele Subtraktions-Operationen erforderlich sind. In der meist benutzten Variante des Euklidischen Verfahrens wird dieses Problem vermieden, indem an Stelle der Subtraktion die Modulo-Operation zum Einsatz kommt, basierend auf dem folgendem Satz der mathematischen Zahlentheorie:

> Für zwei natürliche Zahlen *u* und *v* (mit *u* > *v*) ist der GGT gleich dem GGT von *u* und *u* % *v* (*u* modulo *v*).

Begründung (analog zu Abschnitt 1.1.3): Für natürliche Zahlen *u* und *v* mit *u* > *v* gilt:

x ist gemeinsamer Teiler von *u* und *v*

⇔

x ist gemeinsamer Teiler von *v* und *u* % *v*

Der GGT-Algorithmus per Modulo-Operation läuft für zwei natürliche Zahlen *u* und *v* ($u \geq v > 0$) folgendermaßen ab:

> Es wird geprüft, ob *u* durch *v* teilbar ist.
> Trifft dies zu, ist *v* der GGT.
> Anderenfalls ersetzt man:
>
> > *u* durch *v*
> > *v* durch *u* % *v*
>
> Das Verfahren startet neu mit den kleineren Zahlen.

Die Voraussetzung $u \geq v$ ist nicht wesentlich, weil beim Start mit $u < v$ der erste Algorithmusschritt die beiden Zahlen vertauscht.

Um den Zeitaufwand für beide Varianten zu messen, kann man z.B. mit der Struktur **DateTime** aus dem Namensraum **System** arbeiten. Mit ihrer statischen **Now**-Eigenschaft ermittelt man den aktuellen Zeitpunkt und fragt dann die **Ticks**-Eigenschaft dieser **DateTime**-Instanz ab. Der letzte Satz ist zugegebenermaßen recht kompliziert geraten und enthält etliche noch unzureichend erklärte Details. Man erhält jedenfalls mit einer recht einfachen Syntax die Anzahl der 100-Nanosekunden - Intervalle, die seit dem 1. Januar 0001, 00:00:00 vergangen sind, z.B.:

```
long zeit = DateTime.Now.Ticks;
```

Für die Beispielwerte *u* = 999000999 und *v* = 36 liefern beide Euklid-Varianten sehr verschiedene Laufzeiten (CPU: Intel Core i3 550, Betriebssystem: Windows 7-64):

```
GGT-Bestimmung mit Euklid (Differenz)

Erste Zahl:     999000999

Zweite Zahl:    36

GGT:            9

Benöt. Zeit:    120,1318 Millisek.
```

```
GGT-Bestimmung mit Euklid (Modulo)

Erste Zahl:   999000999

Zweite Zahl:  36

GGT:          9

Benöt. Zeit:  4,0142 Millisek.
```

4 Klassen und Objekte

Softwareentwicklung mit C# besteht im Wesentlichen aus der Definition von **Klassen** (und sonstigen, später noch zu behandelnden Typen), die aufgrund der vorangegangenen objektorientierten Analyse und Modellierung …

- als Baupläne für Objekte
- und/oder als Akteure

konzipiert werden. Wenn ein spezieller Akteur im Programm nur *einfach* benötigt wird, kann eine handelnde Klasse diese Rolle übernehmen. Sind hingegen mehrere Individuen einer Gattung erforderlich (z.B. mehrere Brüche in einem Bruchrechnungsprogramm oder mehrere Fahrzeuge in der Speditionsverwaltung), dann ist eine Klasse mit Bauplancharakter gefragt.

Für eine Klasse und/oder ihre Objekte werden **Merkmale** (Felder), **Eigenschaften**, **Handlungskompetenzen** (Methoden) und weitere (bisher noch nicht behandelte) Bestandteile deklariert bzw. definiert. Diese werden als **Member** der Klasse bezeichnet (dt.: *Mitglieder*).

In den Methoden eines Programms werden vordefinierte (z.B. der Standardbibliothek entstammende) oder selbst erstellte Klassen zur Erledigung von Aufgaben verwendet. Meist werden dabei Objekte aus Klassen mit Bauplancharakter erzeugt und mit Aufträgen versorgt. Mit dem „Beauftragen" eines Objekts oder einer Klasse bzw. mit dem „Zustellen einer Botschaft" ist nichts anderes gemeint als ein Methodenaufruf.

Nach unserer vorläufigen, auch im aktuellen Kapitel 4 zugrunde liegenden Konzeption besteht ein Programm aus ...

- Klassen und Objekten,
- die jeweils einen Zustand haben
- und Botschaften empfangen sowie senden können.

In der Hoffnung, dass die bisher präsentierten Eindrücke von der objektorientierten Programmierung (OOP) neugierig gemacht und nicht abgeschreckt haben, kommen wir nun zur systematischen Behandlung dieser Softwaretechnologie. Für die in Kapitel 1 speziell für größere Projekte empfohlene objektorientierte Analyse und Modellierung, z.B. mit Hilfe der Unified Modeling Language (UML), ist dabei leider keine Zeit (siehe z.B. Balzert 2011; Boch et al. 2007).

4.1 Überblick, historische Wurzeln, Beispiel

4.1.1 Einige Kernideen und Vorzüge der OOP

Lahres & Rayman (2009, Abschnitt 2) nennen in ihrem *Praxisbuch Objektorientierung* unter Berufung auf **Alan Kay**, der den Begriff *Objektorientierte Programmierung* geprägt und die objektorientierte Programmiersprache *Smalltalk* entwickelt hat, als unverzichtbare OOP-Grundelemente:

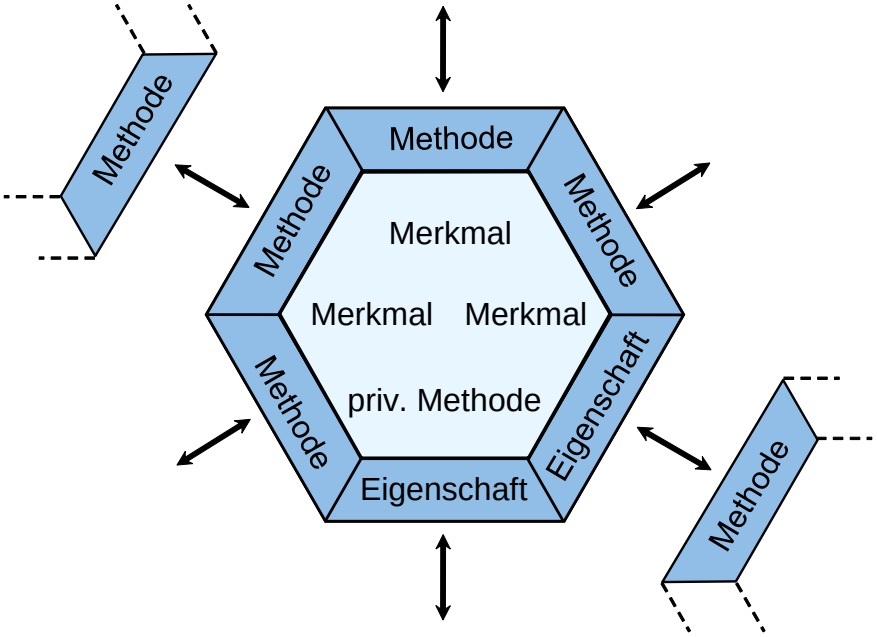
- **Datenkapselung**
  Mit diesem Thema haben wir uns bereits beschäftigt. Das vorhandene Wissen soll gleich vertieft und gefestigt werden.
- **Vererbung**
  Aus einer vorhandenen Klasse lassen sich zur Lösung neuer Aufgaben spezialisierte Klassen ableiten, die alle Member der Basisklasse erben. Hier findet eine Wiederverwendung von Software ohne lästiges und fehleranfälliges Kopieren von Quellcode statt. Beim Design der abgeleiteten Klasse kann man sich darauf beschränken, neue Member zu definieren oder bei manchen Erbstücken Modifikationen vorzunehmen, also z.B. Methoden situationsangepasst zu implementieren.

- **Polymorphie**
  Über Referenzvariablen vom Typ einer Basisklasse lassen sich auch Objekte von abgeleiteten Klassen verwalten, wobei selbstverständlich nur solche Methoden aufgerufen werden dürfen, die schon in der Basisklasse definiert sind. Ist eine solche Methode in abgeleiteten Klassen unterschiedlich implementiert, führt jedes per Basisklassenreferenz angesprochene Objekt sein angepasstes Verhalten aus. Derselbe Methodenaufruf hat also unterschiedliche (polymorphe) Verhaltensweisen zur Folge. Dank Polymorphie wird die Wiederverwendbarkeit einer Klasse weiter gesteigert, weil sie auch mit jüngeren (später entwickelten) Klassen kooperieren kann.

C# bietet sehr gute Voraussetzungen zur Nutzung dieser Konstruktionsprinzipien beim Entwurf von stabilen, wartungsfreundlichen, anpassungsfähigen und auf Wiederverwendung angelegten Softwaresystemen, kann aber keinen Entwickler zur Realisation der Prinzipien zwingen.

4.1.1.1 Datenkapselung und Modularisierung

In der objektorientierten Programmierung (OOP) wird die traditionelle Trennung von Daten und Operationen aufgegeben. Hier besteht ein Programm aus **Klassen**, die durch **Felder (also Daten) und Methoden (also Operationen)** sowie weitere Bestandteile definiert sind. Wie Sie bereits aus dem Einleitungsbeispiel wissen, steht in C# eine **Eigenschaft** für ein Paar von Zugriffsmethoden zum Lesen bzw. Verändern eines Feldes.[1] Eine Klasse wird in der Regel ihre Felder gegenüber anderen Klassen verbergen (**Datenkapselung**, **information hiding**) und so vor ungeschickten Zugriffen schützen. Die Methoden und Eigenschaften einer Klasse sind hingegen von außen ansprechbar und bilden ihre **Schnittstelle**. Dies kommt in der folgenden Abbildung zum Ausdruck, die Sie im Wesentlichen schon aus Abschnitt 1.1.1 kennen:



Es kann aber auch *private Methoden* für den ausschließlich internen Gebrauch geben. Ebenso sind *öffentliche Felder* möglich, die damit zur Schnittstelle einer Klasse gehören. Solche Felder sind oft als *konstant* deklariert (siehe Abschnitt 4.2.5) und damit vor Veränderungen geschützt. Ein Beispiel ist das **double**-Feld **PI** für die Kreiszahl π in der FCL-Klasse **Math**.

Klassen mit Datenkapselung realisieren besser als frühere Software-Technologien (siehe Abschnitt 4.1.2) das Prinzip der **Modularisierung**, das schon Julius Cäsar (100 v. Chr. - 44 v. Chr.) bei seiner

[1] Diese Darstellung ist etwas vereinfachend. Hinter einer Eigenschaft muss nicht unbedingt ein einzelnes Feld stehen.

beruflichen Tätigkeit als römischer Kaiser und Feldherr erfolgreich einsetzte (*Divide et impera!*).[1] Die Modularisierung ist ein probates, ja unverzichtbares Mittel der Software-Entwickler zur Bewältigung von umfangreichen Projekten.

Im Sinne einer gelungenen Modularisierung sind Klassen mit hoher Komplexität (vielfältigen Aufgaben) und auch Methoden mit hoher Komplexität zu vermeiden. Als eine Leitlinie für den Entwurf von Klassen genießt das von **Robert C. Martin**[2] erstmals formulierte Prinzip einer **einzigen Verantwortung** (engl.: ***S**ingle **R**esponsibility **P**rinciple*, SRP) bei den Vordenkern der objektorientierten Programmierung hohes Ansehen (siehe z.B. Lahres & Rayman 2009, Abschnitt 3.1). Multifunktionale Klassen tendieren zu stärkeren Abhängigkeiten von anderen Klassen, wobei die Wahrscheinlichkeit einer erfolgreichen Wiederverwendung sinkt. Ein negatives Beispiel wäre eine Klasse aus einem Personalverwaltungsprogramm, die sich sowohl um Gehaltsberechnungen als auch um die Datenbankverwaltung kümmert (Verbindung zum Datenbankserver herstellen, Fälle lesen und ablegen).

Aus der Datenkapselung und der Modularisierung ergeben sich gravierende Vorteile für die Softwareentwicklung:

- **Vermeidung von Fehlern**
  Direkte Schreibzugriffe auf die Felder einer Klasse bleiben den klasseneigenen Methoden vorbehalten, die vom Designer der Klasse sorgfältig entworfen wurden. Damit sollten Programmierfehler nur sehr selten auftreten. In unserer Beispielklasse `Bruch` haben wir dafür gesorgt, dass unter keinen Umständen der Nenner eines Bruches auf 0 gesetzt wird. Anwender unserer Klasse können einen Nenner einzig über die Eigenschaft `Nenner` verändern, die aber den Wert 0 nicht akzeptiert. Bei einer anderen Klasse kann es wichtig sein, dass für eine Gruppe von Feldern bei jeder Änderung gewissen Konsistenzbedingungen eingehalten werden.
- **Günstige Voraussetzungen für das Testen und die Fehlerbereinigung**
  Treten in einem Programm trotz Datenkapselung pathologische Variablenausprägungen auf, ist die Ursache relativ leicht aufzuklären, weil nur wenige Methoden verantwortlich sein können. Bei der Softwareentwicklung im professionellen Umfeld spielt das systematische Testen eines Programms (**Unit Testing**) eine entscheidende Rolle. Ein objektorientiertes Softwaresystem mit Datenkapselung und guter Modularisierung bietet günstige Voraussetzungen für eine möglichst umfassende Testung.
- **Innovationsoffenheit bei gekapselten Details einer Klassenimplementation**
  Verborgene Details einer Klassenimplementation kann der Designer ändern, ohne die Kooperation mit anderen Klasen zu gefährden.
- **Produktivität durch wiederholt und bequem verwendbare Klassen**
  Selbständig agierende Klassen, die ein Problem ohne überflüssige Anhängigkeiten von anderen Programmbestandteilen lösen, sind potenziell in vielen Projekten zu gebrauchen (Wiederverwendbarkeit). Wer als Programmierer eine Klasse verwendet, braucht sich um deren inneren Aufbau nicht zu kümmern, so dass neben dem Fehlerrisiko auch der Einarbeitungsaufwand sinkt. Wir werden z.B. in GUI-Programmen einen recht kompletten Rich-Text - Editor über eine Klasse aus der Standardbibliothek integrieren, ohne wissen zu müssen, wie Text und Textauszeichnungen intern verwaltet werden.

---

[1] Deutsche Übersetzung: Teile und herrsche!

[2] Der als *Uncle Bob* bekannte Software-Berater und Autor erläutert auf der folgenden Webseite seine Vorstellungen von objektorientierter Software: http://www.objectmentor.com/omSolutions/oops_what.html

- **Erfolgreiche Teamarbeit durch abgeschottete Verantwortungsbereiche**
  In großen Projekten können mehrere Programmierer nach der gemeinsamen Definition von Schnittstellen relativ unabhängig an verschiedenen Klassen arbeiten.

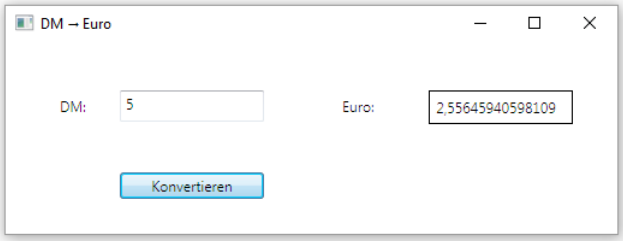
Durch die objektorientierte Programmierung werden auf vielfältige Weise **Kosten reduziert**:

- Vermeidung bzw. schnelles Aufklären von Programmierfehlern
- gute Chancen für die Wiederverwendung von Software
- gute Voraussetzungen für die Kooperation in Teams

4.1.1.2 Vererbung

Zu den Vorzügen der „super-modularen" Klassenkonzeption gesellt sich in der OOP ein Vererbungsverfahren, das beste Voraussetzungen für die Erweiterung von Softwaresystemen bei rationeller **Wiederverwendung** der bisherigen Code-Basis schafft: Bei der Definition einer neuen Klasse können alle Merkmale und Handlungskompetenzen (Methoden, Eigenschaften) einer *Basisklasse* übernommen werden. Es ist also leicht möglich, ein Softwaresystem um neue Klassen mit speziellen Leistungen zu erweitern. Durch systematische Anwendung des Vererbungsprinzips entstehen mächtige Klassenhierarchien, die in zahlreichen Projekten einsetzbar sind. Neben der direkten Nutzung vorhandener Klassen (über erzeugte Objekte oder statische Methoden) bietet die OOP mit der Vererbungstechnik eine weitere Möglichkeit zur Wiederverwendung von Software.

Bei dem in Abschnitt 2.2.4 erstellten Währungskonverter mit grafischer Benutzerschnittelle in WPF-Technik



haben wir mit Assistentenhilfe die Klasse `MainWindow` definiert als Ableitung der WPF-Klasse **Window**:[1]

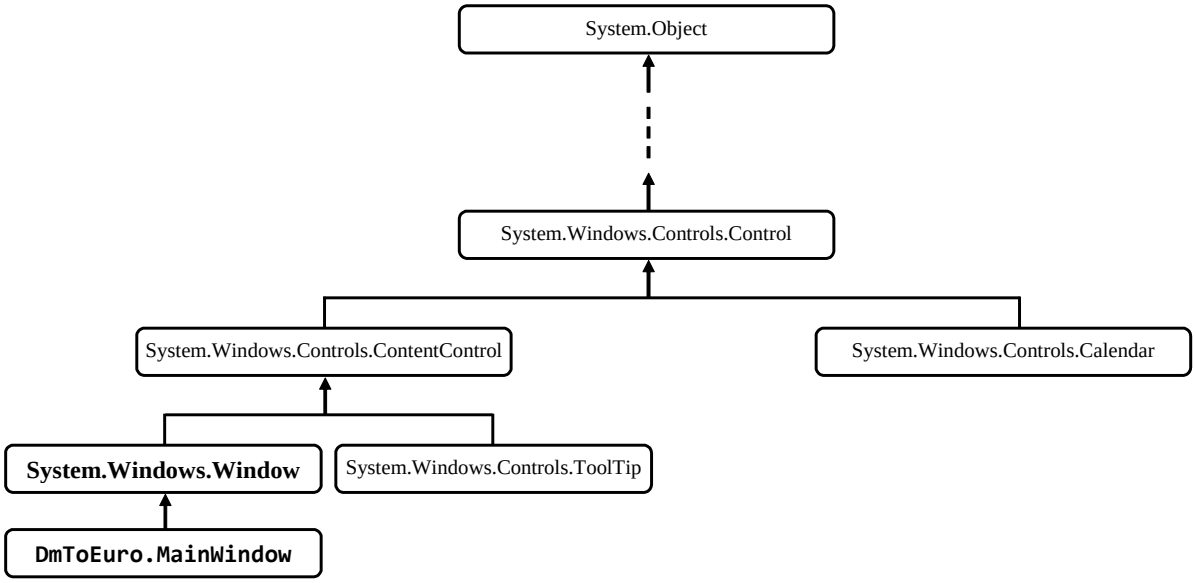
```
public partial class MainWindow : Window {
   public MainWindow() {
      InitializeComponent();
   }
   private void button_Click(object sender, RoutedEventArgs e) {
      double betrag = Convert.ToDouble(textBox.Text);
      label2.Content = Convert.ToString(betrag / 1.98853);
   }
}
```

Die Basisklasse **Window** ist selbst Bestandteil eines komplexen Stammbaums, von dem anschließend nur winziger Ausschnitt zu sehen ist:

---

[1] Die Klasse ist als **partial** gekennzeichnet, weil ihre Implementation auf zwei Dateien verteilt ist, um die Kooperation von Entwicklungsumgebung und Entwickler zu erleichtern:
- Die erste Quellcodedatei ist für den Programmierer zugänglich.
- Den restlichen Quellcode verwaltet die Entwicklungsumgebung in einer versteckten Datei.

Den Quellcode einer Klasse auf zwei Dateien aufzuteilen, ist im Normalfall nicht sinnvoll.



Im .NET - Framework wird das Vererbungsprinzip sogar auf die Spitze getrieben: Alle Klassen (und auch die Strukturen, siehe unten) stammen von der Urahnklasse **Object** ab, die an der Spitze des hierarchischen .NET - Klassensystems steht, das man seiner Universalität wegen als **Common Type System** (CTS, dt. *Allgemeines Typsystem*) bezeichnet.

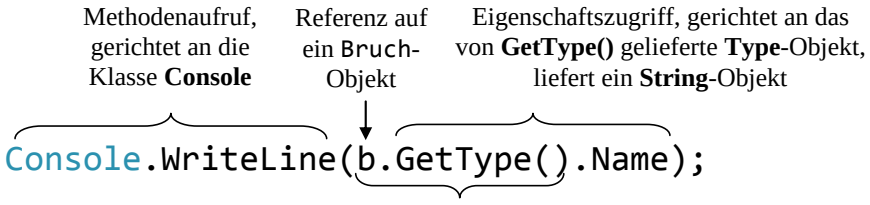
Weil sich im Handlungsrepertoire der Urahnklasse u.a. auch die Methode **GetType()** befindet, kann man beliebige .NET - Objekte und Strukturinstanzen (siehe unten) nach ihrem Datentyp befragen. Im folgenden Programm wird ein `Bruch`-Objekt (vgl. Abschnitt 1.1) um die entsprechende Auskunft gebeten:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Bruch b = new Bruch();` `  Console.WriteLine(b.GetType().Name);` `  }` `}`	`Bruch`

Von **GetType()** erhält man als Rückgabewert eine Referenz auf ein Objekt der Klasse **Type**. Über diese Referenz wird das **Type**-Objekt gebeten, den Wert seiner Eigenschaft **Name** mitzuteilen. Diese Zeichenfolge mit dem Typnamen (ein Objekt der Klasse **String**[1]) bildet schließlich den Parameter des **WriteLine()** - Aufrufs und landet auf der Konsole. In unserem Kursstadium ist es angemessen, die komplexe Anweisung unter Beteiligung von vier Klassen (**Console**, `Bruch`, **Type**, **String**), zwei Methoden (**WriteLine()**, **GetType()**), einer Eigenschaft (**Name**), einer expliziten Referenzvariablen (`b`) und einer impliziten Referenz (**GetType()** - Rückgabewert) genau zu erläutern:[2]

[1] Eine Besonderheit der Klasse **String** ist der aus Bequemlichkeitsgründen vom Compiler unterstützte Aliasname **string** (mit kleinem Anfangsbuchstaben).

[2] Der **Name**-Eigenschaftszugriff ist im Beispiel nicht unbedingt erforderlich, weil die Methode **WriteLine()** auch mit Objekten (z.B. aus der Klasse **Type**) umzugehen weiß und deren garantiert vorhandene, weil bereits in der Urahnklasse **Object** definierte, Methode **ToString()** nutzt, um sich ein ausgabefähiges **String**-Objekt zu besorgen.



Durch die technischen Details darf nicht der Blick auf das wesentliche Thema des aktuellen Abschnitts verstellt werden: Eine abgeleitete Klasse erbt die Merkmale und Handlungskompetenzen ihrer Basisklasse. Wenn diese Basisklasse ihrerseits abgeleitet wurde, kommen indirekt erworbene Erbstücke hinzu. Die als Beispiel betrachtete Klasse `Bruch` stammt direkt von der Klasse **Object** ab, und ihre Objekte beherrschen dank Vererbung die Methode **GetType()**, obwohl in der `Bruch`-Klassendefinition nichts davon zu sehen ist.

4.1.1.3 Polymorphie

Obwohl in unseren bisherigen Beispielen von Polymorphie noch nichts zu sehen war, soll doch versucht werden, die Kernidee hinter diesem Begriff schon jetzt zu vermitteln. In diesem Abschnitt sind einige Vorgriffe auf später ausführlich behandelte Begriffe erforderlich. Wer sich jetzt noch nicht stark für den Begriff der Polymorphie interessiert, kann den Abschnitt ohne Risiko für den weiteren Kursverlauf auslassen.

Beim Klassendesign ist generell das **Open-Closed - Prinzip** zu beachten:[1]

- Eine Klasse soll offen sein für die Verwendung zur Lösung neuer Aufgaben.
- Dabei darf es nicht erforderlich werden, vorhandenen Code zu verändern. Er soll abgeschlossen bleiben, möglichst für immer.

Es ist klar, dass für neue Aufgaben zusätzlicher Quellcode erforderlich wird. Es darf aber nicht passieren, dass bei der Anpassung eines Programms an neue Anforderungen vorhandener Quellcode modifiziert werden muss. In ungünstigen Fällen zieht jede Änderung weitere nach sich, so dass eine Kaskade von Anpassungen unter Beteiligung von vielen Klassen resultiert. Bei einem solchen Programm verursacht eine Anpassung an neue Aufgaben hohe Kosten und oft ein instabiles Ergebnis, so dass man in der Regel auf eine Anpassung verzichten wird.

Einen exzellenten Beitrag zur Erstellung von änderungsoffenem und doch abgeschlossenem Code leistet schon die Vererbungstechnik der OOP. Zur Modellierung einer neuen, spezialisierten Rolle kann man oft auf eine Basisklasse zurückgreifen und muss nur die zusätzlichen Merkmale und/oder Verhaltenskompetenzen ergänzen.

Dank Polymorphie kann eine Klasse nicht nur mit gleich alten oder älteren Klassen, die beim Design schon bekannt waren, zusammenarbeiten, sondern auch mit jüngeren Klassen. Um diese Offenheit für neue Aufgaben zu ermöglichen, verwendet man beim Klassendesign für Felder und Methodenparameter mit Referenztyp einen möglichst allgemeinen Datentyp, der die benötigten Verhaltenskompetenzen vorschreibt, aber keine darüber hinausgehende Einschränkung enthält.

In objektorientierten Programmiersprachen können über eine Referenzvariable Objekte vom deklarierten Typ *und von jedem abgeleiteten Typ* angesprochen werden. In einer abgeleiteten Klasse können nicht nur zusätzliche Methoden erstellt, sondern auch geerbte überschrieben werden, um das

---

[1] Das Open-Closed - Prinzip wird von Robert C. Martin (*Uncle Bob*) in einem Text erläutert, der über folgende Web-Adresse zu beziehen ist: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod

Verhalten an spezielle Voraussetzungen und Bedürfnisse anzupassen. Erfolgen Methodenaufrufe an Objekte aus verschiedenen abgeleiteten Klassen, welche jeweils die Methode überschrieben haben, unter Verwendung von Basisklassenreferenzen, dann zeigen die Objekte ihr artgerechtes Verhalten. Obwohl alle Objekte mit einer Referenz vom selben Basisklassentyp angesprochen wurden, verhalten sie sich unterschiedlich. Genau in dieser Situation spricht man von *Polymorphie*.

Wird z.B. in einer Klasse zur Verwaltung von geometrischen Objekten eine Referenzvariable vom relativ allgemeinen Typ `Figur` deklariert und beim Aufruf der Methode `MeldeInhalt()` verwendet, führt das angesprochene Objekt, das bei einem konkreten Programmeinsatz z.B. aus der abgeleiteten Klasse `Kreis` oder `Rechteck` stammt, seine spezifischen Berechnungen durch. Die Klasse zur Verwaltung von geometrischen Objekten kann ohne Quellcodeänderungen mit beliebigen, eventuell sehr viel später definierten `Figur`-Ableitungen kooperieren.

Weil in der allgemeinen Klasse `Figur` keine Inhaltsberechnungsmethode realisiert werden kann, verzichtet man auf eine Implementation, wobei eine so genannte *abstrakte* Methode entsteht. Enthält eine Klasse mindestens eine abstrakte Methode, ist sie ihrerseits abstrakt und kann nicht zum Erzeugen von Objekten genutzt werden. Eine abstrakte Klasse ist aber gleichwohl als Datentyp erlaubt und spielt eine wichtige Rolle bei der Realisation von Polymorphie.[1]

Dank Vererbung und Polymorphie kann objektorientierte Software **anpassungs- und erweiterungsfähig** bei weitgehend fixiertem Bestands-Code, also unter Beachtung des Open-Closed - Prinzips, gestaltet werden.

4.1.1.4 Realitätsnahe Modellierung

Klassen sind nicht nur ideale Bausteine für die rationelle Konstruktion von Software-Systemen, sondern erlauben auch eine gute Modellierung des Anwendungsbereichs. In der zentralen Projektphase der objektorientierten Analyse und Modellierung sprechen Softwareentwickler und Auftraggeber dieselbe Sprache, so dass Kommunikationsprobleme weitgehend vermieden werden.

Neben den Klassen zur Modellierung von Akteuren oder Ereignissen des realen Anwendungsbereichs sind bei einer typischen Anwendung aber auch zahlreiche Klassen beteiligt, die Akteure oder Ereignisse der virtuellen Welt des Computers repräsentieren (z.B. Bildschirmfenster, Mausereignisse).

4.1.2 Strukturierte Programmierung und OOP

In vielen älteren Programmiersprachen (z.B. C, Fortran, Pascal) sind zur Strukturierung von Programmen zwei Techniken verfügbar, die in weiterentwickelter Form auch bei der OOP genutzt werden:

- **Unterprogramme**
  Man zerlegt ein Gesamtproblem in mehrere Teilprobleme, die jeweils in einem eigenen *Unterprogramm* gelöst werden. Wird die von einem Unterprogramm erbrachte Leistung wiederholt (an verschiedenen Stellen eines Programms) benötigt, muss jeweils nur ein Aufruf mit dem Namen des Unterprogramms und passenden Parametern eingefügt werden. Durch diese Strukturierung ergeben sich kompaktere und übersichtlichere Programme, die leichter erstellt, analysiert, korrigiert und erweitert werden können. Praktisch alle traditionellen Programmiersprachen unterstützen solche Unterprogramme (Subroutinen, Funktionen, Proze-

[1] Neben den abstrakten Klassen, die mindestens *eine* abstrakte Methode (Definitionskopf ohne Implementation) enthalten, spielen bei der Polymorphie auch die so genannten *Schnittstellen* eine wichtige Rolle als veränderungsoffene Datentypen. Eine Schnittstelle kann näherungsweise als Klasse mit *ausschließlich* abstrakten Methoden charakterisiert werden. Abstrakte Klassen und Schnittstellen (Interfaces) werden später ausführlich behandelt.

duren), und meist stehen auch umfangreiche Bibliotheken mit fertigen Unterprogrammen für diverse Standardaufgaben zur Verfügung. Beim Einsatz einer Unterprogrammsammlung klassischer Art muss der Programmierer passende Daten bereitstellen, auf die dann vorgefertigte Routinen losgelassen werden. Der Programmierer hat also seine Datensammlung *und* das Arsenal der verfügbaren Unterprogramme (aus fremden Quellen oder selbst erstellt) zu verwalten und zu koordinieren.

- **Problemadäquate Datentypen**
  Zusammengehörige Daten unter *einem* Variablennamen ansprechen zu können, vereinfacht das Programmieren erheblich. Mit dem Datentyp **struct** der Programmiersprache C oder dem analogen Datentyp **record** der Programmiersprache Pascal lassen sich problemadäquate Datentypen mit mehreren Bestandteilen konstruieren, die jeweils einen beliebigen, bereits bekannten Typ haben dürfen. So eignet sich etwa für ein Programm zur Adressenverwaltung ein neu definierter Datentyp mit Feldern für Name, Vorname, Telefonnummer etc. Alle Adressinformationen zu einer Person lassen sich dann in *einer* Variablen vom selbst definierten Typ speichern. Dies vereinfacht z.B. das Lesen, Kopieren oder Schreiben solcher Daten.

Die problemadäquaten Datentypen der älteren Programmiersprachen werden in der OOP durch *Klassen* ersetzt, wobei diese Datentypen nicht nur durch eine Anzahl von *Merkmalen* (Feldern) beliebigen Typs charakterisiert sind, sondern auch *Handlungskompetenzen* (Methoden, Eigenschaften) besitzen, welche die Aufgaben der Funktionen bzw. Prozeduren der älteren Programmiersprachen übernehmen.

Im Vergleich zur strukturierten Programmierung bietet die OOP u.a. folgende Fortschritte:

- Optimierte Modularisierung mit Zugriffsschutz
  Die Daten sind sicher in Objekten gekapselt, während sie bei traditionellen Programmiersprachen entweder als globale Variablen allen Missgriffen ausgeliefert sind oder zwischen Unterprogrammen „wandern“ (Goll et al. 2000, S. 21), was bei Fehlern zu einer aufwändigen Suche entlang der Verarbeitungskette führen kann.
- Rationelle (Weiter-)Entwicklung von Software nach dem Open-Closed - Prinzip durch Vererbung und Polymorphie
- Bessere Abbildung des Anwendungsbereichs
- Mehr Bequemlichkeit für Bibliotheksbenutzer
  Jede rationelle Softwareproduktion greift in hohem Maß auf Bibliotheken mit bereits vorhandenen Lösungen zurück. Dabei sind die Klassenbibliotheken der OOP einfacher zu verwenden als klassische Unterprogrammbibliotheken.

4.1.3 Auf-`Bruch` zu echter Klasse

In den Beispielprogrammen von Kapitel 3 wurde mit der Klassendefinition lediglich eine in C# unausweichliche formale Anforderung an Programme erfüllt. Die in Abschnitt 1.1 vorgestellte Klasse `Bruch` realisiert hingegen wichtige Prinzipien der objektorientierten Programmierung. Sie wird nun wieder aufgegriffen und in verschiedenen Varianten bzw. Ausbaustufen als Beispiel verwendet. Auf der Klasse `Bruch` basierende Programme sollen Schüler beim Erlernen der Bruchrechnung unterstützen. Eine objektorientierte Analyse der Problemstellung hat ergeben, dass in einer elementaren Ausbaustufe des Programms lediglich eine Klasse zur Repräsentation von Brüchen benötigt wird. Später sind weitere Klassen (z.B. Aufgabe, Übungsaufgabe, Testaufgabe, Schüler, Lernepisode, Testepisode, Fehler) zu ergänzen.

Wir nehmen nun bei der `Bruch`-Klassendefinition im Vergleich zur Variante in Abschnitt 1.1 einige Verbesserungen vor:

- Als zusätzliches Feld erhält jeder Bruch ein `etikett` vom Datentyp der Klasse **String**. Damit wird eine beschreibende Zeichenfolge verwaltet, die z.B. beim Aufruf der Methode `Zeige()` zusätzlich zu den Feldern `zaehler` und `nenner` auf dem Bildschirm erscheint. Objekte der erweiterten Klasse `Bruch` besitzen also auch eine Instanzvariable mit *Referenztyp* (neben den Feldern `zaehler` und `nenner` vom elementaren Typ **int**).
- Weil die Klasse `Bruch` ihre Merkmale kapselt, also fremden Klassen keine *direkten* Zugriffe erlaubt, stellt sie auch für das Feld `etikett` eine Eigenschaft namens `Etikett` (mit großem Anfangsbuchstaben) für das Lesen und das (kontrollierte) Verändern des Feldes zur Verfügung.
- Wir erlauben uns einen erneuten Vorgriff auf die später noch ausführlich zu diskutierende Ausnahmebehandlung per **try-catch** - Anweisung, um in der `Bruch`-Methode `Frage()` sinnvoll auf fehlerhafte Benutzereingaben reagieren zu können. Die kritischen Aufrufe der **Convert**-Methode **ToInt32()** finden nun innerhalb eines **try**-Blocks statt. Bei einem Ausnahmefehler aufgrund einer irregulären Eingabe wird daher *nicht* mehr das Programm beendet, sondern der zugehörige **catch**-Block ausgeführt. Damit die Methode `Frage()` den Aufrufer über eine reibungslose oder verpatzte Ausführung informieren kann, wechselt der Rückgabetyp von **void** zu **bool**. Mit der **return**-Anweisung in der verbesserten `Frage()` - Variante wird nach einer erfolgreichen Ausführung der Wert **true** bzw. nach dem Auftreten einer Ausnahme der Wert **false** zurückgemeldet.
- In der Methode `Kuerze()` wird die performante Modulo-Variante von Euklids Algorithmus zur Bestimmung des größten gemeinsamen Teilers von zwei ganzen Zahlen verwendet (siehe Übungsaufgabe in Abschnitt 3.7.4).

Im folgenden Quellcode der erweiterten Klasse `Bruch` sind die unveränderten Methoden bzw. Eigenschaften gekürzt wiedergegeben:

```
using System;
public class Bruch {
   int zaehler,             // zaehler wird automatisch mit 0 initialisiert
       nenner = 1;
   string etikett = "";     // die Ref.typ-Init. auf null wird ersetzt, siehe Text

   public int Zaehler {
      . . .
   }

   public int Nenner {
      . . .
   }

   public string Etikett {
      get {
         return etikett;
      }
      set {
         if (value.Length <= 40)
            etikett = value;
         else
            etikett = value.Substring(0, 40);
      }
   }
```

```
    public void Kuerze() {
       // größten gemeinsamen Teiler mit dem Euklids Algorithmus bestimmen
       // (performante Variante mit Modulo-Operator)
       if (zaehler != 0) {
          int rest;
          int ggt = Math.Abs(zaehler);
          int divisor = Math.Abs(nenner);
          do {
             rest = ggt % divisor;
             ggt = divisor;
             divisor = rest;
          } while (rest > 0);
          zaehler /= ggt;
          nenner /= ggt;
       } else
          nenner = 1;
}

    public void Addiere(Bruch b) {
       . . .
    }

    public bool Frage() {
       try {
          Console.Write("Zaehler: ");
          int z  = Convert.ToInt32(Console.ReadLine());
          Console.Write("Nenner : ");
          int n = Convert.ToInt32(Console.ReadLine());
          Zaehler = z;
          Nenner = n;
          return true;
       } catch {
          return false;
       }
    }

    public void Zeige() {
       string luecke = "";
       for (int i=1; i <= etikett.Length; i++)
          luecke = luecke + " ";
       Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                         luecke, zaehler, etikett, nenner);
    }
}
```

Im Unterschied zur Präsentation in Abschnitt 1.1 wird die Definition der Klasse `Bruch` anschließend gründlich erläutert. Dabei machen die in Abschnitt 4.2 behandelten Instanzvariablen (Felder) relativ wenig Mühe, weil wir viele Details schon von den *lokalen* Variablen her kennen. Bei den Methoden gibt es mehr Neues zu lernen, so dass wir uns in Abschnitt 4.3 auf elementare Themen beschränken und später noch wichtige Ergänzungen vornehmen.

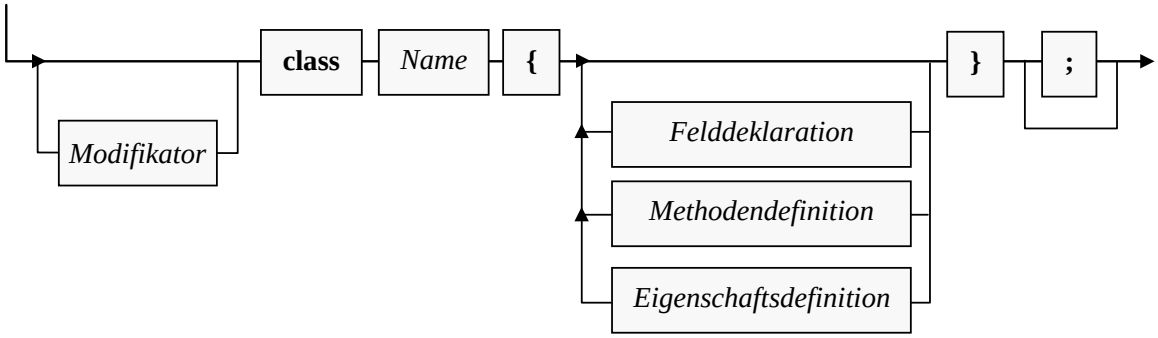
Jede .NET - Anwendung benötigt eine Startklasse mit statischer **Main()** - Methode, die von der CLR (*Common Language Runtime*) beim Programmstart aufgerufen wird. Für die bei diversen Demonstrationen in den folgenden Abschnitten verwendeten Startklassen (mit jeweils spezieller Implementation) werden wir stets den Namen `Bruchrechnung` verwenden, z.B.:

```
using System;
class Bruchrechnung {
   static void Main() {
      Console.WriteLine("Kürzen von Brüchen\n-----------------\n");
      Bruch b = new Bruch();
      b.Frage();
      b.Kuerze();
      b.Etikett = "Der gekürzte Bruch:";
      b.Zeige();
   }
}
```

In der **Main()** - Methode dieses Programms zum Kürzen von Brüchen wird ein Objekt aus der Klasse `Bruch` erzeugt und mit Aufträgen versorgt.

Wir arbeiten im Wesentlichen weiterhin mit dem aus Abschnitt 3.1.2.1 bekannten Syntaxdiagramm zur Klassendefinition, das aus didaktischen Gründen einige Vereinfachungen enthält:

Klassendefinition



Bemerkungen zum Kopf einer Klassendefinition:

- Im Beispiel ist die Klasse `Bruch` als **public** definiert, damit sie uneingeschränkt von anderen Klassen (aus beliebigen Assemblies) genutzt werden kann. Weil bei der startfähigen Klasse `Bruchrechnung` eine solche Nutzung nicht in Frage kommt, wird hier auf den (zum Starten durch die CLR *nicht* erforderlichen) Zugriffsmodifikator **public** verzichtet.
- Klassennamen beginnen einer allgemein akzeptierten C# - Konvention folgend mit einem Großbuchstaben. Besteht ein Name aus mehreren Wörtern (z.B. `ArithmeticException`), schreibt man der besseren Lesbarkeit wegen die Anfangsbuchstaben aller Wörter groß (*Pascal-Casing*, siehe Abschnitt 3.1.5).
- Am Ende der Klassendefinition darf optional ein Semikolon stehen.

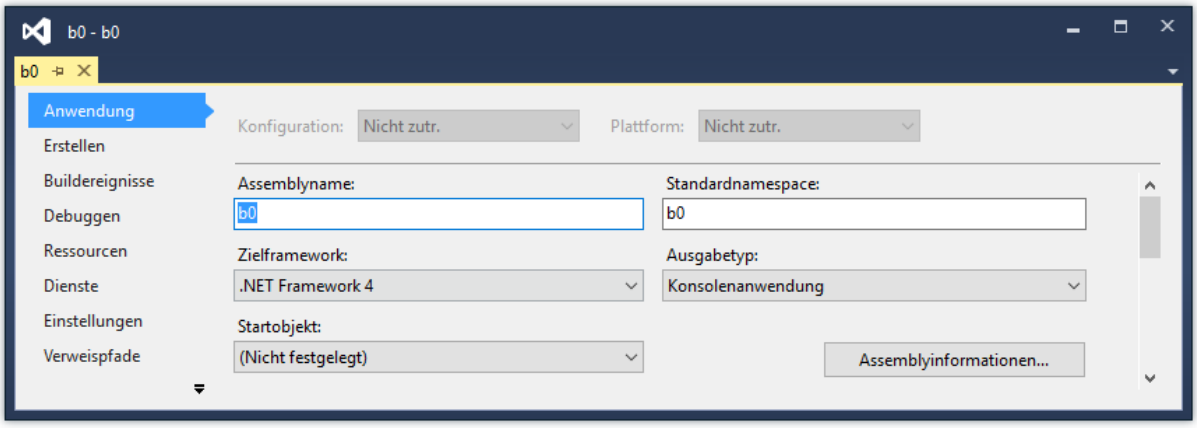
Hinsichtlich der Dateiverwaltung wird vorgeschlagen:

- Die Klassendefinitionen (z.B. `Bruch` und `Bruchrechnung`) sollten jeweils in einer eigenen Datei gespeichert werden.
- Den Namen dieser Datei sollte man aus dem Klassennamen durch Anhängen der Erweiterung **.cs** bilden.

Beim gemeinsamen Übersetzen der Quellcode-Dateien **Bruch.cs** und **Bruchrechnung.cs** entsteht *ein* Exe-Assembly, dessen Namen man im Visual Studio nach dem Menübefehl

Projekt > Eigenschaften

ändern kann, wenn die vom Assistenten für neue Projekte vergebene Voreinstellung nicht (mehr) gefällt, z.B.:



Im Beispiel entsteht die Assembly-Datei **b0.exe**.

4.2 Instanzvariablen

Die Instanzvariablen (bzw. -felder) einer Klasse besitzen viele Gemeinsamkeiten mit den *lokalen* Variablen, die wir im Kapitel 3 über elementare Sprachelemente ausführlich behandelt haben, doch gibt es auch wichtige Unterschiede, die im Mittelpunkt des aktuellen Abschnitts stehen. Unsere Klasse `Bruch` besitzt nach der Erweiterung um ein beschreibendes Etikett die folgenden Instanzvariablen:

- `zaehler` (Datentyp **int**)
- `nenner` (Datentyp **int**)
- `etikett` (Datentyp **String**)

Zu den beiden Feldern `zaehler` und `nenner` vom elementaren Datentyp **int** ist das Feld `etikett` mit dem Referenzdatentyp **String** dazugekommen. Jedes nach dem `Bruch`-Bauplan geschaffene Objekt erhält seine *eigene* Ausstattung mit diesen Variablen.

4.2.1 Sichtbarkeitsbereich, Existenz und Ablage im Hauptspeicher

Von den lokalen Variablen unterscheiden sich die Instanzvariablen (Felder) einer Klasse vor allem bei der *Zuordnung* (vgl. Abschnitt 3.3.3):

- lokale Variablen gehören zu einer *Methode* (oder *Eigenschaft*)
- Instanzvariablen gehören zu einem *Objekt*

Daraus ergeben sich gravierende Unterschiede in Bezug auf den Sichtbarkeitsbereich, die Lebensdauer und die Ablage im Hauptspeicher:

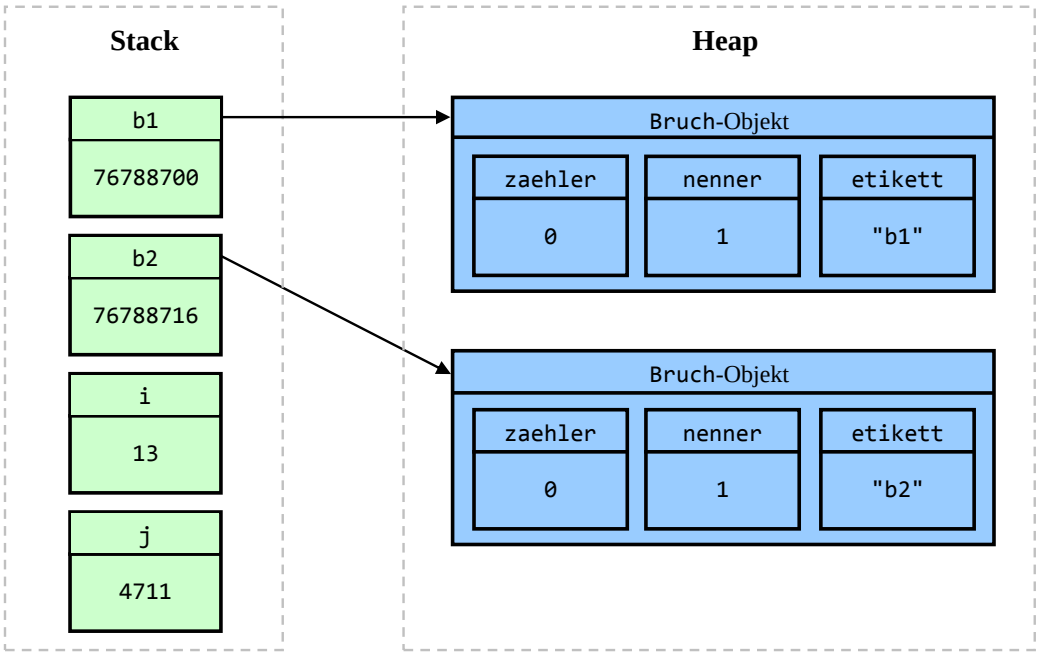
	lokale Variable	**Instanzvariable**
Sichtbarkeit	Eine lokale Variable ist nur in ihrer eigenen Methode sichtbar. Nach der Deklarationsanweisung kann sie in den restlichen Anweisungen des lokalsten Blocks angesprochen werden. Ein eingeschachtelter Block gehört zum Sichtbarkeitsbereich des umgebenden Blocks.	Die Instanzvariablen eines existenten Objekts sind in einer Methode sichtbar, wenn ... • der Zugriff erlaubt ist (siehe Abschnitt 4.10) • eine Referenz zum Objekt vorhanden ist Instanzvariablen werden in klasseneigenen Instanzmethoden durch gleichnamige lokale Variablen überlagert, können jedoch über das vorgeschaltete Schlüsselwort **this** weiter angesprochen

		werden (siehe Abschnitt 4.4.5.3).
Lebensdauer	Sie existiert nur während der Ausführung der zugehörigen Methode.	Für jedes neue Objekt wird ein Satz mit allen Instanzvariablen seiner Klasse erzeugt. Die Instanzvariablen existieren bis zum Ableben des Objekts. Ein Objekt wird zur Entsorgung freigegeben, sobald keine Referenz auf das Objekt mehr vorhanden ist.
Ablage im Speicher	Sie wird auf dem so genannten **Stack** (deutsch: *Stapel*) abgelegt. Innerhalb des programmeigenen Speichers dient dieses Segment zur Verwaltung von Methodenaufrufen.	Die Objekte landen mit ihren Instanzvariablen in einem Bereich des programmeigenen Speichers, der als **Heap** (deutsch: *Haufen*) bezeichnet wird.

Während die folgende **Main()** - Methode

```
class Bruchrechnung {
  static void Main() {
    Bruch b1 = new Bruch(), b2 = new Bruch();
    int i = 13, j = 4711;
    b1.Etikett = "b1";
    b2.Etikett = "b2";
    . . .
  }
}
```

ausgeführt wird, befinden sich auf dem Stack die lokalen Variablen `b1`, `b2`, `i` und `j`. Die beiden `Bruch`-Referenzvariablen (`b1`, `b2`) zeigen jeweils auf ein `Bruch`-Objekt auf dem Heap, das einen kompletten Satz der `Bruch`-Instanzvariablen besitzt:[1]



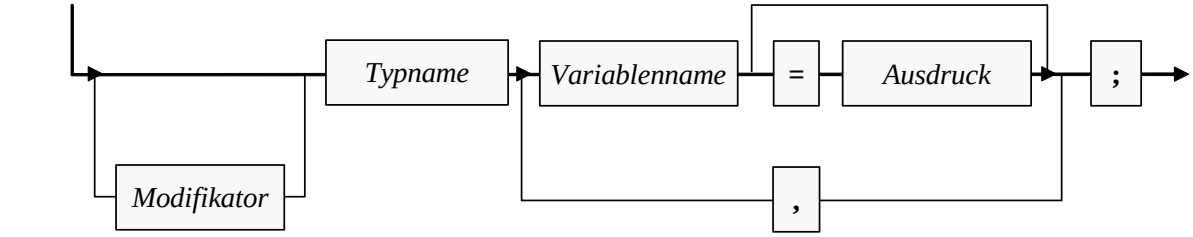
[1] Hier wird aus didaktischen Gründen ein wenig gemogelt: Die beiden Etiketten sind selbst Objekte und liegen „neben“ den `Bruch`-Objekten auf dem Heap. In jedem `Bruch`-Objekt befindet sich eine Referenz-Instanzvariable namens `etikett`, die auf das zugehörige **String**-Objekt zeigt.

4.2.2 Deklaration mit Modifikatoren für den Zugriffsschutz und andere Zwecke

Während lokale Variablen im Anweisungsteil einer Methode (oder Eigenschaft) deklariert werden, erscheinen die Deklarationen der Instanzvariablen in der Klassendefinition *außerhalb* jeder Methoden- oder Eigenschaftsdefinition. Man sollte die Instanzvariablen der Übersichtlichkeit halber am Anfang der Klassendefinition deklarieren, wenngleich der Compiler auch ein späteres Erscheinen akzeptiert.

Für die Deklaration einer lokalen Variablen haben wir **const** als einzigen Modifikator kennen gelernt und in einem speziellen Syntaxdiagramm beschrieben (siehe Abschnitt 3.3.8). Dieser Modifikator ist auch bei Instanzvariablen erlaubt (siehe Abschnitt 4.2.5). Außerdem kommen hier weitere Modifikatoren in Frage, die z.B. zur Spezifikation der **Schutzstufe** dienen. Insgesamt ist es sinnvoll, in das Syntaxdiagramm zur Deklaration von Instanzvariablen den allgemeinen Begriff des Modifikators aufzunehmen:[1]

Deklaration von Instanzvariablen



In C# besitzen alle Instanzvariablen per Voreinstellung die Schutzstufe **private**, so dass sie nur in klasseneigenen Methoden bzw. Eigenschaften angesprochen werden können. Weil bei den `Bruch`-Feldern diese voreingestellte Datenkapselung erwünscht ist, kommen hier die Felddeklarationen ohne Modifikatoren aus:

```
int zaehler,
    nenner = 1;
string etikett = "";
```

Um fremden Klassen trotzdem einen (allerdings kontrollierten!) Zugang zu den `Bruch`-Instanzvariablen zu ermöglichen, ist jeweils eine zugehörige Eigenschaft vorhanden.

Auf den ersten Blick scheint die Datenkapselung nur beim Nenner eines Bruches relevant zu sein, doch auch bei den restlichen Instanzvariablen bringt sie potentiell Vorteile:

- Zugunsten einer übersichtlichen Bildschirmausgabe soll das Etikett auf 40 Zeichen beschränkt bleiben. Mit Hilfe der Eigenschaft `Etikett()` kann dies gewährleistet werden.
- Abgeleitete (erbende) Klassen (siehe unten) können in die Eigenschaften `Zaehler` und `Nenner` neben der Null-Überwachung für den Nenner noch weitere Intelligenz einbauen und z.B. mit speziellen Aktionen reagieren, wenn der Wert auf eine Primzahl gesetzt wird.

Trotz ihrer überzeugenden Vorteile soll die Datenkapselung nicht zum Dogma erhoben werden. Sie ist überflüssig, wenn bei einem Feld Lese- und Schreibzugriffe uneingeschränkt erlaubt sein sollen und auch die eben im Beispiel angedachte Option, bestimmet Wertzuweisungen zu einem Ereignis zu machen, nicht von Interesse ist. Um allen Klassen den Direktzugriff auf ein Feld zu erlauben, wird in seiner Deklaration der Modifikator **public** angegeben, z.B.:

```
public int Delta;
```

In Abschnitt 4.10 finden Sie eine Tabelle mit allen verfügbaren Schutzstufen und zugehörigen Modifikatoren.

---

[1] Es ist sinnlos und verboten, einen Modifikator *mehrfach* auf eine Instanzvariable anzuwenden. Im Syntaxdiagramm zur Instanzvariablendeklaration wird der Einfachheit halber darauf verzichtet, die Mehrfachvergabe durch eine aufwendige Darstellungstechnik zu verbieten.

Für die Benennung von Instanzvariablen werden folgende Regeln empfohlen (vgl. Mössenböck 2016, S. 14):

- Für Felder mit den Schutzstufen **private**, **protected** oder **internal** wird das *Camel Casing* verwendet (z.B. `currentSpeed`). Oft tritt ein klein geschriebenes privates Feld (z.B. `nenner`) als Hintergrund zu einer groß geschriebenen öffentlichen Eigenschaft auf (z.B. `Nenner`).
- Für die Felder mit der Schutzstufe **public** wird das *Pascal Casing* verwendet (z.B. `MinValue`).

Um private Instanzvariablen gut von lokalen Variablen und Formalparametern (siehe Abschnitt 4.3.1.3) unterscheiden zu können, verwenden manche Programmierer ein Präfix, z.B.:

```
int m_nenner; //das m steht für "Member"
int _nenner;
```

4.2.3 Initialisierung

Während bei lokalen Variablen auf jeden Fall der Programmierer für eine Initialisierung sorgen muss, erhalten die Instanzvariablen eines neuen Objekts automatisch die folgenden Voreinstellungswerte, wenn der Programmierer nicht eingreift:

Datentyp	**Voreinstellungswert**
sbyte, **byte**, **short**, **ushort**, **int**, **uint**, **long**, **ulong**	0
float, **double**, **decimal**	0.0
char	0 (Unicode-Zeichennummer)
bool	**false**
Referenztyp	**null**

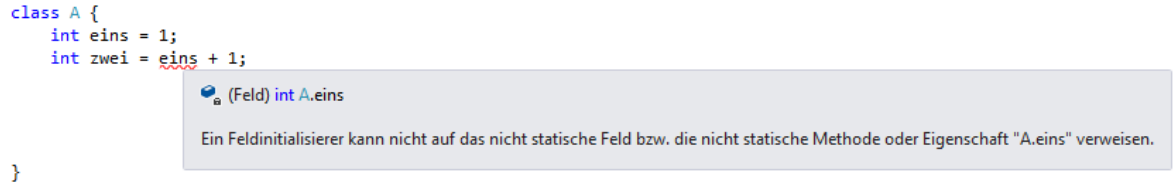
In der Klasse `Bruch`

```
int zaehler,
    nenner = 1;
string etikett = "";
```

wird nur die automatische `zaehler`-Initialisierung unverändert übernommen, denn:

- Beim `nenner` eines Bruches wäre die Initialisierung auf 0 bedenklich, weshalb eine explizite Initialisierung auf den Wert 1 vorgenommen wird.
- Wie noch näher zu erläutern sein wird, ist **string** in C# *kein* primitiver Datentyp, sondern eine *Klasse*, wobei der Compiler als Typbezeichner neben dem Klassennamen **String** auch den Alias **string** (mit kleinem Anfangsbuchstaben) akzeptiert. Variablen von diesem Typ können einen Verweis auf ein Objekt aus dieser Klasse aufnehmen. Solange kein zugeordnetes Objekt existiert, hat eine **String**-Instanzvariable den Wert **null**, zeigt also auf nichts. Weil der `etikett`-Wert **null** z.B. beim Aufruf der `Bruch`-Methode `Zeige()` einen Laufzeitfehler (**NullReferenceException**) zu Folge hätte, wird ein **String**-Objekt mit einer leeren Zeichenfolge erstellt und zur `etikett`-Initialisierung verwendet. Das Erzeugen des **String**-Objekts erfolgt *implizit* (ohne **new**-Operator, siehe unten), indem der **String**-Variablen `etikett` ein Zeichenfolgen-Literal zugewiesen wird.

Bei der optionalen Initialisierung von Instanzvariablen im Rahmen ihrer Deklaration ist es nicht erlaubt, auf andere Instanzvariablen desselben Objekts zuzugreifen, z.B.:



Im Rahmen eines Konstruktors (siehe Abschnitt 4.4.3) ist eine solche Initialisierung hingegen möglich.

4.2.4 Zugriff in klasseneigenen und fremden Methoden

In den Instanzmethoden einer Klasse können die Instanzvariablen des *aktuellen* (die Methode ausführenden) Objekts direkt über ihren Namen angesprochen werden, was z.B. in der `Bruch`-Methode `Zeige()` zu beobachten ist:

```
Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                  luecke, zaehler, etikett, nenner);
```

Im Beispiel zeigt sich syntaktisch kein Unterschied zwischen dem Zugriff auf die Instanzvariablen (`zaehler`, `nenner`, `etikett`) und dem Zugriff auf die lokale Variable `luecke`.

Gelegentlich kann es sinnvoll oder erforderlich sein, einem Instanzvariablennamen über das Schlüsselwort **this** (vgl. Abschnitt 4.4.5.3) eine Referenz auf das handelnde Objekt voranzustellen, wobei das Schlüsselwort und der Feldname durch den **Punktoperator** zu trennen sind:

- Das kann optional der Klarheit halber geschehen, z.B.:
  ```
  Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                    luecke, this.zaehler, this.etikett, this.nenner);
  ```
- Instanzvariablen werden durch gleichnamige lokale Variablen oder Methodenparameter (siehe Abschnitt 4.3) überlagert, können jedoch in dieser (besser zu vermeidenden) Situation über das vorgeschaltete Schlüsselwort **this** weiter angesprochen werden.

Beim Zugriff auf eine Instanzvariable eines *anderen* Objektes *derselben* Klasse muss dem Variablennamen eine Referenz auf das Objekt vorangestellt werden, wobei die Bezeichner durch den Punktoperator zu trennen sind. In der folgenden Anweisung aus der `Bruch`-Methode `Addiere()` greift das handelnde Objekt lesend auf die Instanzvariablen eines anderen `Bruch`-Objekts zu, das über die Referenzvariable `b` angesprochen wird:

```
zaehler = zaehler * b.nenner + b.zaehler * nenner;
```

In einer *statischen* Methode der eigenen Klasse muss zum Zugriff auf eine Instanzvariable eines konkreten Objekts natürlich eine Referenz auf dieses Objekt vorhanden sein und dem Instanzvariablennamen vorangestellt werden (getrennt durch den Punktoperator).

Direkte Zugriffe auf die Instanzvariablen eines Objekts durch Methoden *fremder* Klassen sind zwar nicht grundsätzlich verboten, verstoßen aber gegen das Prinzip der Datenkapselung, das in der OOP von zentraler Bedeutung ist. Würden die `Bruch`-Instanzvariablen mit dem Modifikator **public**, also ohne Datenkapselung deklariert, dann könnte z.B. der Nenner eines Bruches in der **Main**() - Methode der fremden Klasse `Bruchrechnung` direkt angesprochen werden:

```
Console.WriteLine(b.nenner);
b.nenner = 0;
```

In der von uns tatsächlich realisierten `Bruch`-Definition werden solche Zu- bzw. Fehlgriffe jedoch vom Compiler verhindert, z.B.:

```
Der Zugriff auf "Bruch.nenner" ist aufgrund der Sicherheitsebene nicht
möglich
```

4.2.5 Wertfixierung zur Übersetzungszeit oder nach der Initialisierung

Neben der Schutzstufenwahl gibt es weitere Anlässe für den Einsatz von Modifikatoren in Felddeklarationen. Mit dem Modifikator **const** können nicht nur lokale Variablen (siehe Abschnitt 3.3.8) sondern auch Felder einer Klasse als konstant deklariert werden, wobei der initialisierende Ausdruck *zur Übersetzungszeit* berechenbar sein muss.

Als Datentypen sind für Felder mit **const**-Deklaration ausschließlich die elementaren Datentypen und der Typ **String** erlaubt bzw. sinnvoll.[1]

Konstante Felder einer Klasse sind grundsätzlich *statisch*, wobei der überflüssige Modifikator **static** *nicht* angegeben werden darf. Genau genommen passt die Behandlung des Feld-Modifikators **const** also nicht in den Abschnitt 4.2 über Instanzvariablen.

Soll eine Instanzvariable *zur Laufzeit* in einem so genannten Konstruktor (siehe Abschnitt 4.4.3) initialisiert und danach fixiert werden, verwendet man den Modifikator **readonly**, z.B.:

Quellcode	Ausgabe
`using System;` `public class ReadonlyFraction {` `  public readonly int Num, Denom;` `  public ReadonlyFraction(int n, int d) {` `  Num = n;` `  Denom = d != 0 ? d : 1;` `  }` `}` `class Prog {` `  static void Main() {` `  ReadonlyFraction b = new ReadonlyFraction(1, 7);` `  Console.WriteLine(b.Denom);` `  //b.Denom = 13; // verboten` `  }` `}`	`7`

Bei den Konstruktoren handelt es sich um spezielle Methoden, die den Namen ihres Typs tragen und keinen Rückgabetyp besitzen (auch nicht **void**).

Unterschiede zwischen der **const**- und der **readonly**-Deklaration:

---

[1] Begründung:

- Nicht-elementare Strukturen (siehe unten) sind als Datentypen verboten, weil deren Konstruktor zur Laufzeit ausgeführt wird, was dem Prinzip der Berechenbarkeit zur Übersetzungszeit widerstrebt.
- Referenztypen außer **String** sind zwar erlaubt, aber sinnlos, weil bei der obligatorischen Initialisierung nur das Referenzliteral **null** möglich ist.

	const	**readonly**
Erlaubte Datentypen	Elementare Datentypen und **String**	Alle Datentypen Bei einer **readonly**-deklarierten Referenzvariablen ist zu beachten, dass der Variableninhalt (die Objektadresse) nach der Initialisierung schreibgeschützt ist, während das referenzierte Objekt im Rahmen bestehender Zugriffsrechte durchaus verändert werden kann.
Möglicher Bezug	Lokale Variable oder statisches Feld	Instanzvariable oder statisches Feld

Im Zusammenhang mit dem OOP-Prinzip der Datenkapselung sind *wertfixierte und öffentliche* Felder eine sinnvolle Möglichkeit, um fremden Klassen ohne Risiko einen syntaktisch einfachen *Lese*zugriff zu gestatten. Im Vergleich zu einer ausschließlich lesend zu verwendenden Eigenschaft besteht der Vorteil des schnelleren Zugriffs, weil kein Methodenaufruf im Spiel ist. In der FCL-Klasse **Math** (Namensraum **System**) ist z.B. das **double**-Feld **PI** als **public** (allgemein verfügbar) und **const** (und damit implizit auch als **static**) deklariert:

```
public const double PI = 3.14159265358979323846;
```

4.3 Instanzmethoden

In einer Bauplan-Klassendefinition werden Objekte entworfen, die eine Anzahl von Verhaltenskompetenzen (Methoden) besitzen, die per Methodenaufruf genutzt werden können. Objekte sind also Dienstleister, die eine Reihe von Nachrichten interpretieren und mit passendem Verhalten beantworten können.

Ihre Instanzvariablen sind bei konsequenter Datenkapselung für Objekte (bzw. Methoden) fremder Klassen unsichtbar (*information hiding*). Um anderen Klassen trotzdem (kontrollierte) Zugriffe auf ein Feld zu ermöglichen, definiert man in C# in der Regel eine zugehörige *Eigenschaft*, die der Compiler letztlich in Zugriffsmethoden übersetzt (siehe Abschnitt 4.5).

Beim Aufruf einer Methode ist in der Regel über so genannte **Parameter** die gewünschte Verhaltensweise festzulegen, und bei vielen Methoden wird dem Aufrufer ein **Rückgabewert** geliefert, z.B. mit der angeforderten Information.

Ziel einer typischen Klassendefinition sind kompetente, einfach und sicher einsetzbare Objekte, die oft auch noch *reale* Objekte aus dem Aufgabenbereich der Software repräsentieren. Wenn ein anderer Programmierer z.B. ein Objekt aus unserer Klasse `Bruch` verwendet, kann er es mit einen Aufruf der Methode `Addiere()` veranlassen, einen per Parameter benannten zweiten Bruch zum eigenen Wert zu addieren, wobei das Ergebnis auch noch gleich gekürzt wird:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

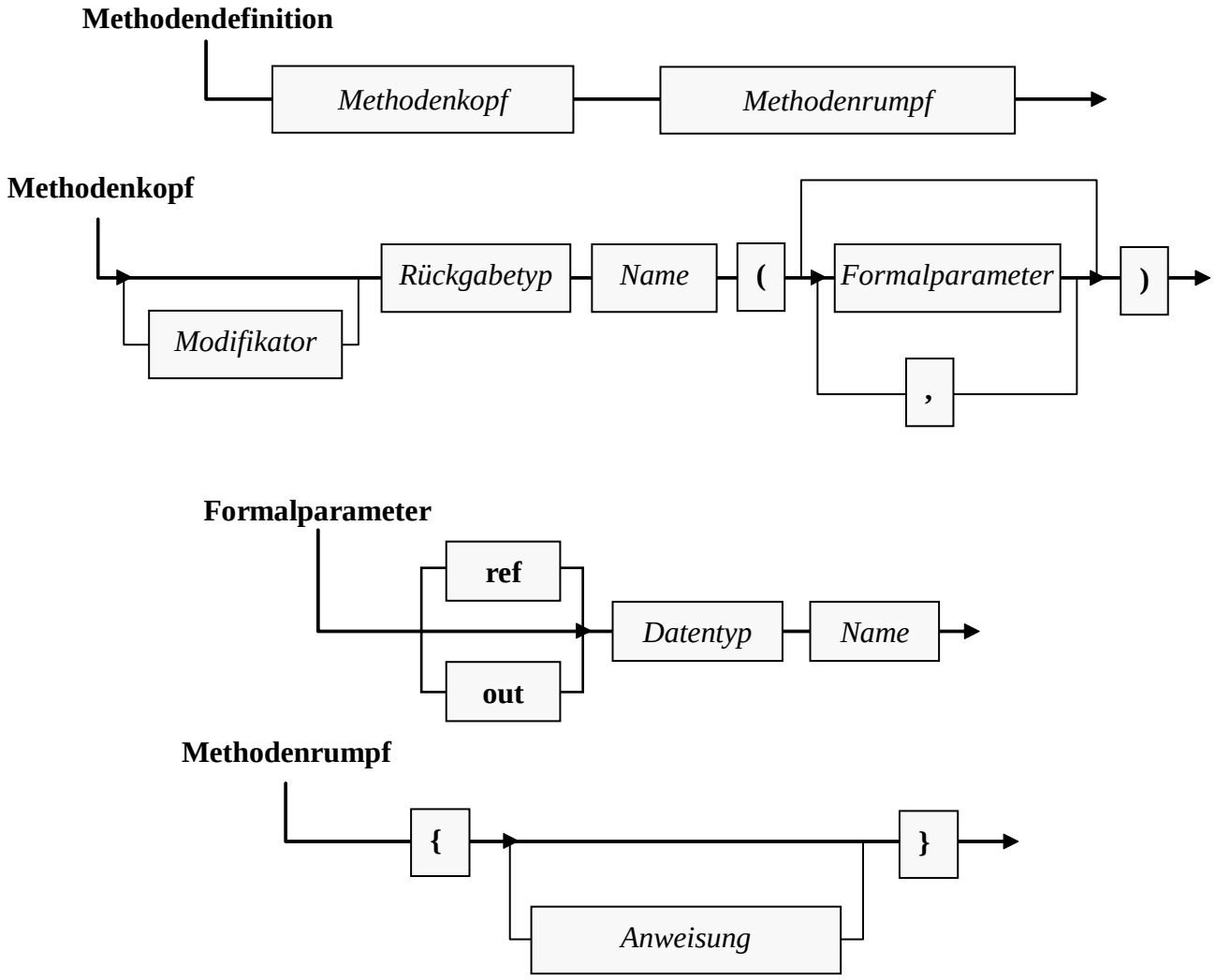
Weil diese Methode auch für fremde Klassen verfügbar sein soll, wird per Modifikator die Schutzstufe **public** gewählt.

Da es vom Verlauf der Auftragserledigung nichts zu berichten gibt, liefert `Addiere()` keinen Rückgabewert. Folglich ist im Kopf der Methodendefinition der Rückgabetyp **void** angegeben.

Während jedes Objekt einer Klasse seine eigenen Instanzvariablen auf dem Heap besitzt, ist der CIL-Code der Instanzmethoden jeweils nur *einmal* im Speicher vorhanden und wird von allen Objekten verwendet.

4.3.1 Methodendefinition

Die folgende Serie von Syntaxdiagrammen zur Methodendefinition unterscheidet sich von der Variante in Abschnitt 3.1.2.2 durch eine genauere Erklärung der (in Abschnitt 4.3.1.3 zu behandelnden) Formalparameterliste:



Anschließend werden die (mehr oder weniger) neuen Bestandteile dieser Syntaxdiagramme erläutert. Dabei werden Methodendefinition und -aufruf keinesfalls so sequentiell und getrennt dargestellt, wie es die Abschnittsüberschriften vermuten lassen. Schließlich ist die Bedeutung mancher Details der Methodendefinition am besten am Effekt beim Aufruf zu erkennen.

Während sich bei *Feldern* die Groß-/Kleinschreibung des Anfangsbuchstabens nach einer generell akzeptierten Konvention an der Schutzstufe orientiert (Camel Casing für Felder mit den Schutzstufen **private**, **protected** oder **internal**, Pascal Casing für öffentliche Felder, vgl. Abschnitt 4.2.2), sind die Empfehlungen für Methodennamen weniger einheitlich. Auf der Webseite

http://msdn.microsoft.com/de-de/library/4df752aw%28en-us,VS.71%29.aspx

empfiehlt Microsoft offenbar, Methodennamen unabhängig von der Schutzstufe mit einem Großbuchstaben beginnen zu lassen (Pascal Casing):

The following rules outline the naming guidelines for methods:

- Use verbs or verb phrases to name methods.
- Use Pascal case.

Die Quellcode-Generatoren (Assistenten) im Visual Studio produzieren bei *privaten* Methoden jedoch Namen mit kleinen Anfangsbuchstaben, z.B.:

```
private void button_Click(object sender, RoutedEventArgs e) { ... }
```

4.3.1.1 Modifikatoren

Bei einer Methodendefinition kann per Modifikator der voreingestellte **Zugriffsschutz** verändert werden. In C# gilt für Methoden gilt wie für Instanzvariablen:

- Voreingestellt ist die Schutzstufe **private**, so dass eine Methode nur in anderen Methoden (oder Eigenschaften) derselben Klasse aufgerufen werden darf.
- Soll eine Methode *allen* Klassen zur Verfügung stehen, ist in ihrer Definition der Modifikator **public** anzugeben. Später werden noch weitere Optionen zur Zugriffssteuerung vorgestellt.

Während man bei Instanzvariablen die Voreinstellung **private** meist belässt, ist sie bei allen Methoden zu ändern, die zur Schnittstelle einer Klasse gehören sollen. In unserer Beispielklasse `Bruch` haben *alle* Methoden den Zugriffsmodifikator **public** erhalten.

Später (z.B. im Zusammenhang mit der Vererbung) werden uns noch Methoden-Modifikatoren begegnen, die anderen Zwecken als der Zugriffsregulation dienen (z.B. **sealed**, **abstract**).

4.3.1.2 Rückgabewert und return-Anweisung

Für den Informationstransfer von einer Methode an ihren Aufrufer kann neben Referenz- und Ausgabeparametern (siehe Abschnitt 4.3.1.3) auch ein Rückgabewert genutzt werden. Hier ist man auf einen *einzigen* Wert (von beliebigem Typ) beschränkt, doch lässt sich die Übergabe sehr elegant in den Programmablauf integrieren. Wir haben schon in Abschnitt 3.5.2 erfahren, dass ein Methodenaufruf einen Ausdruck darstellt und als Argument von komplexeren Ausdrücken oder von Methodenaufrufen verwendet werden darf, sofern die Methode einen Wert von passendem Typ abliefert.

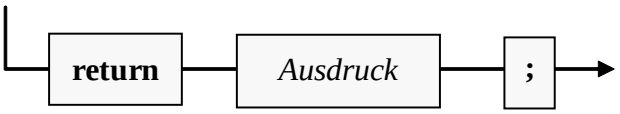
Bei der Definition einer Methode muss festgelegt werden, von welchem Datentyp ihr Rückgabewert ist. Erfolgt *keine* Rückgabe, ist der Ersatztyp **void** anzugeben.

Als Beispiel betrachten wir die aktuelle Variante der `Bruch`-Methode `Frage()` (mit einem Vorgriff auf die Ausnahmebehandlung per **try-catch** - Anweisung), die den Aufrufer durch einen Rückgabewert vom Datentyp **bool** darüber informiert, ob der Benutzer zwei ganze Zahlen im **int**-Wertebereich als Eingaben geliefert hat (**true**) oder nicht (**false**):

```
public bool Frage() {
   try {
      Console.Write("Zaehler: ");
      int z  = Convert.ToInt32(Console.ReadLine());
      Console.Write("Nenner : ");
      int n = Convert.ToInt32(Console.ReadLine());
      Zaehler = z;
      Nenner = n;
      return true;
   } catch {
      return false;
   }
}
```

Ist der Rückgabetyp einer Methode von **void** verschieden, dann *muss* im Rumpf dafür gesorgt werden, dass jeder mögliche Ausführungspfad mit einer **return**-Anweisung endet, die einen Wert von passendem Typ liefert.

return-Anweisung für Methoden *mit* Rückgabewert



In der `Bruch`-Methode `Frage()` wird am Ende eines störungsfrei durchlaufenen **try**-Blocks der boolesche Wert **true** zurückgemeldet. Tritt im **try**-Block eine Ausnahme auf (z.B. beim Versuch, eine irreguläre Benutzereingabe zu konvertieren), dann wird der **catch**-Block ausgeführt, und die dortige **return**-Anweisung sorgt für eine Terminierung mit dem Rückgabewert **false**.

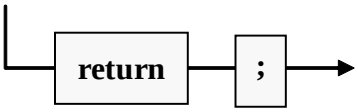
Beim bedenklichen Wunsch des Anwenders, den Nenner auf 0 zu setzen, zeigt die Methode `Frage()` übrigens keine besondere Reaktion. Ein kritische Bewertung bleibt der Eigenschaft `Nenner` überlassen (siehe unten).

Wenn die **Main()** - Methode eines Programms ihrem Aufrufer (also der CLR bzw. dem Betriebssystem) per **return**-Anweisung eine ganze Zahl als Information über den (Miss)erfolg ihrer Tätigkeit liefern möchte (z.B. 0: alles gut gegangen, 1: Beendigung mit Fehler), dann ist in der **Main()** - Methodendefinition der Rückgabetyp **int** anzugeben. Nach Beendigung des Programms findet sich der **return**-Code in der Windows-Umgebungsvariablen **errorlevel** und kann z.B. im Rahmen eines Kommandostapels mit dem **echo**-Befehl angezeigt werden:

```
>echo %errorlevel%
0
```

Bei Methoden *ohne* Rückgabewert ist die **return**-Anweisung nicht unbedingt erforderlich, kann jedoch (in der Variante ohne *Ausdruck*) dazu verwendet werden, um die Methode vorzeitig zu beenden (z.B. im Rahmen einer bedingten Anweisung):

return-Anweisung für Methoden *ohne* Rückgabewert



Beispiel:

```
if (euro <= 0)
    return;
```

4.3.1.3 Formalparameter

Parameter wurden Ihnen bisher vereinfachend als Informationen über die gewünschte Arbeitsweise einer Methode vorgestellt, die beim Aufruf übergeben werden. Tatsächlich kennt C# verschiedene Parameterarten, um den Informationsaustausch zwischen einem Aufrufer und einer angeforderten Methode in *beide* Richtungen optimal zu unterstützen.

Im Kopf der Methoden**definition** werden über so genannte **Formalparameter** Daten von bestimmtem Typ spezifiziert, die entweder den Ablauf der Methode beeinflussen, und/oder durch die Methode verändert werden. Beim späteren **Aufruf** der Methode sind korrespondierende **Aktualparameter** anzugeben (siehe Abschnitt 4.3.2), wobei je nach Parameterart Variablen oder Ausdrücke in Frage kommen.

In den Anweisungen des Methodenrumpfs werden die Formalparameter wie lokale Variablen verwendet, die teilweise (je nach Parameterart) mit den beim Aufruf übergebenen Aktualparameterwerten initialisiert worden sind.

Für jeden Formalparameter sind folgende Angaben zu machen:

- **Parameterart**
  Gleich werden Sie Wert-, Referenz- und Ausgabeparameter kennenlernen.
- **Datentyp**
  Es sind beliebige Typen erlaubt. Man muss den Datentyp eines Formalparameters auch dann explizit angeben, wenn er mit dem Typ des Vorgängers (linken Nachbarn) übereinstimmt.
- **Name**
  Nach den Empfehlungen aus Abschnitt 3.1.5 ist bei Parameternamen das Camel Casing (mit kleinem Anfangsbuchstaben) zu verwenden. Um Namenskonflikte zu vermeiden, hängen manche Programmierer an Parameternamen ein Suffix an, z.B. *par* oder einen Unterstrich. Weil Formalparameter im Methodenrumpf wie lokale Variablen funktionieren, …
  - können Namenskonflikte mit anderen lokalen Variablen derselben Methode auftreten,
  - werden namensgleiche Instanz- bzw. Klassenvariablen überlagert.
    Diese bleiben jedoch über ein geeignetes Präfix weiter ansprechbar:
    - **this** bei Instanzvariablen
    - Klassenname bei Klassenvariablen
- **Position**
  Die Position eines Formalparameters ist natürlich nicht gesondert anzugeben, sondern liegt durch die Methodendefinition fest. Sie wird hier als relevante Eigenschaft erwähnt, weil die beim späteren Aufruf der Methode übergebenen Aktualparameter gemäß ihrer Reihenfolge den Formalparametern zugeordnet werden.

4.3.1.3.1 Wert- bzw. Eingabeparameter

Über einen Wert- bzw. Eingabeparameter werden Informationen in eine Methode kopiert, um diese mit Daten zu versorgen oder ihre Arbeitsweise zu steuern. Als Beispiel betrachten wir folgende Variante der `Bruch`-Methode `Addiere()` Das beauftragte Objekt soll den über **int**-wertige Parameter (`zpar`, `npar`) übergebenen Bruch zu seinem eigenen Wert addieren und optional (**bool**-Parameter `autokurz`) das Resultat gleich kürzen:

```
  public bool Addiere(int zpar, int npar, bool autokurz) {
    if (npar != 0) {
      zaehler = zaehler * npar + zpar * nenner;
      nenner = nenner * npar;
      if (autokurz)
        Kuerze();
      return true;
    } else
      return false;
  }
```

Bei der Definition eines formalen Wertparameters ist vor dem Datentyp *kein* Schlüsselwort anzugeben. Innerhalb der Methode verhält sich ein Wertparameter wie eine lokale Variable, die durch den im Aufruf übergebenen Wert initialisiert wurde.

Methodeninterne Änderungen dieser lokalen Variablen bleiben *ohne* Effekt auf eine als Aktualparameter fungierende Variable der rufenden Programmeinheit. Im folgenden Beispiel übersteht die lokale Variable `i` der Methode **Main()** den Einsatz als Wertaktualparameter beim Aufruf der Methode `WertParDemo()` ohne Folgen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  void WertParDemo(int ipar) {` `  Console.WriteLine(++ipar);` `  }` `  static void Main() {` `  int i = 4711;` `  Prog p = new Prog();` `  p.WertParDemo(i);` `  Console.WriteLine(i);` `  }` `}`	`4712` `4711`

Die Demoklasse `Prog` ist startfähig, besitzt also eine Methode **Main()**. Dort wird ein Objekt der Klasse `Prog` erzeugt und beauftragt, die Instanzmethode `WertParDemo()` auszuführen. Mit dieser (auch in den folgenden Abschnitten anzutreffenden) Konstruktion wird es vermieden, im aktuellen Abschnitt 4.3.1 über Details bei der Definition von *Instanzmethoden* zur Demonstration *statische* Methoden verwenden zu müssen. Bei den Parametern und beim Rückgabetyp gibt es allerdings keine Unterschiede zwischen den Instanzmethoden und den Klassenmethoden (siehe Abschnitt 4.6.3).

Als Wertaktualparameter sind nicht nur (initialisierte!) Variablen erlaubt, sondern beliebige Ausdrücke mit einem Typ, der nötigenfalls erweiternd in den Typ des zugehörigen Formalparameters gewandelt werden kann.

4.3.1.3.2 Referenzparameter

Ein Referenzparameter ermöglicht es der aufgerufenen Methode, eine Variable der aufrufenden Programmeinheit zu *verändern*. Die Methode erhält beim Aufruf keine *Kopie* der betroffenen Variablen, sondern die Speicheradresse des Originals. Alle methodenintern über den Formalparameternamen vorgenommenen Modifikationen wirken sich direkt auf das Original aus.

Als Referenzaktualparameter sind nur *Variablen* erlaubt, die vom *selben* Typ wie der Formalparameter und außerdem initialisiert sein müssen. Es findet also keine implizite Typanpassung statt. Auf den garantiert definierten Wert eines Referenzaktualparameters kann die gerufene Methode (vor einer möglichen Veränderung) auch lesend zugreifen. Im Unterschied zu den Wert- bzw. Eingabe-

parametern und den gleich vorzustellenden Ausgabeparametern ermöglichen Referenzparameter einen Informationsfluss in *beide* Richtungen.

In der Methodendefinition *und* beim Methodenaufruf sind Referenzparameter durch das Schlüsselwort **ref** zu kennzeichnen.

Im folgenden Programm (nach dem aus Abschnitt 4.3.1.3.1 bekannten Strickmuster) tauscht eine Instanzmethode die Werte zwischen den als Referenzaktualparameter übergebenen Variablen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  void Tausche(ref int a, ref int b) {` `  int temp = a;` `  a = b;` `  b = temp;` `  }` `  static void Main() {` `  Prog p = new Prog();` `  int x = 1, y = 2;` `  Console.WriteLine("Vorher:  x = {0}, y = {1}", x, y);` `  p.Tausche(ref x, ref y);` `  Console.WriteLine("Nachher: x = {0}, y = {1}", x, y);` `  }` `}`	`Vorher:  x = 1, y = 2` `Nachher: x = 2, y = 1`

Es folgt ein Satz für Begriffsakrobaten: Wird eine *Referenzvariable* (hat als Inhalt eine Objektadresse) als Referenzaktualparameter übergeben, kann man methodenintern nicht nur auf das Objekt zugreifen (z.B. auf seine Instanzvariablen bei entsprechenden Zugriffsrechten), sondern auch den Inhalt der übergebenen Referenzvariablen ändern, so dass sie z.B. anschließend auf ein anderes Objekt zeigt. Es ist nur selten sinnvoll, bei einer Methodendefinition Referenzparameter mit Referenztyp (vom Typ einer Klasse) zu verwenden. Erlaubt ist diese für Anfänger recht verwirrende Doppelreferenz jedoch. Ein möglicher Einsatzzweck ist eine methodenintern zu verändernde und zum Aufrufer zurück zu transportierende Zeichenfolge. Wie sich in Abschnitt 5.4.1.1 zeigen wird, lässt sich ein **String**-Objekt nicht ändern, sondern nur durch ein neues **String**-Objekt ersetzen. Ein **ref**-Parameter vom Typ **String** bietet die Möglichkeit, die Adresse des alten Strings in eine Methode hinein und die Adresse des neuen Strings zum Aufrufer zurück zu transportieren:

Quellcode	Ausgabe
`using System;` `class Prog {` `  void RefRefParDemo(ref String spar) {` `  Console.WriteLine(spar);` `  spar = "NEU";` `  }` `  static void Main() {` `  string s = "ALT";` `  Prog p = new Prog();` `  p.RefRefParDemo(ref s);` `  Console.WriteLine(s);` `  }` `}`	`ALT` `NEU`

4.3.1.3.3 Ausgabeparameter

Auch über Ausgabeparameter kann man einer Methode die Veränderung von Variablen der rufenden Programmeinheit ermöglichen.

Als Ausgabeaktualparameter sind nur *Variablen* erlaubt, die vom *selben* Typ wie der Formalparameter sein müssen. Es findet also keine implizite Typanpassung statt. Der Compiler interessiert sich *nicht* dafür, ob die als Aktualparameter fungierenden Variablen beim Methodenaufruf initialisiert sind. Stattdessen stellt er sicher, dass jedem Ausgabeparameter vor dem Verlassen der Methode ein Wert zugewiesen wird.

In der Methodendefinition *und* beim Methodenaufruf sind Ausgabeparameter durch das Schlüsselwort **out** zu kennzeichnen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  void Lies(out int z, out int n) {` `  Console.Write("x = ");` `  z = Convert.ToInt32(Console.ReadLine());` `  Console.Write("\ny = ");` `  n = Convert.ToInt32(Console.ReadLine());` `  }` `  static void Main() {` `  Prog p = new Prog();` `  int x, y;` `  p.Lies(out x, out y);` `  Console.WriteLine("\nx % y = " + (x % y));` `  }` `}`	`x = 29` `y = 5` `x % y = 4`

4.3.1.3.4 Serienparameter

Vielleicht haben Sie sich schon darüber gewundert, dass man beim Aufruf der Methode **Console.WriteLine()** hinter einer geeigneten Formatierungszeichenfolge beliebig viele Ausdrücke durch jeweils ein Komma getrennt als Aktualparameter übergeben darf, z.B.:

```
Console.WriteLine("x = {0} ", x);
Console.WriteLine("x = {0}, y = {1} ", x, y);
```

Diese Variabilität wird durch einen Array-Parameter ermöglicht, der an *letzter* Stelle deklariert und durch das Schlüsselwort **params** gekennzeichnet werden muss. Im Syntaxdiagramm zum Methodenkopf wurde diese Option der Einfachheit halber weggelassen. Zwar haben wir uns bisher kaum mit Array-Datentypen beschäftigt, doch kann das folgende Beispiel hoffentlich trotzdem die Verwendung von Serienparametern hinreichend klären:

Quellcode	Ausgabe
`using System;` `class Prog {` `  void PrintSum(params double[] args) {` `  double summe = 0.0;` `  foreach (double arg in args)` `  summe += arg;` `  Console.WriteLine("Die Summe ist = " + summe);` `  }` `  static void Main() {` `  Prog p = new Prog();` `  p.PrintSum(1.2, 1.0);` `  p.PrintSum(1.2, 1.0, 3.6);` `  }` `}`	`Die Summe ist = 2,2` `Die Summe ist = 5,8`

4.3.1.4 Methodenrumpf

Über die Blockanweisung, die den Rumpf einer Methode bildet, haben Sie bereits erfahren:

- Hier werden die Formalparameter wie lokale Variablen verwendet.
- Wert- und Referenzparameter werden von der aufrufenden Programmeinheit initialisiert, so dass diese den Ablauf der Methode beeinflussen kann.
- Über Referenz- und Ausgabeparameter können Variablen der aufrufenden Programmeinheit verändert werden.
- Die **return**-Anweisung dient zur Rückgabe von Werten an den Aufrufer und/oder zum Beenden der Methodenausführung.

Ansonsten können beliebige Anweisungen unter Verwendung von elementaren und objektorientierten Sprachelementen eingesetzt werden, um den Zweck einer Methode zu realisieren. Definitionen (z.B. von Klassen oder Methoden) sind jedoch *nicht* erlaubt.[1]

Weil in einer Methode häufig andere Methoden aufgerufen werden, kommt es in der Regel zu mehrstufig verschachtelten Methodenaufrufen, wobei die Höhe des Stacks (Stapelspeichers) zur Verwaltung der Methodenaufrufe entsprechend wächst (siehe Abschnitt 4.3.4).

4.3.2 Methodenaufruf und Aktualparameter

Beim Aufruf einer Instanzmethode, z.B.:

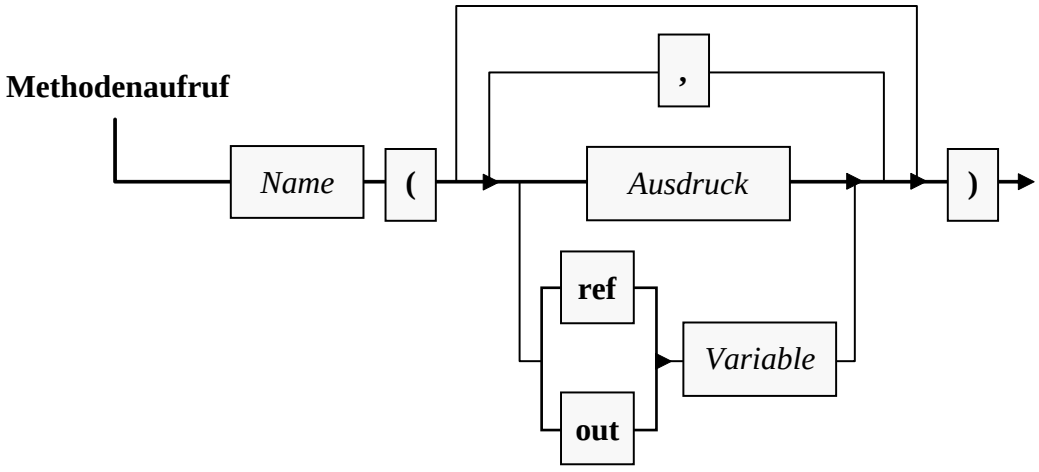
```
b1.Zeige();
```

wird nach objektorientierter Denkweise eine *Botschaft* an ein Objekt geschickt:

„b1, zeige dich!“.

Als Syntaxregel ist festzuhalten, dass zwischen dem Objektnamen (genauer: dem Namen der Referenzvariablen, die auf das Objekt zeigt) und dem Methodennamen der **Punktoperator** zu stehen hat.

Beim Aufruf einer Methode folgt ihrem Namen die in runde Klammern eingeschlossene Liste mit den **Aktualparametern**, wobei es sich um eine synchron zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen passenden Typs handeln muss.



Es ist grundsätzlich eine Parameterliste anzugeben, ggf. eine leere.

---

[1] Im Zusammenhang mit Delegaten und anonymen Methoden werden Sie später doch eine Möglichkeit zum Verschachteln von Methodendefinitionen kennen lernen (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).

Als Beispiel betrachten wir einen Aufruf der in Abschnitt 4.3.1.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`:

```
b1.Addiere(1, 3, true);
```

Einem Referenz- bzw. Ausgabeparameter ist auch beim Aufruf das Schlüsselwort **ref** bzw. **out** voranzustellen, z.B.:

```
p.Tausche(ref x, ref y);
```

Liefert eine Methode einen Wert zurück, stellt ihr Aufruf einen verwertbaren **Ausdruck** dar und kann als Argument in komplexeren Ausdrücken auftreten, z.B.:

```
do
  Console.WriteLine("Welchen Bruch möchten Sie kürzen?");
while (!b1.Frage());
```

Durch ein angehängtes Semikolon wird jeder Methodenaufruf zur vollständigen **Anweisung**, wobei ein Rückgabewert ggf. ignoriert wird, z.B.:

```
b1.Frage();
```

Soll in einer Methodenimplementierung vom aktuell handelnden Objekt eine andere Instanzmethode ausgeführt werden, so muss beim Aufruf *keine* Objektbezeichnung angegeben werden. In beiden Varianten der `Bruch`-Methode `Addiere()` soll das beauftragte Objekt den via Parameterliste übergebenen Bruch zu seinem eigenen Wert addieren und das Resultat (bei der Variante aus Abschnitt 4.3.1.3.1 paramtergesteuert) gleich kürzen. Zum Kürzen kommt natürlich die entsprechende `Bruch`-Methode zum Einsatz. Weil sie vom gerade agierenden Objekt auszuführen ist, wird keine Objektbezeichnung benötigt, z.B.:

```
public void Addiere(Bruch b) {
  zaehler = zaehler*b.nenner + b.zaehler*nenner;
  nenner = nenner*b.nenner;
  Kuerze();
}
```

Wer auch solche Methodenaufrufe nach dem Schema

Empfänger.*Botschaft*

realisieren möchte, kann mit dem Schlüsselwort **this** das aktuell handelnde Objekt ansprechen, z.B.:

```
this.Kuerze();
```

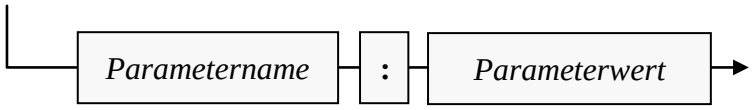
4.3.3 Benannte und optionale Parameter

In diesem Abschnitt werden syntaktische Varianten zur Definition und Verwendung von Parametern nachgeliefert, die oben der Übersichtlichkeit halber weggelassen wurden.

4.3.3.1 Benannte Aktualparameter

Statt beim Aufruf einer Methode mit zwei oder mehr Parametern eine synchron zur Formalparameterliste geordnete Serie von Ausdrücken bzw. Variablen passenden Typs zu liefern, kann man benannte Aktualparameter verwenden (engl. Bezeichnung: *named arguments*):

Benannter Aktualparameter



Der folgende Aufruf der in Abschnitt 4.3.1.1 vorgestellten Variante der `Bruch`-Methode `Addiere()`

```
b1.Addiere(1, 3, true);
```

lässt sich äquivalent auch so formulieren

```
b1.Addiere(npar: 3, zpar: 1, autokurz: true);
```

Als potentielle Vorteile der benannten Aktualparameter sind zu nennen:

- Man muss sich nicht an die Definitionsreihenfolge der Parameter halten.
- Die Lesbarkeit des Quellcodes wird verbessert.

Es ist erlaubt, auf eine Serie von traditionellen Positionsparametern benannte Parameter folgen zu lassen, z.B.:

```
b1.Addiere(1, 3, autokurz: true);
```

Umgekehrt dürfen auf einen benannten Parameter keine Positionsparameter folgen.

Die benannten Argumente wurden vermutlich eingeführt, um die COM-Interoperabilität zu erleichtern, lassen sich aber auch unabhängig von diesem Zweck verwenden.[1]

4.3.3.2 Optionale Parameter

In einer Methodendefinition kann seit C# 4.0 für einen Formalparameter ein Voreinstellungswert angegeben werden, so dass ein *optionaler* Parameter entsteht, der beim Aufruf im Unterschied zu den bisher behandelten *obligatorischen* Parametern weggelassen werden darf, wobei der Voreinstellungswert zum Einsatz kommt. Bei der in Abschnitt 4.3.3.1 als Beispiel betrachteten `Addiere()`-Überladung aus der Klasse `Bruch` könnte z.B. der dritte Parameter einen Voreinstellungswert erhalten und somit zum optionalen Parameter werden:

```
public bool Addiere(int zpar, int npar, bool autokurz = true) {
    . . .
}
```

Ein Aufruf mit gewünschter Ergebniskürzung könnte dann wie im folgenden Beispiel formuliert werden:

```
b1.Addiere(1, 3);
```

Bei optionalen Parametern sind folgende Regeln zu beachten:

- Ein Voreinstellungswert muss schon zur Übersetzungszeit feststehen.
- Auf einen optionalen Formalparameter darf kein obligatorischer mehr folgen.

Optionale Parameter sind auch bei Konstruktoren und Indexern erlaubt (siehe Abschnitt 5.6).

4.3.4 Debug-Einsichten zu (verschachtelten) Methodenaufrufen

Verschachtelte Methodenaufrufe stellen keine Besonderheit, sondern den selbstverständlichen Normalfall dar. Gerade deswegen ist es angemessen, das Geschehen etwas genauer zu betrachten. Anhand der folgenden Bruchrechnungsstartklasse

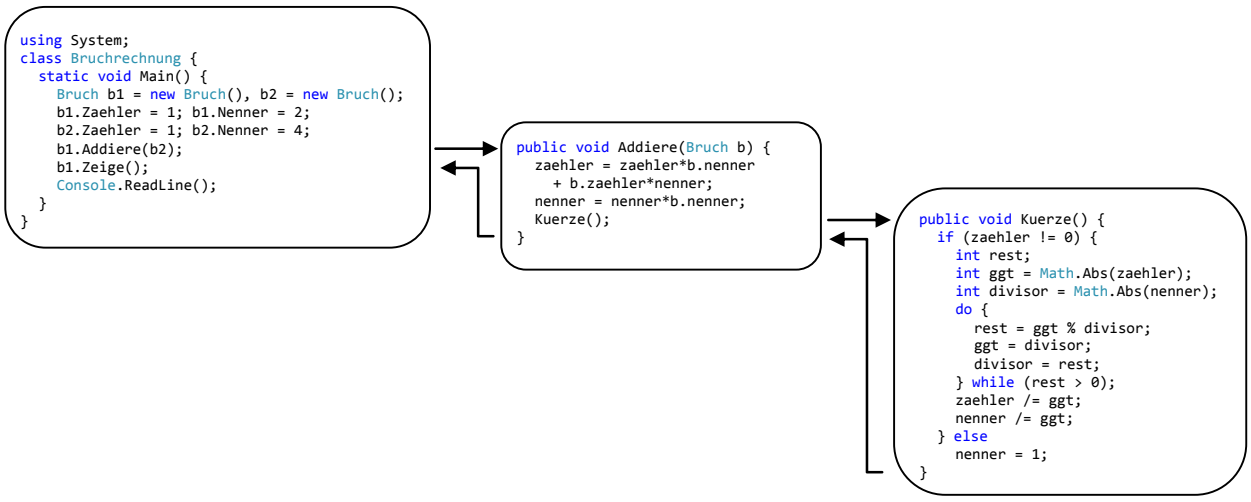
---

[1] Gelegentlich ist es erforderlich, Komponenten mit der COM-Architektur (*Component Object Model*) in einem .NET-Programm anzusprechen. Über solche Komponenten macht z.B. Microsoft Office seine Funktionalität für andere Programme nutzbar.

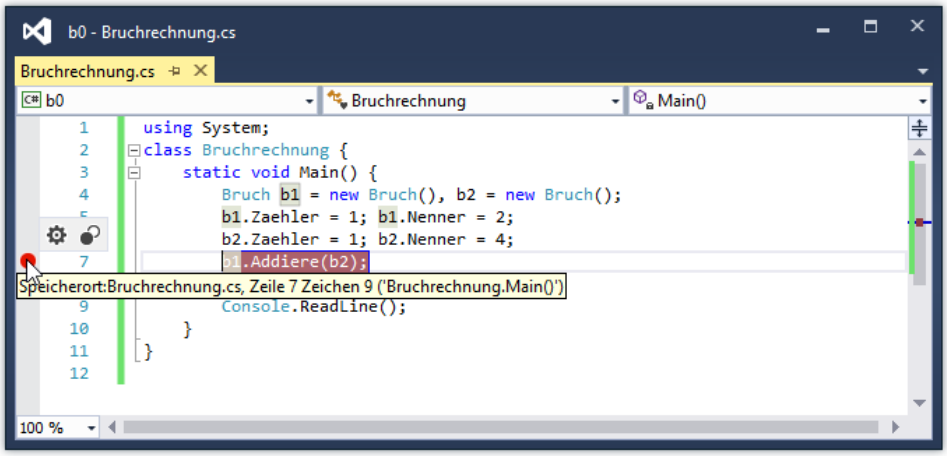
```
using System;
class Bruchrechnung {
   static void Main() {
      Bruch b1 = new Bruch(), b2 = new Bruch();
      b1.Zaehler = 1; b1.Nenner = 2;
      b2.Zaehler = 1; b2.Nenner = 4;
      b1.Addiere(b2);
      b1.Zeige();
      Console.ReadLine();
   }
}
```

soll mit Hilfe unserer Entwicklungsumgebung untersucht werden, was bei folgender Aufrufverschachtelung geschieht:

- Die statische Methode **Main()** der Klasse `Bruchrechnung` ruft die `Bruch`-Instanzmethode `Addiere()`.
- Die `Bruch`-Instanzmethode `Addiere()` ruft die `Bruch`-Instanzmethode `Kuerze()`.



Wir verwenden dabei die zur Fehlersuche konzipierten Debug-Techniken unserer Entwicklungsumgebung. Das Programm soll an mehreren Stellen durch einen so genannten **Halte-** bzw. **Unterbrechungspunkt** (engl. *breakpoint*) gestoppt werden, so dass wir jeweils die Lage im Hauptspeicher inspizieren können. Um einen Haltepunkt zu setzen oder wieder zu entfernen, setzt man einen Mausklick in die grau hinterlegte linke Randspalte neben der betroffenen Anweisung, z.B.



Befindet sich die Einfügemarke in der betroffenen Zeile, hat die Taste **F9** denselben Effekt.

Setzen Sie weitere Unterbrechungspunkte …

- in der Methode **Main()** vor den `Zeige()` - Aufruf,
- in der `Bruch`-Methode `Addiere()` vor den `Kuerze()` - Aufruf,
- in der `Bruch`-Methode `Kuerze()` vor die Anweisung

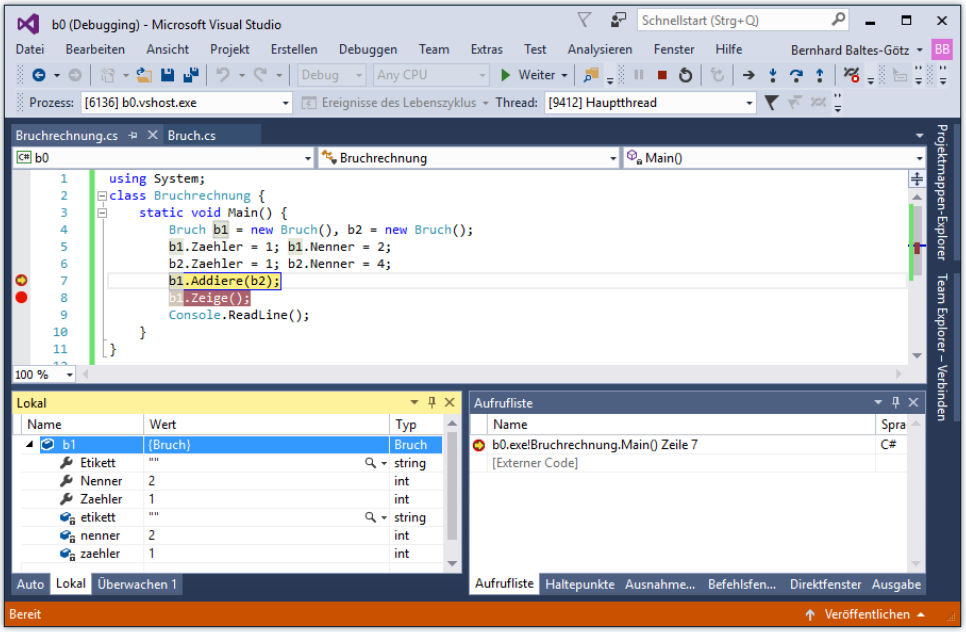
  ```
  ggt = divisor;
  ```

  im Block der **do-while** - Schleife.

Starten Sie das Programm über den Schalter ▶, die Funktionstaste **F5** oder den Menübefehl

Debuggen > Debugging starten

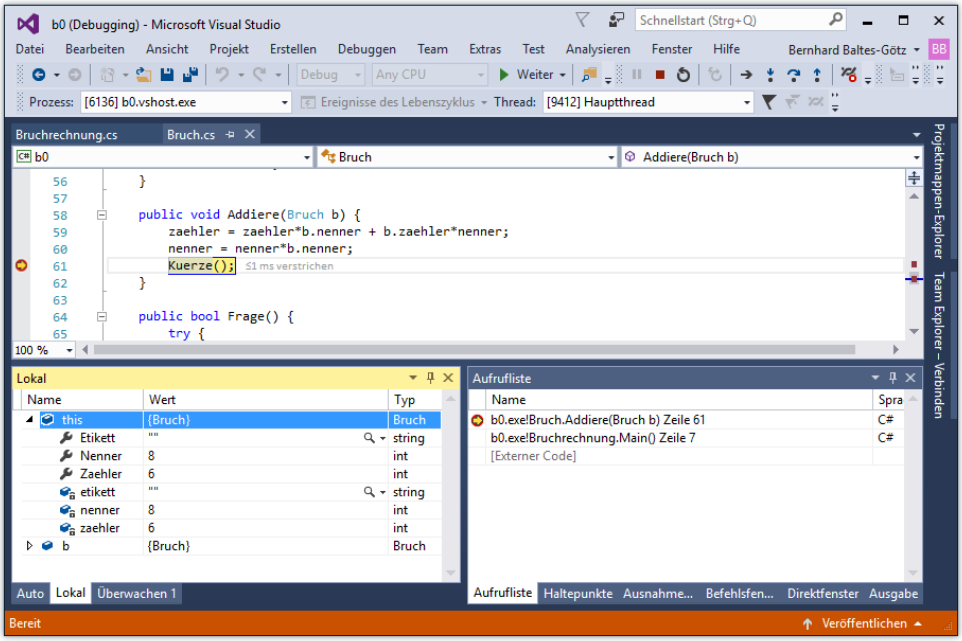
Die Entwicklungsumgebung stoppt das Programm beim ersten Unterbrechungspunkt und zeigt im Quellcode-Editor den erreichten Programmfortschritt an. Unten links zeigt das mit **Lokal** betitelte Fenster die beiden lokalen Referenzvariablen der **Main()** - Methode (`b1`, `b2`) und auf Wunsch (nach einem Mausklick auf einen Plus-Schalter neben einer Referenzvariablen) das Innenleben der referenzierten Objekte. Momentan interessiert besonderes das mit **Aufrufliste** betitelte Fenster unten rechts. Hier ist zu erkennen, dass aktuell nur die Einsprungmethode **Main()** in Bearbeitung befindlich ist (mit Daten im Stack-Bereich des Speichers):



Das Fenster mit den **Diagnosetools** wurde der Übersichtlichkeit halber geschlossen.[1]

Lassen Sie das Programm mit dem Schalter ▶ oder der Funktionstaste **F5** fortsetzen. Beim Erreichen des zweiten Haltepunkts (Anweisung „`Kuerze();`“ in der Methode `Addiere()`) liegen auf dem Stack die Daten und Verwaltungsinformationen (die *Stack Frames*) der Methoden `Addiere()` und **Main()** übereinander:

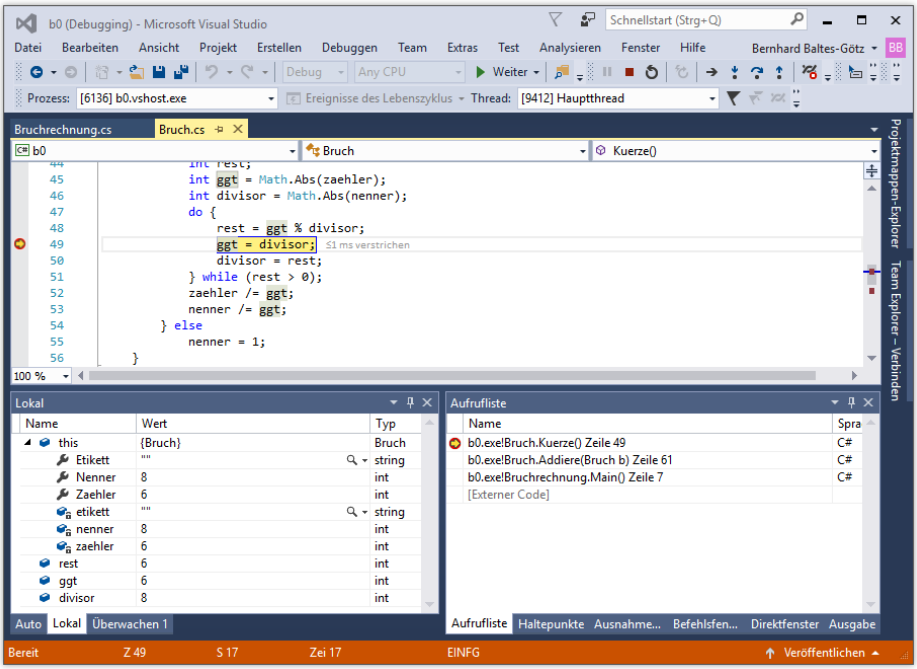
[1] Mit dem folgenden Menübefehl holt man bei Bedarf die **Diagnosetools** zurück:
Debuggen > Fenster > Diagnosetools anzeigen



Das Fenster **Lokal** zeigt als lokale Variablen der Methode `Addiere()`:

- **this** (Referenz auf das handelnde Objekt)
  Erwartungsgemäß ist der Bruch noch nicht gekürzt.
- Parameter `b`

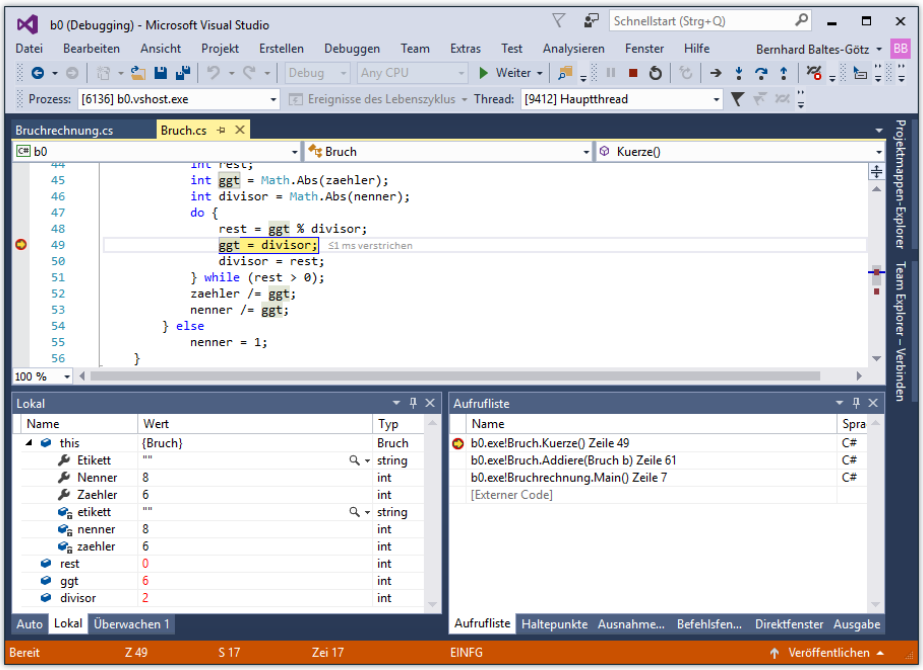
Beim Erreichen des dritten Haltepunkts (Anweisung „`ggt = divisor;`“ in der Methode `kuerze()`) liegen die Stack Frames der Methoden `Kuerze()`, `Addiere()` und **Main()** übereinander:



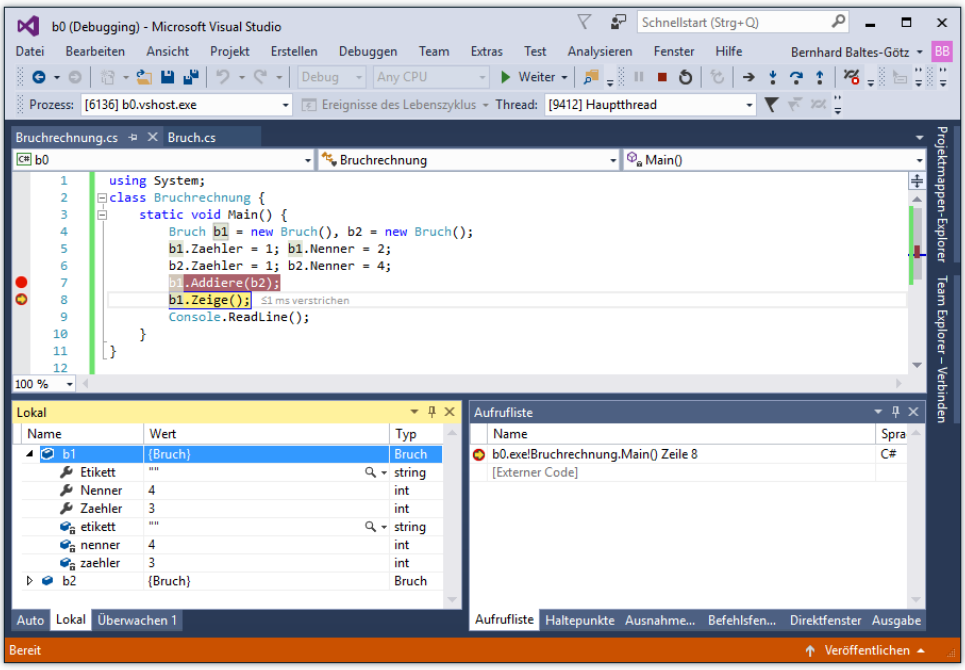
Das Fenster **Lokal** zeigt als lokale Variablen der Methode `Kuerze()`:

- **this** (Referenz auf das handelnde Objekt)
- die lokalen (im Block zur **if**-Anweisung deklarierten) Variablen `rest`, `ggt` und `divisor`.

Weil sich der dritte Unterbrechungspunkt in einer **do** - Schleife befindet, sind mehrere Fortsetzungsbefehle bis zum Verlassen der Methode `Kuerze()` erforderlich, wobei die Werte der lokalen Variablen den Verarbeitungsfortschritt erkennen lassen, z.B.:

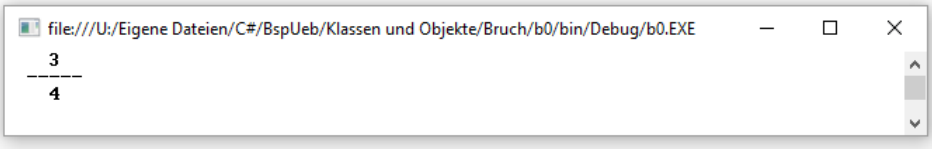


Bei Erreichen des letzten Haltepunkts (Anweisung „`b1.Zeige();`“ in **Main()**) ist nur noch der Stack Frame der Methode **Main()** vorhanden:



Die anderen Stack Frames sind verschwunden, und die dort ehemals vorhandenen lokalen Variablen existieren nicht mehr.

Nach einem weiteren Fortsetzungsklick zeigt sich das `Bruch`-Objekt `b1` im Konsolenfester:



Zum Beseitigen eines Haltepunkts klickt man ihn erneut an. Das simultane Entfernen *aller* Haltepunkte gelingt über den folgenden Menübefehl:

Debuggen > Alle Haltepunkte löschen

Weil der verfügbare Speicher endlich ist, kann es bei einer Aufrufverschachtelung und der damit verbundenen Stapelung von Stack Frames zu einem Laufzeitfehler vom Typ **StackOverflow-Exception** kommen. Das wird aber nur bei einem schlecht entworfenen bzw. fehlerhaften Algorithmus passieren.

4.3.5 Methoden überladen

Die in Abschnitt 4.3.1.1 vorgestellte `Addiere()` - Methode kann problemlos in der `Bruch`-Klassendefinition mit der dort bereits vorhandenen `Addiere()` - Variante koexistieren, weil beide Methoden unterschiedliche Parameterlisten besitzen. Besitzt eine Klasse mehrere Methoden mit demselben Namen, liegt eine so genannte *Überladung* von Methoden vor.

Eine Überladung ist erlaubt, wenn sich die **Signaturen** der beteiligten Methoden unterscheiden. Zwei Methoden besitzen genau dann *dieselbe* Signatur, was *innerhalb einer Klasse* verboten ist, wenn die folgenden Bedingungen erfüllt sind:[1]

- Die Namen der Methoden sind identisch.
- Die Formalparameterlisten sind gleich lang
- Positionsgleiche Parameter stimmen hinsichtlich Datentyp und Parameterart (Wert-, Referenz- bzw. Ausgabeparameter) überein.

Für die Signatur ist der Rückgabetyp einer Methode ebenso irrelevant wie die Namen ihrer Formalparameter und das beim letzten Formalparameter erlaubte **params**-Schlüsselwort (vgl. ECMA 2006, S. 95). Die fehlende Signaturrelevanz des Rückgabetyps resultiert daraus, dass der Rückgabewert einer Methode in Anweisungen oft keine Rolle spielt (ignoriert wird). Folglich muss generell unabhängig vom Rückgabetyp entscheidbar sein, welche Methode aus einer Überladungsfamilie zu verwenden ist.

Ist bei einem Methodenaufruf die angeforderte Überladung nicht eindeutig zu bestimmen, meldet der Compiler einen Fehler, was aber bei einem vernünftigen Entwurf von überladenen Methoden nur sehr selten passiert.

Von einer Methode unterschiedlich parametrisierte Varianten in eine Klassendefinition aufzunehmen, lohnt sich z.B. in folgenden Situationen:

- Für verschiedene Datentypen (z.B. **double** und **int**) werden analog arbeitende Methoden benötigt.
  So besitzt z.B. die Klasse **Math** im Namensraum **System** u.a. folgende Methoden, um den Betrag einer Zahl zu berechnen:

  public static float Abs(**decimal** *value*)
  public static double Abs(**double** *value*)
  public static float Abs(**float** *value*)
  public static int Abs(**int** *value*)
  public static long Abs(**long** *value*)

  Seit der .NET - Version 2.0 bieten allerdings *generische Methoden* (siehe unten) eine elegantere Lösung für die Unterstützung verschiedener Datentypen.[1]

[1] Bei den später zu behandelnden *generischen* Methoden muss die Liste mit den Kriterien für die Identität von Signaturen erweitert werden.

- Für eine Methode sollen unterschiedliche umfangreiche Parameterlisten angeboten werden, sodass zwischen einer bequem aufrufbaren Standardausführung (z.B. mit leerer Parameterliste) und einer individuell gestalteten Ausführungsvariante gewählt werden kann. Über die seit C# 4.0 verfügbaren optionalen Parameter lässt sich die beschriebene Aufgabenstellung allerdings oft auch mit einer einzigen Methodendefinition lösen (siehe Abschnitt 4.3.3).

4.4 Objekte

Im Abschnitt 4.4 geht es darum, wie Objekte erzeugt und im obsoleten Zustand wieder aus dem Speicher entfernt werden.

4.4.1 Referenzvariablen deklarieren

Um irgendein Objekt aus der Klasse `Bruch` ansprechen zu können, benötigen wir eine **Referenzvariable** mit dem Datentyp `Bruch`. In der folgenden Anweisung wird eine solche Referenzvariable definiert und auch gleich initialisiert:

```
Bruch b = new Bruch();
```

Um die Wirkungsweise dieser Anweisung Schritt für Schritt zu untersuchen, beginnen wir mit einer einfacheren Variante *ohne* Initialisierung:

```
Bruch b;
```

Hier wird die **Referenzvariable** `b` mit dem Datentyp `Bruch` deklariert, der man folgende Werte zuweisen kann:

- die Adresse eines `Bruch`-Objekts
  In der Variablen wird kein komplettes `Bruch`-Objekt mit sämtlichen Instanzvariablen abgelegt, sondern ein **Verweis** (eine **Referenz**) auf einen Ort im Heap-Bereich des programmeigenen Speichers, wo sich ein `Bruch`-Objekt befindet.
  Sollte einmal eine Ableitung der Klasse `Bruch` definiert werden, können deren Objekte ebenfalls über `Bruch`-Referenzvariablen verwaltet werden. Vom Vererbungsprinzip der objektorientierten Programmierung haben Sie schon einiges gehört, doch steht die gründliche Behandlung noch aus.
- **null**
  Dieses Referenzliteral steht für einen leeren Verweis. Eine Referenzvariable mit diesem Wert ist nicht undefiniert, sondern zeigt explizit auf nichts.

Wir nehmen nunmehr offiziell und endgültig zur Kenntnis, dass *Klassen als Datentypen* verwendet werden können und haben damit bislang folgende Datentypen zur Verfügung (vgl. Abschnitt 3.3.2):

- Elementare Typen (**bool**, **char**, **byte**, **double**, ...)
  Hier handelt es sich um Werttypen.
- Klassen (Referenztypen)
  Ist eine Variable vom Typ einer Klasse, kann sie (neben **null**) die Adresse eines Objekts aus dieser Klasse oder aus einer daraus abgeleiteten Klasse aufnehmen.

Später kommen mit den *Strukturen* noch Werttypen hinzu, deren Instanzen (wie die Objekte von Klassen) problemadäquat mit beliebig vielen Feldern ausgestattet werden können.

---

[1] So tauscht z.B. die folgende statische Methode die Inhalte von zwei Referenzparametern mit beliebigem (natürlich identischem) Datentyp:

```
static void Tausche<T>(ref T a, ref T b) {
  T temp = a;
  a = b;
  b = temp;
}
```

4.4.2 Objekte erzeugen

Damit z.B. der folgendermaßen deklarierten Referenzvariablen `b` vom Datentyp `Bruch`

```
Bruch b;
```

ein Verweis auf ein `Bruch`-Objekt als Wert zugewiesen werden kann, muss ein solches Objekt erst erzeugt werden, was per **new**-Operator geschieht, z.B. in folgendem Ausdruck:

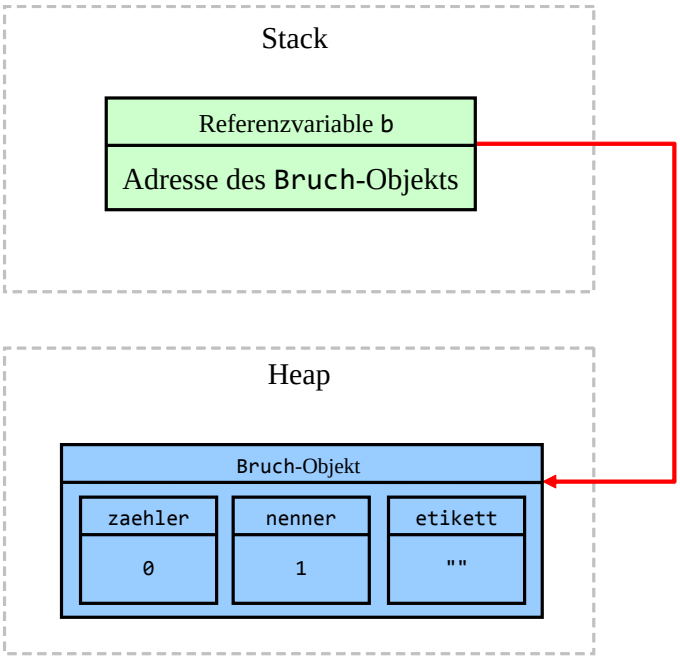
```
new Bruch()
```

Als Operanden erwartet der **new**-Operator einen Klassennamen, dem eine Parameterliste zu folgen hat, weil er hier als Name eines *Konstruktors* (siehe Abschnitt 4.4.3) fungiert. Die involvierte Klasse legt den Typ des Ausdrucks fest. Als Wert resultiert eine Referenz, die einen Zugriff auf das neue Objekt (seine Methoden, Eigenschaften, etc.) erlaubt.

In der **Main()** - Methode der folgenden Startklasse

```
class Bruchrechnung {
  static void Main() {
    Bruch b = new Bruch();
      . . .
  }
}
```

wird die vom **new**-Operator gelieferte Adresse mit dem Zuweisungsoperator in die lokale Referenzvariable `b` geschrieben. Es resultiert die folgende Situation im programmeigenen Hauptspeicher:[1]



Während lokale Variablen im **Stack**-Bereich des Hauptspeichers angelegt werden, entstehen Objekte mit ihren Instanzvariablen auf dem **Heap** .

In einem Programm können durchaus *mehrere* Referenzvariablen auf *dasselbe* Objekt zeigen, z.B.:

---

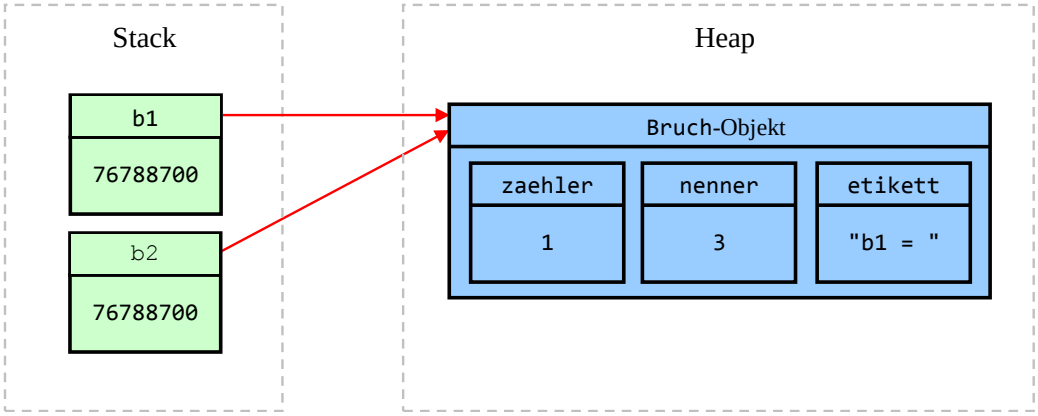
[1] Hier wird aus didaktischen Gründen ein wenig gemogelt. Die Instanzvariable `etikett` ist vom Typ der Klasse **String**, zeigt also auf ein **String**-Objekt, das „neben“ dem `Bruch`-Objekt auf dem Heap liegt. In der `Bruch`-Referenzinstanzvariablen `etikett` befindet sich die Adresse des **String**-Objekts.

Quellcode	Ausgabe
`using System;` `class Bruchrechnung {` `  static void Main() {` `  Bruch b1 = new Bruch();` `  b1.Zaehler = 1;` `  b1.Nenner = 3;` `  b1.Etikett = "b1 = ";` `  Bruch b2 = b1;` `  b2.Etikett = "b2 = ";` `  b1.Zeige();` `  }` `}`	`  1` `b2 = -----` `  3`

In der Anweisung

```
Bruch b2 = b1;
```

wird die neue Referenzvariable `b2` vom Typ `Bruch` angelegt und mit dem Inhalt von `b1` (also mit der Adresse des bereits vorhandenen `Bruch`-Objekts) initialisiert. Es resultiert die folgende Situation im Speicher des Programms:



Hier sollte nur die Möglichkeit der Mehrfachreferenzierung demonstriert werden. Bei einer ernsthaften Anwendung des Prinzips befinden sich die alternativen Referenzen an verschiedenen Stellen des Programms, z.B. in Instanzvariablen verschiedener Objekte. In einem Speditionsverwaltungsprogramm kennen z.B. alle Objekte zu einzelnen Fahrzeugen die Adresse des Planerobjekts, dem sie besondere Ereignisse wie Pannen melden.

4.4.3 Objekte initialisieren

4.4.3.1 Konstruktoren

In diesem Abschnitt werden spezielle Methoden behandelt, die beim Erzeugen von neuen Objekten zum Einsatz kommen, um die Instanzvariablen der Objekte zu initialisieren und/oder andere Arbeiten zu verrichten (z.B. Öffnen einer Datei oder Netzwerkverbindung). Ziel der Konstruktor-Tätigkeit ist es, ein neues Objekt in einem validen Zustand zu bringen und für seinen Einsatz vorzubereiten. Wie Sie bereits wissen, wird zum Erzeugen von Objekten der **new**-Operator verwendet. Als Operand ist ein Konstruktor der gewünschten Klasse anzugeben.

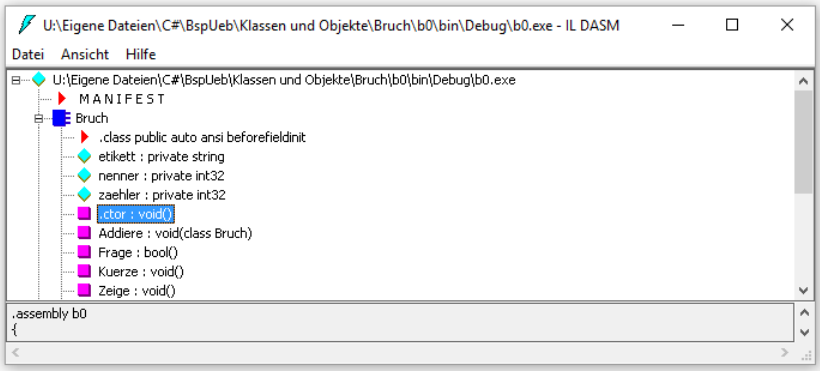
Hat der Programmierer zu einer Klasse *keinen* Konstruktor definiert, erhält sie automatisch einen **Standardkonstruktor**. Weil dieser keine Parameter besitzt, ergibt sich sein Aufruf aus dem Klassennamen durch Anhängen einer leeren Parameterliste, z.B.:

```
Bruch b = new Bruch();
```

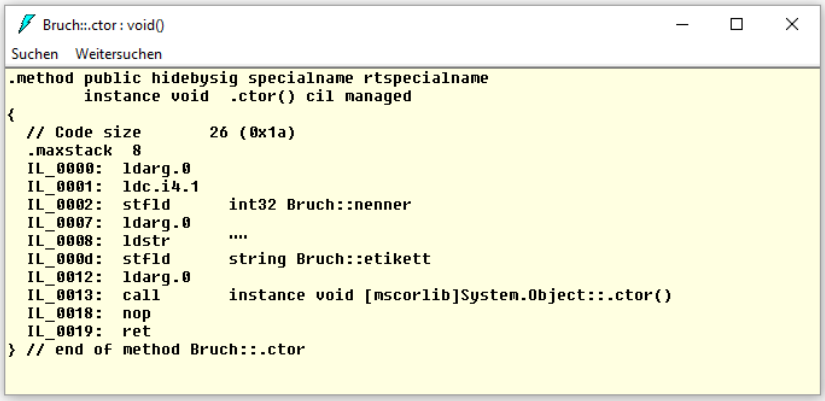
Bei der Deklaration einer *lokalen* Variablen mit Initialisierung kann über das Schlüsselwort **var** und die implizite Typisierung etwas Schreibaufwand gespart und die doppelte Nennung des Klassennamens vermieden werden (vgl. Abschnitt 3.3.6):

```
var b = new Bruch();
```

Wie der CIL-Code zum Standardkonstruktor unserer Klasse `Bruch` (mit dem Namen **.ctor**, Abkürzung für *Konstruktor*) zeigt,[1]



fügt der Compiler automatisch Code für die im Rahmen der Deklaration initialisierten Instanzvariablen ein (betroffen: `nenner` und `etikett`):



Für eine automatische Null-Initialisierung (vgl. Abschnitt 4.2.3) ist hingegen kein CIL-Code erforderlich.

Am Ende (!) des CIL-Codes zum Standardkonstruktor wird der parameterlose Konstruktor der Basisklasse aufgerufen, wobei unsere Klasse `Bruch` direkt von der Urahnklasse **Object** im Namensraum **System** abstammt.

Der Standardkonstruktor hat die Schutzstufe **public**, ist also allgemein verfügbar.

In der Regel ist es beim Klassendesign sinnvoll, mindestens einen Konstruktor *explizit* zu definieren, um das individuelle Initialisieren der Instanzvariablen von neuen Objekten zu ermöglichen. Dabei sind folgende Regeln zu beachten:

---

[1] Zur Anzeige des CIL-Codes wird das im Windows-SDK enthaltene und zusammen mit der Entwicklungsumgebung *Visual Studio 2015 Community* automatisch installierte Hilfsprogramm **ILDasm** verwendet. Nach einer Standardinstallation von Visual Studio 2015 Community (vgl. Abschnitt 2.2) ist das Programm **ildasm.exe** im folgenden Ordner zu finden:

C:\Program Files (x86)\Microsoft SDKs\Windows\v10.0A\bin\NETFX 4.6.1 Tools

- Ein Konstruktor trägt denselben Namen wie die Klasse.
- Ein Konstruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben, auch nicht der Ersatztyp **void**, mit dem wir bei gewöhnlichen Methoden den Verzicht auf einen Rückgabewert dokumentieren müssen.
- Es darf eine Parameterliste definiert werden, was zum Zweck der Initialisierung ja auch unumgänglich ist.
- Sobald man einen expliziten Konstruktor definiert, steht der Standardkonstruktor *nicht* mehr zur Verfügung.
- Ist weiterhin ein paramameterfreier Konstruktor erwünscht, so muss dieser *zusätzlich* definiert werden.
- Der Compiler fügt bei *jedem* Konstruktor automatisch CIL-Code für die im Rahmen der Deklaration initialisierten Instanzvariablen ein (siehe oben), z.B. auch bei einem Konstruktor mit leerem Anweisungsteil.
- Es sind beliebig viele Konstruktoren möglich, die alle denselben Namen und jeweils eine individuelle Parameterliste haben müssen. Das Überladen von Methoden (vgl. Abschnitt 4.3.5) ist also auch bei Konstruktoren erlaubt.
- Während der Standardkonstruktor die Schutzstufe **public** besitzt, haben explizite Konstruktoren wie gewöhnliche Methoden die voreingestellte Schutzstufe **private**. Wenn sie für beliebige fremde Klassen zur Objektkreation verfügbar sein sollen, ist also in der Definition der Modifikator **public** anzugeben.
- Konstruktoren können nicht direkt aufgerufen, sondern nur als Argument des **new**-Operators verwendet werden.

Für die Klasse `Bruch` eignet sich z.B. der folgende Konstruktor mit Parametern zur Initialisierung aller Instanzvariablen:

```
public Bruch(int zpar, int npar, string epar) {
  Zaehler = zpar;
  Nenner  = npar;
  Etikett = epar;
}
```

Beachten Sie bitte: Weil die „beantragten“ Initialisierungswerte nicht direkt den Feldern zugewiesen, sondern durch die Eigenschaften `Zaehler`, `Nenner` und `Etikett` geschleust werden, bleibt die Datenkapselung erhalten. Wie jede beliebige andere Methode einer Klasse muss natürlich auch ein Konstruktor so entworfen sein, dass die Objekte der Klasse unter allen Umständen konsistent und funktionstüchtig sind. In der Klassendokumentation sollte darauf hingewiesen werden, dass dem Wunsch, den Nenner eines neuen `Bruch`-Objekts per Konstruktor auf den Wert 0 zu setzen, *nicht* entsprochen wird, und dass stattdessen der Wert 1 resultiert.[1]

Wenn weiterhin auch ein parameterfreier Konstruktor verfügbar sein soll, muss dieser explizit definiert werden, z.B. mit leerem Anweisungsteil:

```
public Bruch() {}
```

Im folgenden Programm werden beide Konstruktoren eingesetzt:

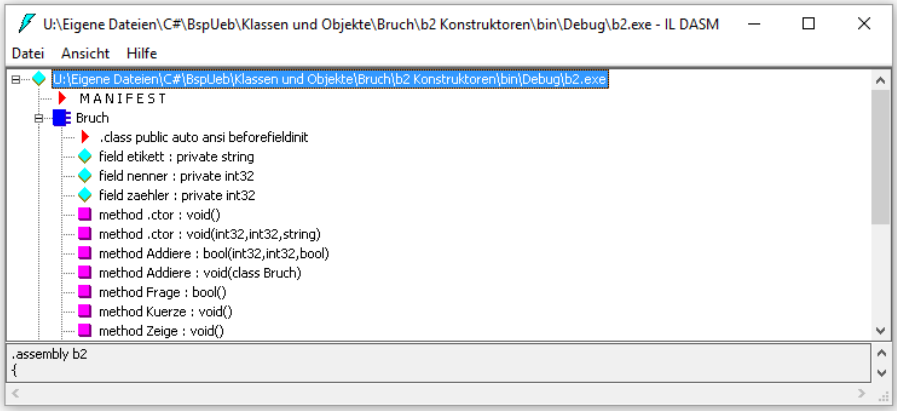
[1] Eine sinnvolle Reaktion auf den Versuch, ein defektes Objekt zu erstellen, kann darin bestehen, im Konstruktor eine so genannte *Ausnahme* zu werfen und dadurch den Aufrufer über das Scheitern seiner Absicht zu informieren. Mit der Kommunikation über Ausnahmeobjekte werden wir uns später beschäftigen.

Quellcode	Ausgabe
`using System;` `class Bruchrechnung {` `  static void Main() {` `  var b1 = new Bruch(1, 2, "b1 = ");` `  var b2 = new Bruch();` `  b1.Zeige();` `  b2.Zeige();` `  }` `}`	`  1` `b1 =  -----` `  2` `  0` `-----` `  1`

Von der Regel, dass Konstruktoren nur über den **new**-Operator genutzt werden können, gibt es eine Ausnahme: Zwischen Parameterliste und Anweisungsblock eines Konstruktors darf ein anderer Konstruktor derselben Klasse über das Schlüsselwort **this** angegeben werden, z.B.:

```
public Bruch() : this(0, 1, "unbekannt") {}
```

Wie das Windows-SDK - Hilfsprogramm **ILDasm** für die aktuelle Ausbaustufe des Bruchrechnungs-Assemblies zeigt, führen die Konstruktoren einer Klasse unter dem Namen **.ctor** die Liste der Instanzmethoden an:



4.4.3.2 Objektinitialisierer

Öffentliche Felder und Eigenschaften (siehe Abschnitt 4.5) können seit der C# - Version 3.0 bei der Objektkreation auch ohne spezielle Konstruktordefinition initialisiert werden. Dazu wird hinter den Konstruktoraufruf eine durch geschweifte Klammern begrenzte Liste von Name-Wert - Paaren gesetzt. Diese neue Option wurde unter der Bezeichnung **Objektinitialisierer** zur Unterstützung der LINQ-Technik (*Language Integrated Query*) eingeführt (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**).

In der `Bruch`-Klassendefinition könnte man auf den parametrisierten Konstruktor

```
public Bruch(int zpar, int npar, string epar) {
  Zaehler = zpar;
  Nenner  = npar;
  Etikett = epar;
}
```

verzichten und stattdessen Objektinitialisierer verwenden, wobei allerdings der Schreibaufwand für die Anwender der Klasse `Bruch` steigt, z.B.:

Quellcode	Ausgabe
`using System;` `class BruchRechnung {` `  static void Main() {` `  var b3 = new Bruch() { Zaehler = 3,` `  Nenner = 7,` `  Etikett = "b3 = " };` `  b3.Zeige();` `  }` `}`	`  3` `b3 = -----` `  7`

Bei der Verwendung eines Objektinitialisierers darf eine leere Parameterliste weggelassen werden, z.B.:

```
var b3 = new Bruch { Zaehler = 3, Nenner = 7, Etikett = "b3 = " };
```

Durch die Objektinitialisierer werden Konstruktoren nicht überflüssig, denn sie können neben Wertzuweisungen beliebige andere Initialisierungsarbeiten ausführen (z.B. Dateien öffnen) und dazu Methoden aufrufen.

4.4.4 Abräumen überflüssiger Objekte durch den Garbage Collector

Wenn keine Referenz mehr auf ein Objekt zeigt, wird es vom **Garbage Collector** (Müllsammler) der CLR entsorgt, und der belegte Speicher wird freigegeben.

Eine lokale Referenzvariable wird beim Verlassen ihres Deklarationsbereichs ungültig, also spätestens beim Beenden der Methode. Man kann eine Referenzvariable aktiv von einem Objekt „entkoppeln“, indem man ihr den Wert **null** (Verweis auf nichts) oder aber ein alternatives Referenzziel zuweist. Es ist jedoch durchaus möglich (und normal), dass ein Objekt die erzeugende Methode überlebt, weil eine Referenz nach Außen transportiert worden ist (z.B. per Rückgabewert, vgl. Abschnitt 4.4.5.2).

Zur Tätigkeit des Garbage Collectors können einige Anlässe führen, die z.B. von Richter (2006, S. 491) beschrieben werden (z.B. Speichermangel). In jedem Fall ist es für Entwickler kaum vorhersehbar, wann und in welcher Reihenfolge obsolete Objekte entsorgt werden.

Vermutlich sind Programmiereinsteiger vom Garbage Collector nicht sonderlich beeindruckt. Schließlich war im Manuskript noch nie die Rede davon, dass man sich um den belegten Speicher nach Gebrauch kümmern müsse. Der in einer Methode von lokalen Variablen belegte Speicher wird bei *jeder* Programmiersprache frei gegeben, sobald die Ausführung der Methode beendet ist. Demgegenüber muss der von Objekten belegte Speicher bei älteren Programmiersprachen (z.B. C++) nach Gebrauch explizit wieder frei gegeben werden. In Anbracht der Objektmassen, die ein typisches Programm (z.B. ein Grafikeditor) benötigt, ist einiger Aufwand erforderlich, um eine Verschwendung von Speicherplatz zu verhindern. Mit seinem vollautomatischen Garbage Collector vermeidet C# lästigen Aufwand und zwei kritische Fehlerquellen:

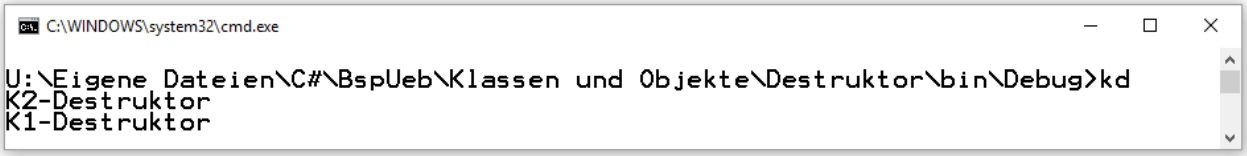
- Weil der Programmierer keine Verpflichtung (und Berechtigung) zum Entsorgen von Objekten hat, kann es nicht zu Programmabstürzen durch Zugriff auf voreilig vernichtete Objekte kommen.
- Es entstehen keine **Speicherlöcher** (engl.: *memory leaks*) durch versäumte Speicherfreigaben bei überflüssig gewordenen Objekten.

Sollen die Objekte einer Klasse vor dem Entsorgen noch spezielle Aufräumaktionen durchführen, ist als Gegenstück zu den Konstruktoren ein so genannter **Destruktor** zu definieren, der ggf. vom Garbage Collector aufgerufen werden. Er trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.

Eine wichtige Tätigkeit von Destruktoren ist die Freigabe von Ressourcen, die *nicht* von der CLR verwaltet werden (z.B. Datei-, Netzwerk- oder Datenbankverbindungen). Weil wir noch keinen Umgang mit solchen Ressourcen hatten, beschränken wir uns auf ein inhaltsfreies Beispiel, das aber immerhin die Tätigkeit des Garbage Collectors zum Programmende dokumentiert. In der **Main()** - Methode der Startklasse `KD` wird ein Objekt der Klasse `K2` erzeugt, die von der Klasse `K1` abstammt:

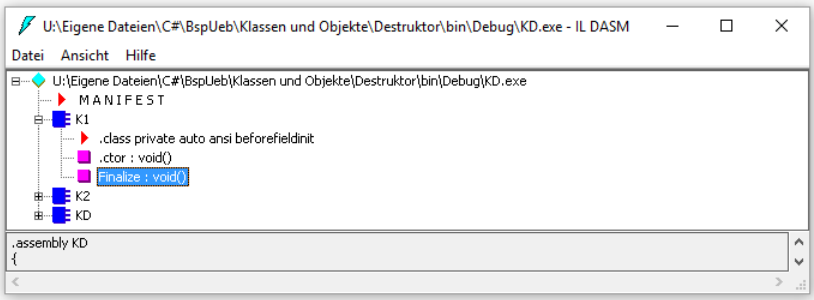
```
using System;

class K1 {
   ~K1() {
      Console.WriteLine("K1-Destruktor");
   }
}

class K2 : K1 {
   ~K2() {
      Console.WriteLine("K2-Destruktor");
   }
}

class KD {
   static void Main() {
      K2 k2 = new K2();
   }
}
```

Wie die Ausgabe zeigt, werden alle Destruktoren entlang des Stammbaums aufgerufen:



Vom Destruktor der Urahnklasse **Object** ist nichts zu sehen, weil er keine Ausgabe macht (und wohl auch sonst nicht viel tut).

Bei einer Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** stellt man fest, dass der Destruktor eigentlich den Namen **Finalize()** trägt:



In C# ist aber die Destruktoren-Syntax mit Tilde und Klassenname zu verwenden. Vermutlich wollten die C# - Designer die Besonderheit der Finalisierung gegenüber anderen Methoden hervorheben (Richter 2006, S. 482).

Solange eine Klasse nur Speicher belegt, muss kein Destruktor definiert werden. Verwendet eine Klasse hingegen auch sogenannte unverwaltete Ressourcen (z.B. Datei-, Netzwerk- oder Datenbankverbindungen), dann muss sich der Klassendesigner um die Freigabe dieser Ressourcen kümmern, wobei die Definition eines Destruktors einen wichtigen Baustein bildet.[1]

---

[1] https://msdn.microsoft.com/en-us/library/498928w2(v=vs.110).aspx

Wir werden später selbstverständlich unverwaltete Ressourcen in unseren Programmen einsetzen. Allerdings werden wir diese Ressourcen nicht direkt ansprechen, sondern FCL-Klassen benutzen, die solche Ressourcen kapseln und sich um das Finalisieren kümmern. Eine gut entworfene Klasse mit Kontakt zu unverwalteten Ressourcen bietet eine **Dispose()** - Methode an, damit man belegte Ressourcen (z.B. geöffnete Dateien) möglichst frühzeitig freigeben kann, statt (eventuell stunden- oder tagelang) auf das Wirken des Garbage Collectors zu warten.

Im weiteren Kursverlauf wird die korrekte Nutzung der **Dispose()** - Methode im Zusammenhang mit unverwalteten Ressourcen mehrfach ein wichtiges Thema sein, während sich keine Notwendigkeit zur Definition eines Destruktors ergeben wird. Wer doch einmal in diese Verlegenheit kommt, muss folgende Regeln beachten:

- Ein Destruktor trägt denselben Namen wie die Klasse, wobei das Tilde-Zeichen (~) voranzustellen ist.
- Ein Destruktor liefert grundsätzlich *keinen* Rückgabewert, und es wird bei der Definition *kein* Typ angegeben.
- Pro Klasse ist nur *ein* Destruktor erlaubt. Dieser muss eine leere Parameterliste haben.
- Destruktoren können nicht aufgerufen werden. Es ist ausschließlich der automatische Aufruf durch den Garbage Collector vorgesehen.
- Bei der Definition eines Destruktors sind Modifikatoren überflüssig und verboten.

4.4.5 Objektreferenzen verwenden

In diesem Abschnitt geht es um Wertparameter und Methodenrückgaben mit Referenztyp sowie um das Schlüsselwort **this**, mit dem in einer Methode das aktuell handelnde Objekt angesprochen werden kann.

4.4.5.1 Objektreferenzen als Wertparameter

Wir haben schon festgehalten, dass die formalen Wertparameter einer Methode wie *lokale Variablen* funktionieren, die beim Methodenaufruf mit den Werten der Aktualparameter initialisiert werden. Methodeninterne Änderungen bei den Werten dieser lokalen Variablen wirken sich *nicht* auf die eventuell als Wertaktualparameter verwendeten Variablen der rufenden Methode aus. Bei einem Wertparameter mit *Referenztyp* wird ebenfalls der Wert des Aktualparameters (eine Objektreferenz) beim Methodenaufruf in eine lokale Variable *kopiert*. Es wird jedoch keinesfalls eine Kopie des referenzierten Objekts (auf dem Heap) erstellt, so dass die aufgerufene Methode über ihre lokale Referenzvariable auf das Originalobjekt zugreift und dort ggf. Veränderungen vornimmt.[1]

Von den beiden `Addiere()` - Überladungen der Klasse `Bruch` verfügt die ältere Variante über einen Wertparameter mit Referenztyp:

```
public void Addiere(Bruch b) {
  zaehler = zaehler*b.nenner + b.zaehler*nenner;
  nenner = nenner*b.nenner;
  Kuerze();
}
```

Mit dem Aufruf dieser Methode wird ein Objekt beauftragt, den via Parameter spezifizierten Bruch zum eigenen Wert zu addieren und das Resultat gleich zu kürzen.

---

[1] Wertparameter mit Referenztyp arbeiten also analog zu den Referenzparametern (vgl. Abschnitt 4.3.1.3.2), insofern beide einer Methode Einwirkungen auf die Außenwelt ermöglichen. Die Referenzparameter sind in C# vor allem deshalb aufgenommen worden, um den Zeit und Speicherplatz sparenden *call by reference* auch bei Werttypen zu ermöglichen. Wir werden mit den so genannten Strukturen noch Werttypen kennen lernen, die ähnlich umfangreich sein können wie Klassentypen.

Zähler und Nenner des „fremden“ `Bruch`-Objekts können per Parametername und Punktoperator trotz Schutzstufe **private** direkt angesprochen werden, weil der Zugriff in einer `Bruch`-Methode stattfindet. Hier liegt *kein* Verstoß gegen das Prinzip der Datenkapselung vor, weil der Zugriff durch eine klasseneigene Methode erfolgt, die vom Klassendesigner gut konzipiert sein sollte.

Dass in einer `Bruch`-Methodendefinition ein Parameter vom Typ `Bruch` verwendet wird, ist übrigens weder „zirkulär“ noch ungewöhnlich; schließlich sollen Brüche auch mit ihresgleichen interagieren können.

In obiger `Addiere()` - Überladung bleibt das per Parameter ansprechbare `Bruch`-Objekt unverändert. Sofern entsprechende Zugriffsrechte vorliegen, was bei Parametern vom Typ des agierenden Objekts stets der Fall ist, kann eine Methode das Parameter-Objekt aber durchaus auch verändern. Als Beispiel erweitern wir die `Bruch`-Klasse um die Methode `DuplWerte()`, die ein Objekt beauftragt, seinen Zähler und Nenner auf ein anderes `Bruch`-Objekt zu übertragen, das per Referenzparameter bestimmt wird:

```
public void DuplWerte(Bruch bc) {
   bc.zaehler = zaehler;
   bc.nenner = nenner;
}
```

In folgendem Programm wird das `Bruch`-Objekt `b1` beauftragt, die `DuplWerte()` - Methode auszuführen, wobei als Parameter eine Referenz auf das Objekt `b2` übergeben wird:

Quellcode	Ausgabe
`using System;` `class Bruchrechnung {` `  static void Main() {` `  var b1 = new Bruch(1, 2, "b1 = ");` `  var b2 = new Bruch(5, 6, "b2 = ");` `  b1.Zeige();` `  b2.Zeige();` `  b1.DuplWerte(b2);` `  Console.WriteLine("Nach DuplWerte():\n");` `  b2.Zeige();` `  }` `}`	`  1` `b1 =  -----` `  2` `  5` `b2 =  -----` `  6` `Nach DuplWerte():` `  1` `b2 =  -----` `  2`

4.4.5.2 Rückgabewerte mit Referenztyp

Soll ein methodenintern erzeugtes Objekt das Ende der Methodenausführung überleben, muss eine Referenz außerhalb der Methode geschaffen werden, was z.B. über einen Rückgabewert mit Referenztyp geschehen kann.

Zur Demonstration des Verfahrens erweitern wir die Klasse `Bruch` um die Methode `Klone()`, welche ein Objekt beauftragt, einen neuen Bruch anzulegen, mit den Werten der eigenen Instanzvariablen zu initialisieren und die Adresse an den Aufrufer zu übergeben:[1]

```
public Bruch Klone() {
   return new Bruch(zaehler, nenner, etikett);
}
```

[1] Bei einer für die breitere Öffentlichkeit gedachten Klasse sollte auch eine die Schnittstelle **ICloneable** (siehe Kapitel 8) implementierende Vervielfältigungsmethode angeboten werden, obwohl diese Schnittstelle durch semantische Unklarheit von begrenztem Wert ist, was im Kapitel 8 über Schnittstellen (Interfaces) noch näher erläutert wird.

Im folgenden Beispiel wird das durch `b2` referenzierte `Bruch`-Objekt in der von `b1` ausgeführten Methode `Klone()` erstellt:

Quellcode	Ausgabe

```
using System;
class Bruchrechnung {
   static void Main() {
      var b1 = new Bruch(1, 2, "b1 = ");
      b1.Zeige();
      var b2 = b1.Klone();
      b2.Zeige();
   }
}
```

```
        1
b1 =  -----
        2

        1
b1 =  -----
        2
```

4.4.5.3 this als Referenz auf das aktuelle Objekt

Gelegentlich ist es sinnvoll oder erforderlich, dass ein handelndes Objekt sich selbst ansprechen bzw. seine eigene Adresse als Methodenaktualparameter verwenden kann. Dies ist mit dem Schlüsselwort **this** möglich, das innerhalb einer Instanzmethode wie eine Referenzvariable funktioniert. In folgendem Beispiel ermöglicht die **this**-Referenz die Verwendung von Formalparameternamen, die mit den Namen von Instanzvariablen übereinstimmen:

```
public bool Addiere(int zaehler, int nenner, bool autokurz) {
   if (nenner != 0) {
      this.zaehler = this.zaehler * nenner + zaehler * this.nenner;
      this.nenner = this.nenner * nenner;
      if (autokurz)
         this.Kuerze();
      return true;
   } else
      return false;
}
```

Außerdem wird beim `Kuerze()` - Aufruf durch die (nicht erforderliche) **this**-Referenz verdeutlicht, dass die Methode vom aktuell handelnden Objekt ausgeführt wird. Später werden Sie noch weit relevantere **this**-Verwendungsmöglichkeiten kennen lernen.

4.5 Eigenschaften

4.5.1 Syntaktisch elegante Zugriffsmethoden

Sollen fremde Klassen Lese- und/oder Schreibzugriff auf ein gekapseltes (also privates) Feld erhalten, sind entsprechende Zugriffsmethoden zu definieren. Im `Bruch`-Beispiel könnte man z.B. für das `nenner`-Feld die folgenden Methoden definieren:

```
public int GibNenner() {
  return nenner;
}

public void SetzeNenner(int value) {
  if (value != 0)
    nenner = value;
}
```

Infolgedessen sähe der klassenfremde Zugriff auf einen Nenner z.B. so aus:

```
b1.SetzeNenner(2);
Console.WriteLine(b1.GibNenner());
```

Mit den *Eigenschaften* (engl.: *properties*), die wegen ihrer großen Bedeutung schon in der ersten Variante des `Bruch`-Beispiels genutzt wurden, bietet C# die Möglichkeit, Zugriffe auf gekapselte Felder syntaktisch zu vereinfachen. Aus den beiden obigen Methodendefinitionen wird die folgende Eigenschaftsdefinition:

```
public int Nenner {
  get {
    return nenner;
  }
  set {
    if (value != 0)
      nenner = value;
  }
}
```

Für den Klassen*designer* ändert sich nicht allzu viel: Im Rahmen einer Eigenschaftsdefinition mit Modifikatoren, Datentyp und Namen ist ein **get**- und ein **set**-Block mit nahe liegender Syntax zu implementieren. Erwähnenswert ist, dass im **set**-Block der vom Aufrufer übergebene neue Wert ohne Formalparameterdefinition über das Schlüsselwort **value** angesprochen wird.

Für den Klassen*anwender* ändert sich mehr, weil eine Eigenschaft weit intuitiver zu verwenden ist als korrespondierende Zugriffsmethoden, z.B.:

```
b1.Nenner = 2;
Console.WriteLine(b1.Nenner);
```

Sogar die Aktualisierungs-Operatoren (vgl. Abschnitt 3.5.8) werden unterstützt, z.B.:

```
b1.Nenner += 2;
```

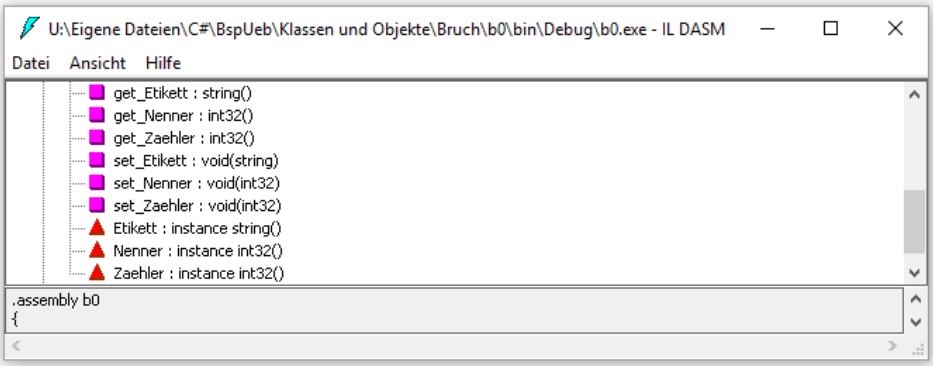
Um aus der Perspektive fremder Klassen eine **Read**- bzw. **Write-Only** Eigenschaft zu realisieren, verzichtet man einfach auf die **set**- bzw. **get**-Implementation. Durch die folgende Variante der `Nenner`-Definition aus der Klasse `Bruch` entsteht eine Eigenschaft, die nur den lesenden Zugriff erlaubt, wobei die klasseneigenen Methoden nach wie vor auf die private Instanzvariable `nenner` schreibend zugreifen dürfen:

```
public int Nenner {
    get {
        return nenner;
    }
}
```

Weiterhin ist es möglich, den für eine Eigenschaft gültigen Zugriffsschutz entweder für den **get**- oder für **set**-Zugriff zu verschärfen. Das „oder“ ist hier im exklusiven Sinn zu verstehen. Durch die folgende Variante der `Nenner`-Definition aus der Klasse `Bruch` entsteht eine Eigenschaft, die den Lesezugriff für beliebige Klassen erlaubt, den Schreibzugriff hingegen auf die klasseneigenen Methoden beschränkt:

```
public int Nenner {
    get {
        return nenner;
    }
    private set {
        if (value != 0)
            nenner = value;
    }
}
```

Wie eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, erstellt der Compiler zu den Eigenschaften unserer Klasse `Bruch` jeweils ein Paar von Zugriffsmethoden:



Wenngleich die Eigenschaften zu gefallen wissen, handelt es sich doch eher um *syntactic sugar* (Mössenböck 2016, S. 3) als um einen essentiellen Vorteil gegenüber anderen Programmiersprachen wie Java und C++.[1]

4.5.2 Automatisch implementierte Eigenschaften

Bei Eigenschaften, die lediglich ein privates Feld kapseln und auf jeden Eingriff beim Lesen und Schreiben verzichten kommt man seit der C# - Version 3.0 mit einem minimalen Definitionsaufwand aus. Man kann sich auf Modifikatoren, Typ, Namen sowie die Schlüsselwörter **get** und **set** beschränken. Die **get**- bzw. **set**-Implementation kann man ebenso dem Compiler überlassen wie die Deklaration des gekapselten Felds. Bei der `Zaehler`-Eigenschaft unserer Klasse `Bruch` Fall können also die Deklaration

```
int zaehler;
```

und die Definition

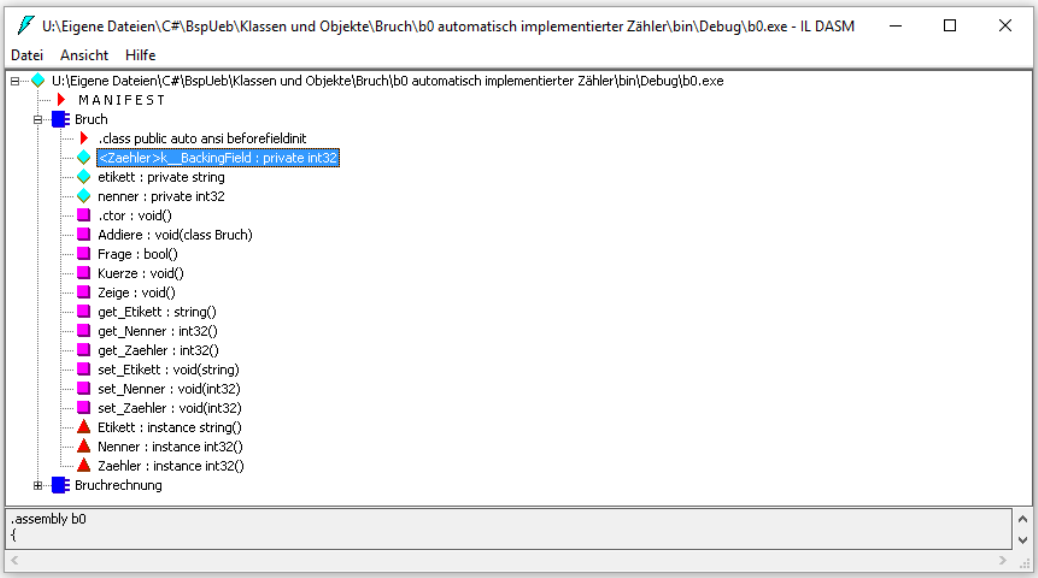
```
public int Zaehler {
   get {
      return zaehler;
   }
   set {
      zaehler = value;
   }
}
```

äquivalent ersetzt werden durch die Definition:

```
public int Zaehler {get; set;}
```

Wie das Windows-SDK - Hilfsprogramm **ILDasm** zeigt, ergänzt der Compiler automatisch ein privates **Backing Field** mit passendem Typ:

[1] Beim schreibenden Zugriff geht im Vergleich zu einer traditionellen **set**-Methode die Möglichkeit verloren, durch eine Rückgabe vom Typ **bool** zu signalisieren, ob die gewünschte Wertzuweisung durchgeführt wurde. Allerdings hat eine **set**-Methode per Konvention den Rückgabetyp **void**, und eine **bool**-Rückgabe wird von fremden Programmierern eventuell ignoriert. Als Warnung vor einer nicht ausgeführten Wertzuweisung sollte eher eine Ausnahme geworfen werden (siehe unten), was natürlich auch in der **set**-Implementation einer Eigenschaft möglich ist.



Dieses Feld ist ausschließlich über die Eigenschaft ansprechbar. Es ist also ausgeschlossen, dass verschiedene Methoden schreibend auf das Feld zugreifen, so dass bei einem Fehler relativ gute Voraussetzungen für die Ursachensuche bestehen.

Mit der C# - Version 6.0 sind für automatisch implementierte Eigenschaften (engl. Bezeichnung *Auto-Implemented Properties*) zwei Verbesserungen eingeführt worden:

- Man kann einen Initialisierungswert vereinbaren, z.B.:
  ```
  public int Auto { get; set; } = 4711;
  ```
- Man kann sich auf den **get**-Zugriff beschränken, z.B.:
  ```
  public int Auto { get; } = 4711;
  ```
  Die in diesem Fall meist unverzichtbare Initialisierung kann bei der Deklaration erfolgen (siehe Beispiel) oder in einem Konstruktor, z.B.:
  ```
  public Bsp() {
      Auto = 4711;
  }
  ```

Im `Bruch`-Einstiegsbeispiel wird der Transparenz halber auf automatisch implementierte Eigenschaften verzichtet.

4.5.3 Zeitaufwand bei Eigenschafts- und Feldzugriffen

Microsoft verspricht, dass Eigenschaftszugriffe aufgrund von Optimierungen des Just-In-Time - Compilers der CLR in der Regel *nicht* aufwändiger sind als Feldzugriffe.[1]

> Generally, because of just-in-time optimizations, properties are no more expensive than fields.

Vom JIT-Compiler der CLR ist zu erwarten, dass er den (meist sehr kleinen) Maschinencode an jeder Aufrufstelle einsetzt, um bei Eigenschaftszugriffen den Aufwand eines gewöhnlichen Methodenaufrufs zu vermeiden. Diese auch von anderen Compilern eingesetzte Technik zur Optimierung von Funktions- bzw. Methodenaufrufen bezeichnet man als **Inlining**.

Um die obige Aussage von Microsoft zu überprüfen, wurden im Bruchrechnungsbeispiel für den lesenden und schreibenden Eigenschafts- bzw. Feldzugriff auf den Zähler bzw. Nenner eines Bruchs Vergleichsmessungen unter den folgenden Bedingungen ausgeführt:

---

[1] Siehe z.B. http://msdn.microsoft.com/en-us/library/65zdfbdt.aspx

- .NET-Framework 4.6.1
- Windows 10 (64 bit)
- Rechner mit Intel-CPU Core i3 550
- Erstellungskonfiguration: Release
  Zum Unterschied zwischen Release- und Debug-Konfiguration siehe Abschnitt 2.2.4.4.
- Zielplattform: Any CPU

Die Messergebnisse für die **Release-Konfiguration** bestätigen Microsofts Aussage:[1]

a) Eigenschafts- bzw. Feldzugriff auf den Zähler eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	66,0593	66,5597
Schreiben	66,0586	68,0614

b) Eigenschafts- bzw. Feldzugriff auf den Nenner eines `Bruch`-Objekts:

Zugriffsart	Zeitaufwand für 100 Millionen Zugriffe in Millisekunden	
	via Eigenschaft	direkter Feldzugriff
Lesen	63,6241	68,6722
Schreiben	66,9649	63,9997

In mehreren Methoden der Klasse `Bruch` sind aktuell noch direkte Feldzugriffe zu finden, z.B.:

```
public void Addiere(Bruch b) {
   zaehler = zaehler*b.nenner + b.zaehler*nenner;
   nenner = nenner*b.nenner;
   Kuerze();
}
```

Schreibende Feldzugriffe durch Eigenschaftszugriffe zu ersetzen, bringt ohne Performanz-Beeinträchtigung (in der Release-Konfiguration) den Vorteil, dass alle schreibenden Zugriffe auf ein Feld in einem einzigen **setter** stattfinden. Ist z.B. in einem komplexen Programm der Grund für mysteriöse Wertveränderungen zu ermitteln, lassen sich über *einen* Haltepunkt im **setter** alle Wertveränderungen erfassen. Wenn viele Methoden schreibend auf ein Feld zugreifen, ist die Fehlersuche weitaus schwieriger. Es wird daher explizit empfohlen, schreibende Feldzugriffe durch Eigenschaftszugriffe zu ersetzen.[2]

In der folgenden Variante der obigen `Addiere()` - Methode sind die schreibenden Feldzugriffe durch Eigenschaftszugriffe ersetzt:

```
public void Addiere(Bruch b) {
   Zaehler = zaehler*b.nenner + b.zaehler*nenner;
   Nenner = nenner*b.nenner;
   Kuerze();
}
```

[1] Für die Debug-Konfiguration ergeben sich stark abweichende Ergebnisse mit einem Performanznachteil der Eigenschaftszugriffe.

[2] Es kann leider nicht garantiert werden, dass im Manuskript ab jetzt keine schreibenden Feldzugriffe mehr zu sehen sind.

4.6 Statische Member und Klassen

Neben den *objektbezogenen* Feldern, Eigenschaften, Methoden und Konstruktoren unterstützt C# auch *klassenbezogene* Varianten. Syntaktisch werden diese Member in der Deklaration bzw. Definition durch den Modifikator **static** gekennzeichnet, und man spricht oft von *statischen* Feldern, Methoden, Eigenschaften etc. Ansonsten gibt es bei der der Deklaration bzw. Definition kaum Unterschiede zwischen einem Instanz-Member und dem analogen statischen Member.

Auch bei den statischen Klassen-Membern gilt für den Zugriffsschutz:

- Voreingestellt ist die Schutzstufe **private**, so dass eine Verwendung nur klasseneigenen Methoden erlaubt ist.
- Durch Modifikatoren kann eine alternative Schutzstufe festgelegt werden (z.B. **public**).

4.6.1 Statische Felder und Eigenschaften

In unserem Bruchrechnungsbeispiel soll ein statisches Feld die Anzahl der bisher im aktuellen Programmlauf erzeugten `Bruch`-Objekte aufnehmen:

```
using System;
public class Bruch {
  int zaehler,
      nenner = 1;
  string etikett = "";

   static int anzahl;

  public Bruch(int zpar, int npar, String epar) {
    Zaehler = zpar;
    Nenner  = npar;
    Etikett = epar;
     anzahl++;
  }

  public Bruch() {
     anzahl++;
  }

   . . .

}
```

Ein statisches Feld kann in klasseneigenen Methoden (objektbezogen oder statisch) direkt angesprochen werden. Im Beispiel wird die (automatisch auf 0 initialisierte) Klassenvariable `anzahl` in den beiden Instanzkonstruktoren inkrementiert.

Sofern Methoden *fremder* Klassen (durch den Modifikator **public**) der direkte Zugriff auf eine Klassenvariable gewährt wird, müssen diese dem Variablennamen ein Vorspann aus Klassennamen und Punktoperator voranstellen, z.B.:

```
Console.Writeline("Bisher wurden " + Bruch.anzahl + " Brüche erzeugt");
```

In unserem Beispiel wird das statische Feld `anzahl` aber *ohne* **public**-Modifikator deklariert, so dass der direkte Zugriff klasseneigenen Methoden vorbehalten bleibt.

Während jedes Objekt einer Klasse über einen eigenen Satz mit allen Instanzvariablen verfügt, existiert eine klassenbezogene Variable nur *einmal*. Sie wird beim Laden der Klasse angelegt und erhält per Voreinstellung dieselbe Null-Initialisierung wie eine Instanzvariable (vgl. Abschnitt 4.2.3). Al-

ternative Initialisierungen können in der Variablendeklaration oder im statischen Konstruktor (siehe Abschnitt 4.6.4) vorgenommen werden.

In der folgenden Tabelle werden wichtige Unterschiede zwischen Klassen- und Instanzvariablen zusammengestellt:

	Instanzvariablen	**Klassenvariablen**
Deklaration	ohne Modifikator **static**	mit Modifikator **static**
Zuordnung	Jedes Objekt besitzt einen eigenen Satz mit allen Instanzvariablen.	Klassenbezogene Variablen sind nur *einmal* vorhanden.
Existenz	Instanzvariablen werden beim Erzeugen des Objektes angelegt und initialisiert. Sie werden ungültig, wenn das Objekt nicht mehr referenziert ist.	Klassenvariablen werden beim Laden der Klasse angelegt und initialisiert.

Damit im erweiterten Bruchrechnungsbeispiel fremde Klassen trotz Datenkapselung die Anzahl der bisher erzeugten `Bruch`-Objekte in Erfahrung bringen können, wird noch eine **statische Eigenschaft** mit **public**-Zugriff ergänzt:

```
public static int Anzahl {
   get {
      return anzahl;
   }
   private set {
      anzahl = value;
   }
}
```

Weil die **set**-Funktionalität als **private** deklariert ist, können fremde Klassen den `anzahl`-Wert zwar ermitteln, aber nicht verändern.

In Methoden der Klasse `Bruch` den privaten **setter** der Eigenschaft `Anzahl` statt direkter Feldzugriffe zu verwenden, hat nach den Überlegungen aus Abschnitt 4.5.3 folgende Effekte:

- Vereinfachte Suche nach fehlerhaften Schreibzugriffen
- Keine Verschlechterung der Performanz in der Release-Konfiguration

4.6.2 Wiederholung zur Kategorisierung von Variablen

Mittlerweile haben wir verschiedene Variablensorten kennen gelernt, wobei die Sortenbezeichnung unterschiedlich motiviert war. Um einer möglichen Verwirrung vorzubeugen, folgt nun eine Zusammenfassung bzw. Wiederholung. Die folgenden Begriffe sollten Ihnen keine Probleme mehr bereiten:

- **Lokale Variablen ...**
  werden in Methoden deklariert,
  landen auf dem Stack,
  werden **nicht** automatisch initialisiert,
  sind nur in den Anweisungen des innersten Blocks verwendbar,
  existieren, bis der innerste Block endet.

- **Instanzvariablen ...**
  werden außerhalb jeder Methode deklariert,
  landen (als Bestandteile von Objekten) auf dem Heap,
  werden automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo eine Referenz zum Objekt vorliegt und Zugriffsrechte bestehen.
- **Klassenvariablen ...**
  werden außerhalb jeder Methode mit dem Modifikator **static** deklariert,
  werden automatisch mit dem typspezifischen Nullwert initialisiert,
  sind verwendbar, wo Zugriffsrechte bestehen.
- **Referenzvariablen ...**
  zeichnen sich durch ihren speziellen *Inhalt* aus (Referenz auf ein Objekt). Es kann sich sowohl um lokale Variablen (z.B. `b1` in der **Main()** - Methode von `Bruchrechnung`) als auch um Instanzvariablen (z.B. `etikett` in der `Bruch`-Definition) oder um Klassenvariablen handeln.

Die Variablen in C# kann man einteilen nach ...

- **Datentyp**
  Es sind vor allem zu unterscheiden:
  - Werttypen (z.B. **int**, **double**, **bool**)
  - Referenztypen (mit Objektreferenzen als Inhalt).
- **Zuordnung**
  Eine Variable kann zu einem Objekt (Instanzvariable), zu einer Klasse (statische Variable) oder zu einer Methode (lokale Variable) gehören. Damit sind weitere Eigenschaften wie Ablageort, Lebensdauer, Sichtbarkeitsbereich und Initialisierung festgelegt (siehe oben).

4.6.3 Statische Methoden

Es ist in vielen Situationen sinnvoll oder sogar unvermeidlich, einer *Klasse* Handlungskompetenzen (Methoden) zu verschaffen. So muss z.B. beim Programmstart die **Main()** - Methode der Startklasse ausgeführt werden, bevor irgendein Objekt existiert. Das Erzeugen von Objekten gehört gerade zu den typischen Aufgaben der statischen Methode **Main()**, wobei es sich nicht unbedingt um Objekte der eigenen Klasse handeln muss.

Wie eine statische (und öffentliche) Methode von fremden Klassen genutzt werden kann, ist Ihnen längst bekannt, weil die statische Methode **WriteLine()** der Klasse **Console** bisher in fast jedem Beispielprogramm zum Einsatz kam, z.B.:

```
Console.WriteLine("Hallo");
```

Vor den Namen der gewünschten Methode setzt man (durch den Punktoperator getrennt) den Namen der angesprochenen Klasse, der eventuell durch den Namensraumbezeichner vervollständigt werden muss, je nach Namensraumzugehörigkeit der Klasse und vorhandenen **using**-Direktiven am Anfang des Quellcodes (vgl. Abschnitt 1.2.7).

Trotz Ihrer Erfahrung mit diversen **Main()** - Methoden soll auch im Kontext unserer Klasse `Bruch` das Definieren einer statischen Methode geübt werden. Zur Vereinfachung von Anweisungsfolgen nach dem folgenden Muster

```
var b = new Bruch(0, 1, "Benutzerdefiniert: ");
b.Frage();
b.Kuerze();
```

definieren wir eine Klassenmethode namens `BenDef()`, die eine Referenz auf ein neues `Bruch`-Objekt mit benutzerdefinierten und gekürzten Werten liefert:

```
  public static Bruch BenDef(string e) {
    var b = new Bruch(0, 1, e);
    if (b.Frage()) {
      b.Kuerze();
      return b;
    } else
      return null;
  }
```

Bei fehlerhaften Benutzereingaben liefert die Methode den Referenzwert **null** zurück. Mit Hilfe der neuen Methode kann die obige Sequenz durch eine einzelne Anweisung ersetzt werden:

Quellcode	Eingaben (fett) und Ausgabe
`using System;` `class Bruchrechnung {` `  static void Main() {` `  var b = Bruch.BenDef("Benutzerdefiniert: ");` `  if (b != null)` `  b.Zeige();` `  else` `  Console.WriteLine("b zeigt auf null");` `  }` `}`	`Zaehler: 26` `Nenner : 39` `  2` `Benutzerdefiniert: -----` `  3`

Wird eine Klassenmethode von anderen Methoden der *eigenen* Klasse (objekt- oder klassenbezogen) verwendet, muss der Klassenname *nicht* angegeben werden. Es ist aber erlaubt und kann der Klarheit dienen.

Weil der Modifikator **static** nicht Signatur-relevant ist (vgl. Abschnitt 4.3.5) kann es in einer Klasse zu einer Instanzmethode keine statische Methode mit demselben Namen und derselben Parameterliste geben.

In früheren Abschnitten waren mit *Methoden* stets *objektbezogene* Methoden (*Instanzmethoden*) gemeint. Dies soll auch weiterhin so gelten.

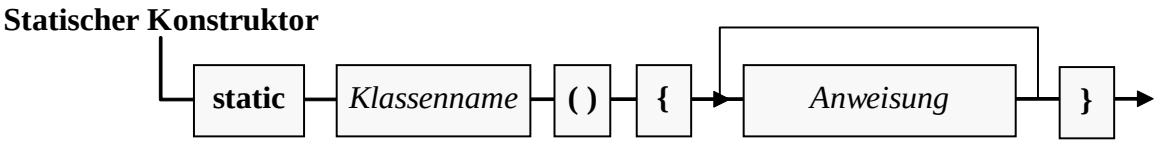
4.6.4 Statische Konstruktoren

Analog zu den Instanzkonstruktoren (siehe Abschnitt 4.4.3), die beim Erzeugen eines Objekts ausgeführt werden und sich um die Initialisierung von Instanzvariablen kümmern, kann für jede Klasse ein statischer Konstruktor zur Initialisierung von Klassenvariablen definiert werden. Er wird beim Laden der Klasse (also beim Erstellen des ersten Objekts oder beim ersten Zugriff auf ein statisches Mitglied) automatisch ausgeführt und kann nirgends explizit aufgerufen werden.

Eine statische Variable durchläuft folgende Initialisierungen:

- Zunächst erhält sie den typspezifischen Nullwert (vgl. Abschnitt 4.2.3).
- Ggf. wird anschließend die Initialisierung aus der Deklarationsanweisung vorgenommen.
- Schließlich wird der statische Konstruktor ausgeführt.

Naheliegender Weise ist pro Klasse nur *ein* statistischer Konstruktor erlaubt, und seine Parameterliste muss leer bleiben. In der Definition ist dem Klassennamen der Modifikator **static** voranzustellen, während andere Modifikatoren verboten sind. Insbesondere dürfen keine Zugriffsmodifikatoren angegeben werden. Diese werden auch nicht benötigt, weil ein statischer Konstruktor ohnehin nur vom Laufzeitsystem aufgerufen wird. Insgesamt erhalten wir das folgende Syntaxdiagramm:



In einer etwas künstlichen Erweiterung des `Bruch`-Beispiels soll der parameterfreie Instanzkonstruktor zufallsabhängige, aber pro Programmlauf identische Werte zur Initialisierung der Felder `zaehler` und `nenner` verwenden:

```
public Bruch() {
  Zaehler = zaehlerVoreinst;
  Nenner = nennerVoreinst;
  Anzahl++;
}
```

Dazu erhält die `Bruch`-Klasse private statische Felder, die vom statischen Konstruktor beim Laden der Klasse auf Zufallswerte gesetzt werden sollen:

```
static readonly int zaehlerVoreinst;
static readonly int nennerVoreinst;
```

Der Modifikator **readonly** sorgt dafür, dass die Felder nach der Initialisierung nicht mehr geändert werden können (vgl. Abschnitt 4.2.2). Im statischen Konstruktor wird ein Objekt der Klasse **Random** aus dem Namensraum **System** erzeugt und dann per **Next()** - Methodenaufruf mit der Produktion von **int**-Zufallswerten beauftragt:

```
static Bruch() {
   var zuf = new Random();
   zaehlerVoreinst = zuf.Next(1,7);
   nennerVoreinst = zuf.Next(zaehlerVoreinst,9);
   Console.WriteLine("Klasse Bruch geladen");
}
```

Außerdem protokolliert der statische Konstruktor noch das Laden der Klasse, z.B.:

Quellcode	Ausgabe
`using System;` `class Bruchrechnung {` `  static void Main() {` `  var b1 = new Bruch(), b2 = new Bruch();` `  b1.Zeige(); b2.Zeige();` `  }` `}`	`Klasse Bruch geladen` `  1` `-----` `  5` `  1` `-----` `  5`

4.6.5 Statische Klassen

Besitzt eine Klasse *ausschließlich* statische Member, ist das Erzeugen von Objekten nicht sinnvoll. Man kann es mit dem Modifikator **static** in der Klassendefinition verhindern, z.B.

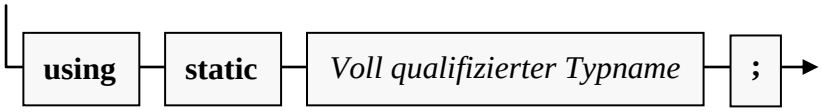
```
public static class Service {
   . . .
}
```

Auch die FCL enthält etliche Klassen, die ausschließlich statische Member enthalten und damit nicht zum Erzeugen von Objekten konzipiert sind. Mit der Klasse **Math** aus dem Namensraum **System** haben wir ein wichtiges Beispiel bereits kennengelernt.

4.6.6 Statische Member eines Typs importieren

Die in Abschnitt 1.2.7 beschriebene und in praktisch jedem unserer Programme verwendete **using**-Direktive dient dazu, einen Namensraum zu importieren, so dass die dort befindlichen Typen im Programm ohne Namensraumpräfix angesprochen werden können. Seit der C# - Version 6.0 ist zusätzlich eine **using**-Variante verfügbar, welche die statischen Member eines Typs importiert, so dass im Programm beim Zugriff weder der Namensraum noch die Klasse anzugeben sind:

Import der statischen Member eines Typs



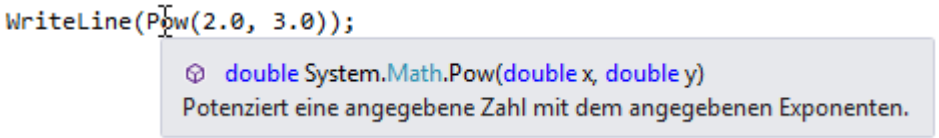
Im folgenden Beispielprogramm aus Abschnitt 3.5.2 können aufgrund einer Verwendung der vertrauten **using**-Variante die beiden Klassen **Console** und **Math** ohne Namensraumpräfix angesprochen werden:

```
using System;
class Prog{
  static void Main(){
    Console.WriteLine(Math.Pow(2.0, 3.0));
  }
}
```

Importiert man mit der in C# 6.0 hinzu gekommenen **using**-Variante die statischen Member aus den beiden Klassen, so lassen sie sich wie statische Member der Klasse `Prog` verwenden:

```
using static System.Console;
using static System.Math;
class Prog {
    static void Main() {
        WriteLine(Pow(2.0, 3.0));
    }
}
```

Mit der statischen **using**-Variante lässt sich zwar Schreibarbeit sparen, doch wird gleichzeitig das Verständnis des Quellcodes erschwert, weil verborgen bleibt, welche Klasse bei einer Methode am Werk ist. Das Visual Studio kompensiert den Nachteil teilweise, indem es die Klassenzugehörigkeit anzeigt, sobald der Mauszeiger über einem Methodennamen verharrt:



4.7 Vertiefungen zum Thema Methoden

4.7.1 Rekursive Methoden

Innerhalb einer Methode darf man selbstverständlich nach Belieben *andere* Methoden aufrufen. Es ist aber auch zulässig und manchmal sogar sinnvoll, dass eine Methode *sich selbst* aufruft. Solche *rekursiven* Aufrufe erlauben eine elegante Lösung für ein Problem, das sich sukzessiv auf stets einfachere Probleme desselben Typs reduzieren lässt, bis man schließlich zu einem direkt lösbaren Problem gelangt.

Als Beispiel betrachten wir die Ermittlung des größten gemeinsamen Teilers (GGT) zu zwei natürlichen Zahlen, die z.B. in der `Bruch`-Methode `Kuerze()` benötigt wird. Sie haben bereits zwei *iterative* (mit einer Schleife realisierte) Realisierungen des Euklidischen Lösungsverfahrens kennen

gelernt: In Abschnitt 1.1 wurde ein sehr einfacher Algorithmus benutzt, den Sie später in einer Übungsaufgabe (siehe Abschnitt 3.7.4) durch einen effizienteren Algorithmus (unter Verwendung der Modulo-Operation) ersetzt haben. Im aktuellen Abschnitt betrachten wir noch einmal die effizientere Variante, wobei zur Vereinfachung der Darstellung der GGT-Algorithmus vom restlichen Kürzungsverfahren getrennt und in eine eigene (private) Methode namens `GGTi()` ausgelagert wird:

```
int GGTi(int a, int b) {
   int rest;
   do {
      rest = a % b;
      a = b;
      b = rest;
   } while (rest > 0);
   return a;
}

public void Kuerze() {
   if (zaehler != 0) {
      int teiler = GGTi(Math.Abs(zaehler), Math.Abs(nenner));
      Zaehler /= teiler;
      Nenner /= teiler;
   } else
      Nenner = 1;
}
```

Die mit einer **do-while** - Schleife operierende Methode `GGTi()` kann durch die folgende rekursive Variante `GGTr()` ersetzt werden:

```
int GGTr(int a, int b) {
   int rest = a % b;
   if (rest == 0)
      return b;
   else
      return GGTr(b, rest);
}
```

Statt eine Schleife zu benutzen, arbeitet die rekursive Methode nach folgender Logik:

- Ist der Parameter `a` durch den Parameter `b` restfrei teilbar, dann ist `b` der GGT, und der Algorithmus ist beendet:

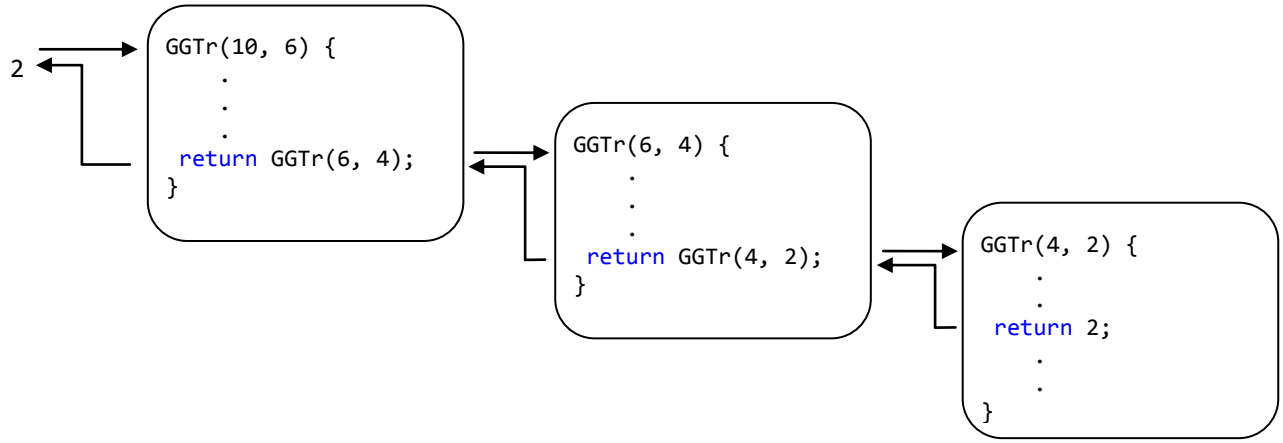
  ```
  return b;
  ```

- Anderenfalls wird das Problem, den GGT von `a` und `b` zu finden, auf das einfachere Problem zurückgeführt, den GGT von `b` und `(a % b)` zu finden, und die Methode `GGTr()` ruft sich selbst mit neuen Aktualparametern auf. Dies geschieht elegant im Ausdruck der **return**-Anweisung:

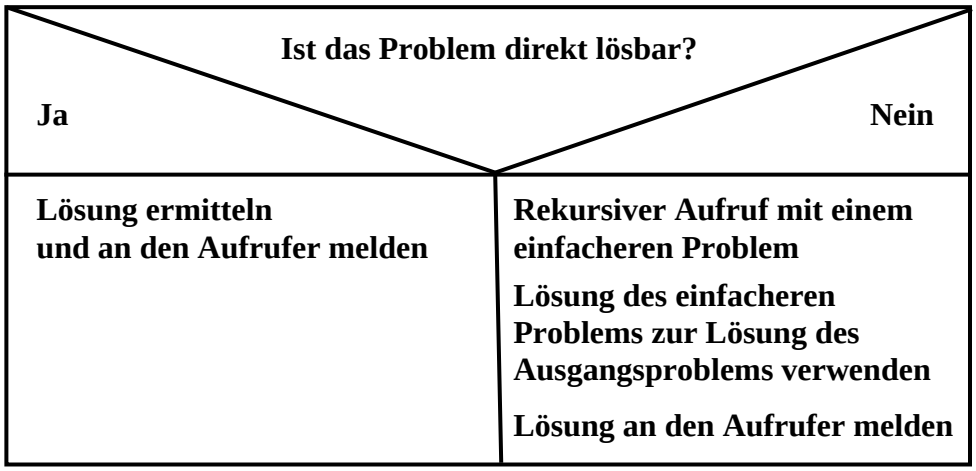
  ```
  return GGTr(b, rest);
  ```

Im iterativen Algorithmus wird übrigens derselbe Trick zur Reduktion des Problems verwendet. Den zugrunde liegenden Satz der mathematischen Zahlentheorie kennen Sie schon aus der oben erwähnten Übungsaufgabe in Abschnitt 3.7.4.

Wird die Methode `GGTr()` z.B. mit den Argumenten 10 und 6 aufgerufen, kommt es zu folgender Aufrufverschachtelung:



Generell läuft eine rekursive Methode mit Lösungsübermittlung per Rückgabewert nach der im folgenden **Struktogramm** beschriebenen Logik ab:



Im Beispiel ist die Lösung des einfacheren Problems sogar identisch mit der Lösung des ursprünglichen Problems.

Wird bei einem fehlerhaften Algorithmus der linke Zweig nie oder zu spät erreicht, dann erschöpfen die geschachtelten Methodenaufrufe die Stack-Kapazität, und es kommt zu einem Ausnahmefehler:

```
Process is terminated due to StackOverflowException.
```

Zu einem rekursiven Algorithmus (per Selbstaufruf einer Methode) existiert stets ein äquivalenter *iterativer* Algorithmus (per Wiederholungsanweisung). Rekursive Algorithmen lassen sich zwar oft eleganter formulieren als die iterativen Alternativen, benötigen aber durch die hohe Zahl von Methodenaufrufen in der Regel mehr Rechenzeit.

4.7.2 Operatoren überladen

Nicht nur Methoden können in C# überladen werden, sondern auch die *Operatoren* (**+, *** etc.), was z.B. in der Klasse **String** mit dem „+“ - Operator geschehen ist, so dass wir Zeichenfolgen bequem verketten können. Generell geht es beim Überladen von Operatoren darum, dem Anwender einer Klasse syntaktisch elegante Lösungen für Aufgaben anzubieten, die letztlich einen Methodenaufruf erfordern. Statt die Argumente in einer Aktualparameterliste anzugeben, können sie bei reduziertem Syntaxaufwand um ein Operatorzeichen gruppiert werden. Dieses Zeichen erhält eine neue, zusätzliche Bedeutung für Argumente aus der betroffenen (mit der Operatorüberladung ausgestatteten) Klasse.

Mit den aktuell in der Klasse `Bruch` vorhandenen Methoden lässt sich nur umständlich ein neues Objekt `b3` als Summe von zwei vorhandenen Objekten `b1` und `b2` erzeugen, z.B.:

```
var b3 = b1.Klone();
b3.Addiere(b2);
```

Es wäre eleganter, wenn derselbe Zweck mit folgender Anweisung erreicht werden könnte:

```
Bruch b3 = b1 + b2;
```

Um dies zu ermöglichen, definieren wir eine neue statische `Bruch`-Methode mit dem merkwürdigen Namen **operator+**, die ausgeführt werden soll, wenn das Pluszeichen zwischen zwei `Bruch`-Objekten auftaucht:

```
public static Bruch operator+(Bruch b1, Bruch b2) {
  var temp = new Bruch(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler,
                       b1.nenner * b2.nenner, "");
  temp.Kuerze();
  return temp;
}
```

Beim Überladen von Operatoren sind u.a. folgende Regeln zu beachten:

- Es ist grundsätzliche eine *statische* Definition erforderlich.
- Als Namen verwendet man das Schlüsselwort **operator** mit dem jeweiligen Operationszeichen als Suffix.

Nähere Hinweise finden sich z.B. bei Mössenböck (2016, S. 78ff).

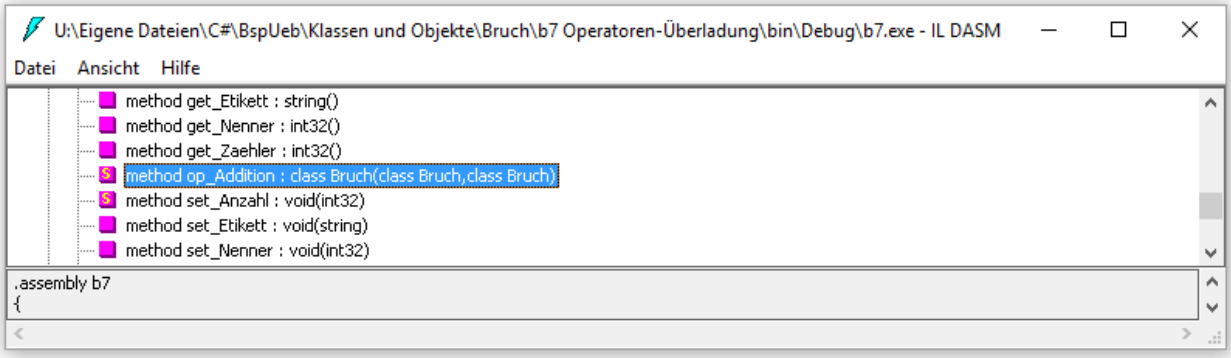
Mit dem überladenen „+“ - Operator lassen sich `Bruch`-Additionen nun sehr übersichtlich formulieren:

Quellcode	Ausgabe

```
using System;
class Bruchrechnung {
  static void Main() {
    var b1 = new Bruch(1, 2, "b1 = ");
    b1.Zeige();
    var b2 = new Bruch(1, 4, "b2 = ");
    b2.Zeige();
    var b3 = b1 + b2;
    b3.Etikett = "Summe = ";
    b3.Zeige();
  }
}
```

```
      1
b1 = -----
      2

      1
b2 = -----
      4

         3
Summe = -----
         4
```

Wie das Windows-SDK - Hilfsprogramm **ILDasm** zeigt, resultiert aus unserer Operatorenüberladung im Assembly die statische Methode **op_Addition()**:

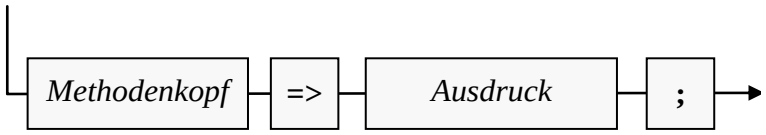


4.7.3 Methodendefinition mit Rumpf im Lambda-Stil

Über die so genannten *Lambda-Ausdrücke* ist es seit C# 3.0 möglich, eine simple (z.B. aus einem einzigen Ausdruck bestehende) Funktionalität, die bei bestimmten Gelegenheiten ausgeführt werden soll, auf syntaktisch einfache Weise zu definieren. Seit C# 6.0 lässt sich die Lambda-Syntax auch dazu verwenden, um den Rumpf einer Methode oder Eigenschaft zu definieren (engl. Bezeichnung: *Expression Bodied Functions and Properties*).

Das folgende Syntaxdiagramm beschreibt die Definition einer Methode mit Hilfe der Lambda-Syntax:

Methodendefinition mit Rumpf im Lambda-Stil



Auf den **Lambda-Operator** (=>) muss ein Ausdruck folgen mit einem Typ, der zum Rückgabetyp der Methode passt. Die im folgenden Beispiel definierte Methode `Max()` liefert das Maximum von zwei **int**-Argumenten:

```
static int Max(int a, int b) => a >= b ? a : b;
```

Ob diese Notation einen Fortschritt darstellt im Vergleich zur Standardvariante,

```
static int Max(int a, int b) {return a >= b ? a : b;}
```

die nur unwesentlich länger und dabei leichter zu lesen ist, scheint fraglich.

Bei einer Methode mit dem Rückgabetyp **void** muss bei Verwendung des Lambda-Stils auch der Ausdruck diesen Typ haben, z.B.:

```
static void WM(int a, int b) =>
   Console.WriteLine("{0} modulo {1} = {2}", a, b, a%b);
```

Das folgende Programm zeigt die beiden Methoden in Aktion:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static int Max(int a, int b) => a >= b ? a : b;` `  static void WM(int a, int b) =>` `  Console.WriteLine("{0} modulo {1} = {2}", a, b, a % b);` `  static void Main() {` `  Console.WriteLine(Max(3, 4));` `  WM(17, 3);` `  }` `}`	`4` `17 modulo 3 = 2`

4.8 Komposition

Bei den Feldern einer Klasse sind beliebige Datentypen zugelassen, auch Referenztypen. Z.B. ist in der aktuellen `Bruch`-Definition eine Instanzvariable vom Referenztyp **String** vorhanden. Es ist also möglich, vorhandene Klassen als Bestandteile von neuen, komplexeren Klassen zu verwenden. Neben der später noch ausführlich zu behandelnden Vererbung ist diese *Komposition* (alias: *Aggregation*) eine effektive Technik zur Wiederverwendung von vorhandenen Typen bei der Definition von neuen Typen. Außerdem ist sie im Sinne einer realitätsnahen Modellierung unverzichtbar, denn

auch ein reales Objekt (z.B. eine Firma) enthält andere Objekte[1] (z.B. Mitarbeiter, Kunden), die ihrerseits wiederum Objekte enthalten (z.B. ein Gehaltskonto und einen Terminkalender bei den Mitarbeitern) usw.

Man kann den Standpunkt einnehmen, dass die Komposition eine selbstverständliche, wenig spektakuläre Angelegenheit sei, eigentlich nur ein neuer Begriff für eine längst vertraute Situation (Instanzvariablen mit Referenztyp). Es ist tatsächlich für den weiteren Lernerfolg im Kurs unkritisch, wenn Sie den Rest des aktuellen Abschnitts mit dem recht länglichen Beispiel zur Komposition überspringen.

Wir erweitern das Bruchrechnungsprogramm um eine Klasse namens `Aufgabe`, die Trainingssitzungen unterstützen soll und dazu mehrere `Bruch`-Objekte verwendet. In der `Aufgabe`-Klassendefinition tauchen vier Instanzvariablen vom Typ `Bruch` auf:

```
using System;

public class Aufgabe {
   Bruch b1, b2, lsg, antwort;
   char op;

   public Aufgabe(char op_, int b1Z, int b1N, int b2Z, int b2N) {
      op = op_;
      b1 = new Bruch(b1Z, b1N, "1. Argument:");
      b2 = new Bruch(b2Z, b2N, "2. Argument:");
      lsg = new Bruch(b1Z, b1N, "Das korrekte Ergebnis:");
      antwort = new Bruch();
      Init();
   }

   private void Init() {
      switch (op) {
         case '+':  lsg.Addiere(b2);
                    break;
         case '*':  lsg.Multipliziere(b2);
                    break;
      }
   }

   public bool Korrekt {
      get {
         Bruch temp = antwort.Klone();
         temp.Kuerze();
         if (lsg.zaehler == temp.zaehler && lsg.nenner == temp.nenner)
            return true;
         else
            return false;
      }
   }

   public void Zeige(int was) {
      switch (was) {
         case 1:  Console.WriteLine("   " + b1.zaehler +
                  "                " + b2.zaehler);
                  Console.WriteLine(" ------     " + op + "     -----");
                  Console.WriteLine("   " + b1.nenner +
                  "                " + b2.nenner);
                  break;
         case 2: lsg.Zeige(); break;
         case 3: antwort.Zeige(); break;
      }
   }
```

[1] Die betroffenen Personen mögen den Fachterminus *Objekt* nicht persönlich nehmen.

```
    public void Frage() {
       Console.WriteLine("\nBerechne bitte:\n");
       Zeige(1);
       Console.Write("\nWelchen Zähler hat Dein Ergebnis:      ");
       antwort.Zaehler = Convert.ToInt32(Console.ReadLine());
       Console.WriteLine("                                     ------");
       Console.Write("Welchen Nenner hat Dein Ergebnis:      ");
       antwort.Nenner = Convert.ToInt32(Console.ReadLine());
    }

    public void Pruefe() {
       Frage();
       if (Korrekt)
          Console.WriteLine("\n Gut!");
       else {
          Console.WriteLine();
          Zeige(2);
       }
    }

    public void NeueWerte(char op_, int b1Z, int b1N, int b2Z, int b2N) {
       op = op_;
       b1.Zaehler = b1Z; b1.Nenner = b1N;
       b2.Zaehler = b2Z; b2.Nenner = b2N;
       lsg.Zaehler = b1Z; lsg.Nenner = b1N;
       Init();
    }
}
```

Die vier `Bruch`-Objekte in einer `Aufgabe` dienen folgenden Zwecken:

- `b1` und `b2` werden dem Anwender (in der `Aufgabe`-Methode `Frage()`) im Rahmen einer Aufgabenstellung vorgelegt, z.B. zum Addieren.
- In `antwort` landet der Lösungsversuch des Anwenders.
- In `lsg` steht das korrekte Ergebnis.

In folgendem Programm wird die Klasse `Aufgabe` für ein Bruchrechnungstraining verwendet:

```
using System;
class Bruchrechnung {
   static void Main() {
      var auf = new Aufgabe('*', 3, 4, 2, 3);
      auf.Pruefe();
      auf.NeueWerte('+', 1, 2, 2, 5);
      auf.Pruefe();
   }
}
```

Man kann immerhin schon ahnen, wie die praxistaugliche Endversion des Programms einmal arbeiten wird:

```
Berechne bitte:

   3               2
 ------     *     -----
   4               3

Welchen Zaehler hat Dein Ergebnis:     6
                                      ------
Welchen Nenner hat Dein Ergebnis:      12

 Gut!
```

```
Berechne bitte:

  1               2
------    +     -----
  2               5

Welchen Zaehler hat Dein Ergebnis:     3
                                     ------
Welchen Nenner hat Dein Ergebnis:      7

                          9
 Das korrekte Ergebnis: -----
                          10
```

4.9 Innere (geschachtelte) Klassen

Eine Klasse darf neben Feldern, Methoden etc. auch Klassendefinitionen enthalten, wobei *innere (geschachtelte) Klassen* entstehen. Eine Klasse innerhalb einer umgebenden Klasse zu definieren, bietet sich an, wenn die innere Klasse nicht allgemein benötigt wird, sondern nur zur Modellierung von speziellem Zubehör der umgebenden Klasse dient. Bei der voreingestellten Schutzstufe **private** ist die innere Klasse im restlichen Programm nicht sichtbar, kann also dort nicht verwendet werden und auch keine Namenskollision verursachen.

Im folgenden Beispiel werden innerhalb der Klasse `Familie` die Klassen `Tochter` und `Sohn` definiert:

```
using System;

public class Familie {
 string name;
 Tochter t;
 Sohn s;

 public Familie(string name_, string nato, int alto, string naso, int also) {
  name = name_;
  t = new Tochter(this, nato, alto);
  s = new Sohn(this, naso, also);
 }

 public void Info() {
  Console.WriteLine("Die Kinder von Familie {0}:\n", name);
  t.Info();
  s.Info();
 }

 class Tochter {
  Familie f;
  string name;
  int alter;

  public Tochter(Familie f_, string name_, int alt_) {
  f = f_;
  name = name_;
  alter = alt_;
  }

  public void Info() {
  Console.WriteLine("  Ich bin die {0}-jährige Tochter {1} von Familie {2}",
                    alter, name, f.name);
  }
 }
```

```
    public class Sohn {
      Familie f;
      string name;
      int alter;

      public Sohn(Familie f_, string name_, int alt_) {
      f = f_;
      name = name_;
      alter = alt_;
      }

      public void Info() {
      Console.WriteLine("  Ich bin der {0}-jährige Sohn {1} von Familie {2}",
                        alter, name, f.name);
      }
    }
  }
```

Die **Main()** - Methode der Testklasse

```
class FamilienTest {
   static void Main() {
      var f = new Familie("Müller", "Lea", 7, "Leo", 4);
      f.Info();
   }
}
```

liefert folgende Ausgabe:

```
Die Kinder von Familie Müller:

  Ich bin die 7-jährige Tochter Lea von Familie Müller.
  Ich bin der 4-jährige Sohn Leo von Familie Müller.
```

Für das Schachteln von Klassendefinitionen gelten u.a. folgende Regeln:

- Die Methoden der inneren Klasse dürfen auf die privaten Member der umgebenden Klasse zugreifen, was z.B. in der folgenden Anweisung des Beispiels geschieht (`name` ist eine private Instanzvariable in der Klasse `Familie`):

  ```
  Console.WriteLine("  Ich bin der {0}-jährige Sohn {1} von Familie {2}.",
                    alter, name, f.name);
  ```

- Umgekehrt hat die umgebende Klasse *keine* Zugriffsrechte für die privaten Member einer inneren Klasse, weshalb im Beispiel die Konstruktoren und die `Info()` - Methoden der inneren Klassen als **public** definiert werden.
- Innere Klassen besitzen wie die sonstigen Klassen-Member (Felder, Methoden, Eigenschaften etc.) die voreingestellte Schutzstufe **private**.
- Erhält eine innere Klasse die Schutzstufe **public**, dann können in beliebigen Klassen Objekte von diesem Typ erstellt werden, wobei dem Namen der inneren Klasse der Name ihrer Hüllenklasse voranzustellen ist, z.B.:

  ```
  Familie.Sohn faso = new Familie.Sohn(f, "Leo", 4);
  ```

4.10 Verfügbarkeit von Klassen und Klassenmitgliedern

Nachdem die Datenkapselung mehrfach als wesentlicher Vorzug bzw. Kernidee der objektorientierten Programmierung herausgestellt wurde und wiederholt Angaben zur Verfügbarkeit von Klassen bzw. Klassenbestandteilen an verschiedenen Stellen eines .NET - Programms gemacht wurden, sollen die Regeln zum Zugriffsschutz nun zusammengestellt werden, obwohl dabei noch einige kleine Vorgriffe auf das Thema *Vererbung* nötig sind.

Bei einer äußeren (nicht geschachtelten) Klasse kennt C# folgende Stufen der Verfügbarkeit (vgl. ECMA 2006, S. 274):

Modifikator	Die Klasse ist verfügbar …	
	im eigenen Assembly	überall
ohne oder **internal**	ja	nein
public	ja	ja

Unsere als **public** definierte Beispielklasse `Bruch` kann in beliebigen .NET-Programmen mit Zugang zur Assembly-Datei genutzt werden, was gleich in Abschnitt 4.11 demonstriert werden soll.

Für Klassen-Member (Felder, Methoden, Eigenschaften, innere Klassen etc.) unterstützt C# folgende Schutzstufen (siehe ECMA 2006, S. 31):

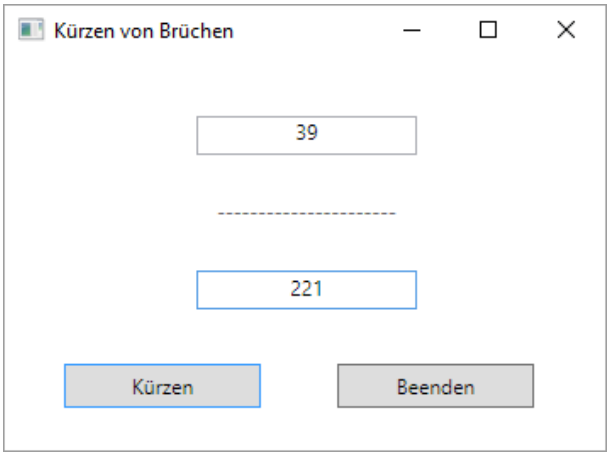
Modifikator(en)	Der Zugriff ist erlaubt für ...				
	eigene Klasse (Abk. K) u. innere Klassen	nicht von K abgel. Klassen im eigenen Assembly	von K abgeleitete Klassen		sonstige Klassen
			im eigenen Assembly	in anderen Assemblies	
ohne oder **private**	ja	nein	nein	nein	nein
internal	ja	ja	ja	nein	nein
protected	ja	nein	geerbte M.	geerbte M.	nein
protected internal	ja	ja	ja	geerbte M.	nein
public	ja	ja	ja	ja	ja

Wir haben die Methoden und Eigenschaften unserer Beispielklasse `Bruch` als **public** definiert und bei den Feldern die voreingestellte Schutzstufe **private** beibehalten.

Die beschriebenen Zugriffsregeln für Klassen und Member gelten analog auch bei später vorzustellenden Typen (Strukturen, Enumerationen, Schnittstellen und Delegaten).

4.11 Bruchrechnungsprogramm mit WPF-Bedienoberfläche

Nachdem Sie nun wesentliche Teile der objektorientierten Programmierung mit C# kennen gelernt haben, ist vielleicht ein weiterer Ausblick auf die nicht mehr allzu ferne Entwicklung von Windows-Programmen mit grafischer Bedienoberfläche als Belohnung und Motivationsquelle angemessen. Schließlich gilt es in diesem Kurs auch die Erfahrung zu vermitteln, dass Programmieren Spaß machen kann. Wir erstellen nun das schon in Abschnitt 1.1.5 präsentierte Bruchkürzungsprogramm mit grafischer Bedienoberfläche:



Bei dieser Gelegenheit werden wir unser Wissen über die Erstellung einer WPF-Anwendung (*Windows Presentation Foundation*) erweitern, um bei der „offiziellen“ Behandlung des Themas in Kapitel 11 schon einige Erfahrungen einbringen zu können. Wir werden im Kurs die WPF - GUI-

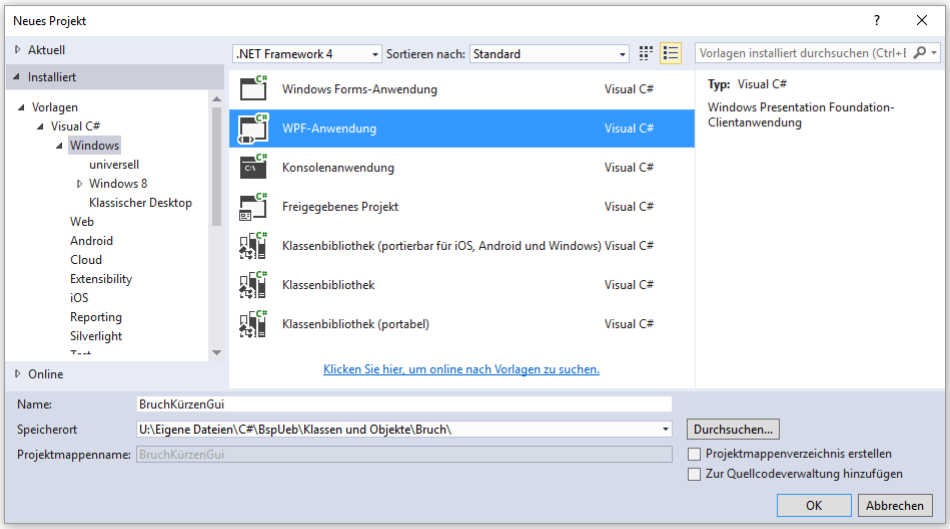
Technik aus später noch im Detail zu diskutierenden Gründen gegenüber den Alternativen Win-Forms (veraltet) und WinRT (alias Metro, keine Unterstützung für Windows 7) bevorzugen.

4.11.1 Projekt anlegen mit der Vorlage WPF-Anwendung

Verwenden Sie im Visual Studio nach

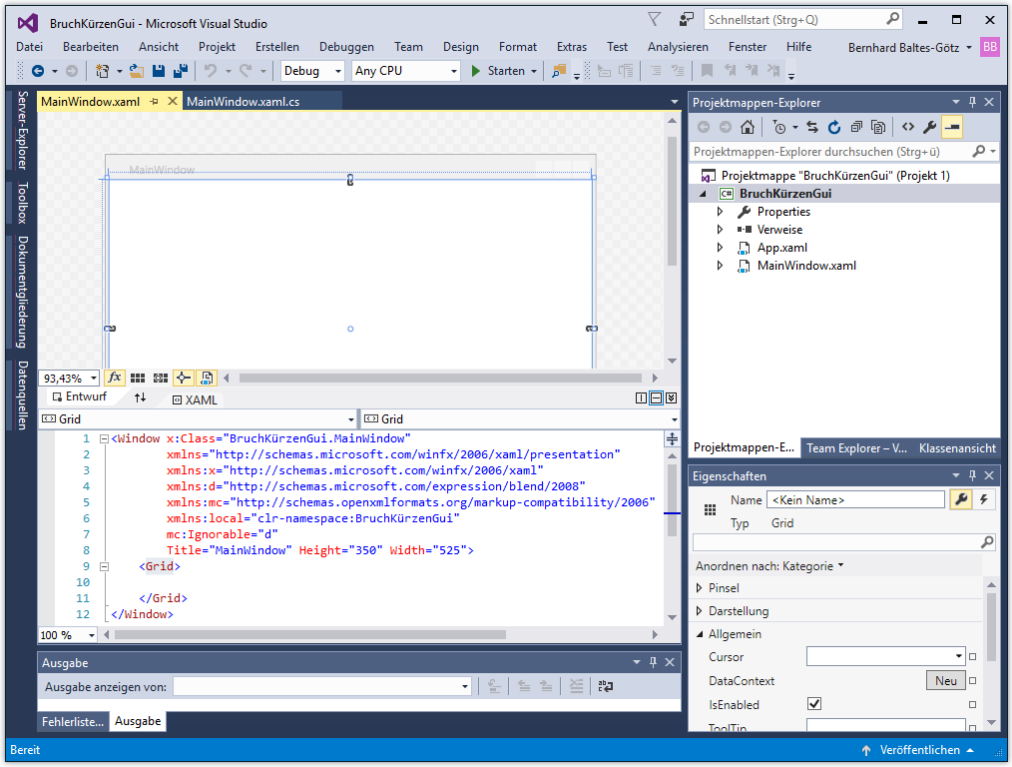
Datei > Neu > Projekt

für ein neues Projekt mit dem Namen `BruchKürzenGui` die Vorlage **WPF-Anwendung**:



Ein möglichst niedriges .NET Framework vorauszusetzen, sorgt für maximale Kompatibilität mit älteren Klienten-PCs. Weil wir ein Assembly mit der Klasse `Bruch` einbinden wollen, darf aber das im dortigen Projekt eingestellte Kompatibilitätsniveau nicht unterschritten werden.

Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im **WPF-Designer** einen Rohling für das Hauptfenster der entstehenden Anwendung:



In der oberen Designer-Zone können wir die Bedienoberfläche unseres Programms mit Hilfe von grafischen Werkzeugen erstellen und dazu konfigurierbare Komponenten (Steuerelemente) aus der Toolbox (siehe Abschnitt 4.11.3) übernehmen. Wie gleich zu sehen sein wird, gestalten wir dabei den Auftritt von Objekten aus der neuen Klasse `MainWindow` im Namensraum `BruchKürzenGui`.

4.11.2 Deklaration der Bedienoberfläche per XAML

Ein zentrales Merkmal der WPF-Technologie besteht darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung (*eXtended Markup Language*) namens XAML (*eXtended Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört also eine XAML-Datei, die im unteren Teil der Designer-Zone erscheint und initial so aussieht:

```
<Window x:Class="BruchKürzenGui.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:BruchKürzenGui"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Element, das mit dem Tag

```
<Window . . . >
```

startete und mit dem Tag

```
</Window>
```

endet, definiert ein Fenster, also das Erscheinungsbild eines Objekts aus einer anwendungseigenen Klasse, die von der FCL-Klasse **Window** abstammt. Initial enthält es im Beispiel:

- ein Attribut namens **x:Class**, das die zugehörige Klasse samt Namensraum nennt, in unserem Fall also `BruchKürzenGui.MainWindow`
- **xmlns**-Attribute, die Namensräume für die anschließend verwendeten XAML-Bestandteile angeben
- ein Attribut namens **mc:Ignorable** mit dem Wert "**d**", das nur bei Verwendung des GUI-Designers *Blend for Visual Studio* relevant ist[1]
- ein **Title**-Attribut für die Fensterbeschriftung
- die Attribute **Height** und **Width** für die initiale Fenstergröße
- ein **Grid**-Element, das als Container für Steuerelemente dient und später noch ausführlich besprochen wird

Der grafische Fenster-Designer ist lediglich ein (willkommenes!) Werkzeug zur Bearbeitung der XAML-Datei.

Die XAML-Datei zu unserem Anwendungsfenster heißt **MainWindow.xaml** und beschreibt die Oberfläche von Objekten der Klasse `MainWindow`. Zu dieser Klasse gehören auch zwei Quellcode-

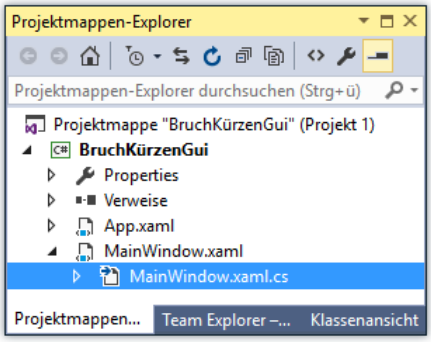
[1] Es sorgt dafür, dass alle Elemente und Attribute mit dem Namensraumpräfix **d** zur Laufzeit ignoriert werden. Um diese Bestandteile kümmert sich das für Animationen und andere aufwendige GUI-Techniken zuständige Programm *Blend for Visual Studio* (frühere Bezeichnung: *Expression Blend*). Dieses Programm ist primär für Designer gedacht, die zusammen mit Entwicklern an großen Projekten arbeiten. Dasselbe Projekt kann alternativ von einem Entwickler mit dem Visual Studio und von einem Designer mit Blend for Visual Studio geöffnet werden.

Dateien mit den Deklarationen bzw. Definitionen der üblichen Klassen-Member (Felder, Methoden, Eigenschaften etc.):

- **MainWindow.xaml.cs**
  Hier werden unsere Beiträge zur Funktionalität der Klasse `MainWindow` landen.
- **MainWindow.g.i.cs**
  Diese Datei wird automatisch durch Übersetzen der XAML - Deklarationen in C# - Quellcode erstellt und vor unseren Blicken relativ gut verborgen, weil direkte Änderungen durch Programmierer *nicht* vorgesehen sind.

Weitere Details zu den beiden C# - Dateien folgen in Abschnitt 4.11.6.

Der Projektmappen-Explorer zeigt die XAML-Datei und die für Programmierer relevante C# - Datei:



Über die Hauptfensterklasse hinaus benötigt eine WPF-Anwendung auch noch eine **Anwendungsklasse**, abstammend von der FCL-Klasse **Application**, wobei erneut eine XAML-Deklarationsdatei und zwei C# - Quellcodedateien beteiligt sind (siehe Abschnitt 4.11.6 für Details). Bei unserem geplanten Beispielprogramm müssen wir uns um diese Dateien *nicht* kümmern. Wer die XAML-Datei **App.xaml** neugierig per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnet, kann z.B. im **Application**-Element feststellen, dass über das Attribut **StartupUri** das beim Programmstart anzuzeigende Fenster festgelegt wird:

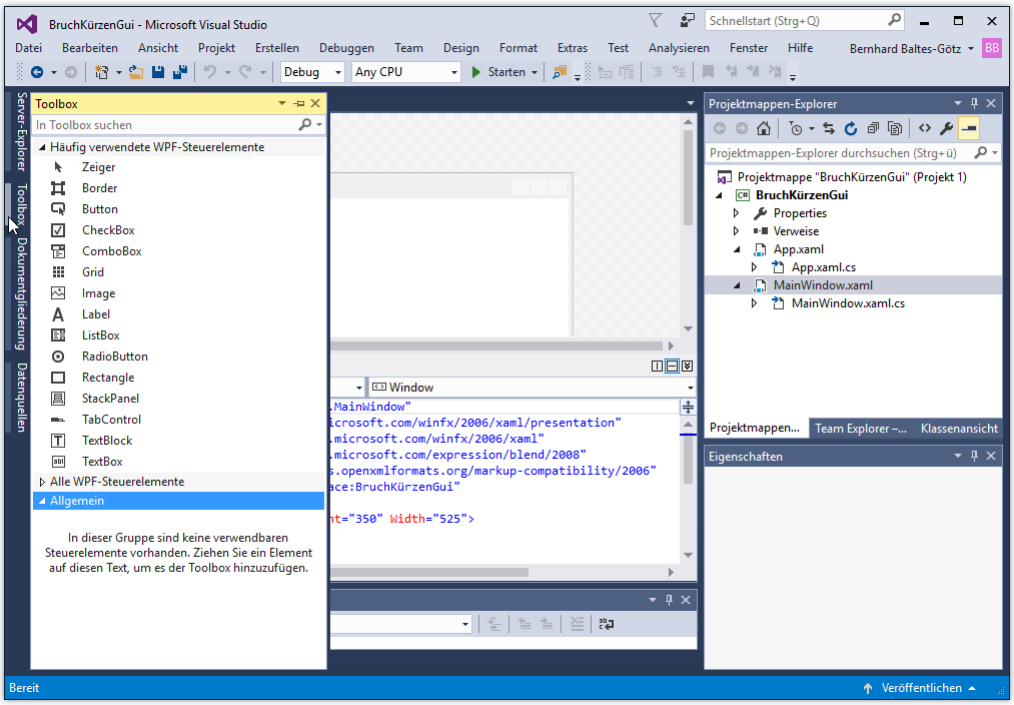
```
<Application x:Class="BruchKürzenGui.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:BruchKürzenGui"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

4.11.3 Steuerelemente aus der Toolbox übernehmen

Öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

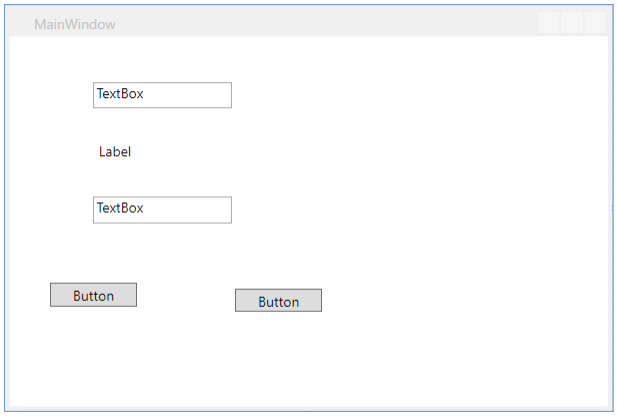
oder per Mausklick auf die **Toolbox**-Schaltfläche am linken Fensterrand:



Erweitern Sie nötigenfalls im **Toolbox**-Fenster die Liste mit den **Häufig verwendeten WPF-Steuerelementen**, und erstellen Sie auf dem Anwendungsfenster zwei **TextBox**-Objekte, ein **Label**-Objekt sowie zwei **Button**-Objekte, ...

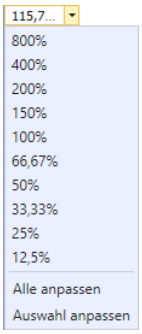
- entweder per Doppelklick auf den jeweiligen **Toolbox**-Eintrag
- oder per Drag & Drop (Ziehen und Ablegen), indem Sie einen linken Mausklick auf den jeweiligen **Toolbox**-Eintrag setzen, den Mauszeiger mit gedrückter Taste zum Ziel bewegen und dort die Taste wieder loslassen.

Das Ergebnis sollte ungefähr so aussehen:

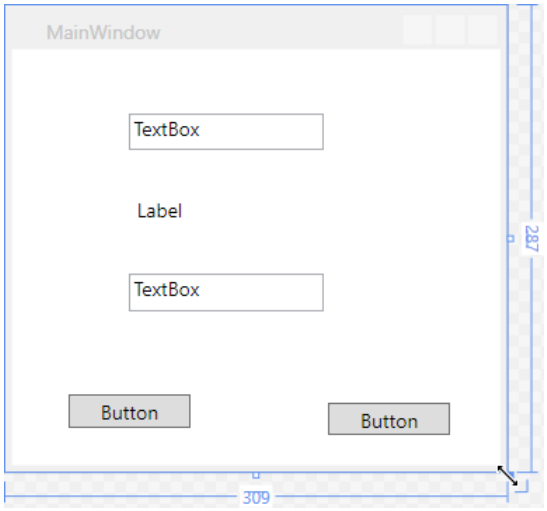


4.11.4 Positionen und Größen der Steuerelemente gestalten

Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen. Zur Erleichterung der Arbeit lässt sich mit dem **Zoom-Werkzeug** des WPF-Designers (unten links) eine passende Ansicht einstellen:

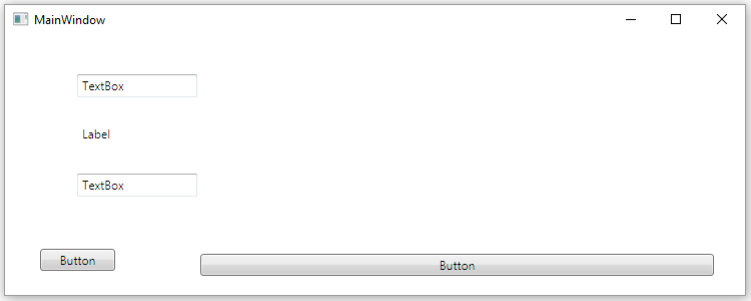


Wählen Sie zunächst für das Hauptfenster eine Größe über den Anfasser unten rechts:

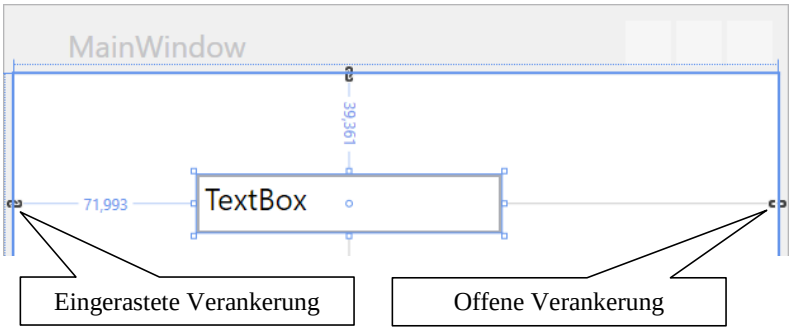


Die aktuelle Breite und Höhe in geräteunabhängigen Pixeln wird numerisch angezeigt. Achten Sie darauf, dass Sie wirklich das Hauptfenster erwischen (ein Objekt der von **Window** abstammenden Klasse `MainWindow`) und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**).

Ändert der Benutzer im laufenden Programm die Fenstergröße, dann hängt das Orts- und Größenverhalten eines Steuerelements davon ab, zu welchen Seiten des umgebenden Containers es einen festen **Randabstand** einhält, z.B.:

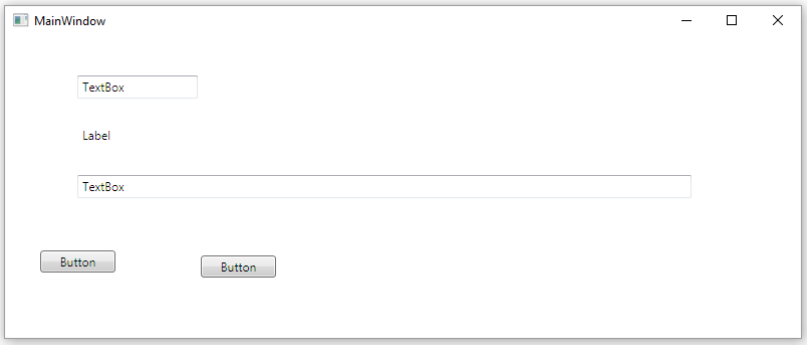


Per Voreinstellung sind die Steuerelemente links und oben angedockt, was bei einem markierten Steuerelement durch Verankerungssymbole an den Fensterrändern angezeigt wird, z.B.:



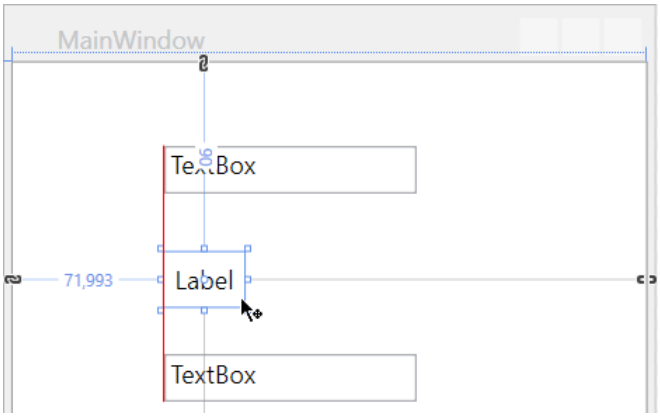
Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Lösen einer Verankerung die gegenüberliegende Seite gerade frei, springt die Verankerung dorthin.

Soll ein Steuerelement z.B. vom horizontalen Größenwachstum des Hauptfensters profitieren, auf vertikale Änderungen aber nicht reagieren,



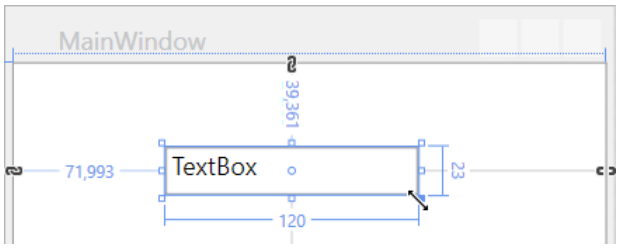
verankert man es oben, links und rechts.

Bei der Positions- und Größenanpassung von Steuerelementen helfen **Ausrichtungs- bzw. Führungslinien**, die auftauchen und dabei anziehend wirken, sobald die horizontale oder vertikale Position von potentiellen Kandidaten für eine Ausrichtung erreicht wird. Im folgenden Beispiel wurde das **Label**-Objekt horizontal bewegt:

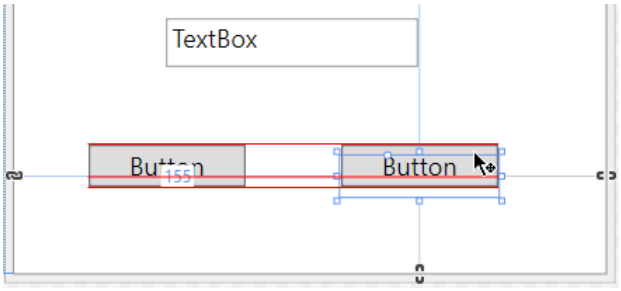


Man kann z.B. so vorgehen, um die Steuerelemente für das Beispielprogramm anzuordnen:

- Größe und Position für das obere Textfeld festlegen, z.B.:



  Breite und Höhe des Textfelds werden numerisch angezeigt.
- Das **Label**-Objekt und das untere Textfeld nacheinander ...
  - linksbündig mit ungefähr gleichem Abstand unter das jeweils darüber liegende Steuerelement setzen
  - und dieselbe Breite wählen
- Bei den **Button**-Objekten helfen die Führungslinien dabei, eine geeignete Position zu finden:

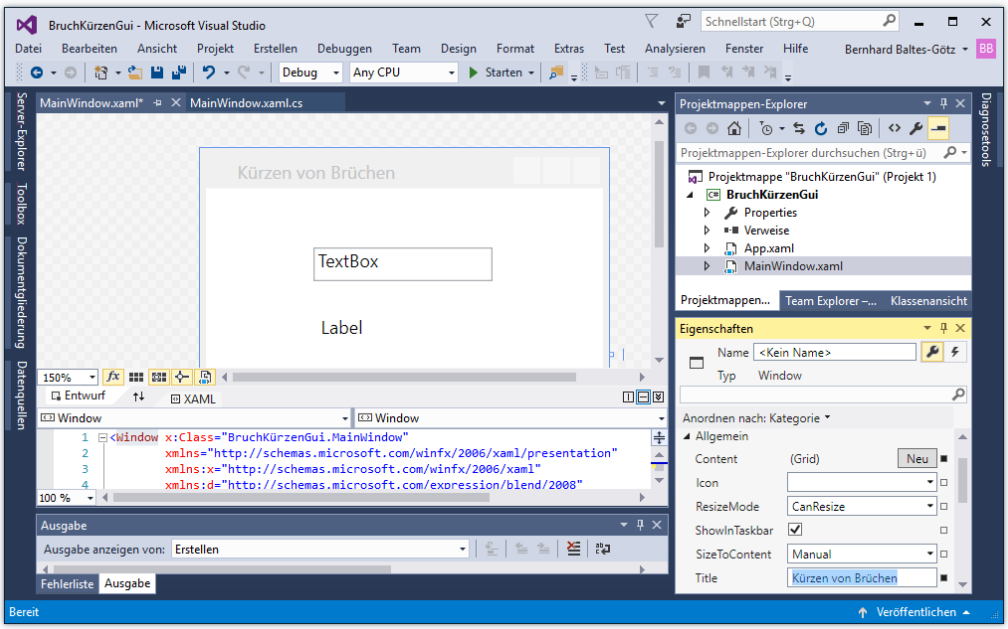


4.11.5 Eigenschaften der Steuerelemente ändern

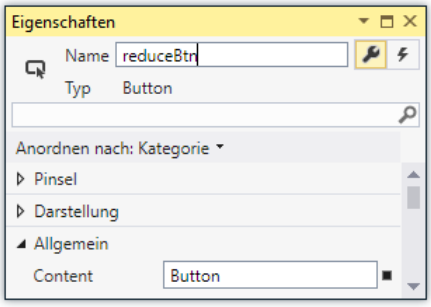
Im Eigenschaftenfenster der Entwicklungsumgebung, das bei Bedarf über das mit der Funktionstaste **F4** oder mit dem Menübefehl

Ansicht > Eigenschaftenfenster

zu öffnen ist, lassen sich diverse Eigenschaften (im Sinne von Abschnitt 4.5!) der markierten Objekte festlegen, z.B. der **Title** des Fensters:



Neben den Eigenschaftsmodifikationen ist insbesondere die Möglichkeit von Relevanz, den Instanzvariablennamen eines Steuerelements zu ändern, z.B.:



Wir ändern …

- die Namen den Instanzvariablen zu den **TextBox**- und den **Button**-Objekten auf `numTb`, `denomTb`, `reduceBtn` und `closeBtn`
- die **Content**-Eigenschaft der Schaltflächen, so dass sinnvolle Beschriftungen entstehen (z.B. Kürzen, Beenden),
- die **Content**-Eigenschaft des Labels, um einen behelfsmäßigen Bruchstrich zu erzielen,
- für beide **TextBox**-Objekte die Eigenschaft **Text** auf eine leere Zeichenfolge und die Eigenschaft **TextAlignment** auf den Wert **Center**,
- für das **Label**-Objekt die Eigenschaft **HorizontalContentAlignment** auf den Wert **Center**,
- die **IsDefault**-Eigenschaft der linken Schaltfläche auf den Wert **True**, so dass diese Schaltfläche im laufenden Programm per **Enter**-Taste angesprochen werden kann.[1]

Beim Verschieben eines Steuerelements im WPF-Designer (siehe Abschnitt 4.11.4) haben wir übrigens seine **Margin**-Eigenschaft verändert und so für jede Seite einen Randabstand festlegt.

Zum Markieren eines Steuerelements (im WPF-Designer und im XAML-Code) stehen folgende Techniken bereit:

- Mausklick auf das Steuerelement im WPF-Designer oder im XAML-Fenster
- Wenn der Designer den Tastaturfokus besitzt, kann man die Reihe der Steuerelemente per Tabulator-Taste vorwärts und bei zusätzlich gedrückter **Umschalt**-Taste rückwärts durchlaufen.

Um *mehrere* Steuerelemente zu markieren, kann man ...

- im WPF-Designer ein Markierungsrechteck um die gewünschten Teilnehmer ziehen,
- im WPF-Designer bei gedrückter **Strg**-Taste die Steuerelemente nacheinander anklicken

Die Eigenschaften von mehreren, gleichzeitig markierten Steuerelementen lassen sich in *einem* Arbeitsgang ändern.

Zwar ist eine Eigenschaftsmodifikation zur *Entwurfszeit* besonders bequem per Eigenschaftenfenster zu bewerkstelligen, doch muss man trotzdem wissen, wie der Eigenschaftszugriff per C# - Anweisung erfolgt, weil viele Steuerelementeigenschaften auch zur *Laufzeit* (dynamisch) geändert werden müssen.

---

[1] Anders als in C# wird das boolesche Literal **True** in XAML groß geschrieben, z.B.:

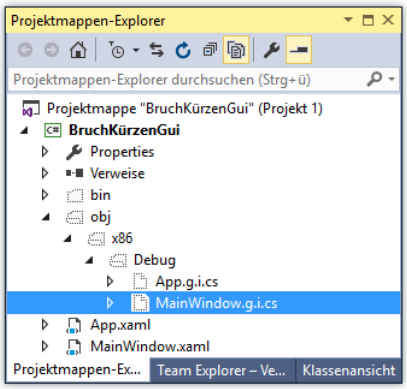
```
<Button x:Name="reduceBtn" Content="Kürzen" ... IsDefault="True" Click="reduceBtn_Click"/>
```

4.11.6 Automatisch erstellter und gepflegter Quellcode

Aufgrund Ihrer kreativen Tätigkeit beim GUI-Design erzeugt die Entwicklungsumgebung im Hintergrund Quellcode zu einer neuen Klasse namens `MainWindow`, die von der Klasse **Window** im Namensraum **System.Windows** abstammt. Aber auch *Sie* werden signifikanten Quellcode zu dieser Klasse beisteuern. Damit sich die beiden Autoren nicht in die Quere kommen, wird der Quellcode der Klasse `MainWindow` auf zwei Dateien verteilt:

- **MainWindow.xaml.cs** im Projektordner
  Hier werden die von Ihnen erstellten Methoden landen (siehe unten).
- **MainWindow.g.i.cs** im Projektunterordner **...\obj\Debug**[1]
  Hier landet der durch Interpretieren der XAML-Datei **MainWindow.xaml** erstellte C# - Quellcode. In dieser Datei dürfen Sie keine Änderungen vornehmen, um die Entwicklungsumgebung nicht aus dem Tritt zu bringen. Daran erinnert der Namensbestandteil *g* für *generated*.[2]

Wer die Datei **MainWindow.g.i.cs** lokalisieren und öffnen möchte, kann den Projektmappen-Explorer über den Symbolschalter dazu überreden, **alle Dateien anzuzeigen**, z.B.:



Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist, z.B. in der Quellcodedatei **MainWindow.xaml.cs**:

```
using System;
using System.Collections.Generic;
   .  .  .
using System.Windows.Shapes;

namespace BruchKürzenGui {
   /// <summary>
   /// Interaktionslogik für MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }
   }
}
```

Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, welcher die Methode **InitializeComponent()** aufruft. Diese wird in der Datei **MainWindow.g.i.cs** von der Entwicklungsumgebung implementiert.

---

[1] Das gilt bei Verwendung der Zielplattform **Any CPU**.

[2] Warum es neben **MainWindow.g.i.cs** im selben Ordner noch eine meist identische Datei mit dem Namen **MainWindow.g.cs** gibt, müssen wir nicht erforschen.

Aufgrund Ihrer Tätigkeit im Formulardesigner enthält die Fensterklasse `MainWindow` im Sinne der in Abschnitt 4.8 behandelten Komposition mehrere Objekte anderer Klassen, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.i.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen:

```
internal System.Windows.Controls.TextBox numTb;
internal System.Windows.Controls.TextBox denomTb;
internal System.Windows.Controls.Label label;
internal System.Windows.Controls.Button reduceBtn;
internal System.Windows.Controls.Button closeBtn;
```

Neben der Klasse `MainWindow` mit der XAML-Datei **MainWindow.xaml** und dem C# - Quellcode-Duo **MainWindow.xaml.cs** und **MainWindow.g.i.cs** enthält das Projekt noch die Klasse `App` mit der XAML-Datei **App.xaml** und dem C# - Quellcode-Duo **App.xaml.cs** und **App.g.i.cs**. Wer die **Main()** - Methode zu unserer GUI-Anwendung vermisst, findet sie in **App.g.i.cs**:

```
public static void Main() {
    BruchKürzenGui.App app = new BruchKürzenGui.App();
    app.InitializeComponent();
    app.Run();
}
```

Weitere Details zu den vom Visual Studio bei WPF-Projekten gepflegten Quellcodedateien folgen in Abschnitt 11.3.4.

4.11.7 Assembly mit der Bruch-Klasse einbinden

Gehen Sie folgendermaßen vor, um Objekte unserer Beispielklasse `Bruch` im neuen Programm nutzen zu können:

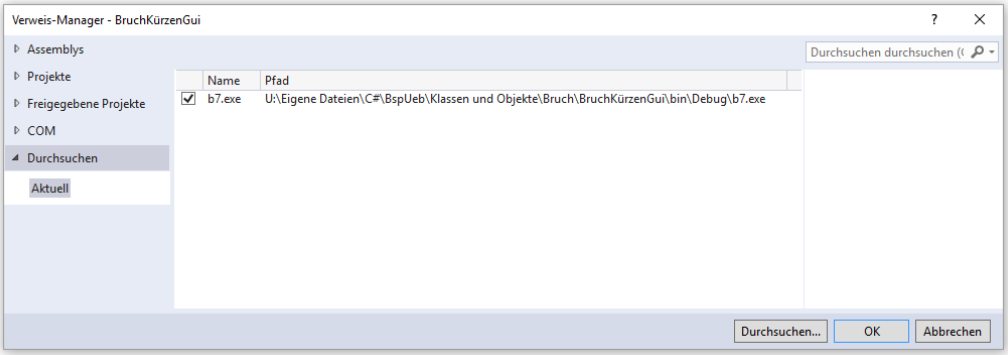
- Kopieren Sie die Assembly-Datei mit der Klasse `Bruch`, z.B.

  …\BspUeb\Klassen und Objekte\Bruch\b7 Operatoren-Überladung\bin\Debug\b7.exe

  in den Debug-Ausgabeordner Ordner des neuen Projekts, also z.B. nach

  U:\Eigene Dateien\C#\BspUeb\Klassen und Objekte\Bruch\BruchKürzenGui\bin\Debug

  In der Praxis werden sich allgemein benötigte Assembly-Dateien an einem sinnvollen Ort befinden (z.B. im Global Assembly Cache, GAC, siehe Abschnitt 1.2.5.4), so dass sie zur Nutzung in einem konkreten Projekt nicht kopiert werden müssen.
- Nehmen Sie per Projektmappen-Explorer die Assembly-Datei **Bruch.exe** in die Verweisliste des Projekts auf (vgl. Abschnitt 2.2.6.1), z.B.:



Die Klasse `Bruch` befindet sich im **globalen Namensraum**, weil wir bei ihrer Definition auf eine **namespace**-Definition verzichtet haben. Folglich ist beim Zugriff (im Unterschied zu den FCL-Klassen) kein Namensraum anzugeben. Wegen der Gefahr von Namenskollisionen ist dieses Verfahren *nicht* empfehlenswert. Unsere Entwicklungsumgebung definiert grundsätzlich einen Na-

mensraum (abgesehen von der *leeren* Projektvorlage) unter Verwendung des Projektnamens, so auch im aktuellen Beispiel:

```
namespace BruchKürzenGui {
   . . .
}
```

4.11.8 Ereignisbehandlungsmethoden anlegen

Zunächst erstellen wir zu den beiden Befehlsschaltern jeweils eine Methode, die durch das Betätigen des Schalters (z.B. per Mausklick) ausgelöst werden soll. Setzen Sie im Formulardesigner einen Doppelklick auf den Befehlsschalter `reduceBtn` (mit der Beschriftung *Kürzen*), so dass die Entwicklungsumgebung in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `reduceBtn_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
private void reduceBtn_Click(object sender, RoutedEventArgs e) {

}
```

und die Datei im Editor öffnet.

Mit Hilfe eines Objekts aus unserer Klasse `Bruch` ist die benötigte Funktionalität leicht zu implementieren, z.B.:

```
private void reduceBtn_Click(object sender, RoutedEventArgs e) {
    var b = new Bruch();
    try {
        b.Zaehler = Convert.ToInt32(numTb.Text);
        b.Nenner = Convert.ToInt32(denomTb.Text);
        b.Kuerze();
        numTb.Text = b.Zaehler.ToString();
        denomTb.Text = b.Nenner.ToString();
    }
    catch {
        MessageBox.Show("Eingabefehler", "Fehler", MessageBoxButton.OK,
                        MessageBoxImage.Error);
    }
}
```

Der Bequemlichkeit halber wird bei jedem Aufruf von `reduceBtn_Click()` ein neues Objekt erzeugt (vgl. Übungsaufgabe in Abschnitt 4.12).

Tritt im **try**-Block der **try-catch** - Anweisung eine Ausnahme auf (z.B. beim Versuch, irreguläre Benutzereingaben zu konvertieren), dann wird der **catch**-Block ausgeführt, und es erscheint eine Fehlermeldung. Im Aufruf der **MessageBox**-Methode **Show**() sorgen Werte der Enumerationen (siehe unten) **MessageBoxButton** bzw. **MessageBoxImage** als Aktualparameter für die gewünschte Ausstattung der Meldungsdialogbox mit Schaltflächen und einem Symbol. Bei einem gelungenen Ablauf wandern Informationen zwischen den **Text**-Eigenschaften der beiden **TextBox**-Objekte (Datentyp **String**) und den `Bruch`-Instanzvariablen `zaehler` und `nenner` (Datentyp: **int**) hin und her.

Erstellen Sie nun per Doppelklick auf den Befehlsschalter `closeBtn` (mit der Beschriftung *Beenden*) den Rohling für seine Klickereignisbehandlungsmethode, und ergänzen Sie einen Aufruf der **Window**-Methode **Close()**, die das Hauptfenster schließt und damit das Programm beendet, z.B.:

```
private void closeBtn_Click(object sender, RoutedEventArgs e) {
    this.Close();
}
```

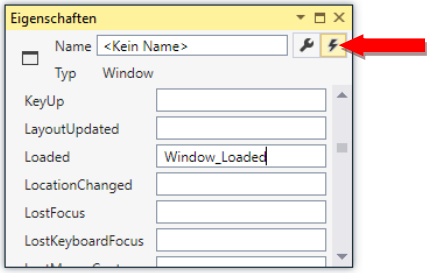
Es wäre nett, wenn nach dem Start unseres Programms das Zähler-Textfeld den Tastaturfokus hätte, so dass der Benutzer unmittelbar mit der Werteingabe beginnen könnte, ohne zuvor den Tastaturfo-

kus (z.B. per Maus) setzen zu müssen. Eine Lösungsmöglichkeit besteht darin, die statische Methode **Focus()** der Klasse **Keyboard** aufzurufen, die als Argument eine Referenz auf das zu privilegierende Objekt erwartet, z.B.:

```
Keyboard.Focus(numTb);
```

Damit diese Anweisung beim Laden des Anwendungsfensters ausgeführt wird, stecken wir sie in eine Ereignisbehandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

- Markieren Sie im WPF-Designer das Anwendungsfenster.
- Wechseln Sie im Eigenschaftenfenster zur Registerkarte **Ereignisse**.



- Setzen Sie einen Doppelklick auf das Texteingabefeld zum Ereignis **Loaded**. Wenn das Anwendungsfenster das Ereignis **Loaded** feuert, ist der richtige Moment für unsere Initialisierungsarbeiten gekommen.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:

  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {

  }
  ```

  Außerdem wird die Methode im Eigenschaftenfenster dem Ereignis **Loaded** zugeordnet.
- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()** - Aufruf.

Veranlassen Sie mit der Tastenkombination **Strg+F5** das Übersetzen und die Ausführung der fertigen Anwendung.

Die Assembly-Datei **BruchKürzenGui.exe** findet sich im Projektunterordner **…\bin\Debug**, weil die Erstellungskonfiguration Debug verwendet wurde (vgl. Abschnitt 2.2.4.4.

Zum Installieren des Programms genügt es, den Programmordner **…\bin\Debug** zu kopieren. Dabei darf die dort befindliche Assembly-Datei **b7.exe** mit der Klasse `Bruch` nicht vergessen werden.

4.12 Übungsaufgaben zu Kapitel 4

1) Welche von den folgenden Aussagen sind richtig?

1. Alle Instanzvariablen einer Klasse müssen von elementarem Typ sein.
2. In einer Klasse können mehrere Methoden mit demselben Namen existieren.
3. Bei der Definition eines Konstruktors ist der Rückgabetyp **void** anzugeben.
4. Mit der Datenkapselung wird verhindert, dass ein Objekt auf die Instanzvariablen anderer Objekte derselben Klasse zugreifen kann.
5. Als Wertaktualparameter sind nicht nur Variablen erlaubt, sondern beliebige Ausdrücke mit kompatiblem (erweiternd konvertierbarem Typ).
6. Ändert man den Rückgabetyp einer Methode, dann ändert sich auch ihre Signatur.

2) Erläutern Sie den Unterschied zwischen einem **readonly** - deklarierten Feld und einer Eigenschaft ohne **set** - Implementierung, die man auch als *getonly* - Eigenschaft bezeichnen könnte.

3) Im folgenden Programm soll die statische `Bruch`-Eigenschaft `Anzahl` ausgelesen werden:

```
using System;
class Bruchrechnung {
  static void Main() {
    var b1 = new Bruch(), b2 = new Bruch();
    Console.WriteLine("Jetzt sind wir " + Bruch.Anzahl);
  }
}
```

Es liegt folgende Eigenschaftsdefinition zugrunde:

```
public static int Anzahl {
  get {
    return Anzahl;
  }
}
```

Statt der erwarteten Auskunft:

```
Jetzt sind wir 2
```

erhält man jedoch (beim Programmstart im Konsolenfenster) die Fehlermeldung:

```
Process is terminated due to StackOverflowException.
```

Offenbar hat sich ein Fehler in die Eigenschaftsdefinition eingeschlichen, den der Compiler nicht bemerkt.

4) Die folgende Aufgabe eignet sich nur für Leser mit Grundkenntnissen in linearer Algebra: Erstellen Sie eine Klasse für Vektoren im $\mathbb{R}^2$, die mindestens über Methoden bzw. Eigenschaften mit folgenden Leistungen verfügt:

- **Länge** ermitteln

  Der Betrag eines Vektors $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ ist definiert durch:

  $$|x| := \sqrt{x_1^2 + x_2^2}$$

  Verwenden Sie die Klassenmethode **Math.Sqrt()**, um die Quadratwurzel aus einer **double**-Zahl zu berechnen.

- Vektor auf Länge Eins **normieren**
  Dazu dividiert man beide Komponenten durch die Länge des Vektors, denn mit

  $\tilde{x} := (\tilde{x}_1, \tilde{x}_2)$ sowie $\tilde{x}_1 := \frac{x_1}{\sqrt{x_1^2 + x_2^2}}$ und $\tilde{x}_2 := \frac{x_2}{\sqrt{x_1^2 + x_2^2}}$ gilt:

  $$|\tilde{x}| = \sqrt{\tilde{x}_1^2 + \tilde{x}_2^2} = \sqrt{\left(\frac{x_1}{\sqrt{x_1^2 + x_2^2}}\right)^2 + \left(\frac{x_2}{\sqrt{x_1^2 + x_2^2}}\right)^2} = \sqrt{\frac{x_1^2}{x_1^2 + x_2^2} + \frac{x_2^2}{x_1^2 + x_2^2}} = 1$$

- Vektoren (komponentenweise) **addieren**

  Die Summe der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$\begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \end{pmatrix}$$

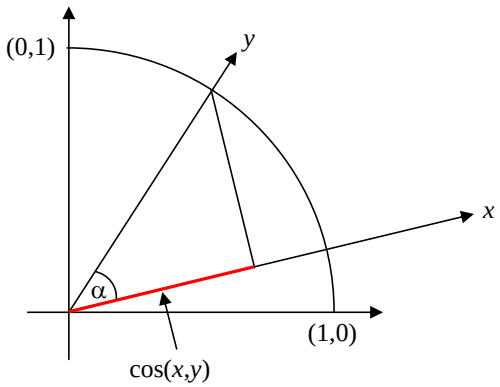
- **Skalarprodukt** zweier Vektoren ermitteln

  Das Skalarprodukt der Vektoren $x = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$ und $y = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix}$ ist definiert durch:

  $$x \cdot y := x_1 y_1 + x_2 y_2$$

- **Winkel** zwischen zwei Vektoren in Grad ermitteln
  Für den Kosinus des Winkels, den zwei Vektoren *x* und *y* im mathematischen Sinn (links herum) einschließen, gilt:[1]

  $$\cos(x, y) = \frac{x \cdot y}{|x||y|}$$

  Um aus cos(*x*, *y*) den Winkel α in Grad zu ermitteln, können Sie folgendermaßen vorgehen:
  - mit der Klassenmethode **Math.Acos()** den Winkel im Bogenmaß ermitteln
  - das Bogenmaß (*rad*) nach folgender Formel in Grad umrechnen (*deg*):

    $$deg = \frac{rad}{2\pi} \cdot 360$$

- **Rotation** eines Vektors um einen bestimmten Winkelgrad
  Mit Hilfe der Rotationsmatrix

  $$D := \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

  kann der Vektor *x* um den Winkel α (im Bogenmaß!) gedreht werden:

  $$x' = D\,x = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} \cos(\alpha)\,x_1 - \sin(\alpha)\,x_2 \\ \sin(\alpha)\,x_1 + \cos(\alpha)\,x_2 \end{pmatrix}$$

  Zur Berechnung der trigonometrischen Funktionen stehen die Klassenmethoden **Math.Cos()** und **Math.Sin()** bereit. Winkelgrade (*deg*) müssen nach folgender Formel in das von **Cos()** und **Sin()** benötigte Bogenmaß (*rad*) umgerechnet werden:

  $$rad = \frac{deg}{360} \cdot 2\pi$$

Erstellen Sie ein Demonstrationsprogramm, das Ihre Vektorklasse verwendet und ungefähr den folgenden Programmablauf ermöglicht (Eingabe fett):

---

[1] Dies folgt aus dem Additionstheorem für den Kosinus.

```
Vektor 1:               ( 1,00;  0,00)
Vektor 2:               ( 1,00;  1,00)

Länge von Vektor 1:      1,00

Länge von Vektor 2:      1,41

Winkel:                 45,00 Grad

Um wie viel Grad soll Vektor 2 gedreht werden: 45

Neuer Vektor 2          ( 0,00;  1,41)
Neuer Vektor 2 normiert ( 0,00;  1,00)

Summe der Vektoren      ( 1,00;  1,00)
```

5) Erstellen Sie eine Klasse mit einer statischen Methode zur Berechnung der Fakultät über einen rekursiven Algorithmus. Erstellen Sie eine Testklasse, welche die rekursive Fakultätsmethode benutzt. Diese Aufgabe dient dazu, an einem einfachen Beispiel mit rekursiven Methodenaufrufen zu experimentieren. Für die Praxis ist die rekursive Fakultätsberechnung weniger geeignet.

6) Ersetzen Sie beim GUI-Bruchkürzungsprogramm in Abschnitt 4.11 die lokale `Bruch`-Referenzvariable in der Klick-Ereignisbehandlungsmethode zum Befehlsschalter `reduceBtn` (mit der Beschriftung *Kürzen*) durch eine Instanzvariable der Hauptfensterklasse `MainWindow`. So wird vermieden, dass bei jedem Methodenaufruf ein neues `Bruch`-Objekt entsteht, das nach Beenden der Methode dem Garbage Collector überlassen wird.

7) Lokalisieren Sie bitte im folgenden Quellcode mit einer Kurzform der Klasse `Bruch`

```
using System;
public class Bruch {
   int zaehler, nenner = 1;                                          1
   string etikett = "";
   static int anzahl;                                                2

   public Bruch(int zpar, int npar, String epar) {
      Zaehler = zpar; Nenner = npar;
      Etikett = epar; Anzahl++;
   }
   public Bruch() {Anzahl++;}                                        3

   public int Zaehler {  .  .  .  }
   public int Nenner {                                               4
      get {return nenner;}
      set {
         if (value != 0)
            Nenner = value;
      }
   }
   public string Etikett {  .  .  .  }                               5

   public void Addiere(Bruch b) {
      Zaehler = zaehler*b.nenner + b.zaehler*nenner;                 6
      Nenner = nenner*b.nenner;
      Kuerze();
   }
   public Bruch Klone() {                                            7
      return new Bruch(zaehler, nenner, etikett);
   }
   public void Kuerze() {  .  .  .  }
   public bool Frage() {  .  .  .  }
   public void Zeige() {
      string luecke = "";                                            8
      for (int i=1; i <= etikett.Length; i++)
         luecke = luecke + " ";
      Console.WriteLine(" {0}   {1}\n {2} -----\n {0}   {3}\n",
                        luecke, zaehler, etikett, nenner);
   }
   public static Bruch operator+ (Bruch b1, Bruch b2) {              9
      Bruch temp = new Bruch(b1.zaehler * b2.nenner + b1.nenner * b2.zaehler,
         b1.nenner * b2.nenner, "");
      temp.Kuerze();
      return temp;
   }
   public static Bruch BenDef(string e) {                            10
      Bruch b = new Bruch(0, 1, e);                                  11
      if (b.Frage()) {
         b.Kuerze();
         return b;
      } else
         return null;
   }
   public static int Anzahl {                                        12
      get {return anzahl;}
      private set {anzahl = value;}
   }
}
```

12 Begriffe der objektorientierten Programmierung, und tragen Sie die Positionen in die folgende Tabelle ein

Begriff	Pos.
Definition einer Instanzmethode mit Referenzrückgabe	
Deklaration lokale Variable	
Def. einer Instanzmeth. mit Wert-parameter vom Typ einer Klasse	
Deklaration von Instanzvariablen	
Methodenaufruf	
Definition einer statischen Eigenschaft	

Begriff	Pos.
Konstruktordefinition	
Deklaration einer Klassenvariablen	
Objekterzeugung	
Definitionskopf einer Klassenmethode	
Definition einer Instanzeigenschaft	
Operatorüberladung	

Zum Eintragen benötigen Sie nicht unbedingt eine gedruckte Variante des Manuskripts, sondern können auch das interaktive PDF-Formular

...\BspUeb\Klassen und Objekte\Bruch\Begriffe lokalisieren.pdf

benutzen. Die Idee zu dieser Übungsaufgabe stammt aus Mössenböck (2003).

5 Weitere .NETte Typen

Nachdem wir uns ausführlich mit elementaren Datentypen und mit Klassen beschäftigt haben, wird in diesem Abschnitt Ihr Wissen über das *Common Type System* (CTS) der .NET - Plattform abgerundet. Sie lernen u.a. die folgenden Datentypen kennen:

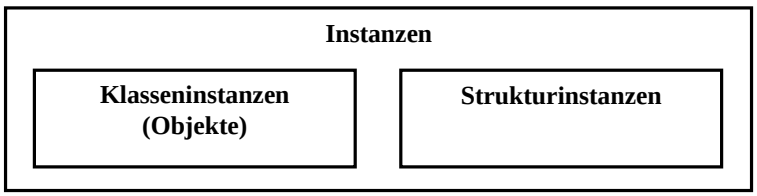
- Strukturen als Klassenalternative mit Wertsemantik
- Arrays als Container für eine feste Anzahl von Elementen desselben Datentyps
- Klassen zur Verwaltung von Zeichenketten (**String**, **StringBuilder**)
- Aufzählungstypen (Enumerationen)

5.1 Strukturen

Klassen und Objekte haben ohne Zweifel einen sehr hohen Nutzen, verursachen aber auch Kosten, z.B. beim Erzeugen von Objekten, bei der Referenzverwaltung und bei der Entsorgung per Garbage Collector. Daher stellt das .NET - Framework mit den *Strukturen* auch *Werttypen* zur Verfügung, die in manchen Situationen bei geringerem Ressourcen-Verbrauch eine „echte“ Klasse ersetzen und somit die Performanz der Software steigern können.

Beim Design eines Strukturtyps können im Wesentlichen dieselben Member eingesetzt werden wie bei einer Klassendefinition (Felder beliebigen Typs, Methoden, Eigenschaften, usw.).[1] Eine Variable vom Typ einer Struktur enthält jedoch keine *Referenz* auf ein Heap-Objekt, sondern die *Daten* ihres Typs. Wie bei Variablen mit einem elementaren Datentyp liegt keine Referenz-, sondern eine **Wertsemantik** vor. Bei einer Zuweisung wird keine Referenz übergeben, sondern der komplette Wert kopiert.

Für ein Individuum nach dem Bauplan einer Struktur soll *nicht* der Begriff *Objekt*, sondern der allgemeinere Begriff *Instanz* verwendet werden.



Von *Instanzvariablen* und *-methoden* sprechen wir bei Objekten *und* bei Strukturinstanzen.

Eine Struktur eignet sich als Datentyp bei folgender Konstellation:

- Kleine Instanzen
  Der Typ ist relativ einfach aufgebaut, hat also nur wenige Instanzvariablen. Microsoft empfiehlt für Strukturinstanzen eine Größe von weniger als 16 Bytes.[2]
- Wertsemantik erwünscht
- Unveränderliche Instanzen
  Es wird überwiegend empfohlen, Strukturinstanzen als **unveränderlich** (engl. *immutable*) zu konzipieren, so z.B. auf einer MSDN-Webseite (*Microsoft Developer Network*) von Microsoft:[3]

  DO NOT define mutable value types.

  Als Begründung für diese Empfehlung wird angeführt, dass es bei einer Struktur aufgrund der Wertsemantik leicht passieren könne, dass man lediglich eine *Kopie* modifiziert an Stelle

---

1 Es ist allerdings kein Destruktor erlaubt.
2 URL: http://msdn.microsoft.com/de-de/library/y23b5415(en-us).aspx
3 https://msdn.microsoft.com/de-de/library/ms229031(v=vs.110).aspx

der eigentlich zu ändernden Instanz (Griffiths 2013, S. 97). Um die Unveränderlichkeit eines Typs zu erreichen, muss man die Datenkapselung realisieren und außerdem auf Methoden und Eigenschaften verzichten, die Instanzvariablen ändern (Richter 2006, S. 166).

- Nullinitialisierung akzeptabel
  Bei einer Struktur muss sichergestellt sein, dass die Nullinitialisierung aller Felder zu einer regulären Instanz führt (siehe unten).
- Keine Vererbung erforderlich
  Bei Strukturen fehlt die Möglichkeit, per Vererbung (siehe Kapitel 6) eine Hierarchie spezialisierter Typen aufzubauen. Das Implementieren von Schnittstellen (siehe Kapitel 8) ist aber möglich.

Werden von einem Typ viele Instanzen benötigt, kann sich die Vermeidung von Objektkreationen durch Verwendung eines Strukturtyps lohnen. Typische Anwendungsbeispiele für Strukturen:

- Punkte in einem zweidimensionalen Zahlenraum
  Ein Beispiel ist die von Microsoft definierte FCL-Struktur **Point** im Namensraum **System.Windows**, die öffentliche Eigenschaften (inkl. **set**-Methode) für die Koordinaten besitzt und folglich veränderlich ist.
- Komplexe Zahlen[1]
  Bei der Struktur **Complex** im Namensraum **System.Numerics** hat sich Microsoft an die eigene Empfehlung gehalten, Strukturen als unveränderlich zu definieren. Hier besitzen z.B. die öffentlichen Eigenschaften **Real** und **Imaginary** für den Zugriff auf den Real- bzw. Imaginärteil einer komplexen Zahl nur eine **get**-Methode.

Empfehlungen aus Griffiths (2013, S. 97) für die Entscheidung zwischen einer Klasse und einer Struktur bei der Realisation eines neuen Datentyps:

- Wenn es für eine Instanz erforderlich ist, ihren Zustand zu ändern, so ist dies ein deutliches Indiz dafür, dass eine Klasse gegenüber einer Struktur bevorzugt werden sollte.
- Im Zweifel sollte man eine Klasse definieren.

Wie das Beispiel der außerordentlich wichtigen Klasse **String** zeigt (siehe Abschnitt 5.4.1), ist bei einem Typ mit unveränderlichen Instanzen noch lange nicht entschieden, dass eine Struktur definiert werden sollte.

5.1.1 Detailvergleich von Klassen und Strukturen

Um den gravierenden Unterscheid zwischen der *Referenzsemantik* der Klassen und der *Wertsemantik* der Strukturen zu demonstrieren, definieren wir sowohl eine *Klasse* als auch eine *Struktur* zur Repräsentation von Punkten der reellen Zahlenebene ($\mathbb{R}^2$). Für die beiden Koordinaten eines Punkts werden Felder mit dem elementaren Datentyp **double** verwendet.

Eine Strukturdefinition unterscheidet sich von der gewohnten Klassendefinition auf den ersten Blick nur durch das neue Schlüsselwort **struct**, das an Stelle von **class** verwendet wird:

[1] Dieser mathematische Begriff meint Paare aus reellen Zahlen, für die spezielle Rechenregeln gelten. Wer nicht mathematisch vorbelastet ist, kann das Beispiel ignorieren.

```
public struct Punkt {
   double x, y;

   public Punkt(double x_, double y_) {
      x = x_;
      y = y_;
   }

   public static void Bewegen(Punkt p, double hor, double vert) {
      p.x = p.x + hor;
      p.y = p.y + vert;
   }

   public void Bewegen(double hor, double vert) {
      x = x + hor;
      y = y + vert;
   }

   public string Pos() {
      return "("+x+";"+y+")";
   }
}
```

Wir *ignorieren* die Mahnung, Strukturtypen als unveränderlich zu konzipieren, und definieren zwei Methoden namens `Bewegen()`:

- Die statische Variante ist ungeschickt und kann wegen der Wertsemantik bei Strukturen die Position der im ersten Parameter benannten `Punkt`-Instanz *nicht* verändern. Um die Macke zu beseitigen, müsste man allerdings lediglich den ersten Parameter zum Referenzparameter machen (vgl. Abschnitt 4.3.1.3.2).
- Die Instanzmethode `Bewegen()` erfüllt ihren Zweck.

Wie die Objekte von Klassen werden auch Strukturinstanzen per **new**-Operator unter Verwendung eines Konstruktors erstellt. Beim folgenden Einsatz der `Punkt`-Struktur wird die Variable `p1` über den expliziten Konstruktor mit dem Wert (1, 2) initialisiert. Der Punkt `p2` erhält (vom *nicht* verlorenen gegangenen!) Standardkonstruktor eine Initialisierung auf den Wert (0, 0). Der Punkt `p3` erhält eine *Kopie* von `p1` (mit allen Instanzvariablen).

Quellcode	Ausgabe (mit `Punkt` als Struktur)
`using System;` `class PunktDemo {` `  static void Main() {` `  var p1 = new Punkt(1, 2);` `  var p2 = new Punkt();` `  var p3 = p1;` `  Punkt.Bewegen(p2, 1, 1); // nutzlos` `  p1.Bewegen(1, 0); // ohne Effekt auf p3` `  Console.WriteLine("p1 = " + p1.Pos()+` `  "\np2 = " + p2.Pos() +` `  "\np3 = " + p3.Pos());` `  }` `}`	`p1 = (2;2)` `p2 = (0;0)` `p3 = (1;2)`

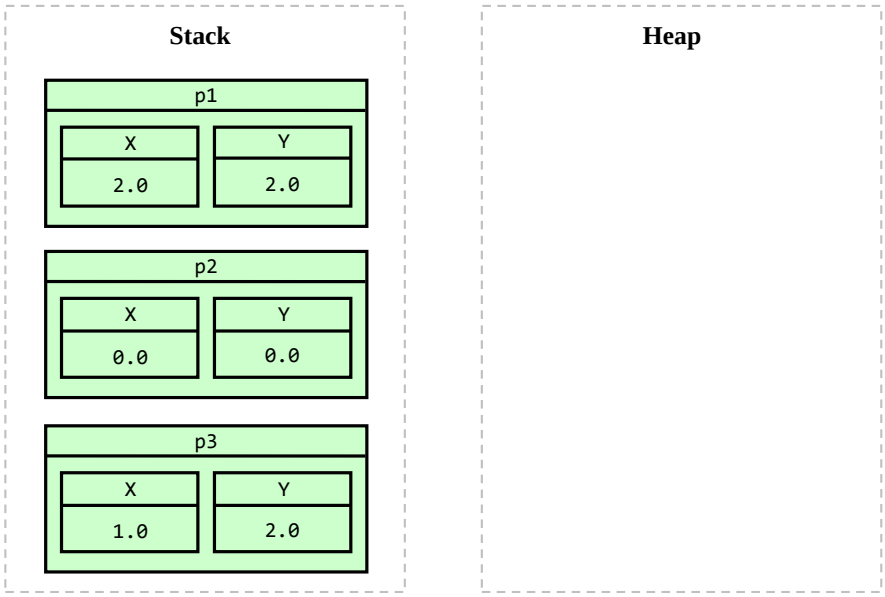
Die Wertsemantik von Strukturen ist im Verhalten des Programms vor allem an zwei Stellen zu beobachten:

- Mit der statischen `Bewegen()` - Überladung kann die Position des ersten Aktualparameters *nicht* verändert werden. Die fehlerhaft implementierte Methode ändert lediglich die per Wertparameter erhaltene Kopie.
- Die Änderung von `p1` bleibt ohne Effekt auf `p3`.

Nach der Anweisung

```
p1.Bewegen(1, 0);
```

haben wir folgende Situation im Speicher des Programms:



Aus der `Punkt`-Struktur wird durch wenige Quellcode-Änderungen eine *Klasse*:

```
public class Punkt {
   double x, y;

   public Punkt(double x_, double y_) {
      x = x_;
      y = y_;
   }

   public Punkt() { }

   public static void Bewegen(Punkt p, int hor, int vert) {
      p.x = p.x + hor;
      p.y = p.y + vert;
   }

   public void Bewegen(int hor, int vert) {
      x = x + hor;
      y = y + vert;
   }

   public string Pos() {
      return "("+x+";"+y+")";
   }
}
```

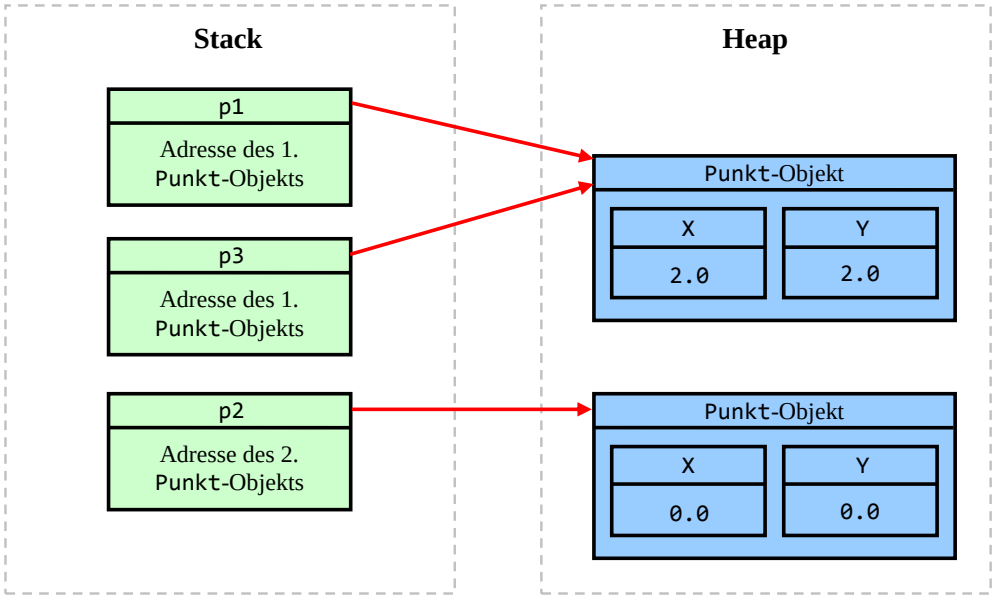
Weil bei Klassen (im Unterschied zu Strukturen) der Standardkonstruktor verloren geht, sobald ein expliziter Konstruktor vorhanden ist (vgl. Abschnitt 4.4.3), hat die Klasse `Punkt` einen parameterlosen Konstruktor erhalten.

Das obige **Main()** - Programm muss beim Wechsel von der `Punkt`-Struktur zur `Punkt`-Klasse *nicht* geändert werden, verhält sich aber anders:

Quellcode	Ausgabe (mit `Punkt` als Klasse)
`using System;` `class PunktDemo {` `  static void Main() {` `  var p1 = new Punkt(1, 2);` `  var p2 = new Punkt();` `  var p3 = p1;` `  Punkt.Bewegen(p2, 1, 1); // wirksam` `  p1.Bewegen(1, 0); // hat Effekt auf p3` `  Console.WriteLine("p1 = " + p1.Pos()+` `  "\np2 = " + p2.Pos() +` `  "\np3 = " + p3.Pos());` `  }` `}`	`p1 = (2;2)` `p2 = (1;1)` `p3 = (2;2)`

Die in der statischen Variante der Methode `Bewegen()` vorgenommenen Ortsveränderungen wirken sich auf das *Objekt* mit der im ersten Aktualparameter übergebenen Adresse aus.

`p1`, `p2` und `p2` sind nun lokale Referenzvariablen, die auf insgesamt *zwei* Objekte zeigen:



Das erste `Punkt`-Objekt kann über die Referenzvariablen `p1` *und* `p3` angesprochen (und z.B. verändert) werden.

Während ein Objekt *eigenständig* auf dem *Heap* existiert (persistiert) und verfügbar ist, solange irgendwo im Programm eine Referenz (Kommunikationsmöglichkeit) vorhanden ist, kann eine Strukturinstanz nur als Objekt-Member das Ende der erzeugenden Methode überstehen (so wie auch die Variablen mit elementarem Typ). Eine *lokale* Variable mit Strukturtyp wird auf dem *Stack* abgelegt und beim Verlassen der Methode gelöscht. Bei entsprechender Methodendefinition kann dem Aufrufer via Rückgabewert nur eine *Kopie* der Strukturinstanz übergeben werden.

Auch bei Verwendung einer Eigenschaft oder eines Indexers findet ein verkappter Methodenaufruf statt, und hier können sich Programmierfehler durch veränderliche Strukturen besonders leicht einschleichen. Im folgenden Beispielprogramm hat die Klasse `PunktDemo` eine Eigenschaft namens `Position` vom Typ `Punkt` erhalten:

```
class PunktDemo {
  public Punkt Position {get; set;} = new Punkt();
  static void Main() {
    var pd = new PunktDemo();
    pd.Position.Bewegen(1, 2);
    Console.WriteLine("Neue Position: " + pd.Position.Pos());
  }
}
```

Ist `Punkt` eine Klasse, kann die `Position` eines Objekts vom Typ `PunktDemo` per Eigenschaftszugriff geändert werden:

```
Neue Position: (1;2)
```

Ist `Punkt` hingegen eine Struktur, bleibt die Ausführung der Methode `Bewegen()` ohne Effekt:

```
Neue Position: (0;0)
```

Um solche Missverständnisse mit beliebig gravierenden Konsequenzen auszuschließen, sollten Strukturen als unveränderlich konzipiert werden.

Neben der eigenständigen Speicherpersistenz fehlt den Strukturen vor allem die Vererbungstechnik, die wir erst in einem späteren Kapitel gründlich behandeln werden. *Jede* Struktur stammt von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahnklasse **Object** erbt (siehe Abbildung in Abschnitt 5.1.2). Alternative Abstammungen sind bei Strukturen nicht möglich, so dass auch keine Strukturhierarchien entstehen können.

Eine Strukturdefinition unterscheidet sich nur geringfügig von einer Klassendefinition, so dass wir uns an Stelle von Syntaxdiagrammen auf einige Hinweise beschränken können:

- Wie bei Klassen wird die Verfügbarkeit eines Strukturtyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 4.10).
- Das Schlüsselwort **class** wird **struct** ersetzt.
- Bei der Deklaration von Instanzfeldern ist *keine explizite Initialisierung* erlaubt. Verboten ist also z.B.:

  ```
  double delta = 1.0;
  ```

  Felder von Strukturinstanzen werden jedoch wie die Felder von Klasseninstanzen (Objekten) per Voreinstellung mit der typspezifischen Null initialisiert.
- Es sind beliebige viele Konstruktoren erlaubt, wobei der (parameterfreie) Standardkonstruktor *nicht* verloren geht.
- Bei einer Struktur darf kein expliziter parameterloser Konstruktor definiert werden, so dass man das Initialisierungsverbot (siehe oben) nicht per Konstruktor aushebeln kann. Folglich muss bei einer Struktur sichergestellt sein, dass die Nullinitialisierung zu einer regulären Instanz führt. Die zeitsparend durchführbare Nullinitialisierung findet immer dann statt, wenn ein Objekt erzeugt wird, das Instanzvariablen mit einem Strukturtyp enthält, z.B. ein Array-Objekt mit 1000 Elementen.
- Auch bei Strukturen ist eine Datenkapselung möglich. Bei der Deklaration bzw. Definition der Member kann der Zugriffsschutz über die Modifikatoren **private** (Voreinstellung), **public** und **internal** reguliert werden. Weil keine Vererbung unterstützt wird, ist der Modifikatoren **protected** verboten.

- Bei Klassen sind Überladungen für die Operatoren == und **!=** vorhanden, die auf einem Vergleich von Speicheradressen basieren. Diese Operatorüberladungen müssen für Strukturen bei Bedarf explizit definiert werden (vgl. Abschnitt 4.7.2). Wenn dies geschieht, besteht der Compiler auf kompatiblen Überschreibungen der Methoden **Equals()** und **GetHashCode()** (siehe Richter 2006, S. 166ff für eine Erläuterung der beiden Methoden).
- Bei Strukturen ist keine Vererbung möglich, was einige zusätzliche Regeln zur Folge hat (siehe ECMA 2006, S. 333).
- Am Ende der Strukturdefinition darf optional ein Semikolon stehen.

In der Regel enthält eine Struktur nur Felder mit einem Werttyp (z.B. mit einem elementaren Datentyp). Felder mit Referenztyp sind ungewöhnlich, aber erlaubt, wobei die zugehörigen Objekte unveränderlich sein sollten, was z.B. bei **String**-Objekten der Fall ist (siehe Abschnitt 5.4.1).

Das folgende Programm vergleicht den Zeit- und Speicheraufwand für Objekte und Strukturinstanzen. Es wird jeweils ein Array (vgl. Abschnitt 5.3) mit 1.000.000 Punkten angelegt. Die Elemente werden initialisiert und im Fall der Objekte auch wieder durch einen expliziten Aufruf des Garbage Collectors aus dem Speicher entfernt:[1]

```
using System;

class Aufwandsvergleich {
   static void Main() {
        int anz = 1000000;

        // Zeitmessung vorbereiten
        long wzeit = DateTime.Now.Ticks;

        long st = DateTime.Now.Ticks;
        SPunkt[] arp = new SPunkt[anz];
        for (int i = 0; i < anz; i++) {
            arp[i].X = i;
            arp[i].Y = i;
        }
        GC.Collect();
        st = DateTime.Now.Ticks - st;
        Console.WriteLine("\nBenöt. Zeit für den Struktur-Array:\t" +
                          st / 1.0e4 + " Millisek.");

        long ct = DateTime.Now.Ticks;
        CPunkt[] arc = new CPunkt[anz];
        for (int i = 0; i < anz; i++) {
            arc[i] = new CPunkt();
            arc[i].X = i;
            arc[i].Y = i;
        }
        GC.Collect();
        ct = DateTime.Now.Ticks - ct;
        Console.WriteLine("\nBenöt. Zeit für den Klassen-Array:\t" +
                          ct / 1.0e4 + " Millisek.");
    }
}
```

Der Zeitaufwand ist für die Objekte ca. 7-mal höher als für die Strukturinstanzen:

```
Benöt. Zeit für den Struktur-Array:     16,0138 Millisek.
Benöt. Zeit für den Klassen-Array:      112,1021 Millisek.
```

[1] Zur besseren Vergleichbarkeit enthält auch die Zeitmessung für die Strukturinstanzen einen Aufruf des Garbage Collectors, der aber hier keinen Unterschied macht.

Um den Speicherbedarf beim isolierten Einsatz des Struktur- bzw. Klassen-Arrays ermitteln, wurden die Anweisungen zum jeweils anderen Datentyp ausgeblendet. Mit den im Debug-Betrieb (Start mit **F5**) aktiven Diagnosetools der Entwicklungsumgebung ergaben sich folgende Werte:

```
Speicherbedarf für den Struktur-Array:     30,7 MB
Speicherbedarf für den Klassen-Array:      56,6 MB
```

Diese Ergebnisse sind nicht verwunderlich, denn ein Array mit 1.000.000 Strukturinstanzen ergibt auf dem Heap ein einziges Objekt mit einem zusammenhängenden Speicherbereich. Ein Array mit 1.000.000 Objekten ergibt hingegen 1000001 Objekte auf dem Heap. Der Array enthält nur die Adressen der 1.000.000 separaten `Punkt`-Objekte. Die Speicherverwaltung ist viel aufwendiger, und der Garbage Collector ist auch gefragt, was Aufwand für die CLR verursacht.

Wir verzichten darauf, von der Klasse `Bruch` eine Strukturalternative zu erstellen, denn:

- Weil bei der Deklaration von Strukturfeldern keine Initialisierung erlaubt ist, und außerdem der parameterfreie Konstruktor nicht verändert werden darf, könnte bei parameterfrei konstruierten `Bruch`-Strukturinstanzen nicht verhindert werden, dass der Nenner den Wert 0 annimmt.
- Um die dringende Empfehlung der Unveränderlichkeit von Strukturinstanzen umzusetzen, müsste das Design der Klasse `Bruch` erheblich geändert werden. Es wäre z.B. nicht zulässig/akzeptabel, dass man eine `Bruch`-Strukturinstanz auffordern kann, sich zu kürzen.
- Es ist nicht damit zu rechnen, dass `Bruch`-Objekte derart massenhaft auftreten, dass ein Spareffekt durch die Verwendung von Strukturen spürbar wird.
- Wir würden die Möglichkeit verlieren, per Vererbung aus dem Typ `Bruch` spezialisierte Varianten abzuleiten.

5.1.2 Strukturen im allgemeinen Typsystem der .NET - Plattform

Aus den bisherigen Ausführungen zu folgern, dass Strukturen wohl eher exotisch und nur für leistungskritische Anwendungen interessant seien, wäre schon deshalb grundverkehrt, weil es sich bei allen elementaren Datentypen um Strukturen handelt. Die früher als Typbezeichner eingeführten reservierten Wörter sind lediglich Aliasnamen für Strukturen aus dem FCL-Namensraum **System**:

Aliasname	Struktur
sbyte	System.SByte
byte	System.Byte
short	System.Int16
ushort	System.UInt16
int	System.Int32
uint	System.UInt32

Aliasname	Struktur
long	System.Int64
ulong	System.UInt64
char	System.Char
float	System.Single
double	System.Double
bool	System.Boolean
decimal	System.Decimal

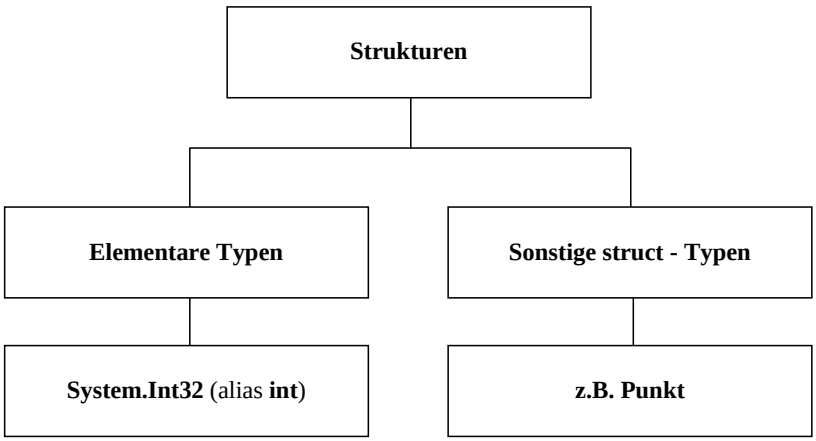
Nun wird z.B. klar, warum die **Convert**-Methode zur Wandlung von Zeichenfolgen in **int**-Werte den Namen **ToInt32()** trägt.

Für die elementaren Datentypen bietet C# im Vergleich zu den sonstigen Strukturtypen neben den reservierten Wörtern als Typaliasnamen noch weitere Vorzugsbehandlungen, z.B. die Erzeugung von Werten über Literale (vgl. ECMA 2006, S. 110). Wie z.B. die Definition der FCL-Typs **Int32** zeigt,[1]

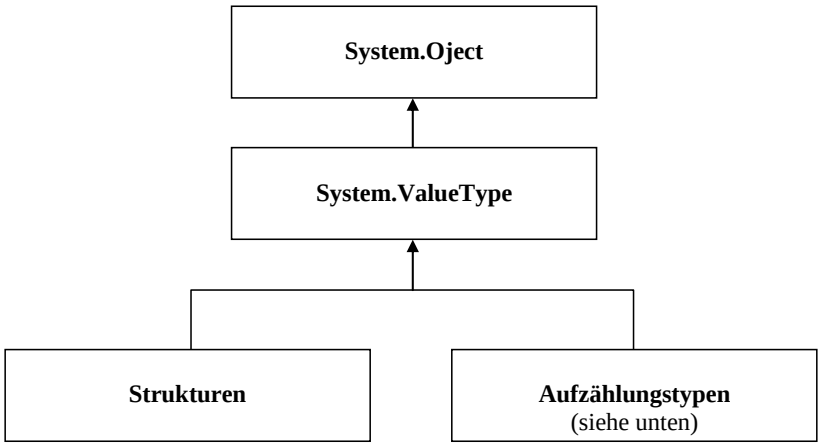
```
public struct Int32 : IComparable, IFormattable, IConvertible { ... }
```

[1] Hinter dem Doppelpunkt im Definitionskopf werden *Schnittstellen* aufgelistet, denen bald ein Kapitel gewidmet wird.

sind die elementaren Datentypen ansonsten reguläre Strukturen:



Jeder Strukturtyp stammt von der Klasse **ValueType** im Namensraum **System** ab, die wiederum direkt von der .NET - Urahnklasse **Object** erbt. Alle Strukturen stammen von der Klasse (!) **System.ValueType** ab, die wiederum direkt von Urahnklasse **System.Object** erbt:



Im folgenden Beispielprogramm wird beim Ganzzahlliteral 13 (vom Strukturtyp **Int32**) über die von **System.Object** geerbte Methode **GetType()** erfolgreich der Datentyp erfragt:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine(13.GetType());` `  }` `}`	`System.Int32`

Trotz des obigen Stammbaums *ist* eine Strukturinstanz kein Objekt, aber aufgrund einer im Nächsten Abschnitt zu beschreibender Raffinesse der .NET - Plattform kann eine Strukturinstanz jederzeit so behandelt werden, als *wäre* sie ein Objekt.

5.2 Boxing und Unboxing

Wie bald im Kapitel 6 über die Vererbung noch näher erläutert wird, kann eine Variable vom Typ **Object** Referenzen auf Objekte aus beliebigen Klassen aufnehmen, weil alle Klassen direkt oder indirekt von **Object** abstammen. Damit das *Common* Type System seinem Namen gerecht wird, muss diese Zuweisungskompatibilität für *beliebige Typen* gelten, also auch für Werttypen. Wie Sie wissen, kann eine Referenzvariable vom Typ **Object** nur die Adresse eines Heap-Objekts aufneh-

men. Was soll nun aber geschehen, wenn einer solchen Referenzvariablen z.B. ein Wert vom elementaren Typ **int** zugewiesen wird?

Eine analoge Situation liegt vor, wenn bei einem Methodenaufruf eine Strukturinstanz als Aktualparameter auftritt, obwohl syntaktisch (gemäß Methodendefinition) und damit auch technisch eine Objektreferenz (die Adresse eines Heap-Objekts) benötigt wird. Wir haben uns längst daran gewöhnt, dass der Compiler Werte von beliebigem Typ als **Object**-Instanzen akzeptiert. Z.B werden im folgenden Programm

```
using System;
class Prog {
  static void Main() {
    int i = 7, j = 3;
    Console.WriteLine("{0} % {1} = {2}", i, j, i % j);
  }
}
```

der Methode **WriteLine()** Aktualparameter vom Werttyp **int** an Positionen mit Formalparametertyp **Object** übergeben:

public static void WriteLine(String *format***, params Object[]** *arg***)**

Diese Zuweisungskompatibilität wird möglich durch ein als **Boxing** bezeichnetes Prinzip: Es sorgt dafür, dass ein Wert bei Bedarf automatisch in ein Objekt einer passenden Hilfs- bzw. Hüllenklasse verpackt wird. Somit existiert ein Heap-Objekt mit den Handlungskompetenzen der Klasse **Object**, und der betroffenen Programmeinheit (z.B. der **Console**-Methode **WriteLine()**) kann die benötigte Adresse übergeben werden. Zur Erläuterung der Boxing-Technik bedienen wir uns in Anlehnung an ECMA (2006, S. 114f) einer nicht ganz korrekten, aber hilfreichen Vorstellung: Zum Werttyp `T` nehmen wir die Existenz der folgenden Boxing-Klasse `TBox` an:

```
class TBox {
  public T Wert;
  public TBox(T w) {
    Wert = w;
  }
}
```

Beim automatischen Verpacken eines Werts `w` vom Typ `T` wird implizit über den Ausdruck

```
new TBox(w);
```

ein `TBox`-Objekt auf dem Heap erzeugt. Es nimmt eine *Kopie* des Wertes auf und besitzt alle **Object**-Handlungskompetenzen, kann z.B. die Methode **GetType()** ausführen.

Im folgenden Programm mit einer Boxing-Trockenübung wird die **int**-Variable `i` einer Referenzvariablen vom Typ **object** (Aliasname für **System.Object**) zugewiesen und dabei automatisch in ein Objekt der zugehörigen Hilfsklasse gesteckt:[1]

---

[1] Neben dem *impliziten* Boxing, das auch als **Autoboxing** bezeichnet wird, ist auch ein *explizites* Boxing möglich, aber nie erforderlich, z.B.

```
int i = 4711;
object iBox = (object) i;
```

```
using System;
class Prog {
  static void Main() {
    int i = 4711;
    object iBox = i;
    int j = (int) iBox;
  }
}
```

Dass die Boxing-Technik nicht nur akademische Trockenübungen ermöglicht, werden Sie z.B. im Zusammenhang mit Arrays und anderen Containern erfahren. Durch Verwendung des Basistyps **Object** erhält man „Gemischtwarenbehälter“, die auch Daten mit Werttyp aufnehmen können. Die erforderlichen Anpassungs- bzw. Verpackungsmaßnahmen laufen automatisch ab.

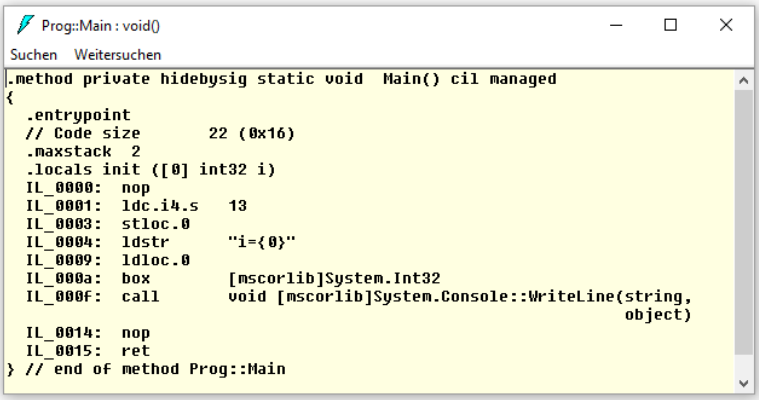
Wie das letzte Beispiel zeigt, benötigt der Compiler beim **Unboxing**, also beim Auspacken eines Wertes aus einem Hüllenobjekt, eine explizite Casting-Operation mit Angabe des Zieltyps:

```
int j = (int) iBox;
```

Weil das Boxing durch die damit verbundene Objektkreation relativ aufwändig ist, sollte man die Verwendung dieser Technik auf das notwendige Maß beschränken. Wer sich vergewissern möchte, dass beim Methodenaufruf

```
Console.WriteLine("i={0}", i);
```

mit der **int**-Variablen `i` als zweitem Aktualparameter tatsächlich eine Boxing-Operation stattfindet, kann mit dem Windows-SDK - Hilfsprogramm **ILDasm** einen Blick auf den CIL-Code werfen. Hier wird die Objektkreation zur Verpackung eines **System.Int32** - Werts mit dem OpCode **box** veranlasst:



Anschließend wird die Methode **Console.WriteLine()** mit Aktualparametern vom Typ **String** bzw. **Object** aufgerufen.

Auch zur Ausführung des **GetType()** - Aufrufs in der folgenden Anweisung

```
Console.WriteLine(13.GetType());
```

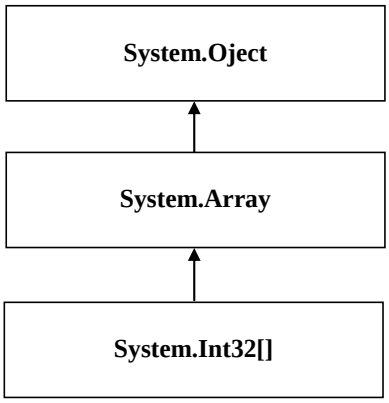
wird per **box**-OpCode ein Objekt erzeugt.

5.3 Arrays

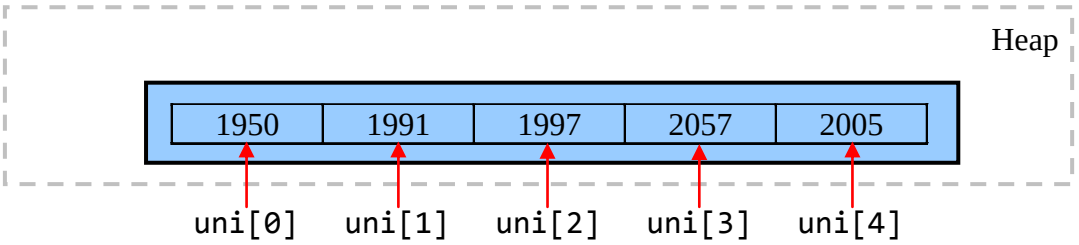
Ein Array ist ein Objekt, das als Instanzvariablen eine feste Anzahl von Elementen desselben Datentyps enthält. Man kann den kompletten Array ansprechen (z.B. als Aktualparameter an eine Methode übergeben), oder auf einzelne Elemente über einen Index zugreifen.

Arrays werden in vielen Programmiersprachen auch *Felder* genannt. In C# bezeichnet man jedoch recht einheitlich die (Instanz-)variablen einer Klasse oder Struktur als *Felder*, so dass der Name hier nicht mehr zur Verfügung steht.

Wir beschäftigen uns erst *jetzt* mit den zur Grundausstattung praktisch jeder Programmiersprache gehörenden Arrays, weil diese Datentypen in C# als *Klassen* realisiert sind und folglich zunächst entsprechende Grundlagen zu erarbeiten waren. Obwohl wir die wichtige Vererbungsbeziehung zwischen Klassen noch nicht offiziell behandelt haben, können Sie vermutlich schon den Hinweis verdauen, dass alle Array-Klassen von der Basisklasse **Array** im Namensraum **System** abstammen, z.B. die Klasse der eindimensionalen Arrays mit Elementen vom Strukturtyp **Int32** (alias **int**):



Hier ist als konkretes Objekt aus dieser Klasse ein Array namens `uni` mit fünf **int**-Elementen zu sehen:



Neben den Array-Elementen enthält das Objekt noch Verwaltungsdaten (z.B. die per **Length**-Eigenschaft ansprechbare Anzahl seiner Elemente).

Beim Zugriff auf ein *einzelnes Element* gibt man nach dem Arraynamen den durch eckige Klammern begrenzten Index an, wobei die Nummerierung bei 0 beginnt und bei *n* Elementen folglich mit *n* - 1 endet. Technisch gesehen liegt ein Array-Zugriffsausdruck mit dem Operator [] vor.

Im Vergleich zur Verwendung einer entsprechenden Anzahl von Einzelvariablen ergibt sich eine gravierende Vereinfachung der Programmierung:

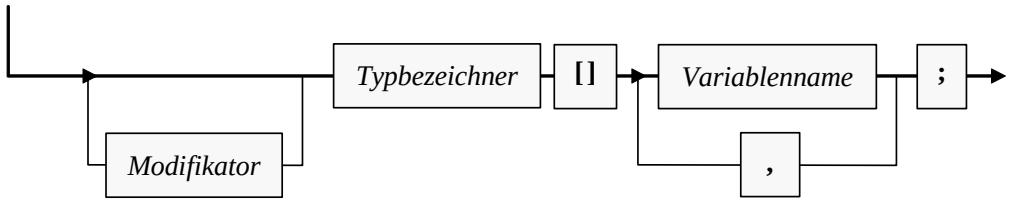
- Weil der Index auch durch einen *Ausdruck* (z.B. durch eine Variable) geliefert werden kann, sind Arrays im Zusammenhang mit den Wiederholungsanweisungen äußerst praktisch.
- Man kann oft die *gemeinsame* Verarbeitung *aller* Elemente (z.B. bei der Ausgabe in eine Datei) per Methodenaufruf mit Array-Aktualparameter veranlassen.
- Viele Algorithmen arbeiten mit Vektoren und Matrizen. Zur Modellierung dieser mathematischen Objekte sind Arrays unverzichtbar.

Wir befassen uns zunächst mit *eindimensionalen* Arrays, behandeln später aber auch den mehrdimensionalen Fall.

5.3.1 Array-Referenzvariablen deklarieren

Im Vergleich zu der bisher bekannten Variablendeklaration (ohne Initialisierung) ist bei Array-Variablen hinter dem Typbezeichner zusätzlich ein Paar eckiger Klammern anzugeben:

Deklaration einer Array-Variablen



Welche Modifikatoren zulässig bzw. erforderlich sind, hängt davon, ob die Variable zu einer Methode, zu einer Klasse oder zu einer Instanz gehört. Die Array-Variable `uni` aus dem einleitend beschriebenen Beispiel gehört zu einer Methode (siehe unten) und ist folgendermaßen zu deklarieren:

```
int[] uni;
```

Bei der Deklaration entsteht nur eine Referenzvariable, jedoch noch kein Array-Objekt. Daher ist auch keine Array-Größe (Anzahl der Elemente) anzugeben.

Einer Array-Referenzvariablen kann als Wert die Adresse eines Arrays mit Elementen vom vereinbarten Typ oder das Referenzliteral **null** zugewiesen werden.

5.3.2 Array-Objekte erzeugen

Mit Hilfe des **new**-Operators erzeugt man ein Array-Objekt mit einem bestimmten Elementtyp und einer bestimmten Größe auf dem Heap. In der folgenden Anweisung entsteht ein Array mit (`max+1`) **int**-Elementen, und seine Adresse landet in der Referenzvariablen `uni`:

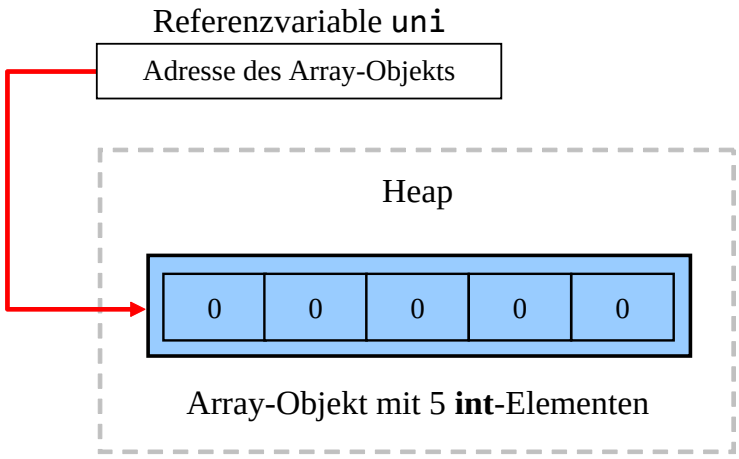
```
uni = new int[max+1];
```

Im **new**-Operanden *muss* hinter dem Datentyp zwischen eckigen Klammern die Anzahl der Elemente festgelegt werden, wobei ein beliebiger Ausdruck mit ganzzahligem Wert (≥ 0) erlaubt ist. Man kann also die Länge eines Arrays *zur Laufzeit* festlegen, z.B. in Abhängigkeit von einer Benutzereingabe.

Die Deklaration einer Array-Referenzvariablen *und* die Erstellung des Array-Objekts kann man natürlich auch in *einer* Anweisung erledigen, z.B.:

```
int[] uni = new int[5];
```

Mit der Verweisvariablen `uni` und dem referenzierten Array-Objekt auf dem Heap haben wir insgesamt die folgende Situation im Speicher:



Weil es sich bei den Array-Elementen um Instanzvariablen eines *Objekts* handelt, erfolgt eine automatische Initialisierung nach den Regeln von Abschnitt 4.2.3. Die **int**-Elemente im Beispiel erhalten folglich den Startwert 0.

Als Objekt wird ein Array vom Garbage Collector entsorgt wird, wenn keine Referenz mehr vorliegt (vgl. Abschnitt 4.4.4). Um eine Referenzvariable aktiv von einem Array-Objekt zu „entkoppeln“, kann man ihr z.B. den Wert **null** (Zeiger auf nichts) oder aber ein alternatives Referenzziel zuweisen. Es ist ohne weiteres möglich, dass mehrere Referenzvariablen auf dasselbe Array-Objekt zeigen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int[] x = new int[3], y;` `  x[0] = 1; x[1] = 2; x[2] = 3;` `  y = x; //y zeigt nun auf das selbe Array-Objekt wie x` `  y[0] = 99;` `  Console.WriteLine(x[0]);` `  }` `}`	`99`

Nachdem ein Array-Objekt erstellt worden ist, lässt sich seine Länge nicht mehr ändern. Seit der .NET - Version 2.0 erlaubt die statische und generische Methode **Resize()** der Klasse **System.Array** scheinbar eine nachträgliche Längenkorrektur, z.B.:

```
Array.Resize(ref uni, 2 * max + 1);
```

Allerdings muss die Methode ein *neues* Objekt erzeugen und die Elemente des alten Arrays umkopieren, was einen erheblichen Aufwand bedeuten kann. Solche Aktionen werden auch bei der in Abschnitt 5.3.10 beschriebenen Kollektionsklasse **ArrayList** mit dynamischer Größenanpassung bei Bedarf im Hintergrund automatisch ausgeführt.

5.3.3 Arrays benutzen

Der Zugriff auf die Elemente eines Array-Objektes geschieht über eine zugehörige Referenzvariable, an deren Namen zwischen eckigen Klammern ein passender Index angehängt wird. Als Index ist ein beliebiger Ausdruck mit ganzzahligem Wert erlaubt, wobei natürlich die Feldgrenzen zu beachten sind. In der folgenden **for**-Schleife wird pro Durchgang ein zufällig gewähltes Element des **int**-Arrays inkrementiert, auf den die Referenzvariable `uni` gemäß obiger Deklaration und Initialisierung zeigt:

```
for (i = 0; i < DRL; i++)
  uni[zzg.Next(5)]++;
```

Den Indexwert liefert die **Random**-Methode **Next()** mit Rückgabetyp **int** (siehe unten). Die **for**-Anweisung stammt aus einer Methode, die in Abschnitt 5.3.5 vorgestellt wird. Dort sind die Variablen `i` und `DRL` aus der **for**-Schleifensteuerung lokal definiert.

Wie in vielen anderen Programmiersprachen hat auch in C# das erste von *n* Array-Elementen die Nummer 0 und folglich das letzte die Nummer *n* - 1. Damit existiert z.B. nach

```
int[] uni = new int[5];
```

kein Element `uni[5]`. Ein Zugriffsversuch führt zum Laufzeitfehler vom Typ **IndexOutOfRange-Exception**, z.B.:

```
Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war
außerhalb des Arraybereichs.
   at UniRand.Main() in ...\BspUeb\Arrays\UniRand\UniRand.cs:line 7
```

Wenn das verantwortliche Programm einen solchen Ausnahmefehler nicht behandelt (siehe Kapitel 12), wird es vom Laufzeitsystem beendet. Man kann sich in C# generell darauf verlassen, dass jede Überschreitung von Feldgrenzen verhindert wird, so dass es nicht zur Verletzung anderer Speicherbereiche und den entsprechenden Folgen (Absturz mit Speicherschutzverletzung, unerklärliches Programmverhalten) kommt.

Die (z.B. durch eine Benutzerentscheidung zur Laufzeit festgelegte) Länge eines Array-Objekts lässt sich über seine (Read-Only -) Eigenschaft **Length** (vom Typ **int)** jederzeit feststellen, z.B.:

Quellcode	Eingaben (**fett**) und Ausgaben

```
using System;
class Prog {
  static void Main() {
    Console.Write("Länge des Vektors: ");
    int[] wecktor =
      new int[Convert.ToInt32(Console.ReadLine())];

    Console.WriteLine();
    for (int i = 0; i < wecktor.Length; i++) {
      Console.Write("Wert von Element " + i + ": ");
      wecktor[i] = Convert.ToInt32(Console.ReadLine());
    }

    Console.WriteLine();
    for(int i = 0; i < wecktor.Length; i++)
      Console.WriteLine(wecktor[i]);
  }
}
```

```
Länge des Vektors: 3

Wert von Element 0: 7
Wert von Element 1: 13
Wert von Element 2: 4711

7
13
4711
```

Auch beim Entwurf von *Methoden* mit Array-Parametern ist es von Vorteil, dass die Länge eines übergebenen Arrays ohne entsprechenden Zusatzparameter in der Methode bekannt ist.

5.3.4 Maximale Array-Größe

Durch den Datentyp **int** (maximaler Wert: 2.147.483.647, siehe Abschnitt 3.3.4) der **Array**-Eigenschaft **Length** wird die Frage nach der maximalen Länge eines Arrays aufgeworfen. Alle relevanten Fakten sind auf der MSDN-Webseite zur Klasse **Array** zusammengestellt.[1] Ist die.NET-Voreinstellung von 2 GB für die maximale Objektgröße in Kraft, ist der **int**-Wertebereich der Eigenschaft **Length** *nicht* der limitierende Faktor für die Array-Länge. Über das Element **gcAllowVeryLargeObjects** der Anwendungskonfigurationsdatei **app.config** lässt sich die 2 GB - Grenze für die maximale Objektgröße allerdings aufheben, was seit .NET 4.5 möglich und nur für 64-Bit - Systeme relevant ist, z.B.:

[1] https://msdn.microsoft.com/en-us/library/System.Array(v=vs.110).aspx

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <startup>
    <supportedRuntime version="v4.0" sku=".NETFramework,Version=v4.6.1"/>
  </startup>
  <runtime>
    <gcAllowVeryLargeObjects enabled="true" />
  </runtime>
</configuration>
```

Dann gelten folgende Größenbeschränkungen für Arrays:

- Der maximale Index in einer Dimension beträgt 0X7FEFFFFF (0X7FFFFFC7, falls ein einzelnes Element nur 1 Byte belegt).
- Ein mehrdimensionaler Array (siehe Abschnitt 5.3.9) darf insgesamt 4 Milliarden Elemente enthalten, wobei sich die Anzahl der Elemente über die Array-Eigenschaft **LongLength** mit Datentyp **long** ermitteln lässt.

5.3.5 Beispiel: Beurteilung des .NET - Pseudozufallszahlengenerators

Oben wurde am Beispiel des 5-elementigen **int**-Arrays `uni` demonstriert, dass die Array-Technik im Vergleich zur Verwendung einzelner Variablen den Aufwand bei der Deklaration und beim Zugriff deutlich verringert. Insbesondere beim Einsatz in einer Schleifenkonstruktion erweist sich die Ansprache der einzelnen Elemente über einen Index als überaus praktisch. Die oben zur Demonstration verwendeten Anweisungen lassen sich leicht zu einem Programm erweitern, das die Verteilungsqualität des .NET - **Pseudozufallszahlengenerators** überprüft. Dieser Generator produziert Folgen von Zahlen mit einem bestimmten Verteilungsverhalten. Obwohl eine Serie perfekt von ihrem Startwert abhängt, kann sie in der Regel echte Zufallszahlen ersetzen. Manchmal ist es sogar von Vorteil, eine Serie über ihren *festen* Startwert reproduzieren zu können. Meist verwendet man aber *variable* Startwerte, z.B. abgeleitet aus einer Zeitangabe. Der Einfachheit halber spricht man oft von *Zufallszahlen* und lässt den *Pseudo*-Zusatz weg.

Man kann übrigens mit moderner EDV-Technik unter Verwendung von physikalischen Prozessen auch *echte* Zufallszahlen produzieren, doch ist der Zeitaufwand im Vergleich zu Pseudozufallszahlen erheblich höher (siehe z.B. Lau 2009).

Nach der folgenden Anweisung zeigt die Referenzvariable `zzg` auf ein Objekt der Klasse **Random** aus dem Namensraum **System**, das als Pseudozufallszahlengenerator taugt:

```
Random zzg = new Random();
```

Durch Verwendung des parameterfreien **Random**-Konstruktors entscheidet man sich für einen aus der Systemzeit abgeleiteten Startwert.

Das angekündigte Programm zieht 10.000 Zufallszahlen aus der Menge {0, 1, ..., 4} und überprüft die empirische Verteilung dieser Stichprobe:

```
using System;
class UniRand {
   static void Main() {
      const int DRL = 10000;
      int i;
      int[] uni = new int[5];
      Random zzg = new Random();
      for (i = 0; i < DRL; i++)
         uni[zzg.Next(5)]++;
      Console.WriteLine("Absolute Häufigkeiten:");
      for (i = 0; i < 5; i++)
         Console.Write(uni[i] + " ");
      Console.WriteLine("\n\nRelative Häufigkeiten:");
      for (i = 0; i < 5; i++)
         Console.Write((double)uni[i]/DRL + " ");
   }
}
```

Die **Random**-Methode **Next()** liefert beim Aufruf mit dem Aktualparameterwert 5 als Rückgabe eine **int**-Zufallszahl aus der Menge {0, 1, 2, 3, 4}, wobei die möglichen Werte mit der gleichen Wahrscheinlichkeit 0,2 auftreten sollten. Im Programm dient der **Next()** - Rückgabewert als Array-Index dazu, ein zufällig gewähltes `uni`-Element zu inkrementieren. Wie das folgende Ergebnis-Beispiel zeigt, stellt sich die erwartete Gleichverteilung in guter Näherung ein:

```
Absolute Häufigkeiten:
1986 1983 1995 1995 2041

Relative Häufigkeiten:
0,1986 0,1983 0,1995 0,1995 0,2041
```

Ein χ^2-Signifikanztest mit der Gleichverteilung als Nullhypothese bestätigt durch eine Überschreitungswahrscheinlichkeit von 0,893 (weit oberhalb der kritischen Grenze 0,05), dass keine Zweifel an der Gleichverteilung bestehen:

uni

	Beobachtetes N	Erwartete Anzahl	Residuum
0	1986	2000,0	-14,0
1	1983	2000,0	-17,0
2	1995	2000,0	-5,0
3	1995	2000,0	-5,0
4	2041	2000,0	41,0
Gesamt	10000		

Statistik für Test

	uni
Chi-Quadrat	1,108[a]
df	4
Asymptotische Signifikanz	**,893**

a. Bei 0 Zellen (,0%) werden weniger als 5 Häufigkeiten erwartet. Die kleinste erwartete Zellenhäufigkeit ist 2000,0.

Über die im Beispielprogramm verwendete Klasse **Random** und deren **Next()** - Methode liefert die FCL-Dokumentation ausführliche Informationen, die vom Visual Studio aus z.B. so zu erreichen sind:

- Einfügemarke auf den Methodennamen setzen
- Funktionstaste **F1** drücken

Es erscheint ein HTML-Dokument, das die Überladungen der Methode auflistet:

Überladungsliste

	Name	Beschreibung
	Next()	Gibt eine nicht negative Zufallsganzzahl zurück.
	Next(Int32)	Gibt eine nicht negative Zufallsganzzahl zurück, die kleiner als das angegebene Maximum ist.
	Next(Int32, Int32)	Gibt eine Zufallsganzzahl zurück, die in einem angegebenen Bereich liegt.

Nach einem Mausklick auf die passende (mittlere) Überladung werden deren Syntax und Funktionsweise erklärt:

Random.Next-Methode: (Int32)

.NET Framework (current version) | Andere Versionen

Veröffentlicht: Oktober 2016

Gibt eine nicht negative Zufallsganzzahl zurück, die kleiner als das angegebene Maximum ist.

Namespace: System
Assembly: mscorlib (in mscorlib.dll)

Syntax

C# | C++ | F# | VB

```
public virtual int Next(
        int maxValue
)
```

Parameter

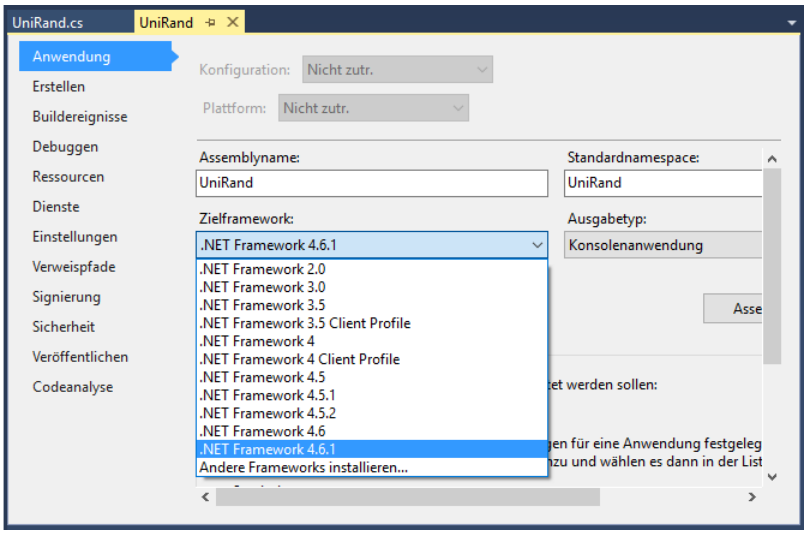
maxValue
Type: System.Int32
Die exklusive obere Grenze der Zufallszahl generiert werden. *maxValue* muss größer als oder gleich 0 sein.

Rückgabewert
Type: System.Int32
Eine 32-Bit-Ganzzahl mit Vorzeichen, die größer als oder gleich 0 und kleiner als *maxValue*; d. h. der Bereich der Rückgabewerte umfasst 0, aber nicht *maxValue*. Jedoch wenn *maxValue* gleich 0 ist, *maxValue* zurückgegeben wird.

Sollte sich die Hilfe auf eine unerwartete .NET - Version beziehen, müssen Sie nach

Projekt > Eigenschaften > Anwendung

das **Zielframework** ändern, z.B.:



Eine weitere Anleitung zur Nutzung der sehr guten FCL-Dokumentation ist in diesem Manuskript sicher nicht mehr erforderlich.

5.3.6 Suchen und Sortieren

Die Klasse **Array** bietet zahlreiche statische Methoden, die einen Array nach dem ersten Auftreten eines Wertes durchsuchen. Im folgenden Programm wird die Methode **IndexOf()** verwendet, die den Index des ersten Treffers liefert oder -1, wenn kein Element mit dem angegebenen Wert gefunden wurde:[1]

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  const int LEN = 100000;` `  const int MXR = 50000;` `  const int NCAND = 5;` `  int[] iar = new int[LEN];` `  var zzg = new Random();` `  for (int i = 0; i < LEN; i++)` `  iar[i] = zzg.Next(MXR);` `  for (int i = 0; i < NCAND; i++)` `  Console.WriteLine("i=" + i + ", Index=" +` `  Array.IndexOf(iar, zzg.Next(MXR)));` `  }` `}`	`i=0, Index=9605` `i=1, Index=92345` `i=2, Index=21988` `i=3, Index=-1` `i=4, Index=13`

Bei alternativen Überladungen der Methode kann durch weitere Parameter festgelegt werden, ...

- dass die Suche an einer bestimmten Indexposition starten soll,
- dass nur eine bestimmte Anzahl von Elementen durchsucht werden soll.

Während die Methode **IndexOf()** einen Array in aufsteigender Ordnung durchsucht, arbeitet die Methode **LastIndexOf()** in umgekehrter Reihenfolge. Während die beiden eben genannten Methoden nach einem festen Wert suchen, fahnden die Methoden **FindIndex()** und **FindLastIndex()** nach einem Wert mit einer bestimmten Eigenschaft (z.B. Teilbarkeit durch 12 bei **int**-Werten). Die eben erwähnten und weitere Suchmethoden der Klasse **Array** werden von Griffith (2013, S. 155ff) ausführlich beschrieben.

Beim Sortieren eines Arrays über eine von den zahlreichen Überladungen der **Array**-Methode **Sort()** resultiert die Vielfalt u.a. daraus, dass

- entweder die „natürliche“ Anordnung der Elemente, basierend auf der **CompareTo()** - Methode des Elementtyps benutzt wird,
- oder ein die Schnittstelle **IComparer<T>** erfüllendes Objekt engagiert wird, das eine beliebig definierte Anordnung von zwei Elementen liefert.

Das folgende simple Programm sortiert **int**-Werte auf kanonische Weise:

---

[1] In der Darstellung wird (die Fähigkeit des Compilers zur Typinferenz ausnutzend) die generische Natur der Methoden **IndexOf<T>()**, **LastIndexOf<T>()**, **FindIndex<T>()** und **FindLastIndex<T>()** unterschlagen (siehe Abschnitt 7.4).

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int[] iar = {7, 2, 4, 3, 5, 1, 6};` `  Array.Sort(iar);` `  foreach (int i in iar)` `  Console.WriteLine(i);` `  }` `}`	1 2 3 4 5 6 7

Ist ein Array sortiert, kann die Suche nach einem Element durch das **binären Suchverfahren** erheblich beschleunigt werden:

- Im ersten Schritt wird der gesuchte Wert mit dem Element in der Mitte des Arrays verglichen:
  - Stimmen beide überein, ist die Suche erfolgreich beendet.
  - Ist der gesuchte Wert größer als das mittlere Array-Element, wird die Suche in der oberen Array-Hälfte fortgesetzt.
  - Ist der gesuchte Wert kleiner als das mittlere Array-Element, wird die Suche in der unteren Array-Hälfte fortgesetzt.
- Im zweiten Schritt wird in der relevanten Array-Hälfte das Element in der Mitte aufgesucht usw.
- Wird der gesuchte Wert nicht gefunden, so liefert das Verfahren immerhin Informationen über die nach Sortierordnung nächstgelegenen Werte.

Das folgende Beispielprogramm nach Griffith (2013, S. 158) zeigt allerdings, dass der Zeitgewinn beim binären im Vergleich zum einfachen Suchen auf keinen Fall den Aufwand des vorherigen Sortierens rechtfertigt:

```
using System;
class Prog {
   static void Main() {
      const int LEN = 1000000;
      const int MXR = 1000000;
      const int NCAND = 10;

      int[] iar = new int[LEN];
      Random zzg = new Random();
      for (int i = 0; i < LEN; i++)
         iar[i] = zzg.Next(MXR);

      long time = DateTime.Now.Ticks;
      time = DateTime.Now.Ticks;
      for (int i = 0; i < NCAND; i++)
         Console.WriteLine("i=" + i + ", Index=" +
            Array.IndexOf(iar, zzg.Next(MXR)));
      time = DateTime.Now.Ticks - time;
      Console.WriteLine("\nBenöt. Zeit für die einfache Suche:\t" +
             time / 1.0e4 + " Millisek.");

      time = DateTime.Now.Ticks;
      Array.Sort(iar);
      time = DateTime.Now.Ticks - time;
      Console.WriteLine("\nBenöt. Zeit für das Sortieren:\t\t" +
                        time / 1.0e4 + " Millisek.");
```

```
        time = DateTime.Now.Ticks;
        for (int i = 0; i < NCAND; i++)
           Console.WriteLine("i=" + i + ", Index=" +
              Array.BinarySearch(iar, zzg.Next(MXR)));
        time = DateTime.Now.Ticks - time;
        Console.WriteLine("\nBenöt. Zeit für die binäre Suche:\t" +
                          time / 1.0e4 + " Millisek.");
     }
  }
```

In einem Array mit 1 Milliarde **int**-Elementen mit Zufallswerten aus dem Bereich von 0 bis 1.000.000-1 werden 10 Werte gesucht:

- Vor dem Sortieren mit der Methode **IndexOf()**
- Nach dem Sortieren mit der Methode **BinarySearch()**

Bei **BinarySearch()** signalisiert eine negative Rückgabe eine gescheiterte Suche und enthält gleichzeitig Informationen über die nach Sortierordnung nächstgelegenen Werte:[1]

```
i=0, Index=849557
i=1, Index=268827
i=2, Index=371815
i=3, Index=-1
i=4, Index=-1
i=5, Index=294784
i=6, Index=221149
i=7, Index=561921
i=8, Index=216117
i=9, Index=-1

Benöt. Zeit für die einfache Suche:     10,0093 Millisek.

Benöt. Zeit für das Sortieren:          91,6215 Millisek.
i=0, Index=393513
i=1, Index=554658
i=2, Index=243872
i=3, Index=167706
i=4, Index=-60951
i=5, Index=708431
i=6, Index=987699
i=7, Index=366170
i=8, Index=644718
i=9, Index=845861

Benöt. Zeit für die binäre Suche:       4,5047 Millisek.
```

Die binäre Suche ist also nur dann sinnvoll, wenn ein Array bereits sortiert vorliegt.

5.3.7 Initialisierungslisten

Bei Arrays mit wenigen Elementen ist die Möglichkeit von Interesse, beim Deklarieren der Referenzvariablen eine Initialisierungsliste anzugeben und das Array-Objekt dabei implizit (ohne Verwendung des **new**-Operators) zu erzeugen, z.B.:

[1] Siehe Beispiel in: https://msdn.microsoft.com/de-de/library/2cy9f6wb(v=vs.110).aspx

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int[] wecktor = {1, 2, 3};` `  Console.WriteLine(wecktor[2]);` `  }` `}`	`3`

Die Deklarations- und Initialisierungsanweisung

```
int[] wecktor = {1, 2, 3};
```

ist äquivalent zu:

```
int[] wecktor = new int[3];
wecktor[0] = 1;
wecktor[1] = 2;
wecktor[2] = 3;
```

Initialisierungslisten sind nicht nur bei der Deklaration erlaubt, sondern auch bei der Objektkreation per **new**-Operator, z.B.:

```
int[] wecktor;
 . . .
wecktor = new int[] {1, 2, 3};
```

Haben alle Elemente einer Initialisierungsliste denselben Datentyp, kann man die Typangabe weglassen und die Typinferenz des Compilers nutzend das eckige Klammernpaar direkt hinter das Schlüsselwort **new** schreiben, z.B.:

```
wecktor = new[] {1, 2, 3};
```

Die implizite Objektkreation (*ohne* Schlüsselwort **new**) akzeptiert der Compiler aber ausschließlich bei der Array-Deklaration mit Initialisierung, im folgenden Beispiel also *nicht*:

```
wecktor = {1, 2, 3}; // Nicht erlaubt!
```

5.3.8 Objekte als Array-Elemente

Für die Elemente eines Arrays sind natürlich auch Referenztypen erlaubt. In folgendem Beispiel wird ein Array mit `Bruch`-Objekten erzeugt:

Quellcode	Ausgabe
`using System;` `class BruchRechnung {` `  static void Main() {` `  Bruch b1 = new Bruch(1, 2, "b1 = ");` `  Bruch b2 = new Bruch(5, 6, "b2 = ");` `  Bruch[] bruvek = {b1, b2};` `  bruvek[1].Zeige();` `  }` `}`	`  5` `b2 = -----` `  6`

5.3.9 Mehrdimensionale Arrays

5.3.9.1 Rechteckige Arrays

In der linearen Algebra und in vielen anderen Anwendungsbereichen werden auch *mehrdimensionale* Arrays benötigt, z.B.:

Quellcode	Ausgabe

```
using System;
class Prog {
   static void Main() {
      int[,] matrix = new int[4, 3];
      int nrow = matrix.GetLength(0);
      int ncol = matrix.GetLength(1);
      int nelem = matrix.Length;
      Console.WriteLine("{0} Dimensionen,\n{1} Zeilen, {2} Spalten" +
                "\n{3} Elemente", matrix.Rank, nrow, ncol, nelem);
      for (int i = 0; i < nrow; i++) {
         for (int j = 0; j < ncol; j++) {
            matrix[i, j] = (i + 1) * (j + 1);
            Console.Write("{0,3}", matrix[i, j]);
         }
         Console.WriteLine();
      }
   }
}
```

```
2 Dimensionen,
4 Zeilen, 3 Spalten
12 Elemente
  1  2  3
  2  4  6
  3  6  9
  4  8 12
```

Im Beispiel wird ein *zweidimensionaler* Array (eine Matrix) mit 4 Zeilen und 3 Spalten erzeugt, auf deren Zellen man per Doppelindizierung zugreifen kann. Bei der Erzeugung bzw. Verwendung eines mehrdimensionalen rechteckigen Arrays werden die in eckigen Klammern eingeschlossenen Dimensionsangaben bzw. Indexwerte durch Komma getrennt.

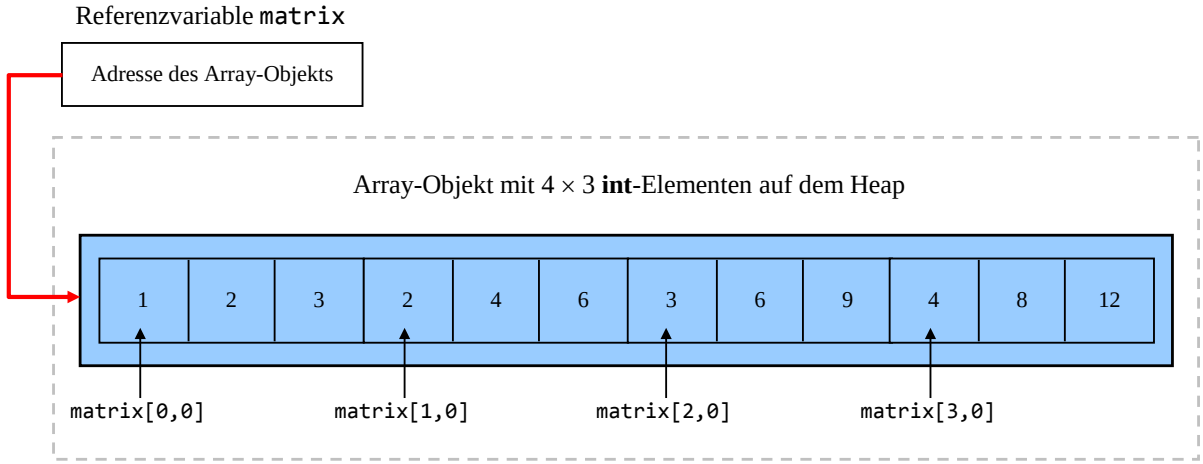
Auch im mehrdimensionalen Fall können Initialisierungslisten eingesetzt werden, wobei z.B. die Elemente einer Matrix zeilendominant anzugeben sind (als Liste von Zeilenvektoren):

```
int[,] matrix = {{1, 2, 3}, {2, 4, 6}, {3, 6, 9}, {4, 8, 12}};
```

Weil alle Arrays von der Basisklasse **Array** im Namensraum **System** abstammen, verfügen sie über entsprechende Methoden und Eigenschaften (siehe FCL-Dokumentation), z.B.:

- Die Eigenschaft **Rank** enthält die Anzahl der Dimensionen.
- Über die Methode **GetLength()** erfährt man von einem Array-Objekt die Anzahl der Indexwerte in der per Parameter angegebenen Dimension.
- Die Eigenschaft **Length** liefert die Gesamtzahl der Array-Elemente, im Beispiel also 4 · 3 = 12.

Bei den bisher behandelten *rechteckigen* Arrays liegen im Speicher alle Elemente unmittelbar hintereinander, was einen schnellen Indexzugriff ermöglicht:



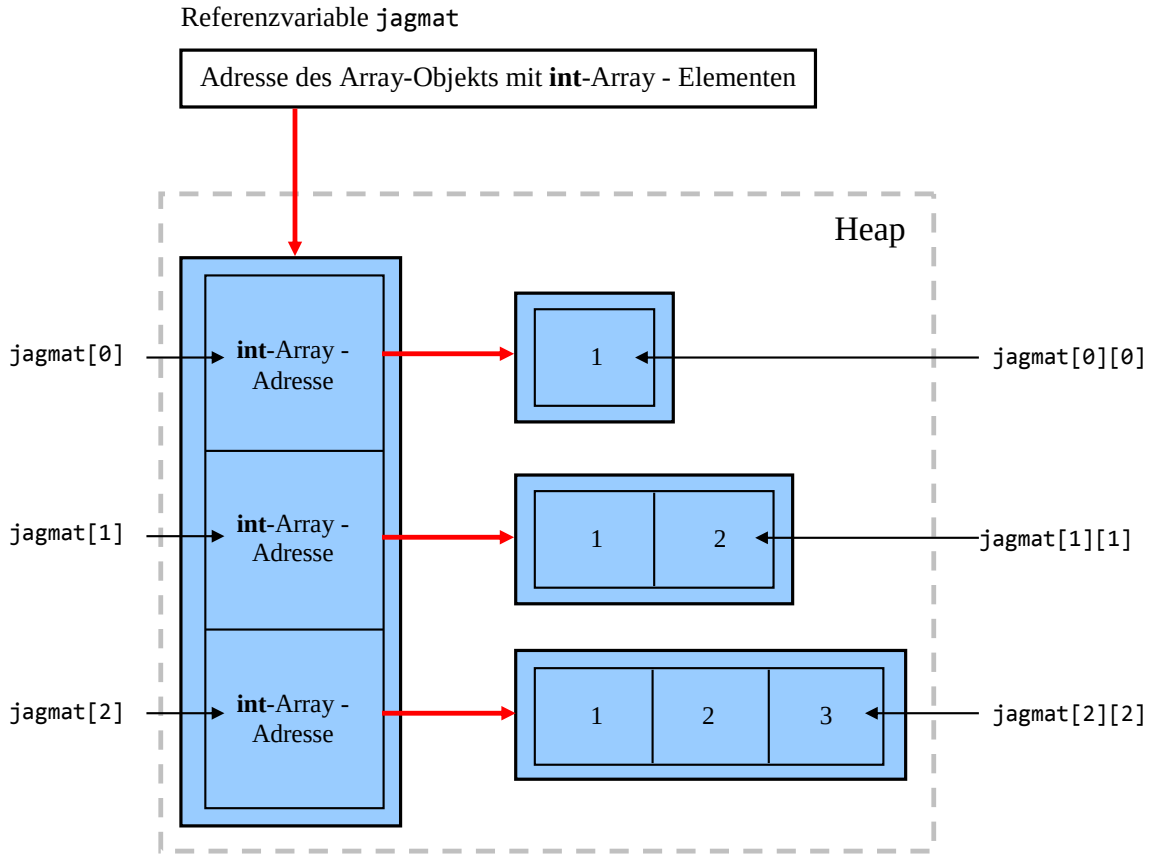
Wie das Speicherabbild zeigt, ist bei regelmäßigen Arrays jeder beliebige Rang möglich, wobei der vertraute Ausdruck *rechteckig* streng genommen nur bei zweidimensionalen Arrays passt.

5.3.9.2 Mehrdimensionale Arrays mit unterschiedlich großen Elementen

Neben den rechteckigen Arrays unterstützt C# auch „ausgesägte“ Exemplare mit unterschiedlich großen Elementen (engl.: *jagged arrays*). So lässt sich etwa eine zweidimensionale Matrix mit unterschiedlich langen Zeilen realisieren:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  int[][] jagmat = new int[3][];` `  jagmat[0] = new int[1] {1};` `  jagmat[1] = new int[2] {1, 2};` `  jagmat[2] = new int[3] {1, 2, 3};` `  for (int i = 0; i < jagmat.Length; i++) {` `  for (int j = 0; j < jagmat[i].Length; j++)` `  Console.Write(jagmat[i][j] + " ");` `  Console.WriteLine();` `  }` `  }` `}`	`1` `1 2` `1 2 3`

Im Unterschied zum **int[,]** - Objekt `matrix` aus dem Beispiel in Abschnitt 5.3.9.1, das doppelt indizierte **int**-Elemente enthält, handelt es sich bei den Elementen des **int[][]** - Objekts `jagmat` um einfach indizierte Referenzen vom Typ **int[]**, die wiederum auf entsprechende Heap-Objekte (oder **null**) zeigen können. Während `matrix` ein *zwei*dimensionaler Array mit **int**-Elementen ist, handelt es sich bei `jagmat` um einen *ein*dimensionalen Array mit **int[]** - Elementen:



Beim Erzeugen des `jagmat`-Objekts darf nur die Elementzahl des äußeren Arrays angegeben werden:[1]

```
int[][] jagmat = new int[3][];
```

Anschließend erzeugt man die Zeilenobjekte und legt ihre Adressen in den `jagmat`-Elementvariablen ab, z.B.:

```
jagmat[2] = new int[3] {1, 2, 3};
```

Auch bei einem ausgesägten Array lässt sich mit Hilfe einer Initialisierungsliste und durch Ausnutzung von Compiler-Intelligenz (automatische Ermittlung des Typs und der Elementzahl des äußeren Arrays) der Schreibaufwand reduzieren. Es folgt eine zulässige Alternative zu der aus didaktischen Gründen oben bevorzugten ausführlichen Schreibweise:

```
int[][] jagmat = {new[] {1}, new[] {1, 2}, new[] {1, 2, 3}};
```

Im Unterschied zur Initialisierungsliste für einen rechteckigen Array (vgl. Abschnitt 5.3.9.1) muss der **new**-Operator für jede Zeile wiederholt werden, weil es sich um einen Array mit selbständigen Arrays als Elementen handelt.

5.3.10 Die Kollektionsklasse ArrayList

Im Namensraum **System.Collections** bietet die FCL etliche Klassen zur flexiblen Verwaltung von Datenbeständen *variablen* Umfangs, mit denen wir uns später noch detailliert beschäftigen werden. Dort findet sich u.a. die Klasse **ArrayList**, deren Objekte analog zu eindimensionalen Arrays genutzt werden können. Man legt jedoch beim Erzeugen eines **ArrayList**-Objekts keinen Umfang fest, sondern kann z.B. mit der Methode **Add()** nach Bedarf neue Elemente einfügen, z.B.:

Quellcode	Ausgabe
`using System;` `using System.Collections;` `class ArrayListDemo {` `  static void Main() {` `  var al = new ArrayList();` `  string s;` `  Console.WriteLine("Was fällt Ihnen zu C# ein?\n");` `  do {` `  Console.Write(": ");` `  s = Console.ReadLine();` `  if (s.Length > 0)` `  al.Add(s);` `  else` `  break;` `  } while (true);` `  Console.WriteLine("\nIhre Anmerkungen:");` `  for(int i = 0; i < al.Count; i++)` `  Console.WriteLine(al[i]);` `  }` `}`	`Was fällt Ihnen zu C# ein?` `: Tolle Sache` `: Nicht ganz trivial` `: Macht Spaß` `:` `Ihre Anmerkungen:` `Tolle Sache` `Nicht ganz trivial` `Macht Spaß`

Das Fassungsvermögen des Containers wird bei Bedarf automatisch erhöht, wobei eine leistungsoptimierende Logik dafür sorgt, dass diese Anpassungsmaßnahme möglichst selten erforderlich ist. Über die Eigenschaft **Capacity** kann die momentane Kapazität festgestellt und auch eingestellt

[1] Nach Griffith (2013, S. 163f) sollte im **new**-Ausdruck die Elementzahl des äußeren Arrays (mit dem Elementtyp **int[]**!) eigentlich durch das zweite Klammernpaar begrenzt werden. Die C# - Designer haben sich aber anders entschieden, eventuell mit Rücksicht auf die Java-Sprachdefinition.

werden. Mit der Methode **TrimToSize()** reduziert man die Größe auf den momentanen Bedarf, z.B. nach der voraussichtlich letzten Neuaufnahme.

Weil die Klasse **ArrayList** einen *Indexer* bietet (siehe Abschnitt 5.6), kann man per Indexsyntax auf die Elemente zugreifen:

- Das erste Element hat den Indexwert 0.
- Das letzte Element hat den Indexwert (**Count** – 1), wobei die **Count**-Eigenschaft die Anzahl der Elemente angibt.

Als Datentyp für die **ArrayList**-Elemente dient **System.Object**, also die Klasse an der Spitze des allgemeinen Typsystems der .NET - Plattform. Folglich kann ein **ArrayList**-Container Daten beliebigen Typs aufnehmen, wobei dank Boxing-Technik (siehe Abschnitt 5.2) auch die Werttypen erlaubt sind, z.B.:

Quellcode	Ausgabe

```
using System;
using System.Collections;
class Prog {
   static void Main() {
      var al = new ArrayList();
      al.Add("Wort");
      al.Add(3.14);
      al.Add(13);
      foreach (object o in al)
         Console.WriteLine(o);
   }
}
```

```
Wort
3,14
13
```

Ein solcher „Gemischtwarenladen“ (allerdings mit *fester* Länge) ist durch Wahl des Element-Datentyps **Object** übrigens auch mit einem einfachen Array zu realisieren.

Es ist selten etwas schon so gut, dass es nicht noch verbessern könnte. So werden wir im Kapitel 7 über *generische* Klassen lernen, dass dem Gemischtwarenladen **ArrayList** bei sehr vielen Anwendungen die generische Klasse **List<T>** überlegen ist. Sie bietet einen größendynamischen Container mit einem festen, beim Erstellen des Containers zu bestimmenden Elementtyp. Wenn jedoch tatsächlich ein Gemischtwarenladen benötigt wird, kommt die Klasse **ArrayList** weiterhin in Frage.

5.4 Klassen für Zeichenketten

C# bietet für den Umgang mit Zeichenketten, die grundsätzlich aus Unicode-Zeichen bestehen, zwei Klassen an:

- **String** (im Namensraum **System**)
  String-Objekte können nach dem Erzeugen nicht mehr geändert werden. Diese Klasse ist für den *lesenden* Zugriff auf Zeichenketten optimiert.
- **StringBuilder** (im Namensraum **System.Text**)
  Für *variable*, d.h. im Programmablauf häufig zu ändernde, Zeichenketten sollte unbedingt die Klasse **StringBuilder** verwendet werden, weil deren Objekte nach dem Erzeugen noch modifiziert werden können.

5.4.1 Die Klasse String für unveränderliche Zeichenketten

Weil Objekte der Klasse **String** aus dem Namensraum **System** in C# - Programmen sehr oft benötigt werden, hat man diesem Datentyp das reservierte Wort **string** (mit *klein* geschriebenen Anfangsbuchstaben) als Aliasnamen spendiert, und das ist nicht die einzige syntaktische Vorzugsbehandlung gegenüber anderen Klassen. In der folgenden Deklarations- und Initialisierungsanweisung

```
string s1 = "abcde";
```

wird:

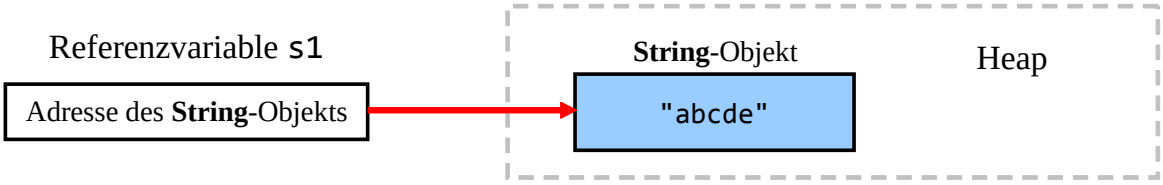
- eine **String**-Referenzvariable namens `s1` angelegt,
- ein neues **String**-Objekt mit dem Inhalt „abcde“ auf dem Heap erzeugt,
- die Adresse des neuen Heap-Objekts in der Referenzvariablen abgelegt.

Soviel objektorientierten Hintergrund sieht man der angenehm einfachen Anweisung auf den ersten Blick nicht an. In C# sind jedoch auch Zeichenketten*literale* als **String**-Objekte realisiert, so dass z.B.

```
"abcde"
```

einen Ausdruck darstellt, der als Wert einen Verweis auf ein **String**-Objekt auf dem Heap liefert.

Weil in der obigen Deklarations- und Initialisierungsanweisung kein **new**-Operator auftaucht, spricht man auch vom *impliziten* Erzeugen eines **String**-Objekts. Die Anweisung bewirkt im Hauptspeicher folgende Situation:



5.4.1.1 String als WORM - Klasse

Nachdem ein **String**-Objekt auf dem Heap erzeugt worden ist, kann es nicht mehr geändert werden. In der Überschrift zu diesem Abschnitt wird für diesen Sachverhalt eine Abkürzung aus der Elektronik ausgeliehen: WORM (***W**rite **O**nce **R**ead **M**any*). Eventuell werden Sie die Unveränderlichkeit des **String**-Inhalts in Zweifel ziehen und ein Gegenbeispiel der folgenden Art vorbringen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  String testr = "abc";` `  Console.WriteLine("testr = " + testr);` `  testr = testr + "def";` `  Console.WriteLine("testr = " + testr);` `  }` `}`	`testr = abc` `testr = abcdef`

In der Zeile

```
testr = testr + "def";
```

wird aber das per `testr` ansprechbare **String**-Objekt (mit dem Text „abc“) nicht geändert, sondern durch ein neues **String**-Objekt (mit dem Text „abcdef“) ersetzt. Das alte Objekt ist noch vorhanden, aber nicht mehr referenziert. Sobald das Laufzeitsystem Langeweile hat oder Speicher benötigt, wird das alte Objekt vom Garbage Collector eliminiert.

5.4.1.2 Methoden für String-Objekte

Von den zahlreichen Methoden der Klasse der **String** werden in diesem Abschnitt nur die wichtigsten angesprochen. Für spezielle Anwendungen lohnt sich also ein Blick in die FCL-Dokumentation.

5.4.1.2.1 Verketten von Strings

Weil die Klasse **String** den „+“- Operator geeignet überladen hat (vgl. Abschnitt 4.7.2), taugt er zum Verketten von **String**-Objekten, wobei Operanden beliebiger Datentypen bei Bedarf automatisch in **String**-Objekte konvertiert werden. Wie Sie aus Abschnitt 5.4.1.1 wissen, entsteht beim Verketten von zwei Zeichenfolgen ein *neues* **String**-Objekt. Im folgenden Beispiel wird mit Klammern dafür gesorgt, dass der Compiler die „+“ - Operatoren jeweils sinnvoll interpretiert (Verketten von Strings bzw. Addieren von Zahlen):

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine("4 + 3 = " + (4 + 3));` `  }` `}`	`4 + 3 = 7`

5.4.1.2.2 Vergleichen von Strings

Angewandt auf **String**-Variablen vergleichen die auf **Identität** prüfenden Operatoren „==“ und „!=“ *nicht* (wie z.B. in Java) die *Adressen*, sondern die *Inhalte* der referenzierten Zeichenfolgenobjekte, z.B.:

Das C#-Programm liefert **true**	Das Java-Programm liefert **false**
`using System;` `class Prog {` `  static void Main() {` `  String s1 = "abcde";` `  String s2 = "de";` `  String s3 = "abc" + s2;` `  Console.WriteLine(s1 == s3);` `  }` `}`	`class Prog {` `  public static void main(String[] args) {` `  String s1 = "abcde";` `  String s2 = "de";` `  String s3 = "abc" + s2;` `  System.out.println(s1 == s3);` `  }` `}`

Mit der etwas umständlichen `s3`-Konstruktion wird verhindert, dass `s1` und `s3` auf dasselbe Objekt im internen **String**-Pool zeigen, weil dann Adress- und Inhaltsvergleich zum selben Ergebnis kämen. Wie in Abschnitt 5.4.1.3 demonstriert wird, ist bei einer großen Anzahl von **String**-Vergleichen durch das so genannte Internalisierung und die Verwendung von Adressvergleichen eine Leistungssteigerung zu erzielen.

Zum Testen auf **lexikographische Priorität** (z.B. beim Sortieren) kann die **String**-Methode **CompareTo()** dienen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  String a = "Müller, Anja", b = "Müller, Kurt",` `  c = "Müller", d = ", Anja";` `  c = c + d; //a und c sollen nicht auf denselben Pool-String zeigen` `  Console.WriteLine("< : " + a.CompareTo(b));` `  Console.WriteLine("= : " + a.CompareTo(c));` `  Console.WriteLine("> : " + b.CompareTo(a));` `  }` `}`	`< : -1` `= : 0` `> : 1`

CompareTo() liefert folgende Ergebnisse zurück:

		CompareTo() - Ergebnis
Die lexikographische Priorität des angesprochenen **String**-Objekts ist im Vergleich zum Parameterobjekt:	kleiner	-1
	gleich	0
	größer	1

5.4.1.2.3 Länge einer Zeichenkette

Über die Länge einer Zeichenkette informiert die **String**-Eigenschaft **Length**, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine("abc".Length);` `  }` `}`	`3`

5.4.1.2.4 Zeichen(folgen) extrahieren, suchen oder ersetzen

Auf einzelne Zeichen eines Strings kann man per Indexsyntax zugreifen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  string s = "abcd";` `  Console.WriteLine(s[0]);` `  Console.WriteLine(s.Substring(0, 2));` `  Console.WriteLine(s.IndexOf("c"));` `  Console.WriteLine(s.IndexOf("x"));` `  Console.WriteLine(s.StartsWith("a"));` `  Console.WriteLine(s.Replace('c', 'C'));` `  }` `}`	`a` `ab` `2` `-1` `True` `abCd`

Über die Methode

public string Substring(int *start***, int** *anzahl***)**

erhält man von einem **String**-Objekt die *anzahl* Zeichen ab Position *start* als Kopie.

Mit der Methode

public int IndexOf(string *gesucht***)**

kann man einen **String** nach der Existenz einer anderen Zeichenkette befragen. Als Rückgabewert erhält man ...

- nach erfolgreicher Suche: die (0-basierte) Startposition der ersten Trefferstelle
- nach vergeblicher Suche: -1

Mit der Methode

public bool StartsWith(string *start***)**

lässt sich feststellen, ob ein String mit einer bestimmten Zeichenfolge beginnt.

Mit den Überladungen der Methode

public string Replace(char *alt*, **char** *neu***)**

public string Replace(string *alt*, **string** *neu***)**

erhält man als Rückgabewert die Adresse eines neuen **String**-Objekts, das aus dem angesprochenen Original durch Ersetzen eines alten Zeichens (einer alten Zeichenfolge) durch ein neues Zeichen (eine neue Zeichenfolge) hervorgeht.

5.4.1.2.5 Groß-/Kleinschreibung normieren

Mit den Methoden

public String ToUpper()

bzw.

public String ToLower()

erhält man einen neuen **String**, der im Unterschied zum angesprochenen Original auf Groß- bzw. Kleinschreibung normiert ist, was vor Vergleichen oft sinnvoll ist, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  String a = "Otto", b = "otto";` `  Console.WriteLine(a.ToUpper() == b.ToUpper());` `  Console.WriteLine(a.ToUpper().IndexOf("T"));` `  }` `}`	`True` `1`

In der letzten Anweisung des Beispiels ist der **WriteLine()** - Parameter etwas komplex geraten, so dass vielleicht eine kurze Erklärung angemessen ist:

- Der linke Punktoperator wird zuerst ausgeführt. Dabei erzeugt der Methodenaufruf `a.ToUpper()` ein neues **String**-Objekt und liefert eine zugehörige Referenz.
- Diese Referenz ermöglicht es, dem neuen Objekt Botschaften zu übermitteln, was im Methodenaufruf `IndexOf("T")` geschieht.

5.4.1.3 Interner String-Pool

Wie oben erläutert, bildet ein Zeichenkettenliteral einen Ausdruck vom Typ **String** mit einer Objektadresse als Wert. Für wiederholt im Quellcode auftretende identische Zeichenkettenliterale lässt dieses Prinzip eine Verschwendung von Speicherplatz befürchten. Um das zu verhindern, verwaltet die CLR eine als **interner String-Pool** bezeichnete Tabelle für die auf Literalen basierenden **String**-Objekte. Wird zu einem Zeichenkettenliteral eine Objektreferenz benötigt, liefert die CLR nach Möglichkeit die Adresse eines bereits vorhandenen, inhaltsgleichen **String**-Objekts aus dem internen Pool. Schlägt die Suche fehl, wird ein neues Objekt erzeugt und im internen Pool registriert. Diese Vorgehensweise ist sinnvoll, weil sich vorhandene **String**-Objekte garantiert nicht mehr ändern. Im folgenden Beispiel zeigen eine Instanzvariable und eine lokale Variable vom Typ **String** auf dasselbe Objekt:

Quellcode	Ausgabe
`using System;` `class Prog {` `  string sf = "Dies ist ein Zeichenketten-Literal";` `  static void Main() {` `  string ls = "Dies ist ein Zeichenketten-Literal";` `  Prog p = new Prog();` `  Console.WriteLine(Object.ReferenceEquals(ls, p.sf));` `  }` `}`	`True`

Über die statische **String**-Methode **Intern()** kann man den internen **String**-Pool zusätzlich bevölkern. Die Methode erwartet einen Aktualparameter vom Typ **String** und liefert einen Rückgabewert vom selben Typ:

- Ist ein Objekt im internen **String**-Pool inhaltsgleich mit dem Parameterobjekt, wird die Adresse dieses Pool-Objekts geliefert.
- Anderenfalls nimmt **Intern()** das Parameterobjekt in den internen **String**-Pool auf und liefert seine Adresse als Rückgabe.

Das Internalisieren kann nicht nur Speicherplatz sparen, wenn viele Strings mit identischem Inhalt zu erwarten sind, sondern es kann vor allem Identitätsvergleiche für Strings (vgl. Abschnitt 5.4.1.2.2) beschleunigen, weil bei Referenzvariablen zu internalisierten Strings aus der Gleichheit der Adressen bereits die Inhaltsgleichheit folgt. Im folgenden Programm werden `ANZ` Zufallszeichenfolgen der Länge `LEN` jeweils `N` mal mit einem zufällig gewählten Partner verglichen. Dies geschieht zunächst per Inhaltsvergleich und dann nach dem zwischenzeitlichen Internalisieren per Adressvergleich:

```
using System;
using System.Text;

class StringIntern {
   public static void Main() {
      const int ANZ = 50000, LEN = 50, N = 50;
      var sb = new StringBuilder();
      var ran = new Random();
      String[] sar  = new String[ANZ];
      for (int i = 0; i < ANZ; i++) {
         for (int j = 0; j < LEN; j++)
            sb.Append((char) (65 + ran.Next(26)));
         sar[i] = sb.ToString();
         sb.Remove(0, LEN);
      }

      // DateTime außerhalb der Messung laden
      long start = DateTime.Now.Ticks;

      start = DateTime.Now.Ticks;
      int hits = 0;
      // N * ANZ Inhaltsvergleiche
      for (int n = 1; n <= N; n++)
         for (int i = 0; i < ANZ; i++)
            if (sar[i] == sar[ran.Next(ANZ)])
               hits++;
      Console.WriteLine((N * ANZ)+" Inhaltsvergleiche ("+hits+
            " hits) benötigen "+((DateTime.Now.Ticks - start)/1.0e4)+
            " Millisekunden");

      start = DateTime.Now.Ticks;
      hits = 0;
      // Internalisieren
      for (int j = 1; j < ANZ; j++)
         sar[j] = String.Intern(sar[j]);
      Console.WriteLine("Zeit für das Internalisieren: " +
            ((DateTime.Now.Ticks - start) / 1.0e4) + " Millisekunden");
```

```
        // N * ANZ Adressvergleiche
        for (int n = 1; n <= N; n++)
           for (int i = 0; i < ANZ; i++)
              if (((Object)sar[i]) == sar[ran.Next(ANZ)])
                 hits++;
        Console.WriteLine((N * ANZ)+" Adressvergleiche ("+hits+
              " hits) benötigen (inkl. Internalisieren) "+
              ((DateTime.Now.Ticks - start)/1.0e4)+" Millisekunden");
        Console.ReadLine();
     }
  }
```

Beim Erzeugen der Zufallszeichenfolgen kommt ein Objekt der Klasse **StringBuilder** zum Einsatz (siehe Abschnitt 5.4.2).

Um den Identitätsoperator zu Adressvergleichen zu zwingen, wird ein Vergleichspartner als Instanz der Klasse **Object** behandelt:

```
((Object)sar[i]) == sar[ran.Next(ANZ)]
```

Es hängt von den Aufgabenparametern `ANZ`, `LEN` und `N` ab, welche Vergleichsmethode überlegen ist:[1]

	Laufzeit in Millisekunden	
	Inhaltsvergleiche	Internalisieren u. Adressvergl.
`ANZ` = 50000, `LEN` = 50, `N` = 5	24,0296	34,0307
`ANZ` = 50000, `LEN` = 50, `N` = 50	230,537	76,068

Erwartungsgemäß ist das Internalisieren umso rentabler, je mehr Vergleiche anschließend mit den Zeichenfolgen angestellt werden. Bei **String**-Vergleichen sind sicher noch weitere Verbesserungen möglich, z.B. durch Ausnutzen der lexikographischen Ordnung.

Auch die statische **String**-Methode **IsInterned()** liefert die Adresse des Parameter-Strings, falls er sich im internen Pool befindet. Anderenfalls wird jedoch *kein* Pool-String erzeugt, sondern der Wert **null** abgeliefert.

5.4.2 Die Klasse StringBuilder für veränderliche Zeichenketten

Für häufig zu ändernde Zeichenketten sollte man statt der Klasse **String** unbedingt die Klasse **StringBuilder** aus dem Namensraum **System.Text** verwenden, weil hier beim Ändern einer Zeichenkette die relativ aufwändige Erzeugung eines neuen Objekts entfällt.

Ein **StringBuilder**-Objekt kann *nicht* implizit erzeugt werden, jedoch stehen bequeme Konstruktoren zur Verfügung, z.B.:

- **public StringBuilder**()
  Beispiel: `StringBuilder sb = new StringBuilder();`
- **public StringBuilder**(**String** *str*)
  Beispiel: `StringBuilder sb = new StringBuilder("abc");`

Im folgenden Programm wird eine Zeichenkette 10000-mal verlängert, zunächst mit Hilfe der **String**-Klasse, dann mit Hilfe der **StringBuilder**-Klasse:

[1] Die Ergebnisse wurden unter Verwendung der Release-Konfiguration der Entwicklungsumgebung auf einem PC mit Intel - CPU Core i3 550 (3,2 GHz) unter Windows 10 ermittelt.

```
using System;
using System.Text;

class StrBldrDemo {
  static void Main() {
    const int N = 10000;
    String s = "*";
    long vorher = DateTime.Now.Ticks; // Laden von DateTime
    vorher = DateTime.Now.Ticks;
    for (int i = 0; i < N; i++)
      s = s + "*";
    long diff = DateTime.Now.Ticks - vorher;
    Console.WriteLine("Zeit für String-Manipulation:\t\t" +
                      diff/1.0e4 + "\tMillisekunden");

    var t = new StringBuilder("*");
    vorher = DateTime.Now.Ticks;
    for (int i = 0; i < N; i++)
      t.Append("*");
    diff = DateTime.Now.Ticks - vorher;
    Console.WriteLine("Zeit für StringBuilder-Manipulation:\t" +
                      diff/1.0e4 + "\tMillisekunden");
  }
}
```

Die (in Millisekunden gemessenen) Laufzeiten unterscheiden sich erheblich:[1]

```
Zeit für String-Manipulation:           24,5221 Millisekunden
Zeit für StringBuilder-Manipulation:    0,5014  Millisekunden
```

Ein **StringBuilder**-Objekt kennt u.a. die folgenden Methoden und Eigenschaften (alle **public**):

StringBuilder-Member	Erläuterung
Length	enthält die Anzahl der Zeichen
Append()	Das **StringBuilder**-Objekt wird um die String-Repräsentation des Argumentes verlängert, z.B.: `t.Append("*");` Es sind **Append()** - Überladungen für zahlreiche Parameterdatentypen vorhanden.
Insert()	Die String-Repräsentation des Argumentes, das von nahezu beliebigem Typ sein kann, wird vom angesprochenen **StringBuilder**-Objekt an einer bestimmten Stelle eingefügt, z.B.: `sb.Insert(4, 3.14);`
Remove()	Ab einer Startposition wird eine Anzahl von Zeichen entfernt, z.B.: `sb.Remove(100, 500);`
Replace()	Ein Zeichen bzw. eine Zeichenfolge des **StringBuilder**-Objekts wird durch anderes Zeichen bzw. eine andere Zeichenfolge ersetzt, z.B.: `sb.Replace("alt", "neu");`
ToString()	Es wird ein **String**-Objekt mit dem Inhalt des **StringBuilder**-Objekts erzeugt. Dies ist z.B. erforderlich, um ein **StringBuilder**- und ein **String**-Objekt nach Inhalt vergleichen zu können: `Console.WriteLine(s == sb.ToString());`

[1] Gemessen auf einem Rechner mit der Intel-CPU Core i3 550 (3,2 GHz) unter Windows 10-64.

5.5 Enumerationen

Angenommen, Sie entwerfen eine Klasse namens `Person` und wollen auch den Charakter einer Person erfassen. Dabei orientieren sie sich an den vier Temperamentstypen des griechischen Philosophen Hippokrates (ca. 460 - 370 v. Chr.): cholerisch, melancholisch, sanguin, phlegmatisch. Um dieses Merkmal mit seinen vier möglichen Ausprägungen in einer Instanzvariablen Ihrer Klasse `Person` zu speichern, kennen Sie bereits verschiedene Möglichkeiten, z.B.:

- Eine **String**-Variable zur Aufnahme der Temperamentsbezeichnung
  Dabei wird relativ viel Speicherplatz benötigt, und es drohen Fehler durch inkonsistente Schreibweisen, z.B.:

```
public string Temp;
  .  .  .
if (otto.Temp == "Flegmatisch") ...
```

- Eine **int**-Variable mit der Kodierungsvorschrift 0 = cholerisch, 1 = melancholisch etc.
  Es wird wenig Speicher benötigt, allerdings ist der Quellcode nur für Eingeweihte zu verstehen, z.B.:

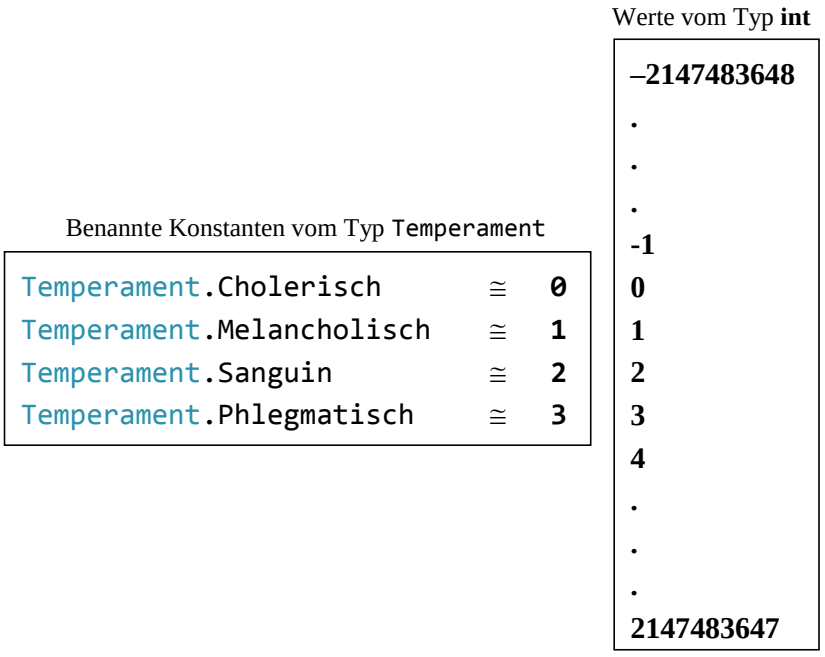
```
public int Temp;
  .  .  .
if (otto.Temp == 3) ...
```

Fehlerhafte Zuweisungen könnte und sollte man bei beiden Lösungsansätzen durch eine sorgfältige Eigenschaftsdefinition verhindern (Abweisen ungeeigneter Werte im **set**-Block, siehe Abschnitt 4.5).

C# bietet mit den **Enumerationen** (**Aufzählungstypen)** eine Lösung, die folgende Vorteile bietet:

- Gut lesbarer Quellcode durch Klartextnamen für die Merkmalsausprägungen
- Falsch geschriebene Klartextnamen werden vom Compiler abgewiesen.
- Geringer Speicherbedarf

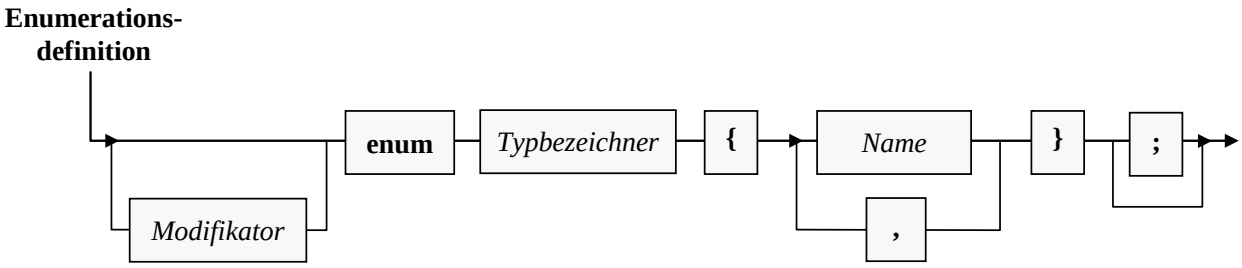
Eine Enumeration basiert auf einem zugrunde liegenden integralen Typ (meist **int**) und enthält eine (meist kleine) Menge von benannten Konstanten dieses Typs, z.B.:



Sofern man auf eine explizite Typkonvertierung verzichtet (siehe unten), können einer Variablen des Enumerationstyps nur die definierten Konstanten zugewiesen werden. Dabei sind nicht die zu-

grunde liegenden Werte (per Voreinstellung 0, 1, 2, …) zu verwenden, sondern die vereinbarten Namen (z.B. `Temperament.Sanguin`).

Bei der Definition eines Aufzählungstyps folgt auf das Schlüsselwort **enum** und den Typbezeichner eine geschweift eingeklammerte Liste mit Namen für die Konstanten:



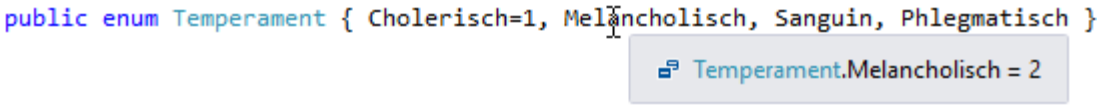
Wie bei Klassen und Strukturen …

- wird die Verfügbarkeit eines Enumerationstyps über Modifikatoren geregelt (Voreinstellung: **internal**, Alternative: **public**, vgl. Abschnitt 4.10),
- ist neben einer Top-Level-Definition auch eine geschachtelte Definition (innerhalb eines anderen Typs) möglich,
- kann optional hinter der schließenden Klammer der Enumerationsdefinition ein Semikolon stehen.

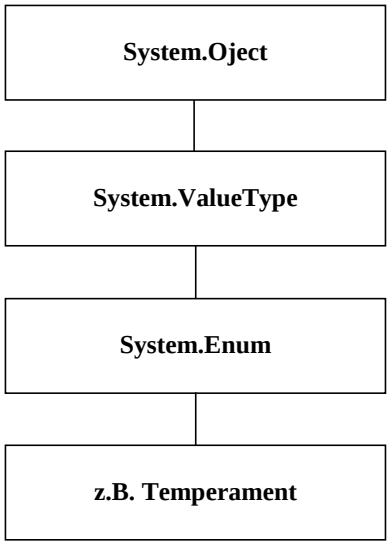
Als Beispiel betrachten wir einen Enumerationstyp zur Erfassung des Charakters von Personen:

```
public enum Temperament {Cholerisch, Melancholisch, Sanguin, Phlegmatisch}
```

Per Voreinstellung stehen die Namen in der Enumerationsdefinition für eine nullbasierte Liste von **int**-Werten, doch lassen sich auch alternative Werte vergeben. Im folgenden Beispiel wird dafür gesorgt, dass die Werteliste bei 1 startet und dann die folgenden natürlichen Zahlen durchläuft. Im Visual Studio wird das numerische Äquivalent zu einer Enumerationskonstanten angezeigt, während sich der Mauszeiger über ihrem Namen befindet, z.B.:



Die Enumerationen sind **Werttypen** und folgendermaßen in das CTS (Common Type System) der .NET - Plattform eingeordnet:



Alternative Abstammungen sind bei Enumerationstypen *nicht* möglich; insbesondere kann man eine Enumeration nicht beerben.

Weil Enumerationskonstanten stets mit dem Typnamen qualifiziert werden müssen, ist einige Tipparbeit erforderlich, die aber mit einem gut lesbaren Quellcode belohnt wird, z.B.[1]

```
public class Person {
   public string Vorname;
   public string Name;
   public int Alter;
   public Temperament Temp;

   public Person(string vorname, string name, int alter, Temperament temp) {
      Vorname = vorname;   Name = name; Alter = alter;
      Temp = temp;
   }
}

class PersonTest {
   static void Main() {
      Person otto = new Person("Otto", "Hummer", 35, Temperament.Sanguin);
      if (otto.Temp == Temperament.Sanguin)
         System.Console.WriteLine("Lustiger Typ!");
}
```

Einer Variable mit Enumerationstyp können leider über eine explizite Typumwandlung neben den benannten Konstanten auch beliebige andere Werte des zugrunde liegenden Typs zugewiesen werden, z.B.:

```
otto.Temp = (Temperament) 13;
```

Daher sollten Enumerations-Instanzvariablen (abweichend von dem obigen schlechten Beispiel) in der Regel gekapselt und nur über eine Eigenschaft mit überwachter Wertzuweisung zugänglich sein, z.B.:

```
Temperament temp;

public Temperament Temp {
   get {
      return temp;
   }
   set {
      if (System.Enum.IsDefined(typeof(Temperament), value))
         temp = value;
   }
}
```

Zum Entscheid über die Gültigkeit eines Wertes lässt sich die statische Methode **IsDefined()** der Klasse **Enum** im Namensraum **System** verwenden. Gegen welchen Enumerationstyp geprüft werden soll, erfährt die Methode **IsDefined()** über das im ersten Aktualparameter zu übergebende **Type**-Objekt, das im Beispiel durch den Operator **typeof** mit dem Enumerationsnamen als Argument geliefert wird.[2]

---

[1] Wir verzichten der Kürze halber bei der Klasse `Person` auf die hier durchaus empfehlenswerte Datenkapselung.

[2] Alternativ zum Operator **typeof** hätte die **Object**-Methode **GetType()** verwendet werden können:

```
value.GetType()
```

5.6 Indexer

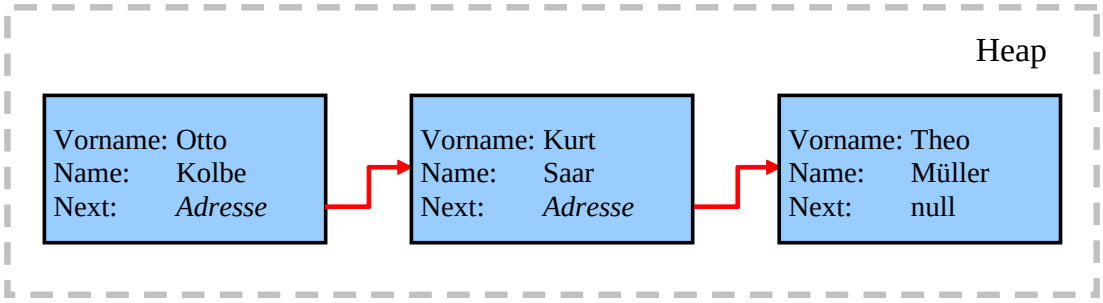
Bei Arrays sowie bei den Klassen **String** und **ArrayList** hat sich der Indexzugriff auf die Elemente eines Objekts per [ ] - Operator als sehr nützlich bis unverzichtbar erwiesen. Um denselben Komfort für eine eigene Klasse oder Struktur zu realisieren, die zur Verwaltung von zahlreichen Elementen desselben Typs dient, muss man ihr einen so genannten **Indexer** spendieren. Analog zur Situation bei einer Eigenschaft handelt es sich auch bei diesem Member letztlich um ein *Paar von Methoden* für den lesenden bzw. schreibenden Zugriff auf ein Element per Indexsyntax.

Ein Indexer kommt also dann in Frage, wenn ein selbst definierter Typ eine Kollektion (z.B. Liste) von Elementen verwaltet und ein wahlfreier Zugriff auf die Elemente ermöglicht werden soll. Im aktuellen Abschnitt geht es also *nicht* um einen neuen Datentyp, sondern um ein zusätzliches Ausstattungsdetail für Klassen und Strukturen.

Als Beispiel betrachten wir eine „Datenbank“, die mit einer so genannten *verketteten Liste* von Objekten der folgenden Klasse `Person` arbeitet:

```
class Person {
   public string Vorname;
   public string Name;
   public Person Next;
   public Person(string vorname, string name) {
      Vorname = vorname;
      Name = name;
   }
}
```

Wir verzichten der Kürze halber bei der Klasse `Person` auf die hier durchaus empfehlenswerte Datenkapselung. Jedes `Person`-Objekt besitzt eine Instanzvariable `Next` zur Aufnahme der Adresse seines Nachfolgers. Wenn man beim Erzeugen eines neuen Objekts dessen Adresse in das `Next`-Feld des bisher letzten Objekts schreibt, entsteht eine (einfach) verkettete Liste. Durch beharrliches Verfolgen der `Next`-Referenzen lässt sich jedes Serienelement erreichen, sofern eine Referenz auf das *erste* Element verfügbar ist. In der folgenden Abbildung ist eine Kette aus drei `Person`-Objekten zu sehen:



Ein Objekt der folgenden Klasse `PersonDB` verwaltet eine verkettete Liste von `Person`-Objekten:

```
class PersonDB {
  int n;
  Person first, last;

  public int Count {
    get {
      return n;
    }
  }
```

```
    public void Add(Person neu) {
    if (neu == null)
      return;
      if (n == 0)
        first = last = neu;
      else {
        last.Next = neu;
        last = neu;
      }
      n++;
    }

    public Person this[int i] {
      get {
        if (i >= 0 && i < n) {
          Person sel = first;
          for (int j = 0; j < i; j++)
              sel = sel.Next;
          return sel;
        } else
          return null;
      }
      set {
        if (i >= 0 && i < n && value != null) {
          if (i == 0) {
            value.Next = first.Next; // Nachfolger des Neulings
            first = value; // Der Neuling wird "Leader"
          } else {
            Person pre = first;
            for (int j = 0; j < i - 1; j++)
              pre = pre.Next;
            value.Next = pre.Next.Next; // Nachfolger des Neulings
            pre.Next = value; // Vorgänger des Neulings
          }
        }
      }
    }
  }
```

Für den lesenden oder schreibenden Zugriff auf das *i*-te Listenelement stellt `PersonDB` einen Indexer mit dem Parameterdatentyp **int** zur Verfügung, der eine `Person`-Referenz liefert (**get**) oder das *i*-te Listenelement durch eine andere `Person`-Referenz ersetzt (**set**).[1]

Einige Regeln für die Indexer-Definition:

- Nach den optionalen Modifikatoren wird der Datentyp angegeben (im Beispiel: `Person`).
- Der Name lautet stets **this**.
- Hinter dem Schlüsselwort **this** wird *eckig* eingeklammert der Indexparameter angegeben.
- Der **set**-Methode wird wie bei Eigenschaften ein impliziter Parameter namens **value** übergeben.

Vom nicht ganz trivialen `PersonDB`-Aufbau merkt ein Anwender dieser Klasse nichts, z.B.:

[1] Durch die Aufmerksamkeit und Hilfsbereitschaft von Herrn Reinhard Dämon konnte ein Fehler im **set**-Teil des Indexers behoben werden. Vielen Dank!

```
using System;
class PersonDbDemo {
  static void Main() {
    var adb = new PersonDB();
    adb.Add(new Person("Otto", "Kolbe"));
    adb.Add(new Person("Kurt", "Saar"));
    adb.Add(new Person("Theo", "Müller"));
    for (int i = 0; i < adb.Count; i++)
      Console.WriteLine($"Nummer {i}: {adb[i].Vorname} {adb[i].Name}");
    Console.WriteLine();
    adb[1] = new Person("Ilse", "Golter");
    for (int i = 0; i < adb.Count; i++)
      Console.WriteLine($"Nummer {i}: {adb[i].Vorname} {adb[i].Name}");
  }
}
```

Das Programm liefert die folgende Ausgabe:

```
Nummer 0: Otto Kolbe
Nummer 1: Kurt Saar
Nummer 2: Theo Müller

Nummer 0: Otto Kolbe
Nummer 1: Ilse Golter
Nummer 2: Theo Müller
```

In den **WriteLine()** - Aufrufen des letzten Beispielprogramms wird übrigens die in Abschnitt 3.2.2.2 beschriebene Zeichenfolgeninterpolation verwendet.

Statt eine eigene Klasse `PersonDB` mit Listenkonstruktion und Indexer zu entwerfen, wird man in der Praxis eine solche Aufgabe übrigens weit ökonomischer unter Verwendung einer generischen Kollektionsklasse aus dem Namensraum **System.Collections.Generic** realisieren (siehe Abschnitt 7.6.2.2). Allerdings gehört der Eigenbau einer verketteten Liste zu einer soliden Programmierer-Grundausbildung, so dass sich der Aufwand des `PersonDB`-Beispiels wohl doch lohnt.

Wenn die Elemente einer Kollektion durch *mehrere* Variablen angesprochen bzw. identifiziert werden können (z.B. Landkreise durch eine laufende Nummer und einen Namen), dann kommt das von C# ermöglichte Überladen des Indexers durch die Verwendung verschiedener Parametertypen gelegen. Im Beispiel könnte man eine Indexer-Überladung mit **string**-Parameter ergänzen, welche die (nach Listenposition) erste Person liefert, deren Vorname mit dem Aktualparameter übereinstimmt:

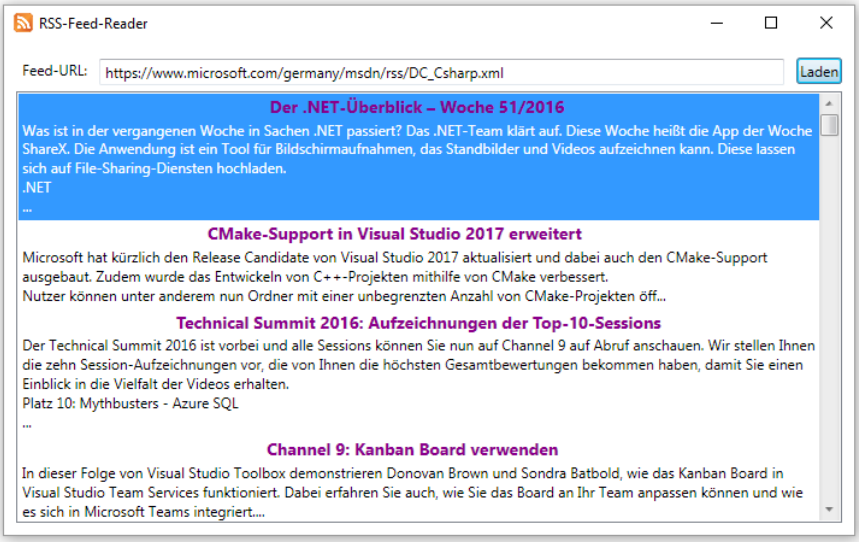
```
public Person this[string vn] {
  get {
       for (int j = 0; j < n; j++)
           if (this[j].Vorname == vn)
               return this[j];
    return null;
  }
}
```

Hier ist ein Einsatz der Indexer-Überladung mit dem Vornamensparameter zu sehen:

```
String s = "Ilse";
Console.WriteLine($"\nName der ersten Person mit dem Vornamen \"{s}\": {adb[s].Name}");
```

5.7 Motivationsnachschub

Eventuell waren die letzten Abschnitte nicht für alle Leser vergnüglich und motivationsfördernd. Damit kein Handtuch fliegt, schieben wir einen Abschnitt ein, der hoffentlich Entspannung bringt und neue Motivation zuführt. Wir erstellen in Anlehnung an einen Artikel von Hajo Schulz in der Computer-Zeitschrift c't (Ausgabe 2010.13, S. 138f) ein Anzeigeprogramm für RSS-Feeds mit frei wählbarer Adresse:



Als *RSS-Feed* (*Really Simple Syndication*) bezeichnet man eine im Internet angebotene, per URL ansprechbare Datei, die neue Beiträge zu einem Thema kurz beschreibt und jeweils einen Link zur Vollinformation bietet.[1] Als Dateiformat dient eine XML-Variante (*eXtended Markup Language*) namens RSS, aktuell in der Version 2.0.

Laut Wikipedia (https://de.wikipedia.org/wiki/RSS_(Web-Feed)) hat eine RSS-Datei der Version 2.0 den folgenden Aufbau:

```
<rss version="2.0">
  <channel>
    <item>
      <title>Titel des Eintrags</title>
      <description>Kurze Zusammenfassung des Eintrags</description>
      <link>Link zum vollständigen Eintrag</link>
      <author>Autor des Artikels, E-Mail-Adresse</author>
      <guid>Eindeutige Identifikation des Eintrages</guid>
      <pubDate>Datum des Items</pubDate>
    </item>
    <item>
      . . .
    </item>
  </channel>
</rss>
```

Von Interesse sind vor allem die `<item>` - Elemente, die jeweils einen Beitrag beschreiben und durch unser Programm formatiert aufgelistet werden sollen.

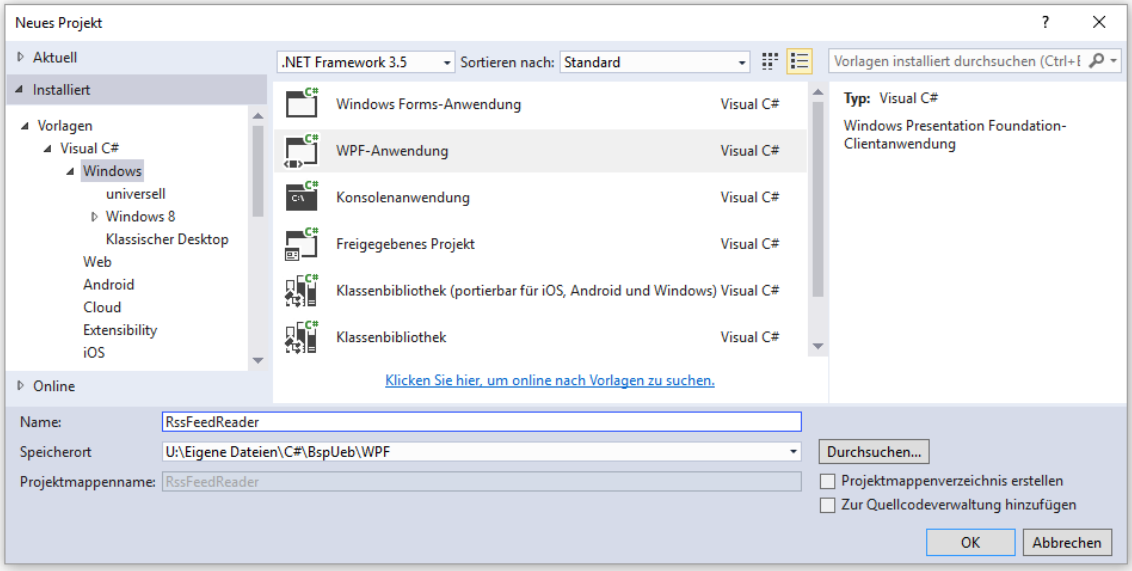
[1] Laut Wikipedia (https://de.wikipedia.org/wiki/Content-Syndication) ist mit *Content-Syndication* im Internet das Zusammenführen der Informationen von verschiedenen Webseiten gemeint.

5.7.1 Projekt anlegen mit Vorlage *WPF - Anwendung*

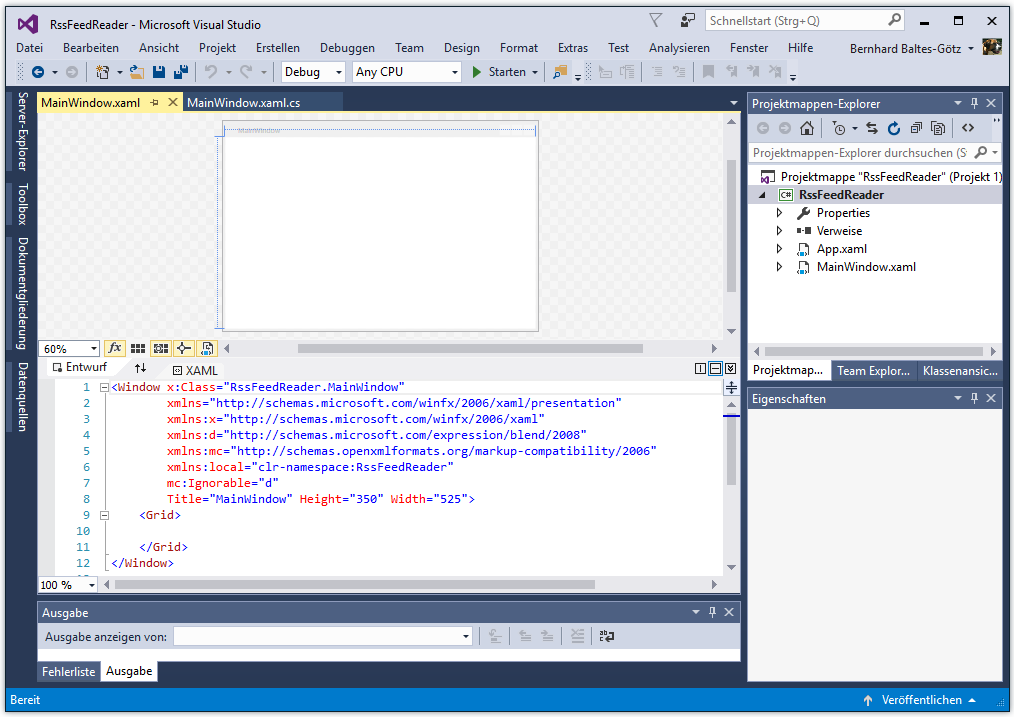
Verwenden Sie nach

Datei > Neu > Projekt

für ein neues Projekt mit dem Namen `RssFeedReader` die Vorlage **WPF - Anwendung**:[1]



Nach einem Mausklick auf **OK** präsentiert die Entwicklungsumgebung im **WPF- bzw. XAML-Designer** einen Rohling für das Fenster der entstehenden Anwendung:



In der oberen Designer-Zone können wir die Bedienoberfläche unseres Programms mit Hilfe von grafischen Werkzeugen erstellen und dazu konfigurierbare Komponenten (Steuerelemente) aus der Toolbox (siehe unten) übernehmen. Wie gleich zu sehen sein wird, gestalten wir dabei den Auftritt von Objekten aus der neuen Klasse `MainWindow` im Namensraum `RssFeedReader`.

---

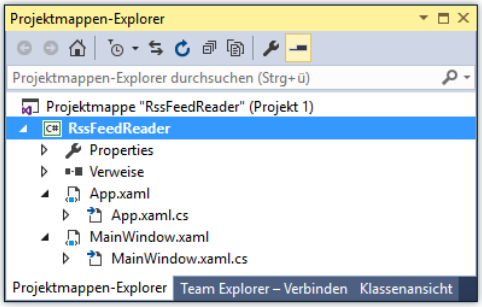
[1] Im Kurs wird das GUI-Design per WPF gegenüber den Alternativen WinForms (veraltet) und WinRT (alias Metro, keine Unterstützung für Windows 7) bevorzugt (zu Details siehe Kapitel 11).

Wie Sie bereits aus Abschnitt 4.11 wissen, besteht ein zentrales Merkmal der WPF-Technologie darin, das GUI-Design einer Anwendung durch eine XML-Spezialisierung namens XAML (*eXtended Application Markup Language*) zu deklarieren. Zu unserem Anwendungsfenster gehört also eine XAML-Datei, die im unteren Teil der Designer-Zone erscheint und initial so aussieht:

```
<Window x:Class="RssFeedReader.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
        xmlns:mc="http://schemas.openxmlformats.org/markup-compatibility/2006"
        xmlns:local="clr-namespace:RssFeedReader"
        mc:Ignorable="d"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Das **Window**-Element definiert ein Anwendungsfenster, also das Erscheinungsbild eines Objekts aus einer Klasse, die von der FCL-Klasse **Window** abstammt. Die initialen Bestandteile des **Window**-Elements wurden schon in Abschnitt 4.11.2 erläutert. Der grafische Fenster-Designer ist ein Werkzeug zur bequemen Bearbeitung der XAML-Datei. Manchmal wird es sich aber als praktisch erweisen, den XAML-Code direkt zu editieren.

Unsere XAML-Datei heißt **MainWindow.xaml** und beschreibt die Oberfläche von Objekten der Klasse `MainWindow`. Zur Definition dieser Klasse trägt auch eine Quellcode-Datei namens **MainWindow.xaml.cs** bei, für die wir als Entwickler verantwortlich sind. Der Projektmappen-Explorer zeigt die XAML- und die zugehörige Quellcode-Datei:



Außerdem trägt zur Definition der Klasse `MainWindow` noch eine zweite Quellcodedatei bei die vom Visual Studio im Hintergrund gepflegt wird (siehe Abschnitt 5.7.4).

Über die Fensterklasse hinaus benötigt eine WPF-Anwendung auch noch eine Anwendungsklasse. Sie stammt von der FCL-Klasse **Application** ab und trägt im Beispiel den Namen `App`. Wie bei der Fensterklasse `MainWindow` sind eine XAML-Deklarationsdatei und eine C# - Quellcodedatei mit einer partiellen Klassendefinition beteiligt. Bei unserem geplanten Beispielprogramm müssen wir uns um beide nicht kümmern. Wer die XAML-Datei **App.xaml** neugierig per Doppelklick auf ihren Eintrag im Projektmappen-Explorer öffnet,

```
<Application x:Class="RssFeedReader.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:RssFeedReader"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

findet im **Application**-Element u.a. die die folgenden Attribute:

- **x:Class**
  Es benennt die zugehörige Klasse `App` samt Namensraum `RssFeedReader`.
- **StartupUri**
  Es nennt die XAML-Datei zum Fenster, das beim Programmstart angezeigt werden soll.

5.7.2 Steuerelemente aus der Toolbox übernehmen

Holen Sie nötigenfalls im Editor die Datei **MainWindow.xaml** in den Vordergrund, und öffnen Sie das **Toolbox**-Fenster mit dem Menübefehl

Ansicht > Toolbox

oder per Mauszeiger durch kurzes Verharren auf der **Toolbox**-Schaltfläche am linken Fensterrand. Erweitern Sie nötigenfalls im **Toolbox**-Fenster die Liste mit den **Häufig verwendeten WPF-Steuerelementen**, und erstellen Sie auf dem Formular folgende Steuerelemente:

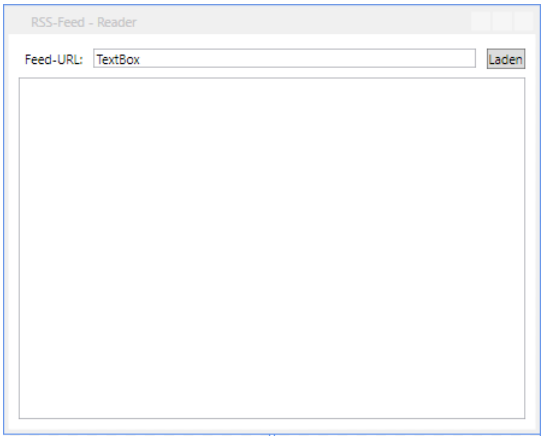
- ein **Label**-Objekt
  Es soll die im Texteingabefeld benötigten Inhalte beschreiben (*Feed-URL:*).
- ein **TextBox**-Objekt
  Hier können die Benutzer die Feed-Adresse eintragen.
- ein **Button**-Objekt
  Damit fordern die Benutzer das Laden einer RSS-Datei an.
- ein **ListBox**-Objekt
  Hier werden die RSS-Items formatiert angezeigt.

Die Übernahme eines Steuerelements aus der Toolbox gelingt ...

- per Doppelklick auf den jeweiligen **Toolbox**-Eintrag
- per Drag & Drop (Ziehen und Ablegen)

5.7.3 Positionen, Größen und sonstige Eigenschaften der Steuerelemente

Nun können Sie wie in einem Grafikprogramm die Positionen und Größen der Fensterbestandteile verändern, um das gewünschte Layout zu erzielen, z.B.:



Für alle Positionen und Größen wird im WPF-Designer die Maßeinheit **DIP** verwendet (*Device Independent Pixel*, dt.: *geräteunabhängige Pixel*). Ein DIP hat eine Breite und Höhe von 1/96 Zoll, so dass z.B. 96 DIP einem Zoll (= 2,54 cm) entsprechen. Bei einer Bildschirmauflösung von 96 DPI (*Dots Per Inch*) ist ein geräteunabhängiges Pixel gerade genauso groß wie ein physikalisches Pixel. Positionen und Größen werden in Variablen vom Typ **double** gespeichert.

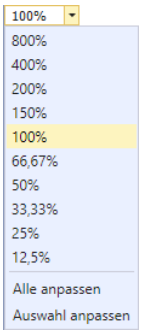
5.7.3.1 Arbeitshilfen

Wir erweitern unser Wissen über die vom WPF-Designer angebotenen Arbeitshilfen im Vergleich zu Abschnitt 4.11.4, ignorieren aber weiterhin die Bedienhilfen mit Bezug zu der noch nicht behandelten WPF-Containertechnik.

5.7.3.1.1 Zoom-Stufe

Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

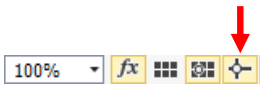
lässt sich eine Zoomstufe wählen:



Bei gedrückter **Strg**-Taste kann man die Zoom-Stufe auch per Mausrad ändern.

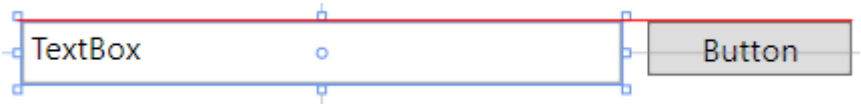
5.7.3.1.2 Ausrichtungslinien

Wenn über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

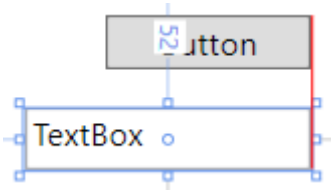


das Andocken an den Ausrichtungslinien aktiviert ist, erscheinen eine rote Linie, wenn

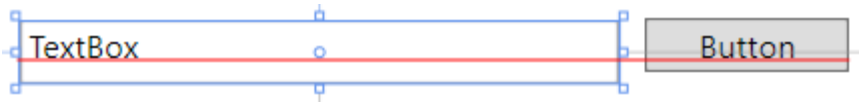
- die *Ränder* von zwei Steuerelementen vertikal

  oder horizontal

  ausgerichtet sind,
- oder wenn die *Textbasislinien* von zwei Steuerelementen vertikal ausgerichtet sind:

5.7.3.1.3 Rasterpositionen und -linien

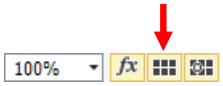
Wenn über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

das Andocken an den Rasterlinien aktiviert ist, wird ...

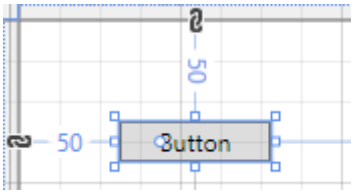
- ein horizontal bewegtes Steuerelement von der nächsten vertikalen Rasterlinie angezogen,
- ein vertikal bewegtes Steuerelement von der nächsten horizontalen Rasterlinie angezogen.

Der Abstand zwischen den Rasterlinien beträgt 5 DIP.

Über die Symbolleiste zwischen Entwurfs- und XAML-Ansicht

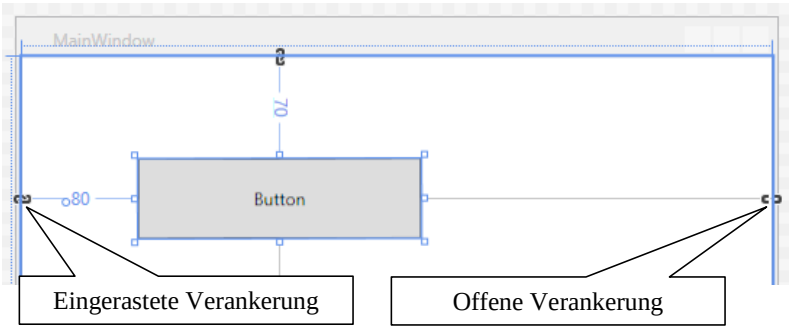


lässt sich eine (ausgedünnte) Version des Rasterliniengitters ein- und ausschalten:



5.7.3.1.4 Verankerung und Abstände

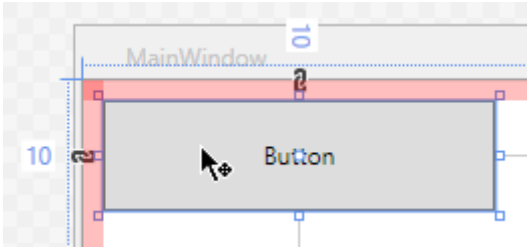
Per Voreinstellung sind die Steuerelemente am linken und am oberen Rand des umgebenden Containers angedockt, was bei einem markierten Steuerelement durch Verankerungssymbole angezeigt wird, z.B.:



Um eine Verankerung vorzunehmen oder aufzuheben, klickt man auf den zugehörigen Verankerungspunkt. Ist beim Lösen einer Verankerung die gegenüberliegende Seite gerade frei, springt die Verankerung dorthin.

Zu den Andockseiten werden die Randabstände numerisch (in DIP) angezeigt.

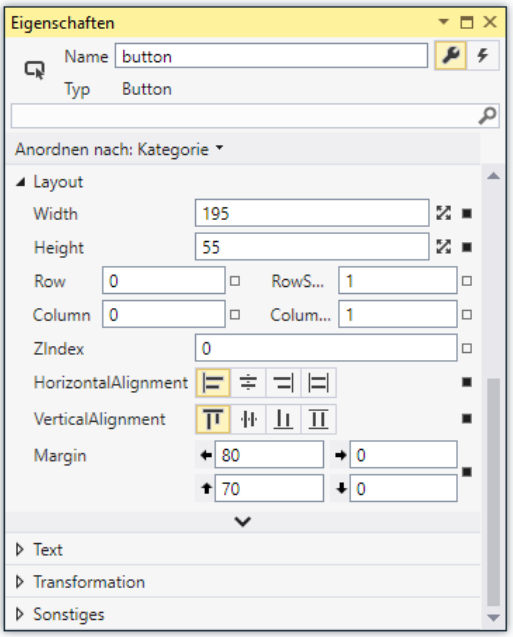
Durch rote Streifen schlägt der WPF-Designer frei zu lassende Zonen mit einer Breite von 10 DIP an den Container-Rändern vor:



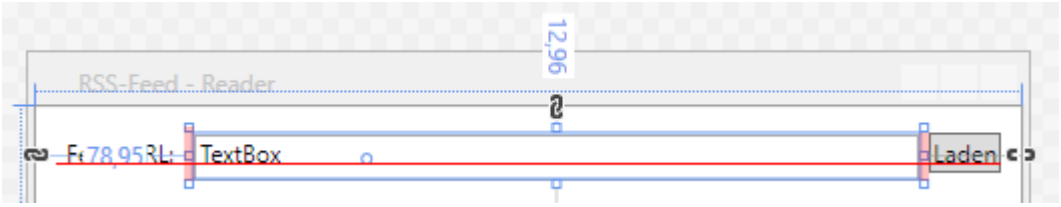
Um bestimmte Randabstände für ein Steuerelement zu realisieren, muss man es nicht unbedingt mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann im XAML-Attribut **Margin** die gewünschten Werte eintragen, z.B.:

```
<Button x:Name="button" Content="Button" VerticalAlignment="Top"
HorizontalAlignment="Left" Margin="80,70,0,0" Width="195" Height="55"/>
```

Eine weitere Möglichkeit zur numerischen Spezifikation bietet die Kategorie **Layout** im **Eigenschaften**-Fenster:



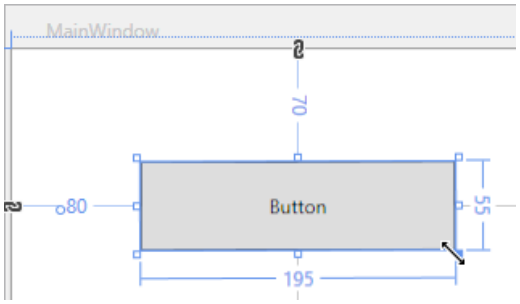
Die vom Designer angezeigten Mindestabstände zu anderen Steuerelementen sind knapp bemessen, z.B.:



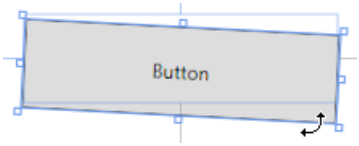
Meist dient es der Übersichtlichkeit, etwas mehr Platz zwischen den Bedienelementen zu lassen.

5.7.3.1.5 Ausdehnungen und Transformationen

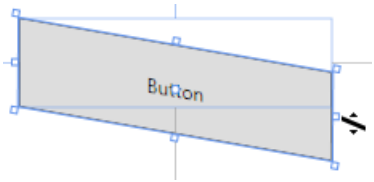
Für das markierte Steuerelement erlauben die aus Grafikprogrammen bekannten Anfasser eine Größenänderung, wobei die aktuellen Werte numerisch (in DIP) angezeigt werden, z.B.:



Es ist auch möglich, ein Steuerelement zu drehen



oder zu neigen:



Um eine bestimmte Breite und Höhe für ein Steuerelement zu realisieren, muss man nicht unbedingt die Anfasser mit Mausgeschicklichkeit zu einer horizontalen bzw. vertikalen Rasterposition bewegen, sondern kann die XAML-Attribute **Width** und **Height** auf die gewünschten Werte setzen, z.B.:

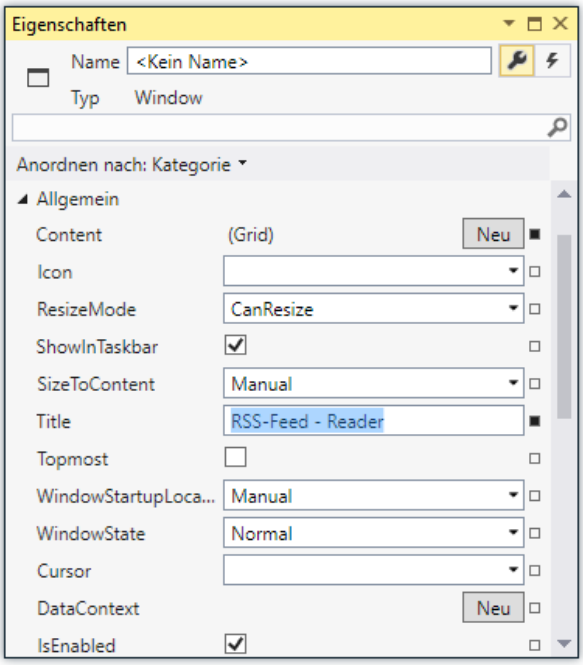
```
<Button x:Name="button" Content="Button" VerticalAlignment="Top"
HorizontalAlignment="Left" Margin="80,70,0,0" Width="195" Height="55"/>
```

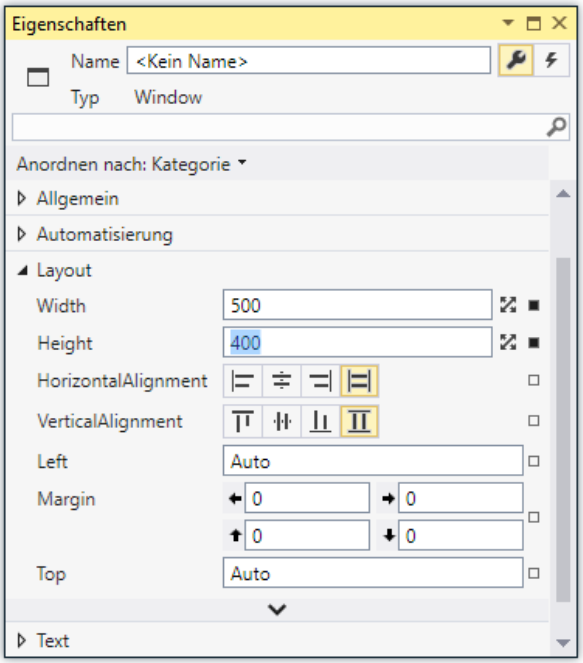
Eine weitere Möglichkeit zur numerischen Spezifikation bietet die Kategorie **Layout** im **Eigenschaften**-Fenster (siehe Bildschirmfoto in Abschnitt 5.7.3.1.4)

5.7.3.2 Arbeitsablauf

5.7.3.2.1 Anwendungsfenster

Markieren Sie das Anwendungsfenster, um dann per **Eigenschaften**-Fenster eine Titelzeilenbeschriftung und die initiale Fenstergröße festzulegen:

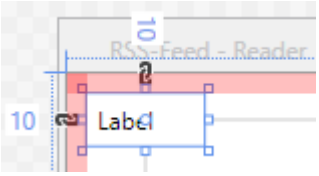




Achten Sie darauf, dass Sie wirklich das Anwendungsfenster erwischen (ein Objekt der Klasse **Window**) und nicht etwa den darin enthaltenen und aus didaktischen Gründen vorläufig ignorierten Container (ein Objekt aus der Klasse **Grid**).

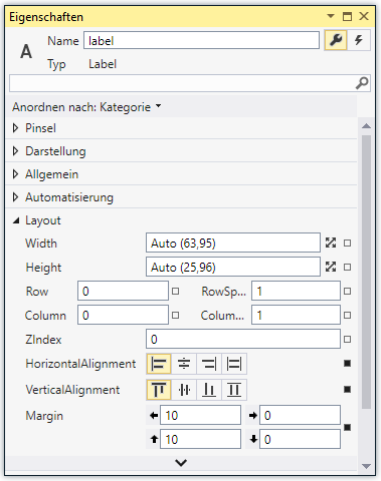
5.7.3.2.2 Label-Objekt

Setzen Sie ein **Label**-Objekt, das den Zweck des Texteingabefelds beschreibt, in die obere linke Ecke des Anwendungsfensters, und akzeptieren Sie die vom Designer vorgeschlagenen Randabstände von 10 DPI zum linken bzw. oberen Fensterrand:



Nach einem *einfachen* Mausklick auf das bereits markierte **Label**-Objekt kann es vor Ort beschriftet werden, z.B. durch „Feed-URL:“.

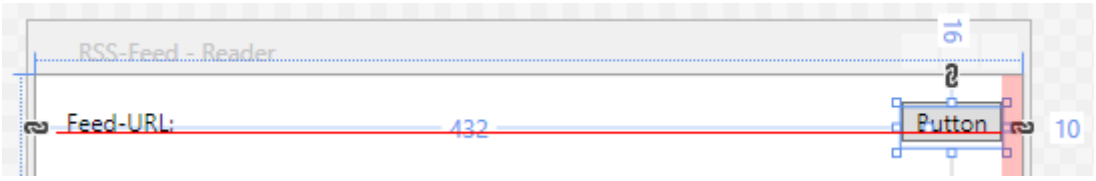
Über Breite und Höhe des **Label**-Objekts darf eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**):



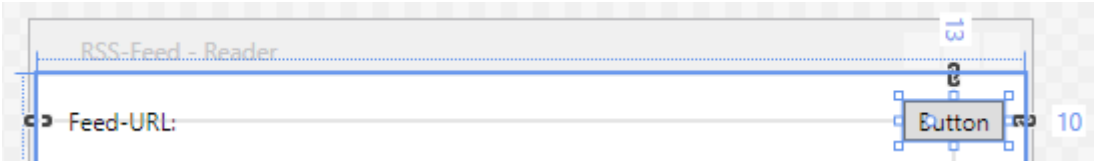
5.7.3.2.3 Button-Objekt

Ergänzen Sie ein **Button**-Objekt, mit dem das Laden einer Feed-Datei angefordert werden soll. Das Objekt sollte so in die rechte obere Fensterecke gesetzt werden, dass es ...

- in vertikaler Richtung die Textbasislinie des **Label**-Objekts übernimmt
- und zum rechten Fensterrand den Standardabstand von 10 DIP einhält.



Das **Button**-Objekt sollte oben und rechts verankert sein, *nicht* jedoch an der linken oder unteren Fensterkante, damit es bei einer Vergrößerung des Fensters *nicht* mitwächst:



Nach einem *einfachen* Mausklick auf das bereits markierte **Button**-Objekt kann es vor Ort beschriftet werden, z.B. durch „Laden“.

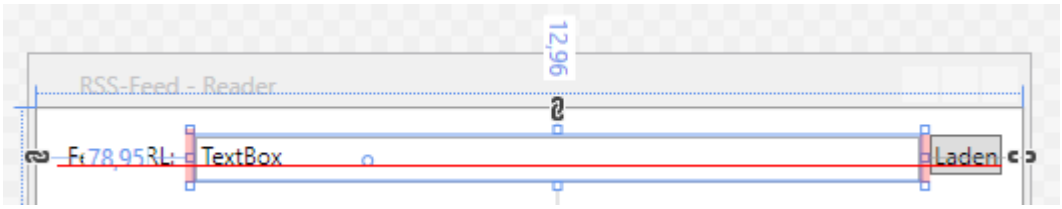
Über Breite und Höhe des **Button**-Objekts darf eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**, siehe oben).

Markieren Sie die **IsDefault**-Eigenschaft der Schaltfläche, so dass diese im laufenden Programm per **Enter**-Taste angesprochen werden kann (Kategorie **Allgemein**).

5.7.3.2.4 TextBox-Objekt

Ergänzen Sie ein **TextBox**-Objekt zur Aufnahme der vom Benutzer gewünschten Feed-URL. Das Objekt sollte so zwischen das **Label**- und das **Button**-Objekt gesetzt werden dass es ...

- in vertikaler Richtung die gemeinsame Textbasislinie der Nachbarn übernimmt
- und horizontal zu beiden Nachbarn den vom Designer vorgeschlagenen Mindestabstand einhält.



Das **TextBox**-Objekt sollte links, oben und rechts verankert sein, damit es bei einer horizontalen (nicht aber bei einer vertikalen) Vergrößerung des Fensters mitwächst:

Sobald ein Steuerelement an zwei *gegenüberliegenden* Seiten verankert ist, wird für seine Ausdehnung in der zugehörigen Richtung (also für seine Breite oder Höhe) vom WPF-Designer die automatische Wertvergabe eingeschaltet.

Wenn Ihnen der Abstand zwischen dem **TextBox**- und dem **Button**-Objekt zu klein erscheint, können Sie per **Eigenschaften**-Fenster den rechten Randabstand des **TextBox**-Objekts fein dosiert vergrößern.

Über die Höhe des **TextBox**-Objekts soll eine am Inhalt orientierte Automatik entscheiden, die im **Eigenschaften**-Fenster über den Symbolschalter aktiviert wird (Kategorie **Layout**).

Auch die Breite des **TextBox**-Objekts sollte automatisch bestimmt werden, damit sie bei einer Änderung der Fensterbreite angepasst werden kann.

Sorgen Sie über den Wert **NoWrap** für die Eigenschaft **TextWrapping** dafür, dass trotz Platznot die Feed-Adresse nicht umgebrochen wird.

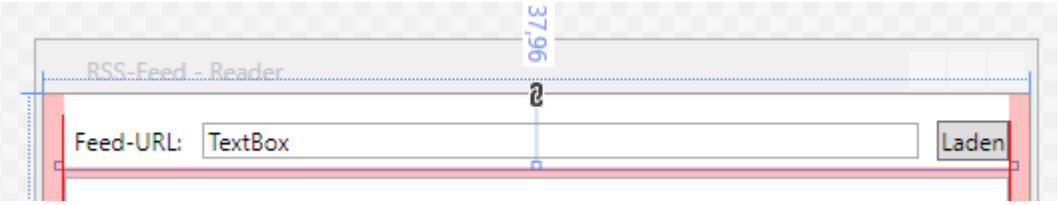
Den initialen Inhalt des **TextBox**-Objekts werden wir per Programm festlegen, so dass wir uns jetzt nicht darum kümmern müssen.

5.7.3.2.5 ListBox-Objekt

Ergänzen Sie ein **ListBox**-Objekt, das zur formatierten Anzeige der Feed-Items dienen soll. Das Objekt sollte so positioniert werden, dass es ...

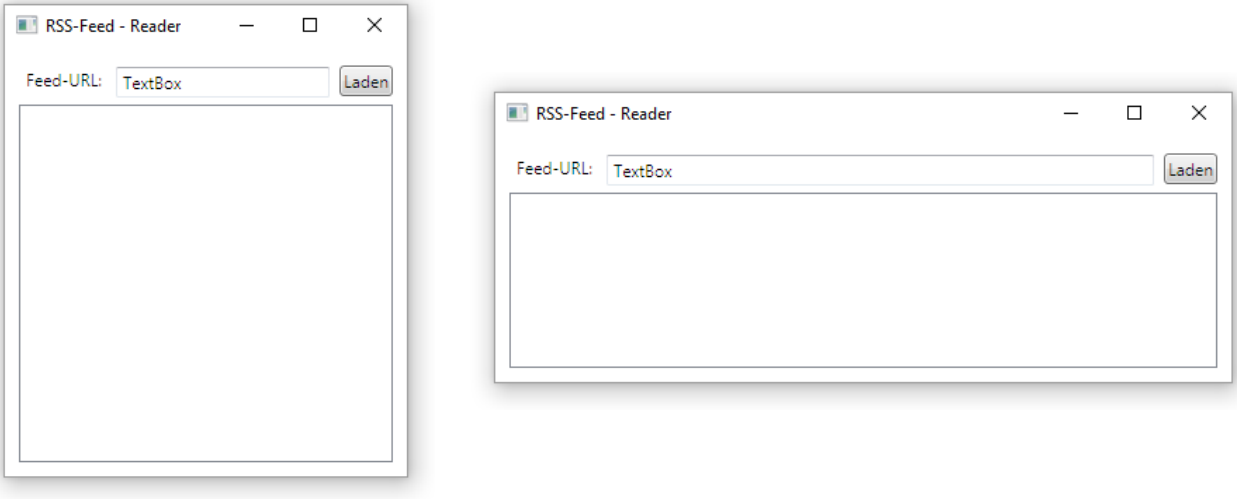
- zum linken, rechten und zum unteren Fensterrand den vom Designer vorgeschlagenen Abstand von 10 DIP einhält

- und nicht allzu dicht unter den drei anderen Bedienelementen sitzt:



Das **ListBox**-Objekt sollte an *allen* Fensterseiten verankert werden, damit es sich bei einer Veränderung der Fenstergröße in horizontaler und in vertikaler Richtung anpasst.

Weil das Programm von Anfang an startfähig ist, kann man sein Verhalten bei variabler Fenstergröße leicht überprüfen, z.B.:



5.7.4 Initialisierung des Anwendungsfensters

Wie Sie bereits aus dem Abschnitt 4.11.6 wissen, erzeugt die Entwicklungsumgebung aufgrund des GUI-Entwurfs per WPF-Designer im Hintergrund Quellcode zu der Fensterklasse `MainWindow`. Weil bei der Klassendefinition sowohl der Entwickler als auch das Visual Studio beteiligt sind, wird der Quellcode auf zwei Dateien verteilt:

- **MainWindow.xaml.cs**
  Hier landen Ihre Beiträge (z.B. die Ereignisbehandlungsmethoden).
- **MainWindow.g.i.cs**
  Der Quellcode in dieser Datei wird bei jedem Erstellen automatisch neu erzeugt, so dass eine Änderung durch den Entwickler sinnlos ist.

Dem C# - Compiler wird durch das Schlüsselwort **partial** in der Klassendefinition signalisiert, dass der Quellcode auf mehrere Dateien verteilt ist. Der `MainWindow`-Quellcode in der Datei **MainWindow.xaml.cs** enthält einen Konstruktor, welcher die Methode **InitializeComponent()** aufruft. Diese wird in der Datei **MainWindow.g.i.cs** von der Entwicklungsumgebung implementiert.

Aufgrund unserer Tätigkeit im WPF-Designer enthält die Fensterklasse `MainWindow` mehrere Objekte anderer Klassen, die Steuerelemente der grafischen Bedienoberfläche repräsentieren. In der Datei **MainWindow.g.i.cs** finden sich die Deklarationen der zugehörigen Instanzvariablen:

```
internal System.Windows.Controls.Label label;
internal System.Windows.Controls.TextBox textBox;
internal System.Windows.Controls.Button button;
internal System.Windows.Controls.ListBox listBox;
```

Damit beim Programmstart im Texteingabefeld die Adresse des MSDN - Feeds (*Microsoft Developer Network*) zu C# erscheint, nehmen wir im `MainWindow`-Konstruktor für die **Text**-Eigenschaft des **TextBox**-Objekts eine entsprechende Initialisierung vor:

```
using System;
  .  .  .
using System.Windows.Shapes;

namespace RssFeedReader {
   /// <summary>
   /// Interaktionslogik für MainWindow.xaml
   /// </summary>
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
         textBox.Text = "https://www.microsoft.com/germany/msdn/rss/DC_Csharp.xml";
      }
   }
}
```

5.7.5 Click-Ereignisbehandlung zum Befehlsschalter (Teil 1)

Wir erstellen eine Ereignisbehandlungsmethode, die durch das Betätigen des Befehlsschalters (per Mausklick oder **Enter**-Taste) ausgelöst wird. Diese Methode soll folgende Leistungen erbringen:

- Unter Verwendung der **Text**-Eigenschaft des **TextBox**-Objekts wird die XML-Datei mit dem gewünschten RSS-Feed aus dem Internet geladen.
- Die Items im RSS-Feed werden an das **ListBox**-Objekt zur formatierten Anzeige übergeben.

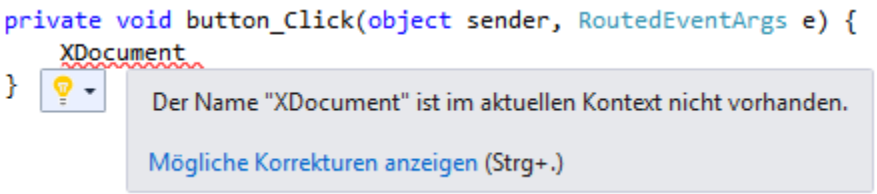
Setzen Sie im WPF-Designer einen Doppelklick auf den Befehlsschalter, so dass die Entwicklungsumgebung in der Datei **MainWindow.xaml.cs** die Instanzmethode `button_Click()` der Klasse `MainWindow` mit leerem Rumpf anlegt

```
private void button_Click(object sender, RoutedEventArgs e) {

}
```

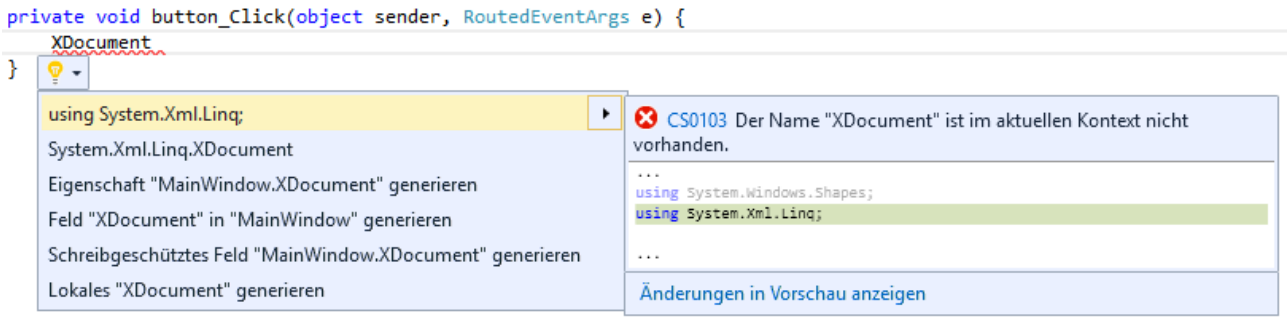
und die Quellcodedatei im Editor öffnet.

Wir befördern den Inhalt der vom Benutzer benannten RSS-Datei in ein Objekt der FCL-Klasse **XDocument**. Für das nicht triviale, mit einem Internetzugriff verbundene Laden der RSS-Datei verwenden wir die statische **XDocument**-Methode **Load()** und übergeben die **Text**-Eigenschaft (mit Datentyp **String**) des **TextBox**-Bedienelements als Aktualparameter.

Der optimistisch in der Methode `button_Click()` als Datentyp für eine lokale Variable eingetippte Klassennamen **XDocument** wird von der Entwicklungsumgebung rot unterschlängelt:



Wenn der Mauszeiger über der Unterschlängelung verharrt, erscheinen neben einer genauen Fehlerbeschreibung auch Unterstützungsangebote. Nach einem Mausklick auf den Pfeil neben der Glühbirne oder auf den Link **Mögliche Korrekturen anzeigen** wird u.a. vorgeschlagen, entweder den Namensraum **System.Xml.Linq**, in dem sich die Klasse **XDocument** befindet, per **using**-Direktive zu importieren, oder dem Klassenamen einen Namensraumpräfix voranzustellen:



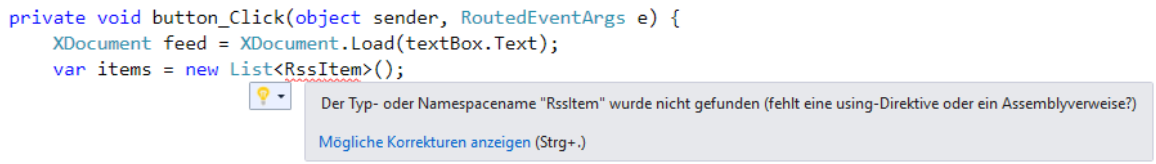
Weil die Klasse **XDocument** in unserem Programm mindestens zweimal angesprochen werden soll, lassen wir eine **using**-Direktive zu dieser Klasse einfügen (1. Vorschlag). Nun komplettieren wir mit Intellisense-Unterstützung die Anweisung durch den Variablennamen `feed` und den eben beschriebenen Aufruf der statischen **XDocument**-Methode **Load()**:

```
XDocument feed = XDocument.Load(textBox.Text);
```

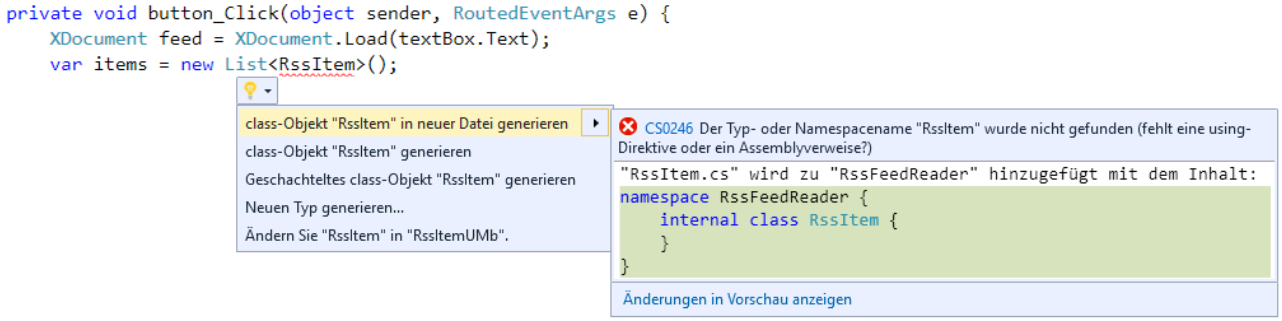
Um ein **ListBox**-Steuerelement zu füllen, kann man seiner Eigenschaft **ItemsSource** ein Objekt aus einer Klasse zuweisen, welche die Schnittstelle **IEnumerable** aus dem Namensraum **System.Collections** erfüllt. Eine Klasse erfüllt eine Schnittstelle, wenn sie alle von der Schnittstelle vorgeschriebenen Methoden besitzt. Im Fall der Schnittstelle **IEnumerable** wird von einer Klasse verlangt, dass sie Elemente mit einem identischen Typ verwalten und sukzessive ausliefern kann, so dass man z.B. in einer **foreach**-Schleife über die Elemente iterieren kann. Wir werden uns in Kapitel 8 mit Schnittstellen im Allgemeinen und in Abschnitt 8.6 mit der Schnittstelle **IEnumerable** im Speziellen beschäftigen.

Für unsere Zwecke eignet sich als **ListBox**-Datenquelle ein Objekt der Klasse **List<`RssItem`>**. Der (noch) ungewohnte Klassenname kommt zustande, weil wir mit der *generischen* Kollektionsklasse **List<T>** arbeiten, die einen größendynamischen Behälter für Elemente mit einem festen, erst beim Erzeugen des Containers festzulegenden Typs darstellt. In unserem Fall treten RSS-Items als Elemente auf, und die modellierende Klasse mit dem Namen `RssItem` muss erst noch definiert werden. Mit generischen Typen und mit Kollektionen werden wir uns bald in Kapitel 7 beschäftigen, so dass es motivationspsychologisch zu begrüßen ist, wenn Ihnen jetzt relevante Beispiele für diese Programmiertechniken begegnen.

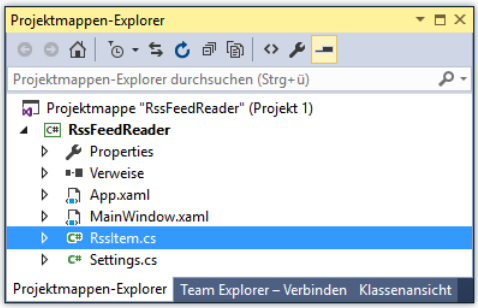
Um die Hilfsbereitschaft und Kompetenz der Entwicklungsumgebung zu testen, erzeugen wir mutig ein Objekt namens `items` aus der Klasse **List<`RssItem`>** unter Verwendung der noch fehlenden Klasse `RssItem`:



Zum erwarteten Fehlerindikator fordern wir die Korrekturvorschläge an



und wählen den ersten. Daraufhin erstellt das Visual Studio eine Klasse mit gewünschten Namen `RssItem` in einer eigenen Datei, die im Projektmappen-Explorer erscheint:



In der automatisch erstellten Klassendefinition

```
namespace RssFeedReader {
   internal class RssItem {
   }
}
```

müssen wir noch Eigenschaften für den Titel, die Kurzbeschreibung und den URL eines RSS-Items ergänzen, was gleich mit Hilfe der Entwicklungsumgebung geschehen soll.

Nun widmen wir uns der Aufgabe, die im **XDocument**-Objekt `feed` enthaltenen RSS-Items zu extrahieren und als Objekte der neuen Klasse `RssItem` in das Kollektionsobjekt `items` vom Typ **List**<`RssItem`> einzufüllen.[1] Die **XDocment**-Instanzmethode **Descendants()** liefert ein Objekt, das die generische Schnittstelle **IEnumerble<XElement>** erfüllt und alle XML-Elemente mit dem per Aktualparameter angegebenen Namen aus der geladenen Feed-Datei enthält:

```
IEnumerable<XElement> xDocItems = feed.Descendants("item");
```

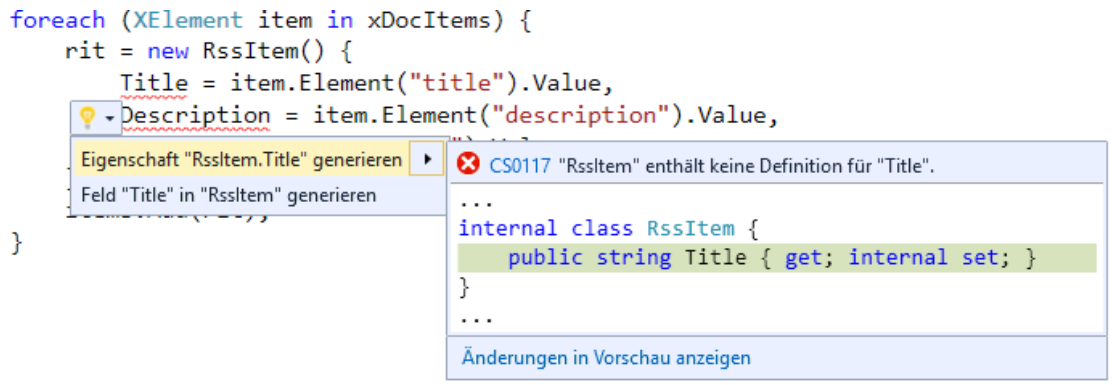
In einer **foreach**-Schleife erstellen wir aus jedem **XElement**-Objekt ein `RssItem`-Objekt und nehmen dieses per **Add()** - Methode in die Kollektion `items` vom Typ **List**<`RssItem`> auf:

```
RssItem rit;
foreach (XElement item in xDocItems) {
   rit = new RssItem() {
      Title = item.Element("title").Value,
      Description = item.Element("description").Value,
      Url = item.Element("link").Value
   };
   items.Add(rit);
}
```

Statt eines explizit definierten initialisierenden `RssItem`-Konstruktors wird hier übrigens die in Abschnitt 4.4.3.2 beschriebene Objektinitialisierung verwendet.

In der **foreach**-Schleife werden Subelemente eines `<item>` - Elements sowie korrespondierende Eigenschaften der Klasse `RssItem` verwendet, welche dort noch nicht definiert sind (`Title`, `Description` und `Url`). Unsere Entwicklungsumgebung erkennt das Problem und schlägt geeignete Maßnahmen vor, z.B.:

[1] Dazu bietet sich eigentlich eine Technik namens **LINQ to XML** an, die uns aber noch nicht zur Verfügung steht. Wir werden uns auf bereits bekannte Programmiertechniken beschränken und dabei nicht schlecht fahren.



Die generierte Definition zur Eigenschaft `Title`

```
internal class RssItem {
  public string Title { get; internal set; }
}
```

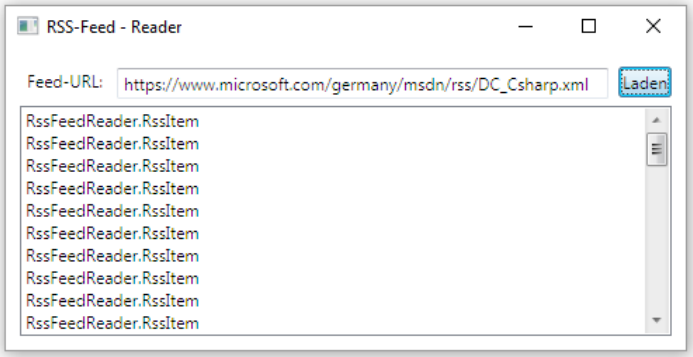
ist durchaus vollständig, weil der C# - Compiler seit der Version 3.0 das zugehörige private *backing field*Feld automatisch erstellen kann (siehe Abschnitt 4.5.2 zu automatisch implementierten Eigenschaften). Analog erhalten wir ohne große Anstrengungen die komplette Definition der Klasse `RssItem`:

```
namespace RssFeedReader {
  internal class RssItem {
    public string Title { get; internal set; }
    public string Description { get; internal set; }
    public string Url { get; internal set; }
  }
}
```

Nun können wir am Ende der Ereignisbehandlungsmethode `button_Click()` der **ListBox**-Eigenschaft **ItemsSource** das Kollektionsobjekt `items` mit den `RssItem`-Elementen zuweisen:

```
private void button_Click(object sender, RoutedEventArgs e) {
  XDocument feed = XDocument.Load(textBox.Text);
  var items = new List<RssItem>();
  IEnumerable<XElement> xDocItems = feed.Descendants("item");
  RssItem rit;
  foreach (XElement item in xDocItems) {
    rit = new RssItem() {
      Title = item.Element("title").Value,
      Description = item.Element("description").Value,
      Url = item.Element("link").Value
    };
    items.Add(rit);
  }
  listBox.ItemsSource = items;
}
```

Ein Startversuch mit dem Laden des voreingestellten RSS-Feeds zeigt, dass wir auf einem guten Weg sind:



5.7.6 Formatierung der Listenelemente per DataTemplate-Objekt

Bislang zeigt das **ListBox**-Objekt zu jedem RSS-Item lediglich den Datentyp an (offenbar die Produktion der Methode **ToString()**, welche die Klasse `RssItem` von der Urahnklasse **Object** gerbt hat). Um zu einer informativen und optisch attraktiven Anzeige zu kommen, verwenden wir ein geeignet konfiguriertes Objekt der Klasse **DataTemplate**, das der **ListBox**-Eigenschaft **ItemTemplate** als Wert zugewiesen wird. Dies gelingt im Visual Studio am besten durch direktes Editieren der XAML-Datei zum Anwendungsfenster:

```
<ListBox x:Name="listBox" Margin="10,40.96,10,10">
  <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
             TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
             Foreground="DarkMagenta" />
          <TextBlock Text="{Binding Path=Description}" Margin="1,1,1,4"
             TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Im XAML-Element **ListBox** wird die sogenannte *Eigenschaftselementsyntax* (vgl. Abschnitt 11.3.2.3) verwendet, um die **ListBox**-Eigenschaft **ItemTemplate** zu versorgen. Dieser Eigenschaft wird ein Objekt der Klasse **System.Windows.DataTemplate** zugewiesen, das in einem eigenen XAML-Element deklariert wird.

Das **DataTemplate**-Objekt verwendet einen Layoutcontainer vom Typ **StackPanel** (siehe Abschnitt 11.6.3) mit vertikaler Orientierung, um zwei **TextBlock**-Objekte übereinander zu präsentieren.

Ein **TextBlock**-Objekt erhält seine Daten über die **Datenbindungstechnologie**, die zu den WPF-Glanzlichtern gehört und später noch ausführlich zu behandeln ist. Durch die folgende Attributsyntax mit einer sogenannten *Markup-Erweiterung* (vgl. Abschnitt 11.3.2.3.6)

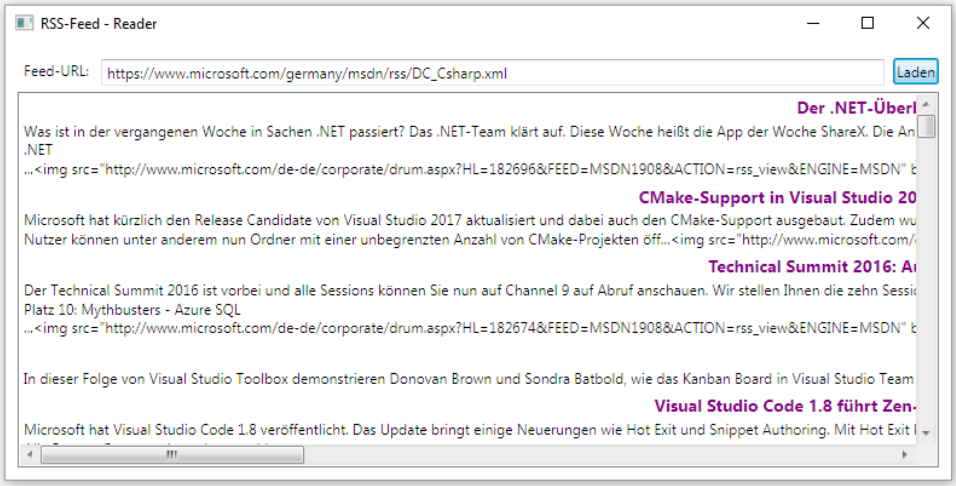
```
Text="{Binding Path=Title}"
```

wird ein Objekt der Klasse **Binding** beauftragt, aus dem aktuellen Element der zum **ListBox**-Objekt gehörigen Kollektion den Wert der Eigenschaft `Title` zu extrahieren. Das Kollektionsobjekt wird in der **Click**-Ereignismethode zum Befehlsschalter (siehe Abschnitt 5.7.5) erzeugt und der **ListBox**-Eigenschaft **ItemsSource** zugewiesen:

```
listBox.ItemsSource = items;
```

Die weiteren **TextBlock** - XAML-Attribute dienen der Formatierung. Wenn ihnen z.B. die Farbe **DarkMagenta** für den Titel missfällt, können Sie einen alternativen Farbnamen aus der Enumeration **ConsoleColor** (Namensraum **System**) verwenden.

Der bei **ListBox**-Steuerelementen per Voreinstellung vorhandene *horizontale* Rollbalken ist bei unserem RSS-Feed - Reader für eine unpraktische Textpräsentation verantwortlich



und wird daher über das folgende Attribut zum **ListBox**-Element der XAML-Datei **MainWindow.xaml** abgeschaltet:

```
<ListBox . . . ScrollViewer.HorizontalScrollBarVisibility="Disabled">
   . . .
</ListBox>
```

HorizontalScrollBarVisibility ist eine sogenannte *Abhängigkeitseigenschaft* der Klasse **ScrollViewer**, und Sie werden noch erfahren, warum man mit ihrer Hilfe für ein Objekt der Klasse **ListBox** den horizontalen Rollbalken beeinflussen kann.

Außerdem ist noch ein Schönheitsfehler festzustellen: Die in den `<description>` - Elementen der RSS-Datei vorhandenen `<img>` - Elemente werden natürlich nicht korrekt interpretiert und sollten daher entfernt werden. Eine bei **StackOverflow**, einem professionellen Forum zu Fragen der Softwareentwicklung, entdeckte Lösung von *Jossef Harush* erledigt diese Aufgabe mit Hilfe der statischen Methode **Replace()** aus der Klasse **Regex** im Namensraum **System.Text.RegularExpressions**:[1]

```
Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                            String.Empty, RegexOptions.IgnoreCase).Trim(),
```

Die zu suchende und durch **String.Empty** zu ersetzende Zeichenfolge wird über einen sogenannten *regulären Ausdruck* definiert:[2]

- <
  Das erste Zeichen muss eine öffnende spitze Klammer sein.
- **[^>]***
  Dann dürfen beliebig viele Zeichen folgen (Quantor ***), die nicht mit der schließenden spitzen Klammer identisch sind (Negative Zeichenauswahl **[^>]**).
- >
  Das letzte Zeichen muss eine schließende spitze Klammer sein.

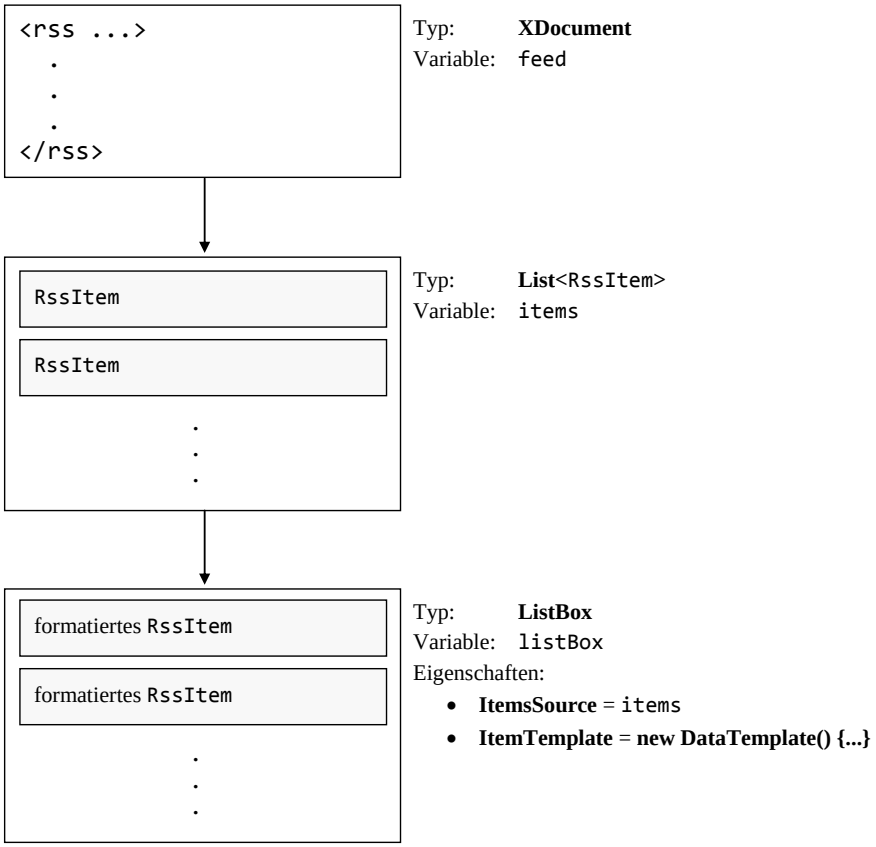
---

1 http://stackoverflow.com/questions/23040422/delete-img-path-from-description
2 Siehe z.B. https://de.wikipedia.org/wiki/Regul%C3%A4rer_Ausdruck

In der folgenden Abbildung wird skizziert, wie die in den Abschnitten 5.7.5 und 5.7.6 beschriebenen Typen

- **XDocument**
- `RssItem`
- **List**`<RssItem>`
- **ListBox**
- **DataTemplate**

zusammenarbeiten, um aus einer RSS-Datei ein **ListBox**-Steuerelement mit formatierten RSS-Items zu erstellen:



5.7.7 Klick-Ereignisbehandlung zum Befehlsschalter (Teil 2)

Weil beim Feed-Abruf und beim Füllen der **ListBox** einiges schief gehen kann, verwenden wir eine **Try-catch** - Anweisung im Vorgriff auf Kapitel 12 und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche auf die **Message**-Eigenschaft des **Exception**-Objekts zugreift:

```
try {
   XDocument feed = XDocument.Load(textBox.Text);
   var items = new List<RssItem>();
   IEnumerable<XElement> xDocItems = feed.Descendants("item");
   RssItem rit;
   foreach (XElement item in xDocItems) {
      rit = new RssItem() {
         Title = item.Element("title").Value,
         Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                                     String.Empty, RegexOptions.IgnoreCase).Trim(),
         Url = item.Element("link").Value
      };
      items.Add(rit);
   }
   listBox.ItemsSource = items;
} catch (Exception ex) {
   listBox.ItemsSource = null;
   MessageBox.Show(this, ex.ToString(), ex.Message);
}
```

Das Laden eines Feeds kann etliche Sekunden dauern. Um den Benutzer darüber zu informieren, dass sein Auftrag in Bearbeitung ist, zeigen wir beim Aufruf der Ereignismethode den Wait-Cursor

an

```
Cursor oldCursor = this.Cursor;
Cursor = Cursors.Wait;
```

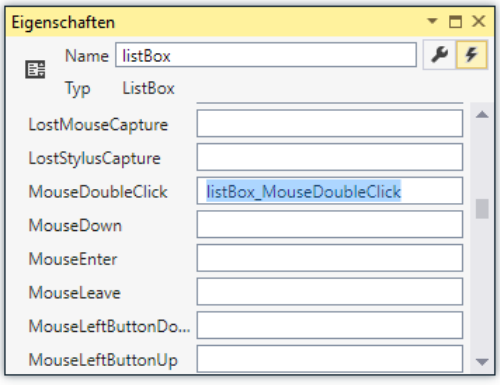
und reaktivieren vor dem Verlassen der Methode den vorherigen Cursor. Damit dieses Restaurieren unter allen Umständen, insbesondere auch nach einem Fehler im **try**-Block, ausgeführt wird, setzen wir die erforderliche Anweisung in einen **finally**-Block, der die **try-catch** - Anweisung erweitert:

```
finally {
   Cursor = oldCursor;
}
```

5.7.8 Doppelklick-Ereignisbehandlung zum ListBox-Steuerelement

Es wäre nett, wenn nach dem Doppelklick auf ein RSS-Item die zugehörige Vollinformation vom bevorzugten Browser angezeigt würde. Um dies zu erreichen, erstellen wir eine Behandlungsmethode zum **MouseDoubleClick**-Ereignis des **ListBox**-Steuerelements:

- Markieren Sie im WPF-Designer das **ListBox**-Steuerelement.
- Wechseln Sie im **Eigenschaften**-Fenster per Mausklick auf das Symbol  zur Anzeige der Ereignisse.
- Setzen Sie einen Doppelklick auf das Ereignis **MouseDoubleClick**:



- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die Instanzmethode `listBox_MouseDoubleClick()` zu unserer Fensterklasse `MainWindow` angelegt.

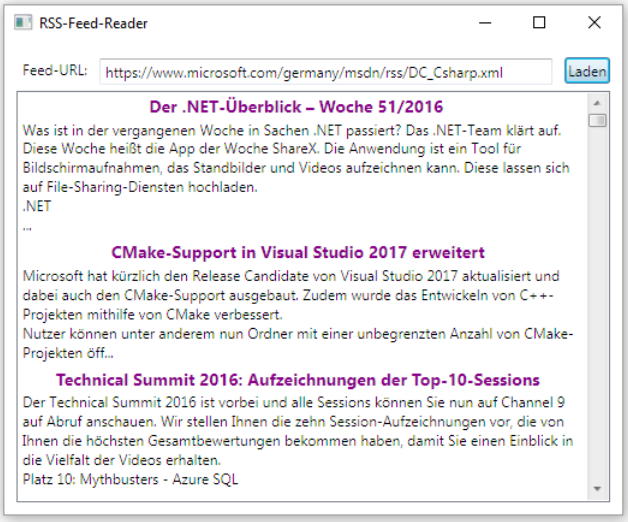
Im Rumpf dieser Methode verwenden wir die statische Methode **Start()** der Klasse **Process** im Namensraum **System.Diagnostics** dazu, das Betriebssystem zu beauftragen, mit einem geeigneten Programm den Link in demjenigen `RssItem`-Objekt zu öffnen, das aktuell im **ListBox**-Objekt ausgewählt ist. Die **ListBox**-Eigenschaft **SelectedItem** besitzt den Typ **Object**, so dass eine explizite Typumwandlung erforderlich ist:

```
RssItem item = (RssItem)listBox.SelectedItem;
```

Weil beim Aufruf eines externen Programms einiges schief gehen kann, verwenden wir erneut im Vorgriff auf Kapitel 12 eine **try-catch** - Anweisung und zeigen ggf. im **catch**-Block eine Fehlermeldung an, welche auf die **Message**-Eigenschaft des **Exception**-Objekts zugreift:

```
private void listBox_MouseDoubleClick(object sender, MouseButtonEventArgs e) {
   RssItem item = (RssItem)listBox.SelectedItem;
   // Exist. SelectedItem? Liefert seine Url-Eigenschaft einen nicht-leeren String?
   if (item != null && !String.IsNullOrEmpty(item.Url))
      try {
         System.Diagnostics.Process.Start(item.Url);
      } catch (Exception ex) {
         MessageBox.Show(this, ex.ToString(), ex.Message);
      }
}
```

Nun ist unser Feed-Reader in einem brauchbaren Zustand:



Die teilweise unschönen Zeilenumbrüche und die Ellipsen (...) sind so in der voreingestellten RSS-Datei vorhanden.

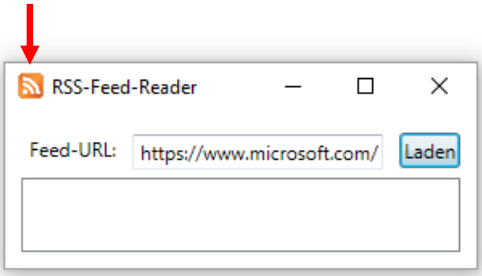
5.7.9 Symbol für das Programm und sein Fenster

Abschließend sollen das Programm und sein Fenster noch ein attraktives Symbol erhalten. Wir beziehen von der Wikipedia-Webseite

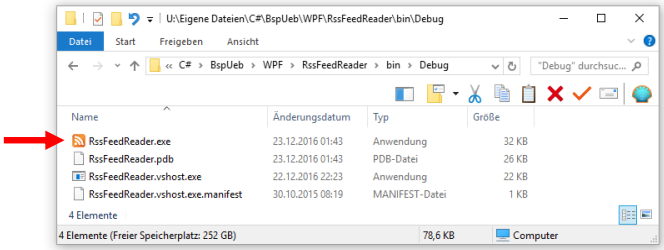
https://de.wikipedia.org/wiki/Datei:Feed-icon.svg

eine Bitmap-Datei mit einem RSS-Emblem im PNG-Format (***P**ortable **N**etwork **G**raphics*) mit einer 500 × 500 Pixelmatrix:

Daraus lässt sich eine Windows-Symboldatei (Namenserweiterung **.ico**) erstellen, die sich für das Fenstersymbol

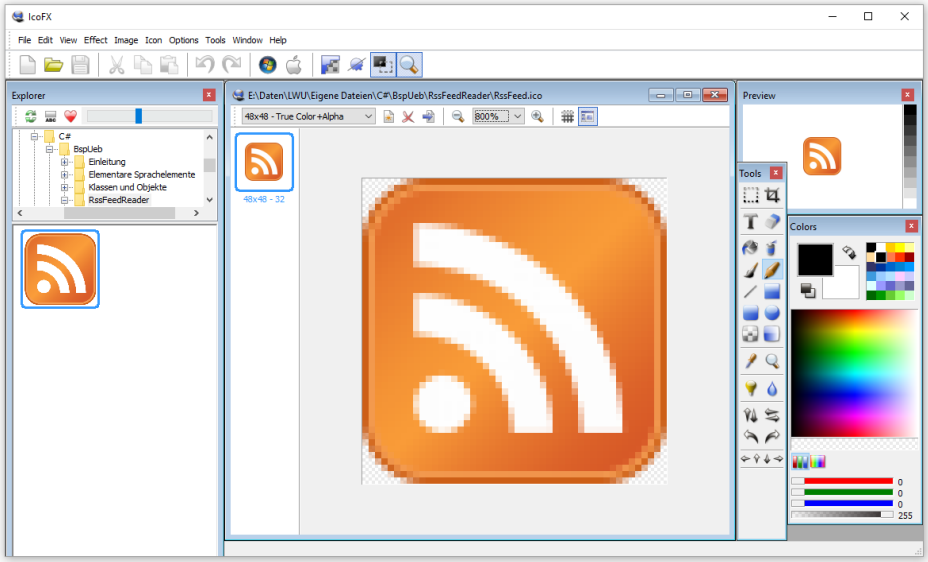


und für das Anwendungssymbol



eignet. Während Arbeitsergebnis von **IrfanView** 4.42 Fehler aufwies, hat das Konvertieren mit **IcoFX** 1.6.4 geklappt.[1]

[1] Die für unsere Zwecke ausreichende Version 1.6.4 von **IcoFX** ist Freeware, während die auf der Hersteller-Webseite (http://icofx.ro/) verfügbare aktuelle Version kostenpflichtig ist. Die Freeware-Version wird auf diversen Webseiten (z.B. Chip, Computerbild) weiterhin angeboten, wobei oft um das eigentliche Installationsprogramm ein „Download-Manager“ gestrickt ist, der explizit daran gehindert werden muss, zusätzlich „Sponsoren-Software“ zu installieren.

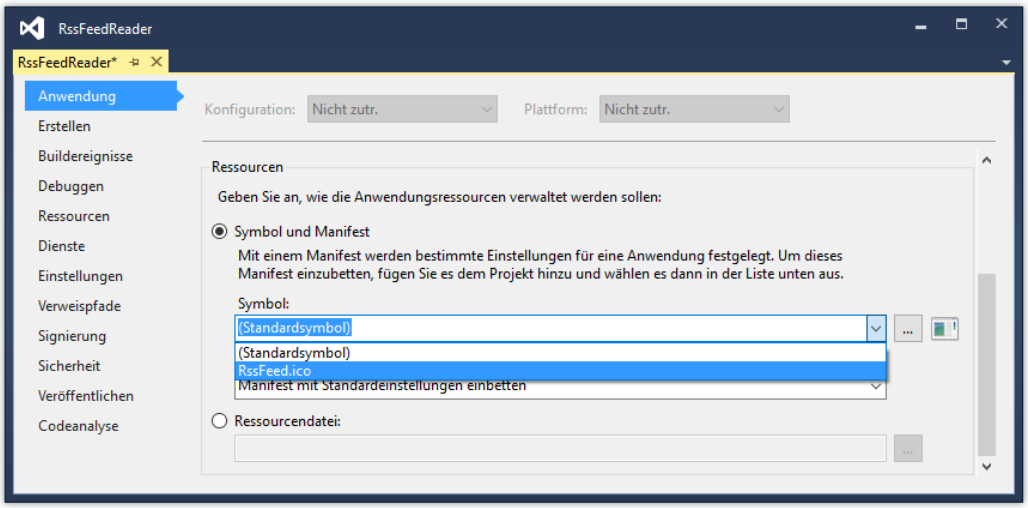


Gehen Sie im Visual Studio folgendermaßen vor, um aus der **ico**-Datei das Anwendungssymbol und das Fenstersymbol zu beziehen:

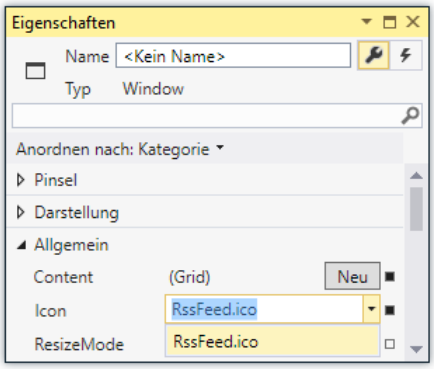
- Kopieren Sie die ICO-Datei in den Projektordner.
- Nehmen Sie die ICO-Datei in das Projekt auf (Item **Hinzufügen > Vorhandenes Element** aus dem Kontextmenü zum Projekteintrag im Projektmappen-Explorer).
- Öffnen Sie das Fenster mit den Projekteigenschaften über den Menübefehl

  Projekt > Eigenschaften

  und wählen Sie im Bereich **Anwendung** das **Symbol**:

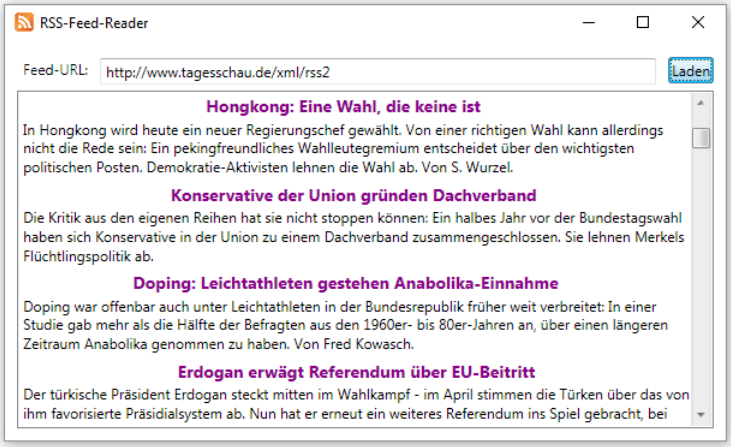


- Markieren Sie im WPF-Designer das Anwendungsfenster, und wählen Sie im **Eigenschaften**-Dialog per Drop-Down - Menü zur Eigenschaft **Icon** die Datei **RssFeed.ico**:



5.7.10 Selbstkritik und Ausblick

Unser Programm präsentiert beliebige gültige RSS-Dateien (mit beliebiger Namenserweiterung) recht ansehnlich, z.B.:



Allerdings lässt sich das Anwendungsfenster nicht verschieben, während eine RSS-Datei geladen wird. Offenbar ist der UI-Thread ausgelastet und kann nicht auf Anweisungen zur Änderungen seines Auftritts reagieren. Mit der in Kapitel 15 behandelten Multithread-Programmierung können wir solche Probleme vermeiden (siehe speziell Abschnitt 15.2.3).

Den vollständigen Projektordner mit dem RSS-Feed -Reader auf dem aktuellen Entwicklungsstand finden Sie hier:

...\BspUeb\WPF\RssFeedReader\BlockedUI

5.8 Übungsaufgaben zu Kapitel 5

1) Im folgenden Programm wird den beiden **object**-Variablen `o1` und `o2` derselbe **int**-Wert zugewiesen. Wieso haben die beiden Variablen anschließend nicht denselben Inhalt?

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  object o1 = 1;` `  object o2 = 1;` `  Console.WriteLine(o1 == o2);` `  }` `}`	`False`

2) Erstellen Sie ein Programm, das 6 Lottozahlen (von 1 bis 49) zieht und sortiert ausgibt. Vermutlich werden Sie für die Lottozahlen einen eindimensionalen **int**-Array verwenden. Dieser lässt sich mit der statischen Methode **Sort()** aus der Klasse **Array** im Namensraum **System** bequem sortieren.

3) Erstellen Sie ein Programm zur Primzahlensuche mit dem **Sieb des Eratosthenes** (ca. 275 - 195 v. Chr.). Dieser Algorithmus reduziert sukzessive eine Menge von Primzahlkandidaten, die initial alle natürlichen Zahlen bis zu einer Obergrenze K enthält, also $\{2, 3, ..., K\}$.

- Im ersten Schritt werden alle echten Vielfachen der Basiszahl 2 (also 4, 6, ...) aus der Kandidatenmenge gestrichen, während die Zahl 2 in der Liste verbleibt.
- Dann geschieht iterativ folgendes:
  - Als neue Basis b wird die kleinste Zahl gewählt, welche die beiden folgenden Bedingungen erfüllt:
    b ist größer als die vorherige Basiszahl.
    b ist im bisherigen Verlauf nicht gestrichen worden.
  - Die echten Vielfachen der neuen Basis (also $2 \cdot b$, $3 \cdot b$, ...) werden aus der Kandidatenmenge gestrichen, während die Zahl b in der Liste verbleibt.
- Das Streichverfahren kann enden, wenn für eine neue Basis b gilt:

$$b > \sqrt{K}$$

In der Kandidatenrestmenge befinden sich dann nur noch Primzahlen. Um dies einzusehen, nehmen wir an, es hätte eine Zahl $n \leq K$ mit echtem Teiler das Streichverfahren überstanden. Mit zwei positiven Zahlen u, v würde dann gelten:

$$n = u \cdot v \text{ und } u < b \text{ oder } v < b \text{ (wegen } b > \sqrt{K} \text{ und } n \leq K)$$

Wir nehmen ohne Beschränkung der Allgemeinheit $u < b$ an und unterscheiden zwei Fälle:

- u war zuvor als Basis dran:
  Dann wurde n bereits als Vielfaches von u gestrichen.
- u wurde zuvor als Vielfaches einer früheren Basis $\tilde{b}$ $(< b)$ gestrichen $(u = k\tilde{b})$
  Dann wurde auch n bereits als Vielfaches von $\tilde{b}$ gestrichen.

Damit erweist sich die Annahme als falsch, und es ist erweisen, dass die Kandidatenrestmenge nur noch Primzahlen enthält.

Sollen z.B. alle Primzahlen kleiner oder gleich 18 bestimmt werden, so startet man mit folgender Kandidatenmenge:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Im ersten Schritt werden die echten Vielfachen der Basis 2 gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Als neue Basis wird die Zahl 3 gewählt (> 2, nicht gestrichen). Ihre echten Vielfachen werden gestrichen:

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18

Als neue Basis wird die Zahl 5 gewählt (> 3, nicht gestrichen). Allerdings ist 5 größer als $\sqrt{18}$ (≈ 4,24) und der Algorithmus daher bereits beendet. Als Primzahlen kleiner oder gleich 18 erhalten wir also:

2, 3, 5, 7, 11, 13 und 17

4) Erstellen Sie eine Klasse für zweidimensionale Matrizen mit Elementen vom Typ **float**. Implementieren Sie eine Methode zum Transponieren einer Matrix.

5) Erstellen Sie ein Programm zum Berechnen einer persönlichen Glückszahl (zwischen 1 und 100), indem Sie:

- den Vor- und Nachnamen als Befehlszeilenargumente einlesen,
- den ersten Buchstaben des Vornamens sowie den letzten Buchstaben des Nachnamens ermitteln (beide in Großschreibung),
- die Nummern der beiden Buchstaben im Unicode-Zeichensatz bestimmen,
- die beiden Buchstabennummern addieren und die Summe als Startwert für den Pseudozufallszahlengenerator aus der Klasse **Random** verwenden.

Beenden Sie Ihr Programm mit einer Fehlermeldung, wenn weniger als zwei Befehlszeilenargumente übergeben wurden.

Tipps:

- Um die durch Leerzeichen getrennten Befehlszeilenargumente im Programm als **String**-Array verfügbar zu haben, definiert man im Kopf der **Main()** - Methode einen Parameter vom Typ **String[]**:
  ```
  static void Main(string[] args) {...}
  ```
- Wie jede andere Methode kann auch **Main()** per **return**-Anweisung spontan beendet werden.

6) Erstellen Sie eine Klasse `StringUtil` mit einer statischen Methode `WrapLine()`, die einen **String** auf die Konsole schreibt und dabei einen korrekten Zeilenumbruch vornimmt. Anwender Ihrer Methode sollen die gewünschte Zeilenbreite vorgeben können und auch die Trennzeichen festlegen dürfen, aber nicht müssen (Methoden überladen!).

Weitere Anforderungen an die Methode:

- Das Leerzeichen soll auf jeden Fall trennend wirken.
- Aufeinanderfolgende Trennzeichen sollen wie ein einzelnes Trennzeichen wirken.
- Ist ein Wort breiter als die Ausgabezeile, ist ein Umbruch *innerhalb* des Wortes unvermeidlich.

Im folgenden Programm wird die Verwendung der Methode demonstriert:

```
using System;
class StringUtilTest {
   static void Main() {
      string s = "Dieser Satz passt nicht in eine Schmal-Zeile, "+
                 "die nur wenige Spalten umfasst.";
      StringUtil.WrapLine(s," -", 40);
      StringUtil.WrapLine(s, 40);
      StringUtil.WrapLine(s);
   }
}
```

Der zweite Methodenaufruf sollte folgende Ausgabe erzeugen:

```
Dieser Satz passt nicht in eine
Schmal-Zeile, die nur wenige Spalten
umfasst.
```

Tipp: Eine wesentliche Hilfe kann die **String**-Methode **Split()** sein, die auf Basis einer einstellbaren Menge von Trennzeichen alle Teilzeichenfolgen der angesprochenen Instanz ermittelt und in einem **String**-Array ablegt. In folgendem Programm wird die Arbeitsweise demonstriert:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `   String s = "Dies ist der Beispiel-Satz, der zerlegt werden soll.";` `   String[] tokens = s.Split(new char[] {' ', '-'},` `                              StringSplitOptions.RemoveEmptyEntries);` `   foreach (String t in tokens)` `    Console.WriteLine(t);` `  }` `}`	`Dies` `ist` `der` `Beispiel` `Satz,` `der` `zerlegt` `werden` `soll.`

Die Trennzeichen sind *nicht* in den produzierten Teilzeichenfolgen enthalten, so dass z.B. ein als Trennzeichen definierter Bindestrich verloren geht. Mit dem Enumerationswert

```
StringSplitOptions.RemoveEmptyEntries
```

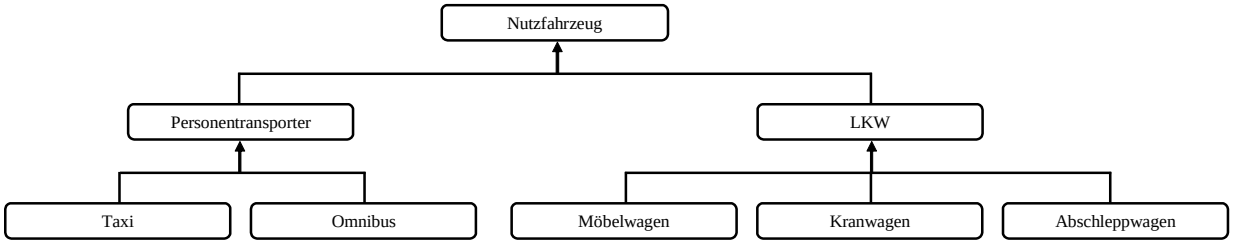
für den zweiten **Split()** - Parameter werden leere Teilzeichenfolgen im resultierenden **String**-Array (z.B. resultierend aus mehreren aufeinander folgenden Leerzeichen) verhindert.

6 Vererbung und Polymorphie

Im Manuskript war schon mehrfach davon die Rede, dass die .NET - Datentypen in eine strenge Abstammungshierarchie eingeordnet sind. Nun betrachten wir die Vererbungsbeziehung zwischen Klassen und die damit verbundenen Vorteile für die Softwareentwicklung im Detail.

Modellierung realer Klassenhierarchien

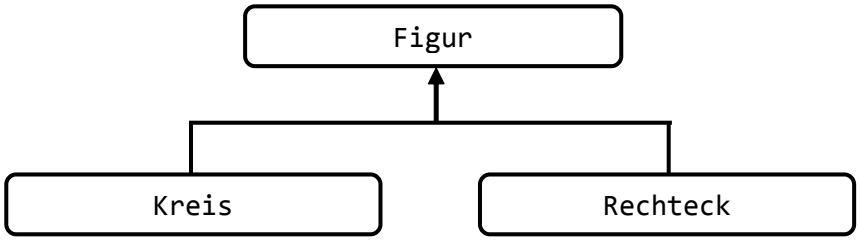
Beim Modellieren eines Gegenstandbereiches durch Klassen, die durch Merkmale (Instanz- und Klassenvariablen) sowie Handlungskompetenzen (Instanz- und Klassenmethoden) gekennzeichnet sind, müssen auch die Spezialisierungs- bzw. Generalisierungsbeziehungen zwischen real existierenden Klassen abgebildet werden. Eine Firma für Transportaufgaben aller Art mag ihre Nutzfahrzeuge folgendermaßen klassifizieren:



Einige Merkmale sind für alle Nutzfahrzeuge relevant (z.B. Anschaffungspreis, momentane Position), andere betreffen nur spezielle Klassen (z.B. maximale Anzahl der Fahrgäste, maximale Anhängelast). Ebenso sind einige Handlungsmöglichkeiten bei allen Nutzfahrzeugen vorhanden (z.B. eigene Position melden, ein Ziel ansteuern), während andere speziellen Fahrzeugen vorbehalten sind (z.B. Fahrgäste befördern, Lasten transportieren). Ein Programm zur Einsatzplanung und Verwaltung des Fuhrparks sollte diese Klassenhierarchie abbilden.

Übungsbeispiel

Bei unseren Beispielprogrammen bewegen wir uns in einem bescheideneren Rahmen und betrachten meist eine einfache Hierarchie mit Klassen für geometrische Figuren:[1]



Die Vererbungstechnik der OOP

In objektorientierten Programmiersprachen ist es weder sinnvoll noch erforderlich, jede Klasse einer Hierarchie komplett neu zu definieren. Es steht eine mächtige und zugleich einfach handhabbare Vererbungstechnik zur Verfügung: Man geht von der allgemeinsten Klasse aus und leitet durch Spezialisierung neue Klassen ab, nach Bedarf in beliebig vielen Stufen. Eine abgeleitete Klasse erbt alle Merkmale und Handlungskompetenzen ihrer **Basisklasse** (jedoch keine Konstruktoren) und

---

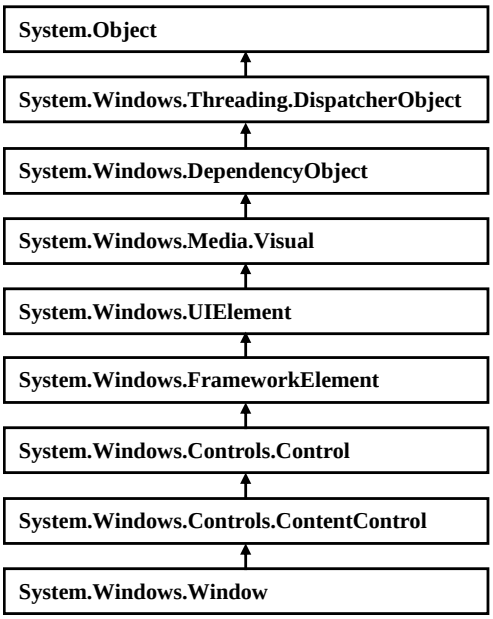
[1] Vielleicht haben manche Leser als Gegenstück zum Rechteck (auf derselben Hierarchieebene) die *Ellipse* erwartet, die ebenfalls zwei ungleiche lange „Hauptachsen“ besitzt. Weiterhin liegt es auf den ersten Blick nahe, den Kreis als Spezialisierung der Ellipse und das Quadrat als Spezialisierung des Rechtecks zu betrachten. Wir werden aber in Abschnitt 6.10 über das *Liskovsche Substitutionsprinzip* genau diese Ableitungen (von Kreis aus Ellipse bzw. von Quadrat aus Rechteck) kritisieren. Daher ist es akzeptabel, an Stelle der Ellipse den Kreis neben das Rechteck zu stellen, um das Erlernen der neuen Konzepte durch ein möglichst einfaches Beispiel zu erleichtern.

kann nach Bedarf Anpassungen bzw. Erweiterungen zur Lösung spezieller Aufgaben vornehmen, z.B.:

- zusätzliche Felder deklarieren
- zusätzliche Methoden oder Eigenschaften definieren
- geerbte Methoden ersetzen, d.h. unter Beibehaltung der Signatur umgestalten

Ihre Konstruktoren muss eine abgeleitete Klasse neu definieren, wobei es aber möglich ist, einen Basisklassenkonstruktor zur Initialisierung von geerbten Instanzvariablen einzuspannen (siehe unten).

Die FCL ist das beste Beispiel für den erfolgreichen Einsatz der Vererbungstechnik. Viele von uns benötigte Klassen haben einen länglichen Stammbaum, z.B. die Klasse **Window** für das Hauptfenster einer WPF-Anwendung:

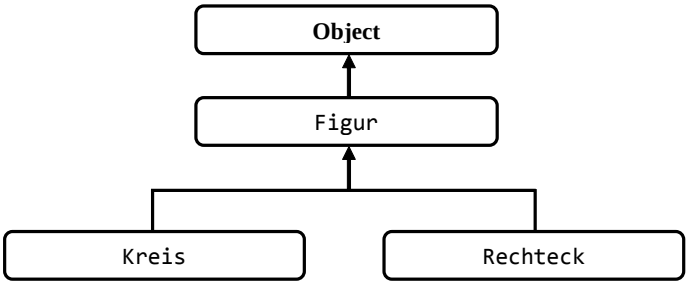


Software-Recycling

Mit ihrem Vererbungsmechanismus bietet die objektorientierte Programmierung ideale Voraussetzungen dafür, vorhandene Software auf rationelle Weise zur Lösung neuer Aufgaben zu verwenden. Dabei können allmählich umfangreiche Softwaresysteme entstehen, die gleichzeitig robust und innovationsoffen sind. Natürlich kann C# nicht garantieren, dass jede Klassenhierarchie exzellent entworfen ist und langfristig von einer stetig wachsenden Entwicklergemeinde eingesetzt wird.

6.1 Das allgemeine Typsystem des .NET - Frameworks

Im .NET - Framework stammen *alle* Klassen und sonstigen Typen (z.B. Strukturen, Enumerationen) von der Klasse **Object** aus dem Namensraum **System** ab. Das gilt sowohl die in der FCL enthaltenen als auch die von Anwendungsentwicklern definierten Typen. Wird (wie bei unseren bisherigen Beispielen) in der Definition einer Klasse *keine* Basisklasse angegeben, dann stammt sie auf direktem Wege von der Urahnklasse **Object** ab. Die oben dargestellte Klassenhierarchie zum Figurenbeispiel muss also folgendermaßen vervollständigt werden:



Auch die Strukturen (vgl. Abschnitt 5.1) sind in das allgemeine Typsystem (engl.: Common Type System, CTS) eingeordnet. Sie stammen implizit von der Klasse **System.ValueType** ab, die wiederum direkt von der Urahnklasse **Object** erbt. Aus einer Struktur kann aber weder eine andere Struktur noch eine Klasse abgeleitet werden. Analoges gilt für die Aufzählungstypen, die von der Klasse **System.Enum** (mit der Basisklasse **System.ValueType**) abstammen (vgl. Abschnitt 5.5).

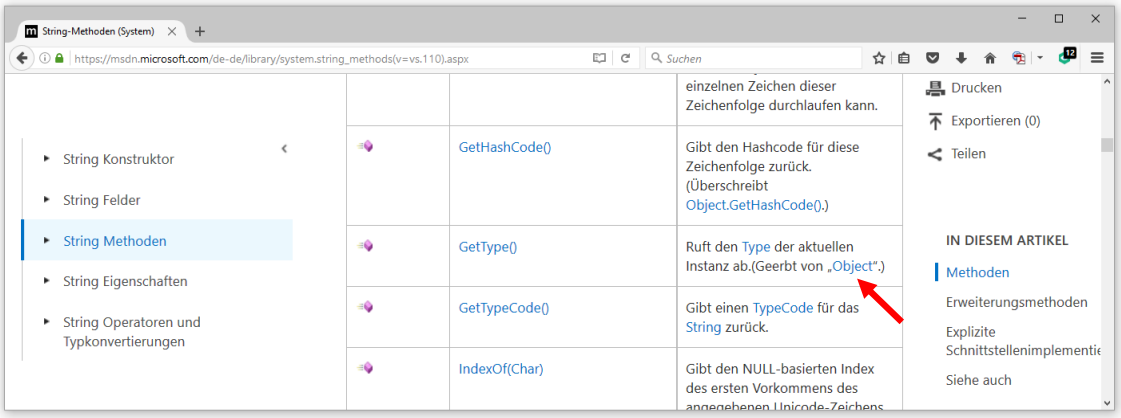
Jeder Typ erbt alle Merkmale und Handlungskompetenzen aus der eigenen Abstammungslinie von der Urahnklasse **Object** beginnend. Folglich kann z.B. jedes Objekt und jede Strukturinstanz die in der Urahnklasse definierte Methode **GetType()** ausführen, die ein auskunftsfreudiges **Type**-Objekt liefert. Im folgenden **WriteLine()** - Aufruf verraten drei **Type**-Objekte (über die implizit aufgerufene Methode **ToString()**) die Typbezeichnung samt Namensraum:

Quellcode	Ausgabe

```
using System;
class Prog {
    static void Main() {
        var o = new Object();
        var s = "abc";
        int i = 13;
        Console.WriteLine(o.GetType() + "\n" +
                          s.GetType() + "\n" +
                          i.GetType());
    }
}
```

```
System.Object
System.String
System.Int32
```

In der FCL-Dokumentation zu einem Datentyp sind die Erbstücke gekennzeichnet, z.B. bei der Klasse **String**:



Nach dieser kurzen Beschäftigung mit der ohne eigene Leistungen verfügbaren Standarderbschaft, machen wir uns daran, eigene Vererbungshierarchien aufzubauen.

6.2 Definition einer abgeleiteten Klasse

Wir definieren im angekündigten Beispiel zunächst die Basisklasse `Figur`, die Instanzvariablen für die X- und die Y-Position der linken oberen Ecke einer zweidimensionalen Figur, zwei Konstruktoren sowie eine Methode `Wo()` zur Positionsmeldung besitzt:

```
using System;
public class Figur {
   double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
      Console.WriteLine("Figur-Konstruktor");
   }
   public Figur() { }

   public void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

Wir definieren die Klasse `Kreis` als Spezialisierung der Klasse `Figur`, indem wir hinter den Klassennamen durch Doppelpunkt getrennt den Basisklassennamen setzen:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
      Console.WriteLine("Kreis-Konstruktor");
   }
   public Kreis() { }

   public double Radius {
      get {return radius;}
      set {if (value >= 0.0) radius = value;}
   }
}
```

Die `Kreis`-Klasse erbt die beiden Positionsvariablen sowie die Methode `Wo()` und ergänzt eine zusätzliche Instanzvariable für den Radius samt Eigenschaft für Zugriffe durch fremde Typen.

Es wird ein initialisierender `Kreis`-Konstruktor definiert, der über das Schlüsselwort **base** den initialisierenden Konstruktor der Basisklasse aufruft. Weil die `Figur`-Instanzvariablen (noch) als **private** deklariert sind, wäre dem `Kreis`-Konstruktor auch kein direkter Zugriff erlaubt.

In der `Kreis`-Klasse wird (wie in der Basisklasse `Figur`) auch ein parameter*freier* Konstruktor definiert. Vielleicht hat jemand gehofft, die `Kreis`-Klasse könnte den parameterfreien Konstruktor ihrer Basisklasse (bei Anpassung des Namens) übernehmen. Konstruktoren werden jedoch grundsätzlich **nicht** vererbt.

In C# ist **keine Mehrfachvererbung** möglich: Man kann also in einer Klassendefinition nur *eine* Basisklasse angeben. Im Sinne einer realitätsnahen Modellierung wäre eine Mehrfachvererbung gelegentlich durchaus wünschenswert. So könnte z.B. die Klasse `Receiver` von den Klassen `Tuner` und `Amplifier` erben. Man hat auf die in anderen Programmiersprachen (z.B. C++) erlaubte Mehrfachvererbung bewusst verzichtet, um von vornherein den kritischen Fall auszuschließen, dass eine abgeleitete Klasse gleichnamige *Instanzvariablen* von mehreren Klassen erbt, woraus Mehrdeutigkeiten und Fehler resultieren können.

Einen gewissen Ersatz bieten die in Kapitel 8 behandelten Schnittstellen (Interfaces), weil ...

- bei Schnittstellen die Mehrfachvererbung erlaubt ist,
- und außerdem eine Klasse mehrere Schnittstellen implementieren darf.

6.3 base-Konstruktoren und Initialisierungs-Sequenzen

Zwar werden Konstruktoren nicht vererbt, doch ist bei der Entstehung einer Instanz eines abgeleiteten Typs ein Konstruktor aus *jeder* Basisklasse entlang der Ahnenreihe durch impliziten oder expliziten Aufruf beteiligt. Das folgende Programm erzeugt ein Objekt aus der Klasse `Figur` und ein Objekt aus der von `Figur` abgeleiteten Klasse `Kreis`, wobei die beteiligten Konstruktoren ihre Tätigkeit melden:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  var fig = new Figur(50.0, 50.0);` `  Console.WriteLine();` `  var krs = new Kreis(150.0, 200.0, 50.0);` `  }` `}`	`Figur-Konstruktor` `Figur-Konstruktor` `Kreis-Konstruktor`

Vom ebenfalls beteiligten **Object**-Konstruktor ist nichts zu sehen, weil die FCL-Designer natürlich keine Kontrollausgabe eingebaut haben. Wir werden die Beiträge der einzelnen Konstruktoren bei der Erstellung eines neuen `Kreis`-Objekts gleich noch genauer analysieren.

Wie schon in Abschnitt 6.2 zu sehen war, erledigt man den *expliziten* Aufruf eines Basisklassenkonstruktors im Kopfbereich eines Unterklassenkonstruktors über das Schlüsselwort **base**, z.B.:

```
public Kreis(double x, double y, double rad) : base(x, y) {
     if (rad >= 0.0)
        radius = rad;
  Console.WriteLine("Kreis-Konstruktor");
}
```

Dadurch ist es möglich, geerbte Instanzvariablen zu initialisieren, die in der Basisklasse als **private** deklariert sind.

In einem Unterklassenkonstruktor *ohne* **base**-Klausel ruft der Compiler implizit den *parameterlosen* Konstruktor der Basisklasse auf. Fehlt ein solcher, weil der Programmierer einen eigenen, parametrisierten Konstruktor erstellt und nicht durch einen expliziten parameterlosen Konstruktor ergänzt hat, dann protestiert der Compiler, z.B.:

```
Kreis.cs(12,12): error CS1501: Keine Überladung für die Methode Figur
erfordert 0-Argumente
```

Es gibt zwei offensichtliche Möglichkeiten, das Problem zu lösen:

- Im Unterklassen-Konstruktor über das Schlüsselwort **base** einen parametrisierten Basisklassen-Konstruktor aufrufen.
- In der Basisklasse einen parameterfreien Konstruktor definieren.

Der parameterlose Basisklassenkonstruktor wird auch vom (implizit definierten) Standardkonstruktor einer abgeleiteten Klasse aufgerufen.

Es ist klar, dass ein Basisklassenkonstruktor mit passender Signatur nicht nur vorhanden, sondern auch verfügbar sein muss (z.B. dank **public**-Deklaration).

Beim Erzeugen eines Unterklassenobjekts laufen folgende Initialisierungs-Maßnahmen ab:

- Alle Instanzvariablen (auch die geerbten) werden (auf dem Heap) angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der Unterklassenkonstruktor führt nacheinander folgende Aktionen aus:
  - Die in Felddeklarationen der Unterklasse enthaltenen Initialisierungen werden ausgeführt. Den zugehörigen CIL-Code erzeugt der Compiler automatisch.
  - Es folgt der Aufruf eines Basisklassenkonstruktors.
  - Nach Beendigung des Basisklassenkonstruktors wird der Rumpf des Unterklassen-Konstruktors ausgeführt.
- Im aufgerufenen Basisklassenkonstruktor läuft dieselbe Sequenz ab (Instanzvariablen der Klasse initialisieren, Aufruf eines Basisklassenkonstruktors, Anweisungsteil). Diese Rekursion endet mit dem Aufruf eines **Object**-Konstruktors.

Betrachten wir zum Beispiel, was beim Erzeugen eines `Kreis`-Objektes mit dem Konstruktor-Aufruf

```
Kreis(150.0, 200.0, 50.0);
```

geschieht:

- Alle Instanzvariablen (auch die geerbten) werden angelegt und mit den typspezifischen Nullwerten initialisiert.
- Der `Kreis`-Konstruktor führt die Initialisierung `radius = 75.0` gemäß `Kreis`-Klassendefinition aus.
- Der explizit über das Schlüsselwort **base** aufgerufene `Figur`-Konstruktor mit Positionsparametern startet und führt die Initialisierungen `xpos = 100.0` sowie `ypos = 100.0` gemäß `Figur`-Klassendefinition aus.
- Der parameterlose **Object**-Konstruktor startet. Die Instanzvariablen der Klasse **Object** erhalten einen Initialisierungswert gemäß **Object**-Klassendefinition. Derzeit sind mir zwar keine **Object**-Instanzvariablen bekannt, doch ist die Existenz von gekapselten Exemplaren durchaus möglich.
- Der Rumpf des parameterlosen **Object**-Konstruktors wird ausgeführt.
- Der Rumpf des parametrisierten `Figur`-Konstruktors wird ausgeführt, wobei `xpos` und `ypos` die Aktualparameterwerte 150 bzw. 200 erhalten.
- Der Rumpf des parametrisierten `Kreis`-Konstruktors wird ausgeführt, wobei `radius` den Aktualparameterwert 50 erhält.

6.4 Der Zugriffsmodifikator protected

Auf **private**-Member einer Basisklasse haben Methoden einer abgeleiteten Klasse (so wie Methoden beliebiger Klassen) *keinen* Zugriff. Um abgeleiteten Klassen besondere Rechte einzuräumen, bietet C# den Zugriffsmodifikator **protected**, welcher den Zugriff durch die eigene Klasse *und* durch alle (direkt oder indirekt) *abgeleiteten* Klassen erlaubt, z.B.:

```
using System;
public class Figur {
   protected double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
      Console.WriteLine("Figur-Konstruktor");
   }
   public Figur() { }
   public void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

Weil die Basisklasse `Figur` ihre Instanzvariablen `xpos` und `ypos` nun als **protected** deklariert, können sie in der `Kreis`-Methode `OLE2Zen()`, welche die obere linke Ecke in das aktuelle Zentrum verschiebt, verändert werden:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
      Console.WriteLine("Kreis-Konstruktor");
   }
   public Kreis() {}

   public double Radius {
      get {return radius;}
         set {if (value >= 0.0) radius = value;}
   }

   public void OLE2Zen() {
      xpos = xpos + radius;
      ypos = ypos + radius;
   }
}
```

Es ist zu beachten, dass hier *geerbte Instanzvariablen* von `Kreis`-Objekten verändert werden. Auf das `xpos`-Feld eines `Figur`-Objekts haben die Methoden der `Kreis`-Klasse jedoch *keinen* Zugriff.

Für Methoden *fremder* Klassen sind **protected**-deklarierte Member ebenso gesperrt wie private, z.B.:

```
using System;
class Prog {
   static void Main() {
      Kreis krs = new Kreis(10.0, 10.0, 5.0);
      //krs.xpos = 77.7;    // In der Prog-Methode ist der Zugriff verboten.
      krs.OLE2Zen();        // In der Kreis-Methode ist der Zugriff erlaubt.
   }
}
```

Der Modifikator **protected** ist natürlich nicht nur bei Feldern erlaubt, sondern bei beliebigen Mitgliedern, z.B. bei (statischen) *Methoden*, z.B.:

```
static protected void ProSt() {
   Console.WriteLine("Protected und statisch!");
}
```

6.5 Erbstücke durch spezialisierte Varianten verdecken

6.5.1 Geerbte Methoden, Eigenschaften und Indexer verdecken

Eine geerbte Basisklassenmethode kann in einer abgeleiteten Klasse durch eine Methode mit gleicher Signatur **verdeckt** werden. Zwei Methoden haben genau dann dieselbe **Signatur**, wenn die Namen und die Parameterlisten (hinsichtlich Typ und Art aller Formalparameter) übereinstimmen, während die Rückgabetypen keine Rolle spielen (vgl. Abschnitt 4.3.5).

Bisher steht in der `Kreis`-Klasse zur Ortsangabe die geerbte Methode `Wo()` zur Verfügung, welche die Position der linken oberen Ecke eines Objekts ausgibt. In der `Kreis`-Klasse kann aber eine bessere Ortsangabenmethode realisiert werden, weil hier auch die rechte untere Ecke definiert ist:[1]

```
using System;
public class Kreis : Figur {
   . . .
   public new void Wo() {
      base.Wo();
      Console.WriteLine("Unten Rechts:\t(" + (xpos + 2 * radius) +
                        ", " + (ypos + 2 * radius) + ")");
   }
}
```

Im Definitionskopf der verdeckenden Methode sollte man den Modifikator **new** angeben, um die folgende Warnung des Compilers zu vermeiden:[2]

```
Kreis.cs(14,15): warning CS0108: "Kreis.Wo()" blendet den vererbten Member
        "Figur.Wo()" aus. Verwenden Sie das new-Schlüsselwort, wenn das
        Ausblenden vorgesehen war.
```

Mit diesem Hinweis soll ein ungeplantes Verdecken (z.B. durch Tippfehler) verhindert werden.

Im Anweisungsteil der neuen Methode kann man sich oft durch Rückgriff auf die verdeckte Basisklassenmethode die Arbeit erleichtern, wobei wieder das Schlüsselwort **base** zum Einsatz kommt.

Das folgende Programm schickt an eine `Figur` und an einen `Kreis` jeweils die Nachricht `Wo()`, und beide zeigen ihr artspezifisches Verhalten:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  var f = new Figur(10.0, 20.0);` `  f.Wo();` `  var k = new Kreis(50.0, 50.0, 10.0);` `  k.Wo();` `  }` `}`	`Oben Links:  (10, 20)` `Oben Links:  (50, 50)` `Unten Rechts:  (70, 70)`

Es liegt *keine* Verdeckung vor, wenn in der Unterklasse eine Methode mit gleichem Namen, aber abweichender Parameterliste definiert wird. In diesem Fall sind die beiden Signaturen verschieden, und es handelt sich um eine *Überladung*.

---

[1] Falls Sie sich über die Berechnungsvorschrift für die Y-Koordinate der rechten unteren Kreis-Ecke wundern: Bei der Grafikausgabe von Computersystemen ist die Position (0, 0) meist in der oberen linken Ecke des Bildschirms bzw. des aktuellen Fensters angesiedelt. Die X-Koordinaten wachsen (wie aus der Mathematik gewohnt) von links nach rechts, während die Y-Koordinaten von oben nach unten wachsen. Wir wollen uns im Hinblick auf die in absehbarer Zukunft anstehende Programmierung graphischer Benutzeroberflächen schon jetzt daran gewöhnen.

[2] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

Ist eine verdeckende Methode als privat deklariert, wird sie nur *klassenintern* verwendet, und in der Schnittstelle der abgeleiteten Klasse bleibt das signaturgleiche Erbstück erhalten, z.B.:

Quellcode	Ausgabe

```
using System;
class Basisklasse {
    public void NenneTyp() {
        Console.WriteLine("Basisklasse");
    }
}

class Spezialklasse : Basisklasse {
    new void NenneTyp() {
        Console.WriteLine("Spezialklasse");
    }
    public void DeinTyp() {
        NenneTyp();
    }
}

class Prog {
    static void Main() {
        var b = new Basisklasse();
        b.NenneTyp();
        var a = new Spezialklasse();
        a.NenneTyp();
        a.DeinTyp();
    }
}
```

```
Basisklasse
Basisklasse
Spezialklasse
```

Eine solche Konstruktion ist aber potentiell verwirrend und wohl nur selten nützlich.

Neben objektbezogenen können auch *statische* Methoden verdeckt werden, wobei die Basisklassenvariante durch Voranstellen des Klassennamens angesprochen werden kann, z.B.:

Quellcode	Ausgabe

```
using System;
class Basisklasse {
    public static void NenneTyp() {
        Console.WriteLine("Basisklasse");
    }
}
class Spezialklasse : Basisklasse {
    public new static void NenneTyp() {
        Console.Write("Spezialklasse\n abgeleitet von: ");
        Basisklasse.NenneTyp();
    }
}
class Prog {
    static void Main() {
        Basisklasse.NenneTyp();
        Spezialklasse.NenneTyp();
    }
}
```

```
Basisklasse
Spezialklasse
 abgeleitet von: Basisklasse
```

Schließlich ist das Verdecken ist nicht nur bei Methoden erlaubt, sondern auch bei Eigenschaften und Indexern.

6.5.2 Geerbte Felder verdecken

Auch geerbte Instanz- und Klassenvariablen lassen sich verdecken, was aber im Sinne eines möglichst leicht nachvollziehbaren Quelltextes nur in Ausnahmefällen geschehen sollte. Verwendet man z.B. in der abgeleiteten Klasse AK für eine Instanzvariable einen Namen, der bereits eine Variable der beerbten Basisklasse BK bezeichnet, dann wird die Basisklassenvariable verdeckt. Sie ist jedoch weiterhin vorhanden und kommt in folgenden Situationen zum Einsatz:

- Von BK geerbte Methoden greifen weiterhin auf die BK-Variable zu, während die zusätzlichen Methoden der AK-Klasse auf die AK-Variable zugreifen.
- In AK-Methoden steht die verdeckte Variante über das Schlüsselwort **base** zur Verfügung.

Im folgenden Beispielprogramm führt ein AK-Objekt eine BK- und eine AK-Methode aus, um die beschriebenen Zugriffsvarianten zu demonstrieren:

Quellcode	Ausgabe

```
using System;
class BK {
   protected string x = "Bast";
   public void BM() {
      Console.WriteLine("x in BK-Methode:\t"+x);
   }
}

class AK : BK {
   new int x = 333;
   public void AM() {
      Console.WriteLine("x in AK-Methode:\t"+x);
      Console.WriteLine("base-x in AK-Methode:\t"+
                        base.x);
   }
}

class Prog {
   static void Main() {
      AK ako = new AK();
      ako.BM();
      ako.AM();
   }
}
```

```
x in BK-Methode:        Bast
x in AK-Methode:        333
base-x in AK-Methode:   Bast
```

In der Deklaration einer verdeckenden Variablen sollte man den Modifikator **new** angeben, um die folgende Warnung des Compilers zu vermeiden:[1]

```
Verdecken.cs(10,6,10,7): warning CS0108: "AK.x" blendet den vererbten Member "BK.x"
aus. Verwenden Sie das new-Schlüsselwort, wenn das Ausblenden vorgesehen war.
```

Mit diesem aufmerksamen Hinweis soll ein ungeplantes Verdecken durch Tippfehler oder Unachtsamkeit verhindert werden.

6.6 Verwaltung von Objekten über Basisklassenreferenzen

Eine Basisklassen-Referenzvariable darf die Adresse eines beliebigen Unterklassenobjektes aufnehmen. Schließlich besitzt Letzteres die komplette Ausstattung der Basisklasse und kann z.B. auf dort definierte Methodenaufrufe geeignet reagieren. Ein Objekt steht nicht nur zur eigenen Klasse in der „ist-ein“ - Beziehung, sondern erfüllt diese Relation auch in Bezug auf die direkte Basisklasse

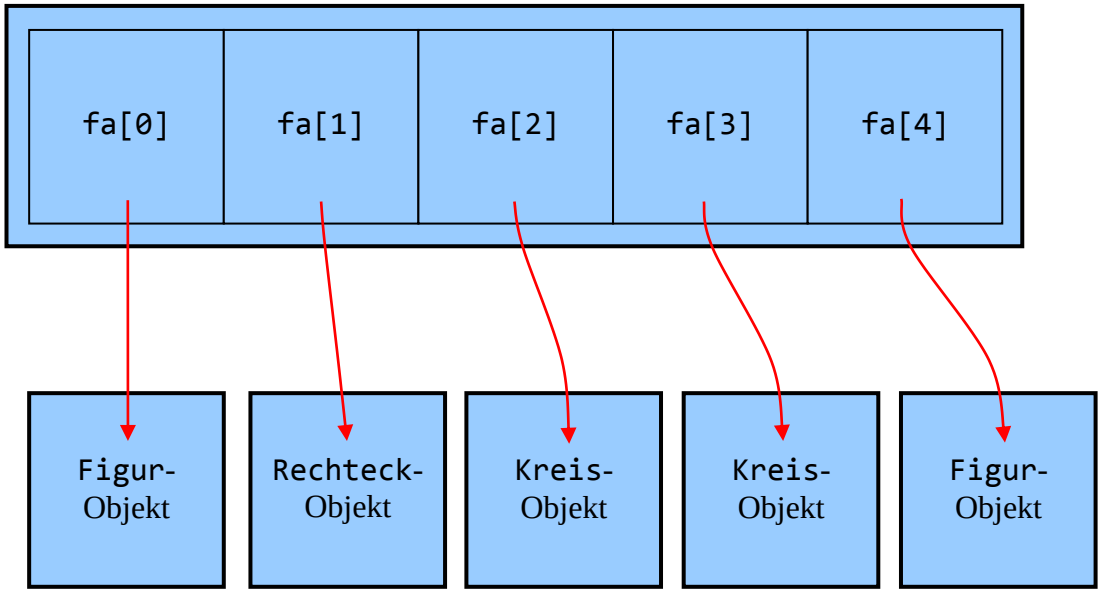
[1] Dieser Modifikator darf nicht mit dem *Operator* **new** verwechselt werden.

sowie in Bezug auf alle indirekten Basisklassen in der Ahnenreihe. Angewendet auf das Beispiel in Abschnitt 6.2 ergibt sich die sehr plausible Feststellung, dass jeder Kreis auch eine Figur ist.

Andererseits verfügt ein Basisklassenobjekt in der Regel *nicht* über die Ausstattung von abgeleiteten (erweiterten bzw. spezialisierten) Klassen. Daher ist es sinnlos und verboten, die Adresse eines Basisklassenobjektes in einer Unterklassen-Referenzvariablen abzulegen.

Über Referenzvariablen vom Typ einer *gemeinsamen* Basisklasse lassen sich also Objekte aus unterschiedlichen Klassen verwalten. Im Rahmen eines Grafikprogramms kommt vielleicht ein Array mit dem Elementtyp `Figur` zum Einsatz, dessen Elemente auf Objekte aus der Basisklasse oder aus einer abgeleiteten Klasse wie `Kreis` oder `Rechteck` zeigen:

Array `fa` mit Elementtyp `Figur`



Ein Array vom Typ `Figur` kann Referenzen auf Figuren und Kreise aufnehmen, z.B.:

```
using System;
class Prog {
   static void Main() {
      Figur[] fa = new Figur[5];
      fa[0] = new Figur(10.0, 10.0);
      fa[1] = new Rechteck(20.0, 20.0, 20.0, 20.0);
      fa[2] = new Kreis(30.0, 30.0, 30.0);
      fa[3] = new Kreis(40.0, 40.0, 30.0);
      fa[4] = new Figur(50.0, 50.0);
      foreach (Figur e in fa)
         e.Wo();
   }
}
```

Bei Ansprache per Basisklassenreferenz führen die Objekte *nicht* die verdeckende, artspezifische `Wo()` - Methode aus, sondern die Basisklassenvariante:

```
Oben Links:     (10, 10)

Oben Links:     (20, 20)

Oben Links:     (30, 30)

Oben Links:     (40, 40)

Oben Links:     (50, 50)
```

In Abschnitt 6.7 über die *Polymorphie* werden Sie erfahren, dass man in der Basisklasse eine *virtuelle* Methode definieren und diese in den abgeleiteten Klassen *überschreiben* muss, damit auch bei Ansprache per Basisklassenreferenz ein artspezifisches Verhalten resultiert.

Über eine `Figur`-Referenzvariable, die auf ein `Kreis`-Objekt zeigt, sind *Erweiterungen* der `Kreis`-Klasse *nicht* unmittelbar zugänglich. Wenn (auf eigene Verantwortung des Programmierers) eine Basisklassenreferenz als Unterklassenreferenz behandelt werden soll, um eine unterklassenspezifische Methode, Eigenschaft oder Variable anzusprechen, dann muss eine explizite Typumwandlung vorgenommen werden, z.B.:

```
((Kreis)fa[1]).Radius
```

Geschieht dies zu Unrecht, tritt ein Ausnahmefehler auf, z.B.:

```
Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs "Figur" kann
nicht in Typ "Kreis" umgewandelt werden.
```

Bisher haben wir die explizite Typumwandlung nur auf Werttypen (meist elementare Typen) angewendet, sie spielt aber auch bei Referenztypen eine wichtige Rolle. Welche expliziten Konvertierungen erlaubt sind, ist der C# - Sprachspezifikation (Microsoft 2012, Abschnitt 6.2) zu entnehmen. Im konkreten Fall wird der deklarierte Typ (`Figur`) durch eine Spezialisierung bzw. Ableitung (`Kreis`) ersetzt. Der Compiler erlaubt die Konvertierung, übernimmt jedoch keine Verantwortung dafür. Die verbleibt beim Programmierer.

Im Zweifelsfall sollte man per **is** - (Typtest-)Operator überprüfen, ob das referenzierte Objekt tatsächlich den vermuteten Laufzeittyp besitzt, z.B.:

```
foreach (Figur e in fa) {
   e.Wo();
   if (e is Kreis)
      Console.WriteLine("Radius:         " + ((Kreis)e).Radius);
}
```

An Stelle des gewohnten Typumwandlungsoperators

```
((Kreis)e).Radius
```

kann im Beispiel auch der **as**-Operator eingesetzt werden:

```
(e as Kreis).Radius
```

Die wichtigsten Regeln für den **as**-Operator:

- Der Zieltyp im zweiten Operanden muss ein Referenztyp oder ein **null**-fähiger Werttyp sein (siehe Abschnitt 7.3). Als **null**-fähig bezeichnet man einen Werttyp dann, wenn neben den normalen Werten auch der Ausnahmewert **null** zur Verfügung steht, was z.B. bei den elementaren Datentypen *nicht* der Fall ist.
- Im ersten Operanden verlangt der Compiler einen Ausdruck, dessen Wert potentiell in den Zieltyp konvertiert werden kann. Dies ist z.B. der Fall, wenn der Typ des Ausdrucks eine Basisklasse des Zieltyps ist (wie im obigen Beispiel). Weitere Details finden sich in der C# - Sprachspezifikation (Microsoft 2012, Abschnitt 7.10.11).

- Stellt sich zur Laufzeit eine Konvertierung als unmöglich heraus, liefert der **as**-Operator im Unterschied zum gewohnten Typumwandlungsoperator *keinen* Ausnahmefehler, sondern den Ergebniswert **null**. Laut C# - Sprachspezifikation lässt sich das Verhalten des **as**-Operators im Beispiel

  ```
  e as Kreis
  ```

  mit Hilfe des Typumwandlungs- und des Konditionaloperators so beschreiben:

  ```
  (e is Kreis)?(Kreis)e:null
  ```

Lahres & Rayman (2009, Abschnitt 5.1.5) bewerten das explizite Konvertieren in einen spezielleren Typ als Notlösung, die durch ein gutes Programmdesign vermieden werden sollte.

6.7 Polymorphie (Methoden überschreiben)

Eine abgeleitete Klasse kann mit Hilfe gleich zu beschreibender Schlüsselwörter eine geerbte Instanzmethode *überschreiben*, statt sie zu *verdecken*. Wird ein Unterklassenobjekt über eine Variable vom Unterklassentyp referenziert, zeigt es bei überschreibenden *und* bei verdeckenden Methoden das Unterklassenverhalten. Bei Ansprache über eine Referenz vom *Basisklassentyp* gilt hingegen:

- Bei verdeckenden Methoden kommt die *Basisklassenvariante* zum Einsatz.
- Bei überschreibenden Methoden wird die *Unterklassenvariante* benutzt.

Während bei einer *Verdeckung* die auszuführende Methode schon beim Übersetzen festliegt, wird im Fall der *Überschreibung* erst zur Laufzeit mit Hilfe einer Verweistabelle die passende Methode gewählt. Man spricht hier von einer *dynamischen* oder *späten Bindung*. Grundsätzlich hat die späte Bindung einen erhöhten Zeitaufwand zur Folge, der jedoch kaum jemals praxisrelevant sein sollte.[1]

Werden Objekte aus verschiedenen Klassen über Referenzvariablen eines gemeinsamen Basistyps verwaltet, sind nur Methoden nutzbar, die schon in der Basisklasse definiert sind. Bei überschreibenden Methoden reagieren die Objekte aber jeweils unterklassentypisch auf dieselbe Botschaft. Genau dieses Phänomen bezeichnet man als **Polymorphie**. Wer sich hier mit einem exotischen und nutzlosen Detail konfrontiert glaubt, sei an die Auffassung von Alan Kay erinnert, der wesentlich zur Entwicklung der objektorientierten Programmierung beigetragen hat. Er zählt die Polymorphie neben der Datenkapselung und der Vererbung zu den Grundelementen dieser Softwaretechnologie (Lahres & Rayman 2009).

Zur Demonstration der Polymorphie definieren wir in der Basisklasse des Figurenbeispiels die Methode `Wo()` mit dem Modifikator **virtual** als überschreibbar:

```
using System;
public class Figur {
   protected double xpos = 100.0, ypos = 100.0;
   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() { }

   public virtual void Wo() {
      Console.WriteLine("\nOben Links:\t(" + xpos + ", " + ypos + ") ");
   }
}
```

[1] Siehe Einschätzungen in http://stackoverflow.com/questions/9937150/performance-impact-of-virtual-methods und Messungen (bei C++) in: http://stackoverflow.com/questions/449827/virtual-functions-and-performance-c

In der abgeleiteten Klasse `Kreis` wird mit dem Schlüsselwort **override** das Überschreiben der geerbte `Wo()` - Methode angeordnet:

```
using System;
public class Kreis : Figur {
   double radius = 75.0;
   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0)
         radius = rad;
   }
   public Kreis() { }

   public double Radius {
      get {return radius;}
      set {if (value >= 0.0) radius = value;}
   }

   public override void Wo() {
      base.Wo();
      Console.WriteLine("Unten Rechts:\t(" + (xpos + 2.0 * radius) +
                ", " + (ypos + 2.0 * radius) + ")");
   }
}
```

Ein Array vom Typ `Figur` kann nach den Erläuterungen in Abschnitt 6.6 Referenzen auf Figuren und Kreise aufnehmen, z.B.:

```
using System;
class Prog {
  static void Main() {
    Figur[] fa = new Figur[3];
    fa[0] = new Figur();
    fa[1] = new Kreis();
    for (int i = 0; i < 2; i++) {
      fa[i].Wo();
      if (fa[i] is Kreis)
        Console.WriteLine("Radius:         " + ((Kreis)fa[i]).Radius);
    }
    Console.Write("\nWollen Sie zum Abschluss noch eine Figur oder einen Kreis erleben?" +
      "\nWählen Sie durch Abschicken von \"f\" oder \"k\": ");
    if (Console.ReadLine().ToUpper()[0] == 'F')
      fa[2] = new Figur();
    else
      fa[2] = new Kreis();
    fa[2].Wo();
  }
}
```

Beim Ausführen der virtuellen und überschriebenen `Wo()` - Methode durch ein per Basisklassenreferenz angesprochenes Objekt stellt das Laufzeitsystem die tatsächliche Klassenzugehörigkeit (den *dynamischen* Typ der Referenzvariablen) fest und wählt die passende Methode aus:

```
Oben Links:     (100, 100)

Oben Links:     (100, 100)
Unten Rechts:   (250, 250)
Radius:         75

Wollen Sie zum Abschluss noch eine Figur oder einen Kreis erleben?
Wählen Sie durch Abschicken von "f" oder "k": k

Oben Links:     (100, 100)
Unten Rechts:   (250, 250)
```

Zum „Beweis“, dass tatsächlich eine späte Bindung stattfindet, darf im Beispielprogramm der Laufzeittyp des Array-Elements `fa[2]` vom Benutzer festgelegt werden.

Bei statischen Methoden sind die Modifikatoren **virtual** und **override** sinnlos und verboten. Das per **new**-Modifikator signalisierte Verdecken einer geerbten statischen Methode ist jedoch sinnvoll und erlaubt.

In der folgenden Tabelle werden die beiden Varianten der Ersetzung einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken und Überschreiben) in semantischer und syntaktischer Hinsicht gegenübergestellt:

Ersetzungsart	Unterstützung der Polymorphie	Syntax
Verdecken	Nein	Mit dem Modifikator **new** im Kopf der Unterklassenmethode wird unabhängig von der Basismethodendefinition das Verdecken gewählt. Ohne den Ersetzungsmodifikator **new** kommt es ebenfalls zu einer Verdeckung und außerdem zu einer Warnung des Compilers. Beim Methodenaufruf per Basisklassenreferenz kommt die Basisklassenvariante zum Einsatz. Auch statische Methoden können verdeckt werden.
Überschreiben	Ja	Im Kopf der Basisklassenmethode muss der Modifikator **virtual** stehen *und* im Kopf der Unterklassenmethode muss der Modifikator **override** stehen. Beim Methodenaufruf per Basisklassenreferenz kommt die Variante der abgeleiteten Klasse zum Einsatz. Statische Methoden können *nicht* überschrieben werden.

Eine virtuelle Basisklassenmethode kann also verdeckt oder überschrieben werden:

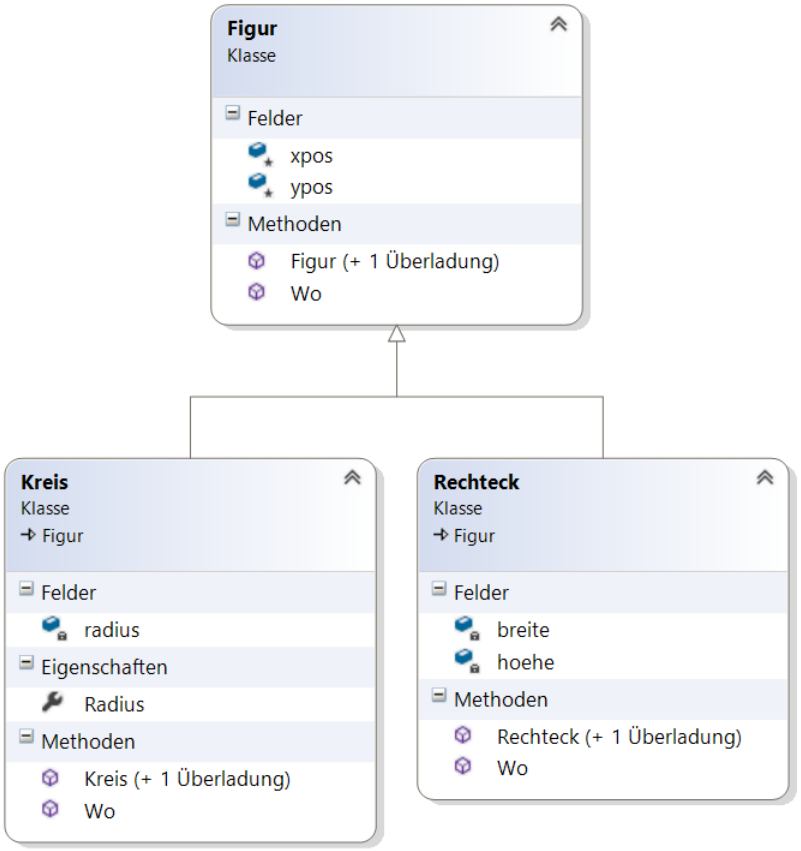
- **Überschreibt** eine abgeleitete Klasse die Methode (Modifikator **override**), ist sie auch in der abgeleiteten Klasse virtuell (mit Bedeutung für die nächste Ableitungsgeneration). Den Modifikator **virtual** zusammen mit dem Modifikator **override** anzugeben, ist überflüssig und verboten.
- **Verdeckt** eine abgeleitete Klasse die Methode (Modifikator **new**), ist sie in der abgeleiteten Klasse nicht mehr virtuell, sofern nicht gleichzeitig auch der Modifikator **virtual** vergeben wird.

Bei einer *nicht*-virtuellen Basisklassenmethode ist nur das Verdecken möglich.

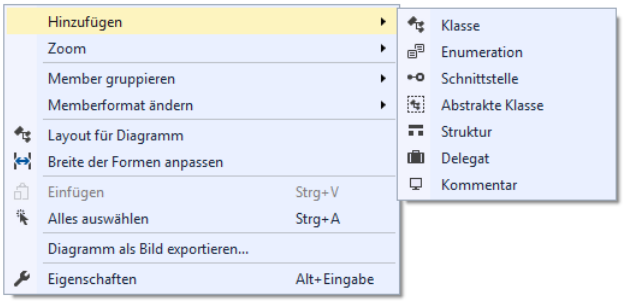
Beide Ersetzungsarten sind auch bei Eigenschaften und Indexern anwendbar.

6.8 Klassendiagramme mit Vererbungsbeziehung

Unsere Entwicklungsumgebung Visual Studio 2015 Community kann mit wenigen Mausaktionen zur Erstellung des folgenden Klassendiagramms für das Figurenbeispiel veranlasst werden, wobei die Vererbungs- bzw. Spezialisierungsbeziehungen zwischen den Klassen nach den Regeln der UML-Notation (*Unified Modeling Language*) dargestellt werden:



Um das Klassendiagramm zu erstellen, markiert man im Projektmappen-Explorer die beteiligten Klassen und wählt aus dem Kontextmenü der Markierung das Item **Klassendiagramm anzeigen**. Das Kontextmenü zum Fenster mit dem Klassendiagramm erlaubt einige Modifikationen und den Export des Diagramms in eine Datei mit wählbarem Format (z.B. EMF):



6.9 Abstrakte Methoden und Klassen

Um die eben beschriebene gemeinsame Verwaltung von Objekten aus diversen Unterklassen über Referenzvariablen vom Basisklassentyp realisieren und dabei Polymorphie nutzen zu können, müssen die beteiligten Methoden in der Basisklasse vorhanden sein. Wenn es für die Basisklasse zu einer Methode keine sinnvolle Implementierung gibt, erstellt man dort eine **abstrakte** Methode:

- Man beschränkt sich auf den Methodenkopf und setzt dort den Modifikator **abstract**.
- Den Methodenrumpf ersetzt man durch ein Semikolon.

Im Figurenbeispiel ergänzen wir eine Methode namens `Skaliere()`, mit der eine Figur zu artspezifischem Wachsen (oder Schrumpfen) um den per Parameter festgelegten Faktor aufgefordert werden kann. Ein Kreis wird auf diese Botschaft hin seinen Radius verändern, während ein Rechteck Breite und Höhe anzupassen hat. Weil die Methode in der Basisklasse `Figur` nicht sinnvoll realisierbar ist, wird sie hier abstrakt definiert:

```
public abstract class Figur {
   . . .
   public abstract void Skaliere(double faktor);
   . . .
}
```

Enthält eine Klasse mindestens *eine* abstrakte Methode, dann handelt es sich um eine **abstrakte Klasse**, und im Klassendefinitionskopf muss der Modifikator **abstract** angegeben werden.

Abstrakte Methoden sind grundsätzlich virtuell (vgl. Abschnitt 6.7), wobei das Schlüsselwort **virtual** überflüssig und verboten ist.

Von einer abstrakten Klasse lassen sich keine Objekte erzeugen, aber man kann andere Klassen daraus ableiten. Implementiert eine abgeleitete Klasse die abstrakten Methoden, lassen sich Objekte von dieser Klasse herstellen; anderenfalls ist sie ebenfalls abstrakt.

Wir leiten aus der nunmehr abstrakten Klasse `Figur` die konkreten Klassen `Kreis` und `Rechteck` ab, welche die abstrakte `Figur`-Methode `Skaliere()` implementieren:

```
public class Kreis : Figur {
   double radius = 75.0;
   . . .
   public override void Skaliere(double faktor) {
      if (faktor >= 0.0)
         radius *= faktor;
   }
   . . .
}

public class Rechteck : Figur {
   double breite = 50.0, hoehe = 50.0;
   . . .
   public override void Skaliere(double faktor) {
      if (faktor >= 0.0) {
         breite *= faktor;
         hoehe *= faktor;
      }
   }
   . . .
}
```

Von den beiden Varianten der Ersetzung einer Basisklassenmethode durch eine signaturgleiche Unterklassenmethode (Verdecken und Überschreiben, vgl. Abschnitt 6.7) kommt bei einer abstrakten Basisklassenmethode nur das Überschreiben in Frage, wobei das zugehörige Schlüsselwort **override** anzugeben ist.[1]

Neben den Methoden können auch Eigenschaften und Indexer abstrakt definiert werden. Im Figurenbeispiel soll mit der Eigenschaft `Inhalt` die Möglichkeit geschaffen werden, den Flächeninhalt eines Objekts zu erfragen. Weil eine polymorphe Nutzung gewünscht ist, muss die Eigenschaft schon in der Basisklasse vorhanden sein. Dort ist aber keine sinnvolle Flächenberechnung möglich, sodass die Eigenschaft abstrakt definiert wird:

```
public abstract double Inhalt {
   get;
}
```

[1] Das Verdecken scheidet aus, weil beim Methodenaufruf per Basisklassenreferenz die nicht implementierte Variante der Basisklasse verwendet werden müsste.

Im Definitionskopf ist der Modifikator **abstract** anzugeben, und bei der **get**-Methode ersetzt ein Semikolon die Implementation. Analog könnte auch eine **set**-Methode abstrakt definiert werden.

In den abgeleiteten Klassen `Kreis` und `Rechteck` wird die Eigenschaft `Inhalt` individuell realisiert:

```
public class Kreis : Figur {
   double radius = 75.0;
   . . .
   public override double Inhalt {
      get {return Math.PI * radius * radius;}
   }
   . . .
}

public class Rechteck : Figur {
   double breite = 50.0, hoehe = 50.0;
   . . .
   public override double Inhalt {
      get {return breite * hoehe;}
   }
   . . .
}
```

Obwohl sich aus einer abstrakten Klasse keine Objekte erzeugen lassen, kann sie doch als Datentyp verwendet werden. Referenzen dieses Typs sind ja auch unverzichtbar, wenn Objekte diverser Unterklassen gemeinsam verwaltet werden sollen, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `   Figur[] fa = new Figur[2];` `   fa[0] = new Kreis(50.0, 50.0, 5.0);` `   fa[1] = new Rechteck(10.0, 10.0, 5.0, 5.0);` `   fa[0].Skaliere(2.0);` `   fa[1].Skaliere(2.0);` `   double ges = 0.0;` `   for (int i = 0;  i < fa.Length; i++) {` `    Console.WriteLine("Fläche Figur {0}: {1,10:f2}\n",` `                      i, fa[i].Inhalt);` `    ges += fa[i].Inhalt;` `   }` `   Console.WriteLine("Gesamtfläche:   {0,10:f2}",ges);` `  }` `}`	`Fläche Figur 0:     314,16` `Fläche Figur 1:     100,00` `Gesamtfläche:       414,16`

Mit Hilfe der in Abschnitt 8.4 vorzustellenden *Schnittstellen* werden wir noch mehr Flexibilität gewinnen und polymorphe Methodenaufrufe auch für Typen *ohne* gemeinsame Basisklasse realisieren.

6.10 Vertiefung: Das Liskovsche Substitutionsprinzip (LSP)

Nachdem wir uns mit den konkreten Vorteilen von abstrakten Klassen und polymorphen Methodenaufrufen beschäftigt haben, muten wir uns in diesem Abschnitt eine etwas formale, aber keinesfalls praxisfremde Vertiefung zum Thema *Vererbung* zu. Das nach Barbara Liskov benannte Substitutionsprinzip verlangt von einer Klassenhierarchie (Liskov & Wing 1999, S. 1):

> Let $\phi(x)$ be a property provable about objects x of type T. Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.

Wird beim Entwurf einer Klassenhierarchie das Liskovsche Substitutionsprinzip (LSP) beachtet, dann können Objekte einer abgeleiteten Klasse stets die Rolle von Basisklassenobjekten perfekt übernehmen, d.h. u.a.:

- Das „vertraglich“ zugesicherte Verhalten der Basisklassenmethoden wird auch von den überschreibenden Unterklassenvarianten eingehalten.
- Unterklassenobjekte werden bei Verwendung in der Rolle von Basisklassenobjekten nicht beschädigt.

Eine Verletzung der Ersetzbarkeitsregel kann auch bei einfachen Beispielen auftreten, wobei oft eine aus dem Anwendungsbereich stammende Plausibilität zum fehlerhaften Design verleitet. So ist z.B. ein Quadrat aus mathematischer Sicht ein spezielles Rechteck. Definiert man in einer Klasse für Rechtecke die Methoden `SkaliereX()` und `SkaliereY()` zur Änderung der Länge in X- bzw. Y-Richtung, so gehört zum „vertraglich“ zugesicherten Verhalten dieser Methoden:

- Bei einem Zuwachs in X-Richtung bleibt die Y-Ausdehnung unverändert.
- Verdoppelt man die Breite eines Objekts, verdoppelt sich auch der Flächeninhalt.

Die simple Tatsache, dass aus mathematischer Perspektive jedes Quadrat ein Rechteck ist, rät offenbar dazu, eine Klasse für Quadrate aus der Klasse für Rechtecke abzuleiten. In der neuen Klasse ist allerdings die Konsistenzbedingung zu ergänzen, dass bei einem Quadrat stets alle Seiten gleich lang bleiben müssen. Um das Auftreten irregulärer Objekte der Klasse `Quadrat` zu verhindern, wird man z.B. die Methode `SkaliereX()` so überschreiben, dass bei einer X-Modifikation automatisch auch die Y-Ausdehnung angepasst wird. Damit ist aber der `SkaliereX()` - Vertrag verletzt, wenn ein Quadrat die Rechteckrolle übernimmt. Eine verdoppelte X-Länge führt etwa nicht zur doppelten, sondern zur vierfachen Fläche. Verzichtet man andererseits in der Klasse `Quadrat` auf das Überschreiben der Methode `SkaliereX()`, ist bei den Objekten dieser Klasse die Konsistenzbedingung identischer Seitenlängen massiv gefährdet. Offenbar haben Plausibilitätsüberlegungen zu einer schlecht entworfenen Klassenhierarchie geführt.

Eine exakte Verhaltensanalyse zeigt, dass ein Quadrat in funktionaler Hinsicht eben doch kein Rechteck ist. Es fehlt die für Rechtecke typische Option, die Ausdehnung in X- bzw. Y-Richtung separat zu verändern. Diese Option könnte in einem Algorithmus, der den Datentyp `Rechteck` voraussetzt, von Bedeutung sein. Es muss damit gerechnet werden, dass der Algorithmus irgendwann (bei einer Erweiterung der Software) auf Objekte mit einem von `Rechteck` abstammenden Datentyp trifft. Passiert dies mit der Klasse `Quadrat` könnte es zum Crash kommen, weil z.B. ein automatisch an die Länge eines Transporters angepasstes Objekt unbemerkt an Höhe zulegt und unterwegs gegen eine Brücke stößt.

Derartige Designfehler können vom Compiler nicht verhindert werden. C# bietet alle Voraussetzungen für eine erfolgreiche objektorientierte Programmierung, kann aber z.B. eine Verletzung der Ersetzbarkeitsregel nicht verhindern.

6.11 Versiegelte Methoden und Klassen

Gelegentlich möchte man das Überschreiben einer Methode *verhindern*, damit auch ein per Basisklassenreferenz angesprochenes Unterklassenobjekt das Originalverhalten zeigt. Dient etwa die Methode `Passwd()` einer Klasse `B` zum Abfragen eines Passworts, will der Programmierer eventuell verhindern, dass `Passwd()` in einer von `B` abstammenden Klasse `C` überschrieben wird. Damit führt auch ein per `B`-Referenz angesprochenes `C`-Objekt garantiert die `B`-Methode `Passwd()` aus.

Um das Überschreiben einer Methode zu verhindern, gibt man in der Definition den Modifikator **sealed** (dt.: *versiegelt*) an. Dies ist allerdings nur bei Methoden möglich, die eine virtuelle Methode überschreiben, also auch den Modifikator **override** besitzen, z.B.:

```
class A {
  public virtual void Passwd() { }
}

class B : A {
  public sealed override void Passwd() { }
}
```

Dass für die zunächst etwas willkürlich erscheinende Kopplung der Schlüsselwörter **sealed** und **override** gute Gründe sprechen, wird durch die Betrachtung der folgenden Fälle klar:

- In `B` wird eine Methode oder eine Methodenüberladung erstellt, die in `A` fehlt:
  Um das Überschreiben zu verhindern, lässt man einfach den Modifikator **virtual** weg.
- In `B` wird eine in `A` vorhandene und dort *nicht* als **virtual** definierte Methode verdeckt, wobei mit dem optionalen Modifikator **new** eine Compiler-Warnung verhindert werden sollte:
  Um das Überschreiben der `B`-Methode zu verhindern, lässt man einfach den Modifikator **virtual** weg.
- In `B` wird eine in `A` vorhandene und dort als **virtual** definierte Methode verdeckt, wobei mit dem optionalen Modifikator **new** eine Compiler-Warnung verhindert werden sollte:
  Die `B`-Methode ist nicht virtuell, und man kann das Überschreiben wiederum einfach dadurch verhindern, dass man der Modifikator **virtual** weglässt.
- In `B` wird eine in `A` vorhandene und dort als **virtual** definierte Methode überschrieben, was durch den Modifikator **override** deklariert werden muss:
  In diesem Fall ist die `B`-Methode virtuell, und der Modifikator **sealed** ist erforderlich, wenn das Überschreiben verhindert werden soll.

Die Aussagen zum Versiegeln von Methoden gelten analog für Eigenschaften und Indexer.

Sicherheitsüberlegungen können auch zum Entschluss führen, eine komplette **Klasse** mit dem Schlüsselwort **sealed** zu fixieren, so dass sie zwar verwendet, aber nicht beerbt werden kann. Microsoft nennt als möglichen Grund die Existenz von sicherheitsrelevanten Geheimnissen in geerbten Mitgliedern mit Schutzstufe **protected** (kann nicht eingeschränkt werden).[1]

Für das Versiegeln einer Klasse können aber noch weitere Gründe sprechen, wobei die oft genannten Performanzvorteile von versiegelten Klassen vermutlich wenig relevant sind. Von den versiegelten FCL-Klassen ist **String** das bekannteste Beispiel:

```
public sealed class String ...
```

In diesem Fall wird durch das Versiegeln die *Unveränderlichkeit* von **String**-Objekten sichergestellt, die z.B. für den internen **String**-Pool wichtig ist (siehe Abschnitt 5.4.1.3) und außerdem von vielen (Bibliotheks)klassen vorausgesetzt wird.

6.12 Erweiterungsmethoden

Soll eine Klasse um zusätzliche Handlungskompetenzen erweitert werden, erstellt man üblicherweise eine abgeleitete Klasse. Seit der C# - Version 3.0 steht eine alternative Erweiterungstechnik zur Verfügung für Situationen, in denen das Ableiten eines neuen Typs nicht möglich ist (z.B. bei Strukturen oder versiegelten Klassen).

Ihre vermutlich wichtigste Anwendung finden Erweiterungsmethoden bei den Standardabfragen der LINQ-Technik (*Language Integrated Query*), doch können Erweiterungsmethoden auch unabhängig von der LINQ-Technik erfolgreich eingesetzt werden.

---

[1] https://msdn.microsoft.com/en-us/library/ms229023(v=vs.110).aspx

6.12.1 Technische Realisation

Um die Instanzen eines vorhandenen Typs mit zusätzlichen Handlungskompetenzen auszustatten, definiert man eine statische Klasse und darin statische Methoden, die einen ersten, über das Schlüsselwort **this** besonders gekennzeichneten Parameter vom zu erweiternden Typ besitzen. Bei den Erweiterungsmethoden ist also durchaus eine neue Klasse erforderlich, doch können den Instanzen des zu erweiternden Typs die neuen Botschaften (Methoden) syntaktisch genauso zugestellt werden wie die typeigenen.

Der Compiler erstellt jedoch statische Methodenaufrufe, und der Zugriffsschutz wird nicht verletzt, weil eine Erweiterungsmethode keinen Zugriff auf private Member (z.B. Methoden, Instanzvariablen) des erweiterten Typs hat.

Durch die Erweiterungsmethode `Empty()` für die Klasse **String** wird festgestellt, ob ein Objekt vorhanden ist, aber keine Zeichen enthält:

Quellcode-Segment	Ausgabe

```
// Datei StringExt.cs
using System;

namespace StringExt {
   public static class StringExt {
      public static bool Empty(this String arg) {
         return (arg != null && arg.Length == 0) ? true : false;
      }
   }
}

// Datei Prog.cs
using System;
using StringExt;

class Prog {
   static void Main() {
      Console.WriteLine("".Empty());
      Console.WriteLine("m".Empty());
   }
}
```

```
True
False
```

Damit eine Erweiterungsmethode in einer Quellcodedatei verfügbar ist, muss der Namensraum mit der definierenden Klasse per **using**-Direktive importiert werden.

Tritt eine Erweiterungsmethode in Konkurrenz mit einer typeigenen, dann gewinnt letztere. Folglich besteht ein erhebliches Risiko beim Einsatz von Erweiterungsmethoden. Wenn z.B. in einer späteren Version der Klasse **String** die Instanzmethode **Empty()** mit identischer Signatur hinzukommt, wird diese bei einem Aufruf gegenüber der Erweiterungsmethode `Empty()` bevorzugt, und ein für die Benutzung der Erweiterungsmethode konzipiertes Programm zeigt vermutlich nicht mehr das intendierte Verhalten.

Im folgenden Beispiel wird für Instanzen der Struktur **Double** das Potenzieren durch die Erweiterungsmethode `H()` syntaktisch vereinfacht:

Quellcode-Segment	Ausgabe
`// Datei MathExt.cs` `using System;` `namespace MathExt {` `  public static class Mathe {` `  public static double H(this double arg, double expo) {` `  return Math.Pow(arg, expo);` `  }` `  }` `}` `// Datei Prog.cs` `using System;` `using MathExt;` `class Prog {` `  static void Main() {` `  double pi = 3.14;` `  Console.WriteLine(pi.H(2));` `  }` `}`	`9,8596`

6.12.2 Anwendungsempfehlungen

Um die Funktionalität einer Klasse zu erweitern, sollte nach einer Empfehlung von Microsoft nach Möglichkeit eine abgeleitete Klasse definiert werden.[1] Erweiterungsmethoden sollten nur dann in Erwägung gezogen werden, wenn die Definition eines abgeleiteten Typs ausgeschlossen ist (bei einer versiegelten Klasse oder bei einer Struktur). Dabei muss das Risiko in Kauf genommen werden, dass eine neue Version des erweiterten Typs eine Instanzmethode erhält, welche die Erweiterungsmethode aus dem Spiel nimmt.

6.13 Übungsaufgaben zu Kapitel 6

1) Warum kann der folgende Quellcode nicht übersetzt werden?

```
using System;
class Basisklasse {
  int ibas = 3;
  public Basisklasse(int i) { ibas = i; }
  public virtual void Hallo() {
  Console.WriteLine("Hallo-Methode der Basisklasse");
  }
}
class Abgeleitet : Basisklasse {
  public override void Hallo() {
  Console.WriteLine("Hallo-Methode der abgeleiteten Klasse");
  }
}

class Prog {
  static void Main() {
  Abgeleitet s = new Abgeleitet();
  s.Hallo();
  }
}
```

[1] https://msdn.microsoft.com/de-de/library/bb383977.aspx

2) Im folgenden Beispiel wird die Klasse `Kreis` aus der Klasse `Figur` abgeleitet:

```
class Figur {
    double xpos = 100.0, ypos = 100.0;
    public Figur(double x, double y) {
        xpos = (x >= 0) ? x : 0;
        ypos = (y >= 0) ? y : 0;
    }
    public Figur() { }
}

class Kreis : Figur {
    double radius = 75.0;
    public Kreis(double x, double y, double rad) {
        xpos = (x>=0)?x:0;
        ypos = (y>=0)?y:0;
        radius = (rad>=0)?rad:0;
    }
    public Kreis() { }
}
```

Trotzdem erlaubt der Compiler im initialisierenden `Kreis`-Konstruktor den `Kreis`-Objekten keinen direkten Zugriff auf ihre geerbten Instanzvariablen `xpos` und `ypos`. Wie ist das Problem zu erklären und zu lösen?

3) Erläutern Sie die folgenden Begriffe:

- Überladen von Methoden
- Verdecken von Methoden
- Überschreiben von Methoden

Welche von den drei genannten Programmiertechniken ist bei statischen Methoden *nicht* anwendbar?

7 Typgenerisches Programmieren und Kollektionen

In C# haben die Felder von Klassen und Strukturen sowie die Parameter von Methoden einen festen Datentyp, so dass der Compiler für Typsicherheit sorgen, d.h. die Zuweisung ungeeigneter Werte bzw. Objekte verhindern kann. Oft werden für unterschiedliche Datentypen völlig analog arbeitende Klassen, Strukturen oder Methoden benötigt, z.B. eine Klasse zur Verwaltung einer geordneten Liste mit Elementen eines bestimmten Typs. Statt die Definition für jeden in Frage kommenden Elementdatentyp zu wiederholen, kann man die Definition *typgenerisch* formulieren. Bei der *Verwendung* einer generischen Listenklasse ist der zu verarbeitende Elementtyp konkret festzulegen. Im Ergebnis erhält man durch *eine* Definition zahlreiche konkrete Klassen, wobei die Typsicherheit nicht abgeschwächt wird.

Ein besonders erfolgreiches Anwendungsfeld für Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen oder (Schlüssel-Wert) - Tabellen. Für die Objekte dieser Klassen wird im Manuskript alternativ zur offiziellen Bezeichnung *Kollektion* aus sprachlichen Gründen gelegentlich auch die Bezeichnung *Container* verwendet. Dass an anderer Stelle auch von Containern zur Verwaltung von GUI-Bedienelementen die Rede ist, sollte keine Verwirrung stiften.

Typgenerische Definitionen eignen sich nicht nur für Klassen, Strukturen und Methoden, sondern auch für die später zu behandelnden Schnittstellen und Delegaten.

7.1 Motive für generische Typen

In Abschnitt 5.3.10 haben wir die Klasse **ArrayList** aus Namensraum **System.Collections** als Container für Objekte beliebigen Typs verwendet:[1]

```
var al = new ArrayList();
al.Add("Text");
al.Add(3.14);
al.Add(13);
```

Im Unterschied zu einem Array (siehe Abschnitt 5.3) bietet die Klasse **ArrayList**:

- eine automatische Größenanpassung
- Typflexibilität (durch Verwendung des Elementtyps **Object**)

Während das obige Beispiel die Größen- *und* die Typflexibilität nutzt, ist oft ein Container mit automatischer Größenanpassung (ein dynamischer Array) für Objekte eines bestimmten, *festen* Typs gefragt (z.B. zur Verwaltung von **String**-Objekten). Bei dieser Einsatzart stören zwei Nachteile der Typbeliebigkeit:

- Wenn beliebige Objekte zugelassen sind, die intern über Referenzvariablen vom Typ **Object** verwaltet werden, kann der Compiler nicht überwachen, dass ausschließlich Objekte des gewünschten Typs in den Container eingefüllt werden. Viele Programmierfehler werden erst zur Laufzeit entdeckt.
- Entnommene Objekte können erst nach einer expliziten Typumwandlung die Methoden ihres Klasse ausführen. Die häufig benötigten Typanpassungen sind lästig und fehleranfällig.
- Wenn Variablen mit Werttyp verwaltet werden, resultieren leistungsschädliche (Un)Boxing-Operationen.

Im folgenden Beispielprogramm sollen **String**-Objekte in einem **ArrayList**-Container verwaltet werden:

---

[1] Bei der Deklaration einer lokalen Variablen mit Initialisierung kann man es sich über das Schlüsselwort **var** (die Typinferenz des Compilers ausnutzend) ersparen, den länglichen Klassennamen zweimal schreiben zu müssen (vgl. Abschnitt 3.3.6).

```
using System;
using System.Collections;
class Prog {
 static void Main() {
  var al = new ArrayList();
  al.Add("Otto");
  al.Add("Rempremerding");
  al.Add('.');
  int i = 0;
  foreach (Object s in al)
    Console.WriteLine($"Länge von Element {++i}: {((String)s).Length}");
 }
}
```

Bevor ein **String**-Element des Containers nach seiner Länge befragt werden kann, ist eine lästige Typanpassung fällig, weil der Compiler nur den deklarierten Typ **Object** kennt:

```
((String)s).Length
```

Beim dritten **Add()** - Aufruf wird eine Instanz vom Typ **char** per Autoboxing in den Container befördert. Weil der Container eigentlich zur Aufbewahrung von **String**-Objekten gedacht war, liegt hier ein Programmierfehler vor, den der Compiler wegen der fehlenden Typsicherheit nicht bemerken kann. Beim Versuch, die **char**-Instanz als **String**-Objekt zu behandeln, scheitert das Programm am folgenden Ausnahmefehler:

```
Unbehandelte Ausnahme: System.InvalidCastException: Das Objekt des Typs
"System.Char" kann nicht in Typ "System.String" umgewandelt werden.
   bei Prog.Main() in Prog.cs:Zeile 11.
```

Es ist nicht schwer, eine spezialisierte Container-Klasse zur Verwaltung von **String**-Objekten zu definieren, um die beiden Probleme (syntaktische Umständlichkeit, mangelnde Typsicherheit) zu vermeiden. Vermutlich werden analoge funktionierende Behälter aber auch für alternative Elementtypen benötigt, und entsprechend viele Klassen zu definieren, die sich nur durch den Inhaltstyp unterscheiden, wäre nicht rationell. Für eine solche Aufgabenstellung bietet C# seit der Version 2.0 die generischen Klassen. Durch Verwendung von Typparametern bei der Definition wird die gesamte Handlungskompetenz einer Klasse typunabhängig formuliert. Bei jedem Instanzieren (z.B. beim Erstellen eines Container-Objekts) sind jedoch konkrete Typen anzugeben. Weil der Compiler die konkreten Typen kennt, kann er bei Zuweisungen für Typsicherheit sorgen, und es sind keine lästigen Typumwandlungen erforderlich.

Mit der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** enthält die FCL seit der Version 2.0 eine perfekte **ArrayList**-Alternative zur Verwaltung einer dynamischen Liste von Elementen mit identischem Typ. Das obige Beispielprogramm ist schnell auf die neue Technik umgestellt:

```
using System;
using System.Collections.Generic;
class Prog {
  static void Main() {
    var gl = new List<String>();
    gl.Add("Otto");
    gl.Add("Rempremerding");
    gl.Add(".");
    int i = 0;
    foreach (String s in gl)
      Console.WriteLine($"Länge von Element {++i}: {s.Length}");
  }
}
```

Bei der Erstellung eines **List<T>** - Objekts ist an Stelle des Typparameters **T** ein konkreter Datentyp anzugeben, z.B.:

```
var gl = new List<String>();
```

Jeder Versuch, ein Element mit abweichendem Typ in den Container einzufügen, wird vom Compiler bemerkt und verhindert, z.B.:

```
gl.Add('.');
```

Die Elemente des auf **String**-Objekte spezialisierten Containers beherrschen ohne Typanpassung die Methoden ihrer Klasse, z.B.:

```
foreach (String s in gl)
  Console.WriteLine($"Länge von Element {++i}: {s.Length}");
```

Bei einem dynamischen Container für Elemente mit einem festen *Werttyp* erspart die Klasse **List<T>** im Vergleich zu **ArrayList** zudem die zeitaufwändigen (Un)boxing-Operationen. Mit diesem Thema sollen Sie sich im Rahmen einer Übungsaufgabe beschäftigen.

Für Container zur Aufnahme von Elementen mit *unterschiedlichen* Typen ist die Klasse **ArrayList** weiterhin gefragt.

7.2 Generische Klassen

Aus der Entwicklerperspektive besteht der wesentliche Vorteil einer generischen Klasse darin, dass mit *einer* Definition beliebig viele konkrete Klassen für spezielle Datentypen geschaffen werden. Dieses Konstruktionsprinzip ist speziell bei den Kollektionsklassen sehr verbreitet (siehe FCL-Namensraum **System.Collections.Generic**), aber keinesfalls auf Container mit ihrer weitgehend inhaltstypunabhängigen Verwaltungslogik beschränkt.

Analog zu den generischen Klassen bietet C# auch generische Strukturen, Schnittstellen und Delegaten. Wir befassen uns in Kapitel 7 hauptsächlich mit generischen Klassen und Strukturen, machen aber auch schon erste Erfahrungen mit generischen Schnittstellen, die wir in Kapitel 8 vertiefen werden.

7.2.1 Definition

Bei der generischen Klassendefinition verwendet man **Typformalparameter**, die im Kopf der Definition hinter dem Klassennamen zwischen spitzen Klammern und durch Kommata getrennt angegeben werden. Wir erstellen als Beispiel eine generische Klasse namens `SimpleStack<T>`, die einen LIFO-Stapel (*last-in-first-out*) verwaltet und mit *einem* Typformalparameter für den beliebig wählbaren Elementtyp auskommt. In der Praxis wird man für eine solche Standardaufgabe allerdings die Klasse **Stack<T>** aus dem FCL-Namensraum **System.Collections.Generic** verwenden.

```
public class SimpleStack<T> {
   int capacity = 5;
   T[] data;
   int size;

   public SimpleStack() {
      data = new T[capacity];
   }
   public SimpleStack(int max) {
      if (max > 0)
         capacity = max;
      data = new T[capacity];
   }
```

```
    public int Count {
       get { return size; }
    }

    public bool Push(T element) {
       if (size < capacity) {
          data[size++] = element;
          return true;
       } else
          return false;
    }

    public T Pop() {
       if (size > 0)
          return data[--size];
       else
          throw new System.InvalidOperationException();
    }
}
```

Mit der Methode `Push()` legt man ein neues Element auf den Stapel, sofern seine Kapazität noch nicht erschöpft ist.

Solange der Vorrat reicht, kann man das jeweils oberste Element mit der Methode `Pop()` abheben. Ist der Stapel leer, wirft die Methode `Pop()` dem Aufrufer per **throw**-Anweisung ein Ausnahmeobjekt aus der Klasse **InvalidOperationException** zu, um ihn über das Problem zu informieren. Nachdem Sie im Vorgriff auf das noch ausstehende Kapitel über die Ausnahmebehandlung schon mehrfach die Rolle des potentiellen *Empfängers* von Ausnahmeobjekten beobachten konnten, erleben Sie nun einen Ausblick auf die Rolle des *Senders* von Ausnahmeobjekten.

Innerhalb der Klassendefinition wird der Typformalparameter wie ein Datentyp verwendet, z.B.:

- als Elementtyp für den internen Array mit den Daten des Stapels
- als Datentyp für den Formalparameter der Methode `Push()`
- als Datentyp für die Rückgabe der Methode `Pop()`

Vielleicht vermissen Sie beim `SimpleStack<T>` die Größendynamik der professionellen Kollektionsklassen (z.B. **Stack<T>**). Um das automatische Wachsen zu realisieren, könnte man den intern zur Datenspeicherung benutzten Array bei Bedarf durch ein größeres Exemplar (z.B. mit doppelter Länge) ersetzen und die bisherigen Elemente kopieren. Im Beispiel wird der Einfachheit halber auf die Größendynamik verzichtet.

In Abschnitt 7.6.4 werden Sie mit der FCL-Klasse **Dictionary<K, V>** ein Beispiel für eine generische Klasse mit *zwei* Typformalparametern kennen lernen.

Der Vollständigkeit halber sei noch erwähnt, dass innerhalb eines Namensraums verschiedene generische Typen denselben Namen verwenden dürfen, solange ihre Typformalparameterlisten verschieden lang sind. Auch ein namensgleicher nichtgenerischer Typ ist erlaubt.

7.2.2 Restringierte Typformalparameter

Häufig muss eine generische Klassendefinition bei den Typen, die einen Typparameter konkretisieren dürfen, gewisse Handlungskompetenzen voraussetzen. Muss z.B. ein generischer Container seine Elemente sortieren, verlangt man in der Regel von einem konkreten Elementtyp, dass er die *Schnittstelle* **IComparable<T>** erfüllt, d.h. eine Methode namens **CompareTo()** mit dem folgenden Definitionskopf besitzt (hier beschrieben unter Verwendung des Typparameters **T**):

public int CompareTo(T *element***)**

In Abschnitt 5.4.1.2.2 haben Sie erfahren, dass die Klasse **String** eine solche Methode besitzt, und wie **CompareTo()** das Prüfergebnis über den Rückgabewert signalisiert:

		CompareTo() - Ergebnis
Das befragte Objekt ist im Vergleich zum Aktualparameter …	kleiner	-1
	gleich	0
	größer	1

Damit sollte klar genug sein, was die Schnittstelle (das Interface) **IComparable<T>** von einem implementierenden Typ verlangt. Mit dem generellen Thema *Schnittstellen* werden wir uns in Kapitel 8 ausführlich beschäftigen. Dabei wird sich herausstellen, dass zur generischen Schnittstelle **IComparable<T>** auch noch die ältere, nichtgenerische Variante **IComparable** existiert, die eine Methode

public int CompareTo(Object *element***)**

vorschreibt. Weil die generische Variante eine Typprüfung durch den Compiler ermöglicht, ist sie zu bevorzugen.

Wir erstellen nun eine generische Listenverwaltungsklasse, die eingefügte Elemente automatisch einsortiert und daher ihren Typformalparameter auf den Schnittstellentyp **IComparable<T>** einschränkt:

```
public class SimpleSortedList<T> where T : System.IComparable<T> {
   int capacity = 5;
   T[] data;
   int size;

   public SimpleSortedList() {
      data = new T[capacity];
   }

   public SimpleSortedList(int len) {
      if (len > 0)
         capacity = len;
      data = new T[capacity];
   }

   public bool Add(T element) {
      if (size == data.Length)
         return false;
      bool inserted = false;
      for (int i = 0; i < size; i++) {
         if (element.CompareTo(data[i]) <= 0) {
            for (int j = size; j > i; j--)
               data[j] = data[j-1];
            data[i] = element;
            inserted = true;
            break;
         }
      }
      if (!inserted)
         data[size] = element;
      size++;
      return true;
   }
```

```
    public T this[int index] {
       get {
          if (index < 0 || index >= size)
             throw new System.ArgumentOutOfRangeException();
          return data[index];
       }
       set {
          if (index < 0 || index >= size)
             throw new System.ArgumentOutOfRangeException();
          data[index] = value;
       }
    }
 }
```

Im Indexer (vgl. Abschnitt 5.6) erlauben wir uns erneut einen Vorgriff auf das Kapitel über Ausnahmefehler und informieren den Aufrufer über einen ungültigen Index, indem wir eine **ArgumentOutOfRangeException** werfen.

Bei der Formulierung von Einschränkungen für einen Typparameter wird das Schlüsselwort **where** verwendet, wobei u.a. folgende Regeln gelten:

- Man kann eine Basisklasse vorschreiben.
- Mit dem Schlüsselwort **class** wird vereinbart, dass nur Referenztypen erlaubt sind.
- Mit dem Schlüsselwort **struct** wird vereinbart, dass nur Werttypen erlaubt sind.
- Man kann auch *mehrere* Restriktionen durch Kommata getrennt angeben, die von einem konkreten Typ allesamt zu erfüllen sind. Die Schlüsselwörter **class** und **struct** müssen ggf. am Anfang einer Liste von Restriktionen stehen.
- Während nur *eine* Basisklasse vorgeschrieben werden darf, sind beliebig viele Schnittstellen (vgl. Kapitel 8) erlaubt, die ein konkreter Typ *alle* erfüllen muss.
- Mit dem Listeneintrag **new()** wird für die konkreten Typen ein parameterfreier Konstruktor vorgeschrieben. In einer *Liste* von Restriktionen muss der Eintrag **new()** ggf. am Ende stehen.
- Sind mehrere Typformalparameter mit Restriktionen vorhanden, ist für jeden Parameter eine eigene **where**-Klausel anzugeben.

Trotz der gebotenen Vielfalt bei der Formulierung von Typrestriktionen, bestehen doch Lücken. So ist z.B. keine Einschränkung auf *numerische* Werttypen möglich. Weitere Details zu den Optionen und Motiven für Typrestriktionen finden sich z.B. bei Griffiths (2013, S. 136ff) und Richter (2006, S. 394ff).

Im folgenden Programm wird aus der generischen Klasse `SimpleSortedList<T>` eine konkrete Klasse zur Verwaltung einer sortierten **int**-Liste erzeugt:

Quellcode	Ausgabe
`class Prog {` `  static void Main() {` `  var si = new SimpleSortedList<int>(5);` `  si.Add(11); si.Add(2); si.Add(1); si.Add(4);` `  for (int i = 0; i < 4; i++)` `  System.Console.WriteLine(si[i]);` `  }` `}`	`1` `2` `4` `11`

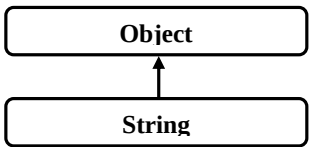
7.2.3 Generische Klassen und Vererbung

Bei der Definition einer generischen Klasse kann man als Basisklasse verwenden:

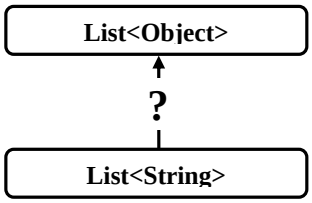
- eine nicht-generische Klasse, z.B.
```
class GenDerived<T> : BaseClass {
 . . .
}
```
- eine konkretisierte generische Klasse
```
class GenDerived<T> : GenBase<int, double> {
 . . .
}
```
- eine generische Klasse mit Typformalparametern, wobei Typrestriktionen der Basisklasse ggf. zu wiederholen sind, z.B.:
```
class GenBase<T1, T2> where T2 : IComparable<T2> {
 . . .
}
class GenDerived<T> : GenBase<int, T> where T : IComparable<T> {
 . . .
}
```
Im Beispiel hat die konkretisierte Version `GenDerived<int>` die Basisklasse `GenBase<int, int>`.

Offenbar ist die nachträgliche Integration der generischen Typen in das *Common Type System* (CTS) der .NET - Plattform gut gelungen.

Im Zusammenhang mit Vererbung und Generizität bleibt die wichtige Frage zu klären, ob sich bei zwei Konkretisierungen einer generischen Klasse mit dem Typparameter **T**, z.B. **List<T>**, eine Spezialisierungsbeziehung zwischen den beiden Typktualparametern, z.B.:



auf die Konkretisierungen überträgt, z.B.:



Obwohl eine positive Beantwortung der aufgeworfenen Frage auf den allerersten Blick plausibel erscheint, hätte sie doch üble Folgen. Man könnte nämlich z.B. ...

- die Adresse eines Objekts vom Typ **List<String>** einer Referenzvariablen vom Typ **List<Object>** zuweisen
- und mit Hilfe dieser Referenzvariablen über die dann verfügbare Methode
  public void Add(Object *element***)**
  eine Instanz beliebigen Typs in die Liste aufnehmen.

Damit wäre das essentielle Prinzip der Sortenreinheit verletzt, und bei nächster Gelegenheit würde der Versuch, dem eingeschmuggelten Objekt eine **String**-Kompetenz abzuverlangen (z.B. Eigenschaft **Length**) zu einem Laufzeitfehler führen.

Um das beschriebene Desaster zu vermeiden, wird in C# eine Spezialisierungsbeziehung zwischen zwei Typaktualparametern **NICHT** auf die zugehörigen Konkretisierungen derselben generischen

Klasse übertragen, und der folgende listige Versuch, das Bemühen des C# - Compilers um Typsicherheit auszutricksen, misslingt:

Man sagt, dass in C# die Typformalparameter bei generischen Klassen **invariant** sind.

Im umgekehrten Fall spricht man von **kovarianten** Typformalparametern,, und dieser Fall ist in C# durchaus anzutreffen. Es ist sogar ein beim Einsatz von Arrays permanent drohender Programmierfehler, den der Compiler aus Kompatibilitätsgründen *nicht* verhindern kann. Die oben als tückisch entlarvte und bei generischen Klassen verbotene Kovarianz ist bei Arrays in C# (und auch in anderen Programmiersprachen wie Java) leider erlaubt. Der folgende Quellcode wird ohne Kritik übersetzt

```
object[] oarr = new String[] { "a", "b" };
oarr[0] = 13;
foreach (String s in oarr)
  Console.WriteLine(s.Length);
```

und verursacht zur Laufzeit einen Ausnahmefehler:

```
Unbehandelte Ausnahme: System.ArrayTypeMismatchException: Es wurde versucht,
auf ein Element zuzugreifen, dessen Typ mit dem Array nicht kompatibel ist.
```

Obwohl bisher wenig Positives über die Kovarianz gesagt wurde, kann sie unter bestimmten Voraussetzungen sinnvoll eingesetzt werden und den Nutzen des generischen Programmierens steigern. Bei generischen *Schnittstellen* und Delegaten ist es möglich und sinnvoll, einen Typformalparameter explizit als kovariant zu deklarieren (siehe Abschnitte 8.2.1 bzw. 9.1.6). Einen Container vom Typ **List<String>** über eine Referenz vom Typ **List<Object>** anzusprechen, kann nämlich durchaus sinnvoll sein, solange der Container dabei *nicht verändert* wird.

7.3 Nullable<T> als Beispiel für generische Strukturen

In diesem Abschnitt wird die generische Struktur **Nullable<T>** aus der FCL vorgestellt:[1]

```
public struct Nullable<T> where T : struct {
  private bool hasValue;
  internal T value;

  public Nullable(T value) {
    this.value = value;
    this.hasValue = true;
  }

  public bool HasValue {
    get {
      return hasValue;
    }
  }
```

[1] Der FCL-Q**uellcode** kann über die Webseite
https://referencesource.microsoft.com/download.html.
herunter geladen oder auf der Webseite
https://referencesource.microsoft.com/
inspiziert werden.

```
    public T Value {
       get { ... }
    }
    . . .
}
```

Durch die Instanzen einer Konkretisierung lassen sich Werte einer Struktur (z.B. Werte eines elementaren Datentyps) so verpacken, dass neben den normalen Werten auch der Ausnahmewert **null** zur Verfügung steht. Er eignet sich etwa dazu, um bei einer Variablen einen undefinierten Zustand zu signalisieren. Verpackt man z.B. eine Variablen vom Typ **bool**, erhält man neben den Werten **true** und **false** noch den dritten Wert **null**, der als *unbekannt* interpretiert werden kann.

Im folgenden Beispiel wird eine Variable vom Typ **Nullable<bool>** deklariert:

```
Nullable<bool> status;
```

Man kann einer **Nullable**-Instanz jeden Wert des Grundtyps und außerdem den Wert **null** zuweisen, z.B.:

```
status = null;
```

Die **null**-fähige Variante zu einem Strukturtyp lässt sich auch durch den Namen des Grundtyps und ein angehängtes Fragezeichen ansprechen, z.B.:

```
bool? status;
```

Eine **Nullable**-Instanz informiert in der booleschen Read-Only - Eigenschaft **HasValue** darüber, ob ein definierter Wert vorhanden ist, und hält diesen Wert ggf. in der Eigenschaft **Value** für den ausschließlich lesenden Zugriff bereit, z.B.:

```
var alter = new int?[2];
alter[0] = 30;
alter[1] = null;
foreach (int? ni in alter)
   if (ni.HasValue)
      Console.WriteLine(ni.Value);
   else
      Console.WriteLine("unbekannt");
```

Ist kein definierter Wert vorhanden, führt ein Leseversuch zu einem Ausnahmefehler vom Typ **InvalidOperationException**.

Während der Compiler den Grundtyp bei Bedarf implizit in den zugehörigen **Nullable**-Typ konvertiert, ist für den umgekehrten Übergang eine *explizite* Konvertierung unter der Verantwortung des Programmierers erforderlich, z.B.:

```
double? dn = 77.7;
double d = (double) dn;
```

Hat das **Nullable**-Argument des Typumwandlungsoperators den Wert **null**, kommt es zu einem Ausnahmefehler vom Typ **InvalidOperationException**.

Die beim Grundtyp unterstützten **Operatoren** sind auch bei der **null**-fähigen Verschachtelung erlaubt, z.B.:

```
double? d1 = 1.0, d2 = 2.0;
double? s = d1 + d2;
```

Hat ein beteiligter Operand den Wert **null**, so erhält auch der Ausdruck diesen Wert, z.B.:

```
double? d1 = 1.0, d2 = null;
double? s = d1 + d2;
Console.WriteLine(s.HasValue); // liefert false
```

Man kann einer gewöhnlichen (nicht **null**-fähigen) Strukturinstanz den Wert **null** nicht zuweisen. Ein Vergleich mit diesem Wert ist hingegen erlaubt, wobei das Ergebnis stets **false** ist, z.B. beim Vergleich:

```
0 == null
```

Ein (statisches) Feld mit **Nullable**-Typ wird mit **null** initialisiert, z.B.:

Quellcode	Ausgabe
`class Prog {` `  int i;` `  int? ni;` `  static void Main() {` `  var p = new Prog();` `  System.Console.WriteLine(p.i+"\n"+p.ni.HasValue);` `  }` `}`	`0` `False`

Von dieser Regel sind auch die Elemente eines Arrays betroffen.

Mit dem so genannten **Null-Sammeloperator**, der durch zwei Fragezeichen ausgedrückt wird, lässt sich die Zuweisung einer **null**-fähigen Strukturinstanz an eine Variable des Grundtyps samt Ersatzwert für die kritische **null**-Situation bequem formulieren, z.B.:[1]

Quellcodesegment	Ausgabe
`int? ni = 4;` `int i = ni ?? -1;` `Console.WriteLine(i);`	`4`

Ist der linke ?? - Operand von **null** verschieden, liefert er den Wert des Ausdrucks. Anderenfalls kommt der rechte Operand zum Zug, der vom Grundtyp und initialisiert sein muss.

Das Bemühen von Compiler und CLR, eine **Nullable**-Instanz wie eine Instanz des zugehörigen Grundtyps zu behandeln, geht so weit, dass bei einer **GetType()** - Anfrage der Grundtyp genannt wird, z.B.:

Quellcodesegment	Ausgabe
`double? d = 3.0;` `Console.WriteLine(d.GetType());`	`System.Double`

7.4 Generische Methoden

Im Vergleich zu mehreren überladenen Methoden (vgl. Abschnitt 4.3.5), die analoge Operationen mit verschiedenen Datentypen ausführen, ist *eine* generische Methode oft die bessere Lösung. Im folgenden Beispiel liefert eine statische und generische Methode das Maximum von zwei Argumenten, wobei der gemeinsame Datentyp der Argumente die Schnittstelle **Comparable<T>** (vgl. Abschnitt 7.2.2) erfüllen, also eine Methode mit dem Definitionskopf

public int CompareTo(T *element***)**

besitzen muss:

---

[1] Die Bezeichnung *Null-Sammeloperator* wurde von der folgenden MSDN-Webseite übernommen:
https://msdn.microsoft.com/de-de/library/ms173224.aspx
Eine alternative Bezeichnung lautet: *Null-Koaleszenz - Operator*.

Quellcode	Ausgabe

```
using System;
class Prog {
  static T Max<T>(T x, T y) where T : IComparable<T> {
    return x.CompareTo(y) >= 0 ? x : y;
  }

  public static void Main() {
    Console.WriteLine("int-max:\t" + Max(12, 13));
    Console.WriteLine("double-max:\t" + Max(2.16, 47.11));
    Console.WriteLine("String-max:\t" + Max("abc", "def"));
  }
}
```

```
int-max:        13
double-max:     47,11
String-max:     def
```

In der Definition einer generischen Methode befindet sich hinter dem Namen zwischen spitzen Klammern mindestens ein Typformalparameter. Mehrere Typparameter werden durch Kommata getrennt. Sie sind als Datentypen für den Rückgabewert, für Parameter und für lokale Variablen erlaubt. Wie bei generischen Klassen kann man über das Schlüsselwort **where** Restriktionen für Typparameter formulieren.

Verwendet eine Methode einer *generischen* Klasse einen Typparameter der Klasse als Formalparameter- oder Rückgabetyp, spricht man *nicht* von einer generischen Methode, weil keine eigenen Typparameter definiert werden, z.B. bei der Methode `Push()` der in Abschnitt 7.2.1 beschriebenen Klasse `SimpleStack<T>`:

```
public bool Push(T element) {
  . . .
}
```

Beim Aufruf einer generischen Methode kann der Compiler fast immer aus den Datentypen der Aktualparameter die passende Konkretisierung ermitteln (**Typinferenz**). Daher konnte im obigen Beispiel an Stelle der kompletten Syntax

```
Console.WriteLine("int-max:\t" + Max<int>(12, 13));
Console.WriteLine("double-max:\t" + Max<double>(2.16, 47.11));
Console.WriteLine("String-max:\t" + Max<string>("abc", "def"));
```

die folgende Kurzschreibweise verwendet werden:

```
Console.WriteLine("int-max:\t" + Max(12, 13));
Console.WriteLine("double-max:\t" + Max(2.16, 47.11));
Console.WriteLine("String-max:\t" + Max("abc", "def"));
```

Das Regelwerk zu der insgesamt recht komplexen Typinferenz ist in der C# - Sprachspezifikation im Abschnitt 7.5.2 beschrieben (Microsoft 2012).

7.5 default(T)

In einer Methode eines generischen Typs oder in einer generischen Methode ist es gelegentlich erforderlich, den „Null-artigen“ Standardwert zu einem Typformalparameter zu verwenden. Aufgrund der zahlreichen möglichen Konkretisierungen eines Typformalparameters ist die Ermittlung des jeweiligen Standardwerts keine triviale Aufgabe, die man zum Glück dem **default**-Operator überlassen kann.[1] Er liefert ...

[1] Das Schlüsselwort **default** wird (leider) in gänzlich anderer Bedeutung auch in der **switch**-Anweisung verwendet (siehe Abschnitt 3.7.2.3).

- den Wert **null**, wenn beim Konkretisieren des generischen Typs **T** ein Referenztyp oder ein **Nullable**-Strukturtyp angegeben wurde,
- die passende numerische Null, wenn für **T** ein numerischer Werttyp angegeben wurde,
- eine Strukturinstanz mit „Null-artiger“ Feldinitialisierung, wenn für **T** ein Strukturtyp angegeben wurde:
  - Numerische Felder erhalten den Wert 0.
  - Felder mit einem Referenztyp oder mit einem **Nullable**-Strukturtyp erhalten den Wert **null**.

Im folgenden Programm

```
using System;

struct MitNullInt {
   int i;
   int? inu;
   public MitNullInt(int i, int? inu) {
      this.i = i;  this.inu = inu;
   }
   override public String ToString() {
      return "(" + i + ", " + (inu == null ? "null" : inu.ToString()) + ")";
   }
}

class DefaultDemo {
   static void WriteDef<T>() {
      Console.Write("default of " + typeof(T) + ": ");
      if (default(T) == null)
         Console.WriteLine("null");
      else
         Console.WriteLine(default(T));
   }
   static void Main() {
      WriteDef<int>();
      WriteDef<System.Numerics.Complex>();
      WriteDef<String>();
      WriteDef<int?>();
      WriteDef<MitNullInt>();
   }
}
```

gilt eine generische Methode den **default**-Wert aus für:

- den numerischen Werttyp **int**,
- den Strukturtyp **Complex** aus dem FCL-Namensraum: **System.Numerics** für komplexe Zahlen im Sinne der mathematischen Analysis, die aus einem Real- und einem Imaginärteil bestehen,[1]
- den Referenztyp **String**,
- den konkretisierten generischen Strukturtyp **Nullable<int>** (alias: **int?**),
- den selbst definierten Strukturtyp `MitNullInt`.

Es resultiert die Ausgabe:

[1] Mathematisch *nicht* vorbelastete Leser können sich unter einer komplexen Zahl ein Paar aus zwei reellen Zahlen vorstellen.

```
default of System.Int32: 0
default of System.Numerics.Complex: (0, 0)
default of System.String: null
default of System.Nullable`1[System.Int32]: null
default of MitNullInt: (0, null)
```

7.6 Wichtige Typen im Namensraum System.Collections.Generic

Ein besonders erfolgreiches Anwendungsfeld für die Typgenerizität sind die Klassen zur Verwaltung von Listen, Mengen oder (Schlüssel-Wert) - Tabellen im Namensraum **System.Collections.Generic**. In diesem Abschnitt werden wichtige Vertreter vorgestellt.[1]

Es ist allgemeiner Konsens, dass in einer objektorientierten Software die Verwaltung und Transformation der Daten nach den Regeln der Geschäftslogik getrennt werden sollte von der Präsentation der Daten und der Benutzerinteraktion. Zur Bewältigung der ersten Teilaufgabe tragen die im aktuellen Abschnitt vorzustellenden Kollektionsklassen entscheidend bei.

Auch das in Abschnitt 5.7 vorgestellte Beispielprogramm zur Präsentation von RSS-Items verwendet zur Verwaltung seiner Daten ein Kollektionsobjekt.

7.6.1 Arrays versus Kollektionen

Die in Abschnitt 5.3 vorgestellten Arrays taugen als Container für Elemente mit einem identischen, frei wählbaren Typ. Sie bieten Speichereffizienz und einen schnellen Indexzugriff auf die Elemente. Man kann sich fragen, wozu eigentlich noch weitere Kollektionsklassen benötigt werden.

Beginnen wir bei den sehr häufig auftretenden listenartigen Datenstrukturen. Dabei zeigen Arrays folgende Schwächen:

- Die Größe eines Arrays muss beim Erstellen festgelegt werden und kann nicht mehr geändert werden.
- Das Einfügen und Entfernen von inneren Elementen ist mit einem hohen Aufwand verbunden.

Kollektionen zur Verwaltung von Listen bieten hingegen Größendynamik sowie performantes Einfügen und Löschen von inneren Elementen. Zwar verwenden viele Listentypen im Hintergrund einen Array zur Verwaltung ihrer Elemente und müssen den Array bei einer Kapazitätsüberschreitung ersetzen, doch geschieht dies automatisch ohne unser Zutun.

Sind für Elementsammlungen häufige Existenzprüfungen erforderlich, bietet ein Array wenig Unterstützung. Sind seine Elemente nicht sortiert, muss für jedes Element geprüft werden, ob es mit dem gesuchten übereinstimmt. Kollektionen zur Verwaltung von Mengen (siehe Abschnitt 7.6.3) bieten hingegen schnelle Detektionsmöglichkeiten und verhindern identische Elemente (Dubletten).

Oft sind Mengen von (Schlüssel-Wert) - Paaren zu verwalten, z.B. eine Tabelle mit den bei einem Web-Dienst aktuell angemeldeten Benutzern, wobei eine eindeutige Kennung als Schlüssel fungiert und auf ein Objekt mit den Eigenschaften des Benutzers zeigt. Eventuell stammen die Eigenschaften aus einer Datenbankzeile, die nach der Anmeldung des Benutzers von einem Datenbankserver bezogen und dann zum schnellen Zugriff im Hauptspeicher aufbewahrt wird. Es melden sich ständig Benutzer an oder ab, und beim Versuch, eine solche Datenstruktur mit einem Array zu verwalten, treten die eben schon beschriebenen Probleme auf (feste Anzahl von Elementen, umständliches Einfügen und Löschen, aufwändige Suche nach den Schlüsseln).

---

[1] Verwendet eine Anwendung mehrere nebenläufige *Ausführungsfäden* (Multithreading, siehe Kapitel 15), wird eventuell eine Thread-sichere Kollektion aus dem mit .NET 4.0 eingeführten Namensraum **System.Collections.Concurrent** benötigt.

Als weiterer Nachteil von Arrays ist ihr kovariantes Verhalten hinsichtlich des Elementtyps zu nennen (siehe Abschnitt 7.2.3).

Im zu modellierenden Aufgabenbereich treten oft Datenstrukturen vom Typ Liste, Menge oder (Schlüssel-Wert) - Tabelle auf, und im FCL-Namensraum **System.Collections.Generic** finden sich oft passende Typen, so dass im Vergleich zur Verwendung von Arrays eine bessere Modellierung und ein besser lesbarer Quellcode resultieren. Sicherlich sind Sie mittlerweile in der Lage, eine generische Kollektionsklasse zu definieren, die ihre Elemente in einem Array lagert und diesen bei Bedarf automatisch durch ein größeres Exemplar ersetzt. Allerdings sind in der FCL bereits exzellente Lösungen für derartige Standardaufgaben vorhanden, so dass wir uns auf andere Herausforderungen konzentrieren können.

7.6.2 Verwaltung einer Liste

Neben den anschließend vorgestellten Typen zur Listenverwaltung enthält die FCL noch Lösungen für wichtige Spezialfälle, die im Manuskript nicht behandelt werden können, z.B.:

- Stapel
  Die Klasse **Stack<T>** verwaltet eine Liste nach dem LIFO-Prinzip (*Last In First Out*). In Abschnitt 7.2.1 haben wir eine simple Variante selbst erstellt.
- Warteschlangen
  Die Klasse **Queue<T>** verwaltet eine Liste nach dem FIFO-Prinzip (*First In First Out*).

7.6.2.1 Listen mit Array-Unterbau

Die generische Klasse **List<T>** zur bequemen und typsicheren Verwaltung einer Sequenz von Elementen eines festen Typs ist schon aus Abschnitt 7.1 bekannt. Ein **List<T>** - Objekt speichert seine Elemente in einem internen Array **T[]**, so dass ...

- Lesezugriffe sehr performant sind,
- Veränderungen hingegen zeitaufwendig sind, wenn ...
  - neue Objekte im Inneren der Liste eingefügt oder entfernt werden müssen
  - oder aufgrund einer Kapazitätsüberschreitung ein neuer Array angelegt werden muss.

Eine automatische Kapazitätserweiterung orientiert sich an der bisherigen Größe, damit diese kostspielige Maßnahme möglichst selten erforderlich wird. In einer Konstruktorüberladung kann man die initiale Kapazität festlegen, z.B.:

```
var nl = new List<String>(500);
```

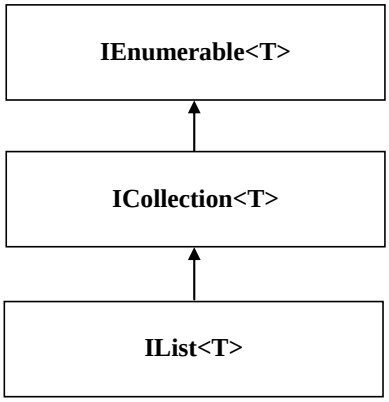
Über die Eigenschaft **Capacity** erfährt man die aktuelle Kapazität einer Liste, welche meist die per **Count**-Eigenschaft abfragbare Länge übertrifft, z.B.:

Quellcode	Ausgabe
`using System;` `using System.Collections.Generic;` `class Prog {` `  static void Main() {` `  var nl = new List<String>(5);` `  nl.Add("Otto");` `  nl.Add("Marita");` `  nl.Add("Theo");` `  Console.WriteLine($"Länge:\t\t{nl.Count}\nKapazität:\t{nl.Capacity}");` `  }` `}`	`Länge:  3` `Kapazität:  5`

Um eine Liste bereits bei der Erstellung auf syntaktisch einfache Weise zu bevölkern, kann ein **Kollektionsinitialisierer** verwendet werden, z.B.:

```
var nl = new List<String> {"Otto", "Marita", "Theo"};
```

Die Klasse **List<T>** implementiert die Schnittstelle **IList<T>** und damit indirekt auch die Schnittstelle **ICollection<T>**, von der **IList<T>** abstammt sowie die Schnittstelle **IEnumerable<T>**, von der **ICollection<T>** abstammt (alle Typen aus dem Namensraum **System.Collections.Generics**):



Eine Schnittstelle kann vorläufig als Klasse mit ausschließlich abstrakten Methoden aufgefasst werden. Implementiert eine Klasse eine Schnittstelle, muss sie deren Methoden realisieren. Weitere Details zu Schnittstellen (engl.: *Interfaces*) folgen in Kapitel 8.

Schon die Schnittstelle **IEnumerable<T>** verpflichtet eine implementierende Klasse, als Rückgabe der Methode **GetEnumerator()** ein Objekt vom Typ **IEnumerator<T>** zu liefern, das ein Iterieren durch die Kollektionselemente erlaubt und in der Regel ausschließlich im Rahmen einer **foreach**-Schleife verwendet wird.

Ein Objekt der Klasse **List<T>** beherrscht u.a. die folgenden Methoden:

- **public void Add(T** *element***)**
  Das Parameterelement wird in die Liste aufgenommen.
- **public void Insert(int** *pos***, T** *element***)**
  Das Parameterelement wird an der gewünschten Indexposition eingefügt.
- **public boole Contains(T** *element***)**
  Diese Methode informiert darüber, ob das Parameterelement in der Liste vorhanden ist.
- **public boole Remove(T** *element***)**
  Das erste Vorkommen des Elements wird aus der Liste entfernt, falls es dort vorhanden ist. Über den Rückgabewert erfährt man, ob die Liste durch den Aufruf verändert worden ist.
- **public void RemoveAt(int** *pos***)**
  Das Element an der angegebenen Indexposition wird entfernt.
- **public void Clear()**
  Alle Elemente werden entfernt.
- **public int IndexOf(T** *element***)**
  Falls das Element in der Liste vorhanden ist, wird der Index des ersten Auftretens geliefert, anderenfalls der Wert -1.
- **public ReadOnlyCollection<T> AsReadOnly()**
  Der Aufrufer erhält eine *Sicht* auf die angesprochene Liste, ...
  - die keine Veränderung der Liste zulässt,
  - aber jede Änderung der zugrunde liegenden Liste sofort berücksichtigt.

Außerdem beherrscht die Klasse **List<T>** zum Suchen und Sortieren grundsätzlich dieselben Methoden wie ein Array (vgl. Abschnitt 5.3.6), wobei die statischen Methoden der Klasse **Array** durch Instanzmethoden der Klasse **List<T>** ersetzt werden.

Auch die folgenden Eigenschaften stehen im Pflichtenheft einer Klasse, die das Interface **IList<T>** implementiert:

- **public int Count { get; }**
  Es wird die Anzahl der Elemente geliefert.
- **public bool IsReadOnly { get; }**
  Man erfährt, ob die Liste schreibgeschützt ist.

Schließlich bietet die Klasse **List<T>** einen Indexer (vgl. Abschnitt 5.6), den die Schnittstelle **IList<T>** ebenfalls vorschreibt, so dass wie bei einem Array ein Elementzugriff über den **[]** - Operator möglich ist.

Im Zusammenhang mit dem Indexer für **List<T>** - Objekte soll noch einmal demonstriert werden, warum viele Autoren dringend von veränderlichen Strukturen abraten (vgl. Abschnitt 5.1.1). Sind die Listenelemente vom Typ einer Klasse, treten beim Indexzugriff keine Überraschungen auf. Im folgenden Beispielprogramm wird ein per Indexer angesprochenes Objekt der Klasse `Punkt` gebeten, sich zu `Bewegen()`. Anschließend wird per **WriteLine()** - Aufruf verifiziert, dass sich das Objekt an der neuen Position befindet:

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;
public class Punkt {
   double x, y;
   public Punkt(double xpar, double ypar) {
      x = xpar;
      y = ypar;
   }
   public void Bewegen(double hor, double vert) {
      x = x + hor;
      y = y + vert;
   }
   override public string ToString() {
      return "(" + x + "; " + y + ")";
   }
}
class Prog {
   static void Main() {
      var pl = new List<Punkt> {new Punkt(1, 2), new Punkt(3, 4)};
      pl[0].Bewegen(5, 5);
      Console.WriteLine(pl[0]);
   }
}
```

```
(6; 7)
```

Wird im Beispiel jedoch eine *Struktur* namens `Punkt` anstatt einer Klasse definiert, dann führt die per Indexer angesprochene Instanz die Methode `Bewegen()` zwar aus, doch produziert der anschließende **WriteLine()** - Methodenaufruf die (vermutlich von vielen Programmierern *nicht* erwartete) Ausgabe

```
(1; 2)
```

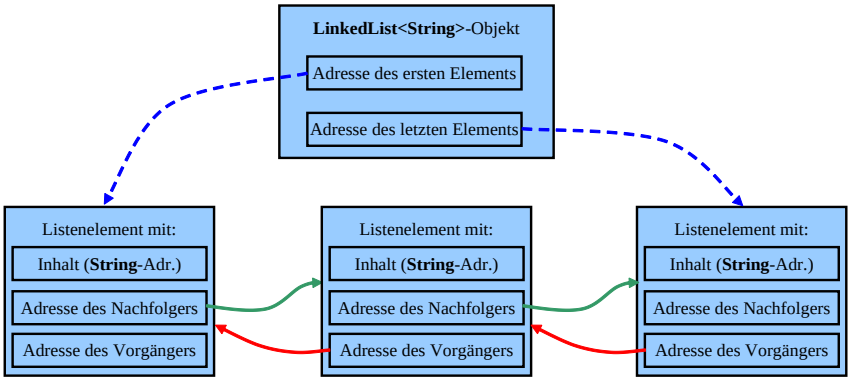
Hinter dem Indexer steckt u.a. eine verkappte **get**-Methode, und die liefert eine *Kopie* der Strukturinstanz mit dem angefragten Indexwert.

Das beschriebene Problem tritt übrigens *nicht* auf, wenn statt eines **List<Punkt>** - Objekts der Array `Punkt[]` verwendet wird, weil dann z.B. der Ausdruck `pl[0]` das Element mit dem Indexwert 0 *identifiziert*, anstatt eine Kopie dieses Elements zu liefern (Griffith 2013, S. 167).

7.6.2.2 Listen mit verketteten Elementen

Die Klasse **List<T>** arbeitet intern mit einem Array zum Speichern der Elemente und bietet daher einen schnellen wahlfreien Zugriff. Auch das Anhängen neuer Elemente am Ende der Liste verläuft flott, wenn nicht gerade die Kapazität des Arrays erschöpft ist. Dann wird es erforderlich, einen größeren Array zu erzeugen und alle Elemente dorthin zu kopieren. Beim Einfügen bzw. Löschen von *inneren* Elementen müssen die neuen bzw. früheren rechten Nachbarn zeitaufwändig nach rechts bzw. links verschoben werden.

Die Klasse **LinkedList<T>** arbeitet intern mit einer **doppelt verketteten Liste** bestehend aus selbständigen Objekten, die jeweils ihren Nachfolger und ihren Vorgänger kennen, z.B.:



In Abschnitt 5.6 haben wir übrigens eine (allerdings nur einfach) verkettete Liste selbst gebaut, um die Definition eines Indexers demonstrieren zu können.

Vorteile einer verlinkten Liste im Vergleich zu einem Array:

- Die Länge der Liste ist zu keinem Zeitpunkt festgelegt, so dass keine aufwendigen Maßnahmen zur Kapazitätsanpassung erforderlich werden.
- Beim Einfügen und Löschen von inneren Elementen müssen keine anderen Elemente verschoben werden. Stattdessen wird ein Listenelement erzeugt bzw. gelöscht, und die Adressketten werden neu verknüpft.

Um das Listenelement an einer bestimmten Position aufzusuchen, muss die Liste allerdings ausgehend vom ersten oder letzten Element durchlaufen werden. Folglich ist die verkettete Liste beim wahlfreien Zugriff auf vorhandene Elemente einem Array drastisch unterlegen, weil dessen Elemente im Speicher hintereinander liegen und nach einer einfachen Adressberechnung direkt angesprochen werden können. Außerdem benötigt eine verkettete Liste mehr Speicher, weil sich jedes Element als selbständiges Objekt auf dem Heap befindet.

Insgesamt sind verkettete Listen besonders geeignet für Algorithmen, die …

- häufig Elemente einfügen oder entfernen und sich dabei nicht auf das Listenende beschränken,
- Elemente überwiegend sequentiell aufsuchen.

Ein mögliches Anwendungsbeispiel ist die Zugplaner-Software für einen Rangierbahnhof, wo Züge neu zusammengestellt werden. Aus einem Zug werden Waggons entnommen und andere Waggons so eingefügt, dass sie auf der Strecke sukzessive ausgeliefert (abgehängt) werden können. Das Auskoppeln und Einhängen von Waggons klappt mit der Zugplaner-Software blendend, weil ein Zug jeweils durch eine verkettete Liste modelliert wird. Bei den realen Zügen sind die Entnahme und das Einfügen von inneren Elementen allerdings genauso umständlich wie bei Arrays.

Die Klasse **LinkedList<T>** implementiert zwar die Schnittstelle **ICollection<T>**, aber *nicht* die Schnittstelle **IList<T>**, so dass insbesondere kein Indexer verfügbar ist. Vorhanden sind u.a. die folgenden Eigenschaften und Methoden:

- **First, Last**
  Diese Eigenschaften zeigen auf den ersten bzw. auf den letzten Knoten, bei einer leeren Liste auf **null**.
- **public LinkedListNode<T> AddFirst(T** *element***)**
  Diese Methode fügt einen neuen Knoten am Anfang ein und liefert ein zugehöriges Objekt der Klasse **LinkedListNode<T>** (siehe unten) zurück.
- **public LinkedListNode<T> AddLast(T** *element***)**
  Es wird ein neuer Knoten am Ende angehängt und ein zugehöriges Objekt der Klasse **LinkedListNode<T>** zurückgeliefert.
- **public LinkedListNode<T> AddBefore(LinkedListNode<T>** *node***, T** *element***)**
  Es wird ein neuer Knoten vor dem angegebenen Exemplar eingefügt und ein zugehöriges Objekt der Klasse **LinkedListNode<T>** zurückgeliefert.
- **public LinkedListNode<T> AddAfter(LinkedListNode<T>** *node***, T** *element***)**
  Es wird ein neuer Knoten hinter dem angegebenen Exemplar eingefügt und ein zugehöriges Objekt der Klasse **LinkedListNode<T>** zurückgeliefert.
- **public void RemoveFirst()**
  Der erste Knoten wird entfernt.
- **public void RemoveLast()**
  Der letzte Knoten wird entfernt.
- **public void Remove(LinkedListNode<T>** *element***)**
  Der per Aktualparameter angegebene Knoten wird entfernt.
- **public bool Remove(T** *element***)**
  Der erste mit der Parameterinstanz inhaltsidentische Knoten wird entfernt. Über den Rückgabewert erfährt man, ob die Liste durch den Aufruf verändert worden ist.

Ein Objekt der Klasse **LinkedListNode<T>** ...

- macht den eigentlichen Inhalt über eine Eigenschaft namens **Value** vom Typ **T** zugänglich
- und kennt über die Read-Only - Eigenschaften **Previous** bzw. **Next** vom Typ **LinkedListNode<T>** den vorherigen bzw. nächsten Knoten.

7.6.3 Verwaltung einer Menge mit der Klasse HashSet<T>

Die generische Klasse **HashSet<T>** eignet sich zur Verwaltung einer Menge von Elementen, wobei im Unterschied zu einer Liste ...

- einerseits *keine* relevante Anordnung besteht,
- andererseits aber das Auftreten von Dubletten verhindert wird.

Ein Objekt dieser Klasse beherrscht u.a. die folgenden Instanzmethoden:

- **public boole Add(T** *element***)**
  Das Parameterelement wird in die Menge aufgenommen, falls es dort noch nicht vorhanden ist. Über den Rückgabewert erfährt man, ob die Menge durch den Aufruf verändert worden ist.
- **public boole Contains(T** *element***)**
  Diese Methode informiert darüber, ob das fragliche Element in der Menge vorhanden ist.
- **public boole Remove(T** *element***)**
  Das angegebene Element wird aus der Menge entfernt, falls es dort vorhanden ist. Über den Rückgabewert erfährt man, ob die Menge durch den Aufruf verändert worden ist.
- **public void Clear()**
  Es werden alle Elemente entfernt.

- **public void IntersectWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt streicht alle Elemente, die sich *nicht* in der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Schnittmenge:

  M (angesprochene Menge) ∩ **P** (Parametermenge)

- **public void UnionWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt nimmt aus der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) alle Elemente auf, die nicht zu Dubletten führen. Man erhält also die Vereinigungsmenge:

  M (angesprochene Menge) ∪ **P** (Parametermenge)

- **public void ExceptWith(IEnumerable<T>** *other***)**
  Das angesprochene **HashSet<T>** - Objekt streicht alle Elemente, die sich in der Parameterkollektion (ein Objekt einer Klasse, welche die Schnittstelle **IEnumerable<T>** erfüllt) befinden. Man erhält also die Differenzmenge:

  M (angesprochene Menge) - **P** (Parametermenge)

Das folgende Programm demonstriert die aufgelisteten Methoden:

Quellcode

```
using System;
using System.Collections.Generic;
class HashSetDemo {
  static void Main() {
    var m1 = new HashSet<char>();
    m1.Add('a'); m1.Add('b'); m1.Add('c');
    Console.WriteLine("Menge 1:");
    foreach (char c in m1)Console.Write(c + " ");

    m1.Add('b');
    Console.WriteLine("\n\nMenge 1 nach zweiter b-Aufnahme:");
    foreach (char c in m1) Console.Write(c + " ");

    var m2 = new HashSet<char>() { 'c', 'd', 'e' };
    Console.WriteLine("\n\nMenge 2:");
    foreach (char c in m2) Console.Write(c + " ");

    Console.WriteLine("\n\nb in der Menge 2? "+m2.Contains('b'));

    var schnitt = new HashSet<char>(m1);
    schnitt.IntersectWith(m2);
    Console.WriteLine("\nSchnittmenge:");
    foreach (char c in schnitt) Console.Write(c + " ");

    var vereinigung = new HashSet<char>(m1);
    vereinigung.UnionWith(m2);
    Console.WriteLine("\n\nVereinigungsmenge:");
    foreach (char c in vereinigung)   Console.Write(c + " ");

    var differenz = new HashSet<char>(m1);
    differenz.ExceptWith(m2);
    Console.WriteLine("\n\nDifferenzmenge:");
    foreach (char c in differenz) Console.Write(c + " ");

    m1.Remove('a');
    Console.WriteLine("\n\nMenge 1 ohne a:");
    foreach (char c in m1) Console.Write(c + " ");

    m1.Clear();
    Console.WriteLine("\n\nMenge 1 nach Clear:");
    foreach (char c in m1) Console.Write(c + " ");
  }
}
```

Ausgabe

```
Menge 1:
a b c

Menge 2:
c d e

b in der Menge 2? False

Schnittmenge:
c

Vereinigungsmenge:
a b c d e

Differenzmenge:
a b

Menge 1 ohne a:
b c

Menge 1 nach Clear:
```

In einem Testprogramm mit den Aufgaben

- eine Menge mit 20.000 **String**-Objekten füllen
- für 20.000 neue **String**-Objekte prüfen, ob sie bereits in der Menge vorhanden sind

zeigen die Klassen **List<String>**, **LinkedList<String>** und **HashSet<String>** folgende Leistungen:[1]

```
Kollektionsklasse:                    List`1
 Zeit zum Füllen:                      3,6 Millisek.
 Zeit für die Existenzprüfungen:       4568,5 Millisek.

Kollektionsklasse:                    LinkedList`1
 Zeit zum Füllen:                      5,5 Millisek.
 Zeit für die Existenzprüfungen:       4690,0 Millisek.

Kollektionsklasse:                    HashSet`1
 Zeit zum Füllen:                      4,5 Millisek.
 Zeit für die Existenzprüfungen:       3,6 Millisek.
```

Eine Erklärung für die drastische Überlegenheit die Klasse **HashSet<String>** bei den Mengenzugehörigkeitsprüfungen findet sich z.B. in Baltes-Götz & Götz (2016, Abschnitt 10.5.2).

7.6.4 Verwaltung von (Schlüssel-Wert) - Paaren mit der Klasse Dictionary<K, V>

Zur Verwaltung einer Menge von (Schlüssel-Wert) - Paaren ist in der FCL die generische Klasse **Dictionary<K, V>** vorhanden. Die Schlüssel (mit einer Konkretisierung des Typformalparameters **K** als Datentyp) werden wie eine Menge verwaltet, so dass Eindeutigkeit garantiert ist (ohne Dubletten), aber keine relevante Anordnung besteht. Über einen Schlüssel ist sein Wert ansprechbar (mit einer Konkretisierung des Typformalparameters **V** als Datentyp). Man könnte z.B. eine Personalverwaltungsdatenbank realisieren mit ...

- einer eindeutigen Personalnummer (Typ **Integer** als **K**-Konkretisierung)
- und einer geeigneten Klasse `Personal` (mit Instanzvariablen für den Namen, die Telefonnummer etc.) als **V**- Konkretisierung.

Ein Objekt der Klasse **Dictionary<K, V>** beherrscht u.a. die folgenden Instanzmethoden:

- **public void Add(K** *key*, **V** *value***)**
  Falls der Schlüssel noch nicht existiert, wird ein neues (Schlüssel-Wert) - Paar aufgenommen. Ansonsten wirft die Methode eine Ausnahme vom Typ **ArgumentException**.
- **public boole ContainsKey(K** *key***)**
  Diese Methode informiert darüber, ob ein Element mit dem fraglichen Schlüssel vorhanden ist.
- **public boole RemoveKey(K** *key***)**
  Das Element mit dem angegebenen Schlüssel wird aus der Kollektion entfernt, falls der Schlüssel vorhanden ist. Über den Rückgabewert erfährt man, ob die Kollektion durch den Aufruf geändert worden ist.

---

[1] Die Zeiten stammen von einem PC unter Windows 10 (64 Bit) mit Intel-CPU Core i3 (3,2 GHz). Ein Visual Studio - Projekt mit dem Testprogramm ist hier zu finden:

...\BspUeb\Typgenerizität und Kollektionen\ListSetContains

- **public void Clear()**
  Es werden alle Elemente entfernt.

Weil die Konkretisierungen der generischen Klasse **Dictionary<K, V>** über einen Indexer verfügen, kann man über einen in eckige Klammern eingeschlossenen Schlüssel den zugehörigen Wert ermitteln und ändern, z.B.:

```
var fred = new Dictionary<char, int>();
fred.Add('c', 1);
fred['c'] = 5;
Console.WriteLine(fred['c']);
```

Durchläuft man eine **Dictionary<K, V>** - Kollektion per **foreach** - Schleife, ist als Elementtyp die passend konkretisierte generische Struktur **KeyValuePair<K, V>** anzugeben, z.B.:

```
foreach (KeyValuePair<char, int> kvp in fred) {
  Console.WriteLine($"{kvp.Key} : {kvp.Value}");
```

Über die Eigenschaften **Key** bzw. **Value** erhält man den Schlüssel bzw. den Wert einer Instanz.

Das folgende Programm verwendet ein **Dictionary<char, int>** - Objekt dazu, für einen **String** die Häufigkeiten der enthaltenen Zeichen zu ermitteln:

Quellcode	Ausgabe
`using System;` `using System.Collections.Generic;` `class DictionaryDemo {` `  public static void CountLetters(String text) {` `  var fred = new Dictionary<char, int>();` `  foreach (char c in text)` `  if (fred.ContainsKey(c)) {` `  fred[c]++;` `  }` `  else` `  fred.Add(c, 1);` `  foreach (KeyValuePair<char, int> kvp in fred) {` `  Console.WriteLine($"{kvp.Key} : {kvp.Value}");` `  }` `  }` `  static void Main() {` `  CountLetters("Otto spielt Lotto.");` `  }` `}`	`O : 1` `t : 5` `o : 3` `  : 2` `s : 1` `p : 1` `i : 1` `e : 1` `l : 1` `L : 1` `. : 1`

Um eine nach den Schlüsseln *sortierte* Tabelle von (Schlüssel-Wert) - Paaren zu erhalten, muss man lediglich die Klasse **Dictionary<K, V>** durch die Alternative **SortedDictionary<K, V>** ersetzen.

7.7 Übungsaufgaben zu Kapitel 7

1) Erstellen Sie eine Variante der in Abschnitt 5.6 vorgestellten Personenverwaltung, indem Sie Sie die Klasse `PersonDB` (Eigenbau einer verketteten Liste mit Indexer) durch die generische Kollektionsklasse **List<T>** ersetzten.

2) Als dynamisch wachsender Container für Elemente mit einem festen *Werttyp* (z.B. **int**) ist die Klasse **ArrayList** nicht gut geeignet, weil der Elementtyp **Object** zeitaufwendige (Un)boxing-Operationen erfordert. Dieser Aufwand entfällt bei einer passenden Konkretisierung der generischen Klasse **List<T>**, welche dieselbe Größendynamik bietet. Vergleichen Sie mit einem Testprogramm den Zeitaufwand beim Einfügen von 1 Million **int**-Werten in einen **ArrayList**- bzw. **List<int>** - Container.

8 Interfaces

Zu vielen Klassen oder Strukturen führt die FCL-Dokumentation hinter dem Namen und einem Doppelpunkt *mehrere* Typen auf, z.B. bei der Klasse **String**:

```
public sealed class String : IComparable, ICloneable, IConvertible,
      IEnumerable, IComparable<string>, IEnumerable<char>, IEquatable<string>
```

Um sieben Basisklassen kann es sich nicht handeln, weil C# keine Mehrfachvererbung unterstützt. Außerdem ist der FCL-Dokumentation zu entnehmen, dass die Klasse **String** direkt von der Urahnklasse **Object** abstammt. Am Anfangsbuchstaben *I* sind in der FCL-Dokumentation zuverlässig die von einer Klasse oder Struktur implementierten *Schnittstellen* (englisch: *Interfaces*) zu erkennen. Hierbei handelt es sich um **Verpflichtungserklärungen** von Klassen oder Strukturen gegenüber dem Compiler. Ein Interface definiert eine Reihe von Handlungskompetenzen abstrakt (ohne Implementierung) über Definitionsköpfe von Methoden oder anderen ausführbaren Mitgliedern (Eigenschaften, Indexern). Wenn sich ein Typ zu einem Interface bekennt, muss er die dort geforderten Handlungskompetenzen implementieren. Als Gegenleistung werden seine Instanzen vom Compiler überall dort akzeptiert (z.B. als Aktualparameter für einen Methodenaufruf), wo die jeweiligen Schnittstellenkompetenzen erforderlich sind.

Die Liste der von einem Typ implementierten Interfaces liefert also wichtige Informationen über die Handlungskompetenzen seiner Instanzen. Über die Klasse **String** ist u.a. zu erfahren:

- **IComparable**, **IComparable<String>**
  Die Klasse implementiert das traditionelle Interface **IComparable** und die moderne Konkretisierung **IComparable<String>** der generischen Schnittstelle **IComparable<T>**, die beide zum Namensraum **System** gehören. Wie bei Klassen und Strukturen ermöglicht auch bei Schnittstellen die generische Definition mit Typformalparametern beliebig viele Konkretisierungen, so dass die Typsicherheit zur Übersetzungszeit gewährleistet ist und lästige Typumwandlungen eingespart werden.
  Weil **String** die Schnittstelle **IComparable** implementiert, muss eine Methode

  public int CompareTo(Object *obj***)**

  vorhanden sein.[1] Um den Vertrag **IComparable<String>** zu erfüllen, wird eine Methode mit dem folgenden Definitionskopf benötigt:

  public int CompareTo(String *str***)**

  Weil **String**-Objekte die Fähigkeit zum Vergleich mit Artgenossen besitzen, kann z.B. ein Array mit Elementen dieses Typs bequem über die (statische) Methode **Array.Sort()** sortiert werden:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  String[] star = {"eins", "zwei", "drei"};` `  Array.Sort(star);` `  foreach (String s in star)` `  Console.WriteLine(s);` `  }` `}`	`drei` `eins` `zwei`

[1] Aus Kompatibilitätsgründen hält die Klasse **String** am Bekenntnis zur nicht-generischen Schnittstelle **IComparable** fest, so dass der Compiler kuriose Methodenaufrufe wie im folgenden Beispiel nicht verhindern kann:

```
Console.WriteLine("Alpha".CompareTo(3));
```

Die **String**-Methode **CompareTo(Object** *obj***)** reagiert darauf mit einer **ArgumentException**.

- **IClonable**
  Die Klasse **String** implementiert auch das Interface **ICloneable** (aus dem Namensraum **System**) und besitzt folglich eine Methode, welche eine Kopie des angesprochenen Objekts erzeugt:

  public Object Clone()

  Weil die Rückgabe den deklarierten Typ **Object** besitzt, ist bei der Zuweisung an eine **String**-Variable eine explizite Typumwandlung erforderlich, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  String s1 = "eins";` `  String s2 = (String) s1.Clone();` `  Console.WriteLine(s2);` `  }` `}`	`eins`

  Zum Interface **ICloneable** gibt es *keine* generische Alternative.[1]

Schnittstellen definieren Verhaltenskompetenzen abstrakt durch Methoden, Eigenschaften und Indexer ohne Anweisungsteil. Außerdem sind Ereignisdeklarationen möglich.[2] Man kann Schnittstellen in erster Näherung als Klassen mit ausschließlich abstrakten Methoden beschreiben. Ein Interface ist also ein **Datentyp**. Es lassen sich zwar keine *Instanzen* von diesem Typ erzeugen, aber *Referenzvariablen* sind erlaubt und als Abstrakationsmittel sehr nützlich. Sie dürfen auf Instanzen beliebiger Typen zeigen, welche die Schnittstelle implementieren. Somit können Instanzen unabhängig von den Vererbungsbeziehungen ihrer Typen gemeinsam verwaltet werden (z.B. in einem Array). Dabei werden Methodenaufrufe **polymorph** ausgeführt (späte bzw. dynamische Bindung, siehe Abschnitt 6.7), weil Interface-Methoden grundsätzlich virtuell sind.

Implementiert eine Klasse oder Struktur ein Interface, dann …

- muss sie die im Interface enthaltenen Methoden, Eigenschaften, Indexer und Ereignisse implementieren,
- werden Instanzen von diesem Typ vom Compiler überall dort akzeptiert, wo der Interface-Datentyp vorgeschrieben ist.

Im Programmieralltag kommen wir auf unterschiedliche Weise mit Schnittstellen in Kontakt, z.B.:

- Implementierung von vorhandenen Schnittstellen in einer eigenen Typdefinition
  Implementiert eine Klasse oder Struktur eine Schnittstelle, werden ihre Instanzen vom Compiler überall dort akzeptiert (z.B. als Aktualparameter), wo der jeweilige Schnittstellentyp gefordert ist.

  Beispiel: Wenn unser Klasse `Bruch` das Interface **IComparable<`Bruch`>** implementiert, können wir die bequeme Methode **Array.Sort()** verwenden, um einen Array mit `Bruch`-Objekten zu sortieren.

---

[1] Ein Grund könnte darin bestehen, dass die **Clone()** - Methode und damit das **ICloneable**-Interface durch eine Implementierungsunklarheit stark an Nutzen eingebüßt haben. Es geht um den Unterschied zwischen der *tiefen* Kopie, die auch alle direkten und indirekten Member-Objekte dupliziert, und der *flachen* Kopie, die Member-Objekte des Originals weiterverwendet. Bei vorhandenen Klassen ist potentiell unklar, welche **Clone()** - Implementierung sie verwenden. Für eine eigene Klasse kann man eine tiefe Kopie nicht garantieren, sobald Member-Objekte vorhanden sind.

[2] Ereignisse werden wir in Abschnitt 9.2 behandeln.

- Schnittstellen als Datentypen in eigenen Methoden- oder Typdefinitionen
  In einer eigenen Methodendefinition ist es oft sinnvoll, Parameterdatentypen und/oder den Rückgabetyp über Schnittstellen festzulegen. In den Anweisungen der Methode werden Verhaltenskompetenzen der Parameterinstanzen genutzt, die durch Schnittstellenverpflichtungen garantiert sind. Damit wird die Typsicherheit ohne überflüssige Einengung erreicht. Diese Strategie ist besonders erfolgreich bei Methoden von generischen Typen. Ein Beispiel für eine Methode mit Interface-Parameter ist die folgende Überladung der Methode **Sort()** in der generischen FCL-Klasse **List<T>**:

  public void Sort(IComparer<T> *comparer***)**

  Sind bei der Definition eines generischen Typs für einen beschränkten Typformalparameter bestimmte Verhaltenskompetenzen zu fordern, gelingt das oft am besten per Schnittstellendatentyp (siehe Abschnitt 7.2.2).
- Schnittstellen definieren
  Natürlich kommen als Datentypen in Methoden- oder Typdefinitionen auch selbst definierte Schnittstellen in Frage.

8.1 Interfaces definieren

Wir behandelt zuerst das im Programmieralltag vergleichsweise seltene Definieren einer Schnittstelle, weil dabei Inhalt und Funktion gut zu erkennen sind. Allerdings verzichten wir auf ein eigenes Beispiel und betrachten stattdessen die angenehm einfach aufgebaute und außerordentlich wichtige Schnittstelle **IComparable<T>** aus der FCL:

```
public interface IComparable<in T> {
   // Interface does not need to be marked with the serializable attribute
   // Compares this object to another object, returning an integer that
   // indicates the relationship. An implementation of this method must return
   // a value less than zero if this is less than object, zero
   // if this is equal to object, or a value greater than zero
   // if this is greater than object.
   //
   int CompareTo(T other);
}
```

Wie der Kommentar im obigen .NET - Originalquellcode zeigt, sind bei einer Schnittstellen-Definition neben den syntaktischen Forderungen meist auch semantische Vorstellungen im Spiel. Der Compiler kann aber z.B. aufgrund der obigen **IComparable<T>** - Definition sinnlose **CompareTo()** - Implementierungen *nicht* verhindern.

Einige Regeln für Schnittstellendefinitionen:

- Zugriffsmodifikatoren
  Bei einem Top-Level - Interface sind die Zugriffsmodifikatoren **public** und **internal** erlaubt. Wird **public** *nicht* angegeben, ist die Schnittstelle nur innerhalb ihres Assemblies verwendbar (Schutzstufe **internal**). Für ein eingeschachteltes Interface sind dieselben Zugriffsmodifikatoren verfügbar wie für andere Member.
- Modifikator **abstract**
  Schnittstellen sind grundsätzlich **abstract**. Der Modifikator **abstract** ist überflüssig und verboten.
- Schlüsselwort **interface**
  Das obligatorische Schlüsselwort dient zur Unterscheidung von Klassen- oder Strukturdefinitionen.

- Schnittstellenname
  Per Konvention beginnt der Interfacename mit einem großen *I*. Bei generischen Schnittstellen folgen die Typformalparameter dem Namen zwischen spitzen Klammern und durch Kommata getrennt. Wie bei generischen Klassen können auch bei generischen Schnittstellen Restriktionen für die Typparameter formuliert werden (vgl. Abschnitt 7.2.2).
  Mit der seit .NET 4.0 erlaubten Kennzeichnung eines Typformalparameters als kovariant (Schlüsselwort **out**) bzw. kontravariant (Schlüsselwort **in**) beschäftigen wir uns in Abschnitt 8.2.
- Erlaubte Interface-Member
  Als Interface-Member sind nur instanzbezogene Methoden, Eigenschaften, Indexer und Ereignisse erlaubt. Verboten sind Konstruktoren, Felder sowie statische Member.
- Definition von Methoden, Eigenschaften, Indexern und Ereignissen
  In einer Schnittstelle sind alle Methoden etc. grundsätzlich **public** und **abstract** (folglich auch **virtual**). Die drei Schlüsselwörter sind überflüssig und verboten. Bei der Implementierung einer Schnittstellenmethode etc. (siehe Abschnitt 8.3) muss (und darf) der Modifikator **override** *nicht* angegeben werden, weil keine Alternative zum Überschreiben besteht.

Ein Interface kann andere Interfaces **beerben** (bzw. erweitern), wobei dieselbe Syntax wie beim Ableiten von Klassen zu verwenden ist. In der FCL wird z.B. das generische Interface **IEnumerable<T>** durch das Interface **ICollection<T>** (beide im Namensraum **System.Collections.Generic**) erweitert:

```
public interface ICollection<T> : IEnumerable<T> {
   int Count { get; }
   bool IsReadOnly { get; }
   void Add(T item);
   void Clear();
   bool Contains(T item);
   void CopyTo(T[] array, int arrayIndex);
   bool Remove(T item);
}
```

Oft werden generische Schnittstellen als Erweiterung einer älteren, nicht-generischen Variante definiert, z.B. bei der Schnittstelle **IEnumerable<T>**:

```
public interface IEnumerable<out T> : IEnumerable {
   new IEnumerator<T> GetEnumerator();
}
```

Im konkreten Fall enthalten die abgeleitete Schnittstelle und die Basisschnittstelle

```
public interface IEnumerable {
   IEnumerator GetEnumerator();
}
```

eine Methode namens **GetEnumerator()** mit leerer Parameterliste. Weil die (unterschiedlichen) Rückgabewerte für die Methodensignatur unerheblich sind, liegen Methoden mit identischer Signatur vor, und die Methode der abgeleiteten Schnittstelle überdeckt die Variante der Basisschnittstelle. Um eine Warnung des Compilers zu vermeiden, wird in der abgeleiteten Schnittstelle der Modifikator **new** vor die Methode **GetEnumerator()** gesetzt (vgl. Abschnitt 6.5.1).
Bei der Implementation einer erweiternden Schnittstelle durch eine Klasse oder Struktur sind auch die Handlungskompetenzen der Basisschnittstellen zu realisieren.

Während bei *Klassen* die Mehrfachvererbung *nicht* unterstützt wird, ist sie bei *Schnittstellen* möglich, wird aber nur selten benötigt.

Weil die Schnittstellenhierarchie von der Klassenhierarchie unabhängig ist, kann ein Interface von beliebigen Klassen und Strukturen implementiert werden.

8.2 Kovariante und kontravariante Typparameter in generischen Schnittstellen

Im Zusammenhang mit den Spezialisierungsbeziehungen zwischen Klassen zeigt sich beim Einsatz der in Kapitel 7 vorgestellten generischen Kollektionen ein Problem, das nun durch eine Erweiterung der Generizitätslösung von C# gelöst werden soll. Um zunächst noch einmal das Problem und anschließend die Lösung zu präsentieren, betrachten wir eine Klasse namens `Figur`, die nur begrenzte Ähnlichkeit mit einer namensgleichen früheren Beispielklasse besitzt und durch Instanzeigenschaften u.a. die X- und die Y-Koordinate ihrer Objekte veröffentlicht:

```
public class Figur {
   protected String name = "unbenannt";
   protected double xpos, ypos;

   public Figur(double x, double y) {
      if (x >= 0.0 && y >= 0.0) {
         xpos = x; ypos = y;
      }
   }
   public Figur() { }

   public String Name { get { return name; } }
   public double X { get { return xpos; } }
   public double Y { get { return ypos; } }
}
```

Von `Figur` stammt die Klasse `Kreis` ab:

```
public class Kreis : Figur {
   protected double radius;

   public Kreis(double x, double y, double rad) : base(x, y) {
      if (rad >= 0.0) radius = rad;
   }
   public Kreis() { }

   public double Radius { get { return radius; } }
}
```

Es wird (in einer beliebigen Klasse) eine statische Methode namens `SmallestX()` definiert, die für eine Liste mit `Figur`-Elementen die insgesamt kleinste X-Koordinate ermittelt:

```
static double SmallestX(List<Figur> li) {
   double smallestX = Double.MaxValue;
   foreach (Figur f in li)
      if (f.X < smallestX)
         smallestX = f.X;
   return smallestX;
}
```

Während die Methode bei einem Aktualparameter mit dem Typ **List**<`Figur`> erwartungsgemäß arbeitet, scheitert die Übersetzung bei einem Aktualparameter mit dem Typ **List**<`Kreis`>, obwohl ein `Kreis`-Objekt alle Eigenschaften und Kompetenzen eines `Figur`-Objekts besitzt:

```
static void Main() {
    var listeF = new List<Figur> { new Figur(1, 2), new Figur(3, 4) };
    var listeK = new List<Kreis> { new Kreis(2, 2, 1), new Kreis(3, 4, 1) };
    Console.WriteLine(SmallestX(listeF));
    Console.WriteLine(SmallestX(listeK)); // verboten!
}
```

(lokale Variable) List<Kreis> listeK

Argument "1": Konvertierung von "System.Collections.Generic.List<Kreis>" in "System.Collections.Generic.List<Figur>" nicht möglich.

Mögliche Korrekturen anzeigen (Strg+.)

Das bereits aus Abschnitt 7.2.3 bekannte Problem besteht darin, dass die Typformalparameter von generischen Klassen *invariant* sind, sodass die Klasse **List**<`Kreis`> *keine* Spezialisierung der Klas-

se **List**<Figur> ist. So wird verhindert, dass z.B. ein Objekt der Klasse **List**<Kreis> einer Referenzvariablen vom Typ **List**<Figur> zugewiesen wird. Über diese Referenz könnten nämlich Figur-Objekte in die Kreis-Liste eingeschmuggelt werden.

Bei der eben intendieren Verwendung eines **List**<Kreis> - Objekts als SmallestX() - Aktualparameter für einen Formalparameter vom Typ **List**<Figur> findet aber *keine* Änderung der Parameterliste statt. Wenn man dem Compiler versichert, dass die Parameterliste nur als *Quelle* von Objekten, nicht aber als *Senke* bzw. Ablage für Objekte verwendet wird, sollte er die Zuweisung erlauben. Genau dies ermöglichen die im nächsten Abschnitt beschriebenen *kovarianten* Typformalparameter von Schnittstellen.

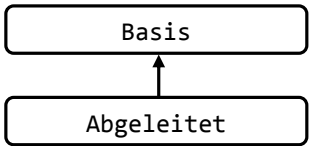
Im Abschnitt 8.2.2 werden *kontravariante* Typformalparameter vorgestellt, die eine weitere Flexibilisierung der Generizität in C# erlauben. Für beide Formen der Varianz (Ko- und Kontravarianz) gelten folgende Regeln:[1]

- Variante Typparameter werden seit .NET 4.0 unterstützt.
- Sie sind nur bei generischen Schnittstellen und generischen Delegaten (siehe Abschnitt 9.1.6) erlaubt, also insbesondere *nicht* bei generischen Klassen. Um das Problem im Beispiel zu beheben, muss also noch eine generische Schnittstelle ins Spiel kommen.
- Immerhin dürfen eine generische Schnittstelle oder ein generischer Delegat sowohl kovariante als auch kontravariante Typparameter besitzen.
- Die Varianz klappt nur mit Referenztypen. Es ist also z.B. nicht erlaubt, einen als kovariant deklarierten Typparameter durch einen Werttyp zu spezialisieren.

8.2.1 Kovarianz

Das generische Interface **IEnumerable<out T>** im FCL-Namensraum **System.Collections.Generic** definiert die Voraussetzungen für eine iterierbare Kollektion (vgl. Abschnitt 8.6). Seit .NET 4.0 ist sein Typformalparameter **T** durch das Schlüsselwort **out** als kovariant definiert. Damit wird festgelegt, dass **T** *nicht* als Parameter einer Schnittstellenmethode, sondern ausschließlich zur Spezifikation der Rückgabe verwendet werden darf.[2]

Infolgedessen bleibt bei zwei Klassen mit der Spezialisierungsbeziehung



für die beiden zugehörigen **IEnumerable<T>** - Implementierungen die **Zuweisungskompatibilität erhalten**. Dies bedeutet, dass der Compiler an Stelle eines Objekts vom **IEnumerable**<Basis> auch ein Objekt vom spezielleren Typ **IEnumerable**<Abgeleitet> akzeptiert.

Für die Schnittstelle **IEnumerable<out T>** ist die Deklaration des Typparameters als kovariant gerechtfertigt, weil die einzige Schnittstellenmethode **GetEnumerator()** ein Objekt liefert, das die Schnittstelle **IEnumerator<T>** erfüllt (vgl. Abschnitt 8.6):

```
IEnumerator<T> GetEnumerator();
```

Auch in der Schnittstelle **IEnumerator<out T>** ist **T** als kovariant definiert ist. Der Typformalparameter hat dort seinen einzigen Auftritt als **get**-Rückgabe der Eigenschaft **Current**:

```
T Current { get; }
```

---

1 Siehe z.B. https://msdn.microsoft.com/en-us/library/dd799517(v=vs.110).aspx

2 Selbst Ausgabeparameter vom Typ **T** sind verboten.

Im Ergebnis kann z.B. einer Variablen vom Typ **IEnumerable<Object>** ein Objekt vom Typ **List<String>** zugewiesen werden, weil dieses Objekt die Schnittstelle **IEnumerable<String>** erfüllt und außerdem **String** von **Object** abstammt:

```
IEnumerable<Object> lob = new List<String>();
```

Über eine Referenz vom Schnittstellentyp **IEnumerable<Object>** wird von einem Objekt der Klasse **List<String>** eine Rückgabe vom Typ **IEnumerator<Object>** erwartet. Das tatsächlich erstellte Objekt vom Typ **IEnumerator<String>** wird dieser Erwartung gerecht, weil seine **Current**-Eigenschaft den Rückgabetyp **String** besitzt.

Generell kann bei einem kovarianten, also ausschließlich ausgaberelevanten Typ ein Objekt mit spezieller Konkretisierung ohne Risiko durch eine allgemeinere Referenz angesprochen werden. Es produziert bei Methodenaufrufen seine speziellen Instanzen, und alles ist in bester Ordnung.

Die oben vorgestellte Methode `SmallestX()` benötigt als Aktualparameter eine Kollektion mit folgenden Eigenschaften:

- Das Interface **IEnumerable<T>** wird erfüllt, damit die **foreach**-Schleife klappt.
- Als Elementtyp ist die Klasse `Figur` oder eine beliebige Ableitung erlaubt, damit die Eigenschaft `X` genutzt werden kann.

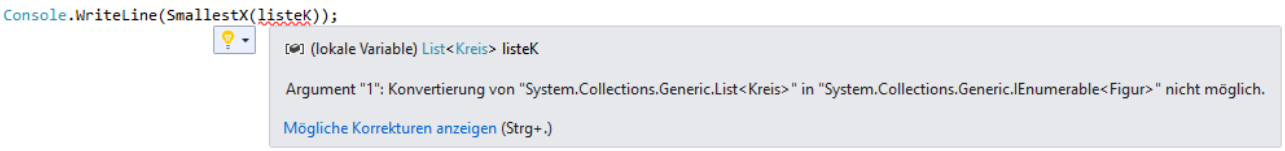
Daher sollte statt **List**`<Figur>` unbedingt **IEnumerable**`<Figur>` als Datentyp verwendet werden:

```
static double SmallestX(IEnumerable<Figur> li) {
   double smallestX = Double.MaxValue;
   foreach (Figur f in li)
      if (f.X < smallestX)
         smallestX = f.X;
   return smallestX;
}
```

Die Kovarianz des **IEnumerable<T>** - Typparameters **T** ausnutzend kann man nun an `SmallestX()` auch eine Liste mit Kreisen übergeben:

```
static void Main() {
   var listeF = new List<Figur> { new Figur(1, 2), new Figur(3, 4) };
   var listeK = new List<Kreis> { new Kreis(2, 2, 1), new Kreis(3, 4, 1) };
   Console.WriteLine(SmallestX(listeF));
   Console.WriteLine(SmallestX(listeK));
}
```

Um zu testen, dass tatsächlich die Kovarianz in der Definition von **IEnumerable<out T>** benötigt wird, kann man im Visual Studio (über den Menübefehl **Projekt > Eigenschaften > Anwendung**) das **Zielframework** auf einen Wert kleiner 4 einstellen, so dass die Kovarianz noch nicht unterstützt wird. Dann scheitert die Übersetzung mit der folgenden Fehlermeldung:



Es ist übrigens keine relevante Einschränkung, dass C# keine generischen *Klassen* mit einem kovarianten Typparameter **T** kennt, denn hier wären etliche Member ausgeschlossen:

- Methoden mit **T**-Parameter
  Auch Ausgabeparameter vom Typ **T** wären verboten.
- Eigenschaften oder Indexer vom Typ **T** mit **set**-Methode
- Veränderliche Felder vom Typ **T**

Ansonsten könnte durch die Verwendung einer allgemeineren Referenzvariablen Daten von einem allgemeineren („schwächeren“) Typ in das Objekt gelangen und damit seine Integrität beschädigen.

Das Konzept einer generischen Klasse mit **out**-Typparameter ist daher wenig sinnvoll und bislang nicht realisiert.[1]

8.2.2 Kontravarianz

Über das generische Interface **IComparable<in T>** im FCL-Namensraum **System** signalisiert ein Datentyp, dass für seine Instanzen eine Ordnung definiert ist. Das Interface verlangt von implementierenden Typen, eine Methode mit dem folgenden Definitionskopf:

public int Compare(T *other***)**

Seit .NET 4.0 ist das Interface **IComparable<in T>** über das Schlüsselwort **in** als *kontravariant* definiert. Damit wird dem Compiler signalisiert, dass **T** in den Schnittstellenmethoden *nicht* zur Spezifikation eines Rückgabewerts, sondern ausschließlich für Methoden*argumente* benutzt wird.

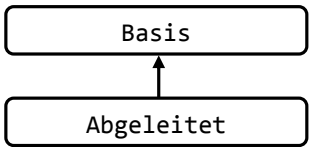
Durch die Kontravarianz des Typparameters von **IComparable<in T>** ist sichergestellt, dass Methoden dieser Schnittstelle ...

- lediglich **T**-Instanzen verarbeiten,
- aber keinesfalls **T**-Instanzen abliefern.

Wenn statt **T** eine Basisklasse verwendet wird, ist die Typsicherheit nicht gefährdet:

- Eine für Basisklassenobjekte geeignete Methode kommt auch mit **T**-Objekten zurecht.
- Es passiert nie, dass statt eines **T**-Objekts ein (schwächer ausgestattetes) Basisklassenobjekt ausgeliefert wird.

Infolgedessen wird bei zwei Klassen mit der Spezialisierungsbeziehung



für die beiden zugehörigen **IComparable<T>** - Implementierungen die **Zuweisungskompatibilität umgekehrt**. Dies bedeutet, dass der Compiler an Stelle eines Objekts vom **IComparable<Abgeleitet>** auch ein Objekt vom spezielleren Typ **IComparable<Basis>** akzeptiert.

Zur Demonstration implementieren wie in der Klasse `Figur` im Vorgriff auf den Abschnitt 8.3 die Schnittstelle **IComparable<Figur>**:

```
using System;
public class Figur : IComparable<Figur> {
   .  .  .
   public int CompareTo(Figur vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

Weil die Klasse `Kreis` von der Klasse `Figur` abstammt, implementiert sie ebenfalls die Schnittstelle **IComparable<Figur>**. Wegen der Kontravarianz des Typformalparameters von **IComparable<T>** ist **IComparable<Figur>** zuweisungskompatibel zu **IComparable<Kreis>**.

---

[1] Siehe http://stackoverflow.com/questions/2733346/why-isnt-there-generic-variance-for-classes-in-c-sharp-4-0/2734070#2734070

Wo eigentlich der Typ **IComparable**<Kreis> verlangt ist, wird also auch **IComparable**<Figur> akzeptiert.

Zum Sortieren einer Liste mit Kreisen

```
var listeK = new List<Kreis> { new Kreis(1, 5, 1), new Kreis(3, 4, 1) };
```

bietet sich die Methode **Sort()** der Klasse **List<T>** an:

```
listeK.Sort();
```

Sie erwartet, dass der Elementtyp **T** die Schnittstelle **IComparable<T>** erfüllt, so dass bei unserer **List**<Kreis> - Kollektion der Elementtyp Kreis die Schnittstelle **IComparable**<Kreis> erfüllen muss, was nach obigen Überlegungen aufgrund der Kontravarianz von **T** auch der Fall ist.

Um zu testen, dass tatsächlich die Kontravarianz in der Definition von **ICompable<in T>** für einen erfolgreichen Aufruf der **Sort()** - Methode benötigt wird, kann man im Visual Studio (über den Menübefehl **Projekt > Eigenschaften > Anwendung**) das **Zielframework** auf einen Wert kleiner 4 einstellen, so dass die Kontravarianz noch nicht unterstützt wird. Dann wird der **Sort()** - Aufruf zwar leider kritiklos übersetzt, doch die Ausführung scheitert mit einer **InvalidOperation-Exception**.

8.3 Interfaces implementieren

Soll für eine Klasse oder Struktur angezeigt werden, dass ihre Instanzen die Kompetenzen bestimmter Schnittstellen besitzen, dann müssen diese Schnittstellen im Kopf der Typdefinition aufgelistet werden. Man setzt hinter den Typbezeichner einen Doppelpunkt, gibt bei Klassen ggf. zunächst eine Basisklasse an und listet dann die Schnittstellen auf, untereinander und von der Basisklasse jeweils durch ein Komma getrennt.

Ein Beispiel kennen Sie bereits, weil in Abschnitt 8.2.2 zur Erläuterung der Kontravarianz eine Variante der Klasse Figur vorgestellt wurde, die das Interface **IComparable**<Figur>, implementiert, damit Figur-Kollektionen bequem sortiert werden können:

```
using System;
public class Figur : IComparable<Figur> {
   protected String name = "unbenannt";
   protected double xpos, ypos;

   public Figur(String n, double x, double y) {
      name = n;
      if (x >= 0.0 && y >= 0.0) {
         xpos = x;
         ypos = y;
      }
   }
   public Figur() {}

   public String Name { get { return name; } }
   public double X { get { return xpos; } }
   public double Y { get { return ypos; } }

   public int CompareTo(Figur vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

Alle Handlungskompetenzen einer Schnittstelle, die im Kopf einer Klassen- oder Strukturdefinition angemeldet wird, müssen implementiert werden. Bei der generischen Schnittstelle **IComparable<T>** ist nur eine **public**-Methode namens **CompareTo()** mit einem Parameter vom Typ **T** und einem Rückgabewert vom Typ **int** erforderlich. In semantischer Hinsicht soll **CompareTo()** eine `Figur` beauftragen, sich mit dem per Aktualparameter bestimmten Artgenossen zu vergleichen. Bei obiger Realisation werden Figuren nach der X-Koordinate ihrer linken oberen Ecke verglichen:

- Liegt die angesprochene Figur links vom Vergleichspartner, dann wird die Zahl -1 zurück gemeldet.
- Haben beide Figuren in der linken oberen Ecke dieselbe X-Koordinate, lautet die Antwort 0.
- Ansonsten wird eine 1 gemeldet.

Weil die Methoden einer Schnittstelle grundsätzlich **public** sind, muss diese Schutzstufe auch für die *implementierenden* Methoden gelten, wozu in deren Definition der Zugriffsmodifikator explizit anzugeben ist. Anderenfalls äußert sich der Compiler so:

```
Figur.cs(2,14): error CS0737: "Figur" implementiert den Schnittstellenmember
        "System.IComparable<Figur>.CompareTo(Figur)" nicht.
        "Figur.CompareTo(Figur)" ist nicht öffentlich und kann daher keinen
        Schnittstellenmember implementieren.
```

Für die implementierenden Methoden muss (und darf) das Schlüsselwort **override** (im Unterschied zur Situation beim Überschreiben von abstrakten Methoden) *nicht* angegeben werden.

Soll eine implementierende Methode überschreibbar sein, ist das Schlüsselwort **virtual** anzugeben.

Eine Klasse kann auf das Implementieren einiger Interface-Handlungskompetenzen verzichten und diese wie auch sich selbst als **abstract** deklarieren.

Weil sich `Figur`-Objekte mit einem Artgenossen vergleichen können, gelingt das Sortieren einer **List**<`Figur`> - Kollektionen mit der Methode **Sort()** mühelos, z.B.:

Quellcode	Ausgabe
`using System;` `using System.Collections.Generic;` `class Prog {` `  static void Main() {` `  var lf = new List<Figur>();` `  lf.Add(new Figur("A", 250.0, 50.0));` `  lf.Add(new Figur("B", 150.0, 50.0));` `  lf.Add(new Figur("C", 50.0, 50.0));` `  Console.WriteLine(lf[0].Name+" "+lf[1].Name+" "+lf[2].Name);` `  lf.Sort();` `  Console.WriteLine(lf[0].Name+" "+lf[1].Name+" "+lf[2].Name);` `}`	`A B C` `C B A`

Wie Sie wissen, ist bei C# - *Klassen* keine **Mehrfachvererbung** erlaubt. Diese Möglichkeit wurde wegen einiger Risiken bewusst *nicht* aus C++ übernommen. Allerdings erlauben Schnittstellen eine Ersatzlösung, denn:

- Eine Klasse darf beliebig viele Schnittstellen implementieren, so dass ihre Objekte entsprechend viele Datentypen erfüllen. So könnte man z.B. die Schnittstellen `ITuner` und `IAmplifier` sowie die Klasse `Receiver` derart definieren, dass sich ein `Receiver`-Objekt …
  - wie ein `ITuner`
  - und wie ein `IAmplifier`

verhalten kann. Wie wir inzwischen wissen, wird einer Klasse beim Implementieren von Schnittstellen aber nichts geschenkt, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Leistungen erbringen.

- Bei Schnittstellen ist die Mehrfachvererbung erlaubt. Diese Möglichkeit wird aber sehr viel seltener benutzt als das Implementieren von mehreren Schnittstellen durch eine Klasse.

Im Zusammenhang mit dem Thema *Vererbung* ist von Bedeutung, dass eine abgeleitete Klasse die Schnittstellen-Zulassungen von ihrer Basisklasse übernimmt, ohne die Verpflichtungserklärungen in ihrem eigenen Definitionskopf wiederholen zu müssen. Wird z.B. die Klasse `Kreis` von der oben vorgestellten Klasse `Figur` abgeleitet, dann erfüllt auch die Klasse `Kreis` die Schnittstelle **IComparable**<`Figur`>, jedoch *nicht unmittelbar* auch die Schnittstelle **IComparable**<`Kreis`>.

Dass im betrachteten Spezialfall mittelbar die Schnittstelle **IComparable**<`Kreis`> doch erfüllt ist, liegt an der seit .NET 4 gegebenen Kontravarianz des Typformalparameters **T** in der generischen Schnittstelle **IComparable<T>** (siehe Abschnitt 8.2.2). Unter dieser Voraussetzung arbeitet die **List<T>** - Methode **Sort()** auch bei einer **List**<`Kreis`> - Kollektion, z.B.:

Quellcode	Ausgabe

```
using System;
using System.Collections.Generic;
class Prog {
   static void Main() {
      var lk = new List<Kreis>();
      lk.Add(new Kreis("A", 250.0, 50.0, 10.0));
      lk.Add(new Kreis("B", 150.0, 50.0, 20.0));
      lk.Add(new Kreis("C", 50.0, 50.0, 30.0));
      Console.WriteLine(lk[0].Name + " " + lk[1].Name + " " + lk[2].Name);
      lk.Sort();
      Console.WriteLine(lk[0].Name + " " + lk[1].Name + " " + lk[2].Name);
  }
}
```

```
A B C
C B A
```

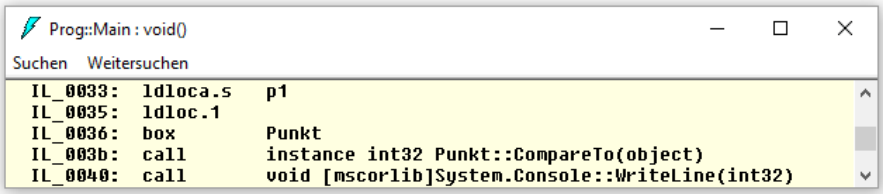
Besteht die Wahl, ein generisches oder ein nicht-generisches Interface zu implementieren, sollte in der Regel die generische Variante gewählt werden. Man gewinnt Typsicherheit, erspart sich lästige Typumwandlungen und vermeidet bei Strukturen zeitaufwendige Boxing-Operationen. Dies soll bei einer einfachen Struktur für Punkte in der Zahlenebene mit einem Vergleich anhand der X-Koordinate demonstriert werden. In der ersten Variante wird die traditionelle Schnittstelle **IComparable** implementiert, also eine **CompareTo()** - Methode mit **Object**-Parameter realisiert:

```
using System;
public struct Punkt : IComparable {
   double xpos, ypos;
   public Punkt(double x, double y) {
      xpos = x; ypos = y;
   }
   public int CompareTo(object v) {
      Punkt vergl = (Punkt) v;
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

In **CompareTo()** ist eine explizite Typumwandlung von **Object** zu `Punkt` erforderlich, die zu einer vom Compiler nicht zu verhindernden **InvalidCastException** führen kann, z.B. im folgenden Programm:

```
using System;
class Prog {
   static void Main() {
      Punkt p1 = new Punkt(1.0, 2.0),
            p2 = new Punkt(2.0, 3.0);
      Console.WriteLine(p1.CompareTo(p2)); // Boxing
      Console.WriteLine(p1.CompareTo(13)); // Laufzeitfehler
   }
}
```

Wie der CIL-Code der **Main()** - Methode zeigt, erfordert außerdem jeder *gelungene* **CompareTo()** - Aufruf eine Boxing -Operation:



Implementiert die Struktur stattdessen das Interface **IComparable**<Punkt>,

```
using System;
public struct Punkt : IComparable<Punkt> {
   double xpos, ypos;
   public Punkt(double x, double y) {
      xpos = x; ypos = y;
   }
   public int CompareTo(Punkt vergl) {
      if (xpos < vergl.xpos)
         return -1;
      else
         if (xpos > vergl.xpos)
            return 1;
      return 0;
   }
}
```

kann die fehlerhafte Quellcodezeile nicht mehr übersetzt werden,

```
Console.WriteLine(p1.CompareTo(13));
```

und ein gelungener **CompareTo()** - Aufruf geht ohne Boxing über die Bühne:

8.4 Interfaces als Referenzdatentypen

Bei der Definition einer Schnittstelle entsteht ein neuer *Referenzdatentyp*, der zur Variablendeklaration und als Formalparameter einsetzbar ist. Eine Referenzvariable des neuen Typs kann auf Instanzen einer implementierenden Klasse oder Struktur zeigen, z.B.:

Quellcode	Ausgabe
`using System;` `public interface IType {` `  string SagWas();` `}` `class K1 : IType {` `  public string SagWas() { return "K1"; }` `}` `class K2 : IType {` `  public string SagWas() { return "K2"; }` `}` `struct S : IType {` `  public string SagWas() { return "S"; }` `}` `class Prog {` `  static void Main() {` `  IType[] ida = {new K1(), new K2(), new S()};` `  foreach (IType idin in ida)` `  Console.WriteLine(idin.SagWas());` `  }` `}`	`K1` `K2` `S`

Damit wird es möglich, Instanzen von beliebigen Klassen und Strukturen, die dasselbe Interface implementieren, in einem Array (oder einer anderen Kollektion) gemeinsam zu verwalten. Über eine Interface-Variable können die Methoden der Schnittstelle sowie die Methoden der Klasse **Object** aufgerufen werden.

Interface-Typen sind grundsätzlich *Referenztypen*, so dass eine Variable mit einem solchen Datentyp nur eine Objektadresse aufnehmen kann. Wird einer solchen Referenzvariablen eine Strukturinstanz zugewiesen, ist ein Boxing fällig.

Weil die Methoden einer Schnittstelle grundsätzlich virtuell sind, erfolgt ihr Aufruf mit dynamischer Bindung (polymorph). In zeitkritischen Programmsituationen muss eine hohe Zahl von polymorphen Methodenaufrufen und Boxing-Operationen eventuell vermieden werden.

8.5 Explizite Schnittstellenimplementierung

In den bisherigen Beispielen wurden Interface-Verpflichtungserklärungen durch **public**-deklarierte Methoden des implementierenden Typs realisiert. Bei dieser sogenannten *impliziten* Implementation kann es zu Namenskollisionen kommen, wenn …

- ein Typ mehrere Schnittstellen implementiert,
- zwei oder mehrere Schnittstellen eine Methode mit demselben Namen und derselben Parameterliste besitzen, für die aber unterschiedliche Funktionsweisen vorgesehen sind.

Für diese relativ seltene Situation bietet C# die so genannte *explizite* Schnittstellenimplementierung, wobei der implementierende Typ …

- die Methode mehrfach implementiert,
- bei jeder Implementation dem Methodennamen den Namen der zugehörigen Schnittstelle durch Punkt getrennt voranstellt,
- *keine* Zugriffsmodifikatoren angibt.

Im folgenden Beispiel implementiert die Klasse `M` die Methoden `I1.M()` und `I2.M()` durch explizite Angabe der jeweiligen Schnittstelle:

```
public interface I1 {
   int M();
}
public interface I2 {
   int M();
}
public class K : I1, I2 {
   int I1.M() {
      return 13;
   }

   int I2.M() {
      return 4711;
   }
}
```

Ihre Objekte können *beide* Methoden ausführen, sofern sie über den passenden Interface-Datentyp angesprochen werden, z.B.:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  K k = new K();` `  I1 i1 = k;` `  I2 i2 = k;` `  Console.WriteLine(i1.M());` `  Console.WriteLine(i2.M());` `  }` `}`	`13` `4711`

Über eine Referenzvariable von eigenen Datentyp angesprochen, ist jedoch *keine* Methode namens `M()` verfügbar, so dass die folgende Anweisung

```
Console.WriteLine(k.M());
```

den Compiler zu der Fehlermeldung veranlasst:

```
"K" enthält keine Definition für "M"
```

Würde die Klasse `K` die Methode `M()` auf bisher gewohnte Weise implementieren, wäre *dasselbe* Verhalten über Referenzen der Typen `I1`, `I2` und `K` abrufbar.

Weitere Hinweise zur expliziten Implementation finden sich z.B. bei Richter (2006, S. 343).

8.6 Iteratoren

Ein Iterator erlaubt es, die Elemente einer Kollektion nacheinander abzurufen und wird meist implizit im Rahmen einer **foreach**-Schleife verwendet.

8.6.1 IEnumerable-Implementation

Für das in Abschnitt 8.2.1 zur Illustration der Kovarianz verwendete und offenbar sehr wichtige (weil z.B. für die **foreach**-Schleife relevante) Interface **IEnumerable<T>**

```
public interface IEnumerable<out T> : IEnumerable {
   new IEnumerator<T> GetEnumerator();
}
public interface IEnumerable {
   IEnumerator GetEnumerator();
}
```

ist (wie auch für die nichtgenerische Variante **IEnumerable**) noch zu erläutern, wie die Methode **GetEnumerator()** zu implementieren ist. Dabei kommt eine spezielle, als *Iterator* bezeichnete Methode ins Spiel. Was zunächst nach Lernaufwand klingt, stellt sich bald als Entlastung heraus, weil wir darum herum kommen, eine Klasse zu erstellen, die das Interface **IEnumerator<T>** inklusive der Erblasten **IDisposable** und **IEnumerator** implementiert:

```
public interface IEnumerator<out T> : IDisposable, IEnumerator {
   new T Current {
      get;
   }
}
public interface IEnumerator {
   bool MoveNext();
   Object Current {
      get;
   }
   void Reset();
}
public interface IDisposable {
   void Dispose();
}
```

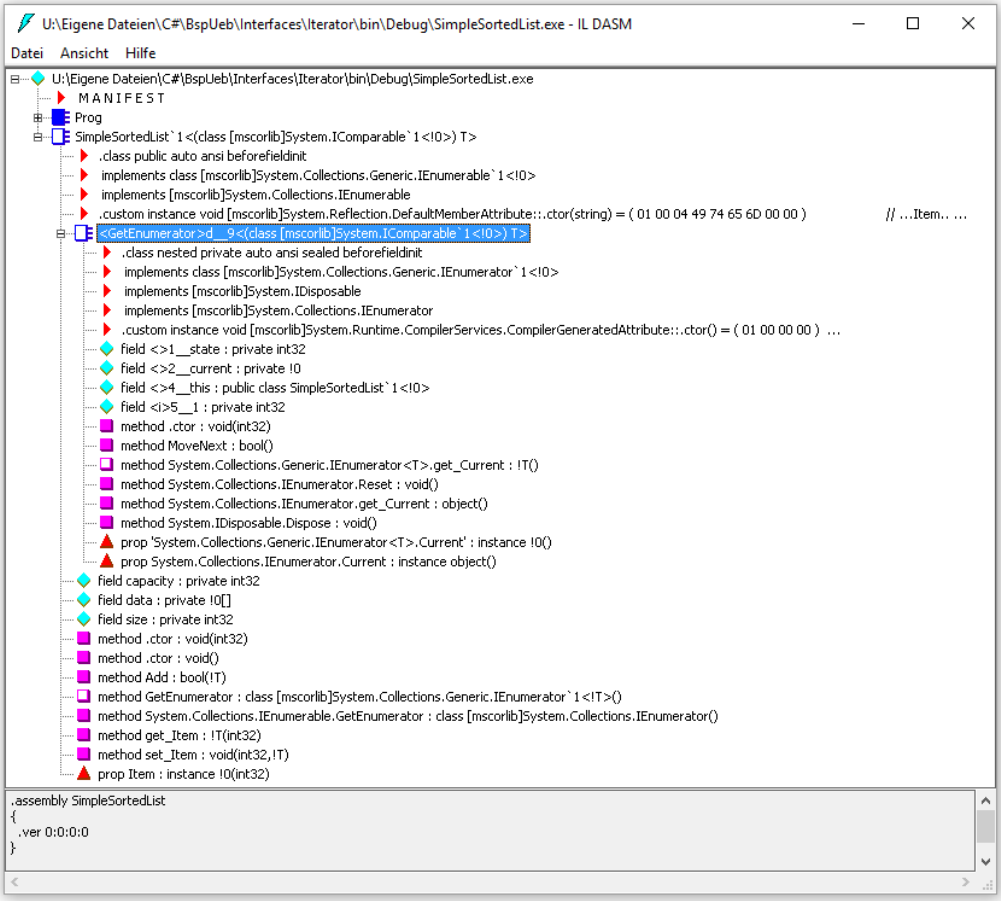
Als Anwendungsbeispiel erweitern wir die in Abschnitt 7.2.2 erstellte Kollektionsklasse `SimpleSortedList<T>`, die ihre Elemente in einem sortieren Zustand hält, um einen Iterator, damit über die Elemente per **foreach**-Schleife iteriert werden kann. Der Zweck ist im Wesentlichen bereits durch die folgende **GetEnumerator()** - Implementation erreicht:

```
public IEnumerator<T> GetEnumerator() {
   for (int i = 0; i < size; i++)
      yield return data[i];
}
```

Bei einem Objekt der Klasse `SimpleSortedList<T>` befinden sich die Elemente in einem privaten Array namens `data`. Der Iterator liefert in einer gewöhnlichen **for**-Schleife die Elemente Stück für Stück in einer Anweisung mit dem Namen **yield return** aus. Es muss nicht unbedingt eine Schleife verwendet werden. Man könnte auch schlicht mehrere **yield return** - Anweisungen hintereinander schreiben.

Im praktischen Einsatz wird nach jeder **yield return** - Ausführung die erreichte Position im Code gespeichert, und beim nächsten Aufruf des Iterators wir die Ausführung an der gespeicherten Position fortgesetzt.[1] Wir wollen gar nicht erst darüber nachdenken, mit welcher Genialität und mit welchem Aufwand der Compiler aus unserem simplen Beitrag eine **IEnumerator<T>** - Implementation erstellt. Eine Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass der Compiler unsere **GetEnumerator()** - Implementation in eine innere Klasse umgesetzt hat:

[1] https://msdn.microsoft.com/de-de/library/mt639331.aspx



Aufgrund der simplen **GetEnumerator()** - Implementation werden Objekte der Klasse `SimpleSortedList<T>` nun von der **foreach**-Schleife akzeptiert:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  var sst = new SimpleSortedList<String>(5);` `  sst.Add("one"); sst.Add("two"); sst.Add("three");` `  sst.Add("four"); sst.Add("five");` `  foreach (string s in sst)` `  Console.WriteLine(s);` `  }` `}`	`five` `four` `one` `three` `two`

Eine **foreach**-Anweisung wird vom Compiler ungefähr so umgesetzt (siehe ECMA 2006, S. 238):

- Durch einen Aufruf der Methode **GetEnumerator()** wird ein Objekt vom Typ **IEnumerable<T>** erstellt (anschließend *Enumerator* genannt).
- In einem geschützten **try**-Block läuft eine **while**-Schleife, die im Bedingungsteil einen Aufruf der **MoveNext()** - Methode des Enumerators enthält. In Anweisungsteil der **while**-Schleife wird die **Current**-Eigenschaft des Enumerators verwendet, um das aktuelle Element abzurufen.

- Im **finally**-Zweig der **try-finally** - Anweisung wird die **Dispose()** - Methode des Enumerators aufgerufen. So ist sichergestellt, dass **Dispose()** auch beim Auftreten eines Fehlers aufgerufen wird. Der Aufruf (und vor allem eine eigene Implementation von **Dispose()**) sind dann relevant, wenn ein Enumerator z.B. die Zeilen einer Datenbanktabelle auflistet und folglich nach getaner Arbeit bzw. nach einem Unfall die Verbindung zur Datenbank auf jeden Fall schließen sollte.

Wegen des Aufwands und des Fehlerrisikos sollte ein Enumerator in der Regel nur indirekt via **foreach** eingesetzt werden.

Wenn die Klasse `SimpleSortedList<T>` von sich behaupten möchte, die Schnittstelle **IEnumerable<T>** zu implementieren,

```
public class SimpleSortedList<T> : IEnumerable<T>  where T : IComparable<T> { ... }
```

muss sie auch die nichtgenerische Überladung von **GetEnumerator()** implementieren, was aber praktisch keinen Zusatzaufwand bedeutet:

```
IEnumerator IEnumerable.GetEnumerator() {
   return GetEnumerator();
}
```

Die vom Compiler erstellte **IEnumerator<T>** - Implementation erfüllt auch das nichtgenerische Interface:

```
public interface IEnumerator<out T> : IDisposable, IEnumerator { ... }
```

Weil die gerade vorgestellte **GetEnumerator()** - Überladung die zur Schnittstelle **IEnumerable** gehörige Methode implementiert, ist der Schnittstellenname voranzustellen (siehe Abschnitt 8.5 zur expliziten Schnittstellenimplementierung).

8.6.2 Andere Iteratoren

Als Alternative oder zur Ergänzung der eben beschriebenen Implementation der (generischen) Schnittstelle **IEnumerable** bietet C# weitere Verfahren zur Realisation von Iteratoren.

In der Klasse `SimpleSortedList<T>` soll als Ergänzung zum Standarditerator auch ein Iterator mit Parametern zur Bereichsspezifikation angeboten werden. Dabei ist ein **benannter Iterator**, zu verwenden und statt **GetEnumerator()** ein alternativer Methodenname zu vergeben. Außerdem ist statt **IEnumerator<T>** der Rückgabetyp **IEnumerable<T>** vorgeschrieben:

```
public IEnumerable<T> RangeIt(int first, int last) {
   if (first < 0 || last >= size)
      yield break;
   for (int i = first; i < last; i++)
      yield return data[i];
}
```

In der **for**-Schleife werden nur noch die Kollektionselemente innerhalb des gewünschten Bereichs per **yield return** ausgeliefert. Wird die Methode mit ungeeigneten Bereichsgrenzen aufgerufen, liefert sie mit Hilfe der Anweisung **yield break** ein leeres **IEnumerable<T>** - Objekt.

Das folgende Programm demonstriert die Bereichsiteration ohne und mit Spezifikationsfehler. In der **foreach**-Schleife ist an das Kollektionsobjekt ein Methodenaufruf mit dem Namen des Bereichsiterators zu richten:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  var si = new SimpleSortedList<int>(5);` `  si.Add(11); si.Add(2); si.Add(1); si.Add(4); si.Add(7);` `  Console.WriteLine("Standard-Iterator:");` `  foreach (int i in si)` `  Console.WriteLine(i);` `  Console.WriteLine("\nBereichs-Iterator:");` `  foreach (int i in si.RangeIt(1, 4))` `  Console.WriteLine(i);` `  Console.WriteLine("\nBereichüberschreitung:");` `  foreach (int i in si.RangeIt(1, 14))` `  Console.WriteLine(i);` `  }` `}`	`Standard-Iterator:` `1` `2` `4` `7` `11` `Bereichs-Iterator:` `2` `4` `7` `Bereichüberschreitung:`

Natürlich lassen sich mit benannten Iteratoren auch andere Aufgaben als die Bereichsauswahl realisieren. Ein Typ darf benannte Iteratoren *unabhängig* von der Implementation der Schnittstelle **IEnumerable** anbieten.

Das gilt auch für die in C# ebenfalls erlaubten **Iteratoren mit Eigenschaftssyntax**. Zur Demonstration erhält die Klasse `SimpleSortedList<T>` einen Abwärtsiterator namens `DownIt`:[1]

```
public IEnumerable<T> DownIt {
  get {
    for (int i = size - 1; i >= 0; i--)
      yield return elements[i];
  }
}
```

In der **foreach**-Schleife ist an das Kollektionsobjekt der Eigenschaftsname anzuhängen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  var si = new SimpleSortedList<int>(5);` `  si.Add(11); si.Add(2); si.Add(1); si.Add(4); si.Add(7);` `  Console.WriteLine("\nAbwärts-Iterator:");` `  foreach (int i in si.DownIt)` `  Console.WriteLine(i);` `  }` `}`	`Abwärts-Iterator:` `11` `7` `4` `2` `1`

Weitere Details zu Iteratoren finden sich z.B. in Griffith (2013, S. 175ff) und Mössenböck (2016, S. 169ff).

[1] Idee übernommen aus https://msdn.microsoft.com/de-de/library/ee5kxzk0(v=vs.100).aspx

8.7 Übungsaufgaben zu Kapitel 8

1) Erstellen Sie eine Variante unserer `Bruch`-Klasse (vgl. z.B. Abschnitt 4.1.3), welche die generischen Schnittstellen **IComparable<T>** und **ICloneable** implementiert. Die vorhandene `Bruch`-Methode `Klone()`

```
public Bruch Klone() {
  return new Bruch(zaehler, nenner, etikett);
}
```

wird dabei *nicht* überflüssig, weil sie im Gegensatz zur **ICloneable**-Variante eine `Bruch`-Referenz abliefert und damit Typumwandlungen erspart.

2) Wie unterscheiden sich Interfaces von abstrakten Klassen?

3) In welchem Sinn kann die .NET-Schnittstellentechnik partiell die Mehrfachvererbung von C++ ersetzen?

9 Delegaten und Ereignisse

Mit den Delegaten lernen wir einen weiteren C# - Datentyp kennen. Ein Delegatenobjekt zeigt auf eine Methode mit einer bestimmten Signatur und einem bestimmten Rückgabetyp, und diese Methode kann über das Delegatenobjekt aufgerufen werden. In der Regel wird das Aufrufziel eines Delegaten *nicht* von derselben Programmeinheit festgelegt, die den Delegaten schließlich verwendet. Stattdessen besitzt z.B. eine Methode einen Formalparameter mit Delegatentyp, sodass beim Aufruf per Delegatenobjekt eine „Hilfsmethode“ zu übergeben ist, die innerhalb der Rahmenmethode als Funktionsergänzung oder Konfiguration verwendet wird. Man kann hier von einer **Funktionsinjektion** sprechen. Weiterhin kann der Rahmenmethode per Delegatenobjekt die Möglichkeit zur **Kommunikation per Rückruf** gegeben werden.

Funktionsinjektion und Rückrufkommunikation sind auch beim Klassendesign von Nutzen und hier über Mitgliedsobjekte mit Delegatentyp zu realisieren.

Bei den von einer Klasse veröffentlichten **Ereignissen** handelt es sich um spezielle Delegatenobjekte mit ausschließlich kommunikativer Funktion. Interessenten können eine Ereignisbehandlungsmethode bei der Ereignisquelle registrieren lassen, die zu bestimmten Gelegenheiten aufgerufen werden soll. So informiert z.B. ein Befehlsschalter in der Bedienoberfläche eines Programms über sein **Click**-Interessenten darüber, dass er vom Benutzer ausgelöst worden ist. Ein anderes Steuerelement lässt beim Ereignis **Changed** eines Objekts vom Typ **ObservableCollection<T>** eine Behandlungsmethode registrieren, um auf eine Änderung von Hintergrunddaten mit einer Aktualisierung seiner Anzeige reagieren zu können.

9.1 Delegaten

In diesem Abschnitt lernen Sie die Delegatentypen kennen, die zunächst etwas abstrakt wirken, aber speziell im Kontext der ereignisorientierten Programmierung unverzichtbar sind. Es handelt sich um Klassen, deren Objekte auf Methoden mit einer bestimmten Parameterliste und einem bestimmten Rückgabetyp zeigen. Dazu enthalten Delegatenobjekte eine (ein- oder mehrelementige) Aufrufliste mit kompatiblen Methoden. Diese Liste kann Instanz- und Klassenmethoden enthalten.

Über ein Delegatenobjekt lassen sich mit *einem* Aufruf alle Methoden seiner Aufrufliste nacheinander starten, wobei die zuletzt ausgeführte Methode beim Rückgabewert und bei eventuell vorhandenen **out**-Parametern dominiert. Somit bietet C# mit den Delegaten eine objektorientierte Variante der Funktions-Pointer aus anderen Programmiersprachen (z.B. C++, Pascal, Modula).

Vielleicht helfen die folgenden Begriffserläuterungen, Missverständnisse zu vermeiden:

Ein **Delegatentyp** besitzt:

- einen Namen, z.B.
  ```
  DemoGate
  ```
- eine Delegatensignatur, bei der im Vergleich zur Methodensignatur auch der Rückgabetyp signifikant ist, z.B.
  ```
  delegate void DemoGate(int w);
  ```

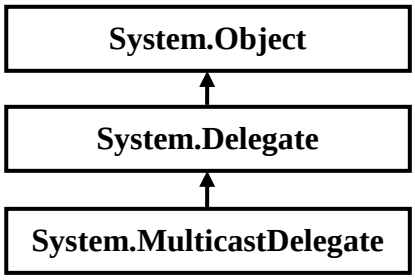
Ein **Delegatenobjekt** besitzt:

- einen Typ, z.B.
  ```
  DemoGate
  ```
- eine veränderbare Aufrufliste mit kompatiblen Instanz- und Klassenmethoden

Ein **Delegatenvariable** besitzt:

- einen Typ, z.B.
  `DemoGate`
- einen Wert, z.B.
  null oder
  die Adresse eines `DemoGate`-Delegatenobjekts

Alle Delegatentypen sind Klassen und stammen implizit von der Klasse **MulticastDelegate** im Namensraum **System** ab:



Delegaten spielen eine unverzichtbare Rolle bei der Ereignisverarbeitung in GUI-Programmen, sind aber auch darüber hinaus von Interesse. Z.B. kann die Funktionalität einer Klasse modifiziert werden, indem einem Feld mit Delegatentyp eine kompatible Methode zugewiesen wird. Wenn die Klasse ihr Delegatenobjekt aufruft, kommt die „injizierte" Methode zum Einsatz. Diese Option zur Verhaltensspezialisierung ist oft flexibler als die Definition von abgeleiteten Klassen.

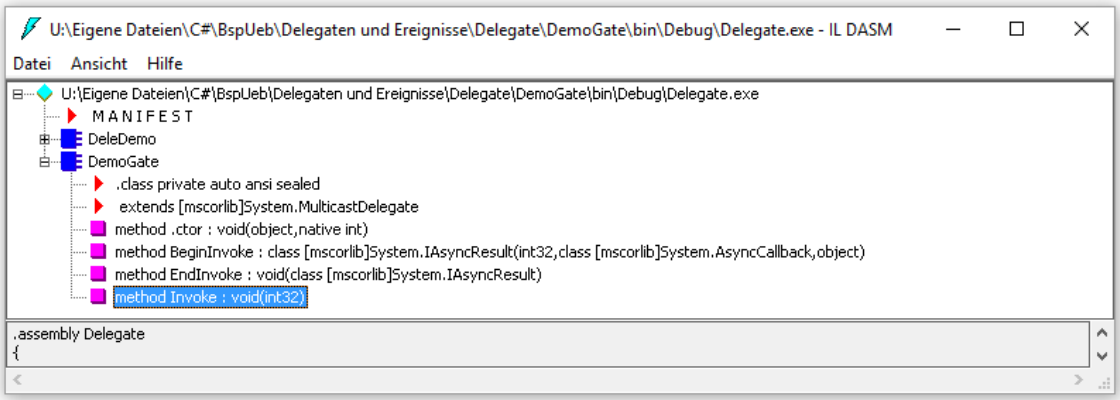
9.1.1 Delegatentypen definieren

Mit der folgenden Anweisung wird unter Verwendung des Schlüsselworts **delegate** der Delegatentyp `DemoGate` definiert:

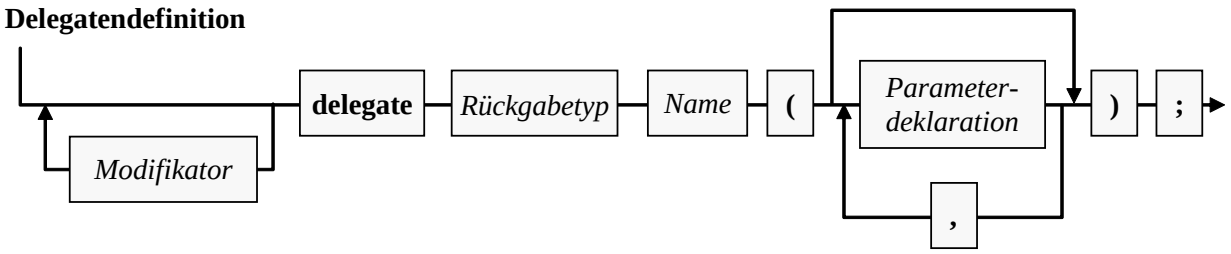
```
delegate void DemoGate(int w);
```

Über Objekte dieses Typs können Instanz- oder Klassenmethoden mit dem Rückgabetyp **void** und einem einzigen Parameter vom Typ **int** aufgerufen werden.

Eine Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass aus der obigen Definition die Klasse `DemoGate` entstanden ist, die u.a. einen Konstruktor und die Instanzmethode **Invoke()** besitzt:



Es folgt ein leicht vereinfachtes Syntaxdiagramm zur Definition eines Delegatentyps:



Bei einem Top-Level - Delegaten sind wie bei anderen Typen die Zugriffsmodifikatoren **public** und **internal** erlaubt. Wird **public** *nicht* angegeben, ist der Delegat nur innerhalb seines Assemblies verwendbar (Schutzstufe **internal**). Für einen eingeschachtelten Delegaten sind dieselben Zugriffsmodifikatoren verfügbar wie für andere Member (siehe Abschnitt 4.10).

In der Literatur wird oft nachlässig behauptet, ein Delegatenobjekt könne auf Methoden mit einer bestimmten *Signatur* zeigen. Wir wissen seit Abschnitt 4.3.5, dass zwei Methoden genau dann dieselbe Signatur haben, wenn die Namen und die Parameterlisten (hinsichtlich Typ und Art aller Formalparameter) übereinstimmen, während die Rückgabetypen keine Rolle spielen. Diese Begriffsdefinition wird in der C# - Sprachdefinition festgelegt (Microsoft 2012, Abschnitt 3.6). Für einen Delegatentypen ist aber der Rückgabetyp essentiell, während der Name einer implementierenden Methode frei wählbar ist. Wir werden im weiteren Verlauf die Anforderungen einer Delegatendefinition an kompatible Methoden als *Delegatensignatur* bezeichnen.

9.1.2 Delegatenobjekte erzeugen und aufrufen

In diesem Abschnitt wird an einem einfachen Beispiel demonstriert, …

- wie der Aufruf einer Methode,
- welche eine bestimmte Delegatensignatur besitzt,
- über ein Objekt dieses Delegatentyps erfolgen kann.

Die folgende Klasse

```
class DeleDemo {
   static void SagA(int w){
      for (int i = 1; i <= w; i++)
         Console.Write('A');
      Console.WriteLine();
   }
   static void Main() {
      DemoGate demoVar = new DemoGate(SagA);
      demoVar(3);
   }
}
```

besitzt zwei statische Methoden:

- Die Methode `SagA()` schreibt eine per Parameter wählbare Anzahl von A's auf die Konsole. Sie erfüllt den Delegatentyp `DemoGate` (definiert in Abschnitt 9.1.1).
- In der **Main()** - Methode wird die lokale Referenzvariable `demoVar` vom Delegatentyp `DemoGate` deklariert und initialisiert. Über den implizit definierten `DemoGate`-Konstruktor wird ein Objekt des Delegatentyps erzeugt, das auf die Methode `SagA()` zeigt.

In der Aufrufliste des über die Referenzvariable `demoVar` ansprechbaren `DemoGate`-Objekts befindet sich ausschließlich die statische Methode `SagA()`. Ein Delegatenobjekt lässt sich wie eine Methode aufrufen. Im Beispiel führt der Aufruf

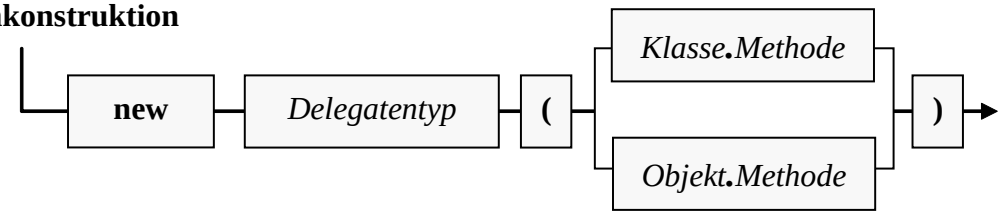
```
demoVar(3);
```

zur Ausgabe:

```
AAA
```

Natürlich unterstützen Delegatenobjekte nicht nur statische Methoden. Dem Standardkonstruktor einer Delegatenklasse übergibt man als einzigen Parameter den Namen einer statischen Methode (nötigenfalls mit vorangestelltem Klassennamen) *oder* den Namen einer Instanzmethode (nötigenfalls mit vorangestellter Objektreferenz):

Delegatenkonstruktion

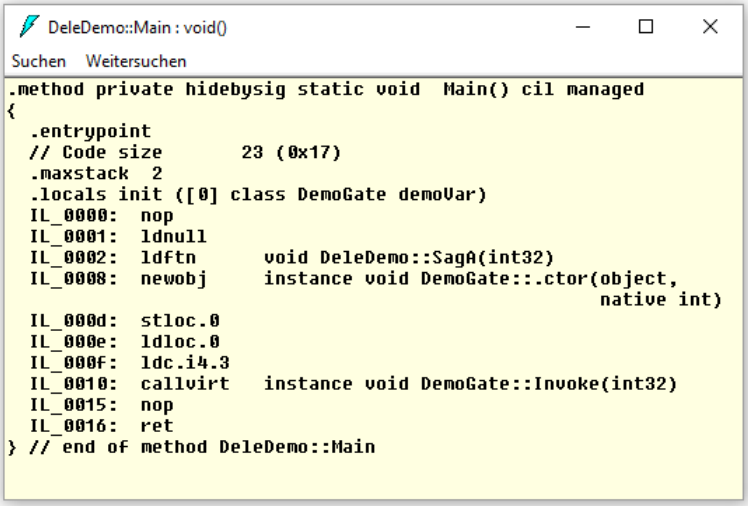


Dabei wird an den Methodennamen *keine* Parameterliste angehängt.

Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** zeigt, dass die Anweisungen

```
demoVar = new DemoGate(SagA);
demoVar(3);
```

in der `DeleDemo`-Methode **Main()**



folgendes bewirken

- Es wird ein `DemoGate`-Objekt erstellt (**newobj**).
- Dessen Adresse landet in der Variablen `demoVar` (**stloc.0**).
- Das Objekt wird beauftragt, die `DemoGate`-Methode **Invoke()** auszuführen (**callvirt**).

An Stelle der expliziten Delegatenobjektkreation

```
DemoGate demoVar = new DemoGate(SagA);
```

erlaubt der Compiler auch die Abkürzung:

```
DemoGate demoVar = SagA;
```

Eine Methode in die Aufrufliste eines Delegatenobjekts aufzunehmen und schlussendlich auf indirekte Weise aufzurufen, kann nicht der Sinn der Delegatentechnik sein. Viel praxisrelevanter ist die Möglichkeit, über einen Parameter mit Delegatentyp eine Funktionalität in eine Methode oder in eine Klasse zu „injizieren“. Ein typisches Beispiel ist die folgende Überladung der Methode **Sort()** aus der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic**:

public void Sort(Comparison<T> *comparison***)**

Sie erwartet als Parameter ein Delegatenobjekt, das den folgendermaßen definierten Typ **Comparison<T>** erfüllt:

public delegate int Comparison<in T>(T *x*, T *y*)

Beim **Sort()** - Aufruf übergibt man eine Methode, welche für zwei **T**-Instanzen die Anordnung definiert, und steuert so die Sortierung.

Durch die folgende Methode mit der Delegatensignatur von **Comparison<int>** werden **int**-Werte aufsteigend sortiert mit der Besonderheit, dass eine gerade Zahl jeder ungeraden Zahl unterlegen ist:

```
static int EvenLower(int first, int second) {
   if (first % 2 == 0)
      if (second % 2 == 0) return first.CompareTo(second);
      else return -1;
   else
      if (second % 2 == 0) return 1;
      else return first.CompareTo(second);
}
```

Wird die folgende Liste mit **int**-Werten

```
var intList = new List<int> {5, 4, 3, 2, 1, 6 };
```

unter Verwendung von `EvenLower()` sortiert,

```
intList.Sort(EvenLower);
```

dann führt die Kontrollausgabe

```
foreach (int i in intList)
   Console.WriteLine(i);
```

zum Ergebnis:

```
2 4 6 1 3 5
```

Gerade wurde per Delegatenobjekt eine Konfiguration bzw. Funktionserweiterung in eine Empfangsmethode übertragen. Oft steht bei der Delegatenübergabe ein kommunikativer Zweck im Vordergrund: Durch einen Aufruf eines Delegatenobjekts kann die Empfangsmethode über ein Ereignis informieren (z.B. über die Beendigung einer Aufgabe).

9.1.3 Delegatenobjekte kombinieren (Multicast)

Nach dem Motto „Wer *A* sagt, muss auch *B* sagen“ erweitern wir die Klasse `DeleDemo` aus Abschnitt 9.1.2 um die Methode `SagB()`:

```
static void SagB(int w){
   for (int i = 1; i <= w; i++)
      Console.Write('B');
   Console.WriteLine();
}
```

In der Methode **Main()** ergänzen wir die Anweisung:

```
demoVar += new DemoGate(SagB);
```

Es entsteht zunächst ein weiteres `DemoGate`-Objekt mit der statischen Methode `SagB()` in seiner einelementigen Aufrufliste. Dieses Objekt wird anschließend per „+=“ - Operator mit dem von `demoVar` referenzierten Objekt kombiniert, wobei ein weiteres `DemoGate`-Delegatenobjekt mit einer *zweielementigen* Aufrufliste entsteht, dessen Adresse schließlich in der Referenzvariablen `demoVar` landet. Die beiden Delegatenobjekte mit einelementiger Aufrufliste werden zu obsoletem Müll (vgl. ECMA 2006, S. 365). Delegatenobjekte sind ebenso unveränderlich wie z.B. die Objekte der Klasse **String** (vgl. Abschnitt 5.4.1.1).

Beim Aufruf des neuen Delegatenobjektes werden nacheinander *zwei* Methoden ausgeführt:

```
AAA
BBB
```

Über den „-=“ - Operator kann man ein Delegatenobjekt mit *verkürzter* Aufrufliste erzeugen, z.B.:

```
demoVar -= SagB;
```

Beim kompletten Entleeren der Aufrufliste entsteht aber kein leeres Delegatenobjekt, sondern die Referenzvariable erhält den Wert **null**.

Neben den Aktualisierungsoperatoren += und -= kann man auch den Additions- und den Subtraktionsoperator verwenden, z.B.:

```
demoVar = demoVar + new DemoGate(SagB);
```

Besitzt ein Delegatenobjekt eine *mehrelementige* Aufrufliste, spricht man von einem *Multicastdelegaten*. Bei einem Aufruf des Delegatenobjekts werden alle Methoden in seiner Aufrufliste nacheinander ausgeführt, wobei die zuletzt aufgerufene Methode beim Rückgabewert und bei eventuell vorhandenen **out**-Parametern dominiert.

9.1.4 Delegaten versus Schnittstellen

Die Injektion von Funktionalität in eine Klasse oder eine Methode kann oft alternativ über einen Delegatentyp oder durch eine Einmethoden-Schnittstelle erreicht werden. In der generischen Klasse **List<T>** im Namensraum **System.Collections.Generic** finden sich z.B. zwei Überladungen der Methode **Sort()**, die per Parameterobjekt ein frei konfigurierbares Sortierkriterium unterstützen. Durch das Parameterobjekt wird letztlich eine Methode beigesteuert, die für zwei Instanzen vom Typ **T** per **int**-Rückgabe die Anordnung festlegt. Die beiden Überladungen sehen sehr ähnlich aus:

- **public void Sort(Comparison<T>** *comparison***)**
  Diese Überladung erwartet als Parameter ein Objekt vom generischen Delegatentyp **Comparison<T>**:
  public delegate int Comparison<in T>(T *x***, T** *y***)**
  Der Typformalparameter **T** ist über das Schlüsselwort **in** als kontravariant definiert (siehe Abschnitt 9.1.6). Ein Objekt vom Typ **Comparison<T>** zeigt auf eine Methode, welche zwei Argumente vom Typ **T** besitzt und eine Rückgabe vom Typ **int** liefert. Wird **T** durch **String** konkretisiert, taugt z.B. die folgende Methode:
  ```
  int StringComp(String x, String y) { ... }
  ```
- **public void Sort(IComparer<T>** *comparer***)**
  Diese Überladung erwartet als Parameter ein Objekt aus einer Klasse, welche die Schnittstelle **IComparer<T>** implementiert, also bei Konkretisierung von **T** durch **String** die folgende Instanzmethode besitzt:
  public int Compare(String *x***, String** *y***) { ... }**

Argumente für ein Design mit Delegatentyp:

- Eine Klasse kann *mehrere* Methoden enthalten, die *denselben* Delegatentyp erfüllen, weil der Methodenname frei wählbar ist. Demgegenüber kann eine Klasse eine Schnittstellenmethode nur einmal implementieren, weil der Name fest vorgegeben ist.
- Ein Delegatenobjekt lässt sich auch auf Basis einer statischen Methode erstellen, während eine Schnittstelle grundsätzlich Instanzmethoden verlangt.
- Über die in Abschnitt 9.1.5 beschriebenen anonymen Methoden lässt sich ein Delegatenobjekt syntaktisch elegant realisieren, während zum Implementieren einer Schnittstelle eine explizite Typdefinition erforderlich ist.

Argumente für ein Design mit Schnittstelle:

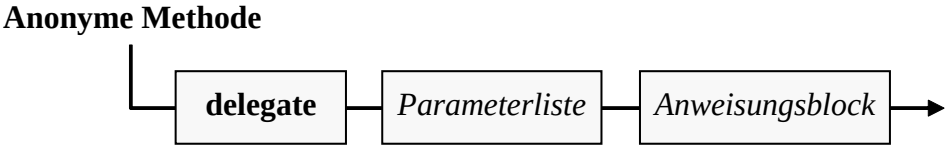
- Während ein Delegatentyp nur *eine* Methode vorschreibt, kann eine Schnittstelle den implementierenden Klassen beliebig viele Methoden diktieren, so dass deren Objekte über mehrere garantierte Kompetenzen verfügen.
- Eine Klasse, die eine Schnittstelle implementiert, kann mit Instanzvariablen zur Verwaltung von Zustandsinformationen ausgestattet werden.

9.1.5 Anonyme Methoden

Über eine anonyme Methode lässt sich ein Delegatenobjekt ohne explizite Methodendefinition erstellen.

9.1.5.1 Traditionelle Syntax

Statt beim Erzeugen eines Delegatenobjekts den Namen einer vorhandenen, andernorts definierten Methode zu übergeben, kann man seit C# 2.0 auch einen Anweisungsblock setzen, dem das Schlüsselwort **delegate** samt Delegatentyp-konformer Parameterliste vorangeht.



Dies wird in der folgenden Variante des Beispiels aus Abschnitt 9.1.2 demonstriert:

Quellcode

```
using System;

delegate void DemoGate(int w);

class AnoMeth {
   static void Main() {
      DemoGate demoVar =
         delegate (int w) {
            for (int i = 1; i <= w; i++)
               Console.Write('A');
            Console.WriteLine();
         };
      demoVar(3);
   }
}
```

Ausgabe

```
AAA
```

Man kann die gewünschte Funktionalität vor Ort realisieren, muss keine Kreativität in einen Methodennamen investieren. Darüber hinaus besitzen anonyme Methoden die außerordentlich wichtige Fähigkeit, Variablen aus ihrem Entstehungskontext zu verwenden (siehe Abschnitt 9.1.5.2).

Bei entsprechender Definition des zu erfüllenden Delegatentyps können (und müssen) anonyme Methoden selbstverständlich auch einen Rückgabewert liefern, z.B.:

Quellcode	Ausgabe
`using System;` `delegate int DemoGate(int w);` `class AnoMeth {` `  static void Main() {` `  DemoGate demoVar = delegate (int w) {return w;};` `  Console.WriteLine(demoVar(3));` `  }` `}`	`3`

Ein Nachteil anonymer Methoden besteht darin, dass sie nicht an anderen Stellen benutzt werden können.

9.1.5.2 Zugriff auf Kontextvariablen

Eine anonyme Methode darf auf lokale Variablen der umgebenden Methode sowie auf (statische) Felder der Umgebung zugreifen. Im folgenden Beispiel wird die lokale Variable `c` der Methode **Main()** verwendet:

Quellcode	Ausgabe
`using System;` `delegate void DemoGate(int w);` `class AnoMeth {` `  static void Main() {` `  char c = 'A';` `  DemoGate demoVar =` `  delegate (int w) {` `  for (int i = 1; i <= w; i++)` `  Console.Write(c);` `  Console.WriteLine();` `  c++;` `  };` `  Console.WriteLine("c vor dem Delegatenaufr.:  " + c);` `  demoVar(3);` `  Console.WriteLine("c nach dem Delegatenaufr.: " + c);` `  }` `}`	`c vor dem Delegatenaufr.:  A` `AAA` `c nach dem Delegatenaufr.: B`

Wie das Beispiel demonstriert, kann in der anonymen Methode eine Variable aus der Umgebung auch *verändert* werden. Damit haben zwei Methoden (die umgebende und die anonyme Methode) Lese- und Schreibzugriff auf dieselbe Variable, was speziell im Zusammenhang mit nebenläufiger Programmierung besondere Sorgfalt erfordert (siehe Kapitel 15).

Die Zusammenfassung einer anonymen Methode mit den „eingefangenen“ Variablen aus der Umgebung wird als *Abschluss* (engl. *closure*) bezeichnet.

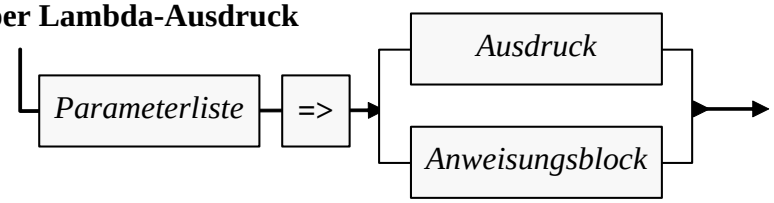
Oft ist die erzeugende Methode schon beendet, wenn ein Delegatenobjekt aufgerufen wird. Damit der Delegatenaufruf auch dann noch klappt, wenn die generierende Methode beendet ist, und ihre lokalen Variablen längst vom Stack verschwunden sind, erzeugt der Compiler zur Aufbewahrung der eingefangenen Variablen ein Objekt auf dem Heap (siehe Griffiths 2013, S. 316ff).

9.1.5.3 Lambda-Ausdrücke

In C# 3.0 wurde zur Unterstützung der LINQ-Technik mit den so genannten *Lambda-Ausdrücken* eine zu den anonymen Methoden funktional äquivalente und dabei kompaktere Syntax eingeführt. In C# 6.0 wurde zusätzlich die Methodendefinition mit Rumpf im Lambda-Stil ermöglicht (siehe Abschnitt 4.7.3).

Zur Definition einer anonymen Methode per Lambda-Ausdruck wird die folgende Syntax verwendet:

Anonyme Methode per Lambda-Ausdruck



Hier sind für einen einfachen Delegatentyp

```
delegate int Rest(int a, int b);
```

die äquivalenten Realisationen über eine anonyme Methode

```
Rest R = delegate(int a, int b) {
  return Math.Max(a, b) % Math.Min(a, b);
};
```

und einen Lambda-Ausdruck zu sehen:

```
Rest R = (int a, int b) => Math.Max(a, b) % Math.Min(a, b);
```

Bei der Parameterliste eines Lambda-Ausdrucks besteht einige Flexibilität:

- Man kann auf die Angabe der Parameter*typen* verzichten, weil sich diese aus dem zu erfüllenden Delegatentyp zwingend ergeben, z.B.:

  ```
  Rest R = (a, b) => Math.Max(a, b) % Math.Min(a, b);
  ```

- Bei einem einzelnen, implizit typisierten Parameter kann man die runden Klammern weglassen, z.B.:

  ```
  public delegate bool Predicate<in T>(T obj);
   . . .
  Predicate<int> even = i => i % 2 == 0;
  ```

  Der generische Delegatentyp **Predicate<in T>** mit einem kontravarianten Typformalparameter **T** (siehe Abschnitt 9.1.6) und einer booleschen Rückgabe ist im Namensraum **System** der FCL definiert.

Ist der Lambda-Rumpf ein Anweisungsblock und der Rückgabetyp des zu erfüllenden Delegatentyps von **void** verschieden, dann muss der Rückgabewert per **return**-Anweisung geliefert werden, z.B.:

```
Predicate<String> evenLen = s => {
  int sl = s.Length;
  return sl % 2 == 0 ? true : false;
```

```
   };
```

Im Beispiel aus Abschnitt 9.1.5.1 ergibt sich im Vergleich zur traditionellen Syntax für anonyme Methoden kein großer Einspareffekt:

Quellcode	Ausgabe
`using System;` `delegate void DemoGate(int w);` `class AnoMethLambda {` `  static char c = 'A';` `  static void Main() {` `  DemoGate demoVar = w => {` `  for (int i = 1; i <= w; i++)` `  Console.Write(c);` `  Console.WriteLine();` `  };` `  demoVar(3);` `  }` `}`	`AAA`

Man spart das Schlüsselwort **delegate** und kann die Fähigkeiten des Compilers zur Typinferenz nutzen.

Nebenbei wird im Beispiel demonstriert, dass eine anonyme Methode (hier mit Lambda-Syntax) auf ein statisches Feld der umgebenden Klasse zugreifen kann.

9.1.6 Generische Delegaten, Ko- und Kontravarianz

Neben generischen Klassen, Strukturen, Schnittstellen und Methoden (siehe Kapitel 7) unterstützt C# auch generische Delegaten. Der Typ **Comparison<T>** aus der FCL

```
public delegate int Comparison<in T>(T x, T y);
```

wurde oben bereits vorgestellt (siehe z.B. Anschnitt 9.1.4). Er kommt in einer **Sort()** - Überladung der generischen Kollektionsklasse **List<T>** (vgl. Abschnitt 7.1) als Parameterdatentyp zum Einsatz. Über ein **Comparison<String>** - Objekt, das auf eine geeignet konstruierte **String**-Vergleichsmethode zeigt, wird im folgenden Beispiel dafür gesorgt, dass in einer sortierten Namensliste „Anton“ stets der Größte ist:

Quellcode	Ausgabe
`using System;` `using System.Collections.Generic;` `class Prog {` `  static int Compar(String a, String b) {` `  if (a.Equals("Anton"))` `  return 1;` `  else` `  if (b.Equals("Anton"))` `  return -1;` `  else` `  return a.CompareTo(b);` `  }` `  static void Main() {` `  var li = new List<String>();` `  li.Add("Fritz"); li.Add("Anton");` `  li.Add("Anita"); li.Add("Theo");` `  li.Sort(Compar);` `  foreach (String s in li)` `  Console.WriteLine(s);` `  }` `}`	`Anita` `Fritz` `Theo` `Anton`

Die restlichen Zeilen des aktuellen Abschnitts sind scheinbar eher für Begriffs-Jongleure als für Programmierer geeignet, doch ist eine Praxisrelevanz nicht ganz ausgeschlossen. Bei generischen Delegaten kann wie bei generischen Schnittstellen für einen Typformalparameter Ko- oder Kontravarianz vereinbart werden. Weil z.B. im Delegatentyp **Comparison<T>** der Typformalparameter über das Schlüsselwort **in** als kontravariant definiert ist, darf einer Delegatenvariablen vom Typ **Comparison<String>** ein Delegatenobjekt vom Typ **Comparison<Object>** zugewiesen werden:

```
Comparison<Object> compObj = (o1, o2) => 1;
Comparison<String> compStr = compObj;
```

Im folgenden Beispiel wird der Typformalparameter eines Delegaten über das Schlüsselwort **out** als kovariant vereinbart:

```
public delegate T GenDelegateCo<out T>();
```

Folglich darf z.B. einer Delegatenvariablen vom Typ `GenDelegateCo<Object>` ein Delegatenobjekt vom Typ `GenDelegateCo<String>` zugewiesen werden:

```
GenDelegateCo<String> genDelCoStr = () => null;
GenDelegateCo<Object> genDelCoObj = genDelCoStr;
```

Eine als Ko- bzw. Kontravarianz zu beschreibende Zuweisungsliberalität besteht auch bei *nichtgenerischen* Delegatentypen (und das schon seit .NET 2.0). Einer Variablen vom Typ

```
public delegate Object NonGenericDelegate(String arg);
```

darf z.B. die folgende Methode mit einem allgemeineren Argumenttyp und einem spezielleren Rückgabetyp

```
static String AORS(Object arg) { return null; }
```

zugewiesen werden:

```
NonGenericDelegate nGenDel = AORS;
```

9.2 Ereignisse

Möchte eine .NET - Klasse anderen Programmeinheiten die Möglichkeit geben, zu besonderen Gelegenheiten eine Botschaft, d.h. einen bestimmten Methodenaufruf, zu erhalten, dann bietet sie ein *Ereignis* (engl.: *event*) an.[1]

Wir lernen hier ein neues Klassenmitglied kennen, wobei es sich um eine Delegatenvariable mit einigen Besonderheiten handelt. Andere Klassen können ein Delegatenobjekt bei einem Ereignis registrieren lassen, so dass eine ein- oder mehrelementige Aufrufliste entsteht. Wie andere Klassen-Member kann auch ein Ereignis Instanz- oder Klassenbezogen sein.

9.2.1 Innenarchitektur von Ereignissen

Ereignisse stellen ein wichtiges Kommunikationsmittel dar. Sie ermöglichen es einer Klasse oder einem Objekt, andere Akteure darüber zu informieren, dass etwas Bestimmtes passiert ist. So kann z.B. ein Befehlsschalter registrierte Interessenten darüber informieren, dass er vom Benutzer betätigt (angeklickt) worden ist. Das zugehörige Instanzereignis der Klasse **Button** im Namensraum **System.Windows.Controls** heißt **Click**.

Obwohl Ereignisse in der Regel als **public** deklariert werden (siehe Abschnitt 9.2.3), resultiert in der veröffentlichenden Klasse stets eine *private* Delegatenvariable. Der Compiler ergänzt jedoch *öffentliche* Zugriffsmethoden zum Erweitern und Verkürzen der Aufrufliste durch fremde Klassen. Diese sind über die Aktualisierungsoperatoren mit dem Ereignisnamen als linkem Argument zu verwenden:

- `+=` nimmt eine Behandlungsmethode in die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts auf
- `-=` entfernt eine Behandlungsmethode aus der Aufrufliste des zum Ereignis gehörigen Delegatenobjekts

Das öffentlich zugänglich Ereignis, das eigentlich ein Paar von Zugriffsmethoden ist, und die privaten Delegatenvariable stehen zueinander in derselben Beziehung wie eine Eigenschaft und eine zugrunde liegende Instanzvariable (vgl. Abschnitt 4.5). Dementsprechend ähnelt die Syntax zur Definition eines Ereignisses ähnelt stark einer Eigenschaftsdefinition, wobei die Schlüsselwörter **get** und **set** zu ersetzen sind durch **add** und **remove**. Wir betrachten als Beispiel das von der WPF-Klasse **Application** definierte Ereignis **Exit**, das anderen Klassen die Möglichkeit bietet, sich über das bevorstehende (und unabwendbare) Programmende informieren zu lassen:[2]

---

[1] Grundsätzlich können auch *Strukturen* Ereignisse anbieten, was aber in der Praxis so gut wie nie geschieht und außerdem durch Komplikationen mit der Thread-Sicherheit belastet ist (siehe http://csharpindepth.com/Articles/Chapter2/Events.aspx).

[2] Der Quellcode stammt der Datei **Application.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann. **Events** ist eine private Eigenschaft der Klasse **Application**, die im Lesezugriff ein Objekt der Klasse **EventHandlerList** liefert. Dieses Kollektionsobjekt verwaltet eine Liste von (**Object**, **Delegate**) - Paaren und beherrscht dazu die Methoden **AddHandler()** und **RemoveHandler()**, welche die Aufrufliste zu dem im ersten Parameter angegebenen Delegatenobjekt erweitern bzw. reduzieren. Die für das Feuern des **Exit**-Ereignisses zuständige **Application**-Methode **OnExit()** holt sich das zum Ereignis gehörige Delegatenobjekt aus der Kollektion und ruft dann im Sinne von Abschnitt 9.1.2 das Delegatenobjekt auf:

```
protected virtual void OnExit(ExitEventArgs e) {
  VerifyAccess();
  ExitEventHandler handler = (ExitEventHandler)Events[EVENT_EXIT];
  if (handler != null) {
     handler(this, e);
  }
}
```

```
public event ExitEventHandler Exit {
  add {
    VerifyAccess();
    Events.AddHandler(EVENT_EXIT, value);
  }
  remove {
    VerifyAccess();
    Events.RemoveHandler(EVENT_EXIT, value);
  }
}
```

Analog zu den automatisch implementierten Eigenschaften (siehe Abschnitt 4.5.2) kann man auch bei der Ereignisdefinition etliche Routinearbeit dem Compiler überlassen, wie das Ereignis **Activated** aus der Klasse **Application** zeigt:

```
public event EventHandler Activated;
```

Diese Ereignisdefinition sieht aus wie die Deklaration einer Delegatenvariablen, wobei zusätzlich das Schlüsselwort **event** anzugeben ist. Allerdings ist es mit der Deklaration allein nicht getan, weil ein sinnvolles Ereignis auch irgendwann ausgelöst werden muss. Dies geschieht meist in einer Methode, deren Namen mit *On* startet und dann den Ereignisnamen übernimmt, z.B.:

```
protected virtual void OnActivated(EventArgs e) {
  VerifyAccess();
  if (Activated != null) {
    Activated(this, e);
  }
}
```

Weil die zu einem Ereignis gehörige Delegatenvariable *privat* ist, können auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen. Über die gerade für das **Application**-Ereignis **Activated** vorgestellte virtuelle (also überschreibbare) Methode mit Zugriffsschutz **protected** können abgeleitete Klassen jedoch die volle Kontrolle über die Ereignisentstehung übernehmen.

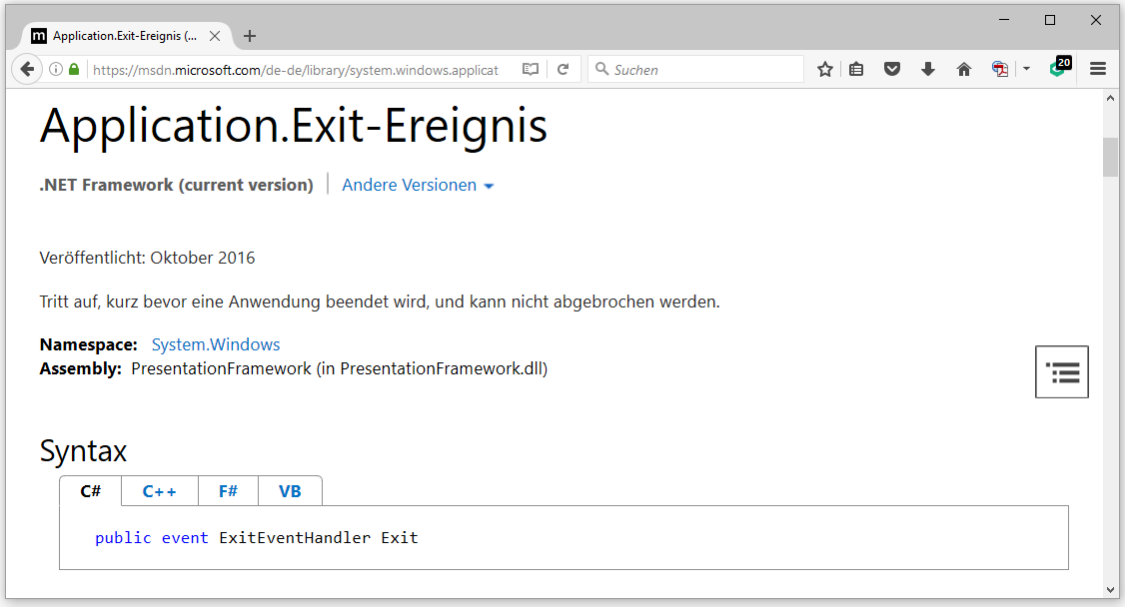
9.2.2 Behandlungsmethoden registrieren

Um auf ein Ereignis einer .NET - Klasse reagieren zu können, …

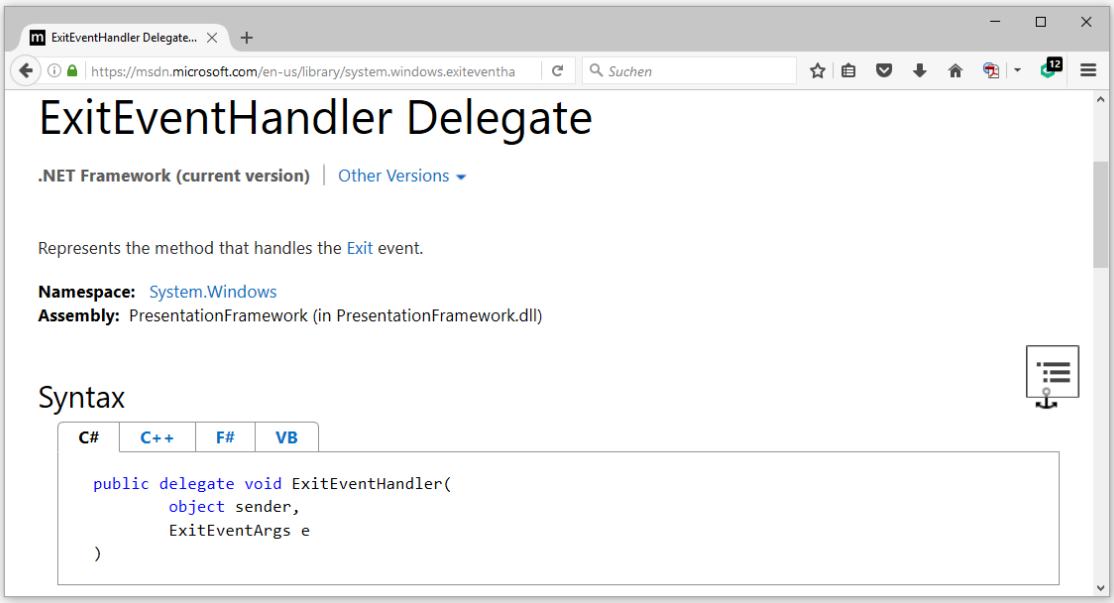
- implementiert man eine Methode, die mit dem Delegatentyp des Ereignisses kompatibel ist (siehe Abschnitt 9.1.6 zur Zuweisungskompatibilität bei Delegaten),
- erzeugt man ein Delegatenobjekt, das auf diese Methode zeigt,
- weist man dem Ereignis dieses Delegatenobjekt per „+=“ - Operator zu.
  In Abschnitt 9.1.3 wird beschrieben, wie bei einer solchen Zuweisung die Aufrufliste des zum Ereignis gehörigen Delegatenobjekts erweitert wird.

Im folgenden Beispiel kümmern wir uns um das Ereignis **Exit**, das anlässlich der unmittelbar bevorstehenden Beendigung einer WPF-Anwendung auftritt. Es wird von einem die Anwendung repräsentierenden Objekt ausgelöst, das zur Klasse **Application** im Namensraum **System.Windows** oder zu einer abgeleiteten Klasse gehört (siehe Abschnitt 11.2.3).

Zu welchem Delegatentyp eine Ereignisbehandlungsmethode kompatibel sein muss, erfährt man in der FCL-Dokumentation. Beim Ereignis **Exit** der Klasse **Application** handelt es sich um den Typ **ExitEventHandler**:



In der Dokumentation zum Delegatentyp **ExitEventHandler** ist zu erfahren, welchen Rückgabetyp und welche Parameterliste eine kompatible Methode benötigt, z.B.:



Die **ExitEventHandler**-Signatur verlangt von kompatiblen Behandlungsmethoden zwei Parameter:

- **Object**-Parameter **sender**
  Die aufgerufenen Behandlungsmethoden erhalten über diesen Parameter eine Referenz zur Ereignisquelle. Man kann eine Behandlungsmethode bei *mehreren* Ereignisquellen anmelden, z.B. bei mehreren Befehlsschaltern. Weil der Methode beim Aufruf die Ereignisquelle im Parameter **sender** mitgeteilt wird, kann sie situationsgerecht reagieren.
- **ExitEventArgs**-Parameter **e**
  Behandlungsmethoden erhalten im Allgemeinen über den zweiten Parameter nähere Informationen zum Ereignis. Dabei wird ein Objekt aus der Klasse **System.EventArgs** oder aus einer abgeleiteten Klasse verwendet.

Im frei wählbaren Namen einer Behandlungsmethode nennt man in der Regel die Ereignisquelle (z.B. die Klasse **Application**) und das Ereignis (im Beispiel: **Exit**). Das Visual Studio setzt hinter den Namen der Quelle einen Unterstrich.

Wie Sie bereits aus eigener Erfahrung wissen (vgl. z.B. Abschnitt 4.11.8), ist bei der praktischen Arbeit mit dem Visual Studio das Erstellen und Registrieren einer Ereignishandlungsmethode eine einfache Angelegenheit. Um ein rudimentäres Beispielprogramm samt **Exit**-Ereignisbehandlung semiautomatisch zu erstellen, kann man so vorgehen:

- Man erstellt eine neue **WPF-Anwendung**.
- Im WPF-Designer versorgt man die Hauptfenster-Eigenschaften **Titel**, **Width** und **Height** mit geeigneten Werten.
- Man öffnet die Datei **App.xaml** per Doppelklick auf ihren Eintrag im Projektmappen-Explorer. Sie dient zur Konfiguration der Klasse `App`, die das Visual Studio als **Application**-Ableitung automatisch erstellt hat. U.a. kann man über die Datei **App.xaml** Behandlungsmethoden zu den Ereignissen der Klasse `App` registrieren lassen.
- Man wählt Eigenschaftenfenster per Mausklick auf das Symbol  die Registerkarte mit den Ereignissen der Klasse `App` und setzt einen Doppelklick auf die Textbox zum Ereignis **Exit**.
- Daraufhin wird in der Datei **App.xaml.cs** mit der vom Entwickler zu verantwortenden partiellen `App`-Klassendefinition eine Ereignisbehandlungsmethode angelegt. Wir komplettieren sie durch einen Aufruf der statischen Methode **Show()** der Klasse **MessageBox**, z.B.:

```
private void Application_Exit(object sender, ExitEventArgs e) {
   MessageBox.Show("Vielen Dank für den Einsatz dieser Software!",
                   "Application Exit");
}
```

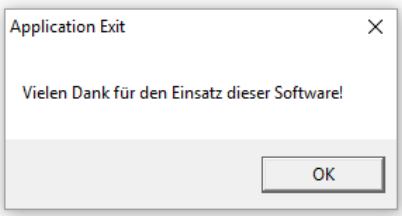
Außerdem wird in der Datei **App.g.i.cs** mit der vom Visual Studio gepflegten partiellen `App`-Klassendefinition (vgl. Abschnitt 11.3.4 zur Erläuterung und Lokalisation der Datei) die Ereignisregistrierung vorgenommen. In der Methode **Main()** wird ein Objekt der Klasse `App` mit dem Namen `app` erzeugt und mit der Ausführung der Methode **InitializeComponent()** beauftragt:

```
public static void Main() {
    ApplicationExit.App app = new ApplicationExit.App();
    app.InitializeComponent();
    app.Run();
}
```

In **InitializeComponent()** nimmt dieses Objekt die Instanzmethode `Application_Exit` in die Aufrufliste zum Ereignis `Exit` auf:

```
this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);
```

Im Beispiel kommt die Methode `Application_Exit` z.B. zum Einsatz, wenn der Benutzer auf das Schließkreuz in der Titelzeile des Anwendungsfensters klickt:



Auf eine unbedingt zu vermeidende Speicherverschwendung im Zusammenhang mit dem Registrierung von Ereignisempfängern in GUI-Programmen wird bei Griffiths (2014, S. 330ff) hingewiesen:

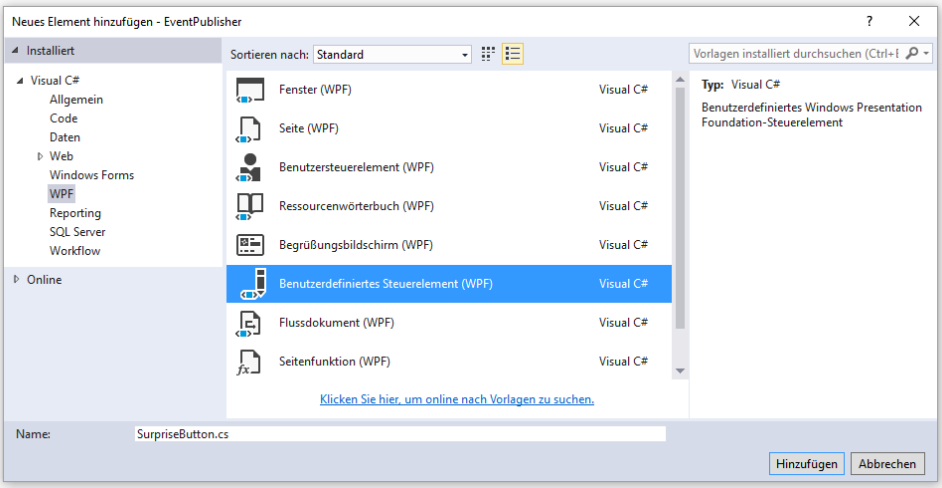
- Ein Programm enthält mehrere Fenster.
- Ein Fenster enthält ein als Ereignisempfänger registriertes Steuerelement.
- Der Programmierer versäumt es, beim Schließen des Fensters die Ereignisregistrierung aufzuheben.

- Solange die Ereignisquelle vorhanden ist, bleiben die zum geschlossenen Fenster gehörigen Objekte erreichbar, können also nicht vom Garbage Collector abgeräumt werden.

9.2.3 Ereignisse anbieten

Zwar kommt das Registrieren eigener Behandlungsmethoden bei Ereignissen von FCL-Klassen häufiger vor als das Veröffentlichen von Ereignissen in einer eigenen Klassendefinition, doch müssen wir auch die Rolle eines Ereignisemittenden beherrschen. Weil Ereignisse meist im Kontext mit grafischen Bedienoberflächen von Interesse sind, soll auch das Beispiel aus diesem Bereich gewählt werden. Statt ein lebensfernes Konsolenbeispiel zu konstruieren, nehmen wir lieber weitere Vorgriffe auf die WPF-Technik in Kauf.

Verwenden Sie im Visual Studio für ein neues Projekt mit dem Namen `EventPublisher` die Vorlage **WPF-Anwendung**. Wir leiten aus der Klasse **Button** eine benutzerdefinierte Steuerelementklasse ab, deren Objekte bei jedem Mausklick im Hintergrund eine Zufallszahl ziehen und das Ereignis `Seven` auslösen, wenn die Zufallszahl restfrei durch 7 teilbar ist. Die nicht ganz triviale Ausgabe, ein benutzerdefiniertes Steuerelement zu erstellen, ist im Visual Studio schnell erledigt. Öffnen Sie im **Projektmappen-Explorer** per Maus-Rechtsklick das Kontextmenü zum *Projekt* (nicht zur *Projektmappe*), und fügen Sie ein **neues Element** hinzu. Wählen Sie im Dialog für **neue Elemente** aus der Abteilung **Visual C# > WPF** die Option **Benutzersteuerdefiniertes Steuerelement (WPF)**, und tragen Sie **SurpriseButton.cs** als **Namen** ein:



Nach dem **Hinzufügen** wird die Datei im Quellcode-Editor der Entwicklungsumgebung geöffnet.

Tragen Sie **Button** (statt **Control**) als Basisklasse für `SurpriseButton` ein.

Der vom Assistenten erstellte statische Konstruktor

```
static SurpriseButton() {
    DefaultStyleKeyProperty.OverrideMetadata(typeof(SurpriseButton),
    new FrameworkPropertyMetadata(typeof(SurpriseButton)));
}
```

ist dafür verantwortlich, dass im Anwendungsfenster von unserer **Button**-Ableitung nichts zu sehen ist. Grundsätzlich ist die Anweisung im statischen Konstruktor sinnvoll und notwendig, wenn sich ein benutzerdefiniertes Steuerelement stilistisch von der Basisklasse unterscheidet, wobei offenbar weitere Bedingungen für einen guten Auftritt zu erfüllen sind. Weil das Erscheinungsbild unserer **Button**-Ableitung aber vollständig durch die Designvorlage der Basisklasse definiert ist, können wir auf die kritische Anweisung verzichten. Daher streichen wir den statischen Konstruktor mit dem Zwischenstand für die Klasse `SurpriseButton`:

```
public class SurpriseButton : Button {
```

```
}
```

Ergänzen Sie (z.B. in der Quellcodedatei **SurpriseButton.cs**) eine von **EventArgs** abstammende Klasse, die zur Übertragung von Informationen an Ereignisempfänger dient:

```
public class SevenEventArgs : EventArgs {
  public int Nr;
}
```

Die von einem Empfänger für das Ereignis `Seven` zu implementierende Methode muss die folgende Delegatensignatur besitzen:

```
public delegate void SevenEventHandler(object sender, SevenEventArgs e);
```

Auch dieser Delegatentyp muss (z.B. in der Quellcodedatei **SurpriseButton.cs**) definiert werden.

Üblicherweise endet der Name eines für Ereignisempfänger zuständigen Delegatentyps mit *EventHandler*. Außerdem besitzt ein solcher Delegatentyp zwei Parameter:

- **Object** *sender*
  Im ersten Parameter identifiziert sich die Ereignisquelle.
- **EventArgs** *e*
  Der zweite Parameter besitzt einen von **EventArgs** abstammenden Typ und liefert spezifische Informationen über das Ereignis.

Unser Schalter mit Überraschungseffekt bietet ein Ereignis namens `Seven` mit dem Zugriffsschutz **public** an. Weil wir die Standardimplementation der Methoden zum Erweitern und Reduzieren der Aufrufliste (siehe Abschnitt 9.2.1) nicht ändern wollen, sieht die Ereignisdefinition in der Klasse `SurpriseButton` fast so aus wie eine Felddeklaration, wobei aber das Schlüsselwort **event** anzugeben ist:

```
public event SevenEventHandler Seven;
```

In der Klasse `SurpriseButton` überschreiben wir die geerbte Methode **OnClick()**, die bei jedem Mausklick aufgerufen wird:

```
protected override void OnClick() {
  base.OnClick();
  if (Seven != null) {
    int losnummer = zzg.Next(10000);
    if (losnummer % 7 == 0) {
      SevenEventArgs sea = new SevenEventArgs();
      sea.Nr = losnummer;
      Seven(this, sea);
    }
  }
}
```

Für das in **OnClick()** als Zufallszahlengenerator verwendete Instanzobjekt aus der Klasse **Random** wird noch eine Deklaration mit Initialisierung nachgetragen:

```
Random zzg = new Random();
```

Unser benutzerdefinierter Schalter reagiert auf einen Mausklick zunächst mit dem Basisklassenverhalten.[1] Ist ein `Seven`-Ereignisempfänger registriert, wird außerdem ...

[1] Warum die Methode **OnClick()** bei einem Mausklick auf den Befehlsschalter aufgerufen wird, erfahren Sie später.

- eine **int**-wertige Zufallszahl gezogen,
- und bei Teilbarkeit durch 7 das Ereignis `Seven` ausgelöst.

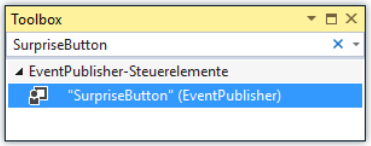
Jede registrierte Ereignisbehandlungsmethode wird über den Absender informiert und mit einem Objekt vom Typ `SevenEventArgs` versorgt, das in der Instanzvariablen `Nr` die gezogene Losnummer bereithält.

Man darf ein Ereignis nur dann auslösen, wenn tatsächlich ein Delegatenobjekt vorhanden ist.[1] Das Delegatenobjekt zu einem Ereignis entsteht beim Registrieren der ersten Behandlungsmethode und verschwindet beim Entleeren seiner Aufrufliste.

Lassen Sie das Projekt neu erstellen mit der Tastenkombination **Strg+Umschalt+B** oder mit dem Menübefehl:

Erstellen > Projektmappe erstellen

Wechseln Sie zum WPF-Designer, und suchen Sie im Toolbox-Fenster nach dem von uns erstellten Steuerelement `SurpriseButton`:



Setzen Sie einen Überraschungsschalter auf das Anwendungsfester, und gestalten Sie nach Bedarf die Bedienoberfläche, z.B.:

Die Hauptfensterklasse `MainWindow` besitzt nun eine Schaltfläche aus der Klasse `Suprise-Button`, wie eine Inspektion der Datei **MainWindow.xaml** zeigt:[2]

```
<Window x:Class="EventPublisher.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Event Publisher" Height="250" Width="500">
    <Grid>
       <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
        Margin="30" FontSize="36"/>
    </Grid>
</Window>
```

Um über ein eingetretenes `Seven`-Ereignis informiert zu werden, muss die Klasse `MainWindow` ...

- eine Methode mit der Delegatensignatur `SevenEventHandler` implementieren
- und diese Methode beim Ereignis `Seven` des `SupriseButton`-Steuerelements registrieren, was z.B. im `MainWindow`-Konstruktor geschehen kann.

---

1 Bei einer Multi Thread - Anwendung (siehe Abschnitt 15) sollte man sogar durch eine geeignete Synchronisation verhindern, dass ein Delegatenobjekt zwischen Existenzprüfung und Aufruf durch einen anderen Thread verworfen wird, z.B. mit der Anweisung:

```
Seven = null;
```

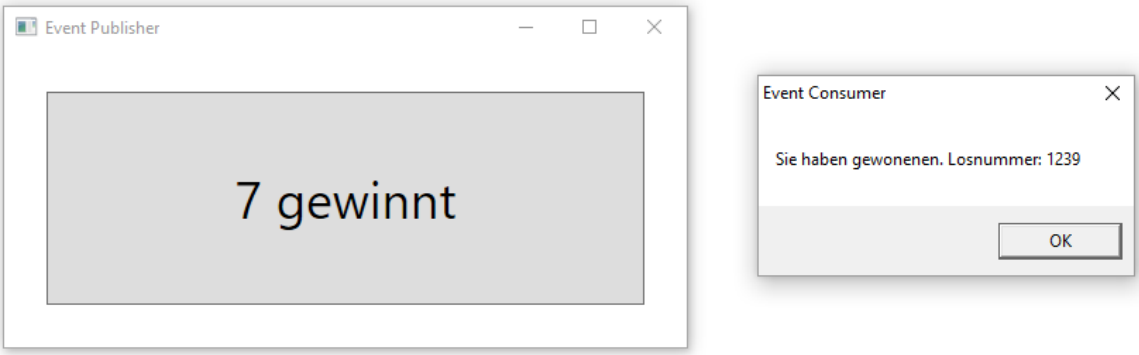
2 Einige irrelevante Zeilen wurden gestrichen (vgl. Abschnitt 11.3.2.1).

Hier ist der vom Entwickler zu verantwortende Quellcode der Klasse `MainWindow` zu sehen:

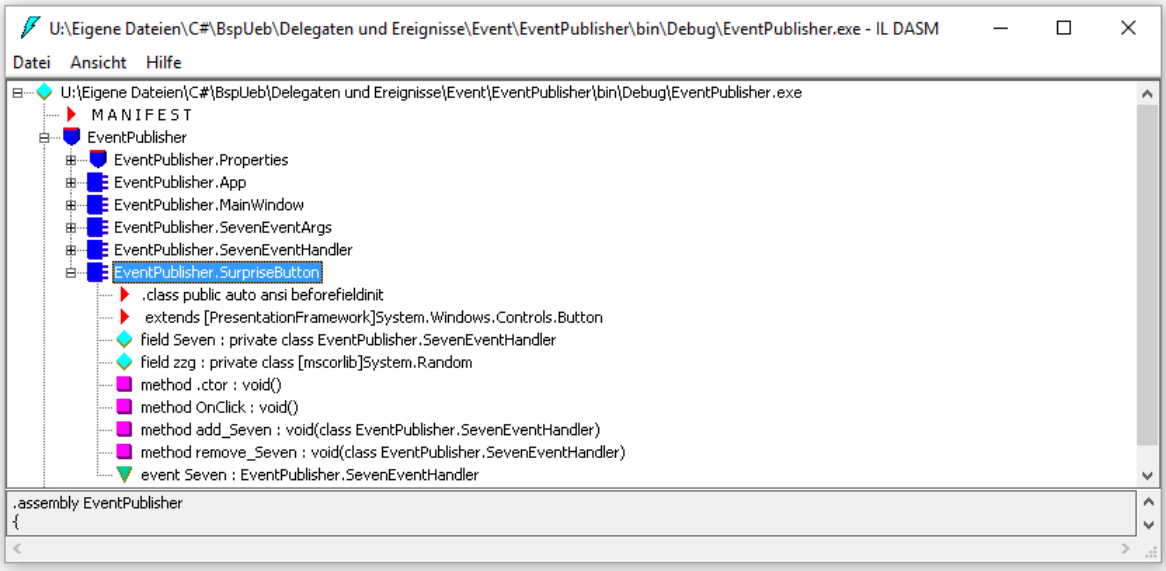
```
public partial class MainWindow : Window {
   public MainWindow() {
      InitializeComponent();
      surpriseButton.Seven += surpriseButton_Seven;
   }

   void surpriseButton_Seven(object sender, SevenEventArgs e) {
      MessageBox.Show("Sie haben gewonenen. Losnummer: " + e.Nr, "Event Consumer"); }
}
```

Beim Namen der Ereignisbehandlungsmethode passen wir uns dem Stil der Entwicklungsumgebung an und lassen auf den Namen der Ereignisquelle einen Unterstrich sowie den Ereignisnamen folgen.

Bei seiner Reaktion auf die Benachrichtigung wertet der Konsument auch die Ereignisbeschreibung im `SevenEventArgs`-Objekt aus, das der Methode `surpriseButton_Seven()` beim Aufruf übergeben wird, z.B.:



Eine Assembly-Inspektion mit dem Windows-SDK - Hilfsprogramm **ILDasm** bestätigt die zu Beginn von Abschnitt 9.2.1 formulierte Bemerkung über die Innenarchitektur von Ereignissen. In der Klasse `SurpriseButton` befinden sich aufgrund der `Seven`-Ereignisdefinition neben einem privaten Feld vom Typ `SevenEventHandler` zwei öffentliche Methoden zur Veränderung der Ereignisaufrufliste (`add_Seven()` und `remove_Seven()`):



Wie zu Beginn von Abschnitt 9.2.1 beschrieben, hat auch bei einem Ereignis mit Schutzstufe **public** die beim Auslösen zu verwendende Delegatenvariable stets die Schutzstufe **private**. Damit

können auch abgeleitete Klassen das Ereignis *nicht* über die Delegatenvariable auslösen, was in vielen Situationen durchaus erwünscht ist. Das übliche Verfahren zur Berechtigung der abgeleiteten Klassen besteht darin, ...

- das Ereignis grundsätzlich in einer Methode mit der Schutzstufe **protected** auszulösen, Damit haben abgeleitete Klassen die Möglichkeit, das Ereignis auszulösen.
- und die auslösende Methode als virtuell (überschreibbar) zu definieren.
  Damit kann sich eine abgeleitete Klasse durch Überschreiben der Auslösungsmethode in die Ereignisverarbeitung einschalten und z.B. das Auslösen verhindern.

Um diese Techniken zu demonstrieren, nehmen wir entsprechende Änderungen an der Klasse `SurpriseButton` vor:

```
public class SurpriseButton : Button {
   internal event SevenEventHandler Seven;
   Random zzg = new Random();

   protected override void OnClick() {
      base.OnClick();
      int losnummer = zzg.Next(10000);
      if (losnummer % 7 == 0 && Seven != null) {
         SevenEventArgs sea = new SevenEventArgs();
         sea.Nr = losnummer;
         OnSeven(sea);
      }
   }
   protected virtual void OnSeven(SevenEventArgs sea) {
      Seven(this, sea);
   }
}
```

Im Namen einer Ereignis-auslösenden Methode gibt man in der Regel nach dem Präfix **On** das Ereignis an.

Nun kann eine abgeleitete Klasse bei der Ereignisentstehung mitreden und z.B. bei einer Losnummer < 5000 den Gewinn verweigern:

```
class EventProdD : EventProd {
   protected override void OnSeven(SevenEventArgs sea) {
      if (sea.Nr < 5000)
         base.OnSeven(mea);
   }
}
```

Das komplette Projekt ist im folgenden Ordner zu finden

…\BspUeb\Delegaten und Ereignisse\Event\EventPublisher

Als Ereignisanbieter kommen keinesfalls nur die mit dem Benutzer interagierenden Steuerelemente in Frage. Auch die mit der Datenverwaltung beschäftigten Klassen nutzen Ereignisse, um z.B. über Änderungen mit Relevanz für GUI-Elemente zu informieren.

9.3 Übungsaufgaben zu Kapitel 9

1) Welche von den folgenden Aussagen sind richtig?

1. Ein Ereignis ist ein Datentyp.
2. Bei der Delegatendefinition über eine anonyme Methode oder einen Lambda-Ausdruck können lokale Variablen sowie (statische) Felder der Umgebung verwendet werden.
3. C# unterstützt bei den Typformalparametern von generischen Delegaten die Ko- und Kontravarianz.
4. Weil die Delegatenvariable zu einem Ereignis grundsätzlich **privat** ist, kann man auch einer abgeleiteten Klasse keine Möglichkeit verschaffen, Kontrolle über die Auslösung eines Ereignisses zu gewinnen.
5. Die Registrierung eines Ereignisempfängers wieder aufzuheben, lohnt sich nicht.

2) Der FCL-Delegatentyp **Predicate<T>** im Namensraum **System** liefert für ein Argument vom Typ **T** eine boolesche Rückgabe, die darüber informiert, ob das Argument einem Kriterium genügt:

public delegate bool Predicate<in T>(T *instance***)**

Ein Argument vom Typ **Predicate<T>** dient in der Methode **FindAll()** der generischen Klasse **List<T>** dazu, aus der angesprochenen Liste eine Teilmenge für eine zu erstellende neue Liste zu gewinnen:

public List<T> FindAll(Predicate<T> *match***)**

Erstellen Sie per Lambda-Syntax ein Objekt vom Typ **Predicate<String>**, um aus einer Liste mit Zeichenfolgen die Elemente mit maximal 4 Zeichen in eine neue Liste zu befördern.

3) Von einem Delegatenobjekt, das an eine Methode zu übergeben ist, werden oft mehrere Varianten in Abhängigkeit von einem Parameter benötigt. Um bei Aufgabe 2 die Teillisten der Namen mit einer Maximallänge von $i = 1, .., 9$ Zeichen auszugeben, sollten die benötigten **Predicate<String>** -Objekte von einer Methode mit **int**-Parameter erzeugt werden, z.B. mit dem folgenden Definitionskopf:

```
public Predicate<String> ListLE(int k)
```

Man kann hier von einer Metamethode sprechen, weil als Rückgabe letztlich Methoden geliefert werden. Implementieren Sie eine solche Methode, und realisieren Sie damit die eben skizzierte Schleife.

10 Sonstige C# - Sprachbestandteile

In diesem kurzen Kapitel werden C# - Sprachbestandteile nachgetragen, die entweder zu keinem bisherigen Kapitel gepasst haben, oder an der thematisch passenden Stelle aufgrund ihrer hohen Komplexität didaktisch gestört hätten.

10.1 Null-bedingter Operator

Vor dem Zugriff auf eine Methode, Eigenschaft oder Instanzvariable eines Objekts sollte seine Existenz zur Vermeidung von **NullReferenceException**-Laufzeitfehlern geprüft werden, was den Quellcode durch häufig auftretende Routine-Passagen belastet. Im folgenden Beispiel soll aus einem **String**-Array die Länge des Elements mit der Nummer 1 ermittelt werden, was eine doppelte Existenzprüfung erfordert:

```
String[] ass = null;
int len1 = -1;
if (ass != null && ass[1] != null)
   len1 = ass[1].Length;
Console.WriteLine("Länge: " + len1);
```

Der **Null-bedingte Operator** (engl.: *null-conditional operator*) erleichtert den Objektzugriff mit vorheriger Existenzprüfung erheblich. Für den Member- bzw. Indexzugriff sind zwei leicht verschiedene Syntaxvarianten zu unterscheiden:

- *objekt***?.***member*
  Als Datentyp hat der Ausdruck den Null-erweiterten *member*-Typ. Wenn das Objekt nicht existiert, liefert der Ausdruck den Wert **null** (Referenz auf Nichts bei einer Klasse oder undefinierte Ausprägung bei einem Werttyp). Im folgenden Beispiel resultiert der Typ **int?** (vgl. Abschnitt 7.3):
  ```
  String s = null;
  int? sl = s?.Length;
  ```
  Eine Null-bedingte Operation unterscheidet sich vom normalen Member-Zugriff nur im Fall einer gescheiterten Existenzprüfung: Dann wird keine **NullReferenceException** geworfen, sondern der Wert **null** abgeliefert.
- *array***?[***index***]**
  Als Datentyp hat der Ausdruck den Null-erweiterten Elementtyp. Wenn das Objekt nicht existiert, liefert der Ausdruck die **null** des Elementtyps. Im folgenden Beispiel resultiert der Typ **String**:
  ```
  String[] ass = null;
  String ass1 = ass?[1];
  ```

Das Einstiegsbeispiel lässt sich mit zwei Null-bedingten Operatoren eleganter formulieren:

```
String[] ass = null;
int? len1 = ass?[1]?.Length;
Console.WriteLine("Länge: " + len1);
```

Bei einer Sequenz von Null-bedingten Operatoren findet eine Kurzschlussauswertung statt: Die Auswertung wird abgebrochen, sobald eine Existenzprüfung zum negativen Ergebnis geführt hat.

Soll das Ergebnis einen Werttyp haben, kombiniert man den Null-bedingten Operator mit dem Null-Sammeloperator (vgl. Abschnitt 7.3), z.B.:

```
int len1 = ass?[1]?.Length ?? -1;
```

Auch beim Aufruf eines Delegatenobjekts (vgl. Abschnitt 9.1) kann man sich mit dem Null-bedingten Operator explizite Existenzprüfungen ersparen, z.B.:

```
delegate void DemoGate(int w);
 . . .
DemoGate demoVar = null;
demoVar?.Invoke(2);
```

Dabei ist der explizite **Invoke()** - Aufruf nicht zu vermeiden.[1]

10.2 nameof-Operator

Der in C# 6.0 eingeführte **nameof**-Parameter macht sich dann nützlich, wenn im Quellcode der Name eines Typs oder Members benötigt wird, z.B. der Name eines Ereignisses im Konstruktor der Klasse **PropertyChangedEventArgs**, der im Zusammenhang mit der Information von Wertveränderungsinteressenten aufgerufen wird:

```
new PropertyChangedEventArgs("Einkommen")
```

Den Namen als Zeichenfolgenliteral anzugeben, ist wenig sinnvoll:

- Der Compiler kann Tippfehler nicht verhindern.
- Wenn sich der Name ändert, werden Zeichenfolgenliterale bei der Refaktorierung durch das Visual Studio nicht berücksichtigt.

Mit Hilfe des **nameof**-Operators lässt sich das Zeichenkettenliteral vermeiden:

```
new PropertyChangedEventArgs(nameof(Einkommen))
```

Wenn im Beispiel das Ereignis `Einkommen` per Refaktorierung umbenannt wird in `Income`, wird der Konstruktoraufruf automatisch mit aktualisiert:

```
new PropertyChangedEventArgs(nameof(Income))
```

Auch bei einer diagnostischen Ausgabe (z.B. in eine Protokolldatei) kann man per **nameof**-Operator die beschriebenen Vorteile nutzen, z.B.:

```
Console.WriteLine($"Falscher Wert {value} bei Eigenschaft {nameof(Income)}");
```

Die zur Demonstration verwendeten Anweisungen stammen aus einer Klasse namens `Person`, welche die Schnittstelle **INotifyPropertyChanged** implementiert und infolgedessen ein Ereignis namens **PropertyChanged** anbietet:[2]

```
using System;
using System.ComponentModel;
using System.Windows;

public class Person : INotifyPropertyChanged {
   public event PropertyChangedEventHandler PropertyChanged;
   double income;
   public double Income {
      get { return income; }
      set { if (value >= 0.0) {
            income = value;
            PropertyChanged?.Invoke(this,new PropertyChangedEventArgs(nameof(Income)));
          } else
            Console.WriteLine($"Falscher Wert {value} bei Eigenschaft {nameof(Income)}");
      }
   }
```

---

[1] https://msdn.microsoft.com/de-de/library/dn986595.aspx

[2] Die Anregung zu diesem Beispiel stammt von der MSDN-Webseite:
https://msdn.microsoft.com/de-de/library/ms743695(v=vs.110).aspx

```
    public String FirstName { get; set; }
    public String LastName { get; set; }

    public Person(String fn, String ln, double income) {
       FirstName = fn; LastName = ln; Income = income;
    }
}
```

Welche Rolle die Schnittstelle **INotifyPropertyChanged** bei der Benachrichtigung von Datenbindungsklienten über geänderte Eigenschaftsausprägungen spielt, wird in Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** erläutert.

10.3 Übungsaufgaben zu Kapitel 10

1) Auch im Umfeld des Null-bedingten Operators muss die Auswertungsreihenfolge von Ausdrücken (vgl. Abschnitt 3.5.10) eindeutig geklärt werden. Wird in der dritten Zeile des folgenden Code-Segments der Ausdruck rechts vom Identitätsoperator ausgeführt, wenn entweder die Variable `ass` ins Leere zeigt, oder das Element 1 des **String**-Arrays fehlt?

```
String[] ass = . . .
int res = 0;
bool sl1ok = ass?[1]?.Length == ++res;
```

11 Einstieg in die GUI-Programmierung mit WPF-Technik

Die *Windows Presentation Foundation* ist ein 2006 als .NET - Bestandteil eingeführtes GUI-Framework, das sich gegenüber seinem Vorgänger *WinForms* durchgesetzt hat und aktuell (Frühjahr 2017) bei der Entwicklung von Desktop-Anwendungen für Windows die erste Wahl ist.[1]

Allerdings steuert Microsoft seit Windows 8 unter dem Motto „Touch-First“ in Richtung auf Apps mit dem *Modern UI*, die aus dem Windows-Store bezogen werden. Das neueste Angebot von Microsoft sind die auf .NET Core (siehe Abschnitt 1.3) basierenden *UWP-Anwendungen* (*Universelle Windows-Plattform*), die auf verschiedenen Gerätegattungen laufen (Desktop, Tablet, Smartphone, Xbox, HoloLens etc.), sofern dort Windows 10 im Einsatz ist.

Derzeit verwenden allerdings weniger als 50% der Desktop-Computer unter Windows die Version 8.x oder 10, und den mit Abstand größten Marktanteil (von ca. 50%) besitzt Windows 7.[2] Wenn eine Software für Desktop-Computer *alle* auf dem Markt befindlichen Windows-Versionen (XP, Vista, 7, 8.x, 10) unterstützen soll, führt kein Weg an der WPF-Technik vorbei.

Lokal installierte WPF-Anwendungen für Desktop-Rechner unter Windows stehen durch das Aufkommen von Internet-basierten Anwendungen unter zunehmendem Konkurrenzdruck, haben aber immer noch klare Vorteile, z.B.:

- Uneingeschränkte Nutzung der PC-Hardware
- Überlegenheit beim wichtigsten Ergonomiekriterium: **Reaktionszeit**
- Benutzerfreundliche, vielfältige Bedienelemente
- Unabhängigkeit vom Internet

Wir werden uns im Kurs auf WPF-Desktop-Anwendungen beschränken, und können dieses Thema, dem umfangreiche Monographien gewidmet sind (z.B. MacDonald 2012, Nathan 2010), nicht annähernd erschöpfend behandeln.

Bei der vielleicht irgendwann anstehenden Umstellung von der WPF auf das Modern UI bleiben wesentliche Teile des erworbenen GUI - Know-Hows erhalten. Z.B. wird auch im Modern UI die Bedienoberfläche mit dem XML-Dialekt XAML deklariert, wenn auch mit leicht geänderter Syntax (Schacherl 2014, Abschnitt 1.3.7).

11.1 Einstimmung und Orientierung

Mit den Eigenschaften und Vorteilen einer grafischen Benutzeroberfläche (engl.: ***G****raphical* ***U****ser* ***I****nterface*) sind Sie sicher sehr gut vertraut. Eine GUI-Anwendung präsentiert dem Benutzer ein oder mehrere Fenster, die neben Bereichen zur Ausgabe bzw. Bearbeitung von programmspezifischen Dokumenten (z.B. Texten oder Grafiken) in der Regel zahlreiche normierte Bedienelemente zur Benutzerinteraktion besitzen (z.B. Menüzeile, Befehlsschalter, Kontrollkästchen, Textfelder, Auswahllisten). Die von einer Plattform (in unserem Fall vom WPF - Framework) zur Verfügung gestellten Bedienelemente bezeichnet man als *Steuerelemente*, *controls* oder *widgets* (Wortkombination aus *window* und *gadgets*). Weil die Steuerelemente intuitiv (z.B. per Maus) und in verschiedenen Programmen weitgehend konsistent zu bedienen sind, erleichtern Sie den Umgang mit Software erheblich.

---

[1] In einer früheren Version dieses Manuskripts (Baltes-Götz 2010) ist eine Behandlung der WinForms-GUI-Technik zu finden.

[2] https://www.netmarketshare.com/operating-system-market-share.aspx

11.1.1 Vergleich zwischen GUI- und Konsolenanwendungen

Im Vergleich zu Konsolenprogrammen geht es nicht nur intuitiver, sondern vor allem auch ereignisreicher[1] und mit mehr Mitspracherechten für den Anwender zu. Ein Konsolenprogramm entscheidet selbst darüber, welche Anweisung als nächstes ausgeführt wird, und wann der Benutzer eine Eingabe machen darf. Um seine Aufgaben zu erledigen, verwendet ein Konsolenprogramm diverse Dienste des Laufzeitsystems, z.B. bei der Aus- oder Eingabe von Zeichen.

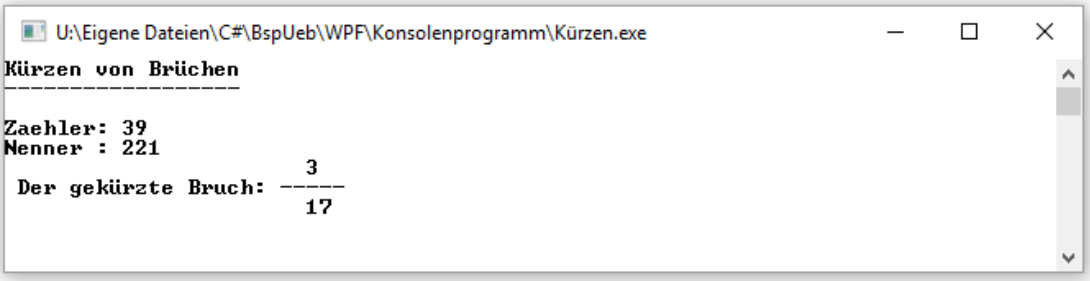
Für den Ablauf einer Applikation mit grafischer Benutzeroberfläche ist ein **ereignisorientiertes und benutzergesteuertes Paradigma** wesentlich, wobei das Laufzeitsystem als Vermittler oder (seltener) als Quelle von Ereignissen in erheblichem Maße den Ablauf mitbestimmt, indem es Methoden der GUI-Applikation aufruft, z.B. zum Zeichnen von Fensterinhalten. Ausgelöst werden die Ereignisse in der Regel vom Benutzer, der mit der Hilfe von Eingabegeräten wie Maus, Tastatur, Touch Screen etc. praktisch permanent in der Lage ist, unterschiedliche Wünsche zu artikulieren. Ein GUI-Programm präsentiert mehr oder weniger viele Bedienelemente und wartet die meiste Zeit darauf, dass eine der zugehörigen Ereignisbehandlungmethoden durch ein (meistens) vom **Benutzer** ausgelöstes **Ereignis** aufgerufen wird.

Im Vergleich zu einem Konsolenprogramm ist bei einem GUI-Programm die dominante Richtung im Kontrollfluss zwischen Anwendung und Laufzeitsystem invertiert. Die Ereignisbehandlungmethoden einer GUI-Anwendung sind Beispiele für so genannte *Call Back - Routinen*. Man spricht auch vom *Hollywood-Prinzip*, weil in dieser Gegend oft nach der Divise kommuniziert wird: „*Don't call us. We call you*".

Bei vereinfachter Betrachtungsweise kann man sagen:

- Eine Konsolenanwendung diktiert den Ablauf und erlaubt dem Benutzer gelegentlich eine Eingabe.
- Eine GUI-Anwendung stellt eine Sammlung von Ereignisbehandlungmethoden dar, wobei die zugehörigen Ereignisse vom Benutzer ausgelöst werden, indem er eines der zahlreichen Bedienelemente benutzt.

Betrachten wir zur Illustration eine Konsolen- und eine GUI-Anwendung zum Kürzen von Brüchen. Bei der Konsolenanwendung (vgl. Abschnitt 4.1.3)



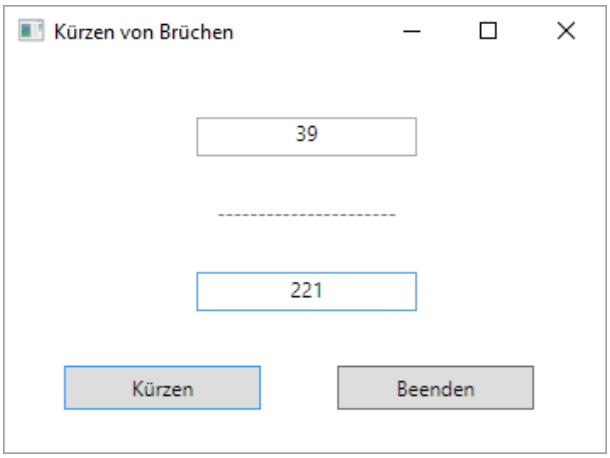
wird der gesamte Ablauf vom Programm diktiert:

- Es fragt nach dem Zähler.
- Es fragt nach dem Nenner.
- Es schreibt das Ergebnis auf die Konsole.

Im Unterschied zu diesem **programmgesteuerten Ablauf** wird bei der GUI-Variante

---

[1] Momentan wird bewusst ein starker Kontrast zwischen den bisher überwiegend benutzten Konsolenanwendungen und den nun vorzustellenden GUI-Anwendungen hinsichtlich der ereignisorientierten Programmierung herausgearbeitet. Allerdings kann grundsätzlich auch eine .NET - Konsolenanwendung mit Ereignissen umgehen. Wir werden z.B. in Abschnitt **14.3.3** ein Konsolenprogramm erstellten, das auf Ereignisse im Dateisystem (z.B. auf das Erstellen, Umbenennen oder Löschen von Dateien) reagiert.



das Geschehen vom Benutzer diktiert, der die vier Bedienelemente (zwei Texteingabefelder und zwei Schaltflächen) in beliebiger Reihenfolge verwenden kann, wobei das Programm mit seinen Ereignisbehandlungsmethoden reagiert (**benutzergesteuerter Ablauf**).

11.1.2 WPF-Leistungsmerkmale

Grundsätzlich ist das Erstellen einer GUI-Anwendung für Windows mit erheblichem Aufwand verbunden. Allerdings bietet die WPF-Technik außerordentlich leistungsfähige Klassen und Unterstützungsleistungen zur GUI-Programmierung, deren Verwendung durch Hilfsmittel der Entwicklungsumgebungen (z.B. Fensterdesigner) zusätzlich erleichtert wird.

Als WPF - Vorteile sind vor allem zu nennen:

- Trennung von Oberflächengestaltung und Interaktionslogik durch die Möglichkeit, die Benutzeroberfläche in einem XML-Dialekt namens *XAML* zu deklarieren
- Bessere Verbindung von Datenbeständen und GUI-Komponenten
- Attraktive 2D- und 3D-Grafik auf vektorieller Basis (ohne Qualitätsverlust beim Skalieren)
- Einheitliche Behandlung von Steuer- und Grafikelementen (z.B. Mausklickereignisse bei Grafikelementen)
- Hardware-beschleunigte Grafikausgabe
- Multimediale Vielfalt (Audio/Video) und Animationen
- Paralleles GUI-Design für Windows-Anwendungen und Web- bzw. Browser-Anwendungen

Eine WPF-*Browser-Anwendung* wird ...

- bei jedem Einsatz aus dem Internet bezogen
- und im Fenster eines WWW-Browsers ausgeführt, der ein Plugin mit der Microsoft-Technik Silverlight bieten muss.[1]

Wir beschränken uns im Kurs auf Desktop-Anwendungen.

Wie die meisten modernen GUI-Frameworks ist auch die WPF-Technik aus Konsistenz- und Performanzgründen nach dem Single-Thread-Prinzip konzipiert. Daher muss der Zugriff auf die GUI-Komponenten dem UI-Thread vorbehalten bleiben. Andererseits muss sich bei allen im UI-Thread ausgeführten Methoden der Zeitaufwand in Grenzen halten (maximal 100 Millisekunden), weil sonst die Bedienoberfläche zäh reagiert. Solange eine Methode läuft (z.B. gestartet als Reaktion auf

[1] Man sollte heutzutage **nicht** mehr damit rechnen, dass der Internet-Browser potentieller Benutzer Silverlight unterstützt. In einem kontrollierbaren Umfeld (z.B. unternehmensintern) ist Silverlight eventuell noch eine relevante Option, um die .NET - Softwareentwicklung mit der Internet-basierten Softwareverteilung zu kombinieren. Allerdings raten Doberenz & Gewinnus (2015, S. 1220) davon ab, noch ein neues Projekt mit Silverlight zu beginnen.

ein Ereignis), kann die Anwendung nicht auf andere Ereignisse (z.B. Mausklicks auf Steuerelemente) reagieren. Auch ist keine Aktualisierung der Anzeige möglich, zu der z.B. das Laufzeitsystem auffordert, weil ein bisher verdeckter Fensterbereich sichtbar geworden ist. Zeitaufwändige Arbeiten (z.B. Netzwerk-, Datei oder Datenbankzugriffe, aufwändige Berechnungen) gehören also in einen Arbeits-Thread. In der Regel müssen aber irgendwann Ergebnisse der Hintergrundtätigkeit an der Oberfläche sichtbar werden, wobei wegen der eingangs genannten Regel aus dem Arbeits-Thread keine direkten Zugriffe auf Steuerelemente möglich sind.

Selbstverständlich gibt es für die gerade skizzierte, höchst alltägliche Aufgabenstellung eine Routinelösung, die im Abschnitt 15.2.3 vorgestellt wird. Im aktuellen Kapitel gehen wird dem Thema *Multithreading* aus dem Weg, weil genügend andere Herausforderungen zu bewältigen sind.

11.2 Elementare Bausteine einer WPF-Anwendung

Bei der routinemäßigen Erstellung einer WPF-Anwendung verwendet man in der Regel den XML-Dialekt XAML (*Extended Application Markup Language*, siehe Abschnitt 11.3) zur Gestaltung der Bedienoberfläche. Man kann den XAML-Code direkt editieren, und/oder die Unterstützung des WPF-Designers im Visual Studio in Anspruch nehmen. Für Projekte mit hohen grafischen und multimedialen Ansprüchen steht außerdem noch das auf die GUI-Gestaltung spezialisierte Microsoft-Produkt *Blend for Visual Studio* zur Verfügung, das zusammen mit dem Visual Studio installiert wird.

Der bei routinemäßiger WPF-Projektarbeit von Designaufgaben entlastete Entwickler kann sich auf das Verfassen von Ereignisbehandlungsmethoden konzentrieren. Diese landen pro Fenster in einer C# - Quellcodedatei mit einer *partiellen* Klassendefinition (in der so genannten *Code-Behind* - Datei). Den ergänzenden Part erstellt die Entwicklungsumgebung aufgrund des XAML-Codes. Der gesamte Prozess bei der Erstellung einer WPF-Anwendung ist komplex, aber durchaus nachvollziehbar, und wir werden auch einige neugierige Blicke hinter die Kulissen werfen (siehe Abschnitte 11.3.3 und 11.3.4).

11.2.1 Eine minimalistische WPF-Anwendung (ohne XAML)

Um die elementaren Klassen und Abläufe bei einer WPF-Anwendung zu studieren, vermeiden wir aber zunächst den komplexen Entstehungsprozess via XAML und verwenden ein extrem einfaches, direkt und komplett in C# verfasstes Beispielprogramm. Wir legen im Visual Studio ein neues Projekt mit dem Namen `WpfOhneXaml` an, das die bei einer WPF-Anwendung erforderlichen Verweise auf DLL-Assemblies enthält, aber keinerlei Quellcode- oder XAML-Dateien. Dazu verwenden wir die Vorlage **WPF-Anwendung** und entfernen mit dem Projektmappen-Explorer die automatisch generierten Bestandteile:

- **App.xaml** (samt der zugehörigen Quellcodedatei **App.xaml.cs**) und
- **MainWindow.xaml** (samt der zugehörigen Quellcodedatei **MainWindow.xaml.cs**)
- alle Einträge im Zweig **Properties**

Im Vergleich zur Verwendung der Vorlage **Leeres Projekt** sparen wir uns die Deklaration der Assembly-Verweise und erhalten außerdem eine Anwendung mit dem Ausgabetyp Windows, so dass beim Starten kein Konsolenfenster erscheint (vgl. Abschnitt 2.2.6.2).

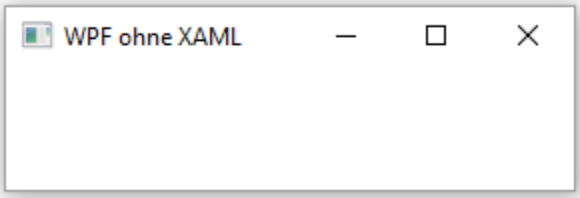
Wir legen über den Kontextmenübefehl **Hinzufügen > Neues Element** zum Projekt eine neue **Codedatei** namens **WpfOhneXaml.cs** an und definieren dort die Klasse `WpfOhneXaml` mit der Basisklasse **Window** aus dem Namensraum **System.Windows**:

```
using System.Windows;

class WpfOhneXaml : Window {
   WpfOhneXaml() {
      Title = "WPF ohne XAML";
      Width = 300;
      Height = 100;
   }

   [System.STAThread]
   static void Main() {
      Application app = new Application();
      WpfOhneXaml hf = new WpfOhneXaml();
      app.Run(hf);
   }
}
```

Auch ein GUI-Programm besteht aus Klassen, wobei eine Startklasse mit einer statischen **Main()** - Methode vorhanden sein muss (vgl. Abschnitt 1.1.4). Beim Programmstart wird die Startklasse vom Laufzeitsystem aufgefordert, ihre **Main()** - Methode auszuführen. Ein Hauptzweck dieser Methode besteht darin, Objekte zu erzeugen und somit Leben auf die objektorientierte Bühne zu bringen. Beim nun verwendeten WPF-Framework finden vermehrt Aktivitäten *hinter* der Bühne statt, die gleich erläutert werden sollen.

Bei einer WPF-Anwendung muss man die Methode **Main()** mit dem Attribut **System.STAThread** dekorieren.[1] Attribute sind Objekte, die mit einer speziellen Syntax (siehe Beispiel) an Klassen, Methoden etc. geheftet werden, um Informationen für den Compiler oder die CLR bereitzustellen. Nähere Informationen folgen in Kapitel 13. Im konkreten Fall muss ein bestimmtes Threading-Modell angemeldet werden, damit die WPF-Anwendung ausgeführt werden kann.

Einen allzu großen Funktionsumfang kann man vom aktuellen Beispielprogramm natürlich nicht erwarten:



Immerhin kann man das Anwendungsfenster (dank Windows und .NET - Framework) verschieben, seine Größe ändern, die Titelzeilen-Standardschaltflächen zum Minimieren, Maximieren und Beenden benutzen usw.

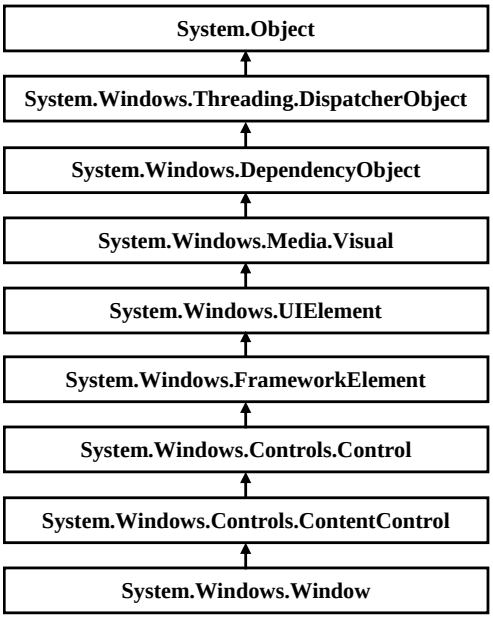
Im aktuellen Kapitel 11 werden Sie folgende Klassen als wichtige Bestandteile einer WPF-Anwendung kennen lernen:

[1] Eigentlich heißt die Klasse **STAThreadAttribute**, doch man darf *Attribute* weglassen, was in der Regel auch geschieht.

- **Window** (Namensraum **System.Windows**)
  Von dieser Klasse stammen alle Anwendungs- oder Dialogfenster ab.
- **Application** (Namensraum **System.Windows**)
  Diese Klasse stellt für eine WPF-Anwendung wichtige Dienste bereit. In der Regel definiert man eine eigene **Application**-Ableitung. Ein Objekt dieser Klasse repräsentiert die Anwendung und wird daher im Manuskript als *Anwendungsobjekt* bezeichnet.
- Klassen für Steuerelemente (z.B. **Label**, **Button**, **TextBox**) und Layoutcontainer zur Verwaltung von Steuerelementen (z.B. **Grid**, **StackPanel**)
- Delegatenklassen (z.B. **RoutedEventHandler**)
  Bei den in Abschnitt 9.1 behandelten Delegaten handelt es sich um Klassen, deren Objekte auf Methoden mit einer bestimmten erweiterten Signatur (mit Relevanz des Rückgabetyps) zeigen. Damit eine Ereignisbehandlungsmethode registriert werden kann, muss sie den zum Ereignis gehörigen Delegatentypen erfüllen.

11.2.2 (Haupt)fenster und die Klasse Window

Alle Fenster der im Manuskript auftauchenden WPF-Anwendungen werden über Objekte einer von **System.Windows.Window** abstammenden Klasse verwaltet. **Window** erbt seine Funktionalität wiederum zum großen Teil von allgemeineren Klassen:



In diesem Stammbaum tauchen viele fundamentale WPF-Klassen auf, sodass es nicht nur im Hinblick auf die Klasse **Window** sinnvoll ist, die Basisklassen mit Ihren Zuständigkeiten kurz zu erläutern:

- **System.Windows.Threading.DispatcherObject**
  Um eine konsistente und verzögerungsfrei reagierende Bedienoberfläche zu garantieren, verwendet das WPF-Framework eine **Single-Thread - Architektur**: Auf ein UI-Element darf nur derjenige Thread zugreifen, der das Element erzeugt hat. Um dies sicherzustellen, stammen die meisten WPF-Objekte von der Klasse **DispatcherObject** ab und kennen daher den **Dispatcher** (ein Objekt der Klasse **System.Windows.Threading.Dispatcher**) des erzeugenden Threads. Der **Dispatcher** ist in einem Thread für die Verwaltung der Warteschlange mit zu erledigenden Aufgaben (Methodenaufrufen) zuständig. Eine Methode mit Zugriff auf ein **DispatcherObject**-Objekt sollte über die **Dispatcher**-Methode **VerifyAccess()** überprüfen, ob sie in den Thread aufgerufen wurde, der das

DispatcherObject-Objekt erzeugt hat. Ist diese Bedingung *nicht* erfüllt, wirft **VerifyAccess()** eine **InvalidOperationException**.

- **System.Windows.DependencyObject**
  Objekte dieser Klasse können *Abhängigkeitseigenschaften* (engl.: *dependency properties*) anbieten. Diese spielen im WPF-Framework eine zentrale Rolle und unterstützen im Vergleich zu gewöhnlichen CLR-Eigenschaften z.B. Veränderungsmitteilungen, Stile und Animationen. Die meisten Eigenschaften von WPF-Steuerelementen sind als Abhängigkeitseigenschaften realisiert, können aber meist durch eine Hüllenkonstruktion wie gewöhnliche CLR-Eigenschaften verwendet werden. Durch Abhängigkeitseigenschaften wird im Vergleich zu gewöhnlichen CLR-Eigenschaften nicht nur eine gesteigerte Funktionalität ermöglicht, sondern auch in erheblichem Umfang Speicherplatz eingespart (Nathan 2010, S. 83).
- **System.Windows.Media.Visual**
  Objekte dieser Klasse besitzen elementare Grafikkompetenzen (z.B. Ausgabe auf dem Bildschirm, Trefferdiagnose bei Mausklicks).
- **System.Windows.UIElement**
  Diese Klasse steuert u.a. Kompetenzen bei der Ereignis- und Fokusbehandlung bei.
- **System.Windows.FrameworkElement**
  Diese Klasse ist verantwortlich für wesentliche WPF-Techniken wie Datenbindung, Stile und Animationen.
- **System.Windows.Controls.Control**
  Diese Klasse, von der viele konkrete Steuerelementklassen (z.B. **Button**, **TextBox**) abstammen, steuert die Fähigkeit zur Interaktion mit dem Benutzer bei. Außerdem unterstützt sie **ControlTemplate**-Objekte mit Einstellungspaketen zur Oberflächengestaltung.
- **System.Windows.Controls.ContentControl**
  Die von **System.Windows.Controls.ContentControl** abstammenden Klassen beherrschen das WPF-Inhaltsmodell, können also untergeordnete Elemente aufnehmen.

Das obige Beispielprogramm besteht aus der von **Window** abgeleiteten Klasse `WpfOhneXaml`

```
class WpfOhneXaml : Window
```

und erzeugt in seiner **Main()** - Methode *ein* Objekt aus dieser Klasse (also ein entsprechendes Fenster):

```
WpfOhneXaml hf = new WpfOhneXaml();
```

Als Aktualparameter im Methodenaufruf

```
app.Run(hf);
```

(gerichtet an das **Application**-Objekt `app`, das wir als *Anwendungsobjekt* bezeichnen)

```
Application app = new Application();
```

wird das `WpfOhneXaml`-Objekt `hf` zum Vertreter des **Anwendungs- oder Hauptfensters** im Programm. Unter den Fenstern eines Programms zeichnet sich das Hauptfenster durch folgende Besonderheiten aus:

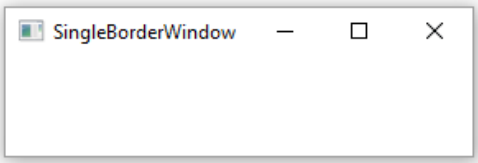
- Über den **Run()** - Aufruf wird für die Anzeige des Hauptfensters gesorgt. Sein Auftritt muss also *nicht* über die **Window**-Methode **Show()** veranlasst werden.
- Beim Schließen des Hauptfensters wird das Programm beendet.
- In der Regel werden erhebliche Teile der Programm-Funktionalität im Hauptfenster angeboten.

So wie im Konstruktor des Beispielprogramms mit der Anweisung

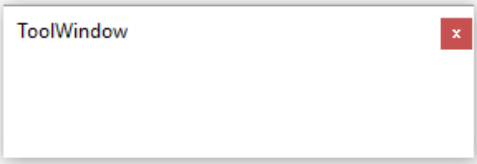
```
Title = "WPF ohne XAML";
```

die Titelzeilenbeschriftung des Fensters über die **Window**-Eigenschaft **Title** gewählt wird, sind zahlreiche weitere Eigenschaften dieser Klasse veränderbar, z.B.:

- Die Startgröße eines Fensters kann über die **FrameworkElement**-Eigenschaften **Height** und **Width** (vom Typ **double**) geändert werden, z.B.:
  ```
  Width = 300; Height = 100;
  ```
- Über die **Window**-Eigenschaft **WindowStyle** beeinflusst man die Ausstattung der Titelzeile mit Standardschaltflächen und die Rahmengestaltung, z.B.



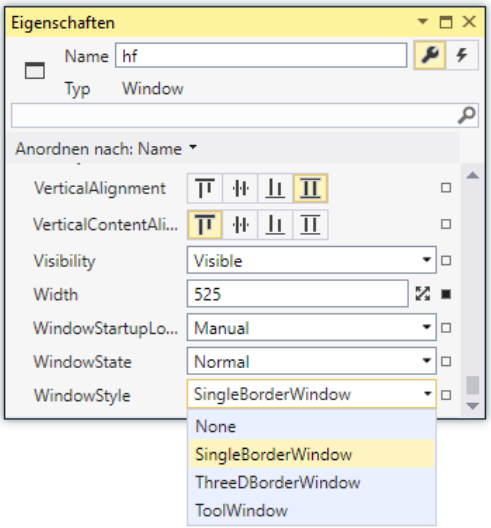
SingleBorderWindow



ToolWindow

None

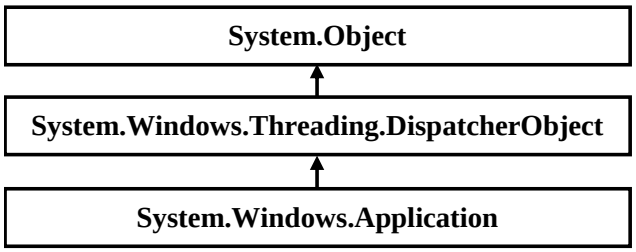
Beim Einsatz des WPF-Designers, mit dem wir schon etliche Erfahrungen gesammelt haben, ist die Eigenschaftsmodifikation zur Entwurfszeit mit dem **Eigenschaften**-Fenster bequem zu erledigen, z.B.:



Wir verzichten momentan auf diesen Komfort, um die Grundstruktur einer WPF-Anwendung studieren zu können.

11.2.3 Windows-Nachrichten und die Klasse Application

Eine WPF-Anwendung wird durch ein Objekt aus der Klasse **Application** oder aus einer Spezialisierung von **Application** repräsentiert. Im Vergleich zur Klasse **Window** hat die Klasse **Application** einen kleinen Stammbaum:



Das **Application**-Objekt erbringt u.a. folgende Leistungen:

- Es präsentiert wichtige Ereignisse in der Karriere einer Programminstanz, auf die Sie eventuell in einer Behandlungsmethode reagieren möchten:
  - **Startup**
    Das Ereignis wird in der Startphase ausgelöst. Eine Behandlungsmethode hat u.a. Gelegenheit, Befehlszeilenparameter auszuwerten.
  - **Activated**, **Deactivated**
    Das Programm (eines seiner Fenster) ist in den Vordergrund geholt bzw. von dort verdrängt worden.
  - **DispatcherUnhandledException**
    Wenn im primären UI-Thread eine unbehandelte Ausnahme auftritt, besteht die Standardreaktion des WPF-Frameworks darin, nach einer Infodialogbox das Programm zu beenden. Eine **DispatcherUnhandledException**-Behandlungsmethode kann z.B. einen Log-Eintrag schreiben und/oder die Beendigung des Programms verhindern.[1]
  - **SessionEnding**
    Der Benutzer ist im Begriff, seine Windows-Sitzung zu beenden (Abmelden des Benutzers oder Herunterfahren des Rechners).
  - **Exit**
    Das Programm sieht seinem Ende entgegen. Wie man auf dieses Ereignis reagieren kann, wurde in Abschnitt 9.2.2 demonstriert.
- Die **Application**-Eigenschaft **MainWindow** zeigt auf das Hauptfenster des Programms, und die Eigenschaft **Windows** zeigt auf eine Liste mit allen Top-Level - Fenstern.
- Die statische **Application**-Eigenschaft **Current** zeigt auf das Anwendungsobjekt.
- Durch die **Run()** - Methode des **Application**-Objekts wird die Verarbeitung der für eine Anwendung relevanten Ereignisse in Gang gesetzt, was im weiteren Verlauf des aktuellen Abschnitts näher erläutert werden soll.

Durch die von Windows registrierten Benutzeraktivitäten (z.B. Mausklicks, Tastenschläge) und sonstige Ursachen entstehen **Ereignisse** (im Sinne des Betriebssystems), die zu **Nachrichten** an betroffene Anwendungen führen. Wird z.B. ein Fenster vom Benutzer aus der Taskleiste zurückgeholt, dann fordert Windows die Anwendung mit der **WM_PAINT**-Nachricht auf, den Klientenbereich des Fensters neu zu zeichnen. Um diese laufend eintreffenden und in eine Warteschlange eingereihten Nachrichten kümmert sich bei einer WPF-Anwendung eine über die **Application**-

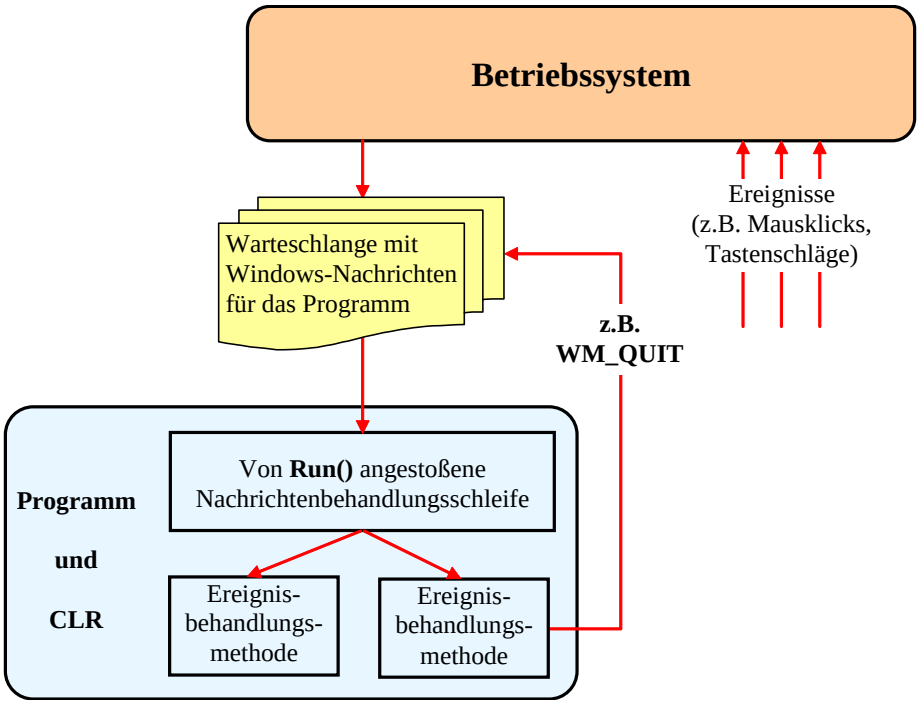
---

[1] Wegen unbehandelter Ausnahmen in anderen Threads des Programms (Hintergrund-Threads, zusätzliche UI-Threads mit eigenem Dispatcher), sind weitere Vorkehrungen erforderlich (siehe Abschnitt 15.1.8 und https://msdn.microsoft.com/de-de/library/system.windows.application.dispatcherunhandledexception(v=vs.110).aspx).

Instanzmethode **Run()** gestartete Routine des Laufzeitsystems in einer **while**-Schleife. Hat der Programmierer zu einer Nachricht eine Behandlungsmethode erstellt und zugeordnet, so wird diese aufgerufen. Man kann ein GUI-Programm als Ansammlung von Behandlungsmethoden auffassen, die beim Eintreffen einer passenden Nachricht aufgerufen werden. Solange eine Behandlungsmethode läuft, kann (im selben UI-Thread) keine weitere gestartet werden.

Zu manchen Nachrichten werden von Windows oder von der CLR (Common Language Runtime) des .NET - Frameworks ohne Zutun des Programmierers Behandlungsroutinen bereitgestellt. So kann unser Beispielprogramm z.B. auf die Standardschaltflächen in der Titelzeile (zum Maximieren, Maximieren oder Schließen) reagieren, ohne dass wir dazu eine Zeile Quellcode schreiben müssten.

Die folgende Abbildung zeigt einige Details zum Nachrichtenverkehr zwischen dem Betriebssystem, der per **Run()** initiierten Nachrichtenbehandlungsschleife und den Ereignisbehandlungsmethoden eines Programms mit *einem* UI-Thread:



Wird die Nachricht **WM_QUIT** aus der Warteschlange gefischt, endet die Nachrichtenbehandlungsschleife, und der **Run()** - Aufruf kehrt zurück.

Verantwortlich für die Nachricht **WM_QUIT** ist die **Application**-Methode **ShutDown()**, die vom Laufzeitsystem oder vom Programm aufgerufen werden kann.

Das Laufzeitsystem ruft die Methode **ShutDown()** in folgenden Fällen auf:

- Das letzte Fenster der Anwendung wird vom Benutzer geschlossen, und die **Application**-Eigenschaft **ShutdownMode** hat den voreingestellten Wert **OnLastWindowClose**.
- Das Hauptfenster der Anwendung wird vom Benutzer geschlossen, und die **Application**-Eigenschaft **ShutdownMode** hat den Wert **OnMainWindowClose**.
- Die Windows-Sitzung endet, weil sich der Benutzer abmeldet, oder das Betriebssystem heruntergefahren wird, und das daraufhin ausgelöste Ereignis **SessionEnding** bleibt entweder unbehandelt oder wird ohne Widerspruch gegen das Sitzungsende behandelt.

Hat die **Application**-Eigenschaft **ShutdownMode** den Wert **OnExplicitShutdown**, dann führt das Schließen des letzten Fensters oder des Hauptfensters *nicht* zur Beendigung des Programms. Um in dieser Lage bei laufender Windows-Sitzung für die Nachricht **WM_QUIT** zu sorgen, muss die Me-

thode **ShutDown()** per Programm aufgerufen werden. Dabei kann ein Return Code übergeben werden, um eine andere Anwendung zu informieren, die das beendete Programm gestartet hat.

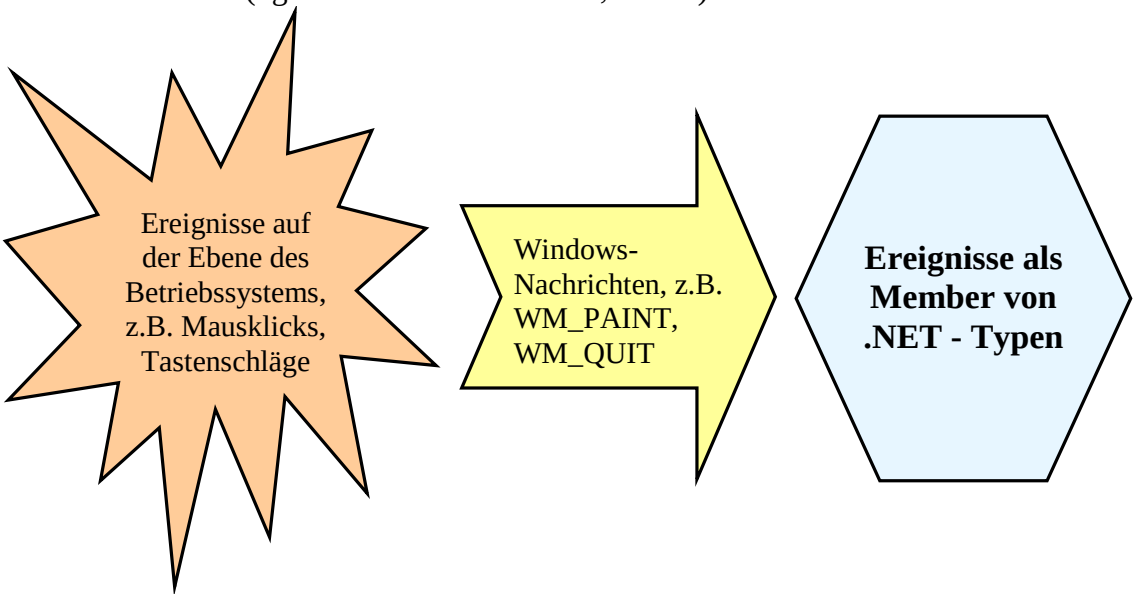
Eine WPF-Anwendung über die statische Methode **Environment.Exit()** zu beenden, ist in der Regel *nicht* sinnvoll, weil der komplette Prozess mit allen eventuell darin zusätzlich vorhandenen Threads beendet wird.

Fassen wir leicht vereinfachend zusammen, was durch den Aufruf der **Application**-Methode **Run()** veranlasst wird:

- Das **Hauptfenster** erscheint.
- Das Programm wird um eine **Nachrichtenschleife** erweitert, welche regelmäßig die von Windows für das Programm verwaltete Nachrichtenwarteschlange inspiziert und auf Ereignisse ggf. mit dem Aufruf einer registrierten Methode reagiert.[1]

Beim Eintreffen der Nachricht **WM_QUIT** enden die Nachrichtenschleife und die **Run()** - Methode. In der Regel stellt eine GUI-Anwendung nach der Rückkehr des **Run()** - Aufrufs ihre Tätigkeit ein (siehe **Main()** - Methode des Beispielprogramms in Abschnitt 11.2.1). Ein erneuter Aufruf der **Run()** - Methode ist *nicht* möglich.

Weil das Windows-API (*Application Programming Interface*) durch das .NET - Framework gekapselt wird, muss sich ein C# - Programmierer nicht direkt um Windows-Nachrichten kümmern, sondern kann die von vielen Klassen und Strukturen präsentierten **Ereignisse** im .NET - Sinn (siehe Abschnitt 9.2) durch eigene Methoden behandeln. Z.B. stellt die Klasse **Application** zur Windows-Nachricht **WM_QUIT** das Ereignis **Exit** zur Verfügung, bei dem .NET - Programmierer eine eigene Methode per Delegatenobjekt registrieren können, wenn sie auf das Ereignis (bzw. auf die zugrunde liegende Windows-Nachricht) regieren möchten (siehe Beispiel in Abschnitt 9.2.2). Eine registrierte Methode wird automatisch aufgerufen, wenn das zugehörige Ereignis eintritt. Insgesamt kann man unterscheiden (vgl. Louis & Strasser 2002, S. 614):



[1] Genau genommen ist für jeden Thread (vgl. Abschnitt 15), der ein Fenster auf dem Bildschirm präsentiert, eine Nachrichtenwarteschlange und dementsprechend eine Nachrichtenschleife erforderlich. Ein WPF-Programm kann also *mehrere* UI-Threads haben, was aber in den Kursbeispielen kaum vorkommen wird.

11.3 Die Extended Application Markup Language (XAML)

Die XML-basierte *Extended Application Markup Language* (XAML) wurde von Microsoft zur Deklaration der Bedienoberfläche einer WPF-Anwendung entwickelt, wird aber mittlerweile auch für Windows-Apps mit dem Modern UI verwendet. Während in der Zeit vor Einführung der WPF-Technik die Anwendungslogik *und* die Bedienoberfläche einer GUI-Anwendung per Quellcode (mit dem WinForms -Framework) zu realisieren waren, hat Microsoft für die WPF-Entwicklung die deklarative Sprache XAML mit folgenden Zielsetzungen entwickelt:

- Vereinfachung der GUI-Gestaltung
- Trennung von Anwendungslogik und GUI-Design

Damit bestehen gute Voraussetzungen für die erfolgreiche Zusammenarbeit von Software-Entwicklern und Designern.

11.3.1 Elementare Regeln zum Aufbau einer XML-Datei

Die *eXtended Markup Language* (XML) hat sich als universelles Mittel zur Deklaration von strukturieren Daten etabliert und wird auch im .NET - Framework intensiv verwendet, wobei besonders zu erwähnen sind:

- Anwendungs- und GUI-Deklaration per XAML
- Anwendungs- und benutzerbezogene Konfigurationsdateien im XML-Format

Daher werden in diesem Abschnitt elementare Regeln zum Aufbau einer XML-Datei erläutert.

Eine XML-Datei enthält Text und ist auch für die Lektüre durch Menschen relativ gut geeignet. In der ersten Zeile steht in der Regel eine Deklaration des Dokumententyps, die meist eine Versions- und eine Kodierungsangabe enthält, z.B.

```
<?xml version="1.0" encoding="utf-8" ?>
```

Bei den im aktuellen Kapitel vornehmlich relevanten XAML-Dateien wird diese Zeile allerdings weggelassen.

Jede XML-Datei besteht aus hierarchisch verschachtelten Elementen und enthält nur *ein Wurzelelement*, z.B. bei einer XAML-Datei zur Deklaration eines WPF-Fensters ein **Window** - Element:

```
<Window . . . >
  <Grid>
    <Button . . . >
      . . .
    </Button>
  </Grid>
</Window>
```

Wie das Beispiel **Window** zeigt, besteht ein XML-Element *mit* Inhalt (z.B. mit untergeordneten Elementen) aus

- Startkennung (oft bezeichnet als *Start-Tag*)
- Inhalt
- Endkennung (oft bezeichnet als *End-Tag*)

Als Bestandteile seiner Startkennung kann ein Element beliebig viele **Attribute** enthalten, die aus Name-Wert - Paaren bestehen. Das folgende **Button** - Element zur Deklaration eines Befehlsschalters besitzt z.B. die Attribute **Name** (benennt die Instanzvariable zum Steuerelement), **HorizontalAlignment** und **VerticalAlignment** (horizontale bzw. vertikale Orientierung des Steuerelements im umgebenden Steuerelement-Container):

```
<Button Name="button" HorizontalAlignment="Center" VerticalAlignment="Center">
  . . .
</Button>
```

Alle Attributausprägungen sind durch (doppelte oder einfache) Anführungszeichen zu begrenzen.

Ein Element *ohne* untergeordnete Elemente (oder sonstige Inhalte) kann sich auf die Startkennung beschränken, die dann mit einem Schrägstrich zu enden hat, z.B. beim folgenden **Image** - Element aus einer XAML-Datei:

```
<Image Source="count.png" />
```

Wie beim C# - Quellcode ist auch beim XML-Code die **Groß-/Kleinschreibung relevant** (mit seltenen Ausnahmen, auf die bei Gelegenheit hingewiesen wird).

Das folgende Beispiel zeigt, wie **Kommentare** in einer XML-Datei untergebracht werden:

```
<!-- Kommentar -->
```

11.3.2 XAML-Kurzbeschreibung

Das Erstellen und Initialisieren der Objekte der Bedienoberfläche kann auch bei einer WPF-Anwendung per Quellcode erfolgen. Es ist jedoch zu empfehlen, für diesen Zweck die Extended Application Markup Language (XAML) zu verwenden. Die XAML-Deklarationen werden in Textdateien mit der UTF-8 - Kodierung und der Namenserweiterung **xaml**. untergebracht.

11.3.2.1 Wurzelelement

Eine XAML-Datei einhält *ein* Wurzelelement, wobei für uns derzeit in Frage kommen:[1]

- **Application**
  Dieses Wurzelelement wird in der XAML-Datei mit der Anwendungsdeklaration verwendet. Es nennt auf jeden Fall die Anwendungsklasse und die XAML-Datei zum Hauptfenster. Außerdem können Behandlungsmethoden zu Ereignissen der Anwendungsklasse sowie Ressourcen deklariert werden. Als Name der zugehörigen XAML-Datei wird meist **App.xaml** verwendet. Welchen XAML-Code das Visual Studio 2015 für eine neue WPDF-Anwendung erzeugt, betrachten wir am Beispiel unseres RSS-Feed-Reader - Projekts (vgl. Abschnitt 5.7):

```
<Application x:Class="RssFeedReader.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:RssFeedReader"
             StartupUri="MainWindow.xaml">
    <Application.Resources>

    </Application.Resources>
</Application>
```

- **Window**
  Dieses Wurzelelement wird in der XAML-Datei zur Deklaration eines Fensters (bzw. einer Fensterklasse) verwendet. Beim Hauptfenster wählt das Visual Studio 2015 den Dateinamen **MainWindow.xaml**.

Jedes Element einer XAML-Datei repräsentiert ein Objekt einer .NET - Klasse, und der Elementname ist identisch mit dem Namen dieser Klasse (z.B. **Button**) oder mit dem Namen ihrer Basisklasse (z.B. **Window**).

Die Startkennung zu einem XAML - Wurzelelement enthält **xmlns**-Attribute zur Deklaration von **XML-Namensräumen** über die Syntax:

xlmns[**:***prefix*]="*URI*"

---

[1] Für die im Kurs nicht behandelten Anwendungen mit einer Browser-analogen Navigation werden zusätzlich XAML-Dateien mit dem Wurzelelement **Page** benötigt.

In einem Namensraum werden erlaubte Elemente und Attribute festgelegt, und zur Vermeidung von Namenskollisionen kann ein Präfix angegeben werden.

Es besteht eine starke Beziehung zwischen den vertrauten Namensräumen für .NET - Klassen und den XML-Namensräumen in einer XAML-Datei. Da jedes XAML-Element mit einer .NET - Klasse assoziiert ist, muss für den XAML-Parser schließlich erkennbar sein, zu welchem Namensraum diese Klasse gehört. Es wäre allerdings sehr umständlich, für die auf verschiedene .NET - Namensräume verteilten WPF-Klassen jeweils den korrekten Präfix ermitteln und verwenden zu müssen. Daher wurden alle .NET - Namensräume mit WPF-Klassen in einem einzigen XAML-Namensraum zusammengefasst. Dies ist möglich, weil keine Namenskollisionen auftreten. Um die Verwendung des sehr wichtigen XAML-Namensraums mit den WPF-Klassen zu erleichtern, hat er *keinen* Präfix:

```
xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
```

Die Bezeichnung für einen XAML-Namensraum verwendet das URI-Format (*Uniform Resource Identifier*), um Eindeutigkeit sicherzustellen. Wie man sich leicht vergewissern kann, existieren die angegebenen Internet-Orte aber *nicht*.

Neben dem Namensraum für die WPF-Klassen gibt es einen weiteren extrem wichtigen XAML-Namensraum. Er enthält u.a. wichtige Attribute der Sprache XAML und verwendet das Präfix **x**:[1]

```
xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
```

Außerdem vereinbart das Visual Studio 2015 einen XAML-Namensraum mit dem Präfix **local** für den projektspezifischen .NET - Namensraum:

```
xmlns:local="clr-namespace:RssFeedReader"
```

Die Vereinbarung von XAML-Namensräumen erfolgt meist im Start-Tag des Wurzelelements, ist aber grundsätzlich in *jedem* Start-Tag erlaubt.

Die folgenden **x**-Attribute werden oft benötigt:

- **x:Class**
  Im **x:Class**-Attribut wird die zu einem Element gehörige Klasse samt Namensraum genannt.
- **x:Key**
  Die sogenannten Ressourcen, die in einer WPF-Anwendung auf einfache Weise die Mehrfachverwendung von Objekten erlauben, erhalten über dieses Attribut einen Schlüssel, unter dem sie später ansprechbar sind.
- **x:Name**
  Über dieses Attribut legt man für die Instanz, die aus der XAML-Laufzeitverarbeitung eines XAML-Elements entsteht, einen Namen fest. Es resultiert eine Instanzvariable mit dem gewählten Namen, die in der Code-Behind - Datei (siehe Abschnitt 11.3.3) einen Instanzzugriff erlaubt. Viele .NET - Klassen besitzen eine **Name**-Instanzeigenschaft. Dazu gehört auch die Klasse **FrameworkElement**, von der alle WPF-Steuerelemente abstammen. Wenn eine Klasse mit **Name**-Instanzeigenschaft außerdem mit dem Attribut (im Sinn von Kapitel 13) **RuntimeNameProperty** dekoriert ist (wie z.B. die Klasse **FrameworkElement**), kann statt des XAML-Attributs **x:Name** auch das Attribut **Name** verwendet werden. In diesem Fall verwendet der XAML-Parser die angegebene Zeichenfolge für die Referenzvariable *und* für die Instanzeigenschaft des Objekts (MacDonald 2012, 28f).

Beim Erstellen und Initialisieren von Objekten per XAML-Code muss man sich nicht auf WPF-Klassen beschränken. Es werden beliebige .NET - Klassen unterstützt, sofern diese über einen parameterlosen und öffentlichen Konstruktor verfügen. Um beliebige CLR-Typen im XAML-Code

---

[1] Weitere Information zu den beiden wichtigen Namensdeklarationen einer XAML-Datei finden sich hier: https://msdn.microsoft.com/de-de/library/ms747086(v=vs.110).aspx

ansprechen zu können, bezieht man ihren .NET - Namensraum ein (siehe obiges Beispiel mit dem Namensraum des Projekts).

11.3.2.2 Instanzelemente

Ein Instanzelement deklariert eine Instanz (z.B. ein Objekt) mit einem beliebigen, in einem zugänglichen Assembly realisierten .NET - Typ. In der Startkennung eines Instanzelements ist nach der öffnenden spitzen Klammer der Typname anzugeben.

Sind keine untergeordneten Elemente vorhanden, ist die Startkennung mit einem Schrägstrich und einer schließenden spitzen Klammer zu komplettieren, z.B.

```
<TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
```

Sind untergeordnete Elemente oder sonstige Inhalte vorhanden (z.B. Steuerelemente in einem Layoutcontainer), beendet man die Startkennung mit einer schließenden spitzen Klammer, lässt die untergeordneten Elemente folgen und setzt eine Endkennung, die zwischen einem Paar spitzer Klammern einen Schrägstrich und den Element- bzw. Typnamen enthält, z.B.:

```
<StackPanel Name="stackPanel" Orientation="Horizontal">
    <Image Width="20" Source="rss.gif" VerticalAlignment="Center"/>
    <TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
</StackPanel>
```

Ein Instanzelement stellt einen Auftrag an die XAML-Verarbeitung dar, eine Instanz vom angegebenen Typ zu erstellen. Das tatsächliche Erstellen findet zur Laufzeit über einen Aufruf des parameterlosen Konstruktors zum jeweiligen Typ statt (siehe Abschnitt 11.3.4).

Das Instanzieren einer *nicht* zum XAML-Standardnamensraum, sondern zum projektspezifischen XAML-Namensraum gehörigen Klasse ist uns übrigens in Abschnitt 9.2.3 im Zusammenhang mit einem benutzerdefinierten Steuerelement begegnet:

```
<Grid>
   <local:SurpriseButton x:Name="surpriseButton" Content="7 gewinnt"
    Margin="30" FontSize="36"/>
</Grid>
```

11.3.2.3 Eigenschaftsausprägungen zuweisen

11.3.2.3.1 Attributsyntax

Viele Instanzeigenschaften können in der Startkennung über die **XML-Attributsyntax** einen Wert erhalten, wobei auf den Namen der Eigenschaft das „=“ - Zeichen und durch Anführungszeichen begrenzt eine Zeichenfolge als Wert folgt, z.B.:[1]

```
<TextBlock Text="Click to Feed" Margin="5" VerticalAlignment="Center" />
```

Bei der Wandlung einer Zeichenfolge in den jeweiligen Eigenschaftstyp sind **Typkonverter** im Spiel, die eventuell eine komplexe Aufgabe zu erfüllen haben. Im Beispiel muss aus der Zeichenfolge zur **TextBlock**-Eigenschaft **Margin**, die für einen Abstand zur Umgebung sorgt, eine Instanz vom Strukturtyp **System.Windows.Thickness** entstehen. Welche Klasse für die Wandlung zuständig ist, wird durch ein Attribut (im Sinn von Kapitel 13) zum Typ der Eigenschaft festgelegt, im Beispiel:

```
[TypeConverter(typeof(ThicknessConverter))]
[Localizability(LocalizationCategory.None, Readability = Readability.Unreadable)]
public struct Thickness : IEquatable<Thickness> { ... }
```

[1] Das Visual Studio verwendet doppelte Anführungszeichen, doch sind auch einfache erlaubt.

11.3.2.3.2 Eigenschaftselementsyntax

Die Attributsyntax eignet sich *nicht*, wenn einer Eigenschaft ein komplexes Objekt zugewiesen werden soll. In diesem Fall ordnet man dem Instanzelement ein **Eigenschaftselement** unter, dessen Name nach dem Schema

Typname.Eigenschaftsname

zu bilden ist. Als Inhalt des Eigenschaftselements tritt ein Instanzelement mit dem Typ der Eigenschaft auf. Im folgenden Beispiel wird der **Content**-Eigenschaft eines **Button**-Objekts (Datentyp: **Object**) ein **StackPanel**-Layoutobjekt zur Verwaltung der **Button**-Oberfläche (siehe unten) zugewiesen:

```
<Button Name="button" Background="WhiteSmoke">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      . . .
    </StackPanel>
  </Button.Content>
</Button>
```

Gleich wird sich allerdings herausstellen, dass bei der **Button**-Eigenschaft **Content** eine Vereinfachung der XAML-Syntax möglich ist. Weil diese Eigenschaft bei der Klasse **Button** sehr oft angesprochen werden muss, ist sie als *XAML-Inhaltseigenschaft* (siehe Abschnitt 11.3.2.3.5) für **Button**-Elemente festgelegt worden, so dass man auf die Einrahmung

```
<Button.Content> ... </Button.Content>
```

verzichten kann.

Die Attributsyntax ist letztlich eine bequeme Kurzform der Eigenschaftselementsyntax. Z.B. kann man statt

```
<Button Name="button" Content ="Click to Feed" Background="WhiteSmoke"/>
```

auch schreiben:

```
<Button Name="button" Content ="Click to Feed">
  <Button.Background>
      <SolidColorBrush Color="WhiteSmoke"/>
  </Button.Background>
</Button>
```

11.3.2.3.3 Textinhaltsyntax

Bei einigen Eigenschaftselementen kann Text (ohne Begrenzung durch Anführungszeichen) als Inhalt angegeben werden, z.B.:

```
<Button>
    <Button.Content>
        Knopf
    </Button.Content>
</Button>
```

Für den Datentyp der betroffenen Eigenschaft muss eine von den folgenden Bedingungen erfüllt sein:[1]

- Einer Eigenschaft von diesem Typ kann eine Zeichenfolge zugewiesen werden, was z.B. beim Typ **Object** (dem Typ der **Button**-Eigenschaft **Content**) der Fall ist.

[1] Siehe http://msdn.microsoft.com/de-de/library/ms752059.aspx

- Für den Typ ist eine Klasse mit entsprechenden Kompetenzen als Typkonverter definiert, z.B.:

```
<Button.Background>
    LightBlue
</Button.Background>
```

- Der Typ ist im XAML-Sprachumfang bekannt, was bei vielen elementaren Datentypen (z.B. **Int32**, **Int64**, **Double**) der Fall ist, z.B.:

```
<Button.Height>
    40
</Button.Height>
```

Die Textinhaltsyntax wird meist bei der *Inhaltseigenschaft* (siehe Abschnitt 11.3.2.3.5) verwendet, so dass sich im ersten Beispiel ergibt:

```
<Button>
    Knopf
</Button>
```

11.3.2.3.4 Kollektionssyntax

Besitzt eine Eigenschaft einen Kollektionstyp, der das Interface **IList** implementiert, darf man die **XAML-Kollektionssyntax** verwenden, d.h.:

- auf ein Eigenschaftselement zum Kollektionsobjekt verzichten
- und eine Liste mit Instanzelementen angeben, die schlussendlich über die **IList**-Methode **Add()** in die Kollektion aufgenommen werden.

Im folgenden Beispiel wird für **LinearGradientBrush.GradientStops** die Eigenschaftselement-syntax verwendet. Statt ein Instanzelement vom Typ **GradientStopCollection** samt Kindelementen anzugeben, werden ausschließlich die Kindelemente aufgelistet:[1]

```
<LinearGradientBrush>
  <LinearGradientBrush.GradientStops>
    <GradientStop Offset="0.0" Color="Red" />
    <GradientStop Offset="1.0" Color="Blue" />
  </LinearGradientBrush.GradientStops>
</LinearGradientBrush>
```

11.3.2.3.5 Inhaltseigenschaft

Jede Klasse kann von ihren Instanzeigenschaften genau eine als **Inhaltseigenschaft** (engl.: *content property*) deklarieren. Dazu wird der Klasse das Attribut **DefaultProperty** angeheftet, wobei hier der Begriff *Attribut* im Sinn von Kapitel 13 zu verstehen ist, z.B. bei der Klasse **ContentControl**:

```
[ContentProperty("Content")]
[Localizability(LocalizationCategory.None, Readability = Readability.Unreadable)]
public class ContentControl : Control, IAddChild {
  . . .
}
```

Ist ein Kindelement als Wert der Inhaltseigenschaft anzugeben, kann auf das Kennungspaar gemäß XAML-Eigenschaftselementsyntax verzichtet werden. Folglich kann das in Abschnitt 11.3.2.3 präsentierte Beispiel mit einer Deklaration für die **Button**-Eigenschaft **Content**

[1] Übernommen aus: https://msdn.microsoft.com/de-de/library/ms752059(v=vs.110).aspx

```
<Button Name="button" Background="WhiteSmoke">
  <Button.Content>
    <StackPanel Orientation="Horizontal">
      . . .
    </StackPanel>
  </Button.Content>
</Button>
```

einfacher geschrieben werden:

```
<Button Name="button" Background="WhiteSmoke">
  <StackPanel Orientation="Horizontal">
      . . .
  </StackPanel>
</Button>
```

Bei der Layoutcontainer - Klasse **StackPanel** ist die (von **Panel** geerbte) Eigenschaft **Children** als Inhaltseigenschaft deklariert:

```
[Localizability(LocalizationCategory.Ignore)]
[ContentProperty("Children")]
public abstract class Panel : FrameworkElement, IAddChild {
  . . .
}
```

Folglich kann das Kennungspaar

```
<StackPanel.Children> . . . </StackPanel.Children>
```

weggelassen werden.

Die Eigenschaft **Children** ist vom Typ **UIElementCollection**, der das Interface **IList** implementiert. Folglich darf man die **XAML-Kollektionssyntax** verwenden (siehe Abschnitt 11.3.2.3.4). Im folgenden Beispiel

```
<StackPanel Orientation="Horizontal">
  <Image Width="20" Source="/Routing;component/rss.gif" />
  <TextBlock HorizontalAlignment="Right" Text="Click to Feed" />
</StackPanel>
```

werden zwei XAML-Vereinfachungsmöglichkeiten genutzt:

- Inhaltseigenschaft
  Das Eigenschaftselement **StackPanel.Children** ist weggelassen.
- Kollektionssyntax
  Aus den Kindelementen wird ein Kollektionsobjekt erstellt und der Inhaltseigenschaft zugewiesen.[1]

Bei der Inhaltseigenschaft wird oft die im Abschnitt 11.3.2.3.3 beschriebene Textinhaltsyntax verwendet, z.B.:

```
<Button Name="button">
  Click to Feed
</Button>
```

[1] Weil die Klasse **UIElementCollection** keinen öffentlichen und parameterfreien Konstruktor anbietet, wäre ein Instanzelement von diesem Typ ohnehin fehlerhaft.

11.3.2.3.6 Markup-Erweiterungen

Mit den bisher vorgestellten Techniken zur Festlegung von Eigenschaftsausprägungen ist es *nicht* möglich, ...

- eine Referenz zu einem bereits existenten Objekt zuzuweisen,
- eine Verbindung zu einer Quelle von dynamisch zu aktualisierenden Werten herzustellen.

Dies gelingt mit einer so genannten *Markup-Erweiterung*, die meist per Attributsyntax unter Verwendung einer geschweift eingeklammerten Zeichenfolge realisiert wird. Im folgenden Beispiel wird einer Eigenschaft eine Referenz auf ein anderenorts definiertes Objekt zugewiesen:

```
Background="{StaticResource BgColor}"
```

Es ist auch möglich, allerdings weniger gebräuchlich, Markup-Erweiterungen in Eigenschaftselementen zu verwenden (siehe MacDonald 2012, S. 34).

Eine Markup-Erweiterung hat einen Typ, der bei Verwendung der Attributsyntax unmittelbar nach der öffnenden Klammer anzugeben ist. Die Erweiterungstypen stammen von der Klasse **MarkupExtension** im .NET - Namensraum **System.Windows-Markup** ab. Viele für WPF-Anwendungen wichtige Markup-Erweiterungstypen wurden in den XAML-Standardnamensraum aufgenommen, sodass bei der Typbezeichnung kein Namensraumpräfix benötigt wird. Besonders wichtig sind:

- **StaticResource**
  In WPF-Anwendungen werden häufig so genannte statische Ressourcen per Markup-Erweiterung zugewiesen. So wird es möglich, einmalig definierte Einstellungsobjekte (z.B. zur Hintergrundgestaltung) in mehreren Steuerelementen oder Fenstern eines Programms zu verwenden. In der folgenden XAML-Datei zur Anwendungsklasse eines Programms wird in die Kollektion (Datentyp **ResourceDictionary**) zur **Application**-Eigenschaft **Resources** ein **SolidColorBrush**-Objekt aufgenommen, das anschließend anwendungsglobal über den vereinbarten Schlüssel `BgColor` genutzt werden kann:

  ```
  <Application x:Class= ... >
      <Application.Resources>
          <SolidColorBrush x:Key="BgColor" Color="WhiteSmoke"/>
      </Application.Resources>
  </Application>
  ```

  In der folgenden XAML-Datei zum Hauptfenster derselben Anwendung wird das **SolidColorBrush**-Objekt per Markup-Erweiterung der **Window**-Eigenschaft **Background** als statische Ressource zugewiesen, wobei nach dem Markup-Erweiterungstyp der Schlüssel anzugeben ist:

  ```
  <Window x:Class="MarkupExtDemo.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Routing" Height="350" Width="525"
      Background="{StaticResource BgColor}">
    . . .
  </Window>
  ```

- **Binding**
  Im Abschnitt 5.7.6 haben wir schon Bekanntschaft mit der WPF-Datenbindungstechnologie gemacht. Für den zur formatierten Anzeige eines **ListBox**-Elements verwendeten **TextBlock**

  ```
  <TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
     TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
     Foreground="DarkMagenta" />
  ```

wird durch die folgende Attributsyntax mit Markup-Erweiterung

```
Text="{Binding Path=Title}"
```

ein Objekt der Klasse **Binding** beauftragt, aus dem aktuellen Element der zum **ListBox**-Objekt gehörigen Kollektion den Wert der Eigenschaft `Title` zu extrahieren.

Manche Markup-Erweiterungsklassen sind *nicht* im XAML-Standardnamensraum für WPF-Klassen vorhanden, sondern im generellen XAML-Namensraum definiert, so dass dem Typnamen der Namensraumpräfix **x** vorangestellt wird. Ein wichtiges Beispiel ist die Markup-Erweiterung **StaticExtension**, die eine Referenz auf ein statisches Feld oder auf eine statische Eigenschaft liefert. Im folgenden Beispiel wird sie dazu verwendet, der Steuerelementeigenschaft **Background** das durch die statische Eigenschaft **ControlDarkBrush** der Klasse **SystemColors** referenzierte Objekt zuzuweisen. Dabei darf der Namensbestandteil *Extension* weggelassen werden:

```
Background="{x:Static SystemColors.ControlDarkBrush}"
```

Nach der Markup-Typbezeichnung ist als Konstruktorparameter eine Zeichenfolge mit dem folgenden Aufbau anzugeben:

*prefix***:***typeName***.***fieldOrPropertyname*

Das Präfix ist nur bei Typen anzugeben, die *nicht* in den XAML-Standardnamensraum eingeblendet wurden.

11.3.2.4 Ereignisbehandlungsmethoden registrieren

Wie die folgende Startkennung zum **Application**-Wurzelelement aus der XAML-Datei zu einer Anwendungsklasse zeigt, kann man per Attributsyntax nicht nur einer Eigenschaft, sondern auch einem Ereignis einen Wert zuweisen, wobei der Name der Behandlungsmethode anzugeben ist:

```
<Application x:Class="ApplicationExitVS.App"
   xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
   xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
   StartupUri="MainWindow.xaml" Exit="Application_Exit">
     . . .
</Application>
```

Implementiert wird eine solche Behandlungsmethode in der Code-Behind - Datei zum XAML-Code (siehe Abschnitt 11.3.3).

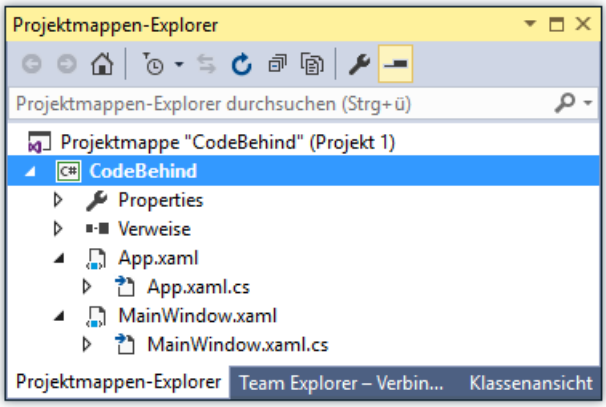
Man kann darüber diskutieren, ob hier eventuell das Prinzip der sauberen Trennung von Anwendungslogik und GUI-Gestaltung verletzt ist.

11.3.3 Code-Behind - Dateien

Ein vom Visual Studio 2015 über die Vorlage **WPF-Anwendung** neu erstelltes Projekt enthält die beiden XAML-Dateien

- **App.xaml** zur Deklaration der Anwendungsklasse
- **MainWindow.xaml** zur Deklaration der Hauptfensterklasse

Wie der Projektmappen-Explorer nach Aufklappen der Zweige zu den beiden XAML-Dateien zeigt, gehört jeweils eine C# - Quellcodedatei dazu, deren Name aus dem XAML-Dateinamen durch Anhängen der Erweiterung **.cs** entsteht:



Diese beiden Dateien sind für die Anwendungslogik zuständig, werden im Wesentlichen vom Entwickler erstellt und als *Code-Behind - Dateien* bezeichnet.

Sie enthalten nach etlichen **using**-Direktiven zum Import von .NET - Namensräumen jeweils eine partielle Klassendefinition:

- **App.xaml.cs**
  In der Datei **App.xaml.cs** findet sich eine partielle Definition der Anwendungslasse `App`, die von der FCL-Klasse **Application** (vgl. Abschnitt 11.2.3) abstammt:

```
using System;
  . . .
namespace CodeBehind {
 /// <summary>
 /// Interaktionslogik für "App.xaml"
 /// </summary>
 public partial class App : Application {
 }
}
```

  Mit dem Schlüsselwort **partial** wird dem Compiler signalisiert, dass es zu der im aktuellen Quellcode vorhandenen Klassendefinition noch ein Ergänzungsstück in einer anderen Quellcodedatei gibt, wobei die beiden partiellen Definitionen gleichberechtigt und voneinander abhängig ein Ganzes ergeben.

  Wenn Sie im Visual Studio eine Ereignisbehandlung zum Anwendungsobjekt anfordern, wird die partielle Klassendefinition in der Datei **App.xaml.cs** um eine Behandlungsmethode erweitert, z.B.

```
public partial class App : Application {
   private void Application_Exit(object sender, ExitEventArgs e) {
   }
}
```

  Diesen Methodenrumpf zum **Exit**-Ereignis der Klasse **Application** kann man z.B. so erstellen:

  - Datei **App.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
  - Das Wurzelelement **Application** markieren
  - Im Eigenschaftenfenster per Mausklick auf das Symbol ⚡ die Registerkarte mit den Ereignissen wählen
  - Doppelklick auf die Textbox zum Ereignis **Exit**

  Dabei erhält in der zugehörigen XAML-Datei das Wurzelelement **Application** einen Wert für das Attribut **Exit**, z.B.:

```
<Application x:Class="CodeBehind.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:CodeBehind"
             StartupUri="MainWindow.xaml" Exit="Application_Exit">
```

- **MainWindow.xaml.cs**
  In der Datei **MainWindow.xaml.cs** findet sich eine partielle Definition der Fensterklasse `MainWindow`, die von der FCL-Klasse **Window** (vgl. Abschnitt 11.2.2) abstammt:

```
using System;
  . . .
namespace CodeBehind {
 /// <summary>
 /// Interaktionslogik für MainWindow.xaml
 /// </summary>
 public partial class MainWindow : Window {
    public MainWindow() {
       InitializeComponent();
    }
 }
}
```

  Wenn Sie im Visual Studio eine Ereignisbehandlung zu einem Steuerelement auf der Fensteroberfläche anfordern, wird die partielle Klassendefinition in der Datei **MainWindow.xaml.cs** um eine Behandlungsmethode erweitert, z.B.

```
public partial class MainWindow : Window {
 public MainWindow() {
    InitializeComponent();
 }
    private void button_Click(object sender, RoutedEventArgs e) {
    }
}
```

  Diesen Methodenrumpf zum **Click**-Ereignis der Klasse **Button** kann man z.B. so erstellen:
  - Datei **MainWindow.xaml** per Doppelklick auf den Eintrag im Projektmappen-Explorer öffnen
  - Im WPF-Designer einen Doppelklick auf das **Button**-Objekt setzen

  Dabei erhält in der zugehörigen XAML-Datei **MainWindow.xaml** das zum **Button**-Objekt gehörige Element einen entsprechenden Wert für das Attribut **Click**:

```
<Button ... Click="button_Click" />
```

  Bei einem von Click verschiedenen Ereignistyp ist das Eigenschaftenfenster zu verwenden:
  - Das Steuerelement im WPF-Designer oder in der XAML-Ansicht markieren
  - Im Eigenschaftenfenster per Mausklick auf das Symbol  die Registerkarte mit den Ereignissen wählen
  - Doppelklick auf die Textbox zum Ereignis

Im nächsten Abschnitt ist zu erfahren, wo die Ergänzungsstücke zu den partiellen Klassendefinitionen in **App.xaml.cs** und **MainWindow.xaml.cs** stecken.

11.3.4 XAML-Verarbeitung beim Erstellen und Starten einer WPF-Anwendung

In diesem Abschnitt werfen wir einen Blick hinter die WPF-Kulissen, wobei interessante Details der technischen Realisation erkennbar werden. So wird z.B. die Startmethode **Main()** lokalisiert, die auch bei einer WPF-Anwendung ihre übliche Rolle in der Startphase spielt. Allerdings ist dieses Hintergrundwissen für die Praxis der Anwendungsentwicklung mit dem Visual Studio nicht erforderlich, so dass Sie den Abschnitt im Vertrauen auf die WPF-Magie ignorieren können.

Enthält eine WPF-Anwendung XAML-Dateien (= Normalfall), dann lässt unsere Entwicklungsumgebung bei jeder Erstellung der Anwendung zuerst zu jeder Fensterklasse die zugehörige XAML-Datei (z.B. **MainWindow.xaml**) vom XAML-Compiler **xamlc.exe** in eine binäre (validierte und effizient zu verarbeitende) Variante mit der Namenserweiterung **.baml** übersetzen (z.B. **MainWindow.baml**). Diese Dateien landen bei einem Projekt mit der Zielplattform **Any CPU** und der Build-Konfiguration **Debug** im Projektunterordner

...\obj\Debug

Die BAML-Dateien zu den Fensterklassen werden als so genannte *Ressourcen* in das Assembly eingebunden und beim Programmstart vom XAML-Parser ausgewertet (siehe unten). Dabei entstehen GUI-Objekte mit den im XAML-Code festgelegten Eigenschaften.

Außerdem erzeugt der XAML-Compiler im eben angegebenen Projektunterordner aus jeder XAML-Datei (zu einem Fenster oder zur Anwendung gehörig) eine C# - Quellcodedatei mit einer partiellen Klassendefinition, bei einer WPF-Anwendung mit der Anwendungsklasse `App` und der *einem* Fenster (Hauptfensterklasse `MainWindow`) also die Dateien:

- **MainWindow.g.i.cs**
  In der partiellen Klassendefinition zum Hauptfenster
  ```
  public partial class MainWindow :
    System.Windows.Window, System.Windows.Markup.IComponentConnector {
      . . .
  }
  ```
  werden die Referenzvariablen zu den Steuerelementen des Fensters deklariert, z.B.:
  ```
  internal System.Windows.Controls.Button button;
  ```
  Die Variablennamen stammen aus den **Name** - oder **x:Name** - Attributen der zugehörigen XAML-Instanzelemente (vgl. Abschnitt 11.3.2.1).
  Außerdem findet sich hier die im Fensterklassenkonstruktor (siehe Code-Behind - Datei) aufgerufene Methode **InitializeComponent()**. Diese sorgt durch einen Aufruf der statischen **Application**-Methode **LoadComponent()** für das Laden der BAML-Ressource und damit für das Instanzieren und Initialisieren der zugehörigen Objekte:
  ```
  public void InitializeComponent() {
    if (_contentLoaded) {
     return;
    }
    _contentLoaded = true;
    System.Uri resourceLocater =
     new System.Uri("/CodeBehind;component/mainwindow.xaml", System.UriKind.Relative);
      . . .
    System.Windows.Application.LoadComponent(this, resourceLocater);
      . . .
  }
  ```
  Schließlich enthält die partielle `MainWindow`-Klassendefinition noch die Methode **Connect**(), die den Vertrag der Schnittstelle **System.Windows.Markup.IComponentConnector** erfüllt und die `MainWindow`-Referenzvariablen mit den BAML-Objekten (Steuerelementen) verbindet:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object
target) {
    switch (connectionId)
    {
    case 1:
    this.button = ((System.Windows.Controls.Button)(target));

    #line 10 "..\..\MainWindow.xaml"
    this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);

    #line default
    #line hidden
    return;
    }
    this._contentLoaded = true;
}
```

Die Klasse `MainWindow` implementiert die Methode **Connect()** *explizit*, sodass der Schnittstellenname angegeben werden muss und kein Zugriffsmodifikator gesetzt werden darf (siehe Abschnitt 8.5). In **Connect()** werden ggf. auch die Ereignisbehandlungsmethoden zu den Steuerelementen registriert, welche in der Regel in der vom Programmierer gepflegten Code-Behind - Datei **MainWindow.xaml.cs** (siehe Abschnitt 11.3.3) mit dem Rest der `MainWindow`-Klassendefinition implementiert werden.

- **App.g.i.cs**
  Hier findet sich die partielle Definition der von **System.Windows.Application** abgeleiteten Anwendungsklasse mit dem (vom Visual Studio gewählten) Namen `App`:

```
public partial class App : System.Windows.Application {
    . . .
  public void InitializeComponent() {

    #line 5 "..\..\App.xaml"
    this.Exit += new System.Windows.ExitEventHandler(this.Application_Exit);

    #line default
    #line hidden

    #line 5 "..\..\App.xaml"
    this.StartupUri = new System.Uri("MainWindow.xaml", System.UriKind.Relative);

    #line default
    #line hidden
  }

  [System.STAThreadAttribute()]
   . . .
  public static void Main() {
    CodeBehind.App app = new CodeBehind.App();
    app.InitializeComponent();
    app.Run();
  }
}
```

Um das Hauptfenster festzulegen, wird in der `App`-Methode **InitializeComponent()** (nicht zu verwechseln mit der gleichnamigen Methode in der Klasse `MainWindow`) der **Application**-Eigenschaft **StartupUri** ein **Uri**-Objekt zugewiesen, das auf der XAML-Deklaration der Hauptfensterklasse basiert.

Schließlich findet sich in der partiellen, automatisch erstellten `App`-Klassendefinition die Startmethode **Main()**. Im Unterschied zu der manuell erstellten **Main()** - Methode unserer minimalen WPF-Anwendung in Abschnitt 11.2.1 wird das Hauptfenster nicht per **Run()** - Parameter festgelegt. Es ist dem Anwendungsobjekt bereits über seine **StartupUri**-Eigenschaft bekannt.

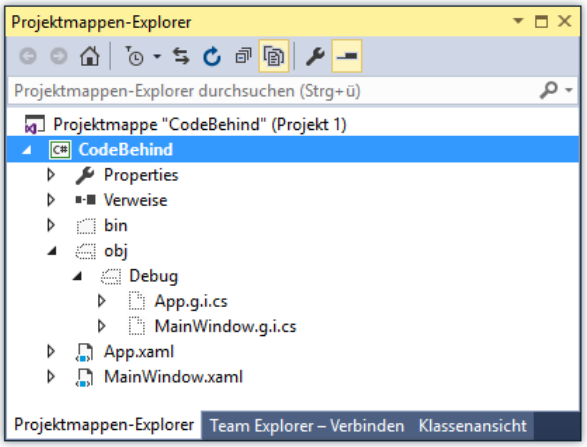
Behandlungsmethoden für App-Ereignisse werden in der eben beschriebenen Methode **InitializeComponent()** registriert und in der Code-Behind - Datei **App.xaml.cs** zur Anwendungsklasse implementiert.

Die mit **#line** startenden Zeilen in **MainWindow.g.i.cs** bzw. **App.g.i.cs** enthalten C#-Präprozessordirektiven:[1]

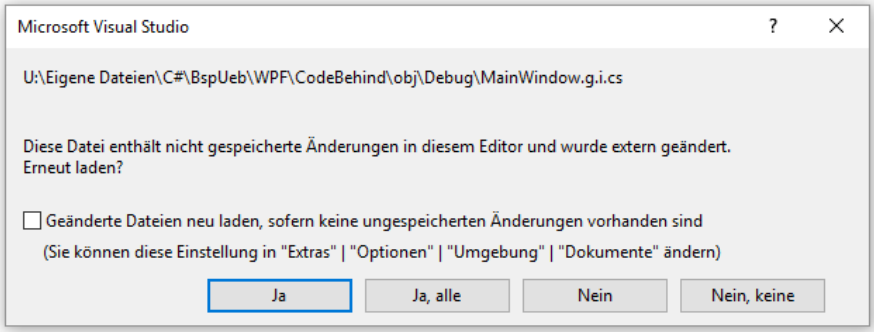
- **#line** *number* "*file*"
  Für die Ausgabe einer Fehlerlokalisierung durch den Compiler werden die Zeilennummer und der Dateiname geändert.
- **#line default**
  Die Fehlerlokalisierung wird auf das Standardverfahren zurückgesetzt.
- **#line hidden**
  Alle Zeilen bis zur nächsten **#line**-Direktive werden vor dem Debugger verborgen, also z.B. bei der schrittweisen Ausführung nicht angezeigt. Auf die Zeilennummerierung hat diese Direktive keinen Einfluss.

Gehen Sie folgendermaßen vor, wenn Sie die generierten Quellcode-Dateien im Visual Studio öffnen wollen:

- Im Projektmappen-Explorer über den Symbolschalter **alle Dateien anzeigen** lassen
- Pfad **obj\Debug** mit den generierten Dateien öffnen, z.B.:



- Gewünschte Datei per Doppelklick im Quellcode-Editor öffnen
  Wenn die folgende Anfrage erscheint, sollten Sie mit **Ja** antworten, um den aktuellen Stand zu erhalten:



Neben den **g.i** - Dateien, die vom Visual Studio für die Intellisense-Funktionalität benutzt werden, gibt es noch die **g** - Dateien. Während die **g.i** - Dateien bei jeder Änderung am Projekt vom Visual Studio aktualisiert werden, erhalten die **g** -Dateien nur beim Erstellen den aktuellen Stand.

---

[1] https://msdn.microsoft.com/de-de/library/ed8yd1ha.aspx

11.4 Routingereignisse

Wie zu Beginn von Kapitel 11 erwähnt, geht es bei der GUI-Programmierung sehr ereignisreich zu. Folgerichtigerweise wurde für das WPF-Framework mit den so genannten *Routingereignissen* eine Ergänzung bzw. Erweiterung der CLR-Ereignistechnik eingeführt. Das MSDN (*Microsoft Developer Network*) nennt zwei Besonderheiten von Routingereignisse im Vergleich zu den in Abschnitt 9.2 behandelten generellen Ereignissen:[1]

- Funktionale Definition
  Ein Routingereignis ist ein Ereignistyp, der Handler für mehrere Listener in einer Elementstruktur aufrufen kann, und nicht nur für das Objekt, von dem das Ereignis ausgelöst wurde.
- Implementierungsdefinition
  Ein Routingereignis basiert auf einem Objekt der Klasse **RoutedEvent**-Klasse und wird vom WPF-Ereignissystem verarbeitet.

Hinsichtlich der automatischen Weiterleitung entlang einer Hierarchie von potentiellen Interessenten zeigt das WPF-Ereignissystem eine Verwandtschaft zur Kommunikation von Ausnahmefehlern durch das Laufzeitsystem (siehe Kapitel 12).

Um jede Verwechslung der in Abschnitt 9.2 beschriebenen Ereignisse mit den nun vorzustellenden Routingereignissen zu vermeiden, werden erstere ab jetzt als *CLR-Ereignisse* bezeichnet.

Weiterhin ist zu betonen, dass in einer WPF-Anwendung keinesfalls alle Ereignisse vom Routing-Typ sind, dass also im WPF-Framework die CLR-Ereignisse nicht ersetzt, sondern ergänzt werden.

Wenn man eine Behandlungsmethode für ein Routingereignis bei der Ereignisquelle registriert, was wir bisher getan haben, dann ist der Unterschied zwischen normalen Ereignissen und Routingereignissen unsichtbar und irrelevant. Wir haben z.B. für den in Abschnitt 5.7 entwickelten RSS-Feed - Reader mit Assistentenhilfe (in der Code-Behind - Datei zur Hauptfensterklasse `MainWindow`) die folgende Behandlungsmethode zum **Click**-Ereignis der WPF-Klasse **Button** erstellt:

```
private void button_Click(object sender, RoutedEventArgs e) {
   . . .
}
```

Diese wird in der vom XAML-Compiler generierten Quellcodedatei **MainWindow.g.i.cs** (vgl. Abschnitt 11.3.4) wie bei einem gewöhnlichen Ereignis (vgl. Abschnitt 9.2.2) „an der Quelle“ registriert:

```
this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);
```

Um diese Verwendung von Routingereignissen mit der vertrauten C# - Syntax für Ereignisse zu ermöglichen, bieten die meisten Routingereignisse einen Wrapper. Diesen Hinweis nehmen wir vorläufig nur dankbar zur Kenntnis. Relevant wird das Thema erst dann, wenn man eigene WPF-Ereignisse definieren möchte.

Neu am WPF-Ereignissystem ist die Möglichkeit, Behandlungsmethoden nicht (nur) bei der Ereignisquelle (hier: **Button**-Objekt `button`) zu registrieren, sondern bei beliebigen Knoten (auch mehreren), die im Baum der hierarchisch verschachtelten GUI-Elemente auf der Route von der Ereignisquelle bis zur Wurzel (hier: Hauptfensterobjekt aus der von **Window** abstammenden Klasse `MainWindow`) auftreten.

---

[1] http://msdn.microsoft.com/de-de/library/ms742806.aspx

11.4.1 Routingereignisse definieren

Der folgende Quellcode zeigt, wie in der FCL-Klasse **ButtonBase** (Namensraum **System.Windows.Controls.Primitives**) das Routingereignis **ClickEvent** (ein Objekt der Klasse **RoutedEvent**) deklariert sowie durch die statische **EventManager**-Methode **RegisterRoutedEvent()** erzeugt und registriert wird:

```
public static readonly RoutedEvent ClickEvent = EventManager.RegisterRoutedEvent(
   "Click", RoutingStrategy.Bubble, typeof(RoutedEventHandler), typeof(ButtonBase));
```

Beim Registrieren sind anzugeben:

- Der Ereignisname
- Die Routingstrategie (siehe Abschnitt 11.4.2)
- Die von Behandlungsmethoden zu erfüllende Delegatenklasse
- Der Datentyp des Eigentümers
  Hier gibt man die Klasse an, die das Routingereignis registriert.

Der Name eines statischen Feldes vom Typ **RoutedEvent** sollte mit *Event* enden.

In der Regel wird ein CLR-Ereignis als **Wrapper** (dt.: *Hülle*) um das Routingereignis konstruiert, damit die normale Syntax zum Registrieren von Ereignisbehandlungsmethoden anwendbar ist. Im Beispiel wird das CLR-Ereignis **Click** mit der in Abschnitt 9.2.1 beschriebenen Syntax definiert, wobei die explizit realisierten **add**- und **remove**-Methoden für die Verbindung zum **RoutedEvent**-Objekt sorgen:[1]

```
public event RoutedEventHandler Click {
   add {
      AddHandler(ClickEvent, value);
   }
   remove {
      RemoveHandler(ClickEvent, value);
   }
}
```

Aufgerufen wird das Routingereignis in der **ButtonBase**-Methode **OnClick()**, wobei aber *nicht* wie bei einem CLR-Ereignis das Delegatenobjekt aufgerufen wird (vgl. Abschnitt 9.2.3), sondern die **UIElement**-Methode **RaiseEvent()**:

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

11.4.2 Routingstrategien

Nach der verwendeten Weiterleitungsstrategie sind folgende Kategorien von Routingereignissen zu unterscheiden:

- **Tunnelereignisse**
  Hier kommt als Routingstrategie das **Tunneling** zum Einsatz. Das von einem GUI-Element (z.B. **Image**-Objekt als Inhaltsbestandteil einer **Button**-Oberfläche) ausgelöste Ereignis wird zuerst dem Wurzelelement im GUI-Baum (z.B. einem **Window**-Objekt) zur Behandlung angeboten. Anschließend durchläuft das Ereignis die hierarchisch *absteigende* Route bis zur Ereignisquelle. Weil die übergeordneten Elemente das Ereignis *vor* dem unmittelbar betroffenen Element sehen, starten die Namen der Tunnelereignisse mit dem Präfix **Pre-**

[1] Der Quellcode stammt der Datei **Application.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann.

view, z.B. **PreviewMouseDown**. Dementsprechend werden Tunnelereignisse gelegentlich auch als *Vorschauereignisse* bezeichnet.
Beispiel: **PreviewMouseLeftButtonDown**-Ereignis der Klasse **UIElement**

- **Blasenereignisse**
  Hier kommt als Routingstrategie das **Bubbling** zum Einsatz. Zuerst werden die bei der Ereignisquelle registrierten Behandlungsmethoden aufgerufen. Dann werden die die bei übergeordneten Elementen im GUI-Baum bis hinauf zur Wurzel registrierten Behandlungsmethoden nacheinander aufgerufen.
  Beispiel: **Click**-Ereignis der Klasse **ButtonBase**
- **Direktereignisse**
  Nur die bei der Quelle registrierten Behandlungsmethoden werden aufgerufen, was der Verarbeitung von normalen CLR-Ereignissen entspricht. Allerdings sind einige Techniken des WPF-Ereignissystems realisiert, z.B. statische Behandlungsmethoden, die für alle Objekte ihrer Klasse zuständig sind und noch vor den Instanz-Behandlungsmethoden aufgerufen werden (siehe Abschnitt 11.4.5).
  Beispiele: **MouseEnter**-Ereignis der Klasse **UIElement**

Ein Tunnel- oder Blasenereignis ist also *nicht* erledigt, nachdem *eine* Behandlungsmethode aufgerufen worden ist, sondern es wird entlang der Hierarchie weiter angeboten, bis es seine Endstation erreicht hat oder als behandelt deklariert wird. Dazu ist in einer Behandlungsmethode die Eigenschaft **Handled** des übergebenen Beschreibungsobjekts auf den Wert **true** zu setzen.

Für Routingereignisse wird in der MSDN-Dokumentation die Routingstrategie angegeben, z.B. beim **Click**-Ereignis der Klasse **ButtonBase**:

Routed Event Information

Identifier field	ClickEvent
Routing strategy	Bubbling
Delegate	RoutedEventHandler

Bei einem Routingereignis erhält die Behandlungsmethode (analog zu einem CLR-Ereignis) die beiden folgenden Parameter:

- **object** *sender*
  Dieses Objekt hat die Behandlungsmethode aufgerufen.
- **RoutedEventArgs** *e*
  Das Ereignisbeschreibungsobjekt besitzt u.a. die beiden folgenden Eigenschaften:
  - **Source**
    Man erfährt das Objekt, welches das Routingereignis ausgelöst hat.
  - **Handled**
    Über diese boolesche Eigenschaft kann der Ereignisbehandlungsstatus eingesehen oder gesetzt werden.

Speziell die mit Eingaben (z.B. per Maus oder Tastatur) beschäftigten Routingereignisse treten meist als Paar aus einem Tunnel- und einem Blasenereignis auf (z.B. **PreviewMouseDown** und **MouseDown**), wobei zur Abfolge gilt:

- Zunächst wird das Tunnelereignis ausgelöst. Auf dem Weg von der Wurzel des Elementbaums bis zur Ereignisquelle werden registrierte Behandlungsmethoden sukzessive aufgerufen.
- Wenn das Tunnelereignis seine Endstation erreicht, wird das zugehörige Blasenereignis ausgelöst.
- Ein Tunnelereignis als behandelt zu markieren, wirkt sich analog auf das zugehörige Blasenereignis aus, weil beide Routingereignisse *dasselbe* Ereignisbeschreibungsobjekt (aus der Klasse **RoutedEventArgs** oder aus einer abgeleiteten Klasse) verwenden. Tunnelereignisse dienen meist dazu, eine Ereignisbehandlung zu blockieren oder vorzubereiten.
- Auch ein als behandelt markiertes Ereignis setzt seine Wanderung fort und wird Behandlungsmethoden angeboten, die auf besondere Weise über die **UIElement**-Methode **AddHandler()** registriert worden sind (mit dem Wert **true** für den booleschen Parameter **handledEventsToo**).

Eine Ausnahme vom Paarungsprinzip ist das Blasenereignis **Click** der Klasse **ButtonBase**, zu dem kein Tunnelereignis existiert.

11.4.3 Praktische Bedeutung und Einsatzempfehlungen

Nachdem etliche technische Details zu den Routingereignissen geklärt worden sind, geht es nun um die praktische Relevanz der verschiedenen Techniken und Empfehlungen zur Verwendung.

Generell ist das Routing vor allem bei Eingabeereignissen wichtig, weil elterliche Steuerelemente von Eingabeereignissen erfahren, die bei einem ihrer Kinder aufgetreten sind. Wenn die Ereignisquelle von Interesse ist, kann sie über die Eigenschaft **Source** des übergebenen Beschreibungsobjekts ermittelt werden.

Blasenereignisse bewähren sich z.B. in den folgenden Einsatzszenerien:

- Gemeinsame Ereignisbehandlung für mehrere Steuerelemente
  Befinden sich z.B. in einem Container mehrere **Button**-Objekte, und sollen deren **Click**-Ereignisse durch eine gemeinsame Methode behandelt werden, dann genügt eine einmalige Registrierung dieser Methode beim Container. Stünde hinter **Click** *kein* „aufperlendes“ Routingereignis, dann müsste die Registrierung für jeden Schalter wiederholt werden (wie beim veralteten WinForms-Framework).
- Ereignisse für zusammengesetzte Steuerelemente
  Ein **Button**-Objekt kann Inhalte aufnehmen, z.B. ein **Image**- und ein **TextBlock**-Objekt:

```
<Button Name="button" ... Click="button_Click">
  <StackPanel>
    <Image  Source="vs2013.png"/>
    <TextBlock Text="Click"/>
  </StackPanel>
</Button>
```

  Die einen **Click**-Ereignis konstituierenden elementaren Ereignisse wie **MouseLeftButtonDown** und **MouseLeftButtonUp** treten jeweils bei einem Inhaltsbestandteil auf, z.B. ...
  - beim **Image**-Objekt das Routingereignis **MouseLeftButtonDown**
  - und wegen einer leichten Mausbewegung beim **TextBlock**-Objekt das Routingereignis **MouseLeftButtonUp**.

  Weil alle Ereignisse an das **Button**-Objekt weitergeleitet werden, kann dort ein Klick auf den Schalter erkannt und als neues, „höherwertigeres“ Routingereignis gefeuert werden.

Tunnelereignisse werden von Anwendungsprogrammen viel seltener behandelt als Blasenereignisse. Manchmal ist es aber für ein elterliches UI-Element relevant, *vor* seinen Kindern von einem Ereignis zu erfahren. So wird es möglich, ...

- ein Ereignis zu blockieren
  Dazu wird in der Behandlungsmethode zu einem Tunnelereignis die **Handled**-Eigenschaft des übergebenen Beschreibungsobjekts auf den Wert **true** gesetzt. Wenn z.B. ein **TextBox**-Steuerelement nur Ziffern erhalten soll, kann man über das Ereignis **PreviewTextInput** eine Eingabevalidierung vornehmen und das zugehörige **TextInput**-Ereignis blockieren, wenn ein irreguläres Zeichen eingegeben wurde (zu weiteren Details der Eingabevalidierung siehe MacDonald 2012, S. 124).
- eine Ereignisbehandlung vorzubereiten

Ein typisches WPF-Programm wird sich ....

- nur um Blasenereignisse kümmern
- und diese dort behandeln, wo sie ausgelöst worden sind.

Oft wird das Ereignisrouting komplett ignoriert, doch ist es durchaus empfehlenswert, nach einer erfolgreichen Ereignisbehandlung die Eigenschaft **Handled** des übergebenen Beschreibungsobjekts auf den Wert **true** zu setzen, um (eventuell unerwartete) weitere Behandlungsmethoden zu informieren.

Der Abschnitt 11.4 muss sich auf generelle Informationen über das WPF-Ereignissystem beschränken. Über spezielle Ereignisgruppen (z.B. Maus- und Tastaturereignisse) informiert z.B. MacDonald (2012, Kapitel 4).

11.4.4 Eine Beobachtungsstudie

Um das WPF-Ereignissystem bei der Arbeit beobachten zu können, erstellen wir eine Anwendung mit dem folgenden Baum von UI-Elementen:

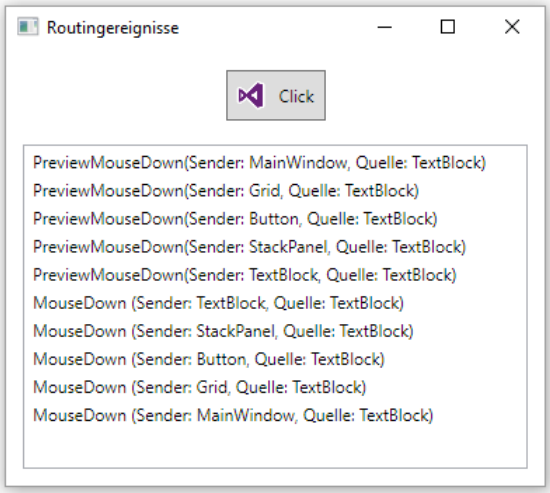
Ein Wurzelelement aus der von **Window** abstammenden Klasse `MainWindow`,
darin ein Layoutcontainer aus der Klasse **Grid** (siehe Abschnitt 11.6.1),
in der **Grid**-Zelle (0, 0) u.a. ein Befehlsschalter aus der Klasse **Button**,
darin ein Layoutcontainer aus der Klasse **StackPanel** (siehe Abschnitt 11.6.2),
darin ein Objekt der Klasse **Image**
und ein Objekt der Klasse **TextBlock**

In der **Grid**-Zelle (0, 0) befindet sich außerdem noch ein **ListBox**-Objekt, das zum Protokollieren von Ereignissen dient.

Das Programm beobachtet die folgenden Routingereignisse bei allen betroffenen GUI-Elementen:

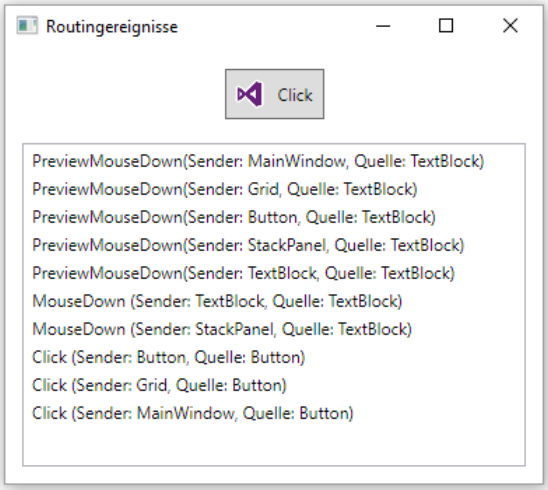
- Tunnelereignis **PreviewMouseDown** aus der Klasse **UIElement**
- Blasenereignis **MouseDown** aus der Klasse **UIElement**
- Blasenereignis **Click** aus der Klasse **ButtonBase** (Basisklasse von **Button**)

Beim folgenden Einsatz



hat das **TextBlock**-Objekt wegen eines *rechten* Mausklicks (Maustaste drücken und loslassen) das Tunnelereignis **PreviewMouseDown** gefeuert. Das Ereignis ist gemäß Abschnitt 11.4.2 mit dem Wurzelelement beginnend von allen GUI-Elementen auf dem Weg von der Wurzel bis zur Quelle behandelt worden. Danach wurde das zugehörige Blasenereignis aus der Klasse **MouseDown** ausgelöst, das den umgekehrten Weg von der Quelle bis zur Wurzel genommen hat.

Nach einem *linken* Mausklick (Maustaste drücken und loslassen) auf das **TextBlock**-Objekt zeigt sich eine andere Ereignishistorie:



Zu Beginn zeigt sich kein Unterschied zum Rechtsklick: Das vom **TextBlock**-Objekt gefeuerte Tunnelereignis **PreviewMouseDown** wird an der Wurzel beginnend auf allen Hierarchieebenen behandelt. Danach startet erwartungsgemäß das aufperlende Gegenstück **MouseDown** an der Quelle, endet allerdings überraschend beim **StackPanel**-Objekt. Stattdessen feuert das **Button**-Objekt das Blasenereignis **Click**, das seinen Weg nach oben bis zur Wurzel des GUI-Baums nimmt.

Das Programm verwendet für die drei beobachteten Routingereignisse jeweils eine Behandlungsmethode, die bei allen UI-Elementen im oben beschriebenen Baum registriert wird. Neben dem Ereignistyp protokollieren die Behandlungsmethoden auch die über den ersten Aktualparameter identifizierte Ereignisquelle. Bei einer WPF-Anwendung mit XAML-Unterstützung bringt man die Behandlungsmethoden in der **Code-Behind** - Datei mit der partiellen Definition der Fensterklasse unter:

```
using System.Windows;
using System.Windows.Input;

namespace Routingereignisse {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
        }
        // Tunnelereignis PreviewMouseDown
        private void HandlePreviewMouseDown(object sender, MouseButtonEventArgs e) {
            listBox.Items.Add($"PreviewMouseDown(Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
        // Blasenereignis MouseDown
        private void HandleMouseDown(object sender, MouseButtonEventArgs e) {
            listBox.Items.Add($"MouseDown (Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
        // Blasenereignis Click
        private void HandleClick(object sender, RoutedEventArgs e) {
            listBox.Items.Add($"Click (Sender: {sender.GetType().Name}, " +
                              $"Quelle: {e.Source.GetType().Name})");
        }
    }
}
```

In den Namen der Ereignisbehandlungsmethoden gibt der erste Bestandteil nicht wie sonst üblich den Absender des Ereignisses an, weil die Methoden für verschiedene Absender zuständig sind.

Um die Protokolleinträge vom **ListBox**-Steuerelement anzeigen zu lassen, werden sie dem zugrunde liegenden Kollektionsobjekt aus der der Klasse **ItemCollection** übergeben, das über die **ListBox**-Eigenschaft **Items** ansprechbar ist und u.a. die Methode **Add()** beherrscht.

Das Registrieren der Ereignisbehandlungsmethoden ist per XAML-Code erheblich einfacher zu bewerkstelligen als mit C# - Quellcode:

```
<Window x:Class="Routing.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Routing" Height="350" Width="525"
        PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
        ButtonBase.Click="HandleClick">
    <Grid PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
          ButtonBase.Click="HandleClick">
        <Button Name="button" HorizontalAlignment="Center" Margin="198,23,192,252"
                PreviewMouseDown="HandlePreviewMouseDown" MouseDown="HandleMouseDown"
                Click="HandleClick">
            <StackPanel Name="stackPanel" Orientation="Horizontal"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown">
                <Image  Name="image" Width="20" Margin="5"
                        Source="rss.gif" VerticalAlignment="Center"
                        PreviewMouseDown="HandlePreviewMouseDown"
                        MouseDown="HandleMouseDown"/>
                <TextBlock Name="textBlock" HorizontalAlignment="Right" Margin="5"
                           Text="Click" VerticalAlignment="Center"
                           PreviewMouseDown="HandlePreviewMouseDown"
                           MouseDown="HandleMouseDown"/>
            </StackPanel>
        </Button>
        <ListBox Name="listBox" Margin="12,76,12,12" />
    </Grid>
</Window>
```

Wie in Abschnitt 11.3.2.4 beschrieben, kann die XAML-Attributsyntax auch für Ereignisse verwendet werden, wobei der jeweilige Methodenname als Zeichenfolgenliteral anzugeben ist.

Besondere Aufmerksamkeit verdient die **Click**-Ereignis - Registrierung für das **Window**- bzw. das **Grid**-Element, z.B.:

```
<Window x:Class="Routing.MainWindow"
   . . .
        Button.Click="HandleClick">
   . . .
</Window>
```

Das **Click**-Ereignis kann beim Wurzelelement registriert werden, obwohl es in der Klasse **Window** nicht definiert ist. Dies ist möglich, weil **Click** ein Blasenereignis ist und folglich als so genanntes **angefügtes Ereignis** (engl.: *attached event*) auf übergeordneten Ebenen der UI-Elementhierarchie behandelt werden kann. Bei der Registrierung ist verständlicherweise vor dem Ereignisnamen der Name der Klasse anzugeben, zu der das Ereignis gehört. Im Beispiel hat die Klasse **Button** das Routingereignis von ihrer Basisklasse **ButtonBase** geerbt.

Die angehängten Ereignisse ermöglichen die in Abschnitt 11.4.3 beschriebene rationelle Behandlung von Blasenereignissen: Befinden sich mehrere Steuerelemente desselben Typs in einem Container lässt sich über ein angefügtes Ereignis auf Container-Ebene eine gemeinsame Ereignisbehandlung durch eine einzige Methode realisieren. In dieser Behandlungsmethode kann die Ereignisquelle über die **Source**-Eigenschaft des übergebenen **RoutedEventArgs**-Objekts identifiziert werden.

Das komplette Projekt ist im folgenden Ordner zu finden

…\BspUeb\WPF\Routingereignisse\Beobachtungsstudie

11.4.5 Ereignisbehandlung durch statische Methoden

Das WPF-Ereignissystem ermöglicht es einer von **System.Windows.DependencyObject** abstammenden Klasse, für ein Routingereignis eine statische Behandlungsmethode zu registrieren, die damit für alle Objekte der Klasse zuständig ist. Ist zusätzlich bei einem Objekt der Klasse eine Behandlungsmethode für dasselbe Routingereignis registriert, wird die statische Behandlungsmethode *vor* der objektbezogenen ausgeführt.

Zur Demonstration definieren wir eine **Button**-Ableitung namens `MaiButton`,[1] die eine statische **ClickEvent**-Behandlungsmethode definiert und beim WPF-Ereignissystem registriert:

```
using System;
using System.Windows;
using System.Windows.Controls;

class MaiButton : Button {
   protected static void StaticClickEventHandler(object sender, RoutedEventArgs e) {
      MessageBox.Show("Statischer Click-Handler der Klasse MaiButton");
   }
   static MaiButton() {
      EventManager.RegisterClassHandler(typeof(MaiButton), ClickEvent,
         new RoutedEventHandler(StaticClickEventHandler), true);
   }
}
```

Der zweite **RegisterClassHandler()** - Parameter ist vom Typ **RoutedEvent**. Unter dem Namen **ClickEvent** hat die Klasse **ButtonBase** beim WPF-Ereignissystem dasjenige Routingereignis regis-

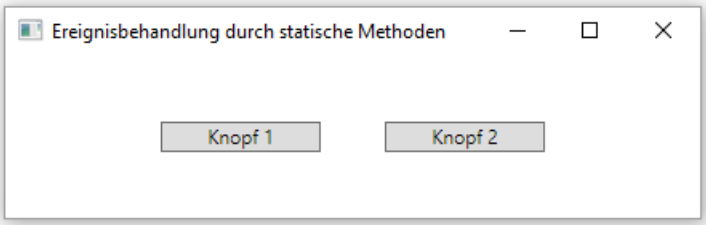
[1] Das Beispiel entstand in einem Mai.

triert, das dank Wrapper (vgl. Abschnitt 11.4.1) als CLR-Ereignis namens **Click** angesprochen werden kann.

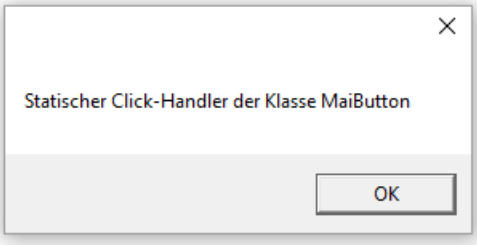
Die folgende WPF-Anwendung (handgestrickt ohne XAML)

```
class StaticEventHandling : Window {
   StaticEventHandling() {
      Height = 140; Width = 450;
      Title = "Ereignisbehandlung durch statische Methoden";
      StackPanel lm = new StackPanel();
      lm.Orientation = Orientation.Horizontal;
      lm.HorizontalAlignment = HorizontalAlignment.Center;
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;
      MaiButton k1 = new MaiButton(); k1.Content = "Knopf 1"; k1.Width = 100;
      k1.Margin = new Thickness(20); lm.Children.Add(k1);
      MaiButton k2 = new MaiButton(); k2.Content = "Knopf 2"; k2.Width = 100;
      k2.Margin = new Thickness(20); lm.Children.Add(k2);

      k1.Click += new RoutedEventHandler(knopf1Click);
   }

   private void knopf1Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf 1");
   }

   [STAThread]
   static void Main() {
      new Application().Run(new StaticEventHandling());
   }
}
```

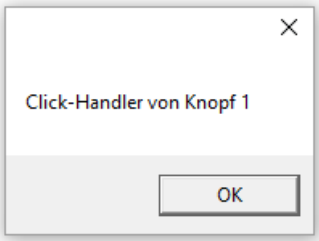
präsentiert zwei Befehlsschalter



aus der Klasse `MaiButton`, wobei der linke über eine eigene **Click**-Behandlungsmethode verfügt. Beim Klick auf einen beliebigen Schalter findet zunächst die klassenbezogene Ereignisbehandlung statt:



Wurde der linke Schalter getroffen, findet danach auch noch die objektbezogene Ereignisbehandlung statt:



11.5 Abhängigkeitseigenschaften

Die WPF-Designer haben nicht nur die CLR-Ereignisse durch die leistungsfähigeren Routingereignisse ergänzt, sondern auch zu den CLR-Ereignissen, die aus einer Instanzvariablen und einem Methodenpaar bestehen (vgl. Abschnitt 4.5), mit den Abhängigkeitseigenschaften (engl.: *dependency properties*) eine funktional erheblich erweiterte Variante geschaffen.

Wie es der Name vermuten lässt, kann die Ausprägung einer Abhängigkeitseigenschaft auf flexible Weise durch verschiedene Quellen beeinflusst werden (z.B. durch Stile, Benutzereinstellungen, Animationen, elterliche Steuerelemente). Außerdem können Ausprägungen validiert und Änderungen an andere Abhängigkeitseigenschaften gemeldet werden (Datenbindung).

Um die neue Funktionalität zu realisieren, wurde keine .NET - Programmiersprache angepasst (von XAML mal abgesehen), sondern das WPF-Framework entsprechend ausgestattet. Alle Klassen, die Abhängigkeitseigenschaften verwenden sollen, müssen von der Klasse **DependencyObject** im Namensraum **System.Windows** abstammen.

Die meisten Abhängigkeitseigenschaften können durch eine Hüllenkonstruktion auch wie gewöhnliche CLR-Eigenschaften verwendet werden. Obwohl wir in Beispielen schon etliche WPF-Abhängigkeitseigenschaften verwendet haben, ist uns bisher keine Besonderheit gegenüber gewöhnlichen CLR-Eigenschaften aufgefallen.

Ob es sich bei einer Eigenschaft einer WPF-Steuerelementklasse um eine Abhängigkeitseigenschaft handelt, verrät die Anwesenheit der **Dependency Property Information** in der FCL-Dokumentation, z.B.:

Dependency Property Information

Identifier field	ActualHeightProperty
Metadata properties set to **true**	None

Inspiziert man z.B. die über 130 Eigenschaften der Klasse **Button**, dann finden sich nur wenige, die keine Abhängigkeitseigenschaften sind.

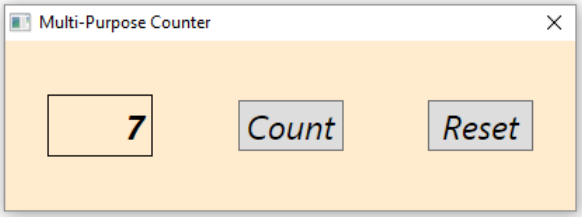
Die allgemeine Technik und Funktionsvielfalt der Abhängigkeitseigenschaften darzustellen (siehe z.B. MacDonald 2012, Natan 2010), würde vermutlich die Motivation der Leser auf eine harte Probe stellen. Ein komplettes Verständnis benötigen ohnehin nur solche Entwickler, die eigene WPF-Steuerelemente entwerfen und dabei Abhängigkeitseigenschaften anbieten wollen. Der aktuelle Abschnitt beschränkt sich auf zwei schon beim WPF-Einstieg unvermeidliche bzw. nützliche Funktionen des WPF-Eigenschaftssystems:

- Bei der Implementation einer Abhängigkeitseigenschaft kann man veranlassen, dass die an ein Element in der GUI-Hierarchie eines Fensters vergebene Ausprägung auf eingeschachtelte Elemente übertragen („vererbt“) wird. Allerdings hängt die Eigenschaftsausprägung bei einem konkreten eingeschachtelten Element eventuell noch von anderen Quellen ab.
- Mit den sogenannten *angefügten Eigenschaften* werden spezielle Abhängigkeitseigenschaften vorgestellt, die (nicht nur) bei Layoutcontainern unverzichtbar sind.

In einem späteren Kapitel über die *Datenbindung* wird eine enorm wichtige Anwendung von Abhängigkeitseigenschaften vorgestellt. Mit der Rolle dieser Technik bei der GUI-Gestaltung durch Stile und bei Animationen werden wir uns aus Zeitgründen im Kurs nicht beschäftigen.

11.5.1 Eigenschaftsvererbung an eingeschachtelte Elemente

Abhängigkeitseigenschaften können beim WPF-Eigenschaftssystem so registriert werden, dass eingeschachtelte Elemente die Ausprägung „erben“, falls keine andere Eigenschaftsquelle dominiert. Im folgenden Beispiel (vgl. Abschnitt 11.7.3 über Befehlsschalter)



werden für das Wurzelelement eine Hintergrundfarbe, eine Schriftgröße und ein Schriftstil festgelegt:

```
<Window x:Class="Button.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Multi-Purpose Counter" Height="149" Width="412" ResizeMode="NoResize"
        Background="BlanchedAlmond" FontSize="24" FontStyle="Italic">
  <Grid Button.Click="Button_Click">
    <Grid.ColumnDefinitions>
      <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <Label Name="label" Content="0" Width="75" HorizontalAlignment="Center"
           VerticalAlignment="Center" BorderThickness="1" BorderBrush="Black"
           HorizontalContentAlignment="Right" FontWeight="Bold"  />
    <Button Name="count" Content="_Count" Width="75" Grid.Column="1"
          HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
    <Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
            HorizontalAlignment="Center"  VerticalAlignment="Center">
    </Button>
  </Grid>
</Window>
```

Während das **Label**-Objekt die Hintergrundfarbe und die Schriftattribute „erbt“, übernehmen die beiden **Button**-Objekte lediglich die Schriftattribute.

Man spricht bei der Übertragung von Eigenschaftsausprägungen auf eingeschachtelte Elemente von *Vererbung*, doch hat diese Spezialität des WPF-Eigenschaftssystems *nichts* mit der in Kapitel 6 beschriebenen Ableitung von Klassen zu tun.

11.5.2 Angefügte Eigenschaften

Bei der Visual Studio - Projektvorlage **WPF-Anwendung** kommt per Voreinstellung ein **Grid**-Layoutcontainer zum Einsatz, der im allgemeinen mehrere Zeilen und/oder Spalten zur Platzierung der enthaltenen Steuerelemente verwaltet (siehe Abschnitt 11.6.1). Macht man für ein Steuerelement keine Ortsangabe, landet es in der **Grid**-Zelle (0, 0), also oben links. Um für ein Steuerelement eine alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der 0-basierten Nummer der Zeile bzw. Spalte versorgen. Hier wird ein **Button**-Objekt in die Zelle (1, 1) gesteckt:

```
<Grid>
 . . .
    <Button Name="butt" Content="Knopf"
            HorizontalAlignment="Center" VerticalAlignment="Center"
            Grid.Row="1" Grid.Column="1" />
</Grid>
```

Während sich in der Steuerelementklasse **Button** zu XAML-Attributen wie **Content**, **HorizontalAlignment** etc. jeweils eine korrespondierende Eigenschaft findet, sucht man in der Klassendefinition die Eigenschaften **Grid.Row** und **Grid.Column** oder auch **Row** und **Column** vergeblich.

Grid.Row und **Grid.Column** sind so sogenannte *angefügte Eigenschaften* (engl.: *attached properties*), die von der Klasse **Grid** zur Verfügung gestellt werden, damit die im Layoutcontainer verwalteten UI-Elemente ihre Positionswünsche formulieren können.

Bei einer angefügten Eigenschaft handelt es sich um eine Abhängigkeitseigenschaft mit den folgenden Besonderheiten:

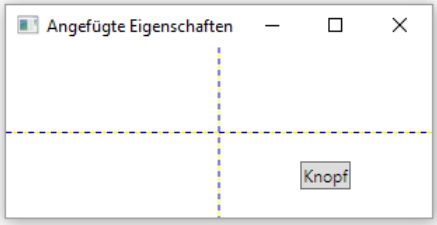
- Bei der Eigenschaftsvergabe ist kein Objekt der definierenden Klasse (im Beispiel: **Grid**) betroffen, sondern ein Objekt einer anderen Klasse (im Beispiel: **Button**).
- Daher ist bei einer angefügten Eigenschaft *keine* Hüllenkonstruktion vorhanden, um die Verwendung wie bei einer CLR-Eigenschaft zu ermöglichen. Stattdessen sind statische **Set**- und **Get**-Methoden vorhanden, was gleich für das Beispiel demonstriert wird.

Im Beispiel stecken hinter der bequemen XAML-Syntax die statischen **Grid**-Methoden **SetRow()** und **SetColumn()**, die als Parameter jeweils ein Objekt der indirekt von **DependencyObject** abstammenden Klasse **UIElement** und eine Positionsangabe (Zeilen- oder Spaltennummer) erwarten. Im folgenden Fensterklassenkonstruktor wird *ohne* XAML-Hilfe ein (2 × 2) - **Grid** erstellt und ein **Button**-Objekt in die Zelle (1, 1) gesteckt, um die Rolle der statischen **Grid**-Methoden **SetRow()** und **SetColumn()** zu demonstrieren:

```
using System;
using System.Windows;
using System.Windows.Controls;

class AttachedProperties : Window {
   AttachedProperties() {
      Title = "Angefügte Eigenschaften";
      Grid grid = new Grid();
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      grid.ColumnDefinitions.Add(new ColumnDefinition());
      grid.RowDefinitions.Add(new RowDefinition());
      grid.RowDefinitions.Add(new RowDefinition());
      grid.ShowGridLines = true;
      Content = grid;
```

```
        Button butt = new Button();
        butt.Content = "Knopf";
        butt.HorizontalAlignment = HorizontalAlignment.Center;
        butt.VerticalAlignment = VerticalAlignment.Center;
        Grid.SetRow(butt, 1);
        Grid.SetColumn(butt, 1);

        grid.Children.Add(butt);
    }

    [STAThread]
    static void Main() {
        Application app = new Application();
        AttachedProperties hf = new AttachedProperties();
        app.Run(hf);
    }
}
```

Das **Button**-Objekt erscheint in der gewünschten Zelle des **Grid**-Layoutcontainers:



Damit die korrekte Positionierung gut erkennbar ist, wurde über die Eigenschaft **ShowGridLines** die Zellenstruktur des **Grid**-Layoutcontainers sichtbar gemacht:

```
grid.ShowGridLines = true;
```

Als Gegenstücke zu den schreibenden Methoden **Grid.SetRow()** und **Grid.SetClumn()** sind die lesenden Methoden **Grid.GetRow()** und **Grid.GetClumn()** vorhanden.

Die zu einer angefügten Eigenschaft gehörige statische Schreib- bzw. Lesemethode (z.B. **Grid.SetRow()** bzw. **Grid.GetRow()**) ruft Ihrerseits beim betroffenen Objekt die von der Klasse **DependencyObject** geerbte Instanzmethode **SetValue()** bzw. **GetValue()** auf, wobei die angefügte Eigenschaft als erster Parameter übergeben wird, z.B.:[1]

```
public static void SetRow(UIElement element, int value) {
    if (element == null) {
        throw new ArgumentNullException("element");
    }
    element.SetValue(RowProperty, value);
}
```

Weil die Methoden **SetValue()** und **GetValue()** öffentlich sind, kann man sie auch direkt aufrufen und somit die statischen Methoden der angefügten Eigenschaft übergehen. Im Beispiel kann man die Anweisungen

```
Grid.SetRow(butt, 1);
Grid.SetColumn(butt, 1);
```

ersetzen durch:

```
butt.SetValue(Grid.RowProperty, 1);
butt.SetValue(Grid.ColumnProperty, 1);
```

---

[1] Der Quellcode stammt aus der Datei **Grid.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann.

11.6 Layoutcontainer

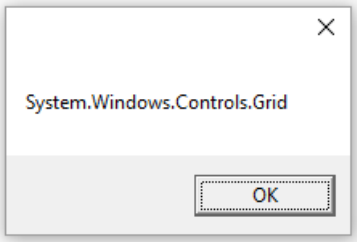
Die in unseren WPF-Anwendungen definierten Fenster besitzen mehr oder weniger viele Steuerelemente. Zur Verwaltung der Steuerelemente (z.B. Neuberechnung von Positionen und Größen bei einer Änderung der Fenstergröße) dient ein Layoutcontainer. Bei einer neuen WPF-Anwendung wird vom Visual Studio für das Hauptfenster ein Layoutcontainer aus der Klasse **Grid** (Namensraum **System.Windows.Controls**) vorgeschlagen. Dies zeigt ein Blick auf den XAML-Code des Hauptfensters:[1]

```
<Window x:Class="LayoutContainer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>

    </Grid>
</Window>
```

Dass dieses Objekt dem Hauptfenster über dessen **Content**-Eigenschaft zugewiesen wird, lässt sich z.B. über eine Behandlungsmethode zum **Window**-Ereignis **Loaded**

```
using System.Windows;
namespace LayoutContainer {
    public partial class MainWindow : Window {
        public MainWindow() {
            InitializeComponent();
            Loaded += Window_Loaded;
        }
        private void Window_Loaded(object sender, RoutedEventArgs e) {
            MessageBox.Show(Content.ToString());
        }
    }
}
```

nachweisen:



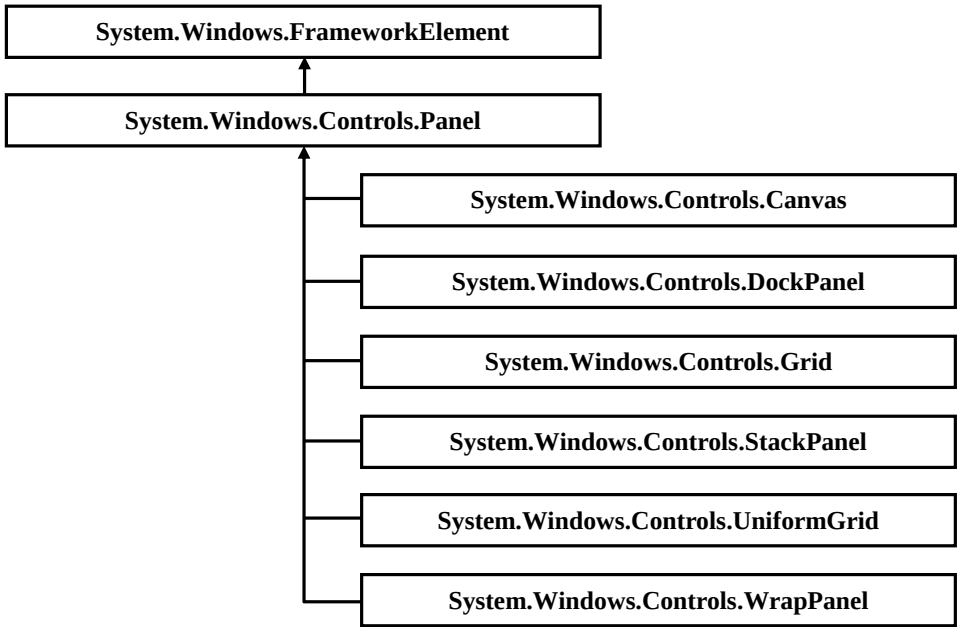
Wenn wir im WPF-Designer Steuerelemente per Drag & Drop aus der Toolbox-Palette auf das Hauptfenster platzieren, landen diese im **Grid**-Element, z.B.:

```
<Window x:Class="LayoutContainer.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="MainWindow" Height="350" Width="525">
    <Grid>
        <Label x:Name="label" Content="Label" HorizontalAlignment="Left"
         Margin="30,30,0,0" VerticalAlignment="Top"/>
    </Grid>
</Window>
```

[1] Der Übersichtlichkeit halber wurden überflüssige XAML-Namensbereichsdeklarationen entfernt.

Von der Gitterstruktur, die der Klassennamen **Grid** erwarten lässt, ist noch nichts zu sehen. Es ist nur eine Zeile und eine Spalte vorhanden. Eingefügte Steuerelemente landen in der **Grid**-Zelle (0, 0). Wie man die **Grid**-Flexibilität nutzt, erfahren Sie ansatzweise in Abschnitt 11.6.1.

Alle WPF-Layoutcontainer stammen von der Klasse **System.Windows.Controls.Panel** ab. Hier sind die meistbenutzten Klassen zu sehen:



Von den zahlreichen Membern, die alle Layoutcontainer von ihrer Basisklasse **Panel** erben, ist besonders die Eigenschaft **Children** zu erwähnen, die auf ein Objekt der Klasse **UIElementCollection** zeigt, das die im Container enthaltenen **UIElement**-Objekte verwaltet.

Für die Top-Level - Fenster einer WPF-Anwendung verwendet man in der Regel ein **Grid**- oder ein **DockPanel**-Objekt als Layoutcontainer. Ein flexibles Fensterdesign erfordert oft geschachtelte Layoutcontainer, und dabei finden auch die übrigen Klassen Verwendung.

11.6.1 Grid

In diesem Abschnitt werden einige Details zum voreingestellten Layoutcontainer einer WPF-Anwendung geschrieben.

11.6.1.1 Zeilen und Spalten definieren

Um die Zeilen bzw. Spalten eines **Grid**-Objekts per XAML zu definieren, ergänzt man im **Grid**-Element untergeordnete Eigenschaftselemente vom Typ **Grid.RowDefinitions** bzw. **Grid.ColumnDefinitions** (vgl. Abschnitt 11.3.2.3 zur XAML-Syntax für Eigenschaftselemente).

Im Element **Grid.RowDefinitions** werden schließlich einzelne Gitterzeilen durch **RowDefinition**-Elemente vereinbart, z.B.:

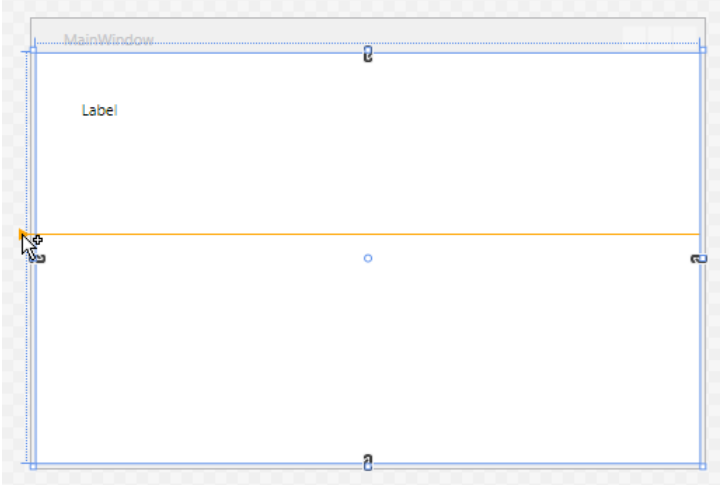
```
<Grid>
    <Grid.RowDefinitions>
        <RowDefinition/>
        <RowDefinition/>
    </Grid.RowDefinitions>
      .  .  .
</Grid>
```

Die **Grid**-Eigenschaft **RowDefinitions** ist vom Typ **System.Windows.Controls.RowDefinitionCollection**, und die XAML-Kollektionssyntax (vgl. Abschnitt 11.3.2.3.4) erlaubt es in diesem Fall, auf ein Instanzelement zum Kollektionsobjekt zu verzichten.

Die XAML-Elemente zur Definition der Zeilenstruktur kann man auch über den WPF-Designer unserer Entwicklungsumgebung erstellen. Markieren Sie zunächst das **Grid**-Element mit einer von den folgenden Techniken:

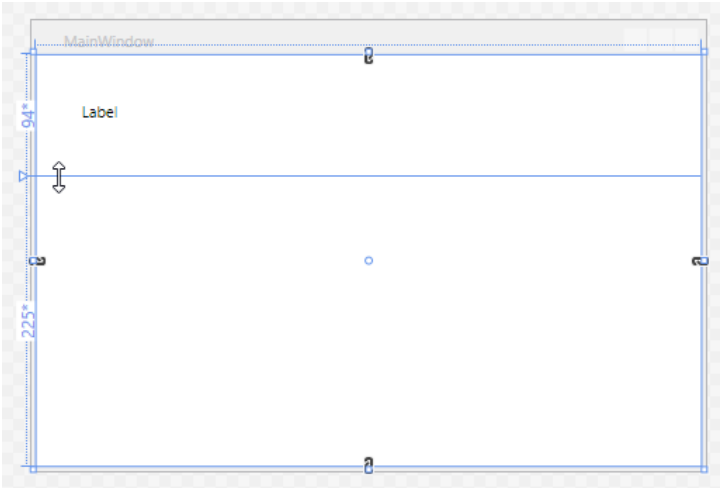
- Mausklick auf das Innere des Fensters
- Mausklick auf das **Grid**-Element im XAML-Fenster
- **Dokumentengliederung** öffnen (z.B. mit **Ansicht > Weitere Fenster > Dokumentengliederung**) und auf das **Grid**-Objekt klicken

Setzen Sie dann per Mausklick auf die am linken Rand des **Grid**-Elements befindliche Zeilendefinitionszone eine Trennlinie:



Dabei wird automatisch im XAML-Code die Liste der **RowDefinition**-Elemente erweitert und nötigenfalls auch ein Element von Typ **Grid.RowDefinitions** erstellt.

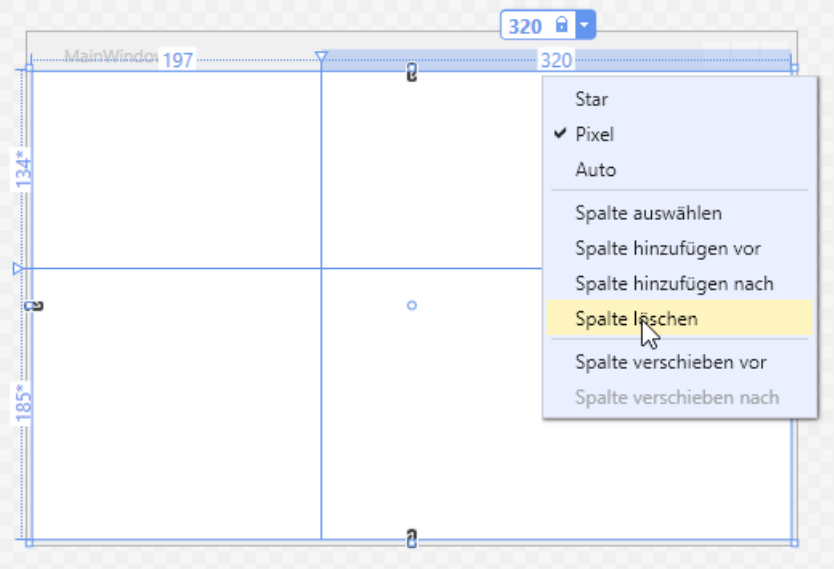
Die Trennlinie kann anschließend per Maus verschoben werden:



Analog lässt sich über die am oberen Rand des **Grid**-Elements befindliche Spaltendefinitionszone die Spaltenstruktur vereinbaren, z.B.:

```
<Grid>
    <Grid.ColumnDefinitions>
        <ColumnDefinition Width="210*" />
        <ColumnDefinition Width="293*" />
    </Grid.ColumnDefinitions>
      .  .  .
</Grid>
```

Wenn man den Mauszeiger links neben eine Zeile oder über eine Spalte positioniert, erscheint ein Werkzeug zur Modifikation von Größe und Position, z.B.:



Über das Symbol in der Mitte wählt man das Verfahren zur Größenbestimmung:

- Angabe einer Pixelzahl:
- Angabe eines Anteils an der Ausdehnung des **Grid**-Containers:
- Automatische Größenberechnung aufgrund des Inhalts:

Der links neben dem Symbol anzeigte, als Pixelzahl oder Anteil zu interpretierende aktuelle Wert kann nach einem Mausklick geändert werden.

Über den Pfeil rechts neben dem Symbol erhält man ein Kontextmenü, das z.B. das Löschen der aktuellen Zeile oder Spalte erlaubt.

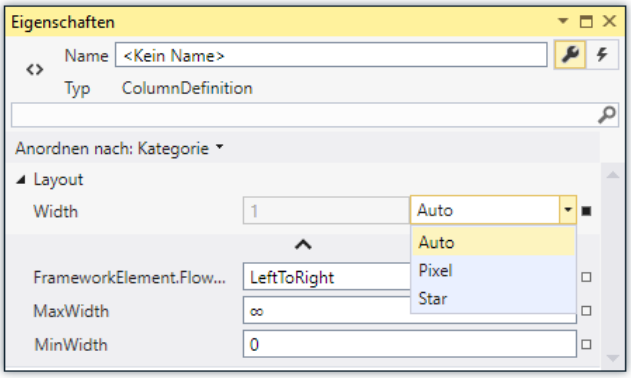
Anschließend wird beschrieben, wie man die Verteilung des Platzes auf Zeilen und Spalten per XAML-Code vornimmt.

11.6.1.2 Platzaufteilung

Die Breite einer Spalte bzw. die Höhe einer Zeile lässt sich per **Width**- bzw. **Height**-Attribut festlegen, wobei folgende Alternativen zur Verfügung stehen:

- Bei einer fehlenden Angabe oder bei der Anforderung
  ```
  <ColumnDefinition Width="*" />
  ```
  beansprucht die Spalte bzw. Zeile den gesamten noch nicht vergebenen Platz. Verhalten sich alle Konkurrenten so, wird der verfügbare Platz gleichmäßig aufgeteilt.
  Über Faktoren für die Sternangabe lässt sich ein mehrfacher Platzbedarf anmelden. So wird z.B. für die beiden folgenden Spalten der verfügbare Platz im Verhältnis 1:2 aufgeteilt:
  ```
  <ColumnDefinition Width="*" />
  <ColumnDefinition Width="2*" />
  ```
  Ein isolierter Stern (siehe erste Spaltendefinition) hat implizit den Faktor 1.
- Besitzen alle Spalten bzw. Zeilen einen festen (nicht als Anteil zu verstehenden) Wert in Pixeln, werden die angeforderten Pixel ausgegeben, solange der Vorrat reicht, so dass eventuell eine hintere Spalte bzw. Zeile komplett verschwindet.
- Vergibt man den Wert **Auto**, dann orientiert sich die Breite einer Spalte bzw. die Höhe einer Zeile am maximalen Platzbedarf der enthaltenen UI-Elemente.

Statt den **Width**- bzw. **Height**-Wert im XAML-Fenster einzutragen, kann man bei passender Markierung auch das Eigenschaftenfenster benutzen, z.B.

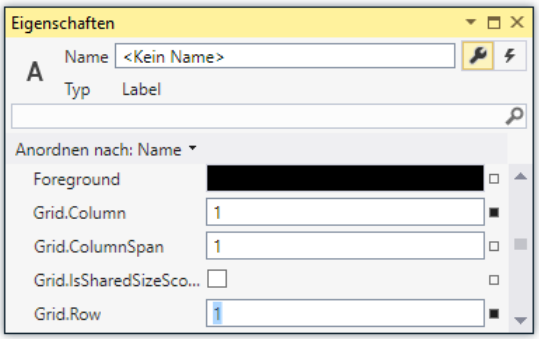


11.6.1.3 Platzanweisung

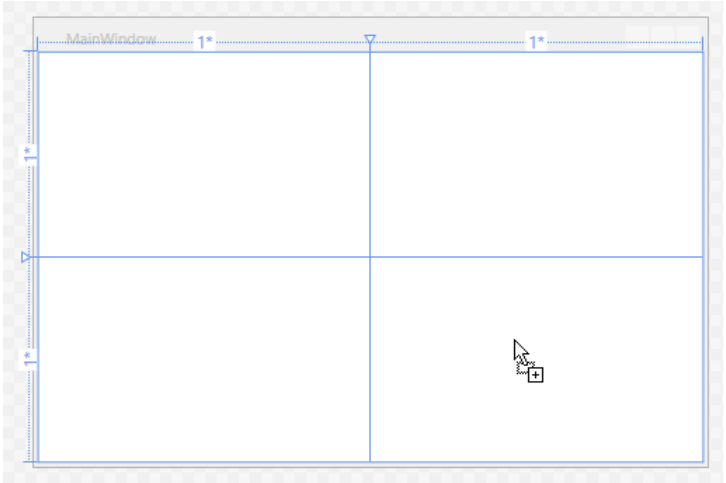
Besitzt ein per **Grid**-Container verwaltetes Steuerelement keine Ortsangabe, landet es in der **Grid**-Zelle (0, 0), also oben links. Um für ein Steuerelement eine alternative Zelle festzulegen, sind im zugehörigen XAML-Element die Attribute **Grid.Row** bzw. **Grid.Column** mit der 0-basierten Nummer der Zeile bzw. Spalte versorgen. Hier wird ein **Label**-Objekt in die Zelle (1, 1) gesetzt:

```
<Grid>
  . . .
    <Label Content="Label" Height="28" HorizontalAlignment="Left"
           Margin="44,40,0,0" Name="label1" VerticalAlignment="Top"
           Grid.Column="1" Grid.Row="1" />
</Grid>
```

Natürlich kann man auch das Eigenschaftenfenster zur Platzanweisung benutzen:



Das bequemste Verfahren zur Wahl einer **Grid**-Zelle bietet der WPF-Designer: Steuerelement per Maus packen und am Ziel ablegen, z.B.:



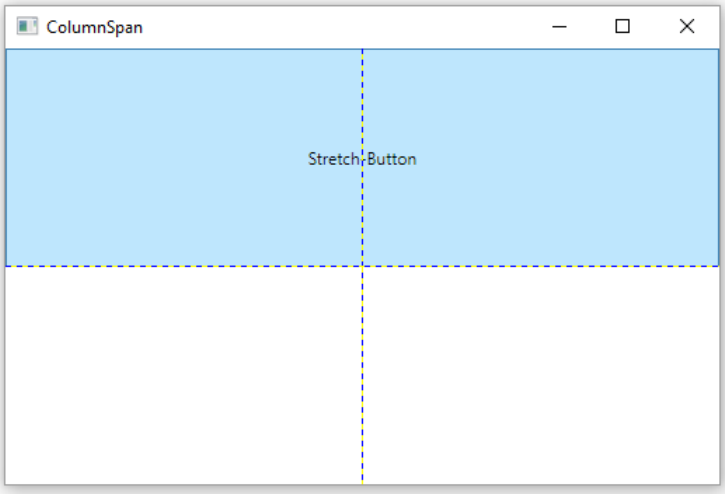
Bei **Grid.Row** bzw. **Grid.Column** handelt es sich um angefügte Eigenschaften (siehe Abschnitt 11.5.2).

11.6.1.4 Mehrzellige Elemente

Es ist nicht festgeschrieben, dass ein Steuerelement genau *eine* Zelle im **Grid**-Container belegen darf. Mit Hilfe der angefügten Eigenschaften **Grid.RowSpan** sowie **Grid.ColumnSpan** kann ein Steuerelement mehrere Zeilen und/oder Spalten beanspruchen. Mit dem folgenden XAML-Code wird ein (2 × 2) - **Grid** - Container definiert, wobei sich ein **Button**-Objekt über die beiden Zellen in der oberen Zeile erstreckt:

```
<Window x:Class="WpfApplication1.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ColumnSpan" Height="350" Width="525">
    <Grid ShowGridLines="True">
        <Grid.ColumnDefinitions>
            <ColumnDefinition /> <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Grid.RowDefinitions>
            <RowDefinition /> <RowDefinition />
        </Grid.RowDefinitions>
        <Button Content="Stretch-Button" Grid.ColumnSpan="2" />
    </Grid>
</Window>
```

Über den Wert **true** für die **Grid**-Eigenschaft **ShowGridLines** wird dafür gesorgt, dass die (2 × 2) - Struktur des Containers optisch präsent ist:[1]



Dass sich mehrere Steuerelemente eine **Grid**-Zelle teilen können, haben Sie schon wiederholt beobachtet (z.B. in Abschnitt 11.4.4). Bei einer neuen WPF-Anwendung im Sinne der entsprechenden Projektvorlage unserer Entwicklungsumgebung hat das Hauptfenster zunächst einen einzelligen **Grid**-Container, und alle aus der Toolbox per Drag & Drop übernommenen Steuerelemente landen in der einzigen Zelle. Sofern die Größen und Verankerungen der Elemente geschickt gewählt werden, resultiert ein sinnvoll nutzbares Fenster. Ein komplexes Layout in einem Fenster mit variabler Größe ergonomisch zu gestalten, ist aber mit (verschachtelten) Layoutcontainern einfacher.

---

[1] Bei der Werten **true** und **false** für Eigenschaften von Typ **bool** ist in XAML die Groß-/Kleinschreibung ausnahmsweise irrelevant.

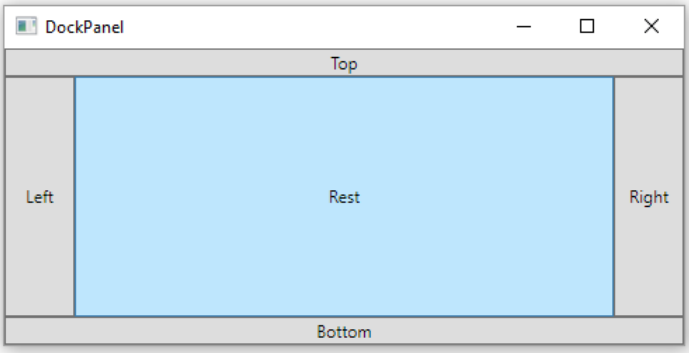
11.6.2 DockPanel

Die Layoutcontainer-Klasse **DockPanel** bietet den verwalteten Steuerelementen über die angefügte Eigenschaft **Dock** mit den Werten **Left**, **Top**, **Right**, **Bottom** die Möglichkeit, sich an einer Seite festzusetzen. Wollen mehrere Elemente eine Seite besetzen, werden sie dort in der Beitrittsreihenfolge gestapelt. Die Steuerelemente erhalten nach Möglichkeit ihre gewünschte Größe, und das zuletzt eingefügte Element erhält den kompletten noch freien Platz.[1]

Bei der folgenden Fensterklassendeklaration

```
<Window x:Class="DockPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="ColumnSpan" Height="250" Width="500">
    <DockPanel>
        <Button DockPanel.Dock="Top">Top</Button>
        <Button DockPanel.Dock="Bottom">Bottom</Button>
        <Button DockPanel.Dock="Left" Width="50">Left</Button>
        <Button DockPanel.Dock="Right" Width="50">Right</Button>
        <Button>Rest</Button>
    </DockPanel>
</Window>
```

sind alle Seiten durch **Button**-Objekte mit **Dock**-Angabe besetzt, und ein fünftes **Button**-Objekt *ohne* **Dock**-Angabe belegt den frei gebliebenen Raum im Zentrum:



Die **DockPanel**-Struktur bietet ein für viele Programme geeignetes UI-Gerüst, z.B. für einen Editor:

- Oben werden das Menü und eine Symbolleiste untergebracht.
- Links befindet sich eine Navigationszone.
- In der Mitte wird das Dokument angezeigt und editiert.
- Rechts befindet sich eine Werkzeugsammlung.
- Unten befindet sich eine Statuszeile.

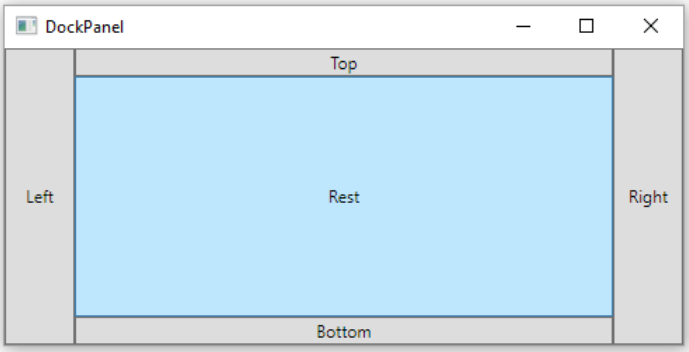
Ob die vier Ecken von den horizontalen oder den vertikalen Steuerelementen eingenommen werden, hängt von der Aufnahmereihenfolge ab. Aus der Layoutdefinition

```
<DockPanel>
    <Button DockPanel.Dock="Left" Width="50">Left</Button>
    <Button DockPanel.Dock="Right" Width="50">Right</Button>
    <Button DockPanel.Dock="Top">Top</Button>
    <Button DockPanel.Dock="Bottom">Bottom</Button>
    <Button>Rest</Button>
</DockPanel>
```

folgt diese Anordnung:

---

[1] Wer sich an das **BorderLayout** im traditionsreichen GUI-Framework **Swing** der Programmiersprache Java erinnert fühlt, liegt genau richtig.



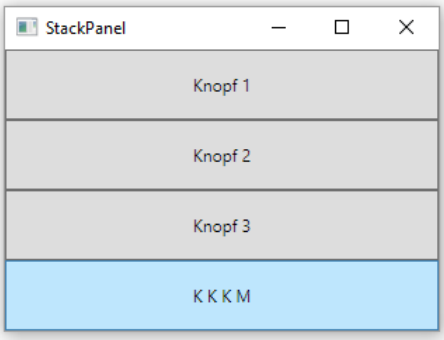
11.6.3 StackPanel

Mit einem Layoutcontainer der Klasse **StackPanel** realisiert man einen simplen vertikalen oder horizontalen Stapel von Steuerelementen. Die Elemente erhalten die gewünschten Ausdehnungen, solange der Vorrat an Pixeln reicht. Ist beim vertikalen Stapel die Gesamthöhe bzw. beim horizontalen Stapel die Gesamtbreite unzureichend, können die zuletzt eingefügten Elemente nicht mehr wunschgemäß versorgt werden.

Durch die folgende XAML-Deklaration

```
<Window x:Class="StackPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        SizeToContent="Height" Width="325" Title="StackPanel">
    <StackPanel>
        <Button Height="50">Knopf 1</Button>
        <Button Height="50">Knopf 2</Button>
        <Button Height="50">Knopf 3</Button>
        <Button Height="50">K K K M</Button>
    </StackPanel>
</Window>
```

werden vier **Button**-Objekte übereinander gestapelt:



Weil die **Width**-Eigenschaften der Schalter keinen Wert erhalten, gilt die Voreinstellung **Stretch** (vgl. Abschnitt 11.7.2.2).

Im **Window**-Element sorgt die Eigenschaftszuweisung

```
SizeToContent="Height"
```

dafür, die Fensterhöhe an den vom **StackPanel** benötigten Platz anzupassen.

Um einen *horizontalen* Stapel zu erhalten, setzt man die **StackPanel**-Eigenschaft **Orientation** auf den Wert **Horizontal**:

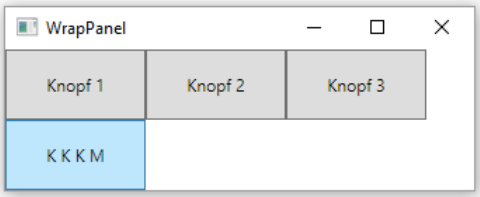
```
<StackPanel Orientation="Horizontal">
  . . .
</StackPanel>
```

11.6.4 WrapPanel

Beim **WrapPanel** kommt im Vergleich zum **StackPanel** ein automatischer Spalten- bzw. Zeilenumbruch hinzu. Durch die folgende XAML-Deklaration

```
<Window x:Class="WrapPanel.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="WrapPanel" SizeToContent="Height" Width="350">
    <WrapPanel Orientation="Horizontal">
        <Button Width="100" Height="50">Knopf 1</Button>
        <Button Width="100" Height="50">Knopf 2</Button>
        <Button Width="100" Height="50">Knopf 3</Button>
        <Button Width="100" Height="50">K K K M</Button>
    </WrapPanel>
</Window>
```

werden vier **Button**-Objekte nebeneinander gestapelt, bis der Platz erschöpft und daher ein Zeilenumbruch erforderlich ist:



11.6.5 UniformGrid

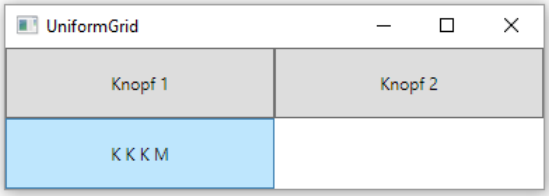
Im Vergleich zum **Grid** - Layoutcontainer bestehen beim **UniformGrid**-Container folgende Unterschiede:

- Die Spalten- und Zeilenzahl sind gleich.
- Diese gemeinsame Zahl wird nach Bedarf ermittelt.
  Ist die Zahl der Elemente z.B. größer als 4 und kleiner als 10, erhält man einen (3 × 3) - Layoutcontainer.

Aus der folgenden XAML-Deklaration

```
<Window x:Class="UniformGrid.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="UniformGrid" SizeToContent="Height" Width="400">
    <UniformGrid>
        <Button Height="50">Knopf 1</Button>
        <Button Height="50">Knopf 2</Button>
        <Button Height="50">K K K M</Button>
    </UniformGrid>
</Window>
```

resultiert ein Container mit (2 × 2) gleich großen Zellen. Die drei Elemente werden nacheinander auf die Zellen verteilt, wobei der Spaltenindex schneller läuft:



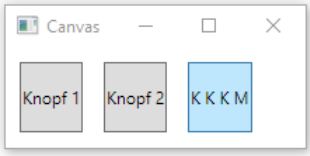
11.6.6 Canvas

Der **Canvas**-Container stellt eine Zeichenfläche ohne jede Layout-Logik bei Größenänderungen zur Verfügung und sollte nur in sehr speziellen Situationen eingesetzt werden. Immerhin lässt sich für die verwalteten Elemente über die angefügten Eigenschaften **Canvas.Left** und **Canvas.Top** ein Abstand zum linken bzw. oberen Rand angeben. Wer unbedingt will, kann mit diesem Container nach schlechter alter Sitte Steuerelemente in liebevoller Kleinarbeit montieren.

Aus der folgenden XAML-Deklaration

```
<Window x:Class="Canvas.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Canvas" Height="110" Width="230">
    <Canvas>
        <Button Height="50" Canvas.Left="10" Canvas.Top="10">Knopf 1</Button>
        <Button Height="50" Canvas.Left="70" Canvas.Top="10">Knopf 2</Button>
        <Button Height="50" Canvas.Left="130" Canvas.Top="10">K K K M</Button>
    </Canvas>
</Window>
```

resultiert das Fenster:



11.6.7 Geschachtelte Layoutcontainer

Sollen z.B. in die Zelle (0, 0) eines **Grid**-Containers drei **Button**-Objekte (Befehlsschalter) übereinander positioniert werden, fügt man in diese Zelle zunächst einen Layoutcontainer der Klasse **StackPanel** ein, z.B.:

```
<Window x:Class="Geschachtelte_Layoutcontainer.MainWindow"
    xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
    xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
    Title="Geschachtelte Layoutcontainer" SizeToContent="Height" Width="400">
  <Grid ShowGridLines="True">
    <Grid.ColumnDefinitions>
        <ColumnDefinition /><ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
                Height="Auto" Width="Auto" Name="stackPanel" Margin="10">
    </StackPanel>
  </Grid>
</Window>
```

Weil im **StackPanel**-Element die angefügten Eigenschaften **Grid.Row** und **Grid.Column** fehlen, wird jeweils der Wert 0 angenommen.

Das **StackPanel**-Objekt soll ...

- sich in seiner Zelle horizontal und vertikal zur Mitte hin orientieren,
- seine Breite und Höhe automatisch an den enthaltenen Steuerelementen orientieren,
- zur Umgebung einen allseitigen Abstand von 10 einhalten.

Es werden drei Button-Objekte in den **StackPanel**-Container eingefügt:

```
<StackPanel HorizontalAlignment="Center" VerticalAlignment="Center"
            Height="Auto" Width="Auto" Name="stackPanel1" >
    <Button Content="Button 1" Height="25" Name="button1" Width="75" Margin="5" />
    <Button Content="Button 2" Height="25" Name="button2" Width="75" Margin="5" />
    <Button Content="Button 3" Height="25" Name="button3" Width="75" Margin="5" />
</StackPanel>
```

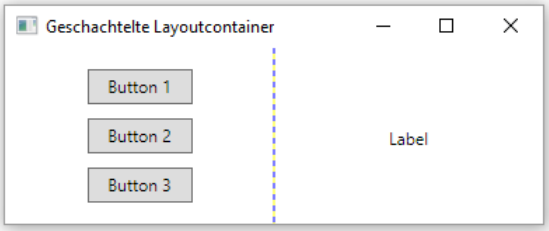
Die **Button**-Objekte sind ...

- 75 Pixel breit und 25 Pixel hoch,
- auf allen Seiten von einem freien Streifen mit 5 Pixeln Breite umgeben.

Außerdem wird ein **Label**-Objekt in die Zelle (0, 1) des **Grid**-Containers eingefügt:

```
<Label Content="Label" Grid.Column="1" Height="23"
       HorizontalAlignment="Center" VerticalAlignment="Center" Name="label1" />
```

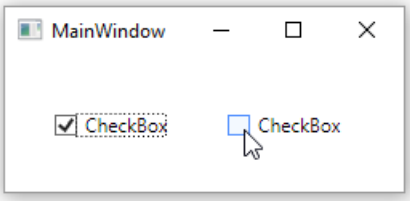
Das Ergebnis:



11.7 Basiswissen über Steuerelemente

Die WPF-Bibliothek bietet zahlreiche Klassen zur Realisation von Steuerelementen (Schaltflächen, Textfeldern, Kontrollkästchen, Listen etc.) an, so dass sich für zahllose Programmieraufgaben auf recht bequeme Weise ergonomische und attraktive Bedienoberflächen erstellen lassen. Als Besonderheiten dieser Klassen (z.B. im Vergleich zu Klassen wie **String** oder **Math**) sind zu nennen:

- Ihre Objekte können **auf dem Bildschirm auftreten** und dabei selbständig **mit dem Benutzer interagieren**. Wenn wir z.B. ein Kontrollkästchen in ein Fenster einbauen, erscheint bzw. verschwindet bei einem Mausklick die Markierung,

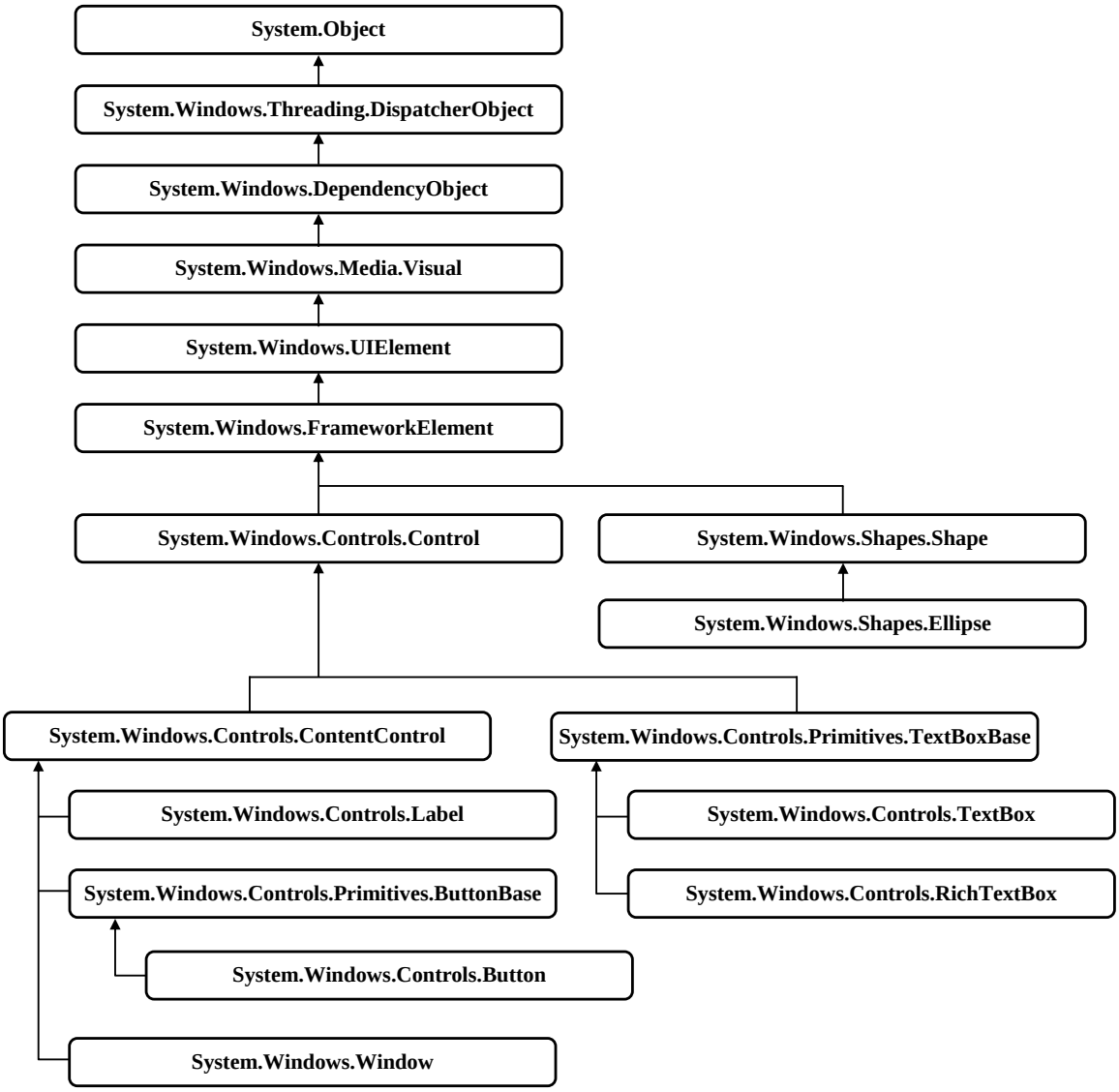


  ohne dass wir uns um diese Anpassung der Optik kümmern müssten.
- Die Steuerelemente kommunizieren über **Ereignisse** (im Sinn der Abschnitte 9.2 und 11.4) mit anderen Klassen. Will man z.B. über die gesetzte Markierung bei einem Kontrollkästchen informiert werden, registriert man eine Behandlungsmethode bei seinem **Checked**-Ereignis.

- Ihre Eigenschaften (z.B. **Text**, **Height**, **HorizontalAlignment**, **Focusable**) können zur Entwurfszeit über Werzeuge der Entwicklungsumgebungen konfiguriert werden (siehe z.B. Abschnitt 4.11.5).

11.7.1 Abstammungsverhältnisse

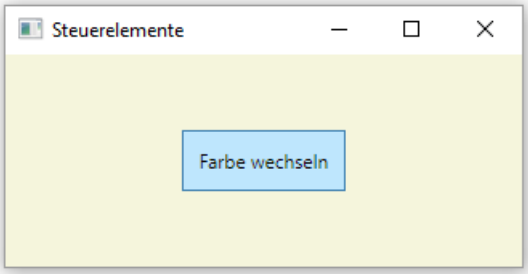
In der .NET - Klassenhierarchie stammen die Steuerelemente (**Button**, **TextBox** etc.) von der Basisklasse **System.Windows.Controls.Control** ab, während die zur optischen Gestaltung eingesetzten und nicht interaktionsfähigen Grafikelemente (**Image**, **Ellipse** etc.) von der Basisklasse **System.Windows.Shapes.Shape** abstammen:



Die von **System.Windows.Controls.ContentControl** abstammenden Klassen beherrschen das WPF-Inhaltsmodell, können also untergeordnete Elemente aufnehmen.

11.7.2 Verwendung

Wir beschäftigen uns zunächst anhand eines sehr einfachen Beispielprogramms damit,



wie man ...

- ein Steuerelement als Memberobjekt in eine Fensterklasse aufnimmt,
- seine Position auf dem Fenster festlegt,
- seine Eigenschaften zur Entwurfszeit bestimmt,
- und Behandlungsmethoden bei seinen Ereignissen registriert.

11.7.2.1 Instanzieren

Wird mit dem WPF-Designer im Visual Studio ein Steuerelement aus der Toolbox auf ein Fenster gezogen, landet es in einem Layoutcontainer. Bei einer neuen WPF-Anwendung ist für das Hauptfenster ein **Grid**-Container zuständig, der sich vorläufig auf die Zelle (0, 0) beschränkt (also nur eine Zeile bzw. Spalte hat). Im XAML-Code zur Fensterklasse erscheint im **Grid**-Element ein Eintrag für das neue Steuerelement, z.B.:

```
<Grid>
    <Button Content="Button" Height="23" HorizontalAlignment="Left"
            Margin="96,48,0,0" VerticalAlignment="Top" Width="75"
            Name="button" />
</Grid>
```

Dem XAML-Code ist auch zu entnehmen,

- dass ein Objekt der Klasse **Button** entsteht,
- dass dieses Objekt über den Referenzvariablennamen `button` ansprechbar ist.
  In einer ernsthaften Anwendung werden Sie die vorgegebenen Namen vermutlich durch individuelle und aussagekräftige Alternativen ersetzen.

Wir haben in Abschnitt 11.3.4 erfahren, dass ...

- bei der Programmerstellung aus der XAML-Datei eine Binär-Variante mit der Namenserweiterung **baml** entsteht,
- beim Programmstart die **baml**-Datei analysiert und im Beispiel das dort beschriebene **Button**-Objekt erzeugt wird.

11.7.2.2 Eigenschaften

Durch die folgenden Eigenschaften eines Steuerelements werden seine Ausdehnung sowie seine Position im umgebenden Container bzw. in der umgebenden Containerzelle geregelt. Sie sind schon in der Klasse **System.Windows.FrameworkElement** definiert, also auch bei Layoutcontainern und Grafikelementen anwendbar:

- **Width**, **Height**
  Mit den Eigenschaften **Width** und **Height** wählt man eine Breite und eine Höhe. Man kann jeweils neben einer Pixelzahl auch den Wert **Auto** angeben. Dann wird die Breite bzw. Höhe passend zum Inhalt und zum angeforderten Innenrand (siehe unten) gewählt.
- **Margin**
  Mit dieser Eigenschaft legt man fest, wie viel Platz um das Element herum (bis zum Rand der umgebenden Containerzelle oder bis zum Nachbarn) frei bleiben soll, wobei unterschiedlich detaillierte Angaben möglich sind:
  - mit *einer* Zahl ...
    legt man für alle Seiten denselben Rand fest
  - von *zwei* Zahlen ...
    legt die erste die die beiden horizontalen und die zweite die beiden vertikalen Ränder fest
  - *vier* Zahlen ...
    beziehen sich auf den linken, oberen, rechten und den unteren Rand.
- **HorizontalAlignment**
  Diese Eigenschaft bestimmt die horizontale Ausrichtung. Es sind vier Werte (aus der Enumeration **HorizontalAlignment** im Namensraum **System.Windows**) möglich:
  - **Left**
    Das Element ist am linken Rand verankert.
  - **Right**
    Das Element ist am rechten Rand verankert.
  - **Stretch**
    Das Element ist am linken *und* am rechten Rand verankert, so dass es seine horizontale Ausdehnung zusammen mit der umgebenden Containerzelle verändert. Besitzt die **Width**-Eigenschaft einen expliziten Wert, wird der **HorizontalAlignment**-Wert **Stretch** ignoriert.
  - **Center**
    Das Element wird horizontal zentriert.
- **VerticalAlignment**
  Diese Eigenschaft bestimmt die vertikale Ausrichtung. Es sind vier Werte (aus der Enumeration **VerticalAlignment** im Namensraum **System.Windows**) möglich:
  - **Top**
    Das Element ist am oberen Rand verankert.
  - **Bottom**
    Das Element ist am unteren Rand verankert.
  - **Stretch**
    Das Element ist am oberen *und* am unteren Rand verankert, so dass es seine vertikale Ausdehnung zusammen mit der umgebenden Containerzelle verändert. Besitzt die **Height**-Eigenschaft einen expliziten Wert, wird der **VerticalAlignment**-Wert **Stretch** ignoriert.
  - **Center**
    Das Element wird vertikal zentriert.

Welche Werte für die Eigenschaften **HorizontalAlignment** und **VerticalAlignment** realisierbar sind, hängt vom Typ des Layoutcontainers ab, z.B. ...

- haben die Elemente in einem **Grid**- oder **UniformGrid**-Container per Voreinstellung die horizontale Ausrichtung **Stretch** und können alternative Ausrichtungen erhalten,
- sind die Elemente in einem horizontal orientierten **StackPanel**- oder **WrapPanel**-Container grundsätzlich von links nach rechts angeordnet.

Es folgen einige Eigenschaften für Steuerelemente im engeren Sinn (für Objekte der Klasse **System.Windows.Controls.Control**):

- **Content**
  Bei vielen Steuerelementen wird über die in **System.Windows.Controls.ContentControl** definierte Eigenschaft **Content** der Inhalt festgelegt, z.B. bei einem **Button**-Objekt mit Beschriftung:
  ```
  <Button Content="Farbe wechseln" ... />
  ```
- **Padding**
  Mit dieser in **System.Windows.Controls.Control** definierten Eigenschaft legt man eine frei zu haltende Zone am inneren Rand des Steuerelements fest. Die Angaben sind analog zur Eigenschaft **Margin** zu machen (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**).
- **HorizontalContentAlignment**, **VerticalContentAlignment**
  Mit diesen in **System.Windows.Controls.Control** definierten Eigenschaften bestimmt man die Ausrichtung des Inhalts innerhalb der rechteckigen Steuerelementfläche. Es sind dieselben Werte möglich wie bei **HorizontalAlignment** bzw. **VerticalAlignment** (siehe Abschnitt 11.7.2.2).

Über die WPF-Eigenschaftsvererbung wurde schon in Abschnitt 11.5.1 berichtet.

11.7.2.3 Ereignisbehandlung

Zur Behandlung der von einem Steuerelement angebotenen Ereignisse kommen zwei Verfahren in Frage:

- Wie schon mehrfach in Beispielen vorgeführt, kann man zu einem Ereignis eine Behandlungsmethode mit passender Delegatensignatur erstellen und beim Ereignis registrieren.
- Wird aus einer Steuerelementklasse eine Ableitung definiert, kann man dort die ein Ereignis auslösende Methode überschreiben.

11.7.2.3.1 Behandlungsmethoden registrieren

Nach Bedarf werden in der Code-Behind - Datei zur Fensterklasse Behandlungsmethoden definiert und bei Ereignissen des Fensters oder der Steuerelemente registriert. Im Beispiel soll das **Click**-Ereignis des **Button**-Objekts behandelt werden. Dazu wählen wir bei markiertem **Button**-Objekt im Eigenschaftsfenster die Registerkarte Ereignisse und setzen einen Doppelklick auf den Eintrag **Click**. Weil das **Click**-Ereignis bei einem Befehlsschalter das Standardereignis ist, führt ein Doppelklick auf das **Button**-Objekt im WPF-Designer zum selben Ziel: Die Datei **MainWindow.xaml.cs** erscheint im Editor mit einer vorbereiteten Ereignisbehandlungsmethode,

```
private void button_Click(object sender, RoutedEventArgs e) {
}
```

die wir nur noch vervollständigen müssen, z.B.:

```
private void button_Click(object sender, RoutedEventArgs e) {
   Background = (Background == Brushes.Beige) ? Brushes.LightGray : Brushes.Beige;
}
```

Im Beispiel soll bei jedem Mausklick auf den Schalter die Hintergrundgestaltung des Fensters zwischen **Brushes.Beige** (Pinsel mit Volltonfarbe Beige) und **Brushes.LightGray** (Pinsel mit Volltonfarbe **LightGray**) wechseln.

Die Registrierung einer Ereignisbehandlungsmethode kann im XAML-Code des Fensters vorgenommen werden, was meist von der Entwicklungsumgebung erledigt wird, z.B.:

```
<Button Name="button" Content="Farbe wechseln"
        HorizontalAlignment="Center" VerticalAlignment="Center"
        Padding="10" Click="button_Click" />
```

Wie Sie aus Abschnitt 11.3.4 wissen, ist die vom XAML-Compiler erstellte Datei **MainWindow.g.i.cs** für das Registrieren der **Click**-Behandlungsmethode per C# - Ereignissyntax zuständig. Wir finden die erwartete Anweisung in der Methode **Connect**:

```
void System.Windows.Markup.IComponentConnector.Connect(int connectionId, object target) {
    switch (connectionId)
    {
    case 1:
    this.button = ((System.Windows.Controls.Button)(target));

    #line 7 "..\..\MainWindow.xaml"
    this.button.Click += new System.Windows.RoutedEventHandler(this.button_Click);

    #line default
    #line hidden
    return;
    }
    this._contentLoaded = true;
}
```

11.7.2.3.2 On-Methoden überschreiben

Zur bisher beschriebenen Technik der Ereignisbehandlung gibt es eine oft benutzte Alternative, die nun vorgestellt werden soll. Bei den für uns relevanten Ereignissen wird vom Laufzeitsystem jeweils eine Methode aufgerufen, deren Name aus dem Präfix **On** und dem Ereignisnamen besteht, z.B.

- **OnClick()**
- **OnMouseDown()**

Weil die **On**-Methoden als **virtual** definiert sind, kann man sie in abgeleiteten Klassen überschreiben, um die gewünschte Reaktion auf ein Ereignis direkt in der **On**-Methode vorzunehmen. Das Überschreiben der zugehörigen **On**-Methode ist bei den Ereignissen von Steuerelementen nicht unbedingt die bevorzugte Technik, weil dazu eine eigene Klassendefinition erforderlich ist. Bei den Fensterereignissen ist die Technik hingegen leicht nutzbar, weil wir die Klasse **Window** regelmäßig beerben.

In der Fensterklasse mit dem folgenden XAML-Code wird auf gewohnte Weise die Methode `Window_MouseDown()` beim **MouseDown**-Ereignis des Fensters registriert:

```
<Window x:Class="OnMouseDownWindow.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="OnMouseDown Window" Height="200" Width="350" MouseDown="Window_MouseDown">
    <Grid>
        <Label x:Name="label"/>
    </Grid>
</Window>
```

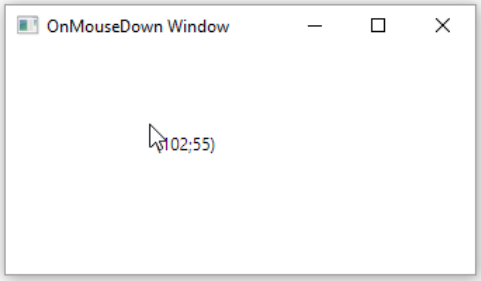
Die in der Code-Behind - Datei implementierte

```
using System;
   . . .
using System.Windows.Shapes;

namespace OnMouseDownWindow {
   public partial class MainWindow : Window {
      public MainWindow() {
         InitializeComponent();
      }

      private void Window_MouseDown(object sender, MouseButtonEventArgs e) {
         label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y,0,0);
         label.Content = "(" + e.GetPosition(this).ToString() + ")";
      }
   }
}
```

Methode `Window_MouseDown()`...

- legt die Position eines Labels über dessen **Margin**-Eigenschaft (siehe Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.**) neu auf die Klickstelle fest, die der Methode über Eigenschaften eines **MouseButtonEventArgs**-Objekts bekannt wird
- und verarbeitet die Koordinaten der Klickstelle zum neuen Wert der **Content**-Eigenschaft des Labels.

Somit kann das Programm die Koordinaten einer Klickstelle am Ort des Geschehens anzeigen, z.B.:



Als alternative Technik zur **MouseDown**-Behandlung kann in der abgeleiteten Klasse die **Window**-Methode **OnMouseDown()** überschrieben werden:

```
protected override void OnMouseDown(MouseButtonEventArgs e) {
    base.OnMouseDown(e);
    label.Margin = new Thickness(e.GetPosition(this).X, e.GetPosition(this).Y, 0, 0);
    label.Content = "(" + e.GetPosition(this).ToString() + ")";
}
```

Den Aufruf der Basisklassenmethode zu unterlassen, hat bei vielen Ereignissen zur Folge, dass registrierte Behandlungsmethoden abgehängt werden, weil das Ereignis in der On-Methode ausgelöst wird. Dies trifft z.B. für die Methode **OnClick()** der Klasse **ButtonBase** zu:[1]

```
protected virtual void OnClick() {
   RoutedEventArgs newEvent = new RoutedEventArgs(ButtonBase.ClickEvent, this);
   RaiseEvent(newEvent);
   MS.Internal.Commands.CommandHelpers.ExecuteCommandSource(this);
}
```

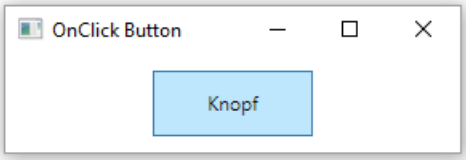
Bei einer Überschreibung ohne Basisklassenmethodenaufruf, werden beim Ereignis registrierte Methoden abgehängt, z.B.:

---

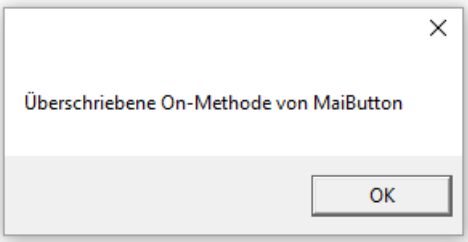
[1] Der Quellcode stammt aus der Datei **ButtonBase.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann.

```
using System;
using System.Windows;
using System.Windows.Controls;

class MaiButton : Button {
   protected override void OnClick() {
      // base.OnClick(); // Erforderlicher Aufruf der überschriebenen Methode
      MessageBox.Show("Überschriebene On-Methode von MaiButton");
   }
}

class OnClickButton : Window {
   OnClickButton() {
      Height = 100; Width = 300;
      Title = "OnClick Button";
      MaiButton k1 = new MaiButton(); k1.Content = "Knopf"; k1.Width = 100;
      AddChild(k1);
      k1.Click += new RoutedEventHandler(knopf_Click);
   }

   private void knopf_Click(object sender, RoutedEventArgs e) {
      MessageBox.Show("Click-Handler von Knopf");
   }

   [STAThread]
   static void Main() {
      new Application().Run(new OnClickButton());
   }
}
```

Aufgrund des Programmierfehlers wird bei einem Mausklick auf den Schalter



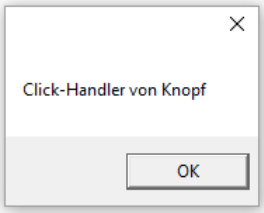
nur noch die **OnClick()** - Überschreibung ausgeführt:



Mit der korrekten Überschreibung

```
protected override void OnClick() {
   base.OnClick();
   MessageBox.Show("Überschriebene On-Methode von MaiButton");
}
```

bleiben andere Ereignisbehandlungsmethoden im Spiel, z.B.:



Bei der **Window**-Methode **OnMouseDown()** hat die Unterlassung des Basismethodenaufrufs übrigens keine erkennbaren Konsequenzen. Halten Sie sich trotzdem grundsätzlich daran, zu Beginn einer überschreibenden Methode die Basisklassenvariante aufzurufen.

11.7.3 Standardkomponenten

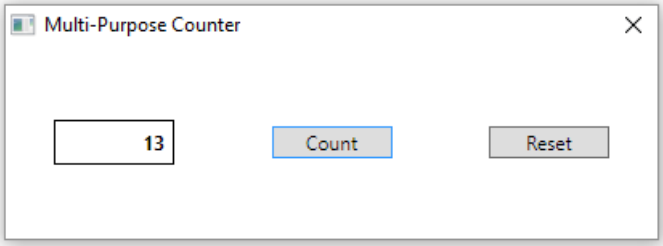
Dieser Abschnitt behandelt Steuerelemente, die sehr oft benötigt werden, und die und größtenteils auch schon in bisherigen Kursbeispielen aufgetreten sind: Befehlsschalter, Kontrollkästchen, Optionsfelder, Testeingabefelder, Listen und Kombinationsfelder.

11.7.3.1 Befehlsschalter

Befehlsschalter werden in der WPF durch die Klasse **Button** realisiert.

11.7.3.1.1 Beispiel

Im folgenden *Multi-Purpose Counter*, der z.B. zur Verkehrszählung taugt, kommen zwei **Button**-Objekte zum Einsatz:



Das GUI-Design und die Ereignisregistrierung werden durch den folgenden XAML-Code vorgenommen:

```
<Window x:Class="Button.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Multi-Purpose Counter" Height="149" Width="412" ResizeMode="NoResize">
    <Grid Button.Click="Button_Click">
        <Grid.ColumnDefinitions>
            <ColumnDefinition />
            <ColumnDefinition />
            <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <Label Name="label" Content="0" Width="75"
           HorizontalAlignment="Center" VerticalAlignment="Center" BorderThickness="1"
           BorderBrush="Black" HorizontalContentAlignment="Right" FontWeight="Bold" />
        <Button Name="count" Content="Count" Width="75" Grid.Column="1"
           HorizontalAlignment="Center" VerticalAlignment="Center" IsDefault="True" />
        <Button Name="reset" Content="Reset" Width="75" Grid.Column="2"
           HorizontalAlignment="Center" VerticalAlignment="Center" />
    </Grid>
</Window>
```

Weil eine Änderung der Fenstergröße *nicht* erwünscht ist, erhält das **XAML**-Attribut **ResizeMode** (also letztlich die gleichnamige **Window**-Eigenschaft) den Wert **NoResize**.

Die **Height**-Eigenschaften der Steuerelemente werden nicht spezifiziert, und WPF orientiert sich infolgedessen an der Schriftgröße (mit ca. 5 Pixeln Randabstand).

Um für das **Label**-Steuerelement eine fette Schrift zu wählen, erhält das **XAML**-Attribut **FontWeight** den Wert **Bold**. Außerdem erhält das Label einen Rahmen

```
BorderThickness="1" BorderBrush="Black"
HorizontalContentAlignment="Right"
```

und die (bei einer Zahlenanzeige übliche) rechtsbündige Textausrichtung:

```
HorizontalContentAlignment="Right"
```

Die für das **Click**-Ereignis *beider* **Button**-Steuerelemente zuständige Ereignisbehandlungsmethode `Button_Click()` wertet die **Source**-Eigenschaft des zweiten Parameters (vom Typ **RoutedEventArgs**) aus und orientiert ihr Verhalten an der Ereignisquelle:

```
public partial class MainWindow : Window {
  long anzahl;
   . . .
  private void Button_Click(object sender, RoutedEventArgs e) {
     if (e.Source == count)
        anzahl++;
     else
        anzahl = 0;
     label.Content = anzahl.ToString();
  }
   . . .
}
```

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **Button**-Objekten registrieren zu müssen:

```
<Grid Button.Click="Button_Click">
     . . .
</Grid>
```

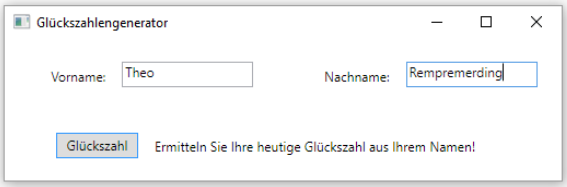
11.7.3.1.2 Standardschaltfläche, Eingabefokus und Zugriffstaste

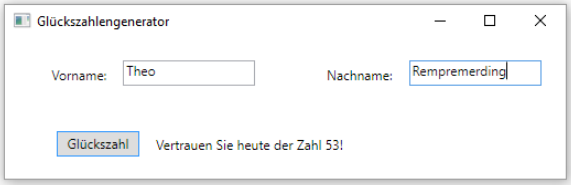
Nach dem Start sehen die beiden Schalter des Beispielprogramms verschieden aus,

und der mit einem blauen Rand dekorierte Schalter verfügt über das Privileg, auf die Enter-Taste wie auf einen Mausklick zu reagieren, solange keine andere Schaltfläche den Eingabefokus (siehe unten) besitzt. Infolgedessen kann ein Multi-Counter - Benutzer nach dem Start sofort die Enter-Taste zum Zählen verwenden, statt nach der Maus greifen zu müssen. Verantwortlich für dieses **Standardschaltflächen**-Privileg ist das XAML-Attribut (bzw. die **Button**-Eigenschaft) **IsDefault**:

```
<Button ... IsDefault="True"/>
```

Im Beispielprogramm zum **TextBox**-Steuerelement (siehe Abschnitt 11.7.3.3) erhält der Schalter zur Anforderung einer persönlichen Glückszahl die Rolle der Standardschaltfläche. Folglich kann der Schalter auch dann per **Enter**-Taste ausgelöst werden, wenn eines der Textfelder den Eingabefokus besitzt (erkennbar an der Texteinfügemarke):



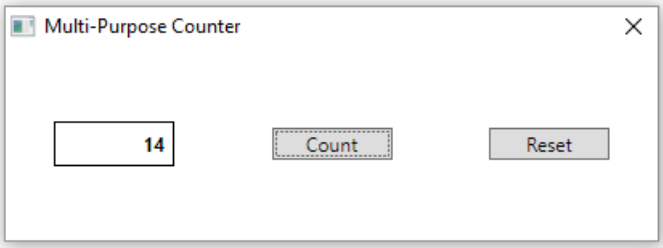


Im Glückszahlenbeispiel bietet es sich sogar an, den Schalter über den Wert **false** seiner **Focusable**-Eigenschaft aus der Liste der fokussierbaren Steuerelemente zu entfernen, damit die Tabulatortaste nur noch zwischen den beiden Textfeldern wechselt:

```
<Button Content="Glückszahl" . . . IsDefault="True" Focusable="False" />
```

Analog zur Standardschaltflächen-Ernennung über die Eigenschaft **IsDefault** kann ein **Button**-Objekt über die Eigenschaft **IsCancel** auch als **Escape-Schaltfläche** festgelegt werden. Ihr **Click**-Ereignis wird per **Esc**-Taste ausgelöst, sofern kein anderes, an der Enter-Taste interessiertes Steuerelement (z.B. ein Schalter oder ein mehrzeiliges Texteingabefeld) den Eingabefokus besitzt.

Nachdem der `count`-Schalter per Mausklick oder Tabulatortaste den **Eingabefokus** erhalten hat, kann der Schalter auch per Leertaste angesprochen werden. Leider ist dieser Status nur dann optisch erkennbar (an einem gestrichelten Innenrand), wenn der Fokus per Tabulatortaste übertragen wurde:



Es wäre nett, wenn der `count`-Schalter schon beim Programmstart den Eingabefokus hätte. Eine Lösungsmöglichkeit besteht darin, die statische Methode **Focus()** der Klasse **Keyboard** aufzurufen, die als Argument eine Referenz auf das zu privilegierende Objekt erwartet, z.B.:

```
Keyboard.Focus(count);
```

Damit diese Anweisung beim Laden des Fensters ausgeführt wird, stecken wir sie in eine Behandlungsmethode zum **Loaded**-Ereignis der Fensterklasse:

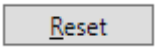
- Markieren Sie im WPF-Designer das Hauptfenster.
- Wechseln Sie im Eigenschaftenfenster zur Registerkarte **Ereignisse**.
- Setzen Sie einen Doppelklick auf die Textbox zum Ereignis **Loaded**.
- Daraufhin wird in der Quellcodedatei **MainWindow.xaml.cs** die private Instanzmethode `Window_Loaded()` zu unserer Fensterklasse `MainWindow` angelegt:
  ```
  private void Window_Loaded(object sender, RoutedEventArgs e) {
  }
  ```
- Ergänzen Sie im Rumpf dieser Methode den oben beschriebenen **Focus()** - Aufruf.

Anschließend hat der `count`-Schalter schon beim Programmstart den Eingabefokus, besitzt aber keinerlei spezielle Dekorierung mehr, weil die blaue Standardschaltflächen-Umrandung wegfällt.

Soll das **Click**-Ereignis eines Befehlsschalters über einen **Alt-Tastenbefehl** (also über eine sogenannte **Zugriffstaste**) auslösbar sein, kommt ein **AccessText**-Element zum Einsatz, z.B.:

```
<Button Name="reset" Width="75" Grid.Column="2" HorizontalAlignment="Center"
        VerticalAlignment="Center" Click="ButtonOnClick">
    <AccessText>_Reset</AccessText>
</Button>
```

Der XAML-Compiler geht von einem Inhaltselement aus, und das Attribut **Content** muss entfallen, weil sonst die Eigenschaft **Content** mehrfach definiert wäre. Man wählt die auslösende Alt-Ergänzungstaste, indem man dem zugehörigen Buchstaben im **AccessText**-Inhalt einen Unterstrich voranstellt. Spätestens nach einmaligem Drücken der Alt-Taste ist der Buchstabe im laufenden Programm durch Unterstreichung hervorgehoben, z.B.:

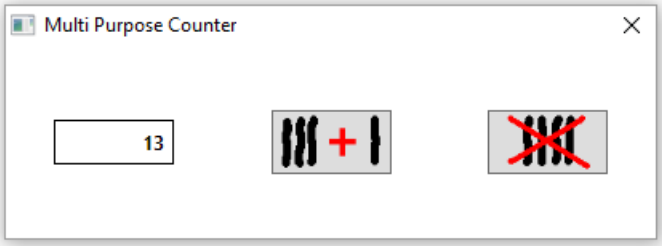


Weil bei **Button**-Objekten häufig eine Zugriffstaste benötigt wird, kann der XAML-Compiler das **AccessText**-Element automatisch einfügen, solange man eine simple Beschriftung anstelle eines zusammengesetzten **Button**-Inhalts verwendet: Setzen Sie einfach in der Beschriftung einen Unterstich vor den Buchstaben zur auslösende Alt-Ergänzungstaste, z.B.:

```
<Button Name="reset" Content="_Reset" Width="75" Grid.Column="2"
    HorizontalAlignment="Center" VerticalAlignment="Center" Click="ButtonOnClick"/>
```

11.7.3.1.3 Bitmaps auf Schaltflächen

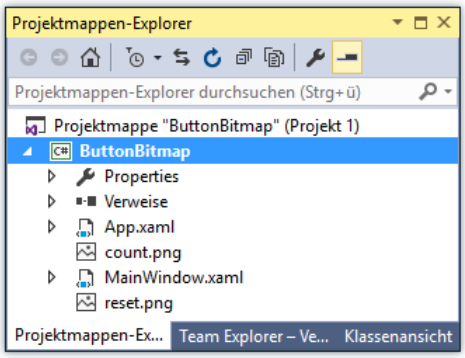
Um dem oben vorgestellten Mehrzweck-Zählprogramm ein individuelles Design zu geben, kann man die Beschriftungen der Schaltflächen durch Grafiken ersetzen (oder ergänzen), z.B.:



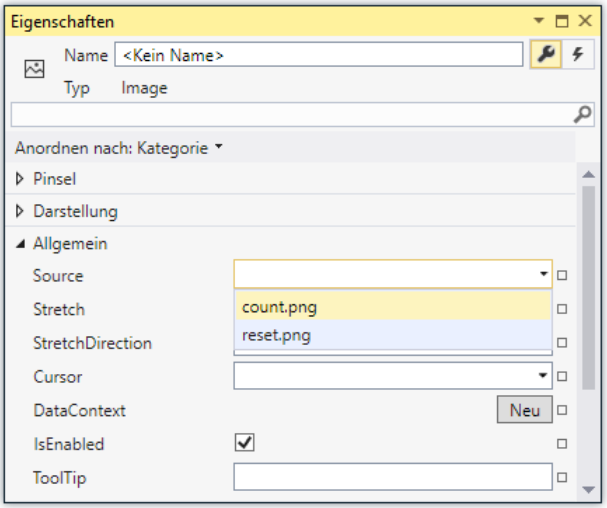
Für das Beispiel sind zunächst Bitmap-Dateien (Format **bmp** oder **png**) mit einer passenden Pixelmatrix zu erstellen (hier gewählt: 50 Zeilen und 100 Spalten), was z.B. mit der Windows-Zugabe *Paint* geschehen kann.

Gehen Sie im Visual Studio folgendermaßen vor, um die Bitmap-Dateien auf die Schaltflächen zu setzen:

- Kopieren Sie die Bitmap-Dateien in den Projektordner.
- Nehmen Sie die Bitmap-Dateien in das Projekt auf (Item **Hinzufügen > Vorhandenes Element** aus dem Kontextmenü zum Projekt). Anschließend werden die Dateien im Projektmappen-Explorer angezeigt:



- Ersetzen Sie im XAML-Code bei den **Button**-Elementen das **Content**-Attribut mit der Beschriftung durch ein Inhaltselement von Typ **Image**.
- Eine Bitmap-Datei als Quelle für die **Image**-Eigenschaft **Source** festzulegen, gelingt besonders bequem über das Eigenschaftsfenster der Entwicklungsumgebung, z.B.:

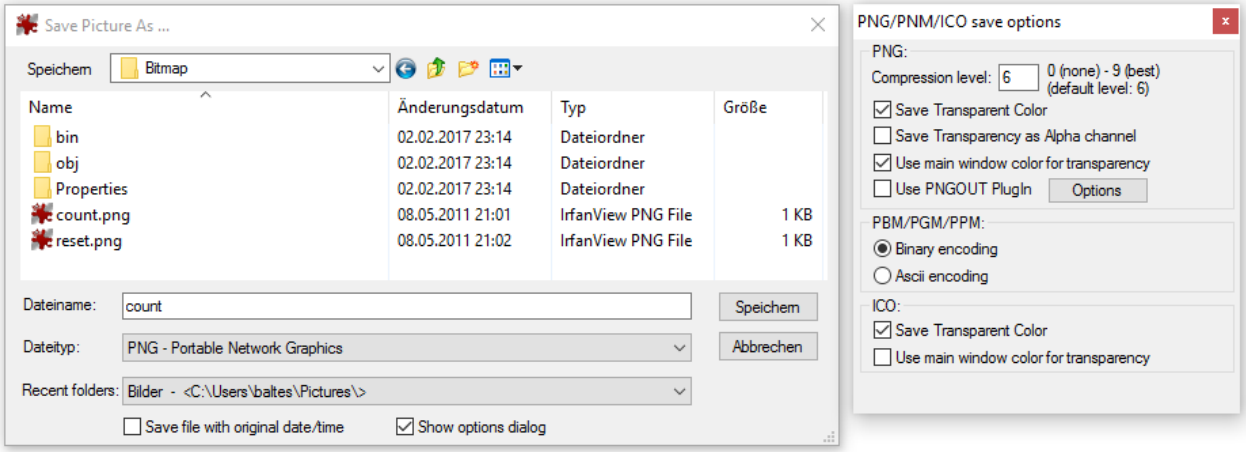


- Es resultiert der folgende XAML-Code:

```
<Button Name="count" Width="75" Height="40" Grid.Column="1" . . . IsDefault="True">
    <Image Source="count.png" />
</Button>
```

  Weil die Höhe des Schalters nicht der Bitmap-Datei überlassen werden sollte, ist ein **Height**-Attribut sinnvoll.

Macht man die Hintergrundfarbe der verwendeten Bitmaps transparent, harmonieren die bemalten Schalter unabhängig vom eingestellten Windows-Design optisch mit den restlichen Fensterbestandteilen. Dieses Ziel ist am einfachsten durch entsprechend präparierte Dateien zu realisieren. Eine PNG-Datei (*Portable Network Graphics*) mit transparenter Hintergrundfarbe lässt sich z.B. über die Freeware IrfanView[1] durch eine Option im Speichern-Dialog erstellen:



Soll ein **Button**-Steuerelement eine **Image**-Oberfläche *und* eine Zugriffstaste erhalten, setzt man unter Verwendung eines horizontal orientierten **StackPanel**-Layoutcontainers zum Bild noch ein **AccessText**-Element (siehe Abschnitt 11.7.3.1.2) auf die Schaltfläche, z.B.:

[1] Homepage: http://www.irfanview.de/

```
<Button Name="reset" Width="75" Height="40" Grid.Column="2"
        HorizontalAlignment="Center" VerticalAlignment="Center">
    <StackPanel Orientation="Horizontal">
        <Image Source="reset.png" />
        <AccessText>_Reset</AccessText>
    </StackPanel>
</Button>
```

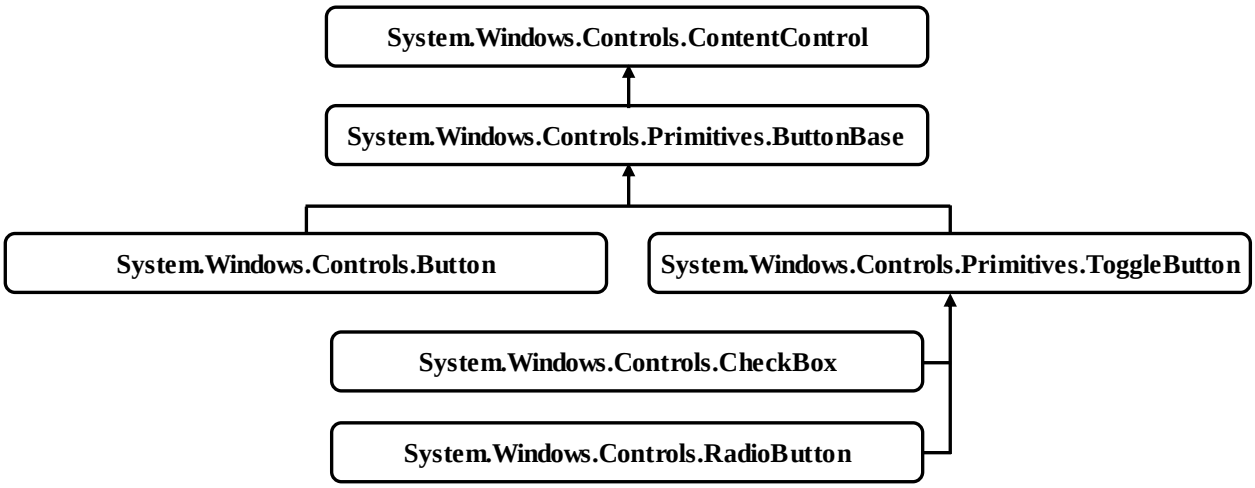
Durch die horizontalen Ausdehnungen von **Button**-Objekt und Bitmap wird ein optischer Auftritt des **AccessText**-Elements verhindert.

11.7.3.2 Kontrollkästchen und Optionsfelder

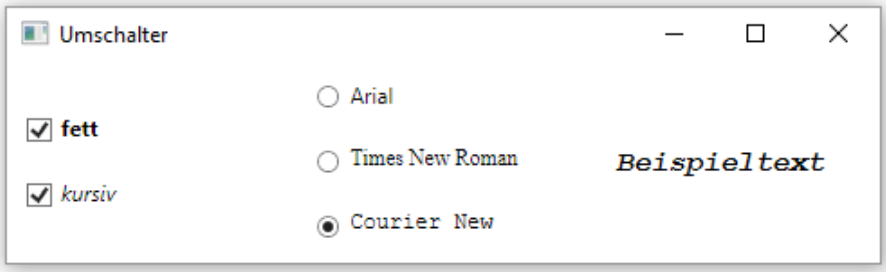
In diesem Abschnitt werden zwei Umschalter vorgestellt:

- Für **Kontrollkästchen** steht die Klasse **CheckBox** zur Verfügung.
- Für ein **Optionsfeld** verwendet man Objekte der Klasse **RadioButton**.

Beide Klassen haben **ToggleButton** als gemeinsame Basisklasse und stammen zusammen mit der schon in Abschnitt 11.7.3.1 vorgestellten Klasse **Button** von der abstrakten Basisklasse **ButtonBase** ab:



In folgendem Programm kann für den Text einer **Label**-Komponente über zwei Kontrollkästchen die Schriftauszeichnung und über ein Optionsfeld die Schriftart gewählt werden:



Beim Fensterdesign per XAML-Code kommt ein dreispaltiger **Grid**-Container zum Einsatz. In den beiden ersten Spalten arbeitet jeweils ein vertikal orientierter **StackPanel**-Container mit den **CheckBox**- bzw. **RadioButton**-Objekten als Insassen. In der dritten Spalte befindet sich das **Label**-Objekt mit dem Beispieltext:

```
<Window x:Class="Umschalter.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Umschalter" Height="150" Width="500">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition /> <ColumnDefinition /> <ColumnDefinition />
        </Grid.ColumnDefinitions>
        <StackPanel  VerticalAlignment="Center" ButtonBase.Click="CheckBox_Click">
            <CheckBox Name="chkBold" Content="fett" Height="16" Margin="10"
                      FontWeight="Bold" />
            <CheckBox Name="chkItalic" Content="kursiv" Height="16" Margin="10"
                      FontStyle="Italic"/>
        </StackPanel>
        <StackPanel  Grid.Column="1" VerticalAlignment="Center"
                     ButtonBase.Click="RadioButton_Click">
            <RadioButton Name="rbArial" Content="Arial" Height="16" Margin="10"
                         IsChecked="True" />
            <RadioButton Name="rbTimesNewRoman" Content="Times New Roman" Height="16"
                         Margin="10" FontFamily="Times New Roman"/>
            <RadioButton Name="rbCourierNew" Content="Courier New" Height="16"
                         Margin="10" FontFamily="Courier New"/>
        </StackPanel>
        <Label  Name="lblTextbeispiel" Grid.Column="2" Content="Beispieltext"
                Margin="10" VerticalAlignment="Center"
                FontFamily="Arial" FontSize="16" />
    </Grid>
</Window>
```

Damit initial die Option *Arial* markiert ist, wird beim zugehörigen **RadioButton**-Objekt die Eigenschaft **IsChecked** auf den Wert **true** gesetzt. Das **Label**-Objekt verwendet beim Start eine Schrift aus der Familie Arial ohne Auszeichnung in der Größe 16.

Alle unmittelbar zu einem Container gehörenden **RadioButton**-Objekte werden als *Gruppe* behandelt, wobei nur *ein* Mitglied eingerastet sein kann (Wert **true** bei der Eigenschaft **IsChecked**). Alternativ kann die Gruppenzugehörigkeit explizit durch das XAML-Attribut **GroupName** festgelegt werden. Alle **RadioButton**-Elemente mit demselben Wert bilden eine Gruppe, z.B.:

```
<RadioButton Name="rbOne" GroupName="g1" Content="One" ... />
```

Für die beiden Kontrollkästchen (`chkBold` und `chkItalic` genannt[1]) ist dieselbe **Click**-Behandlungsmethode `CheckBox_Click()` zuständig, die für das separate Ein- bzw. Ausschalten der Schriftattribute **fett** und *kursiv* sorgt:

```
private void CheckBox_Click(object sender, RoutedEventArgs e) {
   if (chkBold.IsChecked == true)
      lblTextbeispiel.FontWeight = FontWeights.Heavy;
   else
      lblTextbeispiel.FontWeight = FontWeights.Normal;
   if (chkItalic.IsChecked == true)
      lblTextbeispiel.FontStyle = FontStyles.Italic;
   else
      lblTextbeispiel.FontStyle = FontStyles.Normal;
}
```

Wer vermutet, die Eigenschaft **IsChecked** sei vom Typ **bool**, empfindet den logischen Ausdruck

```
chkBold.IsChecked == true
```

[1] Im Beispielprogramm wird die so genannte *Ungarische Notation* zur Bezeichnung der Instanzvariablennamen verwendet, wobei ein Präfix den Datentyp angibt (z.B. `lblTextbeispiel`). Die in früheren Zeiten der Windows-Programmierung sehr verbreitete Konvention gilt mittlerweile als obsolet und veraltet. Wir verwenden sie im aktuellen Beispiel trotzdem.

als umständlich formuliert. Tatsächlich hat **IsChecked** aber den Typ **Nullable<bool>** bzw. **bool?** (vgl. Abschnitt 7.3).

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei *beiden* **CheckBox**-Objekten registrieren zu müssen:

```
<StackPanel  VerticalAlignment="Center" ButtonBase.Click="CheckBox_Click">
    <CheckBox Content="fett" . . . /> <CheckBox Content="kursiv" . . . />
</StackPanel>
```

Im Beispielprogramm wird auch für alle **RadioButton**-Objekte eine gemeinsame **Click**-Behandlungsmethode verwendet:

```
private void RadioButton_Click(object sender, RoutedEventArgs e) {
   if (rbArial.IsChecked == true)
      lblTextbeispiel.FontFamily = ffArial;
   else
      if (rbTimesNewRoman.IsChecked == true)
         lblTextbeispiel.FontFamily = ffTNR;
      else
         lblTextbeispiel.FontFamily = ffCourierNew;
}
```

Hier werden drei Objekte vom Typ **System.Windows.Media.FontFamily** verwendet, ansprechbar über Instanzvariablen, die im Fensterklassenkonstruktor initialisiert werden:

```
public partial class MainWindow : Window {
   FontFamily ffArial = new FontFamily("Arial");
   FontFamily ffTNR = new FontFamily("Times New Roman");
   FontFamily ffCourierNew = new FontFamily("Courier New");
    . . .
}
```

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei den drei **RadioButton**-Objekten registrieren zu müssen:

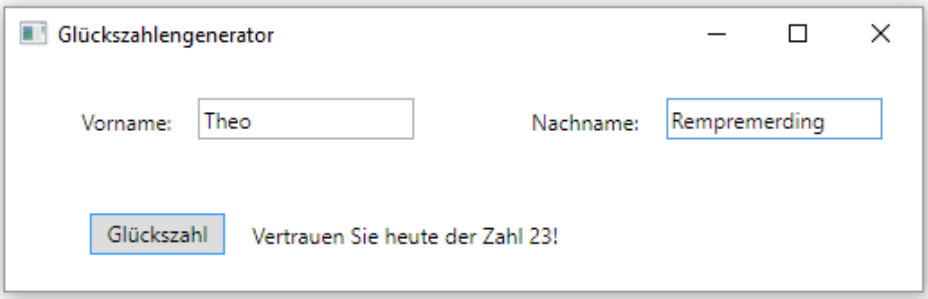
```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            ButtonBase.Click="RadioButton_Click">
    <RadioButton Name="rbArial" . . ./> <RadioButton Name="rbTimesNewRoman" . . . />
    <RadioButton Name="rbCourierNew" . . . />
</StackPanel>
```

Das komplette Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Umschalter

11.7.3.3 Texteingabefelder

Kurze Texteingaben der Benutzer erfasst man in der WPF mit Steuerelementen der Klasse **TextBox**. Auf dem bereits in Abschnitt 11.7.3.1.2 präsentierten Formular eines Programms zur Berechnung der persönlichen Glückszahl in Abhängigkeit vom Vor- und Nachnamen werden zwei **TextBox**-Objekte verwendet:



Das GUI-Design und die Ereignismethodenregistrierung werden durch den folgenden XAML-Code vorgenommen:

```
<Window x:Class="TextBox.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      Title="Glückszahlengenerator" Height="166" Width="525">
  <Grid TextBox.TextChanged="TextBox_TextChanged">
    <Grid.RowDefinitions>
        <RowDefinition /> <RowDefinition />
    </Grid.RowDefinitions>
    <Grid.ColumnDefinitions>
        <ColumnDefinition /> <ColumnDefinition />
    </Grid.ColumnDefinitions>
    <StackPanel Orientation="Horizontal"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10">
      <Label Content ="Vorname:" Height="28" Margin="10"/>
      <TextBox Name="vorname" Height="23" Width="120" VerticalContentAlignment="Center" />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Grid.Column="1"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10">
        <Label Content ="Nachname:" Height="28" Margin="10"/>
        <TextBox Name="nachname" Height="23" Width="120" VerticalContentAlignment="Center" />
    </StackPanel>
    <StackPanel Orientation="Horizontal" Grid.ColumnSpan="2" Grid.Row="1"
                HorizontalAlignment="Center" VerticalAlignment="Center" Margin="10" >
        <Button Name="button" Content="Glückszahl" Height="23" Width="75"
                Click="button_Click" IsDefault="True" Focusable="False" />
        <Label Name="info" Margin="10" Width="320" Height="28" />
    </StackPanel>
  </Grid>
</Window>
```

Über die **TextBox**-Eigenschaft **Text** kann man auf den Inhalt eines Texteingabefeldes zugreifen, z.B. in der folgenden Behandlungsmethode zum **Click**-Ereignis des Befehlsschalters:

```
private void button_Click(object sender, RoutedEventArgs e) {
   String vn = vorname.Text.ToUpper();
   String nn = nachname.Text.ToUpper();
   int seed = 0; // Startwert des Pseudozufallszahlengenerators
   if (vn.Length > 0 && nn.Length > 0) {
      foreach (char c in vn)
         seed += (int)c;
      foreach (char c in nn)
         seed += (int)c;
      Random zzg = new Random(DateTime.Today.Day + seed);
      info.Content = "Vertrauen Sie heute der Zahl " +
         (zzg.Next(100) + 1).ToString() + "!";
      valToRemove = true;
   }
}
```

Über das Ereignis **TextChanged** kann man auf jede Veränderung der **Text**-Eigenschaft eines **TextBox**-Objekts reagieren. Im Beispielprogramm wird dafür gesorgt, dass eine Glückszahlanzeige verschwindet, sobald sich einer der zugehörigen Texte ändert:

```
private void TextBox_TextChanged(object sender, TextChangedEventArgs e) {
   if (valToRemove) {
      info.Content = strInst;
      valToRemove = false;
   }
}
```

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **Grid**-Container zugeordnet werden, statt sie bei *beiden* **TextBox**-Objekten registrieren zu müssen:

```
<Grid TextBox.TextChanged="TextBox_TextChanged">
   . . .
</Grid>
```

Über die folgende Behandlungsmethode zum **Loaded**-Ereignis des Fensterobjekts erhält das Texteingabefeld zum Vornamen beim Programmstart den Eingabefokus:

```
private void Window_Loaded(object sender, RoutedEventArgs e) {
   Keyboard.Focus(vorname);
}
```

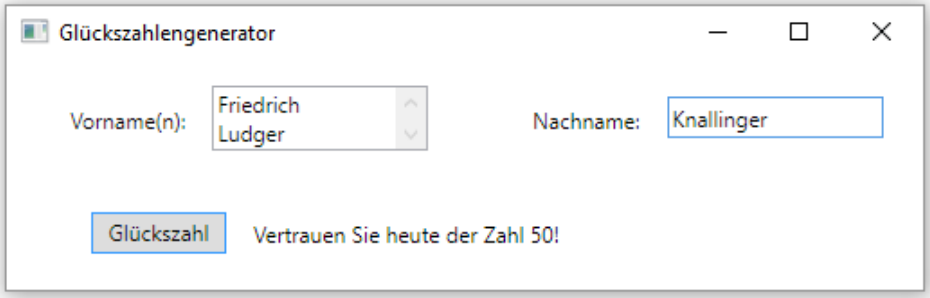
Über das Attribut **Loaded** im **Window**-Wurzelelement der XAML-Datei zur Fensterklasse wird die Methode beim Fensterobjekt registriert:

```
<Window x:Class="TextBox.MainWindow" . . . Loaded="Window_Loaded">
```

TextBox-Steuerelemente besitzen etliche Kompetenzen, die den Benutzer erfreuen und dabei den Programmierer nicht belasten, z.B.:

- Textmarkierung per Maus oder Tastatur
- Kommunikation mit der Zwischenablage über die Tastenkombinationen **Strg+C**, **Strg+X** und **Strg+V**
- Mehrstufige Rücknahme der letzten Änderungen über **Strg+Z**
- Kontextmenü mit **Bearbeiten**-Items

Um ein *mehrzeiliges* Texteingabefeld zu erhalten,



aktiviert man über den Wert **Wrap** für das **TextWrapping**-Attribut den automatischen Zeilenumbruch, ermöglicht über das Attribut **AcceptsReturn** den manuellen Zeilenumbruch per **Enter**-Taste, aktiviert über das Attribut **VerticalScrollBarVisibility** einen vertikalen Rollbalken und sorgt über das **Height**-Attribut für ausreichend Platz:

```
<TextBox Name="vorname" Height="36" Width="120" VerticalContentAlignment="Center"
  TextWrapping="Wrap" AcceptsReturn="True" VerticalScrollBarVisibility="Visible" />
```

Die Ernennung das **Button**-Objekts zur Standardschaltfläche (über den Wert **true** der Eigenschaft **IsDefault**, siehe Abschnitt 11.7.3.1.2) verliert ihre Wirkung, wenn das **Enter**-berechtige Texteingabefeld den Eingabefokus hat.

Trotz **TextWrapping** eignet sich die Klasse **TextBox** nur für kurze Texteingaben. Wir werden später mit Hilfe des leistungsfähigeren Steuerelements **RichTextBox** einen kompletten Texteditor erstellen.

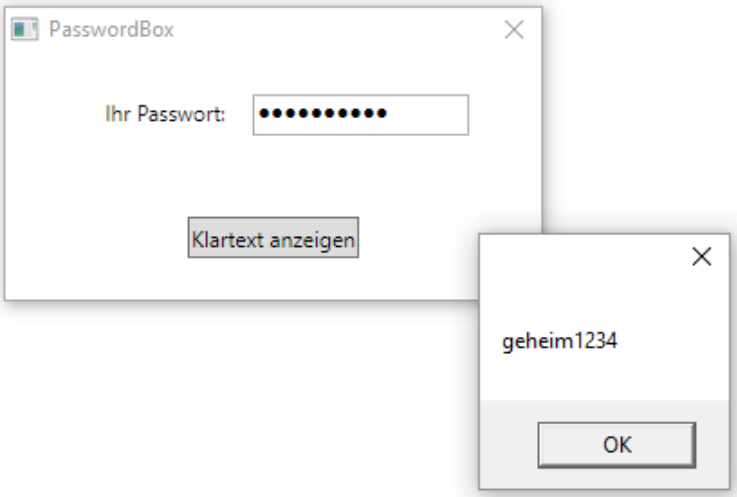
Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Texterfassung\TextBox

Zur Erfassung von **Passwörtern** bietet die WPF-Bibliothek die Klasse **PasswordBox** mit einer gegenüber der verwandten Klasse **TextBox** leicht modifizierten bzw. reduzierten Funktionalität, z.B.:

- Anzeige eines Symbols statt der Passwortzeichen
- Eigenschaft **Text** durch die Eigenschaft **Password** ersetzt
- Kein Schreiben in die Zwischenablage

Trotz des geänderten Eigenschaftsnamens ist ein erfasstes Passwort im Klartext vorhanden:

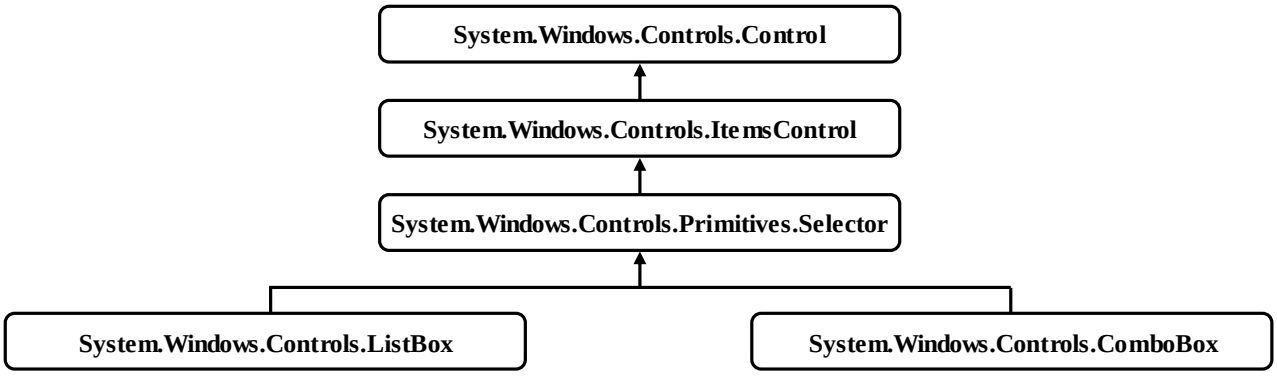


Der XAML-Code zum **PasswordBox**-Objekt:

```
<PasswordBox Name="pw" Height="23" Width="120" />
```

11.7.3.4 Listen- und Kombinationsfelder

In diesem Abschnitt werden die Steuerelementklassen **ListBox** und **ComboBox** vorgestellt, die einen gemeinsamen Stammbaum besitzen:

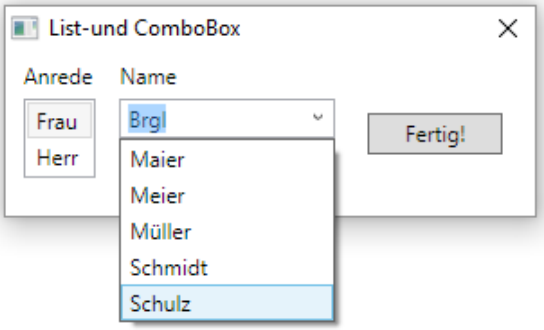


Ein **ListBox**-Steuerelement präsentiert eine Liste von Elementen, von denen der Benutzer per Maus oder Tastatur eines oder mehrere wählen kann.

Das **ComboBox**-Steuerelement bietet eine Kombination aus einem einzeiligen Texteingabefeld und einer Liste. Um seine Wahl zu treffen, hat der Benutzer *zwei* Möglichkeiten (Wert **true** für die Eigenschaft **IsEditable** vorausgesetzt, siehe unten):

- den Text der gewünschten Option eintragen
- das Drop-Down-Menü öffnen und die gewünschte Option wählen

In folgendem Programm wird per **ListBox** eine Anrede gewählt und die Angabe des Namens durch eine **ComboBox** mit einer Liste häufiger Namen erleichtert:



Ein **ListBox**-Steuerelement kann per XAML-Code über die Kollektionssyntax (siehe Abschnitt 11.3.2.3.5) mit Elementen versorgt werden

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" HorizontalAlignment="Left"
         SelectedIndex="0">
    <TextBlock>Frau</TextBlock>
    <TextBlock>Herr</TextBlock>
</ListBox>
```

Diese landen in einem Objekt der Klasse **ItemCollection**, das über die **ListBox**-Eigenschaft **Items** ansprechbar ist und Elemente vom Typ **ListBoxItem** verwaltet. Man darf aber Elemente von einem beliebigen „vorzeigbaren" Typ in eine **ListBox** einfüllen, die automatisch in **ListBoxItem**-Objekte verpackt werden. Mit expliziten **ListBoxItem**-Elementen sieht das Beispiel so aus:

```
<ListBox Name="listBox" Margin="10,0,0,0" Width="40" HorizontalAlignment="Left"
         SelectedIndex="0">
    <ListBoxItem>
        <TextBlock>Frau</TextBlock>
    </ListBoxItem>
    <ListBoxItem>
        <TextBlock>Herr</TextBlock>
    </ListBoxItem>
</ListBox>
```

Die implizite oder explizite **ListBoxItem**-Verpackung wirkt sich auf den Datentyp der Elemente aus, die z.B. von der **ItemCollection**-Methode **GetItemAt()** geliefert werden. In der zuerst vorgestellten Beispielvariante erhält man **TextBlock**-Objekte, in der zweiten hingegen **ListBoxItem**-Objekte.

Im Beispiel (mit unstrukturierten Elementen) ist die **ListBoxItem**-Verpackung überflüssig. Weil die Klasse **ListBoxItem** von **ContentControl** abstammt, bringt ihre explizite Verwendung jedoch den wesentlichen Vorteil, dass ein Item aus mehreren Elementen (z.B. aus einem **Image** und einem **TextBlock**) zusammengesetzt werden kann.

Über die Kollektionsmethoden **Add()**, **Remove()** usw. lässt sich die **ItemCollection** zur Laufzeit per Programm verändern, z.B.:

```
listBox.Items.Add(new TextBlock{Text = "Dyn"});
```

Oft befinden sich die in einem **ListBox**-Steuerelement anzuzeigenden Elemente bereits in einer Datenkollektion, und es soll eine Datenbindung (siehe Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.**) zwischen dieser Datenkollektion als Quelle und dem **ListBox**-Objekt als Ziel vorgenommen werden. Dazu wird die Datenkollektion der **ListBox**-Eigenschaft **ItemSource** zuge-

wiesen. Besonders geeignet sind Datenkollektionen aus der Klasse **ObservableCollection<T>** wegen ihrer Fähigkeit, Änderungen der Listenzusammenstellung per Ereignis an ein verbundenes **ListBox**-Steuerelement zu melden.

Sobald der **ItemsSource**-Eigenschaft eine Datenkollektion zugewiesen wurde, ist es nicht mehr möglich, die „eingebaute“, per **Items**-Eigenschaft ansprechbare Kollektion des **ListBox**-Steuerelements zu verändern. Ein Versuch führt zu einer **InvalidOperationException**.

Wir haben übrigens im `RssFeedReeder`-Projekt (vgl. vor allem Abschnitt 5.7.6) ein realistisches Beispiel für die Bevölkerung eines **ListBox**-Steuerelements durch die Elemente einer Datenkollektion kennengelernt. Meist sind die Elemente in der Datenkollektion komplex. Im `RssFeedReeder`-Projekt handelt es sich um Objekte der Klasse `RssItem`:

```
internal class RssItem {
   public string Title { get; internal set; }
   public string Description { get; internal set; }
   public string Url { get; internal set; }
}
```

Um zu einer informativen und optisch attraktiven Anzeige der **ListBox**-Items zu kommen, verwenden man ein geeignet konfiguriertes Objekt der Klasse **DataTemplate**, das der **ListBox**-Eigenschaft **ItemTemplate** zugewiesen wird. Im `RssFeedReeder`-Projekt haben wir den folgenden XAML-Code mit einem Eigenschaftselement vom Typ **DataTemplate** verwendet:

```
<ListBox x:Name="listBox" Margin="10,40.96,10,10">
  <ListBox.ItemTemplate>
      <DataTemplate>
        <StackPanel>
          <TextBlock Text="{Binding Path=Title}" Margin="1" TextAlignment="Center"
             TextWrapping="Wrap" FontSize="14" FontWeight="Bold"
             Foreground="DarkMagenta" />
          <TextBlock Text="{Binding Path=Description}" Margin="1,1,1,4"
             TextWrapping="Wrap" />
      </StackPanel>
    </DataTemplate>
  </ListBox.ItemTemplate>
</ListBox>
```

Ein **DataTemplate**-Objekt hat die (von **FrameworkTemplate**) geerbte Inhaltseigenschaft **VisualTree** (siehe Abschnitt 11.3.2.3.5 zu Inhaltseigenschaften). Zur Versorgung der Eigenschaft **VisualTree** ist ein Wurzelknoten anzugeben.[1] Im Beispiel wird ein **StackPanel**-Layoutcontainer verwendet, der zwei vertikal gestapelte **TextBlock**-Elemente enthält. Deren **Text**-Eigenschaft wird per Markup-Erweiterung (vgl. Abschnitt 11.3.2.3.6) vom Typ **Binding** mit einer Eigenschaft der Datenkollektionselemente verbunden.

Zur Erläuterung des **ComboBox**-Steuerelements kehren wir zum einfachen Beispielprogramm mit Wahlmöglichkeiten für Anrede und Namen zurück. Im XAML-Code zeigen sich kaum Unterschiede zwischen dem **ComboBox**- und dem **ListBox**-Element:

```
<ComboBox Name="comboBox" Margin="10,0,0,0" Width="120" HorizontalAlignment="Left"
          IsEditable="True">
    <TextBlock>Maier</TextBlock>
     . . .
    <TextBlock>Schulz</TextBlock>
</ComboBox>
```

Es bestehen noch mehr Gemeinsamkeiten, z.B.:

---

[1] https://msdn.microsoft.com/de-de/library/system.windows.frameworktemplate.visualtree(v=vs.110).aspx

- bei der Eigenschaft **Items**, die auch bei der Klasse **ComboBox** auf ein Objekt der Klasse **ItemCollection** zeigt
- bei der Änderung der **ItemCollection** zur Laufzeit
- bei der Datenbindung

In seiner **ItemCollection** verwaltet ein **ComboBox**-Objekt Elemente vom Typ **ComboBoxItem**, die analog zum Typ **ListBoxItem** (siehe oben) entweder implizit oder explizit kreiert werden und von **ContentControl** abstammen (MacDonald 2012, S. 187).

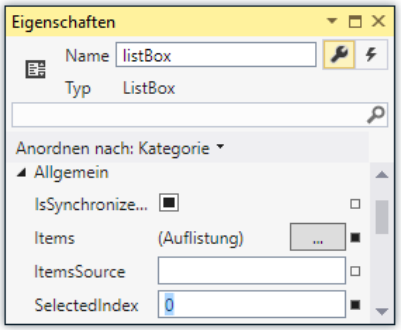
Ob ein **ComboBox**-Objekt ein Texteingabefeld anbietet, hängt von der Eigenschaft **IsEditable** ab, die im Beispiel auf den Wert **true** gesetzt wird.

Wie sich in der **Click**-Behandlungsmethode zur Schaltfläche zeigt, ist beim **ListBox**-Objekt die aktuelle Wahl über die Eigenschaft **SelectedItem** ansprechbar:

```
private void button_Click(object sender, RoutedEventArgs e) {
   MessageBox.Show("Guten Tag, " + ((TextBlock)listBox.SelectedItem).Text + " " +
      comboBox.Text);
}
```

Weil diese Eigenschaft den Datentyp **Object** besitzt, ist im Beispiel die Textextraktion erst nach einer Typumwandlung möglich. Das **ComboBox**-Objekt stellt seinen aktuell sichtbaren Inhalt über die Eigenschaft **Text** zur Verfügung.

Über die Eigenschaft **SelectedIndex** kann man dafür sorgen, dass der Anwenderbequemlichkeit halber schon beim Öffnen des Fensters ein Listenelement markiert ist, z.B.:



Über dieselbe Eigenschaft lässt sich auch der Indexwert zur aktuellen Auswahl feststellen. Bei einem **ComboBox**-Steuerelement ist allerdings die **String**-Eigenschaft **Text** weit relevanter (siehe oben).

Im obigen Beispielprogramm sind die durchaus zahlreich vorhandenen **ListBox**- bzw. **ComboBox**-Ereignisse (z.B. **SelectionChanged**) nicht von Interesse.

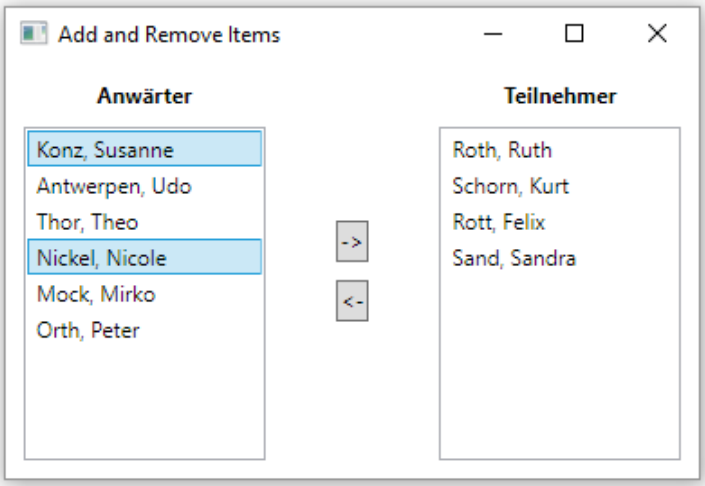
Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Listen\List- und ComboBox

In einem weiteren Beispielprogramm sollen folgende Techniken demonstriert werden:

- Mehrfachauswahl (nur bei **ListBox**-Objekten verfügbar)
- Dynamische Veränderung von Listen

Wir befüllen ein erstes **ListBox**-Objekt mit den Namen von Anwärtern und halten ein zweites **ListBox**-Objekt für die ausgewählten Teilnehmer bereit. Auf dem Anwendungsfenster stehen zwischen den beiden **ListBox**-Objekten zwei Transportschalter für das Verschieben von Namen zwischen den Listen bereit:



Im XAML-Code wird durch verschachtelte Layoutcontainer für ein sinnvolles Verhalten bei variabler Fenstergröße gesorgt:

```
<Window x:Class="ListBox.MainWindow"
        xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
        xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
        Title="Add and Remove Items" Height="270" Width="400">
    <Grid>
        <Grid.ColumnDefinitions>
            <ColumnDefinition Width="2*" />
            <ColumnDefinition Width="*" />
            <ColumnDefinition Width="2*" />
        </Grid.ColumnDefinitions>
        <DockPanel>
            <Label Content="Anwärter" DockPanel.Dock="Top" Margin="5"
                   HorizontalAlignment="Center" FontWeight="Bold" />
            <ListBox Name="listeAnwaerter" Margin="10,0,10,10" SelectionMode="Extended">
                <TextBlock>Rott, Felix</TextBlock>
                <TextBlock>Konz, Susanne</TextBlock>
                                . . .
                <TextBlock>Sand, Sandra</TextBlock>
                <TextBlock>Mock, Mirko</TextBlock>
            </ListBox>
        </DockPanel>
        <StackPanel Grid.Column="1" VerticalAlignment="Center"
                    Button.Click="Button_Click">
            <Button Name="cmdRein" Content="->" Height="23"
                    HorizontalAlignment="Center" Margin="5" />
            <Button Name="cmdRaus" Content="&lt;-" Height="23"
                    HorizontalAlignment="Center" Margin="5" />
        </StackPanel>
        <DockPanel Grid.Column="2" >
            <Label Content="Teilnehmer" DockPanel.Dock="Top" Margin="5"
                   HorizontalAlignment="Center" FontWeight="Bold" />
            <ListBox Name="listeTeilnehmer" Margin="10,0,10,10" SelectionMode="Extended">
            </ListBox>
        </DockPanel>
    </Grid>
</Window>
```

Dem dreispaltigen Top-Level - Container vom Typ **Grid** sind untergeordnet:

- Links und rechts jeweils ein **DockPanel**-Container, das ein **Label**- und ein **ListBox**-Objekt verwaltet, wobei das zuletzt eingefügte **ListBox**-Objekt den gesamten unverbrauchten Raum einnimmt.
- In der Mitte ein **StackPanel**-Container für die beiden Schaltflächen.

Im Beispielprogramm können aus jeder Liste einzelne Kandidaten flexibel ausgewählt und in die jeweils andere Liste transportiert werden. Für die Markierungsflexibilität wird über die **ListBox**-Eigenschaft **SelectionMode** mit dem Wert **Extended** gesorgt.

Beim Rücktransportschalter muss die als Beschriftung erwünschte öffnende spitze Klammer wegen der Kollision mit dem XML-Syntaxelement etwas umständlich notiert werden:

```
Content="&lt;-"
```

In der Ereignismethode zu den beiden Schaltflächen (`cmdRein`, `cmdRaus`) werden die markierten Elemente über die **ListBox**-Eigenschaft **SelectedItems** angesprochen, die auf ein Objekt einer Klasse zeigt, die das Interface **System.Collections.IEnumerable** erfüllt. Offenbar zeigt **SelectedItems** Ähnlichkeiten mit der bereits bekannten Eigenschaft **Items**, die auf eine Liste mit *allen* Elemente zeigt.

```
private void Button_Click(object sender, RoutedEventArgs e) {
   Object[] tempAnwaerter = new Object[listeAnwaerter.SelectedItems.Count];
   Object[] tempTeilnehmer = new Object[listeTeilnehmer.SelectedItems.Count];
   if (e.Source == cmdRein) {
      listeAnwaerter.SelectedItems.CopyTo(tempAnwaerter, 0);
      foreach (object anw in tempAnwaerter) {
         listeAnwaerter.Items.Remove(anw);
         listeTeilnehmer.Items.Add(anw);
      }
   } else {
      listeTeilnehmer.SelectedItems.CopyTo(tempTeilnehmer, 0);
      foreach (object anw in tempTeilnehmer) {
         listeTeilnehmer.Items.Remove(anw);
         listeAnwaerter.Items.Add(anw);
      }
   }
}
```

Weil es um ein Routingereignis geht, kann die Behandlungsmethode dem gemeinsamen **StackPanel**-Container zugeordnet werden, statt sie bei beiden **CheckBox**-Objekten registrieren zu müssen:

```
<StackPanel Grid.Column="1" VerticalAlignment="Center"
            Button.Click="Button_Click">
    <Button Content="->" . . . /> <Button Content="&lt;-" . . . />
</StackPanel>
```

In der Ereignismethode kann daher die Ereignisquelle *nicht* über den Parameter **sender** festgestellt werden, der stets auf das **StackPanel**-Objekt zeigt. Stattdessen ist die Eigenschaft **Source** des Ereignisbeschreibungsobjekts aus der Klasse **RoutedEventArgs** zu verwenden, das über den zweiten Parameter geliefert wird.

Das vollständige Projekt finden Sie im Ordner

…\BspUeb\WPF\Steuerelemente\Listen\AddRemoveItems

11.7.3.5 ToolTip

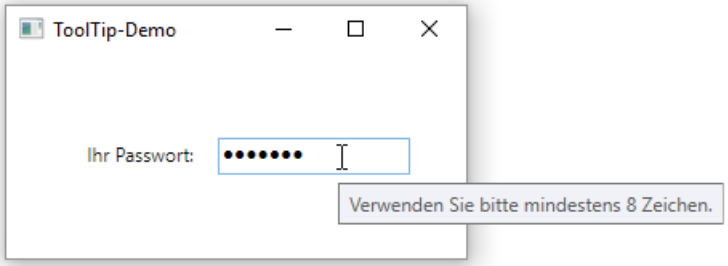
Über ein **ToolTip**-Objekt realisiert man eine in der Regel nur vorübergehend sichtbare Infoanzeige zu einem Steuerelement:

- Die Anzeige erscheint in der Nähe des zu erläuternden Steuerelements, wenn sich die Maus über diesem Element befindet.
- Das **ToolTip**-Element verschwindet, wenn der Mauszeiger den Bereich des zu erläuternden Steuerelements verlässt, oder die Anzeigedauer abgelaufen ist (Voreinstellung: 5000 Millisekunden).

Statt im XAML-Code ein Eigenschaftselement vom Typ **ToolTip** zu deklarieren, kann sich meist darauf beschränken, für das zu erläuternde Element ein **ToolTip**-Attribut zu formulieren, z.B.:

```
<PasswordBox ToolTip="Verwenden Sie bitte mindestens 8 Zeichen."
             Height="23" Name="pw" Width="120"/>
```

Hier erhalten die Benutzer einen Tipp zur Passwortlänge:



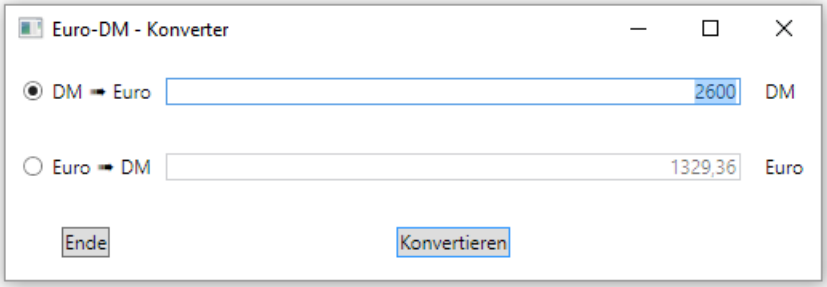
Um die voreingestellte Anzeigedauer von 5000 Millisekunden zu verändert, benutzt man die angefügte Eigenschaft (vgl. Abschnitt 11.5.2) **ShowDuration** der Klasse **ToolTipService**, z.B.:

```
<PasswordBox ToolTip="Verwenden Sie bitte mindestens 8 Zeichen."
             ToolTipService.ShowDuration="10000"
             Height="23" Name="pw" Width="120" Password="default" />
```

Über weitere angefügte Eigenschaften der Klasse **ToolTipService** lassen sich weitere **ToolTip**-Verhaltensweisen ändern (z.B. Startverzögerung, Ausstattung mit einem Schlagschatten).

11.8 Übungsaufgaben zu Kapitel 11

1) Erstellen Sie eine verbesserte Variante des Euro-DM - Konverters, den wir in Abschnitt 2.2.4 entwickelt haben. Die neue Version sollte ungefähr die folgende Bedienoberfläche besitzen;



Hinweise:

- Das untere **TextBox**-Steuerelement soll nur zur Ausgabe dienen. Daher sollten über die Eigenschaft **IsEnabled** Benutzereingaben verhindert werden.
- Bei dem horizontalen Pfeil in den Beschriftungen der Optionsschalter handelt es sich um das Unicode-Zeichen mit der Nummer 0x27A0. Über eine spezielle Escape-Sequenz lassen sich beliebige Unicode-Zeichen in die XAML-Syntax integrieren, z.B.:

```
Content="DM &#x27a0; Euro"
```

12 Ausnahmebehandlung

Durch Programmierfehler (z.B. versuchter Feldzugriff mit ungültigem Indexwert) oder durch besondere Umstände (z.B. z.B. irreguläre Eingabedaten, Speichermangel, unterbrochene Netzverbindungen) kann die reguläre Ausführung einer Methode scheitern. C# bietet ein modernes Verfahren zur Meldung und Behandlung von Problemen: An der Unfallstelle wird ein Ausnahmeobjekt aus der Klasse **Exception** (im Namensraum **System**) oder aus einer problemspezifischen Unterklasse erzeugt und der unmittelbar verantwortlichen Methode „zugeworfen". Diese wird über das Problem informiert und mit relevanten Daten für die Behandlung versorgt.

Die Initiative beim Auslösen einer Ausnahme kann ausgehen …

- von der **CLR**
  Entdeckt die CLR einen Fehler, der nicht zu schwerwiegend ist und vom Anwendungsprogramm prinzipiell behoben werden kann, dann wirft sie ein Ausnahmeobjekt, z.B. ein Objekt aus der Klasse **ArithmeticException** bei einer versuchten Ganzzahldivision durch 0.
- vom **Anwendungsprogramm**, wozu auch die verwendeten Bibliothekstypen gehören
  In jeder Methode kann mit der **throw**-Anweisung (siehe Abschnitt 12.5) eine Ausnahme erzeugt werden.

Die unmittelbar von einer Ausnahme betroffene Methode steht oft am Ende einer Sequenz verschachtelter Methodenaufrufe, und entlang der Aufrufersequenz haben die beteiligten Methoden jeweils folgende Reaktionsmöglichkeiten:

- Ausnahmeobjekt abfangen und das Problem behandeln
  Im tatsächlichen Programmablauf fliegen natürlich keine Objekte durch die Gegend, die mit irgendwelchen Gerätschaften eingefangen werden. Stattdessen überprüft die Laufzeitumgebung, ob die betroffene Methode geeigneten Code zur Behandlung des Ausnahmeobjekts (einen so genannten *Exception-Handler*) enthält. Gegebenenfalls wird dieser Exception-Handler angesprungen und erhält quasi als Aktualparameter das Ausnahmeobjekt mit Informationen über das Problem. Entscheidet sich eine Methode nach der Ausnahmebehandlung *gegen* die Fortführung des ursprünglichen Handlungsplans, dann sollte erneut ein Ausnahmeobjekt geworfen werden, entweder das ursprüngliche oder ein informativeres.
- Ausnahmeobjekt ignorieren
  In diesem Fall besitzt eine Methode keinen zum Ausnahmeobjekt passenden Exception-Handler. Die Methode wird beendet, und das Ausnahmeobjekt wird dem Vorgänger in der Aufrufersequenz überlassen.

Wir werden uns anhand verschiedener Versionen eines Beispielprogramms damit beschäftigen,

- was bei unbehandelten Ausnahmen geschieht,
- wie man Ausnahmen abfängt,
- wie man selbst Ausnahmen wirft,
- wie man eigene Ausnahmeklassen definiert.

Man kann von keinem Programm erwarten, dass es unter allen widrigen Umständen normal funktioniert. Doch müssen Schäden (z.B. Datenverluste) nach Möglichkeit verhindert werden, und der Benutzer sollte eine nützliche Information zum aufgetretenen Problem erhalten. Bei vielen Methodenaufrufen ist es realistisch und erforderlich, auf Störungen des normalen Ablaufs vorbereitet zu sein. Dies folgt schon aus **Murphy's Law** (zitiert nach Wikipedia):

> „Whatever can go wrong, will go wrong."

In C# wird allerdings keine Methode gezwungen, sich auf bestimmte (oder gar alle) zu befürchtende Ausnahmen mit einem passenden Exception-Handler vorzubereiten.

12.1 Unbehandelte Ausnahmen

Findet die CLR zu einer geworfenen Ausnahme entlang der Aufrufersequenz bis hinauf zur Methode **Main()** keine Behandlungsroutine, wird das Programm mit einer Fehlermeldung beendet. Das folgende Programm soll die Fakultät zu einer Zahl berechnen, die beim Start als Befehlszeilenargument übergeben wird. Dabei beschränkt sich die **Main()** - Methode auf die eigentliche Fakultätsberechnung und überlässt die Konvertierung und Validierung der übergebenen Zeichenfolge der Methode `Kon2Int()`. Diese wiederum stützt sich bei der Konvertierung auf die statische Methode **Parse()** der FCL-Struktur **Int32**:

```
using System;

class Fakul {
   static int Kon2Int(String instr) {
      int arg = Int32.Parse(instr);
      if (arg >= 0  &&  arg <= 170)
         return arg;
      else
         return -1;
   }

   static void Main(string[] args) {
      if (args.Length == 0) {
         Console.WriteLine("Kein Argument angegeben");
         Console.Read();
         Environment.Exit(1);
      }
      int argument = Kon2Int(args[0]);
      if (argument != -1) {
         double fakul = 1.0;
         for (int i = 1; i <= argument; i++)
               fakul = fakul * i;
         Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
      }
      else
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
      Console.Read();
   }
}
```

Ein Programm sollte sich generell bemühen, Ausnahmefehler zu vermeiden. Im Beispiel überprüft die Methode **Main()** daher, ob tatsächlich ein Kommandozeilenargument in `args[0]` vorhanden ist, bevor diese **String**-Referenz beim Aufruf der Methode `Kon2Int()` als Parameter verwendet wird. Damit wird verhindert, dass es zu einer **IndexOutOfRangeException** kommt, wenn der Benutzer das Programm *ohne* Befehlszeilenargument startet.

In diesem Fall beendet **Main()** das Programm durch einen Aufruf der Methode **Environment.Exit()**, der als Aktualparameter ein **Exitcode** übergeben wird. Dieser landet beim Betriebssystem und steht in der Umgebungsvariablen ERRORLEVEL zur Verfügung, z.B.:

```
>fakul
Kein Argument angegeben

>echo %ERRORLEVEL%
1
```

Nach einem störungsfrei verlaufenen Programmeinsatz enthält ERRORLEVEL den Exitcode 0.

Eine WPF-Anwendung sollte *nicht* durch den „aggressiven“ Methodenaufruf **Environment.Exit()** beendet werden. Um eine WPF-Anwendung per Programm zu beenden, ist die **Application**-Methode **Shutdown()** zu verwenden.

Die Reaktion des Beispielprogramms auf ein fehlendes Befehlszeilenargument kann als akzeptabel gelten:

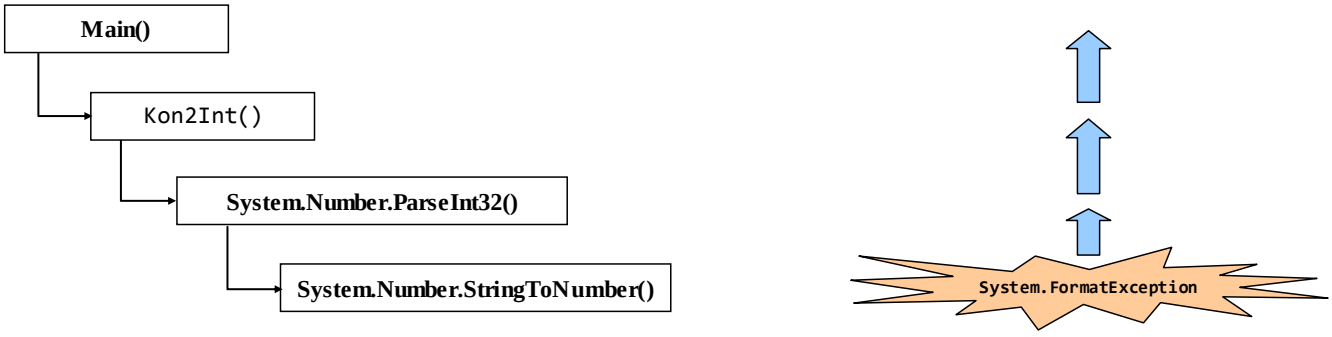
- Es erscheint eine für den Benutzer verwertbare Information an Stelle einer irritierenden und wenig hilfreichen Ausnahmemeldung durch das Laufzeitsystem.

```
Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war außerhalb
des Arraybereichs.
   bei Fakul.Main(String[] args) in Fakul.cs:Zeile 20.
```

- Falls das Programm von einem anderen Programm (z.B. einer Kommando-Prozedur) gestartet worden ist, steht dem Aufrufer ein Exitcode zur Verfügung.

Die Methode `Kon2Int()` überprüft, ob die aus dem übergebenen **String**-Parameter ermittelte **int**-Zahl außerhalb des zulässigen Wertebereichs für eine Fakultätsberechnung (mit **double**-Ergebniswert) liegt, und meldet ggf. den Wert -1 als Fehlerindikator zurück. **Main()** erkennt die spezielle Bedeutung dieses Rückgabewerts, so dass z.B. unsinnige Fakultätsberechnungen für negative Argumente vermieden werden. Diese traditionelle **Fehlerbehandlung per Rückgabewert** (engl.: *Return Code*) ist *nicht* grundsätzlich als überholt und ineffizient zu bezeichnen, aber in vielen Situationen doch der gleich vorzustellenden Kommunikation über Ausnahmeobjekte unterlegen (siehe Abschnitt 12.3 zum Vergleich von Fehlerrückmeldung und Ausnahmebehandlung).

Trotz seiner präventiven Bemühungen ist das Programm leicht aus dem Tritt zu bringen, indem man es mit einer nicht konvertierbaren Zeichenfolge füttert (z.B. „vier“). Die zunächst betroffene, FCL-intern aufgerufene, Methode **Number.StringToNumber()** wirft daraufhin eine **FormatException**. Diese wird vom Laufzeitsystem entlang der Aufrufsequenz an alle beteiligten Methoden bis hinauf zu **Main()** gemeldet:



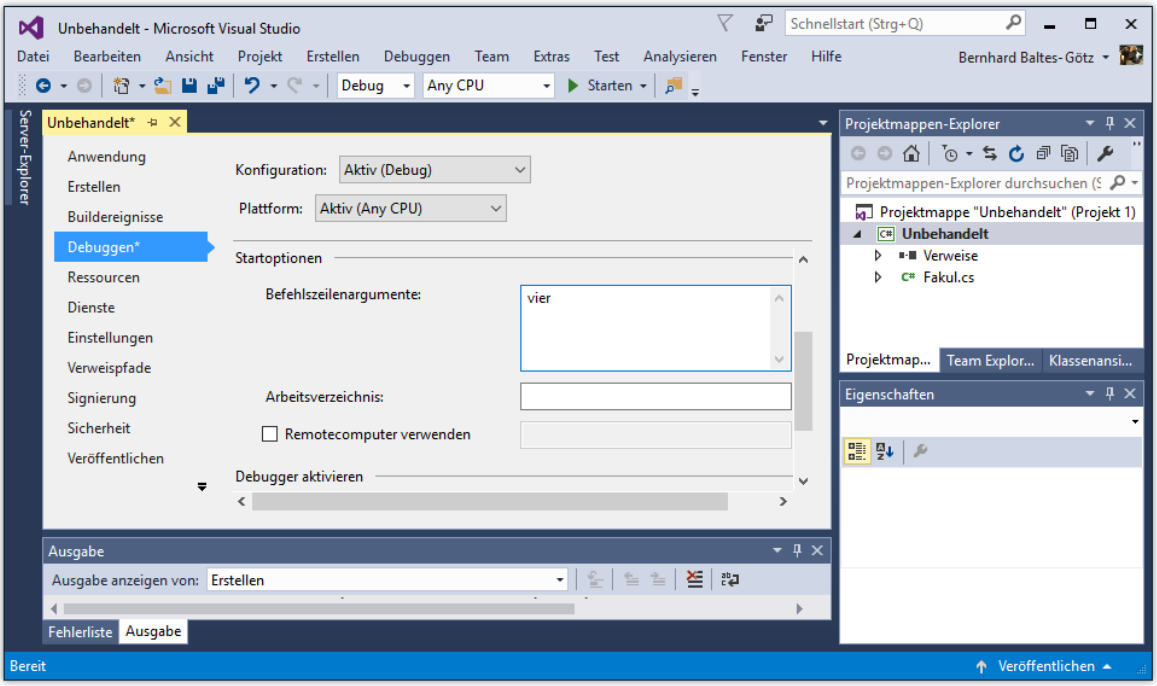
Weil kein Aufrufer eine geeignete Behandlungsroutine bereithält, endet das Programm mit einer Fehlermeldung:

```
Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das
falsche Format.
   bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
   bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
   bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
   bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 18.
```

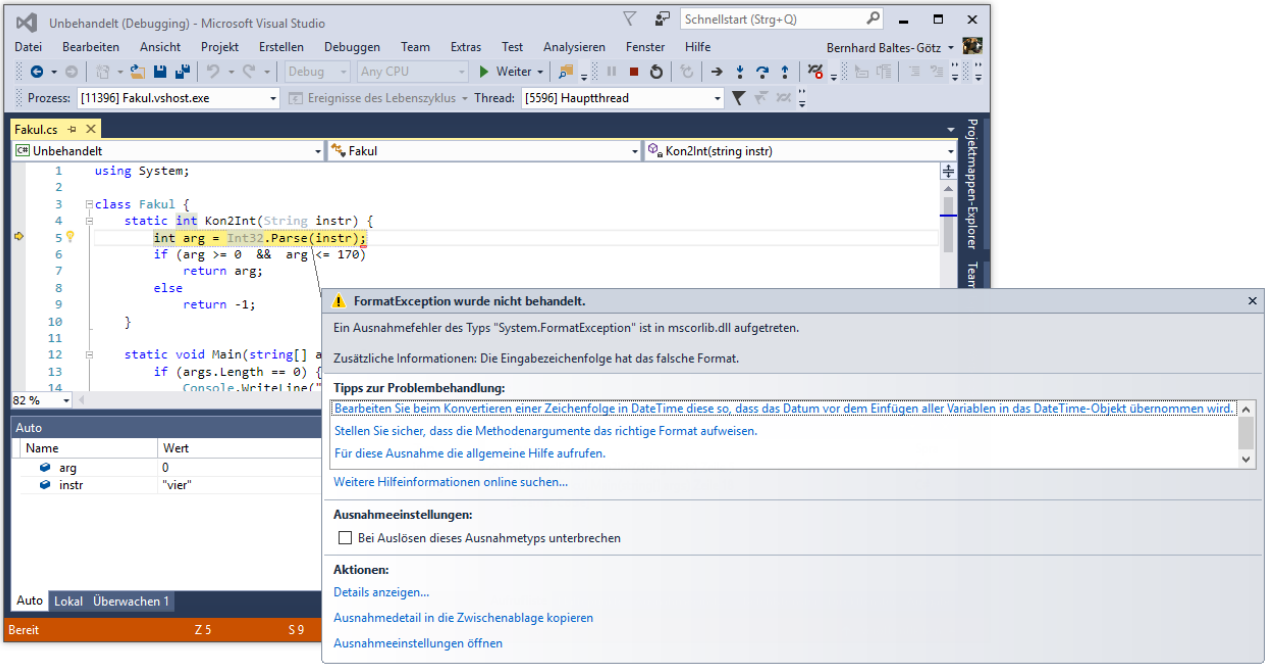
Um die Reaktion des Programms auf ein (fehlerhaftes) Befehlszeilenargument zu beobachten, muss man es übrigens nicht außerhalb der Entwicklungsumgebung in einem Konsolenfenster starten. Nach dem Menübefehl

Projekt > Eigenschaften

kann man auf der Registerkarte **Debuggen** die zu simulierenden **Befehlszeilenargumente** eintragen, z.B.:



Wurde die mit einem unbehandelten Ausnahmefehler endende Anwendung



aus dem Visual Studio (z.B. mit **F5**) im Debug-Modus gestartet, dann liefert die Entwicklungsumgebung ...

- den Typ der Ausnahme
- die Unfallstelle
- Tipps zur Lösung des Problems

Um am Projekt weiterarbeiten zu können, muss der Debug-Modus beendet werden (z.B. mit dem Symbolschalter ■ oder mit der Tastenkombination **Umschalt+F5**).

12.2 Ausnahmen abfangen

Die Startversion des Programms zur Fakultätsberechnung beherrscht weder das Behandeln noch das Werfen von Ausnahmen. Wir machen uns nun daran, diese kommunikativen Kompetenzen nachzurüsten.

12.2.1 Die try - Anweisung

In C# wird die Behandlung von Ausnahmen über die **try** - Anweisung unterstützt:

```
try {
   Überwachter Block mit Anweisungen für den regulären Ablauf
}
catch (Ausnahmeklasse1 Parametername) {
   Anweisungen für die Behandlung von Ausnahmen der ersten Ausnahmeklasse
}
// Optional können weitere Ausnahmen abgefangen werden:
catch (Ausnahmeklasse2 Parametername) {
   Anweisungen für die Behandlung von Ausnahmen der zweiten Ausnahmeklasse
}
  . . .
// Optionaler Block mit Abschluss- bzw. Bereinigungsarbeiten.
// Bei vorhandenem finally-Block, ist kein catch-Block erforderlich.
finally {
   Anweisungen, die unabhängig vom Auftreten einer Ausnahme ausgeführt werden
}
```

Die Anweisungen für den ungestörten Ablauf setzt man in den **try**-Block. Treten bei der Ausführung dieses überwachten Blocks *keine* Fehler auf, wird das Programm hinter der **try**-Anweisung fortgesetzt, wobei ggf. vorher noch der **finally**-Block ausgeführt wird.

Weil es der obigen Syntaxbeschreibung im Quellcodedesign trotz Unterstützung durch Kommentare an Präzision fehlt, sollen Sie in einer Übungsaufgabe ein Syntaxdiagramm erstellen (siehe Abschnitt 12.7).

12.2.1.1 Ausnahmebehandlung per catch-Block

Ein **catch**-Block wird auch als *Exception-Handler* bezeichnet und besitzt im Kopfbereich eine einelementige Parameterliste. Anders als bei einer Methode kann sich Parameterliste eines **catch**-Blocks jedoch auf die Typangabe beschränken oder ganz fehlen (siehe unten).

Tritt im **try**-Block eine Ausnahme auf, wird seine Ausführung abgebrochen. Anschließend sucht das Laufzeitsystem nach einem **catch**-Block, dessen Formalparameter den Typ der zu behandelnden Ausnahme oder einen Basistyp besitzt und führt dann die zugehörige Blockanweisung aus. Weil die Liste der **catch**-Blöcke von oben nach unten durchsucht wird, müssen Ausnahme*basis*klassen stets *unter* abgeleiteten Klassen stehen. Freundlicherweise stellt der Compiler die Einhaltung dieser Regel sicher. Von einer **try** - Anweisung wird maximal *ein* **catch**-Block ausgeführt. Weitere Details zum Programmablauf bei der Ausnahmebehandlung folgen in Abschnitt 12.2.2.

Nun zu den angekündigten Möglichkeiten, den Kopf eines **catch**-Blocks zu vereinfachen:

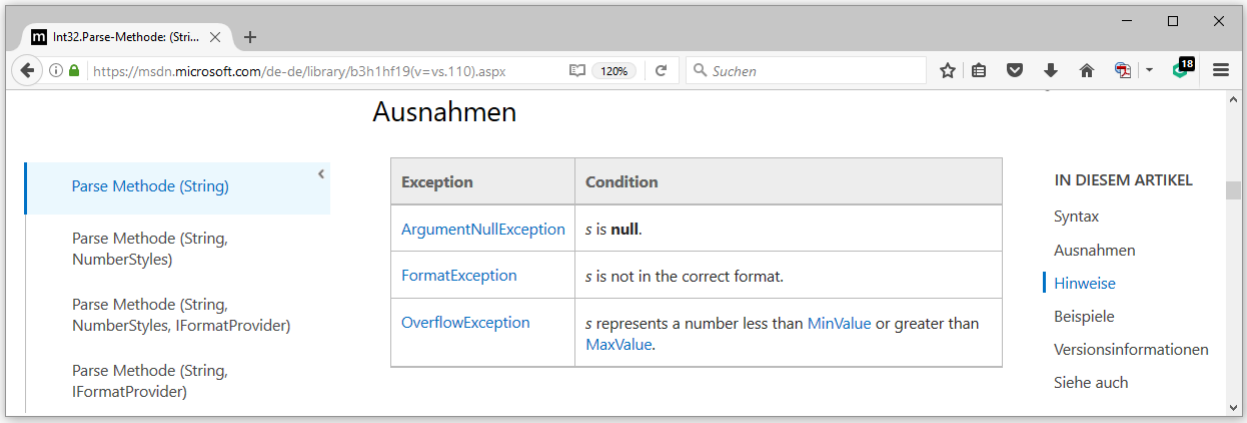
- Man kann auf die Angabe eines Formalparameter*namens* verzichten, hat dann aber im **catch**-Block kein Ausnahmeobjekt (mit Unfallbericht, siehe unten) zur Verfügung:

```
catch (Ausnahmeklasse) {
  Anweisungen für die Behandlung der Ausnahmeklasse
}
```

- Fehlt bei einem **catch**-Block die Parameterliste komplett, ist die (maximal breite) Ausnahme-Basisklasse **Exception** eingestellt, was offensichtlich nur beim *letzten* **catch**-Block sinnvoll ist:

```
catch {
  Anweisungen für die Behandlung der Ausnahmeklasse Exception
}
```

Welche Ausnahmen von den Methoden eines FCL-Typs zu befürchten sind, erfährt man in der Dokumentation, z.B. bei der Methode **Int32.Parse(String)**:



Neben der bereits besprochenen **System.FormatException** sind bei **Int32.Parse(String)** auch Ausnahmen aus den Klassen **System.ArgumentNullException** und **System.OverflowException** möglich.

In der folgenden Variante der Methode `Kon2Int()` werden die von **Int32.Parse()** zu erwartenden Ausnahmen abgefangen. Eine **ArgumentNullException** kann im Beispielprogramm ausgeschlossen werden. Nach einer **OverflowException** liefert `Kon2Int()` den Wert -1 zurück (wie bei einem **int**-Wert außerhalb von [0, 170]). Im **FormatException** - Handler wird versucht, durch sukzessives Streichen des jeweils letzten Zeichens eine numerisch interpretierbare Startzeichenfolge zu gewinnen. Bei Misserfolg landet der Wert -2 beim Aufrufer. Weil beim Reparaturversuch mit hoher Wahrscheinlichkeit mehrere Fehlversuche zu erwarten sind, ist die per Ausnahmeobjekt kommunizierende Methode **Int32.Parse()** aus Performanzgründen ungeeignet. Stattdessen wird die Methode **Int32.TryParse()** verwendet, die auf traditionelle Weise per **bool**-Rückgabewert über die Konvertierbarkeit berichtet und den resultierenden Wert per **out**-Parameter übergibt. Weil `Kon2Int()` die als Aktualparameter erhaltene Zeichenfolge nötigenfalls ändert, wird ein Referenzparameter (siehe Abschnitt 4.3.1.3.2) verwendet, damit der Aufrufer die Korrektur feststellen kann:

```
static int Kon2Int(ref String instr) {
   int arg = -1;
   try {
      arg = Int32.Parse(instr);
   } catch (FormatException) {
      String sk = instr;
      bool ok = false;
      while (sk.Length > 1 && !ok) {
         sk = sk.Substring(0, sk.Length - 1);
         ok = Int32.TryParse(sk, out arg);
      }
      if (ok)
         instr = sk;
      else
         return -2;
   } catch (OverflowException) {
      return -1;
   }
   if (arg >= 0 && arg <= 170)
      return arg;
   else
      return -1;
}
```

Die **catch**-Blöcke verwenden das von **Int32.Parse()** geworfene Ausnahmeobjekt *nicht* und verzichten daher auf einen Parameternamen.

Man kann sich fragen, ob `Kon2Int()` nicht komplett auf den potentielle Ausnahmewerfer **Int32.Parse()** und damit auch auf die Ausnahmebehandlung per **try-catch** - Anweisung verzichten und ausschließlich die mit Rückgabewert arbeitende Methode **Int32.TryParse()** verwenden sollte. Nach den Performanzüberlegungen in Abschnitt 12.3 wäre dies eine akzeptable Vorgehensweise. Trotzdem erfüllt das Beispiel in seiner jetzigen Form wohl die Aufgabe, die in vielen Situationen außerordentlich wichtige **try-catch** - Ausnahmebehandlung zu demonstrieren.

Der **catch**-Block zur **FormatException** beschränkt sich nicht auf eine pure Ausgabe zum Existenznachweis, sondern unternimmt einen ernsthaften Reparaturversuch. Je nach Algorithmus kommen als Aufgaben für einen **catch**-Block auch in Frage:

- Rückabwicklung
  Man kann versuchen, bereits realisierte und aufgrund der Ausnahme nunmehr unerwünschte Effekte des unterbrochenen **try**-Blocks wieder rückgängig zu machen.
- Informationsvermittlung
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 12.6).
- Fehlerprotokollierung
  Wenn ein gescheiterter Algorithmus abgebrochen werden muss, sollte der Benutzer informiert werden. Ein Eintrag in eine Logdatei kann dazu beitragen, die Ursache des Fehlers oder eine Umgehungsmöglichkeit zu finden (siehe Kapitel 14 zur Dateiausgabe).

Das Beispielprogramm endet aufgrund der Verbesserungen in `Kon2Int()` nun beim Auftreten einer **FormatException** (z.B. wegen des Befehlszeilenarguments „vier“) mit der Meldung:

```
Eingabefehler: vier
```

In der **Main()** - Methode des Beispielprogramms muss beim `Kon2Int()` - Aufruf die Parameterart angegeben werden. Außerdem ist die nun differenziertere Fehlerrückmeldung zu berücksichtigen:

```
static void Main(string[] args) {
   if (args.Length == 0) {
      Console.WriteLine("Kein Argument angegeben");
      Console.Read();
      Environment.Exit(1);
   }
   int argument = Kon2Int(ref args[0]);
   if (argument >= 0) {
      double fakul = 1.0;
      for (int i = 1; i <= argument; i++)
         fakul = fakul * i;
      Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
   } else
      if (argument == -1)
         Console.WriteLine("Keine ganze Zahl im Intervall [0, 170]: " + args[0]);
      else
         if (argument == -2)
            Console.WriteLine("Eingabefehler: " + args[0]);
}
```

12.2.1.2 finally

Der **finally**-Block einer **try**-Anweisung wird unter fast allen Umständen ausgeführt:

- Nach der ungestörten Ausführung des **try**-Blocks
- Nach einer Ausnahmebehandlung in einem **catch**-Block
- Nach dem Auftreten einer unbehandelten Ausnahme

Ein **finally**-Block wird auch dann ausgeführt, wenn die Methode durch eine **return**-Anweisung in einem **try**-Block oder **catch**-Block vorzeitig verlassen wird. Er wird aber *nicht* ausgeführt, wenn in einem **try**-Block oder **catch**-Block die statische Methode **Exit()** der Klasse **Environment** aufgerufen und somit der komplette Prozess terminiert wird.

Damit ist ein **finally**-Block der ideale Ort für Anweisungen, die wenn irgend möglich ausgeführt werden sollen, z.B. zur Freigabe von Ressourcen wie Datei - und Netzverbindungen. Wir verwenden (dem Kapitel 14 über Dateibearbeitung vorgreifend) zur **finally**-Demonstration eine statische Methode, die aus einer Textdatei pro Zeile eine **double**-Zahl zu lesen versucht, um den Mittelwert aus den vorhandenen Zahlen zu berechnen:

```
static void Mean(String dateiname) {
   StreamReader sr = null;
   FileStream fs = null;
   try {
      fs = new FileStream(dateiname, FileMode.Open);
   } catch {
      Console.WriteLine("Fehler beim Öffnen der Datei {0}", dateiname);
      throw;
   }
```

```
    try {
       String s;
       int n = 0;
       double summe = 0.0;
       sr = new StreamReader(fs);
       while ((s = sr.ReadLine()) != null) {
          summe += Convert.ToDouble(s);
          n++;
       }
       Console.WriteLine("Deskriptive Statistiken zur Datei {0}\n", dateiname);
       Console.WriteLine("Anzahl:\t" + n);
       Console.WriteLine("Summe:\t" + summe);
       Console.WriteLine("Mittel:\t" + summe/n);
    } catch {
       Console.WriteLine("Fehler beim Lesen der Datei {0}", dateiname);
       throw;
    } finally {
       sr.Close();
    }
}
```

Das Ergebnis eines erfolgreichen Aufrufs:

```
Deskriptive Statistiken zur Datei daten.txt

Anzahl: 18
Summe:  90
Mittel: 5
```

In der ersten **try**-Anweisung (ohne **finally**-Block) werden die vom **FileStream** - Konstruktor, der eine vorhandene Datei öffnen soll, zu erwartenden Ausnahmen behandelt:

- **System.IO.FileNotFoundException**
  Die Datei existiert nicht.
- **System.IO.IOException**
  Diese Ausnahme tritt z.B. auf, wenn ein anderer Prozess die Datei durch sein exklusives Zugriffsrecht blockiert.

Der **catch**-Block schreibt eine Fehlermeldung und wirft per **throw**-Anweisung (siehe Abschnitt 12.5) dieselbe Ausnahme erneut, so dass die Methode `Mean()` beendet und der Aufrufer über das Scheitern informiert wird.

Um den momentan interessanten Fall einer Störung *nach* dem erfolgreichen Öffnen der Datei geht es erst in der zweiten **try**-Anweisung. Eine geöffnete Datei sollte möglichst früh per **Close()** - Aufruf geschlossen werden, um andere Programme möglichst wenig zu behindern. Das muss auch für den Ausnahmefall sichergestellt werden, indem das Schließen in einem **finally**-Block stattfindet. Im Beispiel kommt es zu einer Ausnahme bei geöffneter Datei, wenn die Methode **Convert.ToDouble()** auf eine nicht konvertierbare Zeichenfolge trifft (siehe Abschnitt 12.1). Weil der **Close()** - Aufruf im **finally**-Block steht, wird er auf jeden Fall ausgeführt. Stünde er z.B. am Ende des **try**-Blocks, bliebe im eben geschilderten Ausnahmefall die Datei geöffnet bis zum Programmende.

12.2.2 Programmablauf bei der Ausnahmebehandlung

Findet das Laufzeitsystem für eine Ausnahme in der aktuellen Methode keinen zuständigen **catch**-Block, dann sucht es entlang der Aufrufersequenz weiter. Dies macht es leicht, die Behandlung einer Ausnahme der bestgerüsteten Methode zu überlassen.

12.2.2.1 Beispiel

In folgendem Beispiel dürfen Sie allerdings *keine* optimierte Einsatzplanung erwarten. Es soll demonstrieren, welche Programmabläufe sich bei Ausnahmen ergeben können, die auf verschiedenen Stufen einer Aufrufhierarchie geworfen bzw. behandelt werden. Um das Beispiel einfach zu halten, wird auf Praxisnähe verzichtet. Das Programm nimmt via Kommandozeile ein Argument entgegen, interpretiert es numerisch und ermittelt den Rest aus der Division der Zahl 10 durch das Argument:

```
using System;
class Sequenzen {
   static int Calc(String instr) {
      int erg = 0;
      try {
         Console.WriteLine("try-Block von Calc()");
         erg = Convert.ToInt32(instr);
         erg = 10 % erg;
      } catch (FormatException) {
         Console.WriteLine("FormatException-Handler in Calc()");
      } finally {
         Console.WriteLine("finally-Block von Calc()");
      }
      Console.WriteLine("Nach try-Anweisung in Calc()");
      return erg;
   }

   static void Main(string[] args) {
      try {
         Console.WriteLine("try-Block von Main()");
         Console.WriteLine("10 % " + args[0] + " = " + Calc(args[0]));
      } catch (ArithmeticException) {
         Console.WriteLine("ArithmeticException-Handler in Main()");
      } finally {
         Console.WriteLine("finally-Block von Main()");
      }
      Console.WriteLine("Nach try-Anweisung in Main()");
   }
}
```

Die Methode **Main()** lässt die eigentliche Arbeit von der Methode `Calc()` erledigen und bettet den `Calc()` -Aufruf in eine **try**-Anweisung mit **catch**-Block für die **ArithmeticException** ein, die z.B. bei einer Division durch 0 auftritt. `Calc()` benutzt die Klassenmethode **Convert.ToInt32()** sowie den Modulo-Operator in einem **try**-Block, wobei nur die (potentiell von **Convert.ToInt32()** zu erwartende) **FormatException** abgefangen wird.

Wir betrachten einige Konstellationen mit ihren Konsequenzen für den Programmablauf:

a) Normaler Ablauf
b) Exception in `Calc()`, die dort auch behandelt wird
c) Exception in `Calc()`, die in **Main()** behandelt wird
d) Exception in **Main()**, die nirgends behandelt wird

a) Normaler Ablauf

Beim Programmablauf *ohne* Ausnahmen (hier mit Befehlszeilenargument „8“) kommt es zu folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % 8 = 2
finally-Block von Main()
Nach try-Anweisung in Main()
```

b) Exception in `Calc()`, die dort auch behandelt wird

Wird beim Ausführen der Anweisung

```
erg = 10 % Convert.ToInt32(instr);
```

eine **FormatException** an `Calc()` gemeldet (z.B. wegen Befehlszeilenargument „acht“ von **Convert.ToInt32()** geworfen), dann kommt der zugehörige **catch**-Block zum Einsatz. Dann folgen:

- **finally**-Block in `Calc()`
- restliche Anweisungen in `Calc()`

Im **try**-Block von `Calc()` hinter dem Unfallort stehende Anweisungen werden *nicht* ausgeführt. So wird verhindert, dass ein Algorithmus mit fehlerhaften Zwischenergebnissen weiterläuft. Bei der traditionellen Fehlerbehandlung (siehe Abschnitt 12.3) kann das passieren und zu schwer aufklärbaren Fehlern führen.

An **Main()** wird keine Ausnahme gemeldet, also werden in dieser Methode nacheinander ausgeführt:

- **try**-Block
- **finally**-Block
- restliche Anweisungen

Insgesamt erhält man die folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
FormatException-Handler in Calc()
finally-Block von Calc()
Nach try-Anweisung in Calc()
10 % acht = 0
finally-Block von Main()
Nach try-Anweisung in Main()
```

Zu der wenig überzeugenden Ausgabe

```
10 % acht = 0
```

kommt es, weil die **FormatException** in `Calc()` nicht *sinnvoll* behandelt wird. Das aktuelle Beispiel soll ausschließlich dazu dienen, Programmabläufe bei der Ausnahmebehandlung zu demonstrieren.

c) Exception in `Calc()`, die in Main() behandelt wird

Wird eine **ArithmeticException** an `Calc()` gemeldet (z.B. wegen Befehlszeilenargument „0“), findet sich in der Methode kein passender Exception-Handler. Bevor die Methode verlassen wird, um entlang der Aufrufsequenz nach einem geeigneten Handler zu suchen, wird noch ihr **finally**-Block ausgeführt. Im Aufrufer **Main()** findet sich ein **ArithmeticException**-Handler, der nun zum Einsatz kommt. Dann geht es weiter mit dem zugehörigen **finally**-Block. Schließlich wird das Programm hinter der **try**-Anweisung der Methode **Main()** fortgesetzt. Insgesamt erhält man die folgenden Ausgaben:

```
try-Block von Main()
try-Block von Calc()
finally-Block von Calc()
ArithmeticException-Handler in Main()
finally-Block von Main()
Nach try-Anweisung in Main()
```

d) Exception in Main(), die nirgends behandelt wird

Übergibt der Benutzer gar kein Befehlszeilenargument, tritt in **Main()** beim Zugriff auf `args[0]` eine **IndexOutOfRangeException** auf (von der CLR geworfen). Weil sich kein zuständiger Handler findet, wird das Programm von der CLR beendet. Zuvor wird der **finally**-Block von **Main()**noch ausgeführt, die Anweisungen hinter der **try**-Anweisung aber nicht mehr:

```
try-Block von Main()

Unbehandelte Ausnahme: System.IndexOutOfRangeException: Der Index war außerhalb
des Arraybereichs.
   bei Sequenzen.Main(String[] args) in U:\Eigene Dateien\Sequenzen\Sequenzen.cs:Zeile 23.
finally-Block von Main()
```

12.2.2.2 Komplexe Fälle

Es ist oft erforderlich, **try**-Anweisungen zu schachteln, wobei innerhalb eines **try**-, **catch**- oder **finally**-Blocks wiederum eine komplette **try**-Anweisung stehen darf. Daraus ergeben sich weitere Ablaufvarianten für eine flexible Ausnahmebehandlung.

Wenn eine per Delegatenobjekt aufgerufene Methode wegen einer unbehandelten Ausnahme vorzeitig endet, dann werden ggf. in der Delegatenaufrufliste nachfolgende Methoden *nicht* aufgerufen.

12.2.3 Unbehandelte Ausnahmen in einer WPF-Anwendung abfangen

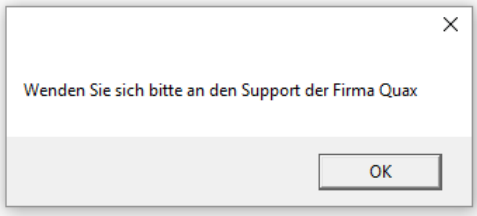
Durch eine Behandlungsmethode zum Ereignis **DispatcherUnhandledException** kann eine WPF-Anwendung auf eine ansonsten unbehandelte Ausnahme reagieren. Wird sie folgende Methode

```
private void App_DispatcherUnhandledException(object sender,
                  System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
   MessageBox.Show("Wenden Sie sich bitte an den Support der Firma Quax");
   e.Handled = true;
   Application.Current.Shutdown();
}
```

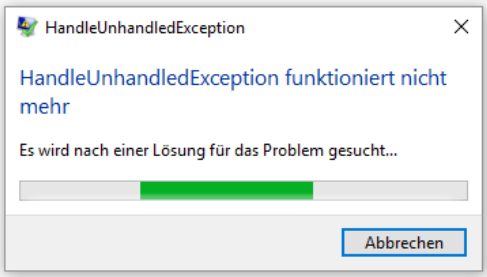
beim App-Ereignis **DispatcherUnhandledException** registriert (in der Datei **App.xaml**),

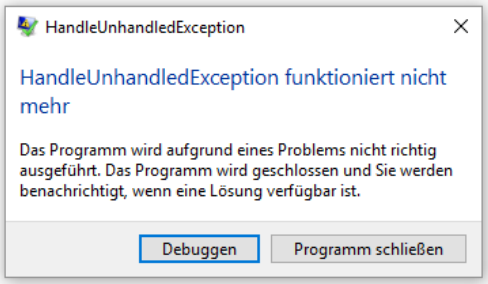
```
<Application x:Class="HandleUnhandledException.App"
             xmlns="http://schemas.microsoft.com/winfx/2006/xaml/presentation"
             xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
             xmlns:local="clr-namespace:HandleUnhandledException"
             StartupUri="MainWindow.xaml"
             DispatcherUnhandledException="App_DispatcherUnhandledException">
    <Application.Resources>
    </Application.Resources>
</Application>
```

dann erscheint nach einer unbehandelten Ausnahme ein individueller Info-Dialog,



Um die Windows-Standarddialoge für havarierte Anwendungen





zu verhindern, wird die zugrundeliegende Ausnahme als behandelt erklärt:

```
e.Handled = true;
```

Dann wird die Anwendung mit der **Application**-Methode **Shutdown()** beendet:

```
Application.Current.Shutdown();
```

Wenn es zu verantworten ist und/oder vom Anwender auf gewünscht wird, kann die Ausnahme als erledigt erklärt und damit ignoriert werden:

```
private void App_DispatcherUnhandledException(object sender,
                 System.Windows.Threading.DispatcherUnhandledExceptionEventArgs e) {
   if (MessageBox.Show("Autsch! Es ist ein unerwartetes Problem aufgetreten." +
      "Soll die Anwendung trotzdem fortgesetzt werden?", "HandleUnhandledException",
       MessageBoxButton.YesNo) == MessageBoxResult.Yes)
      e.Handled = true;
   else {
      e.Handled = true;
      Application.Current.Shutdown();
   }
}
```

12.3 Ausnahmeobjekte im Vergleich zur traditionellen Fehlerbehandlung

Die konventionelle Fehlerbehandlung verwendet meist die **Rückgabewerte** von Methoden zur Berichterstattung über Probleme bei der Ausführung von Aufträgen. Ein Rückgabewert kann …

- ausschließlich zur Fehlermeldung dienen
  Meist wird dann ein ganzzahliger **Returncode** mit dem Datentyp **int** verwendet, wobei die 0 einen erfolgreichen Ablauf meldet, während andere Zahlen für einen bestimmten Fehlertyp stehen. Soll nur zwischen Erfolg und Misserfolg unterschieden werden, bietet sich der Rückgabewert **bool** an.
- neben den Ergebnissen einer ungestörten Ausführung über spezielle Werte auch Problemfälle signalisieren (siehe Methode `Kon2Int()` im Beispielprogramm)

Wenn der Rückgabewert mit der Fehlersignalisierung komplett ausgelastet ist, kann zum Ergebnistransport ein **out**- oder **ref**-Parameter verwendet werden (siehe Abschnitt 4.3.1.3 zu den Parameterarten).

Sollen z.B. drei Methoden, deren Rückgabewerte ausschließlich zur Fehlermeldung dienen, nacheinander aufgerufen werden, dann wird die vom Algorithmus diktierte simple Sequenz:

```
static void Main() {
   M1();
   M2();
   M3();
}
```

nach der Ergänzung der Fehlerbehandlungen zu einer länglichen und recht unübersichtlichen Konstruktion (nach Mössenböck 2005, S. 254):

```
static void Main() {
   int returncode;
   returncode = M1();
   if (returncode == 0) {
      returncode = M2();
      if (returncode == 0) {
         returncode = M3();
         // Behandlung für diverse M3() - Fehler
         if (returncode == 1) {
            // . . .
         }
         // . . .
      } else {
         // Behandlung für diverse M2() - Fehler
      }
   } else {
      // Behandlung für diverse M1() - Fehler
   }
}
```

Mit Hilfe der Ausnahmetechnik bleibt im Kernalgorithmus die Übersichtlichkeit erhalten. Wir nehmen nun an, dass die drei Methoden `M1()`, `M2()` und `M3()` durch Ausnahmeobjekte über Fehler informieren:

```
static void Main() {
   try {
      M1();
      M2();
      M3();
   } catch (ExA a) {
      // Behandlung von Ausnahmen aus der Klasse ExA
   } catch (ExB b) {
      // Behandlung von Ausnahmen aus der Klasse ExB
   } catch (ExC c) {
      // Behandlung von Ausnahmen aus der Klasse ExC
   }
}
```

Es ist zu beachten, dass z.B. nach der Behandlung einer durch die Methode `M1()` verursachten Ausnahme die weiteren Anweisungen des überwachten **try**-Blocks nicht mehr ausgeführt werden. In einem **try**-Block sollten Anweisungen stehen, die allesamt erfolgreich ausgeführt werden müssen, damit ein sinnvolles Ergebnis entsteht. Die bei beliebigen Teilschritten aufgetretenen Ausnahmen sollten an einer Stelle zusammengeführt werden, wo über das weitere Vorgehen nach einem gescheiterten Algorithmus entschieden werden kann.

Ein gut gesetzter Rückgabewert nutzt natürlich nichts, wenn sich der Aufrufer nicht darum kümmert.

Neben dem unübersichtlichen Quellcode und der ungesicherten Beachtung eines Rückgabewerts ist am klassischen Verfahren zu bemängeln, dass eine Fehlerinformation aufwändig entlang der Aufrufersequenz nach oben gemeldet werden muss, wenn sie nicht an Ort und Stelle behandelt werden soll.

Gegenüber der konventionellen Fehlerbehandlung hat die Kommunikation über Ausnahmeobjekte u.a. folgende Vorteile:

- Bessere Lesbarkeit des Quellcodes
  Mit Hilfe einer **try-catch-finally** - Konstruktion erreicht man eine bessere Trennung zwischen den Anweisungen für den normalen Programmablauf und den diversen Ausnahmebehandlungen, so dass der Quellcode übersichtlich bleibt.

- Garantierte Beachtung von Ausnahmen
  Im Unterschied zu Return Codes können Ausnahmen nicht ignoriert werden. Reagiert ein Programm nicht darauf, wird es vom Laufzeitsystem beendet.
- Automatische Weitermeldung bis zur bestgerüsteten Methode
  Oft ist der unmittelbare Aufrufer nicht gut gerüstet zur Behandlung einer Ausnahme, z.B. nach dem vergeblichen Öffnen einer Datei. Dann soll eine „höhere“ Methode über das weitere Vorgehen entscheiden.
- Bessere Fehlerinformationen für den Aufrufer
  Über ein **Exception**-Objekt kann der Aufrufer beliebig genau über einen aufgetretenen Fehler informiert werden, was bei einem klassischen Return Code nicht der Fall ist.

Wie die Realisation des **catch**-Blocks zur **FormatException** in Abschnitt 12.2.1 gezeigt hat, ist die Fehlermeldung per Ausnahmeobjekt dem klassischen Rückgabewert nicht grundsätzlich überlegen. Bei der Entscheidung ist u.a. die Wahrscheinlichkeit für das Auftreten eines Problems relevant:

- Wenn ein **Problem mit erheblicher Wahrscheinlichkeit** auftritt, sollte eine routinemäßige, aktive Kontrolle stattfinden. Daher sollte eine Methode, die ein solches Problem zu melden hat, davon ausgehen, dass der Aufrufer mit dem Problem rechnet und per Rückgabewert kommunizieren. Über ein mit erheblicher Wahrscheinlichkeit auftretendes Problem per Ausnahmeobjekt zu informieren, wäre eine unangemessen aufwendige Kommunikationstechnik.
- Bei **Fehlern mit geringer Wahrscheinlichkeit** haben jedoch häufige, meist überflüssige Kontrollen eine Leistungseinbuße zur Folge. Hier sollte man es besser auf eine Ausnahme ankommen lassen. Eine Überwachung über die Ausnahmetechnik verursacht praktisch nur dann Kosten, wenn tatsächlich eine Ausnahme geworfen wird. Diese Kosten sind allerdings deutlich größer als bei einer Fehleridentifikation auf traditionelle Art.

Eine gute Wahl des Verfahrens zur Fehlerkommunikation ist besonders dann relevant, wenn eine Bibliotheksmethode für einen größeren Nutzerkreis entstehen soll. Microsoft spricht in seinen *Framework Design Guidelines* für diesen Fall eine unmissverständliche Empfehlung aus:[1]

> **X DO NOT** return error codes.
> Exceptions are the primary means of reporting errors in frameworks.

Wer eine Methode (z.B. aus einer FCL-Klasse) *nutzt*, muss das dort realisierte Verfahren zur Fehlerkommunikation kennen und sich darauf einstellen. In seltenen Fällen besteht die Wahl zwischen zwei Methoden, die sich nur beim Verfahren zur Fehlerkommunikation unterscheiden. So bietet die Struktur **Int32** *zwei* statische Methoden zur Wandlung einer Zeichenfolge in einen **int**-Wert an:

- **public static int Parse(String** *s***)**
  Diese Methode transportiert das Ergebnis einer erfolgreichen Wandlung per Rückgabewert und reagiert auf ein ungeeignetes Argument mit einem Ausnahmeobjekt:

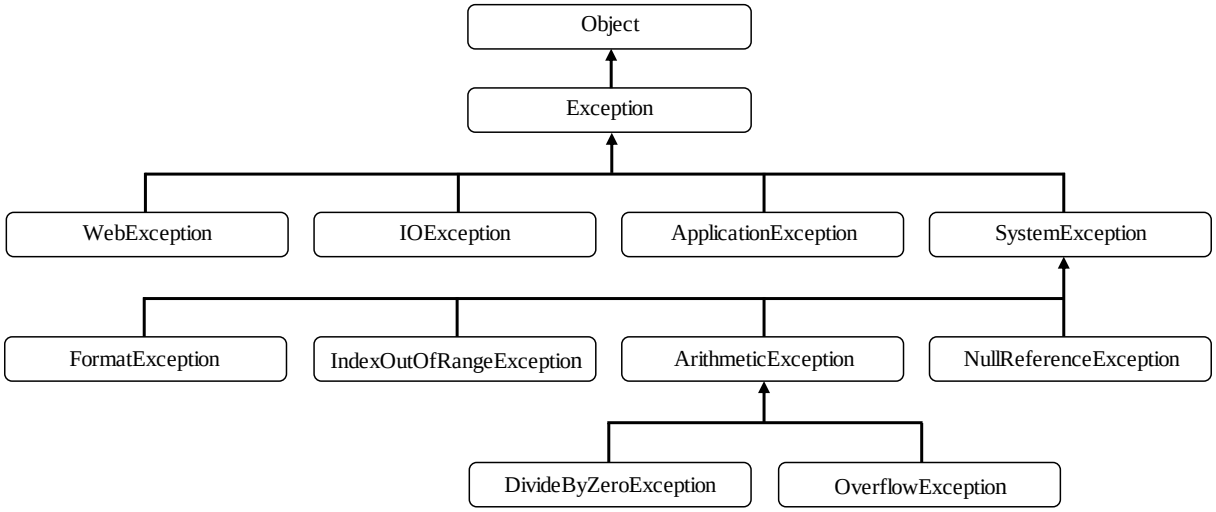
Bedingung	**Ausnahmeklasse**
s ist gleich **null**	**ArgumentNullException**
s ist nicht konvertierbar	**FormatException**
Das Konvertierungsergebnis liegt nicht im **int**-Wertebereich	**OverflowException**

[1] https://msdn.microsoft.com/en-us/library/ms229030(v=vs.110).aspx

- **public static bool TryParse(String** *s***, out int** *result***)**
  Diese Methode benutzt den Rückgabewert als Fehlerindikator und transportiert das Ergebnis mit einem **out**-Parameter.

12.4 Ausnahme-Klassen im .NET - Framework

Das .NET - Framework kennt zahlreiche vordefinierte Ausnahmeklassen, die mit ihren Vererbungsbeziehungen eine Klassenhierarchie bilden, aus der die folgende Abbildung einen kleinen Ausschnitt zeigt:



In einem **catch**-Block können auch *mehrere* Ausnahmen durch Wahl einer entsprechend breiten Basisklasse abgefangen werden.

Schon in der Klasse **Exception** sind u.a. die folgenden Eigenschaften mit Detailinformationen zu einer Ausnahme definiert:

- **Message**
  Diese **String**-Eigenschaft enthält eine Fehlermeldung mit Angaben zur Ursache der Ausnahme. Der Methodenaufruf

  ```
  Convert.ToInt32("Drei")
  ```

  sorgt z.B. für eine **FormatException** mit der **Message**-Eigenschaft:

  ```
  Die Eingabezeichenfolge hat das falsche Format.
  ```

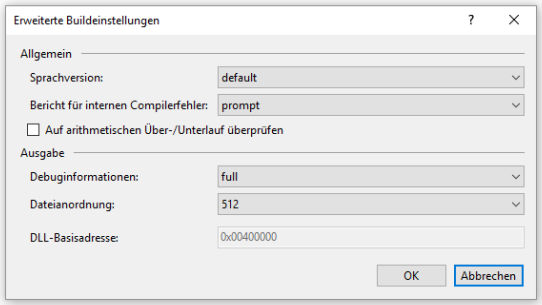
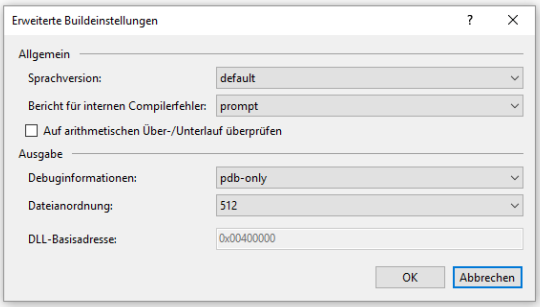
- **StackTrace**
  Diese **String**-Eigenschaft beschreibt den Aufrufstapel mit den beim Auftreten der Ausnahme aktiven Methoden. Am Ende von Abschnitt 12.1 war schon die **StackTrace**-Eigenschaft der **FormatException** zu sehen, die in der ersten Variante des Fakultätsprogramms bei einer irregulären Zeichenfolge von der Methode **Number.StringToNumber()** geworfen wird:

  ```
  Unbehandelte Ausnahme: System.FormatException: Die Eingabezeichenfolge hat das
  falsche Format.
     bei System.Number.StringToNumber(String str, ..., Boolean parseDecimal)
     bei System.Number.ParseInt32(String s, NumberStyles style, NumberFormatInfo info)
     bei Fakul.Kon2Int(String s) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 5.
     bei Fakul.Main(String[] args) in U:\Eigene Dateien\ ... \Fakul.cs:Zeile 18.
  ```

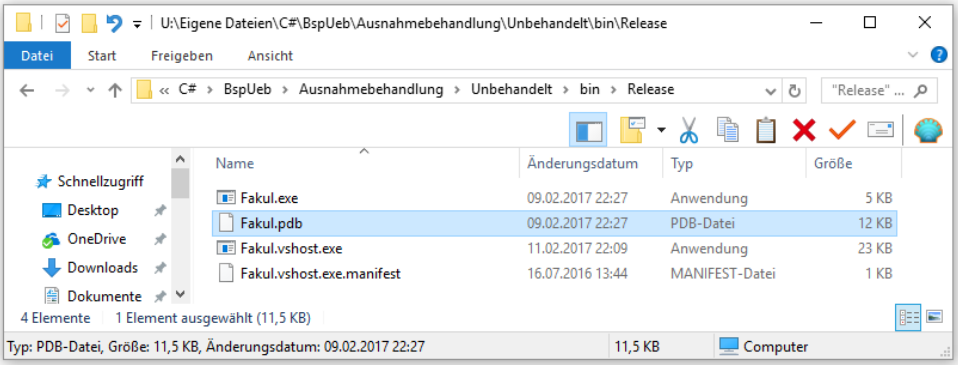
  Dateinamen und Zeilennummern enthält die Aufrufreihenfolge übrigens nur dann, wenn für die betroffene Erstellungskonfiguration (**Debug** oder **Release**) ein ausreichender Umfang an **Debuginformationen** eingestellt ist (mindestens **pdb-only**). Im Visual Studio 2015 nimmt man die Einstellung nach

  Projekt > Einstellungen > Erstellen > Erweitert

  vor, wobei die folgenden Voreinstellungen gelten:

Debug-Konfiguration	**Release**-Konfiguration
	

Somit erscheinen die Fehlerlokalisierungsinformationen per Voreinstellung in beiden Konfigurationen. Die Vorbereitung für die Ausgabe einer genauen Fehlerlokalisation ist an der Anwesenheit einer Datei mit der Namenserweiterung **.pdb** im Projektordner mit dem Assembly zu erkennen, z.B.:



Durch Optimierungsmaßnahmen des Compilers (z.B. Inlining) kann die Aufrufersequenz kürzer als erwartet ausfallen.

- **InnerException**
  Viele **catch**-Blöcke betätigen sich als Informationsvermittler und werfen selbst eine Ausnahme, um dem Aufrufer einen leichter verständlichen Unfallbericht zu liefern (siehe Abschnitt 12.6). Um dem Aufrufer auch die ursprüngliche Ausnahme zur Verfügung zu stellen, kann man ihre Adresse in der Eigenschaft **InnerException** mitliefern.

- **Data**
  Über die **Data**-Eigenschaft mit dem Interface-Typ **IDictionary** kann man Zusatzinformationen zur Ausnahme in einer beliebig langen Schlüssel-Wert - Liste (mit Elementen vom Typ **DictionaryEntry**) unterbringen. Für die Schlüssel wählt man in der Regel den Datentyp **String**, für die Werte einen geeigneten Typ. Im folgenden Beispiel werden drei Einträge in die **Data**-Liste eines neuen Ausnahmeobjekts aufgenommen:

```
if (arg < 0 || arg > 170) {
   BadFaculArgException bfa =
      new BadFaculArgException("Wert ausselhalb [0, 170]");
   bfa.Data.Add("Input", instr);
   bfa.Data.Add("Type", 4);
   bfa.Data.Add("Value", arg);
   throw bfa;
}
```

Die **ToString()** - Methode eines **Exception**-Objekts liefert:

- den Namen der Ausnahmeklasse
- **Message**-Zeichenfolge (die beim Erzeugen der Ausnahme formulierte Fehlermeldung)
- **StackTrace**-Zeichenfolge (die Aufrufreihenfolge)

Beispiel:

```
System.FormatException: Die Eingabezeichenfolge hat das falsche Format.
   bei System.Number.StringToNumber(String str, . . .)
   bei System.Number.ParseInt32(String s, . . .)
   bei System.Convert.ToInt32(String value)
   bei Sequenzen.Calc(String instr) in U:\Eigene Dateien\ ... \Sequenzen.cs:Zeile 8.
```

12.5 Ausnahmen werfen (throw)

Unsere eigenen Methoden und Konstruktoren müssen sich nicht auf das *Abfangen* von Ausnahmen beschränken, die vom Laufzeitsystem oder von Bibliotheksmethoden stammen, sondern können sich auch als Werfer betätigen, um bei misslungenen Aufrufen den Absender mit Hilfe der flexiblen Exception-Technologie zu informieren.

Insbesondere sollten Methoden und Konstruktoren übergebene Parameterwerte routinemäßig prüfen und ggf. die Ausführung durch das Werfen einer Ausnahme abbrechen. In folgender Variante der Methode `Kon2Int()` aus dem Standardbeispiel von Kapitel 12 wird ein Ausnahmeobjekt aus der FCL-Klasse **ArgumentOutOfRangeException** geworfen, wenn die erfolgreiche Interpretation des Parameters `instr` ein unzulässiges Fakultätsargument ergibt:

```
static int Kon2Int(String instr) {
   int arg;
   arg = Int32.Parse(instr);
   if (arg < 0  ||  arg > 170)
      throw new ArgumentOutOfRangeException("instr", arg,
          "Argument ausserhalb [0, 170]");
   else
      return arg;
}
```

Zum Auslösen einer Ausnahme dient die **throw**-Anweisung. Sie enthält nach dem Schlüsselwort **throw** eine Referenz auf ein Ausnahmeobjekt. Wie im Beispiel benutzt man oft den **new**-Operator mit nachfolgendem Konstruktor, um vor Ort das Ausnahmeobjekt zu erzeugen.

Im Beispiel wird ein **ArgumentOutOfRangeException**-Konstruktor mit *drei* Parametern verwendet, wobei Name und Wert des irregulären Arguments sowie eine Fehlermeldung anzugeben sind. Aus dem Aufruf

```
try {
   argument = Kon2Int(args[0]);
} catch (Exception e) {
   Console.WriteLine(e.Message);
}
```

mit dem Argument 188 resultiert die Meldung:

```
Argument ausserhalb [0, 170]
Parametername: instr
Der tatsächliche Wert war 188.
```

In einem **catch**-Block darf das Schlüsselwort **throw** auch *ohne* Ausnahmeobjekt-Referenz stehen. In diesem Fall wird die gerade behandelte Ausnahme erneut geworfen. Dieses Verhalten kommt z.B. in Frage, wenn eine Methode auf eine Ausnahme reagieren und einen Lösungsversuch unternehmen möchte, aber das Problem nicht aus Welt schaffen kann und daher ihren Aufrufer informieren muss.

Statt die ursprüngliche Ausnahme in einem **catch**-Block erneut zu werfen, kommt auch die Verwendung einer anderen Ausnahmeklasse in Frage, die aufgrund der bisherigen Analyse besser geeignet erscheint. Damit die ursprüngliche Ausnahme als Anlage beigefügt werden kann, bieten die FCL-Ausnahmeklassen einen Konstruktor mit einem Parameter vom Typ **Exception**. Ein Handler kann ggf. über die Eigenschaft **InnerException** auf die Anlage zugreifen.

In der obigen `Kon2Int()` - Variante wird auf die Behandlung der von **Int32.Parse()** potentiell zu erwartenden Ausnahmen verzichtet. Folglich hat ein `Kon2Int()` - Nutzer mit folgenden Ausnahmeklassen zu rechnen:

- **FormatException**
- **ArgumentNullException**
- **OverflowException**
- **ArgumentOutOfRangeException**

In der `Kon2Int()` - Dokumentation muss darüber informiert werden.

12.6 Ausnahmen definieren

Mit Hilfe von Ausnahmeobjekten kann eine Methode beim Auftreten von Fehlern den Aufrufer ausführlich über Ursachen und Begleitumstände informieren. Dabei muss man sich keinesfalls auf die in der FCL vorhandenen Ausnahmeklassen beschränken, sondern kann eigene Ausnahmen definieren, z.B.:

```
using System;

public class BadFaculArgException : Exception {
   int type, value = -1;
   String input;

   public BadFaculArgException() {
   }
   public BadFaculArgException(String message) : base(message) {
   }
   public BadFaculArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
   public BadFaculArgException(String message, String input_, int type_, int value_)
      : this(message, input_, type_, value_, null) {
   }
   public BadFaculArgException(String message, String input_,
                               int type_, int value_, Exception innerException)
      : base(message, innerException) {
      input = input_;
      if (type_ >= 0 && type <= 3)
         type = type_;
      if (type_ == 4 && (value_ < 0 || value_ > 170)) {
         type = type_;
         value = value_;
      }
   }

   public String Input {get {return input;}}
   public int Type {get {return type;}}
   public int Value {get {return value;}}
}
```

Wir halten uns bei der Klasse `BadFaculArgException` an Microsofts Empfehlungen für selbst definierte Ausnahmeklassen:[1]

- Als Basisklasse sollte **System.Exception** verwendet werden.
- Der Klassenname sollte mit dem Wort *Exception* enden.

[1] http://msdn.microsoft.com/de-de/library/87cdya3t.aspx

- Die folgenden *allgemeinen Konstruktoren* sollten mit **public** - Verfügbarkeit implementiert werden:
  - Ein parameterfreier Konstruktor
  - Ein Konstruktor mit einem **String**-Parameter für die Fehlermeldung
  - Ein Konstruktor mit einem **String**- Parameter für die Fehlermeldung und einem **Exception**-Parameter für ein inneres Ausnahmeobjekt, das zuvor aufgefangen wurde und nun in ein informativeres Ausnahmeobjekt als Anlage aufgenommen wird.

Beim parameterlosen `BadFaculArgException`-Konstruktor beschränken wir uns auf den (impliziten) Aufruf des parameterlosen Basisklassenkonstruktors. Bei der restlichen Konstruktoren rufen wir explizit eine passende Überladung des Basisklassenkonstruktors auf, um die Instanzvariablen hinter den Eigenschaften **Message** und **InnerException** initialisieren zu können.

Über ein Objekt der selbstdefinierten Ausnahmeklasse `BadFaculArgException` kann ausführlich über Probleme mit Argumenten für die Fakultätsberechnung informiert werden:

- In der Eigenschaft **Message** (geerbt von **Exception**) steht wie üblich eine Fehlermeldung.
- In der Eigenschaft `Input` steht die zu konvertierende Zeichenfolge.
- In der Eigenschaft `Type` wird ein numerischer Indikator für die Fehlerart angeboten:
  - 0: Unbekannt
  - 1: Argument hat den Wert **null**
  - 2: Zeichenfolge kann nicht konvertiert werden
  - 3: **int**-Überlauf
  - 4: **int**-Wert außerhalb [0, 170]
- In der Eigenschaft `Value` steht das Konvertierungsergebnis (falls vorhanden, sonst -1).

Die endgültige `Kon2Int()` - Version kümmert sich um alle von **ToInt32.Parse()** zu befürchtenden Ausnahmen und wirft bei allen Fehlerursachen eine spezielle `BadFaculArgException`:[1]

```
static int Kon2Int(ref String instr) {
   int arg = -1;
   try {
      arg = Convert.ToInt32(instr);
      if (arg < 0  || arg > 170)
         throw new BadFaculArgException("Wert außerhalb [0, 170]", instr, 4, arg);
      else
         return arg;
   } catch (ArgumentNullException e) {
      throw new BadFaculArgException("Argument ist null", null, 1, -1, e);
   } catch (FormatException e) {
      String str = instr;
      bool ok = false;
      while (str.Length > 1 && !ok) {
         str = str.Substring(0, str.Length - 1);
         ok = Int32.TryParse(str, out arg);
      }
      if (ok) {
         instr = str;
         return arg;
      } else
         throw new BadFaculArgException("Fehler beim Konvertieren", instr, 2, -1, e);
   } catch (OverflowException e) {
      throw new BadFaculArgException("Integer-Überlauf", instr, 3, -1, e);
   }
}
```

[1] Die **ArgumentNullException** abzufangen, ist momentan überflüssig, weil an den Referenzparametzer von `Kon2Int()` das Referenzliteral **null** nicht übergeben werden kann.

Den in **catch**-Blöcken geworfenen `BadFaculArgException`-Objekten wird das aufgefangene Ausnahmeobjekt beigefügt, um dem Aufrufer keine Information vorzuenthalten.

In der **Main()** - Methode des Beispielprogramms kann eine abgefangene Ausnahme nun präzise protokolliert werden:

```
static void Main(string[] args) {
   int argument = -1;
   if (args.Length == 0) {
      Console.WriteLine("Kein Argument angegeben");
      Console.Read();
      Environment.Exit(1);
   }

   try {
      argument = Kon2Int(ref args[0]);
   }
   catch (BadFaculArgException e) {
      Console.WriteLine("Message:\t" + e.Message);
      Console.WriteLine("Fehlertyp:\t{0}\nZeichenfolge:\t{1} ", e.Type, e.Input);
      Console.WriteLine("Wert:    \t" + e.Value);
      if (e.InnerException != null)
         Console.WriteLine("Orig. Message:\t" + e.InnerException.Message);
      Console.Read();
      Environment.Exit(1);
   }
   double fakul = 1.0;
   for (int i = 1; i <= argument; i++)
      fakul = fakul * i;
   Console.WriteLine("Fakultät von {0}: {1}", args[0], fakul);
   Console.Read();
}
```

Bei einem Programmstart mit dem Kommandozeilenargument „vier“ resultiert z.B. die Ausgabe:

```
Message:         Fehler beim Konvertieren
Fehlertyp:       2
Zeichenfolge:    vier
Wert:            -1
Orig. Message:   Die Eingabezeichenfolge hat das falsche Format.
```

Eine eigene Ausnahmeklasse passt gut zur der Strategie, Probleme aus diversen Teilschritten einer Ausgabenbearbeitung an einer Stelle zusammen zu führen, wo gut über das weitere Vorgehen nach einem Scheitern entschieden werden kann. An dieser Entscheidungsstelle muss dann nur *ein* Ausnahmeobjekt behandelt werden, das genügend Informationen über die Art des Problems enthält.

Zum Transport von speziellen Zusatzinformationen benötigt eine (selbst erstellte) Ausnahmeklasse nicht unbedingt zusätzliche Felder bzw. Eigenschaften. Alternativ kann man in der **Exception**-Eigenschaft **Data** eine beliebig lange Schlüssel-Wert - Liste mit Elementen vom Typ **DictionaryEntry** unterbringen (siehe Abschnitt 12.4). Bei der Klasse `BadFaculArgException` könnten wir uns auf die folgende Standarddefinition beschränken:

```
public class BadFaculArgException : Exception {
   public BadFaculArgException() {
   }
   public BadFaculArgException(String message) : base(message) {
   }
   public BadFaculArgException(String message, Exception innerException)
      : base(message, innerException) {
   }
}
```

In die **Data**-Liste eines `BadFaculArgException`-Objekts lassen sich Schlüssel-Wert - Einträge mit den benötigten Zusatzinformationen z.B. per **Add()** aufnehmen:

```
BadFaculArgException bfa =
   new BadFaculArgException("Fehler beim Konvertieren", e);
bfa.Data.Add("Input", instr);
bfa.Data.Add("Type", 2);
bfa.Data.Add("Value", -1);
```

Ein **catch**-Block kann per Indexer

```
Console.WriteLine("Wert = {0}", e.Data["Value"]);
```

oder mit Hilfe der **DictionaryEntry**-Eigenschaften **Key** und **Value**

```
foreach (DictionaryEntry de in e.Data) {
   Console.WriteLine("{0}\t:\t{1}", de.Key, de.Value);
```

auf die **Data**-Listeneinträge eines Ausnahmeobjekts zugreifen.

12.7 Übungsaufgaben zu Kapitel 12

1) Erstellen Sie ein Syntaxdiagramm zur **try** - Anweisung (vgl. Abschnitt 12.2.1).

2) Im Beispielprogramm zur Demonstration von möglichen Sequenzen bei der Ausnahmebehandlung (siehe Abschnitt 12.2.2) verzichtet die Methode `Calc()` darauf, die potentiell von der Methode **Convert.ToInt32()** zu erwartende **OverflowException** abzufangen (vgl. Abschnitt 12.2.1). Bleibt die Ausnahme unbehandelt?

3) Beim Rechnen mit Gleitkommazahlen produziert C# in kritischen Situationen *keine* Ausnahmen, sondern operiert mit speziellen Werten wie **Double.POSITIVE_INFINITY** oder **Double.NaN**. Dieses Verhalten ist oft nützlich, kann aber die Fehlersuche erschweren, wenn mit den speziellen Funktionswerten weiter gerechnet wird, und erst am Ende eines längeren Rechenweges das Ergebnis **NaN** auftaucht (in der Ausgabe: **n. def.**). In folgendem Beispiel wird eine Methode namens `Log2()` zur Berechnung des dualen Logarithmus[1] verwendet, welche auf die FCL-Methode **Math.Log()** zurückgreift und daher bei ungeeigneten Argumenten (≤ 0) als Rückgabewert **Double.NaN** liefert.

Quellcode	Ausgabe
`using System;` `class DuaLog {` `  static double Log2(double arg) {` `  return Math.Log(arg) / Math.Log(2);` `  }` `  static void Main() {` `  double a = Log2(-1);` `  double b = Log2(8);` `  Console.WriteLine(a*b);` `  }` `}`	`n. def.`

Erstellen Sie eine Version, die bei ungeeigneten Argumenten eine **ArgumentOutOfRange-Exception** wirft.

---

[1] Für eine positive Zahl a ist ihr Logarithmus zur Basis b (> 0) definiert durch:

$$\log_b(a) := \frac{\ln(a)}{\ln(b)}$$

Dabei steht ln() für den natürlichen Logarithmus zur Basis e (Eulersche Zahl).

4) Erstellen Sie eine Variante der in Abschnitt 7.2.1 vorgestellten generischen Stapelverwaltungsklasse mit einer `Push()` - Methode, die bei besetztem Stapel eine **InvalidOperationException**-Ausnahme wirft, statt den Rückgabewert **false** zu liefern.

13 Attribute

An Typen (Klassen, Strukturen, Schnittstellen, usw.), Member (Methoden, Eigenschaften, usw.), Parameter und Rückgabewerte von Methoden sowie an Assemblies und Module (vgl. Abschnitt 1.2.5.3) kann man *Attribute* anheften, um zusätzliche Metainformationen bereit zu stellen, die beim Übersetzen und/oder zur Laufzeit berücksichtigt werden können. Attribute sind Objekte aus speziellen Klassen, die von der abstrakten Basisklasse **System.Attribute** abstammen. Der Compiler speichert die Attributobjekte als Metadaten im erzeugten Assembly. Bei einfachen Attributen besteht die Information über den Träger in der schlichten An- bzw. Abwesenheit des Attributs. Jedoch kann ein Attributobjekt auch Detailinformationen enthalten, die über Eigenschaften für Interessenten verfügbar sind.

Ein Attribut kann das Laufzeitverhalten eines Programms über seine Signalwirkung auf Methoden und/oder die CLR beeinflussen. Ein Beispiel ist das **SerializableAttribute**, mit dem für die Instanzen eines Typs das Serialisieren, d.h. das Speichern in einen Datenstrom (z.B. in eine Datei) erlaubt wird (vgl. Abschnitt 14.2.3). Fehlt dieses Attribut für eine Instanz, die in einen Serialisierungsstrom gerät (z.B. als direktes oder indirektes Mitglied eines anderen Typs), dann resultiert ein Ausnahmefehler.

Wir lernen mit den Attributen eine weitere Technik zur Kommunikation zwischen Programmbestandteilen kennen. Man kann die Attribute als Option zur *deklarativen Programmierung* auffassen. Sie ergänzen die im C# - Sprachumfang verankerten *Modifikatoren* für Typen, Methoden etc. und bieten dabei eine enorme Flexibilität.[1]

In komplexen objektorientierten Softwaresystemen (Frameworks) spielt generell die als *Reflexion* (engl.: *reflection*) bezeichnete Ermittlung von Informationen über Typen, Methoden usw. zur Laufzeit eine zunehmende Rolle. Dabei leisten Attribute einen wichtigen Beitrag.

Von *deklarativer Programmierung* sprachen wir auch bei der Gestaltung einer WPF-Bedienoberfläche per XAML-Code (siehe Abschnitt 4.11.2). Die aktuell behandelten, in den C# - Code zu platzierenden Attribute für Klassen, Methoden, Assemblies etc. sind streng von den Attributen von XAML-Elementen zu unterscheiden. Man kann aber durchaus im gemeinsamen deklarativen Ansatz der beiden Optionen zur Entwicklung von .Net - Software eine Ursache für die übereinstimmende Wortwahl sehen.

In der FCL wird von Attributen reichlich Gebrauch gemacht, was z.B. die Dokumentation zur Klasse **String** zeigt:

```
[SerializableAttribute]
[ComVisibleAttribute(true)]
public sealed class String : IComparable,
    ICloneable, IConvertible, IComparable<string>, IEnumerable<char>,
    IEnumerable, IEquatable<string>
```

---

[1] Gelegentlich wird im Zusammenhang mit unserem aktuellen Thema von *benutzerdefinierten Attributen* gesprochen (siehe z.B. Richter 2006, S. 403ff), und wichtige Methoden zur Auswertung von Attributen (siehe Abschnitt 13.2) führen das Wort *Custom* in ihrem Namen. Damit sollen die von der **System.Attribute** abstammenden Klassen, deren Objekte vom Software-Entwickler in deklarativer Absicht an diverse C# - Elemente angeheftet werden können, von deklarativen Bestandteilen der Programmiersprache C# unterschieden werden. Während die zum Sprachumfang gehörenden Attribute (wie z.B. die Modifikatoren **sealed**, **static**, **public** usw.) über viele Jahre kaum ergänzt werden, können **System.Attribute** - Ableitungen von den Software-Entwicklern jederzeit neu erstellt und verwendet werden. Wie die Deklarationen zur Klasse **String** zeigen, sind auch in der FCL benutzerdefinierte Attribute zahlreich vorhanden. Das Manuskript orientiert sich bei der Begriffsverwendung an der C# 5.0 - Sprachspezifikation (Microsoft 2012, Kap. 17). Dort ist nur von *Attributen* die Rede, und dabei sind die von **System.Attribute** abstammenden Klassen gemeint.

Hier wird über die Klasse **String** ausgesagt, dass ihre Objekte serialisiert werden dürfen, und dass die Klasse für die Zusammenarbeit mit traditionellen Windows-Komponenten (nach dem *Component Object Model*) geeignet ist. Was das konkret bedeutet, wird sich bei der noch ausstehenden Beschäftigung mit Objektserialisierung bzw. COM-Interoperabilität zeigen.

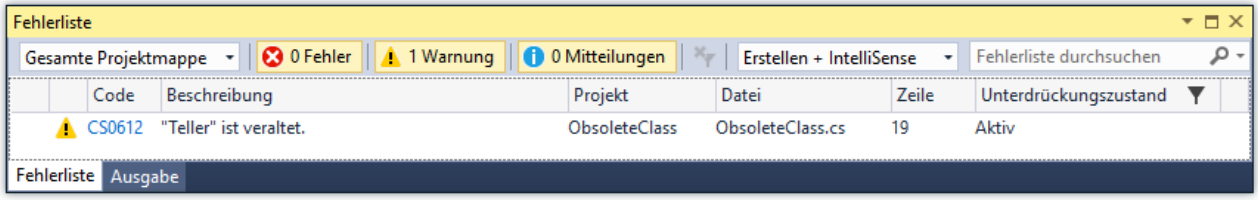
Wir müssen uns nicht auf die Vergabe und Auswertung von FCL-Attributen beschränken, sondern können auch eigene Attribute definieren.

13.1 Attribute vergeben

Sollen z.B. die Benutzer einer Klassenbibliothek dazu gebracht werden, einer neuen Klasse den Vorzug gegenüber einer alten zu geben, kann man der alten Klassendefinition zwischen eckigen Klammern das Attribut **Obsolete** voranstellen, so dass der Compiler bei Verwendung dieser Klasse automatisch eine Warnung ausgibt. Dies geschieht z.B. beim Übersetzen der folgenden Quelle:

```
using System;

[Obsolete]
public class Teller {
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }
}

public class NewTeller {
    public static void Tell() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}

class Prog {
    static void Main() {
        Teller.Tell();
        [veraltet] class Teller

        "Teller" ist veraltet.
    }
}
```

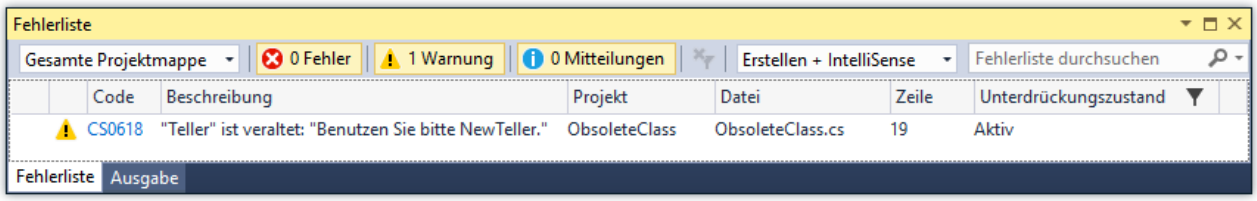
Im Editor der Entwicklungsumgebung wird der unerwünschte Zugriff auf die obsolete Klasse unterschlängelt und bei passender Mauszeigerpositionierung entsprechend kommentiert. In der **Fehlerliste** erscheint eine **Warnung**:



Wer als Klassenbibliotheksdesigner und -renovierer diese Compiler-Meldung zu dürftig findet, kann zum Erstellen des **Obsolete**-Attributs eine alternative Konstruktorüberladung verwenden und die **Message**-Eigenschaft des Objekts mit einer Zeichenfolge versorgen, z.B.:

```
[Obsolete("Benutzen Sie bitte NewTeller.")]
```

Dann meldet der Compiler beim unerwünschten Zugriff:



Bei der Vergabe eines Attributes wird durch einen Konstruktor der Attributklasse ein Objekt erstellt, das die Metadaten der Trägers ergänzt.

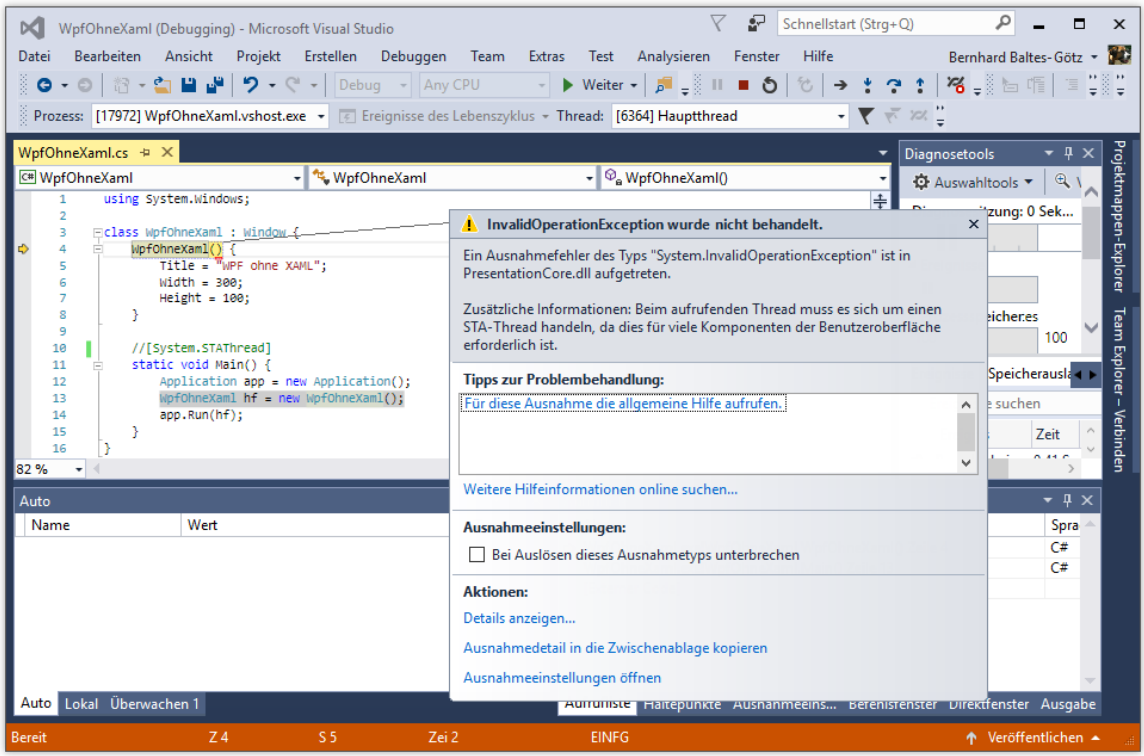
Je nach verwendeter Konstruktorüberladung sind Aktualparameter anzugeben, wobei eine leere Aktualparameterliste weggelassen werden kann. Als weitere syntaktische Besonderheit darf bei der Vergabe eines Attributs der Namensteil *Attribute* im Klassennamen entfallen. Im Beispiel kommt also die Klasse **ObsoleteAttribute** (aus dem Namensraum **System**) zum Einsatz.

Neben Klassen können auch andere Programmbestandteile mit Attributen versehen werden, z.B. Methoden:

```
using System;

class Teller {
    [Obsolete("Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx().")]
    public static void Tell() {
        Console.WriteLine("Hallo!");
    }

    public static void TellEx() {
        Console.WriteLine("Hallo, Wilhelm!");
    }
}

class Prog {
    static void Main() {
        Teller.Tell();
    }
}
```

```
class Teller

"Teller.Tell()" ist veraltet: "Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx()."
```

Wie Sie aus dem Abschnitt 11.2.1 wissen, muss bei einer WPF-Anwendung zur Methode **Main()** generell das **STAThreadAttribut** vergeben werden, weil ansonsten schon die Ausführung des Fensterklassenkonstruktors an einer **InvalidOperationException** scheitert, z.B.:



Mit dem **STAThreadAttribut** wird signalisiert, dass bei der Kooperation mit dem Component Object Model (COM), der noch sehr weit verbreiteten Windows-Komponententechnologie, das COM-Threading-Modell STA (*Singlethread-Apartment*) zum Einsatz kommen soll, um Synchronisationsprobleme bei Steuerelementen zu verhindern.

Im weiteren Verlauf von Kapitel 13 wird noch klarer, dass man bei der Vergabe von Attributen in der Regel nicht nur den Quellcode kommentiert und den Compiler informiert, sondern signalisierend den Programmablauf beeinflusst, sofern andere Akteure (z.B. andere Typen oder die CLR) die Attribute kennen und bei ihrem Verhalten berücksichtigen.

13.2 Attribute per Reflexion auswerten

Das .NET - Framework bietet leistungsfähige *Reflexionstechniken*, die es u.a. erlauben, die Attributausstattung von Programmbestandteilen zur Laufzeit zu analysieren. Mit der statischen Methode **IsDefined()** der Klasse **System.Attribute** kann man feststellen, ob ein bestimmtes Attribut vorhanden ist. Das folgende Programm prüft für eine Klasse und für eine Methode, ob das **ObsoleteAttribute** angeheftet ist:

```
using System;
using System.Reflection;

[Obsolete("Benutzen Sie bitte NewTeller.")]
public class Teller {
   [Obsolete("Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx().")]
   public static void Tell() {
      Console.WriteLine("Hallo!");
   }
}
```

```
class Prog {
   static void Main() {
      Type typeTeller = typeof(Teller);
      Type typeObsolete = typeof(ObsoleteAttribute);

      if (Attribute.IsDefined(typeTeller, typeObsolete))
         Console.WriteLine("Der Typ {0} ist obsolet", typeTeller.Name);

      MemberInfo[] mi = typeTeller.FindMembers(MemberTypes.Method,
         BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
         Type.FilterName, "Tell");
      if (Attribute.IsDefined(mi[0], typeObsolete))
         Console.WriteLine("\nDie Methode {0} ist obsolet", mi[0].Name);
   }
}
```

Beide Prüflinge werden als obsolete erkannt:

```
Der Typ Teller ist obsolet

Die Methode Tell ist obsolet
```

Die verwendete **IsDefined()** - Überladung erwartet als ersten Parameter ein Objekt der Klasse **MemberInfo** aus dem Namensraum **System.Reflection**, von der auch die Klasse **Type** abstammt, die einen Datentyp (Klasse, Struktur, Schnittstelle etc.) repräsentiert. Im Beispiel wird **IsDefined()** zweimal aufgerufen:

- Im ersten Aufruf ist eine Klasse der potentielle Attributträger:
  `Attribute.IsDefined(typeTeller, typeObsolete)`
- Im zweiten Aufruf wird die Anwesenheit eines Methodenattributs untersucht:
  `Attribute.IsDefined(mi[0], typeObsolete)`

Als zweiter Parameter ist das **Type**-Objekt zur fraglichen Attributklasse anzugeben.

Im Beispiel werden Referenzen auf **Type**-Objekte per **typeof**-Operator gewonnen. Anders als bei der Attributvergabe (siehe Abschnitt 13.1) ist dabei der Name der Attributklasse vollständig (inkl. Namenbestandteil *Attribute*) zu schreiben, z.B.:

```
Type typeObsolete = typeof(ObsoleteAttribute);
```

Das im zweiten **IsDefined()** - Aufruf benötigte **MemberInfo**-Objekt zur `Teller`-Methode `Tell()` besorgt die Instanzmethode **FindMembers()** der Klasse **Type**. Sie liefert Information über die Member des befragten **Type**-Objekts in einem Array vom Typ **MemberInfo**:

- Im ersten **FindMembers()** - Parameter wählt man die Member-Kategorie.
- Mit dem zweiten Parameter lässt sich die Suche über eine ODER-Verknüpfung von Werten der Enumeration **BindingFlags** steuern. Im Beispiel werden öffentliche Instanz- und Klassen-Member zugelassen.
- Der dritte Parameter sorgt im Beispiel für eine namensorientierte Filterung.
- Im letzten Parameter spezifiziert man den geforderten Namen, wobei auch das Jokerzeichen ***** genutzt werden kann.

Soll nicht nur die Existenz eines Attributs festgestellt, sondern auch sein Innenleben exploriert werden, eignet sich die statische Methode **GetCustomAttribute()** der Klasse **System.Attribute**. Sie rekonstruiert das angeheftete Objekt aus den Metadaten im Assembly, so dass öffentliche Felder und Eigenschaften zur Verfügung stehen. Wegen der Objektkreation sind die Kosten eines Aufrufs höher als bei der Methode **IsDefined()**. Die folgende Methode `ObsoleteMethodCheck()` prüft für alle öffentlichen Methoden eines Typs, ob sie als obsolet markiert sind:

```
static void ObsoleteMethodCheck(Type tt) {
 Attribute attrib;
 MemberInfo[] members = tt.FindMembers(MemberTypes.Method,
  BindingFlags.Instance | BindingFlags.Static | BindingFlags.Public,
  Type.FilterName, "*");
 Console.WriteLine("Obsolete-Prüfung für die Methoden des Typs {0}:",
                   tt.FullName);
 foreach (MemberInfo mi in members) {
  attrib = Attribute.GetCustomAttribute(mi, typeof(ObsoleteAttribute));
  if (attrib != null) {
   Console.WriteLine("\nDie Methode {0}() ist obsolet.", mi.Name);
   Console.WriteLine("Message: " + (attrib as ObsoleteAttribute).Message);
  } else
   Console.WriteLine("\nDie Methode {0}() ist noch aktuell.", mi.Name);
 }
}
```

Per **GetCustomAttribute()** erhalten wir für jede Methode ggf. das angeheftete **ObsoleteAttribute**-Objekt und fragen dieses Objekt nach seiner **Message**-Eigenschaft.

Über die Klasse

```
class Teller {
   [Obsolete("Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx().")]
   public static void Tell() {
      Console.WriteLine("Hallo!");
   }
   public static void TellEx() {
      Console.WriteLine("Hallo, Wilhelm!");
   }
}
```

erhalten wir den Bericht:

```
Obsolete-Prüfung für die Methoden der Klasse Teller:

Die Methode Tell() ist obsolet
Message: Der Support für Tell() läuft aus. Benutzen Sie bitte TellEx().

Die Methode TellEx() ist noch aktuell.

Die Methode ToString() ist noch aktuell.

Die Methode Equals() ist noch aktuell.

Die Methode GetHashCode() ist noch aktuell.

Die Methode GetType() ist noch aktuell.
```

In der **Main()** - Methode des nächsten Beispielprogramms werden mit der statischen **Attribute**-Methode **GetCustomAttributes()** für eine per **Type**-Objekt beschriebene Klasse *alle* angehefteten benutzerdefinierten Attribute ermittelt, wobei *geerbte* Attribute nicht interessieren (Wert **false** für den Parameter **inherit**):

Quellcode	Ausgabe
`using System;` `[Obsolete] [Serializable]` `class Teller {` `  public static void Tell() {` `  Console.WriteLine("Hallo!");` `  }` `}` `class Prog {` `  static void Main() {` `  Type type = typeof(Teller);` `  Attribute[] atar =` `  Attribute.GetCustomAttributes(type, false);` `  foreach (Attribute at in atar)` `  Console.WriteLine(at);` `  }` `}`	`System.ObsoleteAttribute` `System.SerializableAttribute`

Mit alternativen Überladungen lassen sich Assemblies, Member, Parameter etc. analog untersuchen.

13.3 Attribute definieren

Bei einer eigenen Attributklasse sollte man …

- die Basisklasse **System.Attribute** verwenden,
- den Klassennamen mit dem Wort *Attribute* enden lassen.

Um den Compiler darüber zu informieren, welchen Programmbestandteilen das Attribut angeheftet werden darf, verwendet man ein Attribut aus der Klasse **AttributeUsageAttribute**. Im folgenden Beispiel erhält der Konstruktorparameter **validOn** den Wert **AttributeTargets.Class**, so dass nur *Klassen* das neu definierte `NonsenseAttribute` erhalten dürfen. Außerdem wird mit dem Wert **false** für die **AttributeUsageAttribute**-Eigenschaft **Inherited** verhindert, dass eine dekorierte Klasse das `NonsenseAttribute` an abgeleitete Klassen weitergibt:

```
[AttributeUsage(AttributeTargets.Class, Inherited=false)]
public class NonsenseAttribute : Attribute {
   int level;
   public NonsenseAttribute(int level_) {
      level = level_;
   }
   public int Level {
      get {return level;}
   }
}
```

Bei der Attributvergabe kommt zur Eigenschaftsinitialisierung innerhalb der Konstruktor-Parameterliste eine spezielle Syntax unter Verwendung von Name-Wert - Paaren zum Einsatz. Man spricht hier von *Namensparametern*, die *nach* den regulären Parametern (hier *Positionsparameter* genannt) in beliebiger Reihenfolge stehen dürfen. Namensparameter werden nicht als Konstruktorargument definiert. Stattdessen kann jede öffentliche instanzbezogene Eigenschaft oder Variable als Namensparameter verwendet und auf diese Weise initialisiert werden. Unter der Bezeichnung *Objekt-* bzw. *Instanz-Initialisierer* besteht übrigens seit C# 3.0 für beliebige Klassen eine ähnliche Initialisierungsmöglichkeit zur Verfügung (siehe Abschnitt 4.4.3.2).

Bei Positions- oder Namensparameter von **Attribut**-Konstruktoren sind ausschließlich die folgenden Datentypen erlaubt:

- **bool**, **byte**, **char**, **short**, **int**, **long**, **float**, **double**
- **Object**, **String**, **Type** (alle aus dem Namensraum **System**)
- Aufzählungstypen
- Eindimensionale Arrays mit einem Elementtyp aus der obigen Liste

In einer Übungsaufgabe zum aktuellen Kapitel sollen Sie das eben definierte Attribut auf eine Klasse anwenden und zur Laufzeit auswerten (siehe Abschnitt 13.6).

13.4 Attribute für Assemblies und Module

Auch Assemblies und Module können Attribute erhalten, wobei aber mangels syntaktischer Entsprechung für diese Übersetzungseinheiten der Bezug nicht durch die Platzierung der Attribute im Quellcode hergestellt werden kann. Stattdessen benutzt man Attribute mit expliziter Widmung, z.B.:

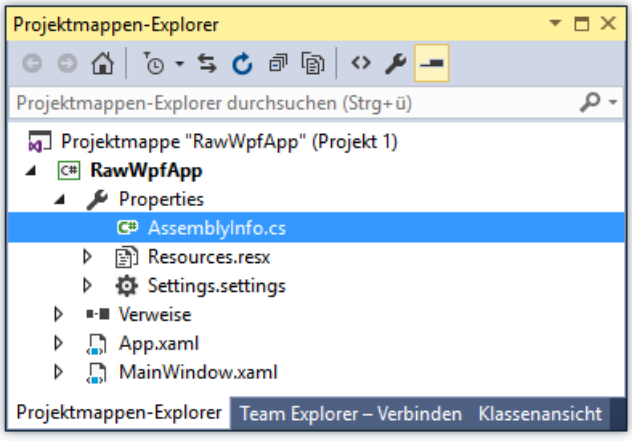
```
[assembly: AssemblyCompany("Marco Saft")]
[assembly: AssemblyProduct("YourTools")]
[assembly: AssemblyVersion("1.4.2.3")]
```

Hier wird jeweils das vom Compiler zu erzeugende Assembly als Träger des nachfolgenden Attributs festgelegt.

Zu einer neuen **WPF-Anwendung** erstellt das Visual Studio die Datei **AssemblyInfo.cs** mit Vorschlägen für wichtige Assembly-Attribut - Deklarationen, z.B. (**using**-Direktiven und Kommentare aus Platzgründen weggelassen):

```
[assembly: AssemblyTitle("RawWpfApp")]
[assembly: AssemblyDescription("")]
[assembly: AssemblyConfiguration("")]
[assembly: AssemblyCompany("Universität Trier")]
[assembly: AssemblyProduct("RawWpfApp")]
[assembly: AssemblyCopyright("Copyright © Universität Trier 2017")]
[assembly: AssemblyTrademark("")]
[assembly: AssemblyCulture("")]
[assembly: ComVisible(false)]
[assembly: ThemeInfo(ResourceDictionaryLocation.None,
                     ResourceDictionaryLocation.SourceAssembly)]
[assembly: AssemblyVersion("1.0.0.0")]
[assembly: AssemblyFileVersion("1.0.0.0")]
```

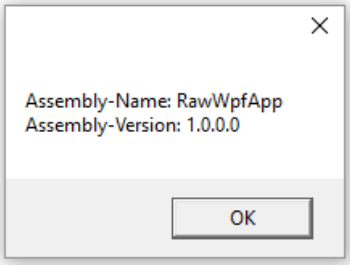
Zur Modifikation dieser Attribute öffnet man die Datei **AssemblyInfo.cs** über den Projektmappen-Explorer:



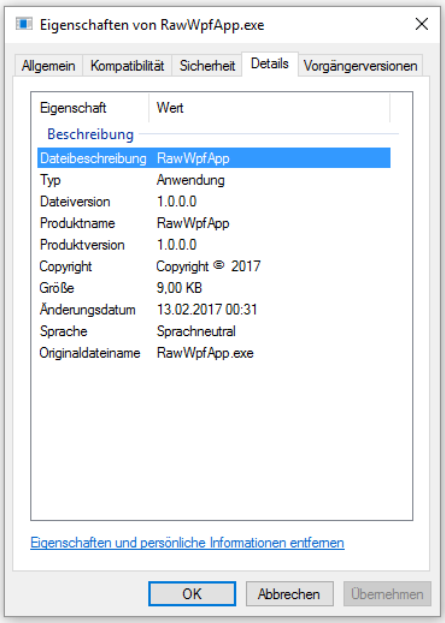
In einem Programm kann man auf einige Attribute des gerade ausgeführten Assemblies über Eigenschaften des zugehörigen **AssemblyName**-Objekts zugreifen, z.B.:

```
private void button_Click(object sender, RoutedEventArgs e) {
 System.Reflection.Assembly ass = System.Reflection.Assembly.GetExecutingAssembly();
 MessageBox.Show("Assembly-Name: " + ass.GetName().Name+
                 "\nAssembly-Version: " + ass.GetName().Version);
}
```

Aufgrund der obigen Deklarationen in **AssemblyInfo.cs** erhält man folgende MessageBox:



Außerdem erscheinen die Assembly-Attribute im Eigenschaftsdialog einer Assembly-Datei, z.B.:



13.5 Eine Auswahl nützlicher FCL-Attribute

13.5.1 Bitfelder per FlagsAttribute

Bei einem Enumerationstyp (vgl. Abschnitt 5.5) signalisiert der Designer mit dem **System.FlagsAttribute**, dass ein Wert als *Bitfeld* interpretierbar ist, d.h.:

- Die ersten *k* Bits (mit dem niederwertigsten beginnend) des zugrunde liegenden Datentyps (meist **int**) stehen als unabhängige Informationsträger jeweils für ein dichotomes Merkmal (mit den Werten 0 und1). Ein Enumerationswert kodiert also die Ausprägungen von *k* dichotomen Merkmalen. Bei der Enumeration **ModifierKeys** aus dem Namensraum **System.Windows.Input** stehen die ersten vier Bits für die Vorschalttasten **Alt**, **Strg**, **Umschalt** und **Windows**:[1]

```
[Flags]
 . . .
public enum ModifierKeys {
   None = 0,
   Alt = 1,
   Control = 2,
   Shift = 4,
   Windows = 8
}
```

- Jede bitweise ODER-Kombination von zwei benannten Werten der Enumeration ergibt ein sinnvoll interpretierbares Bitfeld, also eine zulässige Kombination der dichotomen Einzelmerkmale. Mit dem folgenden **ModifierKeys**-Wert lässt sich z.B. prüfen, ob die **Strg**- und die Umschalttaste simultan gedrückt sind:

```
ModifierKeys.Control | ModifierKeys.Shift
```

Bei einem gewöhnlichen Enumerationstyp (ohne **FlagsAttribute**) …

- stehen die Werte für die sich *gegenseitig ausschließenden* Ausprägungen *eines* Merkmals. Z.B. kodieren bei der in Abschnitt 11.7.2.2 erwähnten Enumeration **HorizontalAlignment** die ersten vier nicht-negativen **int**-Werte jeweils eine horizontale Orientierung eines WPF-Steuerelements gegenüber der umgebenden Containerzelle (**Left**, **Center**, **Right**, **Stretch**):[1]

```
public enum HorizontalAlignment {
   Left = 0,
   Center = 1,
   Right = 2,
   Stretch = 3,

}
```

- Eine bitweise ODER-Verknüpfung der Werte ist zwar syntaktisch erlaubt, aber sinnlos.

In der FCL-Enumerationsbasisklasse **Enum** wird die Existenz des **Flags**-Attributs per **IsDefined()** -Aufruf (vgl. Abschnitt 13.2) überprüft[2]

```
if (!eT.IsDefined(typeof(System.FlagsAttribute), false)) {...}
else {...}
```

und ggf. in der **ToString()** - Überschreibung für jeden Wert der Enumeration eine kommaseparierte Liste der Merkmale mit einem angeschalteten Bit ausgegeben. Hier orientiert also eine Methode ihr Verhalten an der Anwesenheit eines Attributs. Weil alle Enumerationen die **ToString()** - Implementation der Klasse **Enum** erben, kann im folgenden Programm demonstriert werden, dass die Summe von zwei **ModifierKeys** -Werten wieder ein sinnvoller Wert dieses Typs ist:[3]

---

[1] Der Quellcode stammt aus den Dateien **ModifierKeys.cs** bzw. **FrameworkElement.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden können.

[2] Der Quellcode stammt aus der Datei **enum.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann.

[3] Das Programm benötigt Verweise auf die Assemblies **PresentationFramework.dll** und **WindowsBase.dll**.

Quellcode	Ausgabe
`using System;` `using System.Windows;` `using System.Windows.Input;` `class Prog {` `  static void Main() {` `  Console.WriteLine("ModifierKeys-Werte:");` `  for (ModifierKeys i = 0; (int)i <= 8; i++)` `  Console.WriteLine(" {0}: {1}", (int)i, i);` `  Console.WriteLine("\nHorizontalAlignment-Werte:");` `  for (HorizontalAlignment i = 0; (int)i <= 8; i++)` `  Console.WriteLine(" {0}: {1}", (int)i, i);` `  }` `}`	`ModifierKeys-Werte:` ` 0: None` ` 1: Alt` ` 2: Control` ` 3: Alt, Control` ` 4: Shift` ` 5: Alt, Shift` ` 6: Control, Shift` ` 7: Alt, Control, Shift` ` 8: Windows` `HorizontalAlignment-Werte:` ` 0: Left` ` 1: Center` ` 2: Right` ` 3: Stretch` ` 4: 4` ` 5: 5` ` 6: 6` ` 7: 7` ` 8: 8`

Bei der Enumeration **HorizontalAlignment** gilt die analoge Aussage nicht.

13.5.2 Unions per StructLayoutAttribute und FieldOffsetAttribute

Bei der in Abschnitt 3.3.5.1 zur Erläuterung der binären Gleitkommadarstellung benutzten (aber nicht erklärten) Anwendung `FloatBits` werden die Attribute **StructLayoutAttribute** und **FieldOffsetAttribute** aus dem Namensraum **System.Runtime.InteropServices** dazu verwendet, eine **Union** im Sinn der Programmiersprache C nachzubilden. In unseren Begriffen handelt es sich dabei um eine Struktur, deren Instanzvariablen im Speicher (zumindest teilweise) überlappen. In der Regel soll damit nicht etwa Speicherplatz gespart, sondern eine unterschiedliche Interpretation desselben Speicherinhalts ermöglicht werden.

In den folgenden Zeilen wird eine Struktur mit dem (frei gewählten) Namen `Union` sowie Feldern von Typ **float** und **int** definiert:

```
[StructLayout(LayoutKind.Explicit)]
public struct Union {
  [FieldOffset(0)]
  public float f;
  [FieldOffset(0)]
  public int i;
}
```

Per **StuctLayoutAttribut** mit Konstruktor-Parameter **LayoutKind.Explicit** wird dem Compiler mitgeteilt, dass die Speicheradressen der beiden Felder explizit durch **FieldOffsetAttribut**e festgelegt werden sollen. So wird es möglich, die beiden Felder an derselben Anfangsadresse 0 beginnen zu lassen. Die beiden Typen (**float**, **int**) haben denselben Platzbedarf von vier Bytes (siehe Abschnitt 3.3.4).

Das Programm `FloatBits` schreibt den vom Benutzer angegebenen **float**-Wert in das `f`-Feld einer `Union`-Instanz und liest anschließend über das `i`-Feld der Instanz aus demselben Speicherbereich einen **int**-Wert, der über bitorientierte Operatoren (siehe Abschnitt 3.5.6) untersucht werden kann:

```
using System;
using System.Runtime.InteropServices;

class FloatBits {
   static void Main() {
      Union uni = new Union();
      float f;
      Console.Write("float: ");
      f = Convert.ToSingle(Console.ReadLine());
      uni.f = f;
      int bits = uni.i;
      Console.WriteLine("\nBits:\n1 12345678 12345678901234567890123");
      for (int i = 31; i >= 0; i--) {
         if (i == 30 || i == 22)
            Console.Write(' ');
         if ((1 << i & bits) != 0)
            Console.Write('1');
         else
            Console.Write('0');
      }
   }
}
```

13.6 Übungsaufgaben zu Kapitel 13

1) Ergänzen Sie im Beispiel von Abschnitt 13.3 die ziemlich sinnlose Klasse `Dummy`

```
[Serializable] [NonsenseAttribute(13)]
class Dummy {
}
```

und eine Testklasse mit **Main()** - Methode, die alle benutzerdefinierten Attribute der Klasse `Dummy` und den `Level`-Wert des `NonsenseAttribut`-Objekts ausgibt.

14 Ein- und Ausgabe über Datenströme

Praktisch jedes Programm muss Daten aus externen Quellen einlesen und/oder Verarbeitungsergebnisse in externe Senken schreiben. Wir haben uns bisher auf die Eingabe per Tastatur sowie die Ausgabe auf den Bildschirm beschränkt und müssen allmählich alternative Quellen bzw. Senken kennen lernen (z.B. Dateien, Netzwerkverbindungen, Datenbankserver). Im aktuellen Kapitel werden auf der **Datenstrom**-Abstraktion basierende Verfahren vorgestellt, mit denen Werte elementarer Typen (z.B. **int**, **double**), Zeichenfolgen oder beliebige Objekte in **Dateien** geschrieben bzw. von dort gelesen werden können. Außerdem werden wir uns mit der Verwaltung von Dateien und Verzeichnissen beschäftigen.

Vorausblick auf zwei verwandte Themen:

- In einem späteren Kapitel werden Sie mit den **Netzwerkverbindungen** weitere, außerordentlich wichtige Datenquellen bzw. -senken kennen lernen und dabei von Ihren Kenntnissen über die Datenstromtechnik profitieren.
- Im Kapitel über **Datenbankprogrammierung** werden anspruchsvolle Datenverwaltungstechniken vorgestellt, die sich auch im Netzwerk- und Mehrbenutzerkontext bewähren. Dabei überlassen wir den direkten Kontakt mit Dateien einer speziellen Software, dem Datenbankmanagement-System (DBMS).

Die .NET - Klassen zur Datenein- und -ausgabe befinden sich im Namensraum **System.IO**, den wir folglich bei Quellcodedateien mit entsprechender Funktionalität in der Regel zu Beginn importieren:

```
using System.IO;
```

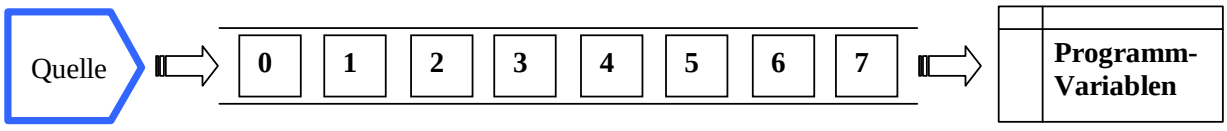
14.1 Datenströme aus Bytes

Praktisch alle Datentypen sind schlussendlich aus Bytes zusammengesetzt, und moderne Computer-Hardware ist so konstruiert, dass Bytes (einzeln oder in Gruppen) effizient verarbeitet werden können. Daher stützt sich ein beliebiger Datenstrom letztlich auf einen Strom aus Bytes, und wir befassen uns zunächst mit diesem Basisstrom.

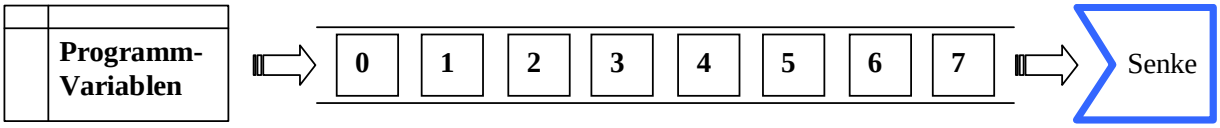
14.1.1 Das Grundprinzip

Im .NET - Framework wird die Ein- und Ausgabe von Daten über so genannte *Ströme* (engl.: ***streams*)** abgewickelt.

Ein Programm **liest** Daten aus einem **Eingabestrom**, der aus einer Datenquelle (z.B. Datei, Eingabegerät, Netzwerkverbindung) gespeist wird:



Ein Programm **schreibt** Daten in einen **Ausgabestrom**, der die Werte von Programmvariablen zu einer Datensenke befördert (z.B. Datei, Ausgabegerät, Netzverbindung):



Ein- bzw. Ausgabeströme werden in .NET - Programmen durch Objekte aus Klassen des Namensraums **System.IO** repräsentiert, wobei die Auswahl u.a. von der angeschlossenen Datenquelle bzw. -senke (z.B. Datei versus Netzverbindung) sowie vom Typ der zu transportierenden Daten abhängt.

Mit dem Datenstromkonzept wird bezweckt, Anweisungen zur Ein- oder Ausgabe von Daten möglichst unabhängig von den Besonderheiten konkreter Datenquellen und -senken formulieren zu können.

14.1.2 Beispiel

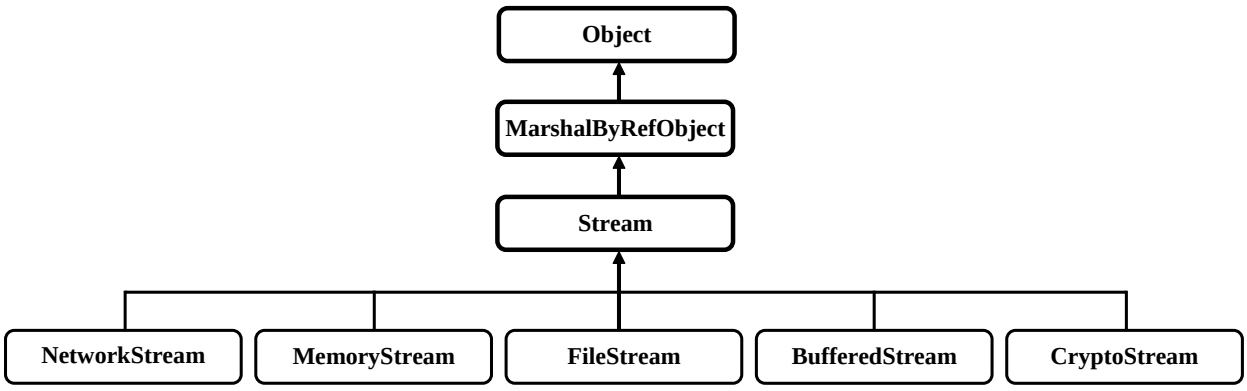
Nach so vielen allgemeinen bzw. abstrakten Bemerkungen wird es Zeit für ein konkretes Beispiel, wobei der Einfachheit halber auf Praxisnähe und Ausnahmebehandlung verzichtet wird. Das folgende Programm erstellt eine Datei, schreibt einen **byte**-Array hinein, liest die Daten wieder zurück und löscht schließlich die Datei. Über den Dateieröffnungsmodus **FileMode.CreateNew** (vgl. Abschnitt 14.1.6.1) wird dafür gesorgt, dass nur eine *neu angelegte* Ausgabedatei in Frage kommt:

Quellcode	Ausgabe

```
using System;
using System.IO;

class FSDemo {
   static void Main() {
      String name = "demo.bin";
      byte[] arr = {0,1,2,3,4,5,6,7};
      FileStream fs = new FileStream(name, FileMode.CreateNew);
      fs.Write(arr, 0, arr.Length);
      fs.Position = 0;
      fs.Read(arr, 0, arr.Length);
      foreach (byte b in arr)
         Console.WriteLine(b);
      fs.Close();
      File.Delete(name);
   }
}
```

```
0
1
2
3
4
5
6
7
```

Für die Ein- und die Ausgabe wird ein Objekt der Klasse **FileStream** eingesetzt. Diese Spezialisierung der abstrakten Basisklasse **Stream** implementiert das Stromkonzept für *Dateien*. Zur Einordnung ist hier ein kleiner Ausschnitt aus der **Stream**-Klassenhierarchie wiedergegeben:



Mit den Details des Beispielprogramms beschäftigen wir uns gleich.

14.1.3 Wichtige Methoden und Eigenschaften der Basisklasse Stream

Alle Ableitungen der abstrakten Basisklasse **Stream** verfügen u.a. über die folgenden Methoden und Eigenschaften:

- **public long Position {get; set;}**
  Über diese Eigenschaft wird die aktuelle Position im Strom angesprochen, an der das nächste Byte gelesen bzw. geschrieben wird. Im Beispielprogramm von Abschnitt 14.1.2 wird die Position der geöffneten Datei mit der folgenden Anweisung auf den Dateianfang zurück gesetzt:

  ```
  fs.Position = 0;
  ```

- **public int ReadByte()**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *ein* Byte per Rückgabewert vom Typ **int** zu liefern und seine Position entsprechend zu erhöhen. Ist das Ende des Stroms erreicht, wird der Rückgabewert -1 geliefert.

- **public int Read(byte[]** *buffer***, int** *offset***, int** *count***)**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *count* Bytes zu liefern, im **byte**-Array *buffer* ab Position *offset* abzulegen und seine Position entsprechend zu erhöhen. Als Rückgabewert erhält man die Anzahl der tatsächlich gelieferten Bytes, die bei unzureichendem Vorrat kleiner als *count* ausfallen kann.

- **public void WriteByte()**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *ein* Byte zu schreiben und seine Position entsprechend zu erhöhen.

- **public void Write(byte[]** *buffer***, int** *offset***, int** *count***)**
  Mit dieser Methode wird ein **Stream**-Objekt aufgefordert, *count* Bytes zu schreiben, die im **byte**-Array *buffer* ab Position *offset* liegen und seine Position entsprechend zu erhöhen.

- **public void Flush()**
  Viele Strom-Objekte verwenden beim Schreiben aus Performanzgründen einen Puffer, um die Anzahl der zeitaufwendigen Zugriffe auf eine angeschlossene Senke (z.B. Datei) möglichst gering zu halten. Mit der Methode **Flush()** verlangt man die sofortige Ausgabe des Puffers, so dass der komplette Inhalt für Abnehmer zur Verfügung steht. Bei der Klasse **FileStream** wird eine voreingestellte Puffergröße von 4096 Bytes benutzt.

- **public void Close()**
  public void Dispose()
  Mit dem Schließen eines Datenstroms per **Close()** - Aufruf oder durch eine alternative Technik beschäftigen wir uns gleich in Abschnitt 14.1.4. Statt **Close()** kann auch **Dispose()** aufgerufen werden.

- **public long Length {get;}**
  Mit dieser Eigenschaft wird die Länge des Stroms (z.B. die Dateigröße) in Bytes angesprochen.

- **public long Seek(long** *offset***, SeekOrign** *origin***)**
  Diese Methode fordert einen Strom auf, seine Position relativ zu einem per **SeekOrigin**-Wert festgelegten Bezugspunkt (**Begin**, **Current**, **End**) neu zu setzen, z.B. vom aktuellen Stand aus um vier Bytes zurück:

  ```
  fs.Seek(-4, SeekOrigin.Current);
  ```

  Als Rückgabe erhält man die neue Position.

- **public bool CanRead {get;}**, **public bool CanSeek {get;}**, **public bool CanWrite {get;}**
  Über diese Eigenschaften lässt sich feststellen, ob ein Strom das Lesen, Schreiben oder Positionieren erlaubt, z.B.:

Quellcode-Segment	Ausgabe
`Console.WriteLine("CanRead:  " + fs.CanRead);` `Console.WriteLine("CanSeek:  " + fs.CanSeek);` `Console.WriteLine("CanWrite: " + fs.CanWrite);`	`CanRead:  True` `CanSeek:  True` `CanWrite: True`

Die beschriebenen Ein-/Ausgabemethoden arbeiten synchron, blockieren also den aktuellen Thread bis zu ihrer Rückkehr, was sehr variable und oft inakzeptable Wartezeiten verursachen kann. In Kapitel 15 werden asynchrone, nicht blockierende Ein-/Ausgabemethoden vorgestellt (siehe Abschnitte 15.5.5 und 15.6.1.2).

14.1.4 Schließen von Datenströmen

Nachdem ein Datenstromobjekt seine Aufgabe erfüllt hat, muss es geschlossen werden, um den Verlust von Daten und das unnütze Blockieren von Ressourcen zu vermeiden. Bei den von **Stream** abgeleiteten Klassen hat das Schließen folgende Effekte:

- Wenn ein Puffer vorhanden ist (z.B. bei der Klasse **FileStream**), wird er durch einen automatischen **Flush()** - Aufruf entleert.
- Betriebssystem-Ressourcen (z.B. Datei-Handles, Netzwerk-Sockets) werden freigegeben.

14.1.4.1 Close(), Dispose()

Um einen Datenstrom aus der **Stream**-Hierarchie zu schließen, kann man seine **Close()** - oder **Dispose()** - Methode aufrufen, z.B.:

```
fs.Close();
```

Diese beiden parameterfreien Methoden rufen letztlich eine **Dispose()** - Überladung mit **bool**-Parameter auf, welche die eigentlichen Aufräumungsarbeiten (inklusive Entleeren des Puffers) verrichtet. Die Klasse **Stream** muss eine **Dispose**() - Methode bereithalten, weil sie das Interface **IDisposable** (man sagt auch: *das Beseitigungsmuster*) implementiert.

Nach einem **Close()** - bzw. **Dispose()** - Aufruf existiert das angesprochene .NET - Objekt weiterhin, doch führen Lese- bzw. Schreibversuche zu Ausnahmefehlern. Rufen Sie **Close()** bzw. **Dispose()** also *nicht* auf, wenn solche Ausnahmefehler möglich sind, weil im Programm noch weitere Referenzen auf das **Stream**-Objekt existieren.

14.1.4.2 Garbage Collector

Die eben beschriebene Panne ist ausgeschlossen, wenn man überflüssig gewordene Datenstromobjekte dem Garbage Collector überlässt, was Richter (2006, S. 501) nachdrücklich empfiehlt. Über die Finalisierungsmethode (vgl. Abschnitt 4.4.4) der von **Stream** abgeleiteten Klassen ist sichergestellt, dass vor dem Entfernen eines Objekts per Garbage Collector alle verwendeten Ressourcen (Dateien, Netzwerkverbindungen) freigegeben werden, wobei die eben erwähnte eine **Dispose()** - Überladung mit **bool**-Parameter zum Einsatz kommt.

Bei den Aufräumungsarbeiten des Garbage Collectors sind allerdings Zeitpunkt und Reihenfolge unbestimmt. Setzt z.B. im Rahmen einer schreibenden Datenstrom-Verarbeitungskette ein Ausgabeobjekt mit eigenem Puffer (z.B. aus der Klasse **StreamWriter)** auf einem **Stream**-Objekt auf, darf man dem Garbage Collector das Schließen auf keinen Fall überlassen (siehe Abschnitt 14.2.2). Es kommt zu Datenverlusten wenn der Garbage Collector bei seinen Aufräumarbeiten (ohne garantierte Reihenfolge!) das **Stream**-Objekt *vor* dem **StreamWriter**-Objekt beseitigt.

Unter ungünstigen Umständen kann die Finalisierungsmethode zu einem (Ausgabe-)Objekt von der CLR überhaupt nicht ausgeführt werden, weil z.B. der vorherige Aufruf einer Finalisierungsmethode blockiert.[1]

---

[1] Auf den folgenden MSDN-Webseite werden Details zum Finalisierungsverhalten der CLR dokumentiert: http://msdn.microsoft.com/en-us/library/system.object.finalize.aspx

Außerdem ist es oft wichtig, (Ein- oder Ausgabe-)Ströme durch ein explizites **Close()** so früh wie möglich zu schließen, um (exklusive) Zugriffe durch andere Interessenten zu ermöglichen.

Im Beispiel `FSDemo` von Abschnitt 14.1.2 ist das Schließen der Datei erforderlich, um sie anschließend löschen zu können.

Trotz der von Richter formulierten Warnung, scheint es in der Regel sinnvoll, Datenstromobjekte explizit zu schließen.

14.1.4.3 using-Anweisung

Mit der **using**-Anweisung (nicht zu verwechseln mit der **using**-*Direktive* zum Importieren eines Namensraums) kann man einen Block definieren und Objekte erzeugen, die nur innerhalb des **using**-Blocks gültig (referenziert) sind. Beim Verlassen des Blocks werden automatisch per **Dispose()** - Aufruf alle verwendeten Ressourcen (Dateien, Netzwerkverbindungen) freigegeben, z.B.:

```
using System;
using System.IO;

class FSDemoUsing {
  static void Main() {
    String name = "demo.bin";
    byte[] arr = {0,1,2,3,4,5,6,7};
    using (FileStream fs = new FileStream(name, FileMode.CreateNew)) {
      fs.Write(arr, 0, arr.Length);
      fs.Position = 0;
      fs.Read(arr, 0, arr.Length);
      foreach (byte b in arr)
        Console.WriteLine(b);
    }
    File.Delete(name);
  }
}
```

Per **using**-Anweisung wird das Schließen eines Stroms elegant gelöst, wobei der Compiler im Hintergrund eine **try** - Anweisung mit **finally** - Block erstellt. Das letzte Beispielprogramm ist also annähernd äquivalent zu:

```
using System;
using System.IO;

class FSDemoUsing {
   static void Main() {
      String name = "demo.bin";
      byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
      FileStream fs = null;
      try {
         fs = new FileStream(name, FileMode.CreateNew);
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      } finally {
         if (fs != null)
            fs.Close();
      }
      File.Delete(name);
   }
}
```

Ausnahmen, mit denen bei Dateisystemzugriffen jederzeit zu rechnen ist, werden von der **using**-Anweisung nicht behandelt und folglich an den Aufrufer weitergereicht.

14.1.5 Ausnahmen behandeln

Weil Methodenaufrufe bei Datenstromobjekten diverse Ausnahmen produzieren können (z.B. **FileNotFoundException**, **UnauthorizedAccessException**, **IOExeption**), sind sie am besten in einem **try** - Block untergebracht. Damit ein **Close()** - Aufruf zum Schließen eines Datenstromobjekts auf jeden Fall ausgeführt wird, gehört er in den **finally**-Block der **try**-Anweisung. In der folgenden Variante des Einstiegsbeispiels ist das Abfangen der Ausnahmen immerhin angedeutet:

```
using System;
using System.IO;

class FSDemoCatch {
   static void Main() {
      String name = "demo.bin";
      byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
      FileStream fs = null;
      try {
         fs = new FileStream(name, FileMode.CreateNew);
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      } catch (Exception e) {
         Console.WriteLine("Der Schreib-, Lesevorgang ist gescheitert:\n" + e.Message);
      } finally {
         if (fs != null)
            fs.Close();
      }
      try {
         File.Delete(name);
      } catch (Exception e) {
         Console.WriteLine("\nDie Datei kann nicht gelöscht werden:\n" + e.Message);
      }
   }
}
```

Mit Hilfe der **using**-Anweisung (siehe Abschnitt 14.1.4) lässt sich das Verhalten des Programms verbessern und dabei Schreibaufwand sparen:

```
static void Main() {
   String name = "demo.bin";
   byte[] arr = { 0, 1, 2, 3, 4, 5, 6, 7 };
   try {
      using (FileStream fs = new FileStream(name, FileMode.CreateNew)) {
         fs.Write(arr, 0, arr.Length);
         fs.Position = 0;
         fs.Read(arr, 0, arr.Length);
         foreach (byte b in arr)
            Console.WriteLine(b);
      }
      File.Delete(name);
   } catch (Exception e) {
      Console.WriteLine("Die FileStream-Demo ist gescheitert:\n" + e.Message);
   }
}
```

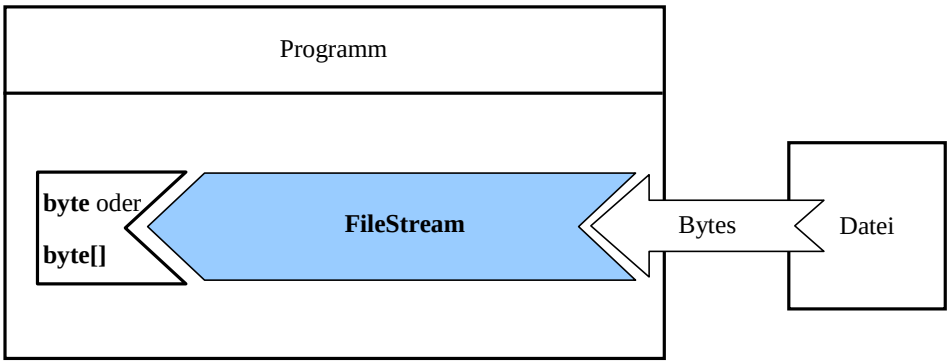
Nun steht der implizite **Dispose()** - Aufruf im **finally**-Block der vom Compiler aus der **using**-Anweisung erstellten **try**-Anweisung. Folglich sind zwei **try**-Anweisungen verschachtelt, so dass die **Delete()** - Methode nach einem Fehler bei den vorangegangenen Operationen (z.B. wegen einer

bereits vorhandenen Datei **demo.bin**) nicht aufgerufen wird. Von der vorherigen Programmversion wird eine vorgefundene Datei gelöscht.

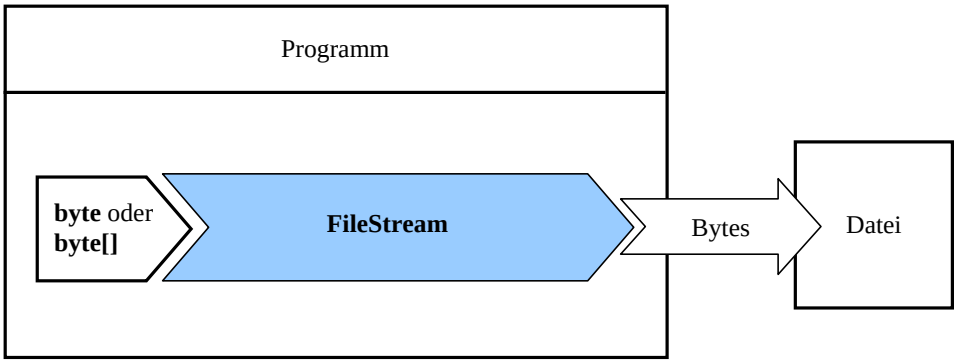
In den weiteren Verlauf von Kapitel 14 werden wir der Einfachheit halber meist auf eine Ausnahmebehandlung verzichten.

14.1.6 FileStream

Mit einem **FileStream**-Objekt können Bytes aus einer Datei gelesen



oder dorthin geschrieben werden:



Ein **FileStream**-Objekt beherrscht prinzipiell *beide* Transportrichtungen, kann aber auch auf unidirektionalen Betrieb eingestellt werden.

Im folgenden **FileStream**-Konstruktoraufruf wird eine per Pfadname identifizierte Datei nötigenfalls erstellt und zum Lesen und/oder Schreiben geöffnet:

```
FileStream fs = new FileStream("demo.bin", FileMode.Create);
```

Falls die Datei bereits existiert, wird sie überschrieben.

Für hinreichende Flexibilität bei der Ansprache und Behandlung von Dateien sorgen die Werte des im Beispiel benutzten Konstruktorparameters **FileMode** (siehe Abschnitt 14.1.6.1) sowie insgesamt 11 (nicht als veraltet deklarierte) Überladungen des Konstruktors.

FileStream-Objekte verwenden einen Puffer, um die Anzahl der Zugriffe auf die angeschlossene Datei möglichst gering zu halten. Beim Schließen einer Datei (durch den Garbage Collector oder einen expliziten **Close()** - Aufruf) werden gepufferte Schreibvorgänge automatisch ausgeführt. Einige Überladungen des Konstruktors bieten einen Parameter, um die voreingestellte Puffergröße von 4096 Bytes zu ändern.

14.1.6.1 Öffnungsmodus

Beim Erstellen eines **FileStream**-Objekts kann man den Öffnungsmodus über einen Wert des Enumerationstyps **FileMode** wählen:

Modus	Beschreibung
Append	Die Datei wird geöffnet und auf das Ende positioniert oder neu erzeugt.
Create	Ist die Datei noch *nicht* vorhanden, wird sie angelegt (wie bei **CreateNew**), anderenfalls wird sie überschrieben (wie bei **Truncate**).
CreateNew	Es wird eine neue Datei angelegt, oder eine **IOEexeption** geworfen, falls eine Datei mit dem gewünschten Namen bereits existiert.
Open	Es wird eine vorhandene Datei geöffnet, oder eine **IOEexeption** geworfen, falls keine Datei mit dem angegebenen Namen existiert.
OpenOrCreate	Es wird eine neue Datei erzeugt oder eine vorhandene geöffnet, jedoch im Unterschied zu **Create** *nicht* automatisch überschrieben.
Truncate	Es wird eine vorhandene Datei geöffnet und entleert, oder eine **IOEexeption** geworfen, falls keine Datei mit dem gewünschten Namen existiert.

Beim Öffnungsmodus **Append** befindet sich der Dateizeiger am Ende der Datei (hinter dem letzten Byte), und es ist kein lesender Zugriff möglich. Bei den anderen Öffnungsmodi befindet sich der Dateizeiger vor dem ersten Byte.

Mit einem Öffnungsmodus wählt man ein Bündel von Einstellungen:

- Ist eine vorhandene Datei Voraussetzung oder Grund für einen Ausnahmefehler?
- Initiale Position des Dateizeigers
- Soll der aktuelle Inhalt einer vorhandenen Datei gelöscht oder beibehalten werden?
- Welche Zugriffsmöglichkeiten sind erlaubt (siehe Abschnitt 14.1.6.2)?

14.1.6.2 Zugriffsmöglichkeiten für das erstellte FileStream-Objekt

Bei einigen Überladungen des **FileStream**-Konstruktors lassen sich über einen Parameter vom Enumerationstyp **FileAccess** die Zugriffsmöglichkeiten für den eigenen Prozess vereinbaren, wobei auf Verträglichkeit mit dem Eröffnungsmodus zu achten ist. Es sind folgende Alternativen verfügbar:

- **FileAccess.Read**
- **FileAccess.Write**
- **FileAccess.ReadWrite**

In folgendem Beispiel wird eine Datei zum Lesen geöffnet:

```
FileStream fs = new FileStream("demo.bin", FileMode.Open,
                               FileAccess.Read);
```

Die in Abschnitt 14.1.6.1 beschriebenen Öffnungsmodi für Dateien sind nur eingeschränkt mit den **FileAccess**-Werten kombinierbar:

Öffnungsmodus	Erlaubte FileAccess-Werte	Voreinstellung
Append	Write	Write
Create	Write, ReadWrite	ReadWrite
CreateNew	Write, ReadWrite	ReadWrite
Open	Read, Write, ReadWrite	ReadWrite
OpenOrCreate	Read, Write, ReadWrite	ReadWrite
Truncate	Write, ReadWrite	ReadWrite

14.1.6.3 Zugriffsmöglichkeiten für andere Interessenten

Für eine per **FileStream**-Konstruktor geöffnete Datei dürfen andere Interessenten (im eigenen oder in einem fremden Prozess) per Voreinstellung simultan *lesend* zugreifen. Bei einigen **FileStream**-Konstruktorüberladungen kann man die Freigabe für den gemeinsamen Zugriff über einen Parameter vom Enumerationstyp **FileShare** regeln. Dabei sind im Wesentlichen die folgenden Alternativen verfügbar:

- **FileShare.None**
- **FileShare.Read**
- **FileShare.Write**
- **FileShare.ReadWrite**

In folgendem Beispiel wird die gemeinsame Nutzung komplett verweigert:

```
FileStream fs = new FileStream("demo.bin", FileMode.Create,
                                           FileAccess.ReadWrite,
                                           FileShare.None);
```

Beim Simultanzugriff auf eine Datei über zwei verschiedene **FileStream**-Objekte, ist zu beachten, dass *beide* Objekte über einen geeigneten **FileShare**-Parameterwert im Konstruktor den Partner berechtigen müssen (Eller & Kofler, 2005, S. 369). Diese Konstellation

```
FileStream fs1 = new FileStream(name, FileMode.Create, FileAccess.ReadWrite);
   .  .  .
FileStream fs2 = new FileStream(name, FileMode.Open, FileAccess.Read);
```

scheitert mit einem Ausnahmefehler, weil `fs2` den voreingestellten **FileShare**-Wert **Read** verwendet, also das von `fs1` benötigte Schreibrecht nicht einräumt. So klappt es:

```
FileStream fs2 = new FileStream(name,
                                FileMode.Open, FileAccess.Read, FileShare.ReadWrite);
```

14.2 Verarbeitung von Daten mit höherem Typ

Bisher haben wir uns auf das Schreiben und Lesen von *Bytes* beschränkt. Im Abschnitt 14.2 lernen Sie Verfahren kennen, um Daten mit einem beliebigen (aus mehreren Bytes bestehenden) Typ (z.B. **int**, **double**, **String**, beliebige Klasse oder Struktur) in Dateien zu schreiben oder von dort zu lesen.

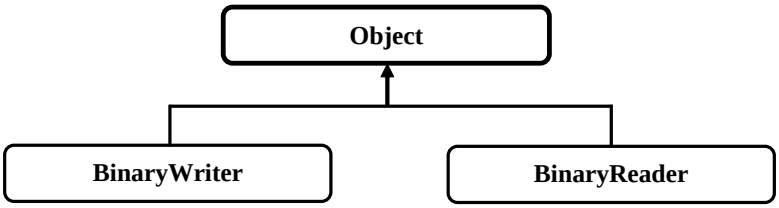
Bei den Dateien auf einem Rechner kann man unterscheiden:

- **Binärdateien**
  In einer Binärdatei werden Daten im Wesentlichen genauso dargestellt wie im Arbeitsspeicher eines Rechners, sodass Programme wenig Mühe dabei haben, Daten beliebigen Typs in eine Binärdatei zu schreiben oder aus einer Binärdatei mit bekanntem Aufbau zu lesen. Für den menschlichen Konsum ist eine Binärdatei allerdings nicht geeignet. Öffnet man sie mit einem Texteditor, ist nur eine wirre Folge von (Sonder-)zeichen zu sehen. In Abschnitt 14.2.1 werden die zum Schreiben bzw. Lesen binärer Daten konstruierten Klassen **BinaryWriter** bzw. **BinaryReader** vorgestellt.
- **Textdateien**
  Diese Dateien können von Menschen mit Hilfe eines Texteditors gelesen und/oder bearbeitet werden, sofern der Texteditor die beim Erstellen der Datei verwendete Zeichenkodierung versteht. Per Programm lassen sich numerische Daten und Zeichenfolgen leicht in eine Textdatei schreiben, doch ist das Lesen *numerischer* Daten aus einer Textdatei mit erhöhtem Aufwand verbunden. In Abschnitt 14.2.2 werden die zum Schreiben bzw. Lesen von Zeichenfolgen konstruierten Klassen **StreamWriter** bzw. **StreamReader** vorgestellt.

In Abschnitt 14.2.3 beschäftigen wir uns mit dem Schreiben und Lesen von kompletten Objekten (oder auch Strukturinstanzen), wobei eine Binärdatei zum Einsatz kommt.

14.2.1 Schreiben und Lesen im Binärformat

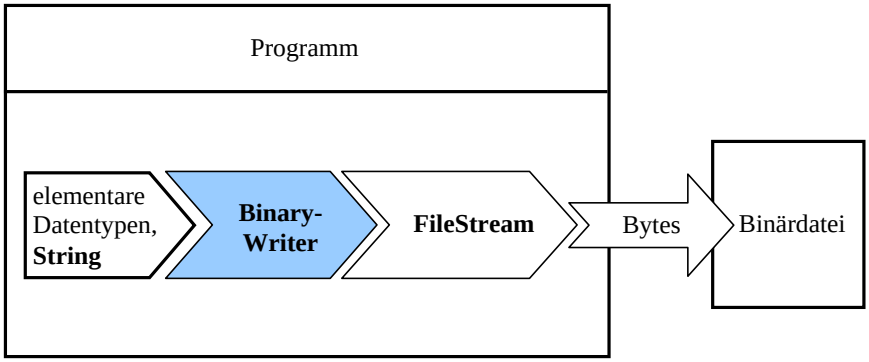
Um Werte von einem beliebigen elementaren Datentyp (z.B. **int**, **double**) sowie Zeichenfolgen in einen binär organisierten Strom (z.B. in eine Binärdatei) zu schreiben bzw. aus einem Binärstrom mit bekanntem Aufbau zu lesen, verwendet man ein Objekt der Klasse **BinaryWriter** bzw. **BinaryReader**.



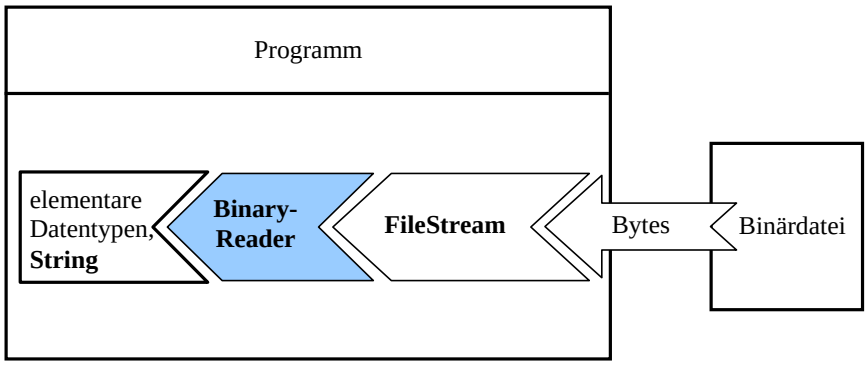
Die beiden Klassen stammen *nicht* von der in Abschnitt 14.1 behandelten Klasse **Stream** ab, verwenden aber für die Verbindung mit einer Datenquelle oder -senke ein **Stream**-Objekt, das im Konstruktor anzugeben ist, z.B.:

public BinaryWriter(Stream *ausgabestrom***)**

Im Unterschied zu den „bidirektionalen" **Stream**-Klassen sind für das Schreiben bzw. Lesen von elementaren Datenwerten „gerichtete" Klassen zuständig. Schreibt man per **BinaryWriter** in eine Datei, entsteht folgende Verarbeitungskette:



Beim *Lesen* aus einer Binärdatei reisen die Daten in umgekehrter Richtung:



Das folgende Beispielprogramm schreibt einen **int**- und einen **double**-Wert sowie eine Zeichenfolge per **BinaryWriter** über einen **FileStream** in eine Datei. Anschließend werden die Daten über eine **BinaryReader** - **FileStream** - Konstruktion eingelesen:

```
using System;
using System.IO;

class BinWrtRd {
   static void Main() {
      String name = "demo.bin";
      FileStream fso = new FileStream(name, FileMode.Create);
      BinaryWriter bw = new BinaryWriter(fso);
      bw.Write(4711);
      bw.Write(3.1415926);
      bw.Write("Nicht übel");
      bw.Close();

      FileStream fsi = new FileStream(name, FileMode.Open, FileAccess.Read);
      BinaryReader br = new BinaryReader(fsi);
      Console.WriteLine(br.ReadInt32() + "\n" +
                        br.ReadDouble() + "\n" +
                        br.ReadString());
      br.Close();
   }
}
```

Um das Schreiben und das Lesen unabhängig voneinander vorzuführen, wird im Beispiel jeweils ein eigenes **FileStream**-Objekt mit passenden **FileMode**- bzw. **FileAccess**- Konstruktorparameterwerten erstellt. Es wäre möglich, mit *einem* **FileStream**-Objekt zu arbeiten und dessen **Position**-Eigenschaft nach dem Schreiben wieder auf 0 zu setzen (siehe Beispiel in Abschnitt 14.1.1).

Die von beiden **FileStream**-Objekten verwendete Datei wird zunächst mit dem **FileMode.Create** geöffnet. Nach den Schreibzugriffen per **Write()** - Methode wird die Datei per **Close()** - Aufruf geschlossen, damit das anschließende Öffnen mit dem **FileMode.Open** gelingt. Der **Close()** - Aufruf kann sich an das **FileStream**- oder an das **BinaryWriter**-Objekt richten, wobei er im letztgenannten Fall durchgereicht wird.

Durch den **Close()** - Aufruf wird der Schreibpuffer des **FileStream**-Objekts `fso` geleert, so dass die geschriebenen Daten komplett in der Datei ankommen. Ein **BinaryWriter** verwaltet übrigens *keinen* eigenen Puffer, sondern reicht Schreibaufträge stets direkt an das angeschlossene **Stream**-Objekt weiter.[1] Erhält ein **BinaryWriter**-Objekt einen **Flush()** - Aufruf zum Entleeren des Puffers, reicht es ihn an das verbundene **Stream**-Objekt weiter.

Während die Klasse **BinaryWriter** für alle unterstützten Datentypen eine Überladung der Methode **Write()** besitzt, sind in der Klasse **BinaryReader** typspezifisch benannte Lesemethoden vorhanden (z.B. **ReadInt32()**, **ReadDouble()**).

Die Ausgabe des Programms:

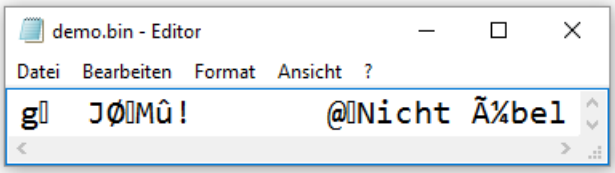
```
4711
3,1415926
Nicht übel
```

---

[1] Z.B werden bei einem Aufruf der **Write()** - Methode mit **int**-Parameter die vier Bytes sofort an den Ausgabestrom übergeben (Source Code bezogen von Microsofts Webseite http://referencesource.microsoft.com/):

```
public virtual void Write(int value) {
   _buffer[0] = (byte) value;
   _buffer[1] = (byte)(value >> 8);
   _buffer[2] = (byte)(value >> 16);
   _buffer[3] = (byte)(value >> 24);
   OutStream.Write(_buffer, 0, 4);
}
```

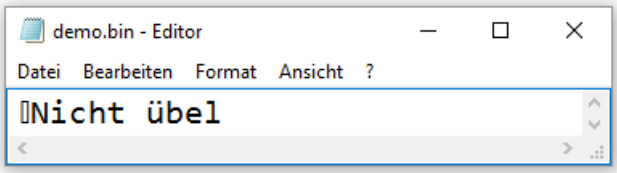
Es macht wenig Sinn, die vom Beispielprogramm erzeugte Binärdatei mit einem Texteditor zu öffnen:



Im Beispiel wird die vom **BinaryWriter** geschriebene Zeichenfolge allerdings vom Windows-Texteditor **notepad.exe** fast korrekt dargestellt. Bei der Ausgabe von **String**-Variablen ist die verwendete *Ausgabekodierung* der Zeichen relevant, die im Arbeitsspeicher eines Rechners bekanntlich in Unicode-Kodierung vorliegen. Die Klasse **BinaryWriter** verwendet per Voreinstellung dasselbe **UTF8Encoding** wie die später vorzustellenden **TextWriter**-Klassen. Offenbar ist die vom Windows-Texteditor unterstellte ANSI-Kodierung bei den am häufigsten verwendeten Zeichen mit dem **UTF8Encoding** kompatibel. [1] Über einen **BinaryWriter**-Konstruktor mit entsprechendem Parameter stehen auch andere Kodierungen zur Verfügung:

public BinaryWriter(Stream *ausgabestrom***, Encoding** *kodierung***)**

Schreibt man per **BinaryWriter** an Stelle der gemischten Ausgaben ausschließlich Text, kann der Windows-Texteditor beim Öffnen der Ergebnisdatei die zugrunde liegende UTF-8 - Kodierung übrigens erkennen, und das Ergebnis ist perfekt bis auf einen kleinen Defekt am Anfang, der gleich erklärt wird:



Trotz der gemeinsamen Kodierungsvoreinstellung gibt es zwischen der Klasse **BinaryWriter** und den **TextWriter**-Klassen einen kleinen Unterschied bei der Textausgabe. Der **BinaryWriter** schreibt zur Unterstützung des **BinaryReader**s vor jede Zeichenfolge ihre Länge (Anzahl der Bytes) und verwendet dabei den Datentyp **uint** mit einer speziellen 7-Bit - Kodierung:

- Von den 32 **uint**-Bits wird nur der signifikante Anteil als Byte-Sequenz ausgegeben. Führende Nullen werden also weggelassen.
- Ein Ausgabebyte enthält 7 **uint**-Bits (mit der niedrigsten Wertigkeit beginnend). Im achten Bit signalisiert eine Eins, dass noch ein weiteres Paket mit (7+1) Bits folgt.

Bei der Zeichenfolge „Nicht übel“ mit 11 Bytes Länge (neun Single-Byte-Zeichen und ein Double-Byte-Zeichen bei UTF-8 - Kodierung, siehe Abschnitt 14.2.2) schreibt der **BinaryWriter** als Längenpräfix *ein* Byte. Bei einer Zeichenfolge mit 258 Bytes Länge resultiert ein Längenpräfix mit zwei Bytes:

Länge der Zeichenfolge in Bytes	**uint**-Bits	Längenpräfix
11	0…0 00000000 00001011	00001011
258	0…0..00000001 00000010	10000010 00000010

[1] Im Bereich von 0 bis 127 enthalten der Unicode und der ANSI-Code dieselben Zeichen wie der ASCII-Code (*American Standard Code for Information Interchange*) aus der Steinzeit der Datenverarbeitung

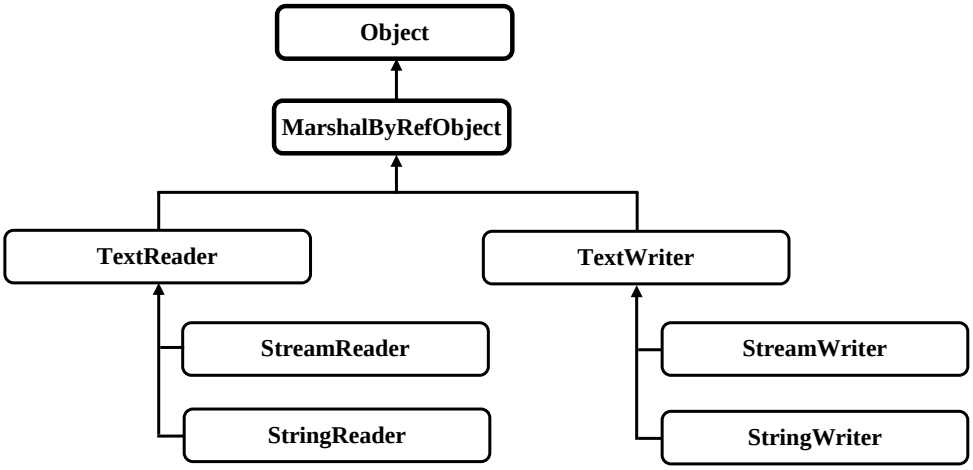
Es besteht *kein* Risiko, wenn ein Gespann aus einem **BinaryWriter**- und einem **FileStream**-Objekt dem Garbage Collector anheimfallen, obwohl beim automatischen Finalisieren (ohne garantierte Reihenfolge!) zuerst das **FileStream**-Objekt beseitigt werden könnte:

- **BinaryWriter**-Objekte besitzen keinen lokaler Puffer, der beim Abräumen geleert werden müsste.
- In der Klasse **BinaryWriter** ist keine Finalisierungsmethode vorhanden, so dass beim Abräumen kein Zugriff auf das zugrunde liegende (und eventuell nicht mehr existente) **Stream**-Objekt stattfinden kann.

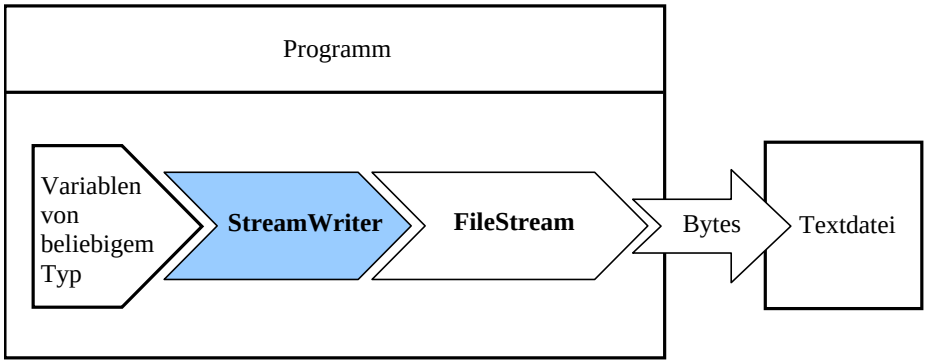
Objekte der anschließend behandelten Klasse **StreamWriter** müssen aufgrund ihres lokalen Puffers jedoch unbedingt *vor* dem zugrunde liegenden **Stream**-Objekt geschlossen werden, was nur durch einen **Close()** - Aufruf (oder einen äquivalenten **Dispose()** - Aufruf) an das **StreamWriter**-Objekt sichergestellt ist (eventuell per **using**-Block automatisiert, vgl. Abschnitt 14.1.4).

14.2.2 Schreiben und Lesen im Textformat

Mit einem **TextWriter**-Objekt kann man die Zeichenfolgenrepräsentation von Variablen beliebigen Typs ausgeben. Das Gegenstück **TextReader** liefert stets Zeichen ab, so dass bei der Versorgung von numerischen Variablen aus textuellen Eingabedaten etwas Eigeninitiative gefragt ist (siehe Übungsaufgabe in Abschnitt 14.4). Beide Klassen sind abstrakt, doch bietet die FCL auch konkrete Ableitungen für **Stream**- bzw. **String**-Objekte als Senken bzw. Quellen:

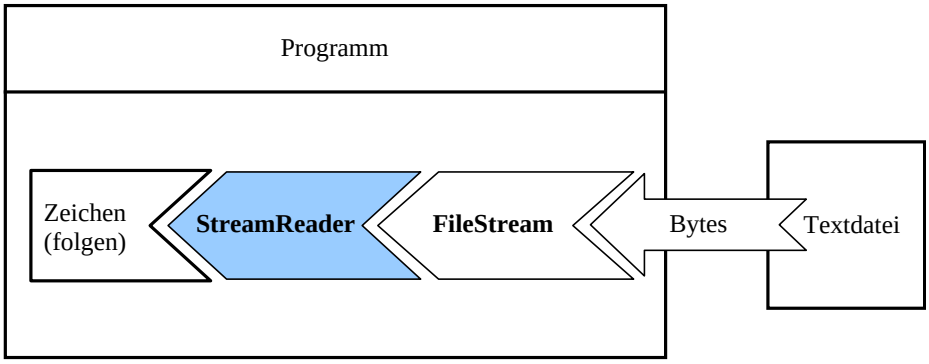


Um in eine Textdatei zu schreiben bzw. von dort zu lesen, verwendet man Objekte der Klassen **StreamWriter** bzw. **StreamReader**, die jeweils über eine Instanzvariable mit einem **FileStream**-Objekt verbunden sind (analog zu den Klassen **BinaryWriter** und **BinaryReader**).[1] Beim Schreiben haben wir also folgende Situation:



[1] Die Bezeichnungen **StreamWriter** und **StreamReader** sind nicht ganz glücklich, weil auch ein **BinaryWriter** in ein **Stream**-Objekt schreibt und ein **BinaryReader** aus einem **Stream**-Objekt liest.

Beim Lesen aus einer Textdatei reisen die Daten in umgekehrter Richtung:



Man kann den Basisstrom für einen **StreamWriter** oder -**Reader** auch *implizit* erzeugen lassen, wenn die voreingestellten Kreationsparameter akzeptabel sind, z.B.:

```
StreamWriter sw = new StreamWriter("demo.txt");
```

Hier wird implizit ein **FileStream**-Objekt mit der voreingestellten Puffergröße 4096 erzeugt, das auf eine Datei mit dem Eröffnungsmodus **FileMode.Create** zugreift.

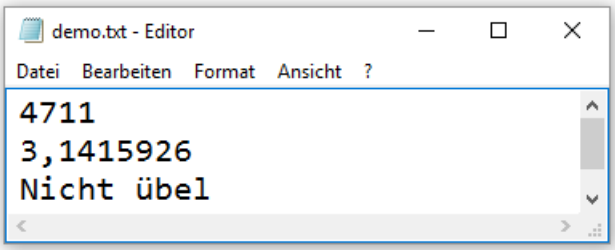
Das folgende Beispielprogramm schreibt einen **int**- und einen **double**-Wert sowie eine Zeichenfolge per **StreamWriter** über ein implizit erzeugtes **FileStream**-Objekt in eine Datei. Anschließend werden die Daten über einen **StreamReader** eingelesen, der sich auf ein explizit erzeugtes **FileStream**-Objekt stützt:

```
using System;
using System.IO;

class StreamWrtRd {
   static void Main() {
      String name = "demo.txt";
      StreamWriter sw = new StreamWriter(name);
      sw.WriteLine(4711);
      sw.WriteLine(3.1415926);
      sw.WriteLine("Nicht übel");
      sw.Close();

      StreamReader sr = new StreamReader(
                           new FileStream(name, FileMode.Open, FileAccess.Read));
      Console.WriteLine("Inhalt der Datei {0}:\n", ((FileStream)sr.BaseStream).Name);
      for (int i = 0; sr.Peek() >= 0; i++ ) {
         Console.WriteLine("{0}:\t{1}", i, sr.ReadLine());
      }
      sr.Close();
   }
}
```

Das Beispielprogramm liest so lange die nächste Dateizeile mit der **TextReader**-Methode **ReadLine()**, bis die **TextReader**-Methode **Peek()** durch die Rückgabe -1 das Dateiende signalisiert. Wir erhalten die folgende Ausgabe:

```
Inhalt der Datei U:\Eigene Dateien\C#\EA\StreamWrtRd\bin\Debug\demo.txt:

1:      4711
2:      3,1415926
3:      Nicht übel
```

Auch die erzeugte Textdatei ist ansehnlich:



Bei den **TextWriter**-Methoden **Write()** und **WriteLine()** treffen wir im Wesentlichen auf dieselben Signaturen wie bei den gleichnamigen **Console**-Methoden, die Ihnen aus zahlreichen Beispielen vertraut sind.

Im Unterschied zu einem **BinaryWriter** (siehe Abschnitt 14.2.1) besitzt ein **StreamWriter** einen lokalen Puffer (Datentyp: **char[]**, voreingestellte Größe: 1024). Daher muss ein **StreamWriter** unbedingt nach Gebrauch per **Close()** (oder **Dispose()**) geschlossen werden. Das zugrunde liegende **Stream**-Objekt wird dabei automatisch ebenfalls geschlossen. Es wäre riskant, das Schließen dem Garbage Collector zu überlassen, für den keine Arbeitsreihenfolge garantiert ist. Wenn er das **Stream**-Objekt *vor* dem **StreamWriter**-Objekt schließt, kann letzteres seinen Puffer nicht mehr ausgeben.

Über die boolesche **StreamWriter**-Eigenschaft **AutoFlush** (Voreinstellung: **false**) wird festgelegt, ob die per **Write()** oder **WriteLine()** geschriebenen Zeichen sofort in den Ausgabestrom wandern (bei bestimmten Ausgabegeräten sinnvoll) oder zwischengepuffert werden sollen (höhere Performanz).[1]

Arbeitet ein **StreamWriter** mit einem **FileStream** zusammen, findet eine *Doppelpufferung* statt:

- Ein **StreamWriter**-Objekt enthält als Puffer einen **char**-Array (voreingestellte Größe: 1024).
- Ein **FileStream**-Objekt enthält als Puffer einen **byte**-Array (voreingestellte Größe: 4096).

Beim **Flush()** - Aufruf an ein **StreamWriter**-Objekt …

- wird zunächst der **StreamWriter**-Puffer in den Ausgabestrom geschrieben
- und dann ein **Flush**() - Aufruf an das **Stream**-Objekt gerichtet.

Hinweise zu einigen **TextReader**-Methoden:

---

[1] Die folgende Implementierung der **Write()** - Methode mit **char**-Parameter aus der Klasse **StreamWriter** zeigt, wie sich die öffentliche **AutoFlush**-Eigenschaft über das private **autoFlush**-Feld auf das Pufferungsverhalten auswirkt:

```
public override void Write(char value) {
  if (charPos == charLen) Flush(false, false);
  charBuffer[charPos++] = value;
  charPos++;
  if (autoFlush) Flush(true, false);
}
```

Der Quellcode stammt aus der Datei **streamwriter.cs**, die über Microsofts .NET - Source Code - Webseite (http://referencesource.microsoft.com/) inspiziert werden kann.

- **public String ReadLine()**
  Diese Methode liest eine Zeile, liefert das Ergebnis als **String**-Objekt ab und verschiebt die Position des Eingabestroms entsprechend. Unter einer *Zeile* ist eine Folge von Zeichen zu verstehen, auf die eine von den folgenden Terminierungen folgt:
  - das Steuerzeichen Carriage Return (0x000D),
  - das Steuerzeichen Line Feed (0x000A),
  - eine Sequenz aus den Sonderzeichen Carriage Return und Line Feed,
  - die in der statischen **Environment.NewLine** hinterlegte Zeichenfolge,
  - das Ende des Stroms

  Im abgelieferten **String**-Objekt ist die Zeilenterminierung *nicht* enthalten. Enthält der Eingabestrom keine Zeichen mehr, liefert **ReadLine()** als Rückgabe den Wert **null**.
- **public int Read()**
  Diese Methode liefert als Rückgabewert die Unicode-Nummer des nächsten Zeichens oder aber den Wert -1, wenn der Strom kein Zeichen mehr enthält, und verschiebt die Position des Eingabestroms entsprechend.
- **public int Peek()**
  Diese Methode liefert wie **Read()** die Unicode-Nummer des nächsten Zeichens oder aber den Wert -1, wenn der Strom kein Zeichen mehr enthält. Die Position des Stroms bleibt dabei aber unverändert.

Per Voreinstellung schreiben bzw. lesen die **StreamWriter** bzw. -**Reader** Unicode-Zeichen unter Verwendung der Platz sparenden **UTF-8** - Kodierung. Bei diesem Schema werden die Unicode-Zeichen durch eine variable Anzahl von Bytes kodiert. So können alle Unicode-Zeichen (2^{16} an der Zahl) ausgegeben werden, ohne eine Speicherplatzverschwendung durch führende Null-Bytes bei den sehr oft auftretenden Zeichen mit Unicode-Nummern ≤ 127 in Kauf nehmen zu müssen: [1]

Unicode-Zeichen		Anzahl Bytes
von	**bis**	
\u0000	\u0000	2
\u0001	\u007F	1
\u0080	\u07FF	2
\u0800	\uFFFF	3

Bei einigen Überladungen des **StreamWriter**-Konstruktors lassen sich auch alternative Kodierungen einstellen, z.B.:

```
FileStream fs = new FileStream("unicode.txt", FileMode.Create);
StreamWriter swUnicode = new StreamWriter(fs, Encoding.Unicode);
```

Die statische Eigenschaft **Unicode** der Klasse **Encoding** im Namensraum **System.Text** zeigt auf ein Objekt der Klasse **UnicodeEncoding**, das die Kodierung übernimmt. Es verzichtet auf Platzsparmaßnahmen und verwendet für *jedes* Zeichen 2 Bytes.

Auch bei Objekten der Klasse **StreamReader** lässt sich die voreingestellte UTF-8 - Kodierung per Konstruktorparameter ersetzen, was z.B. beim Lesen der häufig anzutreffenden Textdateien mit **ANSI-Kodierung** erforderlich ist. Im folgenden Programm

[1] Im Bereich von 0 bis 127 befinden sich im Unicode dieselben Zeichen wie im ASCII-Code (*American Standard Code for Information Interchange*) aus der Steinzeit der Datenverarbeitung

```
using System;
using System.IO;
using System.Text;

class AnsiTextLesen {
   static String name = "AnsiText.txt";
   static void Main() {
      FileStream fs = new FileStream(name, FileMode.Open, FileAccess.Read);
      StreamReader sr = new StreamReader(fs);
      Console.WriteLine("Mit UTF-8 - Kodierung gelesen:");
      while (sr.Peek() >= 0)
         Console.WriteLine(sr.ReadLine());
      fs.Position = 0;
      sr = new StreamReader(fs, Encoding.Default);
      Console.WriteLine("\nMit ANSI - Kodierung gelesen:");
      while (sr.Peek() >= 0)
         Console.WriteLine(sr.ReadLine());
      fs.Close();
   }
}
```

wird beim Lesen einer Textdatei zuerst die UTF-8 - und dann die ANSI-Kodierung unterstellt. Bei einer Datei mit ANSI-Kodierung und folgendem Inhalt

```
ANSI-kodierte Umlaute: üöä
```

resultiert die Ausgabe:

```
Mit UTF-8 - Kodierung gelesen:
ANSI-kodierte Umlaute: ???

Mit ANSI - Kodierung gelesen:
ANSI-kodierte Umlaute: üöä
```

Über die statische **Encoding**-Eigenschaft **Default** erhält man eine zur aktuellen ANSI-Codepage von Windows passende Kodierung.

14.2.3 Binäres Serialisieren von Instanzen

Wer objektorientiert programmiert, möchte natürlich auch objektorientiert speichern und laden. Erfreulicherweise können in C# Objekte und Strukturen tatsächlich in der Regel genau so einfach wie Werte mit einem elementaren Typ in einen Datenstrom geschrieben bzw. von dort gelesen werden. Die keinesfalls triviale Übersetzung einer Instanz mit allen Feldern und den eventuell von Feldern referenzierten Objekten in einen Bytestrom bezeichnet man als *Instanzserialisierung*. Beim Einlesen einer Instanz werden alle Felder wiederhergestellt und die Referenzen zwischen Objekten in den Ausgangszustand gebracht. Die FCL-Dokumentation spricht von (De)serialisieren eines *Instanzdiagramms*.

Das Serialisieren der Instanzen eines Typs muss explizit bei der Definition über das Typattribut **Serializable** erlaubt werden. Dies darf nicht unbedacht erfolgen, z.B. weil die Serialisierbarkeit aller Datentypen von Feldern erforderlich ist. Aus nahe liegenden Gründen haben die FCL-Designer das Attribut **Serializable** als *nicht* vererbbar definiert, so dass es nicht auf abgeleitete Klassen übertragen wird.

Bei Bedarf können einzelne Felder über das Attribut **NonSerialized** ausgeschlossen werden. Dies kommt z.B. in Frage, wenn ...

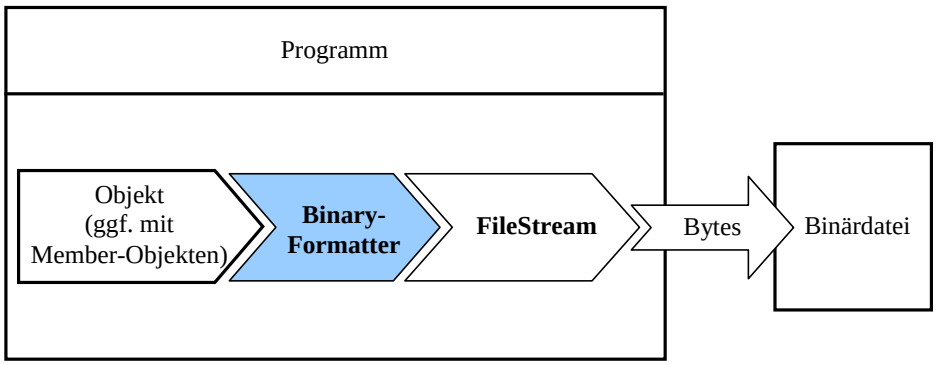
- ein Feld aus Sicherheitsgründen nicht in den Ausgabestrom gelangen soll,
- ein Feld temporäre Daten enthält, so dass ein Speichern überflüssig bzw. sinnlos ist,
- ein Feld einen nicht-serialisierbaren Datentyp hat.
  Wird ein solches Feld nicht von der Serialisierung ausgeschlossen, kommt es ggf. zu einer **SerializationException**.

Über das Implementieren der Schnittstelle **ISerializable** gewinnt der Typdesigner Kontrolle über die (De)serialisierung, muss aber dazu die **ISerializable**-Methode **GetObjectData()** implementieren.

Wir beschränken uns im weiteren Verlauf des Abschnitts auf das (De)serialisieren von Objekten. Die im folgenden Quellcode definierte Klasse `Kunde` ist als serialisierbar deklariert, wobei jedoch für das Feld `stimmung` eine Ausnahme gemacht wird:

```
using System;
using System.Runtime.Serialization;

[Serializable]
public class Kunde {
   int nr;
   string vorname;
   string name;
   int nkaeufe;
   double aussen;
   [NonSerialized]
   int stimmung;

   public Kunde(int nr_, string vorname_, string name_, int stimmung_,
                int nkaeufe_, double aussen_) {
      nr = nr_;
      vorname = vorname_;
      name = name_;
      stimmung = stimmung_;
      nkaeufe = nkaeufe_;
      aussen = aussen_;
   }

   public void prot() {
      Console.WriteLine("Kundennummer: \t" + nr);
      Console.WriteLine("Name: \t\t" + vorname + " " + name);
      Console.WriteLine("Stimmung: \t" + stimmung);
      Console.WriteLine("Anz.Einkäufe: \t" + nkaeufe);
      Console.WriteLine("Aussenstände: \t" + aussen+ "\n");
   }
}
```

Den recht anspruchsvollen Job der (De)Serialisierung übernimmt ein Objekt aus einer Klasse, die das Interface **IFormatter** (aus dem Namensraum **System.Runtime.Serialization**) implementiert. Wir arbeiten anschließend mit der Klasse **BinaryFormatter** (aus dem Namensraum **System.Runtime.Serialization.Formatters.Binary**), die ein kompaktes Binärformat verwendet. Beim Abspeichern in eine Datei resultiert die folgende Verarbeitungskette:



Im folgenden Programm wird ein ...

- Aufzählungsobjekt vom Typ **List**<Kunde>
- mit zwei Kunde-Elementobjekten
- mitsamt den jeweils enthaltenen **String**-Objekten
- aber ohne das Feld stimmung

(de)serialisiert:

```
using System;
using System.Collections.Generic;
using System.IO;
using System.Runtime.Serialization.Formatters.Binary;

class Serialisierung {
   static string name = "demo.bin";
   static void Main() {
      var kunden = new List<Kunde>();
      kunden.Add(new Kunde(1, "Fritz", "Orth", 1, 13, 426.89));
      kunden.Add(new Kunde(2, "Ludwig", "Knüller", 2, 17, 89.10));

      Console.WriteLine("Zu sichern:\n");
      foreach (Kunde k in kunden)
         k.Prot();

      FileStream fs = new FileStream(name, FileMode.Create);
      BinaryFormatter bifo = new BinaryFormatter();
      bifo.Serialize(fs, kunden);

      fs.Position = 0;
      Console.WriteLine("\nRekonstruiert:\n");
      var desKunden = (List<Kunde>) bifo.Deserialize(fs);
      foreach(Kunde k in desKunden)
         k.Prot();
      fs.Close();
   }
}
```

Pro **Serialize()** - Aufruf wird *ein* Wurzelobjekt mitsamt den Werttyp-Feldern sowie den direkt oder indirekt referenzierten Objekten, also ein komplettes Objektdiagramm, geschrieben. Um weitere Objektdiagramme in denselben Datenstrom zu befördern, sind entsprechend viele Aufrufe erforderlich.

Beim Lesen eines Objekts durch die Methode **Deserialize()** wird zunächst die zugehörige Klasse festgestellt und in die Laufzeitumgebung geladen (falls noch nicht vorhanden). Dann wird das Objekt auf dem Heap angelegt, und die Instanzvariablen erhalten die rekonstruierten Werte, wobei *kein* Konstruktor aufgerufen wird. Das ganze wiederholt sich (ggf. auf mehreren Ebenen) für die referenzierten Objekte.

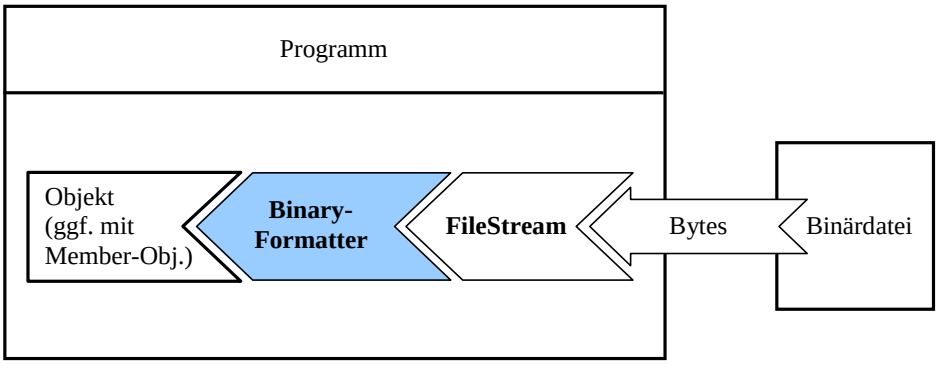
Weil **Deserialize()** den Rückgabewert **Object** hat, ist eine explizite Typumwandlung erforderlich. Ein **Deserialize()** - Aufruf liest das nächste im Datenstrom befindliche Wurzelobjekt samt Anhang (also *ein* Objektdiagramm). Um weitere Wurzelobjekte aus demselben Datenstrom zu lesen, sind entsprechend viele Aufrufe erforderlich.

Das Beispielprogramm produziert folgende Ausgabe:

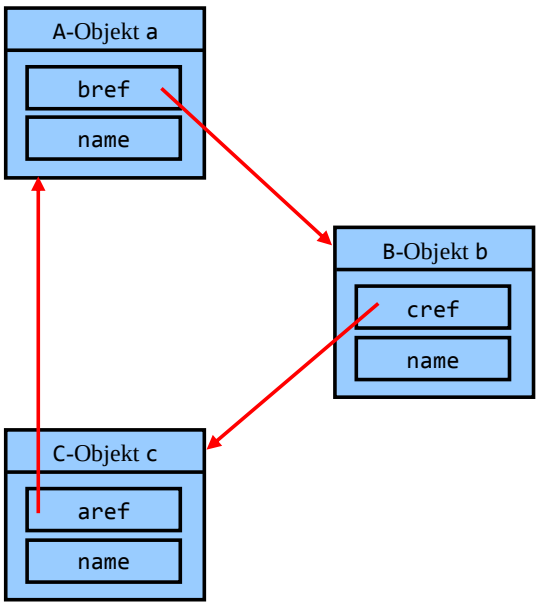
```
Zu sichern:

Kundennummer:   1
Name:           Fritz Orth
Stimmung:       1
Anz.Einkäufe:   13
Aussenstände:   426,89

Kundennummer:   2
Name:           Ludwig Knüller
Stimmung:       2
Anz.Einkäufe:   17
Aussenstände:   89,1


Rekonstruiert:

Kundennummer:   1
Name:           Fritz Orth
Stimmung:       0
Anz.Einkäufe:   13
Aussenstände:   426,89

Kundennummer:   2
Name:           Ludwig Knüller
Stimmung:       0
Anz.Einkäufe:   17
Aussenstände:   89,1
```

Die Instanzvariable `stimmung` der eingelesenen Kunden besitzen den Initialwert 0, während die übrigen Elementvariablen bei der (De)serialisierung ihre Werte behalten.

In der folgenden Abbildung wird die Rekonstruktion der Objekte skizziert:



Auch zirkuläre Referenzen wie in folgender Situation



bringen den **BinaryFormatter** nicht aus dem Tritt. Wenn man Objekte aus den Klassen `A`, `B` und `C` geeignet initialisiert

```
A a = new A(); B b = new B(); C c = new C();
a.bref = b; a.name = "a-Obj";
b.cref = c; b.name = "b-Obj";
c.aref = a; c.name = "c-Obj";
```

und anschließend das über die Referenzvariable `a` ansprechbare Objekt serialisiert,

```
FileStream fs = new FileStream("demo.bin", FileMode.Create);
IFormatter bifo = new BinaryFormatter();
bifo.Serialize(fs, a);
```

dann landen alle *drei* Objekte im Datenstrom. Beim Deserialisieren des Wurzelobjekts

```
fs.Position = 0;
A na = (A) bifo.Deserialize(fs);
Console.WriteLine("Rekonstruiert: " + na.bref.cref.aref.name);
```

werden auch die beiden anderen Objekte rekonstruiert, so dass der **WriteLine()** - Aufruf zu folgender Ausgabe führt:

```
Rekonstruiert: a-Obj
```

Beim Deserialisieren entstehen *neue* Objekte, sodass im Beispiel die Referenzvariable `b` *nicht* auf das restaurierte Objekt der Klasse `B` zeigt, und der Ausdruck

```
b == na.bref
```

den Wert **false** besitzt.

Über die vom **BinaryFormatter** unterstützten *Versions-tolerante Serialisierung* wird es möglich, eine Klasse weiterzuentwickeln, ohne dass konservierte Objekte mit einem älteren Versionsstand inkompatibel werden.[1]

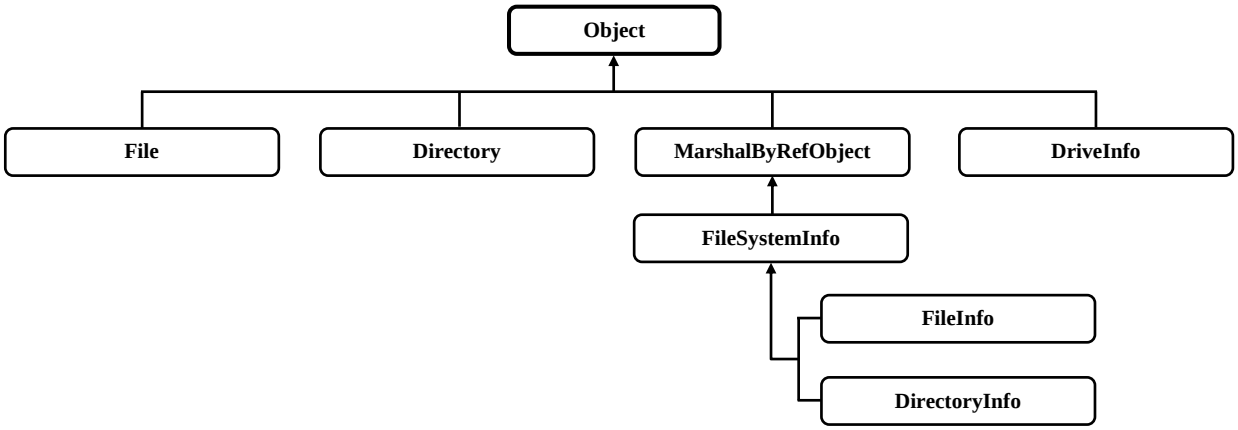
Ist eine XML-basierte Ausgabe gefragt, verwendet man zum Serialisieren die Klasse **XmlSerializer**, die allerdings im Unterschied zur Klasse **BinaryFormatter** keine privaten Felder und keine zirkulären Referenzen unterstützt.

Für die Persistenz von Objekten stellt die Serialisierung eher eine „kleine“ Lösung dar. Im professionellen Umfeld sind andere Lösungen wie das ADO.NET Entity Framework zu bevorzugen

[1] https://msdn.microsoft.com/en-us/library/ms229752(v=vs.80).aspx

14.3 Verwaltung von Dateien und Verzeichnissen

Zur Verwaltung von Dateien bzw. Verzeichnissen enthält die FCL jeweils eine Klasse mit statischen Methoden (**File** bzw. **Directory**) sowie eine Klasse mit Instanzmethoden (**FileInfo** bzw. **DirectoryInfo**):



Man kann Dateien bzw. Verzeichnisse erstellen, kopieren, löschen, umbenennen und verschieben sowie diverse Datei- bzw. Verzeichnisattribute einsehen und verändern. Im Zusammenhang mit dem **TreeView**-Steuerelement werden wir später noch die Klasse **DriveInfo** behandeln, die ein komplettes Laufwerk repräsentiert.

14.3.1 Dateiverwaltung

Im folgenden Beispielprogramm werden einige Methoden und Eigenschaften der Klassen **File** und **FileInfo** demonstriert:

```
using System;
using System.IO;

class Dateiverwaltung {
   static void Main() {
      const string PFAD1 = @"U:\Eigene Dateien\C#\EA\demo.txt";
      const string PFAD2 = @"U:\Eigene Dateien\C#\EA\kopie.txt";
      const string PFAD3 = @"U:\Eigene Dateien\C#\EA\nn.txt";

      StreamWriter sw = File.CreateText(PFAD1);
      sw.WriteLine("File-Demo");
      sw.Close();

      File.Copy(PFAD1, PFAD2, true);

      File.Delete(PFAD2);

      Console.WriteLine("\nExistiert " + PFAD3 + "? " + File.Exists(PFAD3));

      File.Move(PFAD1, PFAD3);

      File.SetCreationTime(PFAD3, new DateTime(2007, 12, 29, 22, 55, 44));
      File.SetLastWriteTime(PFAD3, new DateTime(2005, 12, 29, 22, 55, 44));

      FileInfo fi = new FileInfo(PFAD3);
      Console.WriteLine($"Die Datei {fi.Name} wurde\n  erstellt:\t\t{fi.CreationTime}"+
                        $"\n  zuletzt geändert:\t{fi.LastWriteTime}");

      fi.Delete();
   }
}
```

Wie in Abschnitt 3.3.9.5 über die Syntax von Zeichenkettenliteralen besprochen, wird mit dem Präfix „@“ vor einem Zeichenkettenliteral die Auswertung von Escape-Sequenzen abgeschaltet, so dass die in Windows-Pfadnamen üblichen Rückwärtsschrägstrich nicht mehr durch Verdoppeln von ihrer Sonderfunktion befreit werden müssen.

Anschließend werden wichtige Methoden der Klasse **File** vorgestellt, die allesamt statisch sind und meist noch weitere Überladungen besitzen:

- **public static FileStream Create(String** *pfadname***)**
  Die **File**-Methode **Create()** erzeugt eine Datei und ein zugehöriges **FileStream**-Objekt, so dass die Datei anschließend mit **FileMode.Create** geöffnet ist. Die Anweisung
  ```
  FileStream fs = File.Create(PFAD1)
  ```
  ist äquivalent mit
  ```
  FileStream fs = new FileStream(PFAD1, FileMode.Create)
  ```
- **public static StreamWriter CreateText(String** *pfadname***)**
  Die **File**-Methode **CreateText()** erzeugt eine Datei und ein **StreamWriter**-Objekt (mit UTF8-Kodierung). Es entsteht auch das vermittelnde **FileStream**-Objekt, und die Datei ist anschließend mit **FileMode.Create** geöffnet. Die Anweisung
  ```
  StreamWriter sw = File.CreateText(PFAD1)
  ```
  ist äquivalent mit
  ```
  StreamWriter sw = new StreamWriter(PFAD1)
  ```
- **public static bool Exists(String** *pfadname***)**
  Mit **File.Exists()** überprüft man die Existenz einer Datei.
- **public static void Delete(String** *pfadname***)**
  Mit der **File**-Methode **Delete()** kann man eine Datei löschen.
- **public static void Copy(String** *pfadQuelle***, String** *pfadZiel***, bool** *überschreiben***)**
  Bei dieser **Copy()** - Überladung erlaubt der Wert **true** des dritten Parameters das Überschreiben einer vorhandenen Zieldatei.
- **public static void Move(String** *pfadQuelle***, String** *pfadZiel***)**
  Zum Umbenennen oder Verschieben einer Datei verwendet man die **File**-Methode **Move()**. Bei identischem Ordner von Quelle und Ziel wird die Datei umbenannt, anderenfalls wird die Datei verschoben.
- **public static void SetCreationTime(String** *pfadname***, DateTime** *kreation***)**
  Mit der Methode **SetCreationTime()** lässt sich für eine Datei das Kreationsdatum setzen.
- **public static void SetLastWriteTime(String** *pfadname***, DateTime** *letzteÄnderung***)**
  Mit der Methode **SetLastWriteTime()** lässt sich für eine Datei das Datum der letzten Änderung setzen.

Zu praktisch allen **File**-Klassenmethoden finden sich Entsprechungen in der Klasse **FileInfo** (als Instanzmethoden oder Eigenschaften). Exemplarisch wird im Beispiel das Löschen einer Datei per **FileInfo**-Objekt vorgeführt.

Wie die Ausgabe des Programms zeigt, lassen sich wichtige Dateieigenschaften leicht fälschen:

```
Die Datei nn.txt wurde
  erstellt:             29.12.2007 22:55:44
  zuletzt geändert:     29.12.2005 22:55:44
```

14.3.2 Ordnerverwaltung

Im folgenden Beispielprogramm werden einige Methoden der Klassen **Directory** und **DirectoryInfo** demonstriert:

```
using System;
using System.IO;

class Ordnerverwaltung {
   static void Main() {
      const string DIR1 = @"U:\Eigene Dateien\C#\EA\";
      const string DIR2 = @"U:\Eigene Dateien\C#\EA\Sub\";

      Directory.CreateDirectory(DIR2);
      Directory.SetCurrentDirectory(DIR1);

      StreamWriter sw = File.CreateText(DIR1 + "demo.txt");
      sw.WriteLine("Directory-Demo");
      sw.Close();

      DirectoryInfo di = new DirectoryInfo(".");

      // Über Dateien informieren (mit Filter)
      FileInfo[] fia = di.GetFiles("*.txt");
      Console.WriteLine("txt-Dateien in {0}", di.FullName);
      Console.WriteLine("  {0, -20} {1, -20}", "Name", "Letzte Änderung");
      foreach (FileInfo fi in fia)
         Console.WriteLine("  {0, -20} {1, -20}", fi.Name, fi.LastWriteTime);

      // Über Unterordner informieren
      DirectoryInfo[] dia = di.GetDirectories();
      Console.WriteLine("\n\nOrdner in {0}", di.FullName);
      Console.WriteLine("  {0, -20} {1, -20}", "Name", "Letzte Änderung");
      foreach (DirectoryInfo die in dia)
         Console.WriteLine("  {0, -20} {1, -20}", die.Name, die.LastWriteTime);

      Directory.SetCurrentDirectory(DIR1+ "\\..");
      Directory.Delete(DIR1, true);
   }
}
```

Wichtige statische Methoden der Klasse **Directory**:

- **public static String GetCurrentDirectory()**
  public static void SetCurrentDirectory(String *pfadname***)**
  Mit **GetCurrentDirectory()** bzw. **SetCurrentDirectory()** kann man das aktuelle Verzeichnis zum laufenden Programm ermitteln bzw. setzen. Das im Aufruf von **SetCurrentDirectory()** angegebene Verzeichnis muss vorhanden sein.
- **public static bool Exists(String** *pfadname***)**
  Mit **Exists()** überprüft man die Existenz eines Ordners.
- **public static DirectoryInfo CreateDirectory(String** *pfadname***)**
  Mit der Methode **CreateDirectory()** lässt sich ein Ordner erstellen, sofern die erforderlichen Zugriffsrechte vorhanden sind. Bei Bedarf werden auch Zwischenordner im Pfad erstellt. Z.B. klappt der folgende Methodenaufruf auch dann, wenn auf dem Laufwerk **U:** noch *kein* Ordner namens **Eigene Dateien** vorhanden ist:

  ```
  Directory.CreateDirectory(@"U:\Eigene Dateien\C#\EA\Sub\");
  ```

- **public static void Delete(String** *pfadname***)**
  Mit der angegebenen **Delete()** - Überladung lässt sich nur ein *leerer* Ordner löschen. Bei einer alternativen Überladung mit Parameter vom Typ **bool** kann man auch ein rekursives Löschen von Unterverzeichnissen und Dateien erzwingen (siehe Beispielprogramm).

Im Konstruktor der Klasse **DirectoryInfo** ist ein Ordnerpfad anzugeben, wobei der aktuelle Pfad des Programms über einen Punkt angesprochen werden kann. Wichtige Instanzmethoden der Klasse **DirectoryInfo**:

- **public FileInfo[] GetFiles(String** *filter***)**
  Bei Aufruf seiner Instanzmethode **GetFiles()** liefert ein **DirectoryInfo**-Objekt einen Array mit **FileInfo**-Objekten zu allen Dateien im Ordner mit passendem Namen. Erlaubte Jokerzeichen im Dateiauswahlfilter:

Jokerzeichen	Bedeutung
?	ersetzt genau ein beliebiges Zeichen
*	steht für eine beliebige (eventuell leere) Zeichenfolge

- **public DirectoryInfo[] GetDirectories(String** *filter***)**
  Analog liefert die Instanzmethode **GetDirectories()** einen Array mit **DirectoryInfo**-Objekten zu den Unterordnern.

14.3.3 Überwachung von Ordnern

Mit einem Objekt der Klasse **FileSystemWatcher** aus dem Namensraum **System.IO** lassen sich die Veränderungen in einem Ordner überwachen (Erzeugen, Löschen, Umbenennen von Einträgen). Das folgende Programm überwacht für die Textdateien (Extension **.txt**) in dem per Befehlszeilenargument angegebenen Ordner die Änderungen beim Datum des letzten Zugriffs und beim Namen:

```
using System;
using System.IO;

public class TxtWatcher {
   static void Main(String[] args) {
      using (FileSystemWatcher watcher = new FileSystemWatcher(args[0])) {

         // Zu Überwachen: Ändern und Umbenennen von Dateien
         watcher.NotifyFilter = NotifyFilters.LastWrite | NotifyFilters.FileName;
         // Filter für Dateinamen
         watcher.Filter = "*.txt";

         // Ereignisbehandlungmethoden registrieren
         watcher.Changed += watcher_Changed;
         watcher.Created += watcher_Changed;
         watcher.Deleted += watcher_Changed;
         watcher.Renamed += watcher_Renamed;

         // Überwachung aktivieren
         watcher.EnableRaisingEvents = true;

         Console.WriteLine("TxtWatcher gestartet. Beenden mit 'q'\n");
         Console.WriteLine("Überwachter Ordner: " + args[0] + "\n");
         ConsoleKeyInfo cki;
         do
            cki = Console.ReadKey(true);
         while (cki.KeyChar != 'q');
      }
   }
```

```
      // Ereignisroutinen implementieren
      static void watcher_Changed(Object source, FileSystemEventArgs e) {
         Console.WriteLine("Datei {0} {1}", e.Name, e.ChangeType);
      }

      static void watcher_Renamed(Object source, RenamedEventArgs e) {
         Console.WriteLine("Datei {0} umbenannt in {1}", e.OldName, e.Name);
      }
   }
```

Im Übrigen demonstriert das Programm, dass Ereignisse nicht unbedingt von GUI-Komponenten stammen müssen, und dass auch Konsolenanwendungen auf Ereignisse reagieren können.

Eine Beispielausgabe:[1]

```
TxtWatcher gestartet. Beenden mit 'q'

Überwachter Ordner: U:\Eigene Dateien\C#\EA

Datei Neues Textdokument.txt Created
Datei Neues Textdokument.txt Changed
Datei Neues Textdokument.txt umbenannt in test.txt
Datei test.txt Changed
Datei test.txt Deleted
```

Mit der **Console**-Methode **ReadKey**() kann man sofort auf Tastendrücke reagieren und dabei die Ausgabe von Zeichen auf dem Bildschirm verhindern (Wert **true** für den ersten und einzigen Parameter). Man erhält eine Instanz der Struktur **ConsoleKeyInfo**, die u.a. das zu einer Taste gehörige Unicode-Zeichen kennt.

Während der Haupt-Thread des Konsolenprogramms auf Tastatureingaben lauert, werden die die Dateisystemereignis-Behandlungsmethoden in einem separaten Thread ausgeführt.

Im Beispiel wird mit Hilfe einer **using**-Anweisung (vgl. Abschnitt 14.1.4.3) dafür gesorgt, dass alle mit dem **FileSystemWatcher** verbundenen Ressourcen nach dem Beobachtungsende auf jeden Fall frei gegeben werden.

14.4 Übungsaufgaben zu Kapitel 14

1) Erstellen Sie ein Statistikprogramm zur Berechnung des Mittelwerts, das als Eingabe eine Textdatei mit Daten akzeptiert, wobei das Semikolon als Trennzeichen dient. In folgender Beispieldatei liegen drei Variablen (Spalten) für fünf Fälle vor:

```
12;3;345
7;5;298
9;4;411
10;2;326
5;6;195
4;sieben;120
```

Die gültigen Werte der zweiten Spalte haben z.B. den Mittelwert 4,0. Ihr Programm sollte auf irreguläre Daten folgendermaßen reagieren:

- Warnung ausgeben
- mit den verfügbaren Werten rechnen

Auf obige Daten sollte Ihr Programm ungefähr so reagieren:

---

[1] Auf einem Rechner unter Windows 10 war zu beobachten, dass bei einer Dateiänderung das Ereignis **Changed** zweimal gefeuert wird.

```
Mittelwertsberechnung für die Datei daten.txt

Warnung: Token 2 in Zeile 6 ist keine Zahl.

            Variable           Mittelwert         Valide Werte
                   1                7,833                    6
                   2                4,000                    5
                   3              282,500                    6
```

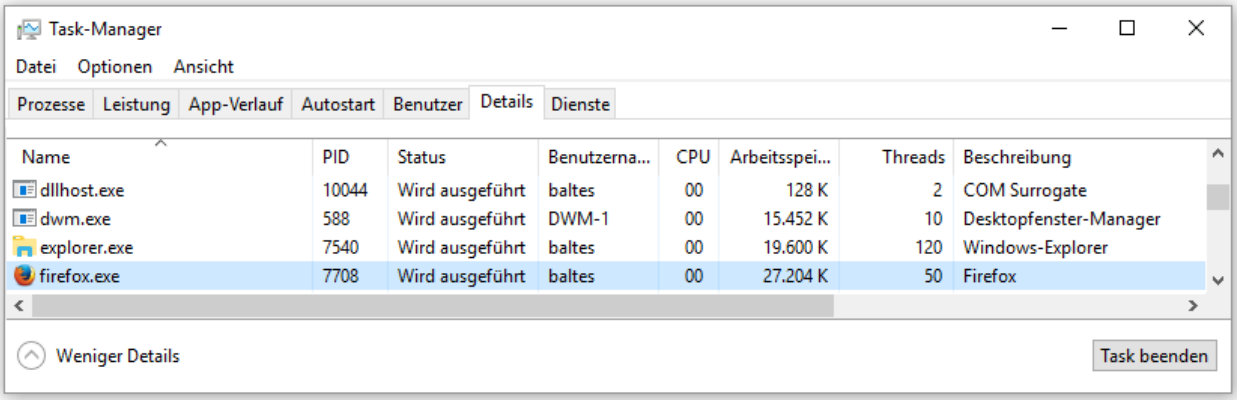
2) Wie kann man den Quellcode des folgenden Programms vereinfachen und dabei auch noch die Laufzeit erheblich reduzieren?

```
using System;
using System.IO;

class StreamWriterDemo {
   static void Main() {
      long zeit = DateTime.Now.Ticks;;
      StreamWriter sw = new StreamWriter("demo.txt");
      sw.AutoFlush = true;
      for (int i = 1; i < 30000; i++) {
         sw.WriteLine(i);
      }
      sw.Close();
      Console.WriteLine("Zeit: "+((DateTime.Now.Ticks-zeit)/1.0e4) + " Millisek.");
   }
}
```

15 Multithreading

Wir sind längst daran gewöhnt, dass moderne Betriebssysteme mehrere Programme (Prozesse) parallel betreiben können, so dass z.B. ein längerer Ausdruck keine Zwangspause des Benutzers zur Folge hat. Während der Druckertreiber die Ausgabeseiten aufbaut, kann z.B. ein C# - Programm entwickelt oder im Internet recherchiert werden. Weil in der Regel weniger Prozessorkerne vorhanden sind als Programme, muss das Betriebssystem die verfügbare CPU-Leistung nach einem Zeitscheibenverfahren auf die rechenwilligen Programme verteilen. Dadurch reduziert sich zwar die Ausführungsgeschwindigkeit jedes Programms im Vergleich zum Solobetrieb, doch ist in den meisten Anwendungen trotzdem ein flüssiges Arbeiten möglich.

Als Ergänzung zum gerade beschriebenen **Multitasking**, das ohne Zutun der Anwendungsprogrammierer vom Betriebssystem bewerkstelligt wird, ist es oft sinnvoll oder gar unumgänglich, auch *innerhalb* einer Anwendung nebenläufige *Ausführungsfäden* zu realisieren, wobei man hier vom **Multithreading** spricht. Bei einem Internet-Browser muss man z.B. nach dem Anstoßen eines längeren Downloads nicht untätig den Fortschrittsbalken im Download-Fenster anstarren, sondern kann parallel mit anderen Fenstern arbeiten. Wie unter Windows die Detailsanzeige im Task-Manager zeigt, sind z.B. bei einer typischen Firefox-Sitzung 50 Threads aktiv, wobei die Anzahl ständig schwankt, z.B.:



Bei einer GUI-Anwendung sorgt die Multithreading-Technik dafür, dass die Bedienoberfläche auch dann noch auf Benutzereingaben reagiert, wenn im Hintergrund ein zeitaufwändiger Auftrag erledigt wird. Eine Server-Anwendung kann dank Multithreading mehrere Klienten simultan versorgen.

Die Multithreading-Technik kommt aber nicht nur dann in Frage, wenn eine Anwendung mehrere Aufgaben gleichzeitig erledigen soll. Sind auf einem Rechner *mehrere* Prozessoren oder Prozessorkerne verfügbar, dann sollten aufwändige Einzelaufgaben (z.B. das Rendern einer 3D-Ansicht, Virenanalyse einer kompletten Festplatte) in Teilaufgaben zerlegt und auf mehrere CPU-Kerne verteilt werden. Mittlerweile (2017) sind 4 reale Kerne guter Standard, und dank Intels Hyper-Threading - Technologie sieht das Betriebssystem bei einer Quad-Core - CPU in der Regel sogar 8 logische Kerne. Multi-Core - CPUs erhöhen den Druck auf die Software-Entwickler, per Multithreading für gut skalierende Anwendungen zu sorgen, die auf einem Quad-Core - Rechner deutlich schneller als auf einem Single-Core - Rechner laufen.

Beim Multithreading ist allerdings eine sorgfältige Einsatzplanung erforderlich, denn:

- Das Erstellen, Terminieren, Blockieren und Reaktivieren von Threads ist zeitaufwendig und sollte daher nicht unnötig häufig passieren.
- Threads belegen Speicher (z.B. per Voreinstellung 1 MB für ihren individuellen Stack), so dass ihre Zahl nicht zu groß werden sollte.
- Die nebenläufige Programmierung ist erheblich anspruchsvoller als die Single-Thread - Programmierung. Wenn mehrere Threads auf gemeinsame Datenbestände zugreifen, stellt das Synchronisieren der Threads eine Herausforderung dar. Hier kommt es oft zu Fehlern, die zudem aufgrund variabler Folgen schwer zu analysieren sind.

Während jeder *Prozess* einen eigenen Adressraum besitzt, laufen die *Threads* eines Programms im selben Adressraum ab. Sie haben einen gemeinsamen Heap-Speicher, wohingegen jeder Thread als selbständiger Kontrollfluss aber einen eigenen Stack-Speicher benötigt.

Bei C# ist die Multithreading-Unterstützung in Sprache, Standardbibliothek und Laufzeitumgebung integriert. Folglich gehört diese Technik in C# nicht zum Guru-HighTech - Repertoire, sondern kann von *jedem* Programmierer ohne großen Aufwand genutzt werden.

Übrigens sind auch ohne unser Zutun in jeder .NET - Anwendung mehrere Threads aktiv. So läuft z.B. der Garbage Collector stets in einem eigenen Thread.

Wir erarbeiten uns in diesem Kapitel zunächst ein Multithreading-Basiswissen durch den Einsatz von dedizierten, für einen bestimmten Zweck erstellten Threads.

In der Praxis geht es darum, mit möglichst wenigen Threads eine gute Performanz zu erzielen und außerdem das aufwändige Kreieren und Terminieren von Threads so weit wie möglich zu vermeiden. Um diese Ziele zu unterstützen, verwaltet die CLR einen Threadpool. Hier befinden sich Threads in einer zur Hardware (zur Anzahl der logischen CPU-Kerne) passenden Zahl, die nach Erledigung einer Aufgabe nicht beendet werden, sondern auf neue Aufgaben warten. Statt für eine konkrete Aufgabe (z.B. Bedienung eines Webzugriffs) jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, werden eingehende Aufträge einem freien Pool-Thread zugeteilt oder in eine Warteschlange gestellt.

Der Threadpool wird von diversen Klassen im .NET-Framework verwendet, wobei sich im Laufe der .NET - Evolution unterschiedliche Programmiermuster etabliert haben. Das *Asynchronous Programming Model* (APM) war vom Anfang dabei und galt speziell bei der asynchronen Ein- und Ausgabe (Dateisystem, Netzwerk, Datenbankzugriff) lange als Standardlösung. Seit .NET 4.0 sollte zur performanten und skalierbaren Nutzung von Mehrkernsystemen die *Task Parallel Library* (TPL) bevorzugt werden. Auch die mit C# 5 eingeführte Multithreading-Unterstützung durch die Programmiersprache (Methodenmodifikator **async** und Operator **await**) arbeitet in der Regel mit der TPL.

Eine vollständige Behandlung der Multithreading - Optionen in C# bzw. .NET 4.x erfordert eine eigenständige Monographie von beträchtlichem Umfang. Wer über das aktuelle Kapitel hinaus weiterer Informationen benötigt, findet diese z.B. in Agafonov & Koryavchenko (2015), Albahari & Albahari (2015), Cleary (2014) und Griffiths (2013).

15.1 Threads

Ein Thread wird im .NET - Framework über ein Objekt der gleichnamigen Klasse aus dem Namensraum **System.Threading** realisiert.[1] Das direkte Erzeugen von Threads über Objekte aus der Klasse **Thread** wird zunehmend abgelöst durch die Nutzung von Frameworks, die im Hintergrund mit Multithreading-Techniken arbeiten. Allerdings erleichtert es der traditionelle Direktkontakt mit Threads, grundlegende Eigenschaften der Technik kennen zu lernen.

Der Abschnitt 15.1 startet elementar, lässt aber im weiteren Verlauf fast keinen Aspekt der potentiell komplexen Multithreading-Programmierung aus. Vermutlich wäre eine Verteilung der (inhaltlich zusammengehörigen) Themen auf mehrere, nach Schwierigkeit und Verwendungshäufigkeit gegliederte Kapitel eine didaktisch sinnvolle Alternative, um den Einstieg zu erleichtern.

15.1.1 Threads erzeugen

Jede Instanz- oder Klassenmethode, die entweder den Delegatentyp

public delegate void ThreadStart()

oder den Delegatentyp

public delegate void ParameterizedThreadStart(Object *obj***)**

erfüllt, kann in einem eigenen Thread gestartet werden, indem ein zugehöriges Delegatenobjekt erzeugt und dem **Thread**-Konstruktor übergeben wird, z.B.:

```
Thread pt = new Thread(new ThreadStart(pro.Run));
```

Man kann sich das explizite Notieren des Delegaten-Konstruktors sparen:

```
Thread pt = new Thread(pro.Run);
```

Einige Überladungen das **Thread**-Konstruktors ermöglichen es, über einen Parameter vom Typ **int** die voreingestellte Stapelspeichergröße (von 1 MB) zu ändern, was aber nur selten erforderlich ist.[2] Möglicherweise kann bei einem rekursiven Algorithmus (siehe Abschnitt 4.7.1) eine **StackOverflowException** durch eine Stapelvergrößerung verhindert werden.

15.1.1.1 Produzent - Konsument - Beispiel

Das obige Beispiel zur Thread-Kreation stammt aus einem „betriebswirtschaftlichen“ Programm mit einem Objekt aus einer Klasse `Produzent` und einem Objekt aus einer Klasse `Konsument`, die auf einen Lagerbestand einwirken, der von einem Objekt der Klasse `Lager` gehütet wird. Produzent und Konsument entfalten ihre Tätigkeit jeweils im Rahmen einer Methode namens `Run()`, die in einem eigenen Thread läuft (siehe unten).

15.1.1.2 Die Klasse Lager

Ein Objekt der Klasse `Lager` beherrscht die folgenden Methoden, um den Produzenten und den Konsumenten zu bedienen, solange nicht eine Maximalzahl von Lagerzugriffen überschritten ist:

- `public bool Ergaenze(int add)`
  Diese Methode wird vom Produzenten genutzt, um Ware virtuell einzuliefern. Ist das Lager bereits geschlossen, wird **false** zurückgemeldet, sonst **true**.

---

[1] Ein Thread im Sinne der CLR (ein *verwalteter Thread*) muss nicht unbedingt 1:1 auf einen Thread im Sinne des zugrunde liegenden Betriebssystems abgebildet werden (siehe https://msdn.microsoft.com/en-us/library/74169f59(v=vs.110).aspx).

[2] Auf dieser MSDN-Webseite ist die voreingestellte Stapelgröße dokumentiert:
https://msdn.microsoft.com/de-de/library/8cxs58a6.aspx

- `public bool Liefere(int sub)`
  Diese Methode wird vom Konsumenten genutzt, um Ware virtuell zu beziehen. Ist das Lager bereits geschlossen, wird **false** zurückgemeldet, sonst **true**.
- `void Rumoren()`
  Diese private Methode dient dazu, Aufwand beim Ausführen der Aufträge zu simulieren.

In den Methoden `Ergaenze()` und `Liefere()` wird zur formatierten Zeitausgabe eine spezielle Überladung der **DateTime**-Methode **ToString()**

```
DateTime.Now.ToString("T", ci)
```

verwendet, wobei ein Objekt der Klasse **CultureInfo** (Namensraum **System.Globalization**) beteiligt ist:

```
System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("de-DE");
```

Ein `Lager`-Objekt verwaltet in den privaten Feldern `bilanz` und `anz` den aktuellen Lagerbestand (initialisiert mit der Konstanten `STARTKAP`) und die Anzahl der bisherigen Lagerzugriffe (nach oben begrenzt durch die Konstante `MANZ`):

```
using System;
using System.Threading;

public class Lager {
 int bilanz;
 int anz;
 const int MANZ = 20;
 const int STARTKAP = 100;
 System.Globalization.CultureInfo ci = new System.Globalization.CultureInfo("de-DE");

 public Lager(int start) {
  bilanz = start;
 }

 public bool Ergaenze(int add) {
  if (anz < MANZ) {
   bilanz += add;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
     anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
     ", es ist Feierabend!");
   return false;
  }
 }

 public bool Liefere(int sub) {
  if (anz < MANZ) {
   bilanz -= sub;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
     anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
     ", es ist Feierabend!");
   return false;
  }
 }
```

```
  void Rumoren() {
    double d;
    for (int i = 0; i < 40000; i++)
      d = i * i;
  }

  static void Main() {
    Lager lager = new Lager(STARTKAP);
    Console.WriteLine("Das Lager ist offen (Bestand: {0})\n", STARTKAP);
    Produzent pro = new Produzent(lager);
    Konsument kon = new Konsument(lager);
    Thread pt = new Thread(pro.Run);
    Thread kt = new Thread(kon.Run);
    pt.Name = "Produzent";
    kt.Name = "Konsument";
    pt.Start();
    kt.Start();
  }
}
```

15.1.1.3 Die Klassen `Produzent` und `Konsument`

Produzent und Konsument kommen mit einer recht simplen Klassendefinition aus:

```
using System;
using System.Threading;

public class Produzent {
   Lager pl;
   Random rand = new Random(1);

   public Produzent(Lager ptr) {
      pl = ptr;
   }

   public void Run() {
      while (pl.Ergaenze(5 + rand.Next(100)))
         Thread.Sleep(1000 + rand.Next(3000));
   }
}

public class Konsument {
   Lager pl;
   Random rand = new Random(2);

   public Konsument(Lager ptr) {
      pl = ptr;
   }

   public void Run() {
      while (pl.Liefere(5 + rand.Next(100)))
         Thread.Sleep(1000 + rand.Next(3000));
   }
}
```

Weil Produzent und Konsument mit dem `Lager`-Objekt kooperieren sollen, erhalten sie als Konstruktor-Parameter eine entsprechende Referenz.

Neben dem Konstruktor ist jeweils eine Methode namens `Run()` vorhanden, welche den Delegatentyp **ThreadStart** erfüllt und sich damit als Startmethode für einen Thread eignet. Die `Run()` - Methoden beschränken sich auf eine **while**-Schleife, wobei in jedem Durchgang ein Auftrag zum Ein- bzw. Auslagern einer zufallsbestimmten Menge an das `Lager`-Objekt geschickt wird. Zwischen

zwei Aufträgen machen die `Run()` - Methoden eine Pause von zufallsabhängiger Länge, indem Sie die statische **Thread**-Methode **Sleep()** aufrufen. Diese befördert den Thread vom Zustand **Running** in den Zustand **WaitSleepJoin** (siehe unten). Mit festen Startwerten für die Pseudozufallszahlengeneratoren aus der Klasse **Random** soll dafür gesorgt werden, dass stets derselbe Lagerverlauf resultiert. Dies gelingt aber nur teilweise, weil der etwas früher gestartete Produzenten-Thread aufgrund von CLR-internen Vorgängen nicht unbedingt als erster beim Lager ankommt.

15.1.2 Threads starten

Die **Main()** - Methode der Klasse `Lager` erzeugt als Startmethode des Programms die beteiligten Objekte:

- einen Lageristen (Objekt `lager` aus der Klasse `Lager`)
  ```
  Lager lager = new Lager(STARTKAP);
  ```
- einen Produzenten (Objekt `pro` aus der Klasse `Produzent`)
  ```
  Produzent pro = new Produzent(lager);
  ```
- einen Konsumenten (Objekt `kon` aus der Klasse `Konsument`)
  ```
  Konsument kon = new Konsument(lager);
  ```
- einen Thread, dessen Ausführung mit der `Run()` - Methode des Produzenten startet (Objekt `pt` aus der Klasse **Thread**)
  ```
  Thread pt = new Thread(pro.Run);
  ```
- einen Thread, dessen Ausführung mit der `Run()` - Methode des Konsumenten startet (Objekt `kt` aus der Klasse **Thread**)
  ```
  Konsument kon = new Konsument(lager);
  ```

Schließlich erhalten die Threads einen Namen und werden gestartet:

```
pt.Name = "Produzent";
kt.Name = "Konsument";
pt.Start();
kt.Start();
```

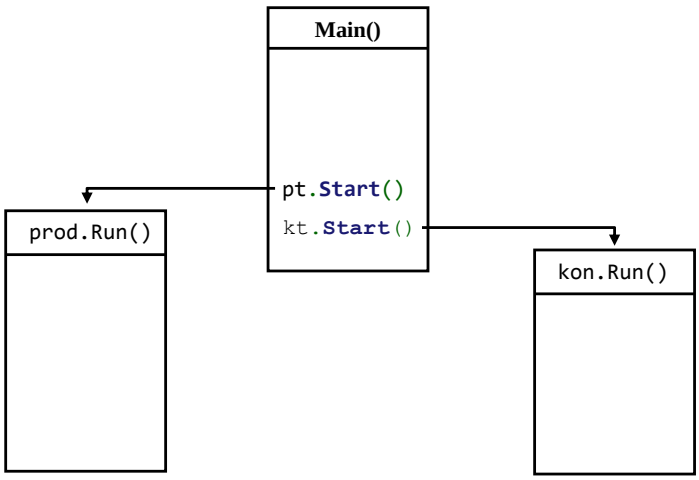
Der **Thread** `pt` startet mit der Ausführung der `Run()` - Methode durch das Objekt `pro` aus der Klasse `Produzent`, und der **Thread** `kt` startet mit der Ausführung der `Run()` - Methode durch das Objekt `kon` aus der Klasse `Konsument`.

Beide Threads laufen im *Vordergrund*, und ein Prozess gilt als aktiv, solange er mindestens einen aktiven Vordergrund-Thread besitzt. Ist ein Thread über seine Eigenschaft **IsBackground** als **Hintergrund-Thread** markiert, kann er ein Programm nicht am Leben erhalten, sondern wird ggf. abrupt mit dem letzten Vordergrund-Thread beendet. Folglich eignet sich ein Hintergrund-Thread nur für Tätigkeiten, die ohne Risiko abgebrochen werden dürfen (z.B. Musik abspielen in einem Spiel).

Unmittelbar vor dem Ende der **Main()** - Methode sind **drei** Vordergrund-Threads aktiv: [1]

- Der **primäre** Thread des Programms lebt, solange die **Main()** - Methode läuft.
- Außerdem agieren zu diesem Zeitpunkt die beiden zusätzlich gestarteten **sekundären** Threads. Sie enden mit ihrer Startmethode (`prod.Run()` bzw. `kon.Run()`), sofern sie nicht zuvor abgebrochen werden (siehe unten).

[1] Zu einem Programm gehören noch weitere Threads, die im Hintergrund von der CLR verwaltet werden (z.B. für den Garbage Collector). Endet der letzte Vordergrund-Thread eines Programms, werden seine Hintergrund-Threads automatisch ebenfalls beendet.



Die beiden Aufrufe der **Thread**-Methode **Start()** kehren praktisch unmittelbar zurück, und anschließend endet mit der **Main()** - Methode auch der primäre Thread. Die beiden sekundären Threads leben weiter bis zum Ende ihrer jeweiligen Startmethode, und das Programm endet mit seinem letzten Vordergrund-Thread.

Ein Thread endet, wenn seine Startmethode abgearbeitet ist. Er befindet sich dann im Zustand **Stopped** und kann *nicht* erneut gestartet werden. Im Beispiel passiert dies, sobald eine der `Run()` - Methoden zu Beginn eines **while** - Schleifendurchgangs ein geschlossenes Lager festgestellt, wenn also der Methodenaufruf `lager.Ergaenze()` bzw. `lager.Liefere()` zum Rückgabewert **false** führt.

Der primäre Thread des Programms ist zu diesem Zeitpunkt ebenfalls bereits Geschichte, weil er mit der `Lager`-Methode **Main()** seine Tätigkeit einstellt. Folglich endet das Programm, wenn Produzenten- und Konsumenten-Thread sich verabschiedet haben.

15.1.3 Klassen aus dem Anwendungsbereich und aus der Informatik

Das Beispiel demonstriert mit seinem Objekt-Ensemble, dass einige Objekte direkt aus der Abbildung des Anwendungsbereichs stammen (Lagerist, Produzent, Konsument), während die Objekte der Klasse **Thread** als Konstrukte der Informatik zur Parallelisierung von Aufgaben dienen. Man kann einen Thread als *Ausführungsfaden* oder *Aktivitätsträger*[1] bezeichnen. Wir haben früher gelegentlich von der *objektorientierten Bühne* gesprochen und können nun zur Veranschaulichung von Multithreading zum Plural übergehen, in einem sekundären *Aktivitätsträger* also eine zusätzliche Bühne mit eigenem Handlungsablauf sehen.

Im Ausführungsfaden `pt` des Beispielprogramms wird die Startmethode `Run()` von einem Objekt aus der Klasse `Produzent` ausgeführt. Alle von einer Startmethode via Methodenaufruf direkt oder indirekt initiierten Aufträge an andere Objekte oder Klassen laufen ebenfalls im selben Thread (auf derselben Bühne) ab, so dass in einem Ausführungsfaden beliebig viele Akteure tätig werden können. Wir reden zwar reden vom *Produzenten*-Thread, weil dieser Aktivitätsträger mit der Ausführung der Methode `Run()` durch ein Objekt der Klasse `Produzent` startet, doch wird in diesem Thread auch das `Lager`-Objekt aktiv (über Aufrufe seiner `Ergaenze()`-Methode).

Andererseits kann ein einzelner Akteur (z.B. ein Objekt) in mehreren Threads arbeiten, wenn er entsprechende Botschaften erhält. Im Beispiel kommt das `Lager`-Objekt sowohl im Produzenten- als auch im Konsumenten-Thread zum Einsatz: Die Methoden `Ergaenze()` und `Liefere()` erhöhen oder reduzieren den Lagerbestand, aktualisieren die Anzahl der Lagerveränderungen und

---

[1] http://de.wikipedia.org/wiki/Thread_(Informatik)

protokollieren jede Maßnahme. Dazu besorgen sie sich mit der stationären **Thread**-Methode **CurrentThread()** eine Referenz auf den aktuell ausgeführten Thread und ermitteln dessen **Name**-Eigenschaft.

Wenn die Vorstellung eines Lageristen stört, der simultan in zwei Threads (auf zwei Bühnen) tätig ist, dann stelle man sich ein *Team* von Lagerarbeitern vor, was der Realität vieler Betriebe recht gut entspricht.

15.1.4 Threads koordinieren

In einem typischen Ablaufprotokoll des Produzenten - Konsumenten - Programms zeigen sich einige Ungereimtheiten, verursacht durch das unkoordinierte Agieren der beiden Threads:

```
Das Lager ist offen (Bestand: 100)

Nr.  2:  Produzent ergänzt   29 um 02:01:58 Uhr. Stand: 47
Nr.  2:  Konsument entnimmt  82 um 02:01:58 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:02:00 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:02:01 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:02:03 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:02:04 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:02:05 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:02:06 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:02:09 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:02:09 Uhr. Stand: 75
Nr. 11:  Konsument entnimmt  81 um 02:02:10 Uhr. Stand: -6
Nr. 12:  Konsument entnimmt   5 um 02:02:11 Uhr. Stand: -11
Nr. 13:  Produzent ergänzt    7 um 02:02:12 Uhr. Stand: -4
Nr. 14:  Konsument entnimmt  43 um 02:02:13 Uhr. Stand: -47
Nr. 15:  Produzent ergänzt   37 um 02:02:14 Uhr. Stand: -10
Nr. 16:  Konsument entnimmt  75 um 02:02:15 Uhr. Stand: -85
Nr. 17:  Konsument entnimmt  78 um 02:02:17 Uhr. Stand: -163
Nr. 18:  Produzent ergänzt   73 um 02:02:18 Uhr. Stand: -90
Nr. 19:  Konsument entnimmt  13 um 02:02:18 Uhr. Stand: -103
Nr. 20:  Konsument entnimmt  37 um 02:02:20 Uhr. Stand: -140

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

U.a. fällt negativ auf:

- Im ersten Protokolleintrag wird berichtet, dass vom Startwert 100 ausgehend eine Lieferung von 29 Einheiten zu einem Bestand von 47 Einheiten geführt habe, und auch die Auftragsnummer 2 ist falsch.
- Im zweiten Eintrag wird behauptet, dass die Entnahme von 82 Einheiten ohne Effekt auf den Lagerbestand geblieben sei.
- Der Bestand wird negativ, was in einem realen Lager nicht passieren kann.

Wenn es sich nicht vermeiden lässt, dass mehrere Threads gemeinsame Daten verwenden und dabei auch schreibend zugreifen, sind Maßnahmen zur Koordination der Zugriffe erforderlich.

15.1.4.1 Zugriffsexklusivität

Während ein Thread auf gemeinsam genutzte Daten zugreift und potentiell verändert, müssen andere Threads am simultanen Zugriff gehindert werden, um inkonsistente Zustände bzw. fehlerhafte Interpretationen zu verhindern.

15.1.4.1.1 Die lock-Anweisung

Am Anfang des oben wiedergegebenen Ablaufprotokolls stehen zwei „wirre“ Einträge, die folgendermaßen durch eine so genannte **Race Condition** zu erklären sind:

- Der etwas früher gestartete Produzenten-Thread kommt als erster beim Lager an, ruft die `Lager`-Methode `Ergaenze()` mit dem Parameterwert 29 auf und bringt mit den Anweisungen
  ```
  bilanz += add;
  anz++;
  ```
  die `bilanz` auf den Wert 129 sowie die Auftragsnummer auf den Wert 1.
- Dann unterbricht das Laufzeitsystem den Produzenten-Thread und aktiviert den Konsumenten-Thread.
- Dieser ruft die `Lager`-Methode `Liefere()` mit dem Parameterwert 82 auf und bringt mit den Anweisungen
  ```
  bilanz -= sub;
  anz++;
  ```
  die Lagerbilanz auf 47 sowie die Auftragsnummer auf den Wert 2.
- Nun kommt der Produzenten-Thread wieder zum Zug und schreibt seinen Protokolleintrag, wobei er die mittlerweile vom Konsumenten-Thread veränderten Werte von `anz` und `bilanz` verwendet.
- Dann schreibt auch der Konsumenten-Thread seine Protokollzeile. Allerdings ist der Stand von 47 nur dann nachvollziehbar, wenn man die vorherige, nicht korrekt protokollierte Lieferung berücksichtigt.

Es kann nicht nur zu wirren Protokolleinträgen kommen, sondern auch zu einem fehlerhaften `bilanz`-Wert. Scheinbar einschrittige Operationen wie die folgende Anweisung in der vom Produzenten-Thread aufgerufenen Methode `Ergaenze()`

```
bilanz += add;
```

haben in einen Rechner mehrere Teilschritte zur Folge, sind also nicht **atomar**, z.B.:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) erhöhen
- Neuen Wert in den Hauptspeicher schreiben

In der vom Konsumenten-Thread aufgerufenen Methode `Liefere()` führt die Anweisung

```
bilanz -= sub;
```

analog zu folgenden Teilschritten:

- aktuellen `bilanz`-Wert aus dem Hauptspeicher in ein CPU-Register einlesen
- Wert (der lokalen Kopie!) reduzieren
- Neuen Wert in den Hauptspeicher schreiben

Durch unglückliche Thread-Wechsel kann es z.B. zu folgender Sequenz kommen:

- Der Produzent liest den Wert 100.
- Der Konsument liest den Wert 100.
- Der Produzent erhöht seine `bilanz`-Kopie um 10 auf 110 und schreibt das Ergebnis in den Hauptspeicher.
- Der Konsument reduziert seine `bilanz`-Kopie um 10 auf 90 und schreibt das Ergebnis in den Hauptspeicher. Damit ist der Beitrag des Produzenten verloren gegangen.

Auf einem Rechner mit 32-Bit-Betriebssystem kann es sogar passieren, dass ein Thread beim Schreiben eines **long**- oder **double**-Werts (64 Bit groß) unterbrochen wird, und dass schlussendlich

die 64 Bit einer Variablen von zwei verschiedenen Threads geschrieben werden. Auf einem Rechner mit 64-Bit-Betriebssystem werden **long**- und **double**-Werte atomar gelesen und geschrieben, nicht jedoch die 128 Bit großen Werte vom Typ **decimal**.

Im Beispielprogramm muss offenbar verhindert werden, dass während eines Lagerzugriffs ein Thread-Wechsel stattfindet. Dies ist in C# leicht zu realisieren, indem per **lock**-Anweisung der kritische Anweisungsblock mit einer Sperre versehen wird, z.B.:

```
Object lockObject = new Object();
 . . .
public bool Ergaenze(int add) {
 lock(lockObject) {
  if (anz < MANZ) {
   bilanz += add;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Zum Sperren kann jedes beliebige Objekt verwendet werden. Richter (2006, S. 642ff) rät aber zur Verwendung eines *privaten* Member-Objekts, weil ein öffentlich bekanntes Sperrobjekt zum Blockieren der Anwendung durch eine unbefugte oder ungeschickte Verwendung führen kann. Im Beispiel wird dieser Vorschlag realisiert.

Ferner sollte zum Sperren kein Objekt einer *fremden* Klasse verwendet werden. Das ist auch dann riskant, wenn alle Referenzen auf das Objekt unter Kontrolle sind, weil in der unbekannten Klassendefinition das Objekt eventuell per **this**-Referenz für interne Sperrzwecke verwendet wird (Griffiths 2013, S. 616).

Im Beispiel muss auch der kritische Bereich in der Methode `Liefere()` durch *dasselbe* Objekt gesperrt werden:

```
public bool Liefere(int sub) {
 lock (lockObject) {
  if (anz < MANZ) {
   bilanz -= sub;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Ein Anweisungsblock mit **lock**-reguliertem Zugang wird auch als *synchronisiert* bezeichnet.

Beim Betreten eines noch freien geschützten Bereichs setzt ein Thread per **lock** die Sperre. Man kann sich vorstellen, dass er den einzigen Schlüssel für den geschützten Bereich erwirbt. Jedem

anderen Thread wird der Zutritt verwehrt. Er wird in den Zustand **WaitSleepJoin** versetzt und muss warten. Beim Verlassen des geschützten Bereichs wird die Sperre aufgehoben, und ggf. kann ein wartender Thread seine Arbeit fortsetzen. Um andere Threads möglichst wenig zu behindern, muss ein Sperrobjekt so schnell wie möglich wieder frei gegeben werden.

Aufgrund der Thread-Synchronisation per Sperrobjekt produziert unser Beispielprogramm keine wirren Protokolleinträge mehr, doch kann es nach wie vor zu einem negativen Lagerzustand kommen, z.B.:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 02:29:23 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 02:29:23 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:29:24 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:29:25 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:29:27 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:29:29 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:29:30 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:29:31 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:29:33 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:29:33 Uhr. Stand: 75
Nr. 11:  Konsument entnimmt  81 um 02:29:34 Uhr. Stand: -6
Nr. 12:  Konsument entnimmt   5 um 02:29:35 Uhr. Stand: -11
Nr. 13:  Produzent ergänzt    7 um 02:29:36 Uhr. Stand: -4
Nr. 14:  Konsument entnimmt  43 um 02:29:38 Uhr. Stand: -47
Nr. 15:  Produzent ergänzt   37 um 02:29:38 Uhr. Stand: -10
Nr. 16:  Konsument entnimmt  75 um 02:29:40 Uhr. Stand: -85
Nr. 17:  Konsument entnimmt  78 um 02:29:41 Uhr. Stand: -163
Nr. 18:  Produzent ergänzt   73 um 02:29:42 Uhr. Stand: -90
Nr. 19:  Konsument entnimmt  13 um 02:29:43 Uhr. Stand: -103
Nr. 20:  Konsument entnimmt  37 um 02:29:45 Uhr. Stand: -140

Lieber Produzent, es ist Feierabend!

Lieber Konsument, es ist Feierabend!
```

Es ist zu beachten, dass durch **lock**-Anweisungen ein geschützter *Code*-Bereich entsteht, nicht aber ein geschützter *Speicher*bereich. Damit ein geschützter Speichbereich resultiert, müssen *alle* Code-Passagen mit Zugriff auf diesen Speicherbereich in die Zone mit exklusivem Zugriff einbezogen werden (Morrison 2005a).

Wenn eine komplette Methode in den geschützten Bereich einbezogen werden soll, der bei Instanzmethoden vom handelnden Objekt (anzusprechen mit **this**) bzw. bei statischen Methoden vom **Type**-Objekt zur Klasse (anzusprechen mit **typeof(***Klasse***)**) überwacht wird, kann der Methode ein **MethodImplAttribute** mit passendem Parameterwert vorangestellt werden, z.B.:[1]

```
[MethodImplAttribute(MethodImplOptions.Synchronized)]
public bool Ergaenze(int add) {
 if (anz < MANZ) {
  bilanz += add;
  anz++;
  Rumoren();
  Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
     anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
  return true;
 } else {
  Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
   ", es ist Feierabend!");
  return false;
 }
}
```

[1] Vorbild für diese Sperrtechnik war wohl der Modifikator **synchronized** für Methoden in der Programmiersprache Java.

Man überlässt dem Compiler das Erstellen der **lock**-Anweisung und muss dabei ein *öffentliches* Sperrobjekt in Kauf nehmen, das Blockaden durch ungeschickte oder böswillige Software ermöglicht (siehe oben).

15.1.4.1.2 Die Klasse Monitor

In diesem Abschnitt wird sich herausstellen:

- Für die Umsetzung der **lock**-Anweisung ist die Klasse **Monitor** zuständig.
- Die statischen **Monitor**-Methoden **Wait()** und **Pulse()** ermöglichen den beteiligten Threads eine intelligente Kooperation.

15.1.4.1.2.1 Realisation der lock-Anweisung

Für die Thread-Synchronisation per **lock**-Anweisung sorgt letztlich die Klasse **Monitor** aus dem Namensraum **System.Threading**. Sie eignet sich weder zum Ableiten neuer Klassen noch zum Erzeugen von Objekten, spielt aber eine zentrale Rolle bei Multithreading-Anwendungen im .NET - Framework. Die **lock**-Anweisung im folgenden Code-Segment

```
var lockObj = new Object();
lock(lockObj) {
   . . .                                          // Synchronisierte Anweisungen
}
```

wird vom C# - Compiler der Roslyn-Generation umgesetzt in: [1]

```
var lockObj = new Object();
bool lockTaken = false;
try {
   Monitor.Enter(lockObj, ref lockTaken);
   . . .                                          // Synchronisierte Anweisungen
} finally {
   if(lockTaken)
      Monitor.Exit(lockObj);
}
```

Hier kommen zwei statische **Monitor**-Methoden zum Einsatz:

- **public static void Enter(Object** *obj***)**
  Das Sperrobjekt wird erworben, wobei der Referenzparameter `lockTaken` den Erfolg dokumentiert.
- **public static void Exit(Object** *obj***)**
  In dieser Methode wird das Sperrobjekt zurückgegeben. Weil eine Ausnahme bei **Enter()** aufgetreten sein könnte (z.B. durch einen Aufruf der **Thread**-Methode **Abort()**, siehe Abschnitt 15.1.5), wird **Exit()** nur aufgerufen, wenn `lockTaken` den Wert **true** besitzt.

Durch den **Exit()** - Aufruf im **finally**-Block ist sichergestellt, dass der kritische Block auch nach einem Ausnahmefehler verlassen wird. Mit dieser Vorbeugungsmaßnahme gegen eine Systemverklemmung wird allerdings das Risiko in Kauf genommen, dass andere Threads Zugang zu einem „Unfallort“ erhalten, an dem inkonsistente Verhältnisse bestehen (Griffiths 2013, S. 219).

---

[1] Dies ist einem Quellcode-Kommentar auf der folgenden Github-Webseite zu Roslyn zu entnehmen:
https://github.com/dotnet/roslyn/blob/56f605c41915317ccdb925f66974ee52282609e7/src/Compilers/CSharp/Portable/Lowering/LocalRewriter/LocalRewriter_LockStatement.cs

Um einer Blockade vorzubeugen, kann sich ein Thread mit der statischen **Monitor**-Methode **TryEnter()** um ein Sperrobjekt bewerben:

public static bool TryEnter(Object *obj***)**

Diese Methode endet auf jeden Fall sofort und informiert mit einem Rückgabewert vom Typ **bool** darüber, ob die Berechtigung erteilt worden ist. Es sind Überladungen mit einem Timeout-Parameter vorhanden, so dass die maximale Wartezeit auf die Sperre festgelegt werden kann.

Wenn auf die **lock**-Anweisung und die damit verbundene Compiler-Unterstützung verzichtet und direkt mit der **Monitor**-Klasse gearbeitet wird (per **Enter()** oder **TryEnter()**), dann erhöhen sich die Flexibilität und die Verantwortung des Programmierers, z.B. muss eine erworbene Sperre mit der statischen **Monitor**-Methode **Exit()** zurückgegeben werden.

15.1.4.1.2.2 Koordination per Wait() und Pulse()

Mit Hilfe der statischen **Monitor**-Methoden **Wait()** und **Pulse()** können in unserem betriebswirtschaftlichen Beispiel negative Lagerbestände verhindert werden. Trifft eine Konsumentenanfrage auf einen unzureichenden Lagerbestand, dann wird der zugehörige Thread mit der Methode **Wait()** in den Zustand **WaitSleepJoin** versetzt:

```
public bool Liefere(int sub) {
 lock (lockObject) {
  if (anz < MANZ) {
   while (bilanz < sub) {
      Console.WriteLine("!!!!!!! {0,10} muss warten: " +
       "Keine {1, 3} Einheiten vorhanden um {2} Uhr.",
       Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci));
      Monitor.Wait(lockObject);
   }
   bilanz -= sub;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
      anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Mit dem **Wait()** - Aufruf wird das exklusive Zutrittsrecht für den synchronisierten Block zurückgegeben, so dass im Beispiel der Produzenten-Thread zum Zug kommt und für Nachschub sorgen kann. Als Parameter ist im **Wait()** - Aufruf das synchronisierende Objekt anzugeben.

Um eine erfolgreiche Kooperation zu gewährleisten, muss der Produzenten-Thread nach jeder Lieferung die **Monitor**-Methode **Pulse()** aufrufen, um den Konsumenten-Thread zu reaktivieren, d.h. in den Zustand **Running** zu versetzen:

```
public bool Ergaenze(int add) {
 lock (lockObject) {
  if (anz < MANZ) {
   bilanz += add;
   anz++;
   Rumoren();
   Console.WriteLine("Nr. {0,2}: {1,10} ergänzt  {2,3} um {3} Uhr. Stand: {4}",
    anz, Thread.CurrentThread.Name, add, DateTime.Now.ToString("T", ci), bilanz);
   Monitor.Pulse(lockObject);
   return true;
  } else {
   Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
    ", es ist Feierabend!");
   return false;
  }
 }
}
```

Als Parameter wird im **Pulse()** - Aufruf das synchronisierende Objekt angegeben, das die Warteschlange verwaltet. Der reaktivierte Konsumenten-Thread bewirbt sich wieder um Prozessorzeit und Lagerzugangsberechtigung. Weil keinesfalls sicher ist, dass der Konsument nach der Reaktivierung einen ausreichenden Vorrat antrifft, findet sein **Wait()** - Aufruf in einer **while**-Schleife mit einleitender Bedingungsprüfung statt.

Nun produziert das Beispielprogramm nur noch realistische Lagerprotokolle, z.B.:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 02:26:27 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 02:26:27 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 02:26:29 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 02:26:29 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 02:26:32 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 02:26:33 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 02:26:34 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 02:26:35 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 02:26:38 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 02:26:38 Uhr. Stand: 75
!!!!!!!  Konsument muss warten: Keine  81 Einheiten vorhanden um 02:26:39 Uhr.
Nr. 11:  Produzent ergänzt    7 um 02:26:41 Uhr. Stand: 82
Nr. 12:  Konsument entnimmt  81 um 02:26:41 Uhr. Stand: 1
!!!!!!!  Konsument muss warten: Keine   5 Einheiten vorhanden um 02:26:42 Uhr.
Nr. 13:  Produzent ergänzt   37 um 02:26:43 Uhr. Stand: 38
Nr. 14:  Konsument entnimmt   5 um 02:26:43 Uhr. Stand: 33
!!!!!!!  Konsument muss warten: Keine  43 Einheiten vorhanden um 02:26:45 Uhr.
Nr. 15:  Produzent ergänzt   73 um 02:26:47 Uhr. Stand: 106
Nr. 16:  Konsument entnimmt  43 um 02:26:47 Uhr. Stand: 63
!!!!!!!  Konsument muss warten: Keine  75 Einheiten vorhanden um 02:26:48 Uhr.
Nr. 17:  Produzent ergänzt   33 um 02:26:50 Uhr. Stand: 96
Nr. 18:  Konsument entnimmt  75 um 02:26:50 Uhr. Stand: 21
!!!!!!!  Konsument muss warten: Keine  78 Einheiten vorhanden um 02:26:51 Uhr.
Nr. 19:  Produzent ergänzt   75 um 02:26:52 Uhr. Stand: 96
Nr. 20:  Konsument entnimmt  78 um 02:26:52 Uhr. Stand: 18

Lieber Konsument, es ist Feierabend!

Lieber Produzent, es ist Feierabend!
```

Zu jedem Sperrobjekt gehören zwei Warteschlangen:

- In der **Ready**-Schlange befinden sich arbeitswillige Threads, die auf das Sperrobjekt warten.
- In der **Wait**-Schlange befinden sich Threads, die auf eine Zustandsänderung angewiesen sind und danach das Sperrobjekt erwerben möchten. Ohne Benachrichtigung über eine Zustandsänderung verbleiben die wartenden Threads auch dann passiv, wenn das Sperrobjekt frei ist.

Die Methode **Pulse()** informiert aus der **Wait**-Schlange den Thread mit dem ältesten **Wait()** - Aufruf über eine Zustandsänderung und befördert ihn in die **Ready**-Schlange. Als Alternative ist die Methode **PulseAll()** verfügbar, die *alle* wartenden Threads in die **Ready**-Schlange befördert.

Zu **Wait()** sind Überladungen mit Timeout Parameter vorhanden. Wird ein wartender Thread nicht innerhalb der Timeout-Zeitspanne per **Pulse()** oder **PulseAll()** reaktiviert, wechselt er in den Zustand **Ready**, bewirbt sich also erneut um die Sperre.

Die **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** dürfen nur in einem synchronisierten Block aufgerufen werden.

15.1.4.1.3 SpinLock

Versucht ein Thread vergeblich, über die **Monitor**-Methode **Enter()** ein Sperrobjekt zu erwerben (entweder durch den direkten Aufruf der Methode oder per **lock**-Anweisung), dann wird er blockiert. Stattdessen könnte der Thread aktiv bleiben und sich in einer Schleife wiederholt um den Zugang zum geschützten Bereich bemühen. Diese Technik kann prinzipiell aufgrund ihrer Einfachheit überlegen sein, wenn mit sehr kurzen Wartezeiten zu rechnen ist, weil im geschützten Bereich nur sehr einfache Operationen stattfinden (z.B. ein Lese- oder Schreibzugriff auf eine **decimal**-Variable). Die Verwaltung des exklusiven Zugangsrechts per Warteschleife statt Blockade wird durch die Struktur **SpinLock** im Namensraum **System.Threading** unterstützt.

Allerdings ist es mir nicht gelungen, ein Anwendungsbeispiel für den potentiellen Einspareffekt der Warteschleifentechnik zu konstruieren.[1] Weil die Verwendung der **SpinLock**-Struktur (im Unterschied zur **lock**-Anweisung) nicht direkt durch den Compiler unterstützt wird, muss der Programmierer außerdem einen Mehraufwand betreiben. Folglich lohnt sich die Verwendung der **SpinLock**-Sperre wohl kaum.

15.1.4.1.4 ReaderWriterLockSlim

Anders als bei einer Sperre per **Monitor** oder **SpinLock** bietet eine Sperre mit Hilfe der Klasse **ReaderWriterLockSlim** die Option, lese- und schreibwillige Threads unterschiedlich zu behandeln:

- Mehrere lesende Threads können sich simultan im geschützten Bereich aufhalten.
- Ein schreibender Thread benötigt exklusiven Zutritt.

Ein Einsatz der Klasse **ReaderWriterLockSlim** verspricht ein höheres Maß an Parallelität, doch führen die komplexen Spielregeln für die Vergabe von Zutrittsrechten[2] zu einem hohen Aufwand der CLR bei der Synchronisation. Folglich dauert der Erwerb einer Zugangsberechtigung länger als bei der Klasse **Monitor**, und vor einer Verwendung der Klasse **ReaderWriterLockSlim** sollten vergleichende Leistungsmessungen durchgeführt werden (Griffiths 2013, S. 623).

Am ehesten ist ein Nutzen der Klasse **ReaderWriterLockSlim** zu erwarten, wenn die zu synchronisierenden Threads auf die gemeinsame Ressource meist lesend und selten schreibend zugreifen:

---

[1] Auch das Demonstrationsprogramm auf der folgenden MSDN-Seite liefert je nach gewählter Schleifenobergrenze `N` keine Überlegenheit der **SpinLock**-Sperre gegenüber der **lock**-Anweisung:
https://msdn.microsoft.com/de-de/library/dd460716(v=vs.110).aspx

[2] Die Beschreibung der Klasse **ReaderWriterLockSlim** auf der folgenden MSDN-Seite erfordert eine sehr sorgfältige Lektüre:
https://msdn.microsoft.com/de-de/library/system.threading.readerwriterlockslim(v=vs.110).aspx

- Nachdem ein schreibwilliger Thread die Methode **EnterWriteLock()** oder die mit einem Timeout-Parameter ausgestattetet Alternative **TryEnterWriteLock()** erfolgreich aufgerufen hat, ist kein weiterer Zugriff möglich, bis die Schreibberechtigung mit einem Aufruf der Methode **ExitWriteLock()** zurückgegeben wird.
- Nachdem ein lesewilliger Thread die Methode **EnterReadLock()** oder die mit einem Timeout-Parameter ausgestattetet Alternative **TryEnterReadLock()** erfolgreich aufgerufen hat, ist kein Schreibzugriff möglich, während auch andere Threads eine Leseberechtigung erwerben können. Hat jeder Thread mit Leseberechtigung die Methode **ExitReadLock()** aufgerufen, kann wieder eine Schreibberechtigung erworben werden.

Im folgenden Beispiel greift *ein* Thread schreibend auf eine Ressource (einen **decimal**-Wert) zu, während 3 andere Threads nur lesend zugreifen:

```
using System;
using System.Threading;

class ReaderWriterLockSlimDemo {
  static Thread rt;
  static decimal dwert = 13.0M;
  static ReaderWriterLockSlim rwls = new ReaderWriterLockSlim();

  static void WM() {
    for (int i = 1; i <= 3; i++) {
      rwls.EnterWriteLock();
        dwert++;
        Console.WriteLine("\n*** Schreibzugriff " + i + ". Neuer Wert: " + dwert +
          ", aktive Leser: " + rwls.CurrentReadCount + " (" + DateTime.Now.ToString()+")");
      rwls.ExitWriteLock();
      Thread.Sleep(3000);
    }
  }

  static void RM() {
    for (int i = 1; i <= 3; i++) {
      rwls.EnterReadLock();
        Console.WriteLine(Thread.CurrentThread.Name + " ermittelt Wert " + dwert +
          ", aktive Leser: " + rwls.CurrentReadCount + " (" + DateTime.Now.ToString()+")");
      rwls.ExitReadLock();
      Thread.Sleep(3000);
    }
  }

  static void Main() {
    new Thread(WM).Start();
    for (int i = 0; i < 3; i++) {
      rt = new Thread(RM);
      rt.Name = "RT" + Convert.ToString(i);
      rt.Start();
    }
  }
}
```

Wie das folgende Ablaufprotokoll zeigt, behindern sich die 3 lesenden Threads nicht gegenseitig. Sie müssen aber warten, während der schreibende Thread zugreift:

```
*** Schreibzugriff 1. Neuer Wert: 14,0, aktive Leser: 0 (01.05.2017 12:30:06)
RT0 ermittelt Wert 14,0, aktive Leser: 1 (01.05.2017 12:30:06)
RT1 ermittelt Wert 14,0, aktive Leser: 1 (01.05.2017 12:30:06)
RT2 ermittelt Wert 14,0, aktive Leser: 2 (01.05.2017 12:30:06)

*** Schreibzugriff 2. Neuer Wert: 15,0, aktive Leser: 0 (01.05.2017 12:30:09)
RT0 ermittelt Wert 15,0, aktive Leser: 1 (01.05.2017 12:30:09)
RT1 ermittelt Wert 15,0, aktive Leser: 1 (01.05.2017 12:30:09)
RT2 ermittelt Wert 15,0, aktive Leser: 2 (01.05.2017 12:30:09)

*** Schreibzugriff 3. Neuer Wert: 16,0, aktive Leser: 0 (01.05.2017 12:30:12)
RT0 ermittelt Wert 16,0, aktive Leser: 1 (01.05.2017 12:30:12)
RT2 ermittelt Wert 16,0, aktive Leser: 1 (01.05.2017 12:30:12)
RT1 ermittelt Wert 16,0, aktive Leser: 2 (01.05.2017 12:30:12)
```

Die aktuelle Zahl der mit Leserecht versorgten Threads kann über die **ReaderWriterLockSlim**-Eigenschaft **CurrentReadCount** ermittelt werden.

Wenn in der Regel der Lesezugriff genügt, unter bestimmten Umständen aber doch geschrieben werden muss, kann mit **EnterUpgradeableReadLock()** bzw. **TryEnterUpgradeableReadLock()** eine Sperre erworben werden, die sich zum Schreibzugriff auf- oder zum Lesezugriff abwerten lässt.

Auf keinen Fall sollte die ältere Klasse **ReaderWriterLock** verwendet werden, die nur noch aus Kompatibilitätsgründen vorhanden ist. Sie ist bekannt für eine schlechte Performanz und eine hohe Deadlock-Neigung.[1]

15.1.4.1.5 Mutex und Semaphore

Die Klasse **Mutex** (engl. *mutual exclusion*) erbringt ähnliche Leistungen wie die Klasse **Monitor** und erlaubt auch eine Thread-Synchronisation über Prozessgrenzen hinweg (z.B. zwischen verschiedenen Programmen). Diese Flexibilität ist allerdings mit einem erheblichen Zeitaufwand verbunden, was speziell bei häufigem Thread-Wechsel von Bedeutung ist.

Über die Klasse **Semaphore** kann die Zutrittsexklusivität insofern relativiert werden, dass nicht nur *ein* Thread den geschützten Bereich betreten darf, sondern eine im Konstruktor festgelegte Anzahl. Wird der **WaitOne()** - Antrag eines Threads positiv entschieden, reduziert sich die Zahl der verbleibenden „Eintrittskarten". Trifft ein **WaitOne()** - Antrag auf den Zählerstand 0, wird der anfragende Thread blockiert. Per **Release()** kann ein Thread seine Eintrittskarte zurückgegen. Wie die Klasse **Mutex** kann auch die Klasse **Semaphore** prozessübergreifend operieren.

15.1.4.2 Atomare Operationen

In Abschnitt 15.1.4.1.1 wurde erläutert, dass auch eine einfache Operation wie das Inkrementieren einer Ganzzahlvariablen *nicht atomar* ist, also durch Thread-Wechsel unterbrochen werden kann, wobei es zu irregulären Variableninhalten und damit zu einem sehr schwer aufzuklärendem Programmfehler kommen kann. Durch statische Methoden der Klasse **Interlocked** im Namensraum **System.Threading** erhält man atomare Varianten für diverse Operationen, z.B.:

- **public static long Add(ref long** *location***, long** *value***)**
  Zur Variablen in *location* wird der Wert des zweiten Parameters addiert.
- **public static long Increment(ref long** *location***)**
  Die Variable in *location* wird inkrementiert.

[1] Beim Deadlock blockieren sich Threads gegenseitig (siehe Abschnitt 15.1.7).

Eine solche Sicherung verursacht geringere Kosten als eine **lock**-Anweisung. Sie wird oft als *lock-free* bezeichnet, wobei die Bezeichnung *low-lock* zutreffender ist. Morrison (2005b) nennt zwei Nachteile der low-lock - Synchronisation:

- Schmaler Anwendungsbereich
- Erhebliches Fehlerrisiko
  Es sind Details im Speichersystem und Compiler zu beachten, die normalerweise für Programmierer irrelevant sind.

Insgesamt empfiehlt Morrison (wie auch Griffiths 2013, S. 632), die low-lock - Techniken erst dann einzusetzen, wenn einfachere und damit unproblematischere Synchronisationstechniken wie die **lock**-Anweisung zu hohe Kosten verursachen.

Im Zusammenhang mit der Klasse **Interlocked** wird oft der für (statische) Felder erlaubte Modifikator **volatile** diskutiert, z.B.:

```
static volatile int ivar;
```

Er sorgt dafür, dass der Compiler bei Mehrkernprozessoren darauf verzichtet, CPU-lokalen Cache-Speicher zur Leistungssteigerung einzusetzen. Bei so ausgezeichneten Variablen sehen alle Threads zu einem Zeitpunkt denselben Speicherinhalt. Eine Thread-Synchronisation wird jedoch im Allgemeinen *nicht* erreicht. Es kann z.B. weiterhin passieren, dass ein Thread beim Inkrementieren einer Variablen zwischen dem Lesen des alten und dem Schreiben des neuen Werts unterbrochen wird, sodass schlussendlich ein falscher Wert im Speicher landet. Nur unter sehr speziellen Umständen kann der Modifikator **volatile** Thread-Kommunikationsfehler verhindern, z.B.:

- Auf eine Variable greift nur *ein* Thread schreibend zu.
- Zugriffe auf diese Variable erfolgen atomar.

15.1.4.3 Signalisierungsobjekte

Mit dem **Monitor**-Methoden **Wait()**, **Pulse()** und **PulseAll()** lässt sich ein koordiniertes Agieren und eine Kommunikation von Threads realisieren, doch bestehen zwei wesentliche Einschränkungen:

- Die Methoden dürfen nur in einem synchronisierten Block aufgerufen werden.
- Ein **Pulse()** - oder **PulseAll()** - Aufrufe verhallt ohne Effekt, wenn gerade kein Thread auf diese Botschaft wartet.

Anschließend wird beschrieben, wie ein Thread an einer beliebigen Stelle ein mehr oder weniger dauerhaftes Signal für andere Threads setzen kann.

Über ein Signalisierungsobjekt der Klasse **AutoResetEvent** kann sich ein Thread in den einstweiligen Ruhezustand versetzen, bis ein anderer Thread das Signal „auf Grün“ stellt. Beide Threads sprechen dasselbe **AutoResetEvent**-Objekt an, wobei der erste Thread seinen Weckauftrag über die Methode **WaitOne()** erteilt und der zweite Thread über die Methode **Set()** einen wartenden Thread reaktiviert. Wie bei einer Drehkreuztür mit kostenpflichtigem Einzeldurchlass kann pro **Set()** - Aufruf nur *ein* Thread reaktiviert werden, wobei das gesetzte Signal automatisch verbraucht wird (Auto-Reset, siehe Klassenname).

Soll stattdessen bei der Reaktivierung eine Tür für *alle* auf das Signal wartenden Threads geöffnet werden, verwendet man statt der Klasse **AutoResetEvent**[1] das Gegenstück **ManualResetEvent**.

---

[1] Bei den Klassen **AutoResetEvent** und **ManualResetEvent** hat der Namensbestandteil *Event* keinen Bezug zu .NET - Ereignissen im Sinne von Kapitel 9. Hintergrund der Benennung ist ein Bezug zu der im Win32-API für die Thread- Synchronisation verfügbaren Funktion **CreateEvent()**.

Bei einem Signalisierungsobjekt dieser Klasse bleibt ein gesetztes Signal erhalten, bis es über die Methode **Reset()** storniert wird.

Im folgenden Programm wird über ein **AutoResetEvent**-Objekt dafür gesorgt, dass zwei Threads streng alternierend tätig werden:

```
using System;
using System.Threading;

class AutoResetEventDemo {
 static Thread t1, t2;
 static AutoResetEvent are = new AutoResetEvent(false); // Signal ist initial aus.

 static void M() {
  int t = 1;
  if (Thread.CurrentThread == t2) {
   t = 2;
   are.WaitOne(); // t2 geht als erster in den Wartezustand.
  }
  for (int j = 1; j <= 2; j++) {
   Console.Write("\n" + Thread.CurrentThread.Name + " ist am Zug: ");
   for (int i = 1; i <= 5; i++) {
    Console.Write(t + " ");
    Thread.Sleep(200);
   }
   Console.WriteLine("\n" + Thread.CurrentThread.Name +
                     " Setzt das Signal und dann sich selbst zur Ruhe.");
   are.Set();
   Thread.Sleep(100); // Nötig, damit der andere Thread das Signal verbraucht.
   are.WaitOne();
  }
  Console.WriteLine("\n"+Thread.CurrentThread.Name+" endet.");
  are.Set(); // Nötig, damit der wartende Kollege zum letzten Auftritt geweckt wird.
 }

 static void Main() {
  t1 = new Thread(M); t2 = new Thread(M);
  t1.Name = "t1";
  t2.Name = "t2";
  t1.Start();
  t2.Start();
 }
}
```

Ein Programmlauf produziert folgende Ausgaben:

```
t1 ist am Zug: 1 1 1 1 1
t1 Setzt das Signal und dann sich selbst zur Ruhe.

t2 ist am Zug: 2 2 2 2 2
t2 Setzt das Signal und dann sich selbst zur Ruhe.

t1 ist am Zug: 1 1 1 1 1
t1 Setzt das Signal und dann sich selbst zur Ruhe.

t2 ist am Zug: 2 2 2 2 2
t2 Setzt das Signal und dann sich selbst zur Ruhe.

t1 endet.

t2 endet.
```

Per Konstruktorparameter wird ein **AutoResetEvent** - Objekt mit initial abgeschalteten Signal erzeugt:

```
static AutoResetEvent are = new AutoResetEvent(false);
```

Hat der momentan aktive Thread eine Etappe bewältigt, setzt er das Freigabesignal und wartet ein Weilchen, damit das Signal vom wartenden Kollegen verbraucht wird. Dann stellt er sich selbst wieder hinter der Drehtür an:

```
are.Set();
Thread.Sleep(100); // Nötig, damit der andere Thread das Signal verbraucht.
are.WaitOne();
```

Wenn mehrere Threads aufgrund eines **WaitOne()** - Aufrufs vor einer Drehkreuztür vom Typ **AutoResetEvent** warten, kann beim Setzen des Signals genau *einer* passieren, wobei das Signal automatisch abgeschaltet wird. Während ein Aufruf der **Monitor**-Methode **Pulse()** ungehört und effektfrei verhallen kann, weil gerade kein Thread auf das zugehörige Sperrobjekt wartet, ermöglicht ein Aufruf der **AutoResetEvent**-Methode **Set()** unabhängig von der aktuellen Nachfragesituation bei der überwachten Drehtür eine Einzelpasssage. Ist das Signal bereits gesetzt, bleibt ein weiterer **Set()** - Aufruf allerdings folgenlos.

In der gemeinsamen Basisklasse **EventWaitHandle** von **AutoResetEvent** und **ManualResetEvent** sind die statischen Methoden **WaitAny()** und **WaitAll()** verfügbar. In einem Aufruf dieser Methoden kann ein Thread einen ganzen Array mit Signalisierungsobjekten angeben, um geweckt zu werden, wenn *irgendein* Signal gesetzt worden ist bzw. wenn *alle* Signale gesetzt worden sind. Allerdings wird die Methode **WaitAll()** nicht unterstützt, wenn ein Thread im **STAThread** - Modus läuft. Das gilt z.B. für die Threads im Rahmen einer WPF-Anwendung, weil deren **Main()** - Methode mit dem **STAThreadAttribute** dekoriert werden muss (siehe Abschnitt 11.2.1).

Die Klassen **AutoResetEvent** und **ManualResetEvent** erlauben eine Thread-Koordination über Prozessgrenzen hinweg. Mit dieser (in der Regel nicht benötigten) Option ist ein nennenswerter Zeitaufwand verbunden, sodass ein Einsatz der **EventWaitHandle**-Ableitungen beim kurz getakteten Start-Stopp - Betrieb *nicht* ratsam ist.

Seit der .NET - Version 4.0 bietet die Klasse **ManualResetEventSlim**, die von **System.Object** abstammt, eine zur Klasse **ManualResetEvent** analoge Funktionalität und arbeitet dabei aus zwei Gründen etwas flotter:

- Bevor ein Anwärter-Thread blockiert, versucht er mehrmals sein Glück (analog zur Struktur **SpinLock**, siehe Abschnitt 15.1.4.1.3), sodass gelegentlich der relativ zeitaufwendige Kontakt mit der Thread-Verwaltung des Betriebssystems eingespart wird. Nach einer begrenzten Zahl von gescheiterten Anträgen, begibt sich ein **ManualResetEventSlim**-Objekt aber doch in den blockierten Zustand.
- Es wird auf die Option zur Überschreitung von Prozessgrenzen verzichtet. Wo diese Option benötigt wird, ist weiterhin die Klasse **ManualResetEvent** zu verwenden.

Zur Klasse **AutoResetEvent** existiert *keine* Slim-Variante.

Über ein Objekt der Klasse **CountdownEvent**, kann sich ein Thread reaktivieren lassen, wenn ein Signal *k* mal gesetzt worden ist (siehe Übungsaufgabe in Abschnitt □).

15.1.4.4 Einfache Verfahren zur Thread-Koordination

In diesem Abschnitt werden zwei einfache Verfahren beschrieben, die es einem Thread erlauben, auf eine Bedingung zu warten:

- Auf den Ablauf einer Wartezeit
- Auf die Beendigung eines anderen Threads

15.1.4.4.1 Ein Schläfchen in Ehren

In diesem Zusammenhang kann man die schon mehrfach benutzte statische **Thread**-Methode **Sleep()** noch einmal erwähnen, mit der sich der aufrufende Thread für eine per Parameter bestimmte Anzahl von Millisekunden in den Zustand **WaitSleepJoin** versetzt und den restlichen Threads die CPU-Leistung überlässt, z.B.:

```
Thread.Sleep(1000 + rand.Next(3000));
```

15.1.4.4.2 Weck mich, wenn Du fertig bist

Will ein Thread `t1` in den Wartezustand wechseln, bis der Thread `t2` seine Tätigkeit beendet hat, kann er diesen Plan durch einen Aufruf der **Thread**-Methode **Join()** realisieren, z.B.:

Quellcode	Ausgabe

```
using System;
using System.Threading;

class JoinDemo {
   static Thread t1, t2;

   static void M1() {
      Console.WriteLine("t1 in M1");
      Thread.Sleep(100);
      Console.WriteLine("\nt1 beginnt seine Arbeit:");
      for (int i = 1; i <= 3; i++) {
         Console.Write(1 + " ");
         Thread.Sleep(500);
      }
      Console.WriteLine("\nt1 endet");
   }

   static void M2() {
      Console.WriteLine("t2 in M2, wartet auf t1");
      t1.Join();
      Console.WriteLine("\nt2 beginnt seine Arbeit:");
      for (int i = 1; i <= 3; i++) {
         Console.Write(2 + " ");
         Thread.Sleep(500);
      }
      Console.WriteLine("\nt2 beenden mit Enter");
      Console.Read();
   }

   static void Main() {
      t1 = new Thread(new ThreadStart(M1));
      t2 = new Thread(new ThreadStart(M2));
      t1.Start(); t2.Start();
   }
}
```

```
t1 in M1
t2 in M2, wartet auf t1

t1 beginnt seine Arbeit:
1 1 1
t1 endet

t2 beginnt seine Arbeit:
2 2 2
t2 beenden mit Enter
```

15.1.5 Threads stoppen

Ein Thread endet „auf natürlichem Weise“ mit der zugrunde liegenden Startmethode. Um ihn früher zu stoppen, kann man die **Thread**-Methode **Abort()** aufrufen. Diese befördert den Thread in den Zustand **AbortRequested** und löst eine **ThreadAbortException** aus, so dass die betroffenen Methoden per Ausnahmebehandlung für einen sinnvollen Abgang sorgen können. Am Ende eines entsprechenden **catch**-Blocks wird die Ausnahme automatisch erneut ausgelöst, sofern dies nicht über einen Aufruf der Methode **ResetAbort()** verhindert wird (siehe unten).

Wir betrachten zunächst den Fall, dass **Abort()** im betroffenen Thread aufgerufen wird. Als Beispiel soll eine kundenunfreundliche Variante des Produzent-Lager-Konsument - Programms dienen: Der Lagerverwalter terminiert den Konsumenten-Thread, sobald dieser mit einem Wunsch über den Lagerbestand hinausgeht:

```
public bool Liefere(int sub) {
      lock (lockObject) {
   if (anz < MANZ) {
    if (bilanz < sub) {
     Console.WriteLine("!!!!!!! {1,10} fordert  {2, 3}" +
      " um {3} Uhr und wird abgewiesen.",
      anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci));
     Thread.CurrentThread.Abort();
    }
    anz++;
    bilanz -= sub;
    Rumoren();
    Console.WriteLine("Nr. {0,2}: {1,10} entnimmt {2, 3} um {3} Uhr. Stand: {4}",
     anz, Thread.CurrentThread.Name, sub, DateTime.Now.ToString("T", ci), bilanz);
    return true;
   } else {
    Console.WriteLine("\nLieber " + Thread.CurrentThread.Name +
     ", es ist Feierabend!");
    return false;
   }
  }
}
```

Der Konsument nutzt die Ausnahmebehandlung für eine Beschwerde:

```
public void Run() {
   Random rand = new Random(2);
   try {
      do {
         offen = pl.Liefere((5 + rand.Next(100)));
         Thread.Sleep(1000 + rand.Next(3000));
      } while (offen);
   } catch (ThreadAbortException) {
      Console.WriteLine("Als Kunde muss ich mir so etwas " +
         "nicht gefallen lassen!");
   }
}
```

Nach der Ausnahmebehandlung gelangt der Konsumenten-Thread in der Zustand **Stopped**, und im Lager taucht nur noch der Produzent auf:

```
Das Lager ist offen (Bestand: 100)

Nr.  1:  Produzent ergänzt   29 um 03:22:27 Uhr. Stand: 129
Nr.  2:  Konsument entnimmt  82 um 03:22:27 Uhr. Stand: 47
Nr.  3:  Produzent ergänzt   51 um 03:22:29 Uhr. Stand: 98
Nr.  4:  Konsument entnimmt  21 um 03:22:30 Uhr. Stand: 77
Nr.  5:  Produzent ergänzt   70 um 03:22:32 Uhr. Stand: 147
Nr.  6:  Konsument entnimmt  15 um 03:22:34 Uhr. Stand: 132
Nr.  7:  Produzent ergänzt   40 um 03:22:34 Uhr. Stand: 172
Nr.  8:  Konsument entnimmt  85 um 03:22:36 Uhr. Stand: 87
Nr.  9:  Konsument entnimmt  27 um 03:22:38 Uhr. Stand: 60
Nr. 10:  Produzent ergänzt   15 um 03:22:38 Uhr. Stand: 75
!!  Konsument fordert   81 um 03:22:39 Uhr und wird ausgeschlossen (Abbruch des Threads).
Als Kunde muss ich mir so etwas nicht gefallen lassen!
Nr. 11:  Produzent ergänzt    7 um 03:22:41 Uhr. Stand: 82
Nr. 12:  Produzent ergänzt   37 um 03:22:43 Uhr. Stand: 119
Nr. 13:  Produzent ergänzt   73 um 03:22:47 Uhr. Stand: 192
Nr. 14:  Produzent ergänzt   33 um 03:22:50 Uhr. Stand: 225
Nr. 15:  Produzent ergänzt   75 um 03:22:53 Uhr. Stand: 300
Nr. 16:  Produzent ergänzt   99 um 03:22:56 Uhr. Stand: 399
Nr. 17:  Produzent ergänzt   21 um 03:22:57 Uhr. Stand: 420
Nr. 18:  Produzent ergänzt   84 um 03:22:59 Uhr. Stand: 504
Nr. 19:  Produzent ergänzt   84 um 03:23:01 Uhr. Stand: 588
Nr. 20:  Produzent ergänzt   87 um 03:23:03 Uhr. Stand: 675

Lieber Produzent, es ist Feierabend!
```

Bevor ein abgebrochener Thread in den Zustand **Stopped** gelangt, werden natürlich alle beteiligten **finally**-Blöcke ausgeführt, was den Abbruch beliebig lange hinauszögern kann.

Ein Thread im Zustand **AbortRequested** kann durchaus in den Zustand **Running** zurückkehren. Dazu muss lediglich in einer **ThreadAbortException**-Ausnahmebehandlung die **Thread**-Methode **ResetAbort()** aufrufen werden, was in der folgenden Variante unseres Beispiels der Konsument tut:

```
public void Run() {
   Random rand = new Random(2);
   do {
      try {
         offen = pl.Liefere((5 + rand.Next(100)));
      } catch (ThreadAbortException) {
         Console.WriteLine("Als Kunde muss ich mir so etwas " +
            "nicht gefallen lassen!");
         Thread.ResetAbort();
      }
      Thread.Sleep(1000 + rand.Next(3000));
   } while (offen);
}
```

Wenn ein Thread sich selbst unterbricht, ist der Programmablauf unter Kontrolle und vorhersagbar. Wenn einem Thread hingegen von einem *anderen* Thread eine **ThreadAbortException** - Ausnahme aufgezwungen wird, sind der Unterbrechungspunkt und die möglichen Schäden unkalkulierbar. Außerdem ist nicht garantiert, dass die **ThreadAbortException** tatsächlich zur Beendigung des Threads führt:

- Der Thread kann die Methode **ResetAbort()** aufrufen
- oder in einem **finally**-Block eine lang dauernde Aktivität entfalten.

Statt einen „fremden“ Thread per **Abort()** zu beenden, sollte man ihm besser eine vorher vereinbarte Nachricht zustellen (z.B. durch das Setzen einer Eigenschaftsausprägung). Generell sollte man

einen Thread nur dann starten, wenn seine kooperative Reaktion auf ein Terminierungssignal sichergestellt ist.[1]

Im folgenden Beispiel ruft die **Run()** - Methode einer Klasse namens `Runner` eine private Methode namens `Write()` auf, die sich für einen Ausgabeauftrag mehrere Sekunden Zeit nimmt, so dass eine Unterbrechung gut sichtbar ist:

```
using System;
using System.Threading;

public class Runner {
   public bool Stopped { get; set; }

   void Write(String s) {
      for (int i = 0; i < s.Length; i++) {
         Console.Write(s[i]);
         Thread.Sleep(100);
      }
      Console.WriteLine();
   }

   public void Run() {
      try {
         for (int i = 0; i < 5; i++) {
            Write("Rumoren im Runner, i = " + i);
            Thread.Sleep(1000);
            if (Stopped) {
               Console.WriteLine(" Tschüss!");
               break;
            }
         }
      } catch (ThreadAbortException) {
            Console.WriteLine(" Autsch!");
      }
   }
}
```

In der **Main()** - Methode der folgenden Klasse

```
using System.Threading;
public class Prog {
   static void Main() {
      Runner runner = new Runner();
      Thread kt = new Thread(runner.Run);
      kt.Start();
      Thread.Sleep(5000);
      kt.Abort();
   }
}
```

wird ein Thread auf `Run()` - Basis gestartet und dann von außen per **Abort()** unterbrochen, was in der Regel eine defekte Ausgabe zur Folge hat:

```
Rumoren im Runner, i = 0
Rumoren im Runner, i = 1
Rumoren im Runne  Autsch!
```

[1] Stack Overflow - Beitrag von Eric Lippert (2010): http://stackoverflow.com/questions/2251964/c-sharp-thread-termination-and-thread-abort

Wird die von der `Runner`-Klasse angebotene Eigenschaft `Stopped`

```
runner.Stopped = true;
```

gesetzt, statt die Methode **Abort()** aufzurufen, dann macht der kooperative `Runner` bei der nächsten passenden Gelegenheit einen geordneten Abgang:

```
Rumoren im Runner, i = 0
Rumoren im Runner, i = 1
 Tschüss!
```

15.1.6 Thread-Lebensläufe

In diesem Abschnitt wird zunächst die Vergabe von Arbeitsberechtigungen für konkurrierende Threads durch die Laufzeitumgebung behandelt. Dann fassen wir unsere Kenntnisse über die verschiedenen Zustände eines Threads und über Anlässe für Zustandswechsel zusammen.

15.1.6.1 Scheduling und Prioritäten

Die Zuweisung von Rechenzeit auf den real oder logisch vorhandenen CPU-Kernen an die Threads im Zustand **Running** überlässt die CLR dem Betriebssystem, wo für diese Aufgabe der so genannte **Scheduler** zuständig ist.

Er orientiert sich u.a. an den **Prioritäten** der Threads, die sich über ihre **Priority**-Eigenschaft ermitteln und verändern lassen. Erlaubt sind die folgenden Werte des Enumerationstyps **ThreadPriority**:

- **Highest**
- **AboveNormal**
- **Normal**
- **BelowNormal**
- **Lowest**

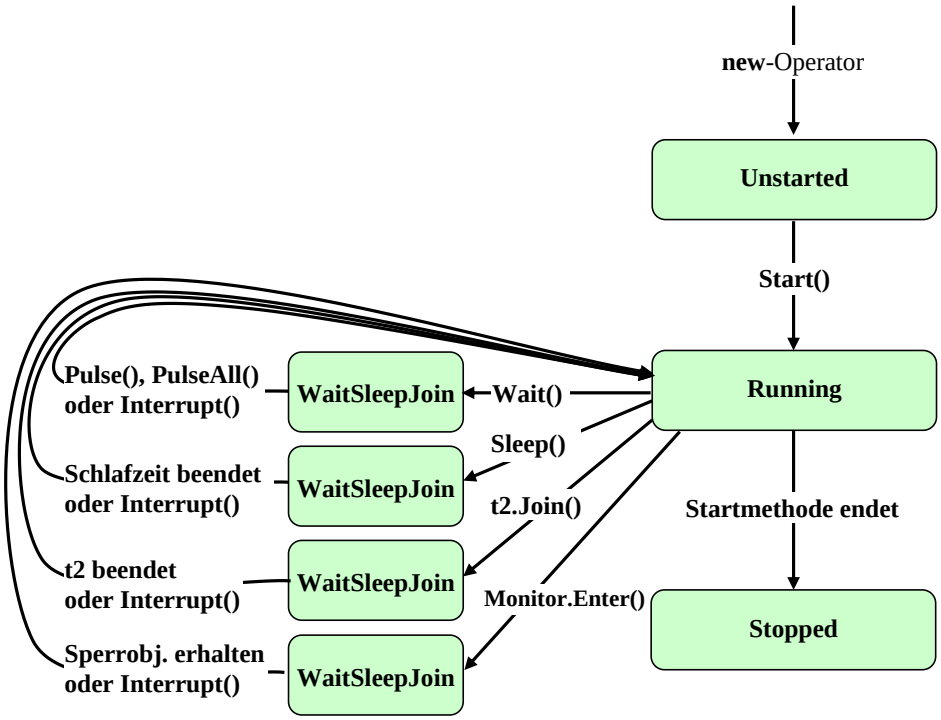
Per Voreinstellung haben Threads die Priorität **Normal**.

Der Scheduler bevorzugt Threads mit höherer Priorität in einem *strengen* Sinn und verwendet bei gleicher Priorität ein **preemtives Zeitscheibenverfahren**. Auf einem *Einprozessor*-System resultiert folgendes Verhalten:

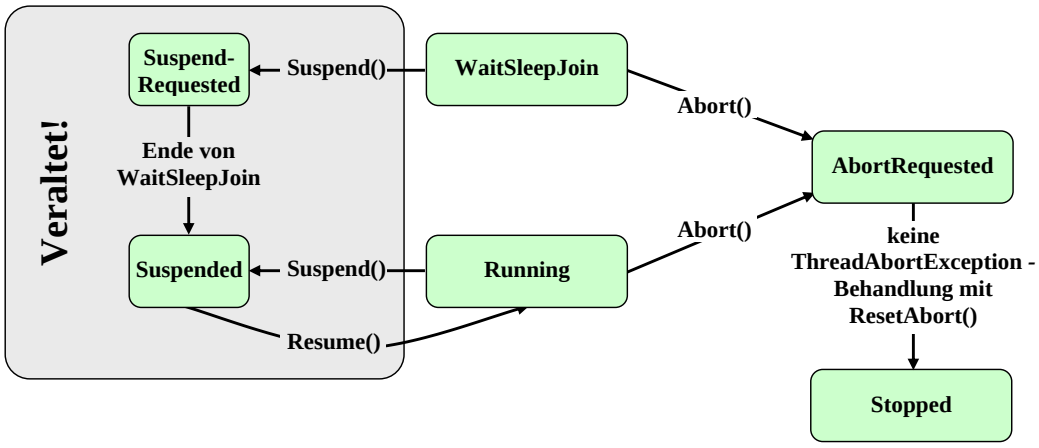
- Ein Thread kann nur dann Zugang zum Prozessor erhalten, wenn *kein* Thread mit höherer Priorität arbeitswillig ist. Ein *Verhungern* (engl. *starvation*) von Threads mit niedriger Priorität, die permanent den Kürzeren ziehen, wird also *nicht* verhindert.
- Die Threads gleicher Priorität werden reihum (*Round-Robin*) jeweils für eine festgelegte Zeitspanne ausgeführt.

15.1.6.2 Zustände von Threads

In der folgenden Abbildung werden wichtige Thread-Zustände (bezeichnet durch Werte der Enumeration **ThreadState**) und Anlässe für Zustandsübergänge dargestellt:



Der Übersichtlichkeit halber folgt eine separate Abbildung für das Abbrechen bzw. Unterbrechen eines Threads:



Einige in den Abbildungen enthaltene **Thread**-Instanzmethoden wurden bisher noch nicht behandelt:

- **Interrupt()**
  Diese Instanzmethode dient dazu, einen Thread vom Zustand **WaitSleepJoin** in den Zustand **Running** zu versetzen. Dazu wird dem angesprochenen Thread eine **ThreadInterrupted-Exception** zugeworfen, wenn er sich (jetzt oder später) im **WaitSleepJoin**-Zustand befindet. Ein **Interrupt()** - Aufruf kann also auch prophylaktisch erfolgen. Begibt sich der angesprochene Thread nie in den Zustand **WaitSleepJoin**, bleibt ein **Interrupt()** - Aufruf ohne Folgen. Um auf eine **ThreadInterruptedException** vorbereitet zu sein, müssen **WaitSleepJoin**-einleitende Methoden in einer **try-catch** - Anweisung aufgerufen werden, z.B.:

```
try {
   Console.WriteLine("T1 möchte 10 Minuten schlafen.");
   Thread.Sleep(600000);
} catch {
   Console.WriteLine("T1 beim Schlafen gestört");
}
```

- **Suspend()**, **Resume()**
  Mit diesem als *veraltet* (engl.: *deprecated*) eingestuften Methoden kann man einen Thread anhalten bzw. wieder starten. Die Methoden sind in Misskredit geraten, weil ihr Einsatz zu Deadlock-Situationen (siehe Abschnitt 15.1.7) führen kann.

Weitere Hinweise:

- **Running** ist ein arbeitswilliger Thread auch dann, wenn er gerade auf die Zuteilung eines Prozessors durch das Betriebssystem wartet.
- Gemäß der FCL-Dokumentation[1] zur Enumeration **ThreadState** ist ein Thread auch dann im Zustand **WaitSleepJoin**, wenn er auf ein Signalisierungsobjekt (z.B. aus der Klasse **AutoResetEvent**) wartet (siehe Abschnitt 15.1.4.3).
- Einige Thread-Zustände können *gleichzeitig* bestehen. Erhält z.B. ein Thread im Zustand **WaitSleepJoin** einen **Abort()** - Aufruf, bestehen simultan die Zustände **WaitSleepJoin** und **AbortRequested** bis der Thread den Zustand **WaitSleepJoin** verlässt und dann mit der **ThreadAbortException** konfrontiert wird.

15.1.7 Deadlock

Wer sich beim Einsatz von Sperrobjekten zur Thread-Synchronisation ungeschickt anstellt, kann einen genannten *Deadlock* produzieren, wobei sich Threads gegenseitig blockieren. Im folgenden Beispiel begeben sich die Threads mit den Namen `T1` und `T2` jeweils in einen exklusiven Block, der durch die Objekte `lock1` bzw. `lock2` geschützt ist:

```
using System;
using System.Threading;

class DeadLock {
   static object lock1 = new object();
   static object lock2 = new object();

   static void M1() {
      lock (lock1) {
         Console.WriteLine(Thread.CurrentThread.Name+" in M1");
         Thread.Sleep(100);
         Console.WriteLine(Thread.CurrentThread.Name+" möchte M2 aufrufen");
         M2();
      }
   }

   static void M2() {
      lock (lock2) {
         Console.WriteLine(Thread.CurrentThread.Name+" in M2");
         Thread.Sleep(100);
         Console.WriteLine(Thread.CurrentThread.Name+" möchte M1 aufrufen");
         M1();
      }
   }

   static void Main() {
      Thread t1 = new Thread(new ThreadStart(M1));
      Thread t2 = new Thread(new ThreadStart(M2));
      t1.Name="T1"; t1.Start();
      t2.Name="T2"; t2.Start();
   }
}
```

[1] http://msdn.microsoft.com/de-de/library/system.threading.threadstate.aspx

Durch kurze Schläfchen ist dafür gesorgt, dass beide Threads ihren „eigenen“ synchronisierten Block ungestört betreten können. Anschließend versuchen beide, in den jeweils anderen Block zu gelangen, und das Programm hängt fest:

Das vorgestellte Problem hat folgende Struktur:

- Thread `T1` besetzt den vom Objekt `lock1` geschützten Block.
- Thread `T2` besetzt den vom Objekt `lock2` geschützten Block.
- Thread `T1` möchte den von `lock2` geschützten Block betreten, wartet also darauf, dass Thread `T2` diesen Block verlässt.
- Dies wird Thread `T2` aber nicht tun, bevor er in dem von `lock1` geschützten Block gewesen ist. Thread `T2` wartet also darauf, dass Thread `T1` diesen Block verlässt.

15.1.8 Unbehandelte Ausnahmen

Als Ergebnis einer unbehandelten Ausnahme wird die Anwendung von den CLR gestoppt. Das gilt für den primären Thread, einen per **Start()** explizit aktivierten sekundären Thread und auch für einen Pool-Thread.

Vor .NET 2.0 hat die CLR hingegen die Anwendung nach einer unbehandelten Ausnahme in explizit gestarteten (sekundären) Threads oder in einem Pool-Thread (siehe Abschnitt 15.2) weiterlaufen lassen. Bei einer Konsolenanwendung erschien das Ausnahmeprotokoll mit Typangabe, Message und Stack-Trace, bei einer GUI-Anwendung hat der Anwender nichts von dem Problem erfahren. Über die folgende Anwendungskonfiguration (in der Datei **app.config**) kann dieses, heute als riskant beurteilte Verhalten auch für modernen Anwendungen angeordnet werden:

```
<configuration>
  <runtime>
    <legacyUnhandledExceptionPolicy enabled="1" />
  </runtime>
</configuration>
```

Als zentrale Einsprungstelle zur Reaktion auf unbehandelte Ausnahmen, die in einem beliebigen Thread aufgetreten sind, bietet die Klasse **AppDomain** das Ereignis **UnhandledException** an.[1] Das folgende Programm registriert eine Behandlungsmethode für das Ereignis **UnhandledException** bei der voreingestellten Anwendungsdomäne uns startet dann einen sekundären Thread, der nach kurzer Ruhe eine unbehandelte Ausnahme wirft:

[1] In einer Anwendung können neben der voreingestellten Anwendungsdomäne noch weitere Domänen existieren, die im Speicher getrennt sind und separat entladen werden können, um die Stabilität der Anwendung zu erhöhen. Das Konzept der Anwendungsdomäne hat keine große praktische Bedeutung und wird daher im Kurs nicht behandelt.

```
using System;
using System.Threading;

class UnhandledException {
   public static void Main() {
      AppDomain.CurrentDomain.UnhandledException += UnHandler;
      new Thread(Werfer).Start();
      for(int i = 0; ; i++) {
         Thread.Sleep(1000);
         Console.WriteLine("Ausgabe durch den main-Thread: "+ i);
      }
   }

   static void Werfer() {
      Thread.Sleep(3000);
      throw new InvalidOperationException("Aus einem sekundären Thread");
   }

   static void UnHandler(object sender, UnhandledExceptionEventArgs e) {
      Console.WriteLine("\nUnbehandelte Ausnahme: " +
                        $"{((Exception) e.ExceptionObject).Message}\n");
      // Protokoll in eine Logdatei schreiben
   }
}
```

Unter einer aktuellen .NET - Version (≥ 2.0) wird die **UnhandledException**-Ereignismethode ausgeführt und anschließend die Anwendung gestoppt. Die Methode kann z.B. einen Logdatei-Eintrag schreiben oder eine Datenrettung versuchen, aber das Anwendungsende nicht verhindern. Zusammen mit der oben beschriebenen **legacyUnhandledExceptionPolicy** erhält man hingegen eine Ausnahmebehandlung *ohne* Anwendungsstopp.

15.2 Treadpool

Durch eine große Zahl von Threads mit kurzer Lebensdauer wird eine Anwendung eher ausgebremst als beschleunigt. Statt für viele einzelne Aufgaben jeweils einen neuen Thread zeitaufwändig zu erzeugen und anschließend wieder zu zerstören, sollten die vom .NET-Framework zur Verfügung gestellten Pools von Arbeits- und Ein-/Ausgabe -Threads genutzt werden. Eingehende Aufträge gelangen in eine Warteschlange und werden vom nächsten freien Thread aus dem „Bereitschaftsteam“ übernommen.

In einem Threadpool befinden sich nur *Hintergrund*-Threads, die nach dem Ende des letzten Vordergrund-Threads von der CLR automatisch gestoppt werden. Vor dem Beenden des letzten Vordergrund-Threads muss man also eventuell prüfen, ob ein Pool-Thread noch Aufgaben ausführt, die nicht abgebrochen werden dürfen.

Anstelle eines Threadpool - Auftrags ist in einigen Situationen ein eigener Thread erfoderlich bzw. empfehlenswert, z.B.:[1]

- Es wird ein *Vordergrund*-Thread benötigt.
- Es wird eine spezielle Thread-Priorität bevorzugt.
  Alle Pool-Threads haben eine *normale* Priorität (siehe Abschnitt 15.1.6.1).

[1] https://msdn.microsoft.com/en-us/library/0ka9477y(v=vs.110).aspx

- Es ist mit einer langen Bearbeitunsgdauer zu rechnen.
  Statt in dieser Situation einen Thread gem. Abschnitt 15.1 zu erstellen, kann man eine Aufgabe mit der **Task.Factory** - Methode **StartNew()** erstellen und mit dem Parameterwert **TaskCreationOptions.LongRunning** dafür sorgen, dass kein Pool-Thread, ein sondern eigenständiger Thread verwendet wird (siehe Abschnitt 15.2.2).

Der Threadpool passt seine „Personalstärke“ dynamisch an den Bedarf an und bevorzugt dabei die Anzahl der verfügbaren (logischen) Prozessorkerne an, sofern so viele Threads ausgelastet werden können. Wenn allerdings viele Threads bei entsprechend geringem CPU-Zeitbedarf auf das Dateisystem oder auf das Netzwerk warten müssen, nimmt die CLR zusätzliche Threads in Betrieb (Griffiths 2013, S. 609). Selbstverständlich wird die Anzahl der Pool-Threads auch nach unten korrigiert, wenn die Anzahl zu erledigender Aufträge zurückgeht. Schließlich belegt ein Thread Ressourcen (z.B. 1 MB Stack-Speicher). Es ist nur selten sinnvoll, mit den **ThreadPool**-Methoden **SetMaxThreads()** und **SetMinThreads()** die Entscheidung der CLR über die angemessene Zahl von Threads im Pool zu beeinflussen.

Im Threadpool sind Spezialisten für zwei Aufgabetypen tätig:

- Worker Threads
  Die mit Hilfe der Klassen **ThreadPool** (siehe Abschnitt 15.2.1) oder **Task** (siehe Abschnitt 15.2.2) übergebenen Delegatenobjekte werden durch Worker Threads abgearbeitet.
- I/O Completion Threads
  Soll nach Abschluss eines asynchronen Zugriffs auf das Dateisystem oder das Netzwerk eine Methode ausgeführt werden, die z.B. den Delegatentyp **AsyncCallback** implementiert, kommt ein I/O Completion Thread zum Einsatz.

15.2.1 Traditionelle Threadpool-Technik

Bei der traditionellen Threadpool-Technik (vor .NET 4.0) ist nur *eine* Warteschlange vorhanden, und die Arbeitsaufträge werden in der Reihenfolge ihres Eingangs abgearbeitet (FIFO, First-In-First-Out). Wenn viele Aufträge anfallen und gleichzeitig mehrere (logische) CPU-Kerne verfügbar sind, kann das Leistungspotential der Hardware durch die traditionelle Threadpool-Technik nicht optimal ausgeschöpft werden. Folglich sollte die traditionelle Threadpool-Technik nicht mehr benutzt werden. Sie wird trotzdem anschließend beschrieben, weil entsprechender Code noch oft anzutreffen ist, der verstanden, gepflegt und eventuell modernisiert werden muss.

Soll eine Methode im Hintergrund durch den konventionell verwalteten Threadpool ausgeführt werden, muss sie dem Delegatentyp **WaitCallback** entsprechen:

public delegate void WaitCallback(Object *state***)**

entsprechen, z.B.

```
static void HintergrundAktion(Object anz) {
   double summe;
   for (int i = 0; i < (int)anz; i++) {
      summe = 0.0;
      for (int j = 0; j < 10000000; j++)
         summe += zzg.Next(100);
      Console.WriteLine("Aktuelle Zufallssumme: " + summe);
   }
}
```

Diese Methode berechnet eine wählbare Anzahl von Summen, jeweils bestehend aus reichlich vielen Zufallszahlen, die von einem **Random**-Objekt erstellt werden.

Zur Übergabe eines **WaitCallback**-Arbeitsauftrags an den traditionellen Threadpool dient die **ThreadPool**-Methode **QueueUserWorkItem()** mit den folgenden Überladungen:

public static bool QueueUserWorkItem(WaitCallback *callBack*)

public static bool QueueUserWorkItem(WaitCallback *callBack*, Object *state*)

Der *state*-Parameter der zweiten Überladung ist für Daten vorgesehen, die der **WaitCallback**-Methode beim Aufruf übergeben werden sollen. Im Beispiel lässt so festlegen, wie viele Zufallssummen berechnet werden sollen.

In der folgenden Anweisung wird ein Arbeitsauftrag in die Warteschlange des traditionellen Threadpools gestellt:

```
ThreadPool.QueueUserWorkItem(HintergrundAktion, 5);
```

15.2.2 Threadpool 4.0

Mit .NET 4.0 wurde die parallele Programmierung durch die *Task Parallel Library* (TPL) reformiert, die in Abschnitt 15.4 behandelt wird. Dort wird es um TPL-Optionen zur flexiblen Aufgabensteuerung gehen. Man kann z.B. ...

- auf die Fertigstellung von (mehreren) Aufträgen warten,
- Ergebnisse von Aufträgen entgegen nehmen,
- Aufträge abbrechen bzw. fortsetzen,
- Aufträge als Langläufer deklarieren, um den Aufgabenplaner (engl.: *task scheduler*) zu unterstützen.

Die TLP benutzt den CLR-Threadpool, und der hat in .NET 4.0 Optimierungen erfahren, die nun behandelt werden sollen. Arbeitsabläufe werden so organisiert, dass die in Mehrkernprozessoren für jeden Kern vorhandenen lokalen Zwischenspeicher optimal genutzt und Zugriffe auf den langsameren Hauptspeicher nach Möglichkeit vermieden werden (Griffiths 2013, S. 607f):

- Für jeden (logischen) Prozessorkern wird eine separate Warteschlange verwendet, damit ein Auftrag möglichst vom selben Kern erledigt wird, der ihn erstellt hat. So besteht eine hohe Wahrscheinlichkeit, dass sich benötigte Daten im lokalen Cache des Kerns befinden.
- Die einem (logischen) Prozessorkern zugeordneten Aufgaben werden nach dem LIFO-Prinzip (Last-In-First-Out) erledigt, um den Durchsatz zu erhöhen. Bei den zuletzt eingelieferten Arbeitsaufträgen ist nämlich die Chance am größten, dass sich relevante Zustandsdaten noch im lokalen Cache des Prozessorkerns befinden.
- Bei Beschäftigungsmangel bedient sich ein Prozessorkern aus den Warteschlangen der Kollegen, wobei ältere, nicht mehr im lokalen Cache befindliche Aufträge bevorzugt werden (FIFO-Prinzip). In der englischsprachigen Literatur spricht man vom *work stealing*, obwohl doch eine partnerschaftliche Hilfe stattfindet.

Mit der in Abschnitt 15.2.1 beschriebenen **ThreadPool**-Methode **QueueUserWorkItem()** lassen sich die Threadpool-Optimierungen *nicht* nutzen. Sie sollte daher ab .NET 4.0 durch die gleich vorzustellenden Methoden unter Verwendung der Klasse **Task** ersetzt werden. Wie eingangs angedeutet, verzichten wir vorläufig auf eine flexible Aufgabenverwaltung und verwenden den leistungsoptimierten Threadpool weiterhin für Aufaben nach dem Motto „Fire & Forget“.

Zur Übergabe eines Arbeitsauftrags dient die Methode **StartNew()** der Klasse **TaskFactory**, die sich (wie auch die Klasse **Task**) im Namensraum: **System.Threading.Tasks** befindet. Wir beschränken uns auf die folgende Überladung:

public Task StartNew(Action<Object> *action***, Object** *state***)**

Als erster Parameter ist ein Objekt vom Delegatentyp **Action<Object>** zu übergeben:

public delegate void Action<Object>(Object *state***)**

Offenbar verlangt dieser Delegatentyp von einer Methode nichts anderes als der Delegatentyp **WaitCallback**, den die Methode **QueueUserWorkItem()** für ihren ersten Parameter verwendet (siehe Abschnitt 15.2.1).

Die per *state*-Parameter an **StartNew()** übergebenen Daten werden beim Aufruf der **Action<Object>** - Methode weitergereicht.

Ein einsatzfähiges Objekt der Klasse **TaskFactory** ist über die statische Eigenschaft **Factory** der Klasse **Task** ansprechbar, z.B.:

```
Task.Factory.StartNew(G1Task, i);
```

Die folgende Methode `G1Task()` erfüllt den Delegatentyp **Action<Object>**, kann also als erster Parameter an **StartNew()** übergeben werden:

```
static void G1Task(Object nr) {
 Random zzg = new Random();
 const int SIZE = 10;
 for (int i = 0; i < 4; i++) {
  double[] d = new double[SIZE];
  d[0] = (int)nr;
  d[1] = i;
  for (int j = 2; j < SIZE; j++)
   d[j] = zzg.Next(size);
  Task.Factory.StartNew(G2Task, d);
 }
 long dauer = DateTime.Now.Ticks - start;
 Console.WriteLine($"G1Task {nr}, Zeit seit Start: {(dauer/1.0e7),6:f4} Sek., "+
                   $"ThreadId: {Thread.CurrentThread.ManagedThreadId}");
}
```

`G1Task()` startet vier neue Aufträge unter Verwendung der Methode `G2Task()`, die ebenfalls den Delegatentyp **Action<Object>** erfüllt und als *state*-Parameter einen **double**-Array mit Zufallszahlen erhält:

```
Task.Factory.StartNew(G2Task, d);
```

Am Ende eines `G1Task()` - Aufrufs wird die Zeitdifferenz seit dem Programmstart und die Kennung des ausführenden Pool-Threads ausgegeben.

In `G2Task()` wird durch Aufsummieren von quadrierten Zufallszahlen Rechenzeit verbraucht. Abschließend werden die Zeitdifferenz seit dem Programmstart und die Kennung des ausführenden Pool-Threads ausgegeben:

```
static void G2Task(Object da) {
   double[] d = (double[])da;
   double summe = 0.0;
   for (int i = 0; i < 10000000; i++) {
      summe = 0.0;
      for (int j = 0; j < d.Length; j++)
         summe += d[j] * d[j];
   }
   long dauer = DateTime.Now.Ticks - start;
   Console.WriteLine($"G2Task ({d[0]},{d[1]})" +
                     $", Zeit seit Start: {(dauer/1.0e7),6:f4} Sek., " +
                     $"ThreadId: {Thread.CurrentThread.ManagedThreadId}");
}
```

Im der **Main()** - Methode des folgenden Programms werden vier `G1Task()` - basierte Aufträge gestartet:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class TaskPool {
   static long start;

   static void G1Task(Object nr) {
       . . .
   }

   static void G2Task(Object da) {
       . . .
   }

   static void Main() {
      start = DateTime.Now.Ticks;
      int anz = 4;
      for (int i = 0; i < anz; i++) {
         Task.Factory.StartNew(G1Task, i);
      }
      Console.ReadLine(); // Hält den Vordergrund-Thread aktiv
   }
}
```

Die auf eine per Enter abgeschickte Zeile lauernde Methode **ReadLine()** verhindert das Ende der Methode **Main()** und damit das Ende des Haupt-Threads und das Ende des Programms. Die im Threadpool tätigen Hintergrund-Threads könnten das Programm nicht am Leben erhalten.

Auf einem Rechner mit vier logischen Prozessorkernen hat sich das folgende Ablaufprotokoll ergeben:

```
G1Task 0, Zeit seit Start: 0,0230 Sek., ThreadId: 3
G1Task 1, Zeit seit Start: 0,0230 Sek., ThreadId: 4
G1Task 2, Zeit seit Start: 0,0230 Sek., ThreadId: 5
G1Task 3, Zeit seit Start: 0,0235 Sek., ThreadId: 6
G2Task (2,3), Zeit seit Start: 1,4788 Sek., ThreadId: 5
G2Task (0,3), Zeit seit Start: 1,4813 Sek., ThreadId: 3
G2Task (1,3), Zeit seit Start: 1,4883 Sek., ThreadId: 4
G2Task (3,3), Zeit seit Start: 1,4843 Sek., ThreadId: 6
G2Task (1,2), Zeit seit Start: 2,8311 Sek., ThreadId: 4
G2Task (2,2), Zeit seit Start: 2,8506 Sek., ThreadId: 5
G2Task (0,2), Zeit seit Start: 2,8506 Sek., ThreadId: 3
G2Task (3,2), Zeit seit Start: 2,8881 Sek., ThreadId: 6
G2Task (1,1), Zeit seit Start: 4,1958 Sek., ThreadId: 4
G2Task (2,1), Zeit seit Start: 4,2018 Sek., ThreadId: 5
G2Task (0,1), Zeit seit Start: 4,2108 Sek., ThreadId: 3
G2Task (3,1), Zeit seit Start: 4,2523 Sek., ThreadId: 6
G2Task (2,0), Zeit seit Start: 5,5540 Sek., ThreadId: 5
G2Task (1,0), Zeit seit Start: 5,5625 Sek., ThreadId: 4
G2Task (0,0), Zeit seit Start: 5,5645 Sek., ThreadId: 3
G2Task (3,0), Zeit seit Start: 5,5875 Sek., ThreadId: 6
```

Die vier Aufgabenpakete (jeweils einer `G1Task` zugehörig) werden von den vorhandenen vier logischen Prozessorkernen quasi-parallel ausgeführt. In den Warteschlangen, die fest mit einem Thread und vermutlich auch mit einem Prozessorkern assoziiert sind, werden die „lokalen“ Aufträge nach dem LIFO-Prinzip abgearbeitet:

Queue zur ThreadId 3	Queue zur ThreadId 4	Queue zur ThreadId 5	Queue zur ThreadId 6
(0, 0)	(1, 0)	(2, 0)	(3, 0)
(0, 1)	(1, 1)	(2, 1)	(3, 1)
(0, 2)	(1, 2)	(2, 2)	(3, 2)
(0, 3)	(1, 3)	(2, 3)	(3, 3)

Um das Beispielprogramm auf die Nutzung der traditionellen Threadpool-Technik umzustellen, sind die **StartNew()** - Aufrufe

```
Task.Factory.StartNew(G1Task, i);
 . . .
Task.Factory.StartNew(G2Task, d);
```

zu ersetzen durch **QueueUserWorkItem()** - Aufrufe (siehe Abschnitt 15.2.1):

```
ThreadPool.QueueUserWorkItem(G1Task, i);
. . .
ThreadPool.QueueUserWorkItem(G2Task, d);
```

Auf einem Rechner mit vier logischen Prozessorkernen hat sich das folgende Ablaufprotokoll ergeben:

```
G1Task 0, Zeit seit Start: 0,0095 Sek., ThreadId: 3
G1Task 1, Zeit seit Start: 0,0100 Sek., ThreadId: 4
G1Task 2, Zeit seit Start: 0,0100 Sek., ThreadId: 5
G1Task 3, Zeit seit Start: 0,0105 Sek., ThreadId: 6
G2Task (0,1), Zeit seit Start: 1,4438 Sek., ThreadId: 4
G2Task (0,0), Zeit seit Start: 1,4448 Sek., ThreadId: 3
G2Task (0,3), Zeit seit Start: 1,4478 Sek., ThreadId: 6
G2Task (0,2), Zeit seit Start: 1,4488 Sek., ThreadId: 5
G2Task (1,1), Zeit seit Start: 2,7940 Sek., ThreadId: 3
G2Task (1,2), Zeit seit Start: 2,7955 Sek., ThreadId: 6
G2Task (1,0), Zeit seit Start: 2,8080 Sek., ThreadId: 4
G2Task (1,3), Zeit seit Start: 2,8215 Sek., ThreadId: 5
G2Task (2,0), Zeit seit Start: 4,1322 Sek., ThreadId: 3
G2Task (2,1), Zeit seit Start: 4,1432 Sek., ThreadId: 6
G2Task (2,2), Zeit seit Start: 4,1502 Sek., ThreadId: 4
G2Task (2,3), Zeit seit Start: 4,1863 Sek., ThreadId: 5
G2Task (3,1), Zeit seit Start: 5,4820 Sek., ThreadId: 6
G2Task (3,0), Zeit seit Start: 5,5030 Sek., ThreadId: 3
G2Task (3,2), Zeit seit Start: 5,5035 Sek., ThreadId: 4
G2Task (3,3), Zeit seit Start: 5,5200 Sek., ThreadId: 5
```

Die Aufgaben stehen in *einer* Warteschlange und werden nach dem FIFO-Prinzip abgearbeitet. Bei der Verteilung auf die vier Threads im Pool wird die Familienzugehörigkeit *nicht* beachtet;

Nr. in *der* Queue	Task	ThreadId	Startreihenfolge
1	(0, 0)	3	
2	(0, 1)	4	
3	(0, 2)	5	
4	(0, 3)	6	
5	(1, 0)	4	
6	(1, 1)	3	
7	(1, 2)	6	
8	(1, 3)	5	
9	(2, 0)	3	
10	(2, 1)	6	
11	(2, 2)	4	
12	(2, 3)	5	
13	(3, 0)	3	
14	(3, 1)	6	
15	(3, 2)	4	
16	(3, 3)	5	

Weil die Prozessorkerne unterschiedlich leistungsfähig bzw. verfügbar sind, werden die Aufgaben nicht in der perfekten FIFO-Reihenfolge fertig.

Im Beispiel führt die verbesserte Threadpool-Verwaltung von .NET 4.0 *nicht* zu einer Leistungssteigerung. Das sollte aber niemanden davon abhalten, in der Regel der vom Microsoft-Insider Eric Eilebrecht in einem MSDN-Blogbeitrag ausgesprochenen Empfehlung zu folgen:[1]

> For new code, Task is now the preferred way to queue work to the thread pool.

Die im Beispiel verwendete **StartNew()** - Überladung

public Task StartNew(Action<Object> *action*, **Object** *state*)

verwendet für den ersten Parameter den konkretisierten generischen Delegatentyp **Action<Object>**. Eine realisierende Methode muss mit dem Parametertyp **Object** arbeiten, was eine explizite Typumwandlung erzwingt und keine perfekte Typüberwachung durch den Compiler erlaubt, z.B.:

```
static void G2Task(Object da) {
   double[] d = (double[])da;
   double summe = 0.0;
    . . .
}
```

Nach einem Vorschlag von Griffiths (2013, S. 607) lässt sich der Parametertyp **Object** so vermeiden:

- Man verwendet die **StartNew()** - Überladung mit einem einzigen Parameter vom Delegatentyp **Action**:
  public Task StartNew(Action *action*)
- Eine realisierende Methode muss den Delegatentyp **Action** erfüllen:
  public delegate void Action()
- Man verwendet eine per Lambda-Ausdruck definierte anonyme Methode, die in ihrem Rumpf den eigentlichen Arbeitsauftrag enthält, z.B.:
  ```
  Task.Factory.StartNew(() => G2Task(d));
  ```
  Von großem Nutzen ist dabei die Möglichkeit, in einer anonymen Methode auf lokale Variablen der umgebenden Methode zuzugreifen (siehe Abschnitt 9.1.5.1).

[1] https://blogs.msdn.microsoft.com/ericeil/2009/04/23/clr-4-0-threadpool-improvements-part-1/

- Die als Task auszuführende Methode kann den gewünschten Parametertyp verwenden, z.B.:

```
static void G2Task(double[] d) {
  double summe = 0.0;
   . . .
}
```

Bei einigen **StartNew()** - Überladungen kann man das Verhalten des Aufgabenplaners durch einen Parameter vom Enumerationstyp **TaskCreationOptions** beeinflussen, z.B.:

public Task StartNew(Action<Object> *action***, Object** *state***, TaskCreationOptions** *options***)**

Mit dem häufig verwendeten Wert **TaskCreationOptions.LongRunning** kündigt man einen Langläufer an und ermuntert den Aufgabenplaner, die Anzahl der Pool-Threads über die Anzahl der logischen CPU-Kerne hinaus zu steigern. Eventuell entscheidet sich der Aufgabenplaner auch dafür, den Threadpool überhaupt nicht mit der Aufgabe zu betrauen und stattdessen einen dedizierten Thread zu verwenden.[1]

Seit .NET 4.5 beherrscht die Klasse **Task** die statische Methode **Run()** in diversen Überladungen, z.B.:

public static Task Run(Action *action***)**

Damit kann man sich den „Umweg“ über die **Factory** ersparen. Wie Sie bereits wissen, ist es trotz des parameterfreien Delegatentyps **Action** mit Hilfe eines Lambda-Ausdrucks möglich, an eine per Pool-Thread auszuführenden Methode Daten zu übergeben:

```
Task.Run(() => G2Task(d));
```

15.2.3 Worker-Threads in WPF-Anwendungen

Um eine konsistente und verzögerungsfrei reagierende Bedienoberfläche zu garantieren, verwendet das WPF-Framework eine **Single-Thread - Architektur**: Auf ein UI-Element darf nur derjenige Thread zugreifen, der das Element erzeugt hat. Andererseits müssen zeitaufwändige Arbeiten aus dem UI-Thread heraus gehalten werden, um die die Reaktionsfähigkeit der Bedienoberfläche sicher zu stellen. Wenn nun z.B. ein Pool-Thread eine Aufgabe erledigt hat, muss eine Möglichkeit geschaffen werden, das Ergebnis unter Beachtung der Single-Thread-Regel in der Bedienoberfläche anzuzeigen. Genau dies ermöglicht die Klasse **SynchronizationContext** aus dem Namensraum **System.Threading**:

- In einer WPF-Ereignisbehandlungsmethode erhält man über die statische **SynchronizationContext**-Eigenschaft **Current** ein **SynchronizationContext**-Objekt, das mit dem UI-Thread assoziiert ist.
- Durch einen Aufruf der **SynchronizationContext**-Methode **Post()** mit einem Delegatenobjekt vom Typ **SendOrPostCallback** als Aktualparameter sorgt man dafür, dass die realisierende Methode im UI-Thread ausgeführt wird.

Einen Worker-Thread in einer WPF-Anwendung zu verwenden, ist nicht mehr unbedingt Stand der C# - Programmierkunst. In Abschnitt 15.5 wird eine modernere und bequemere Lösung vorgestellt für das Problem, eine zeitaufwändige Aufgabe per Pool-Thread zu erledigen und anschließend die Bedienoberfläche zu aktualisieren (asynchrone Programmierung mit den Schlüsselwörtern **async** und **await**). Ein paar Argumente sprachen dafür, den kurzen Abschnitt über den Einsatz von Worker-Threads in WPF-Anwendungen noch nicht aus dem Manuskript zu streichen:

---

[1] https://msdn.microsoft.com/de-de/library/system.threading.tasks.taskcreationoptions(v=vs.110).aspx

- Manche Leser wollen oder müssen sich vielleicht mit der traditionellen Lösung auseinandersetzen.
- Die traditionelle Lösung kann als Kontrast dienen, um die Überlegenheit der später behandelten modernen Lösung zu demonstrieren.

Mit Hilfe eines Worker-Threads soll eine Schwäche des RSS-Feed - Readers behoben werden, den wir in Abschnitt 5.7 erstellt haben. Während eine angeforderte Feed-Datei aus dem Internet geladen wird, können viele Sekunden vergehen, und in dieser Zeit reagiert das Programm nicht auf Benutzerwünsche, so dass z.B. das Anwendungsfenster nicht verschoben werden kann.

Zur Beseitigung des Problems sind lediglich einige kleine Änderungen in der **Click**-Behandlungsmethode zum Befehlsschalter der Anwendung vorzunehmen:

```
private void button_Click(object sender, RoutedEventArgs e) {
   SynchronizationContext contextUI = SynchronizationContext.Current;
   var waitInfo = new List<RssItem>() { new RssItem() { Title = "Bitte Warten ..." } };
   listBox.ItemsSource = waitInfo;
   button.IsEnabled = false;

   String s = textBox.Text;
   Task.Factory.StartNew(() => {
      try {
         XDocument feed = XDocument.Load(s);
         var items = new List<RssItem>();
         IEnumerable<XElement> xDocItems = feed.Descendants("item");
         RssItem rit;
         foreach (XElement item in xDocItems) {
            rit = new RssItem() {
               Title = item.Element("title").Value,
               Description = Regex.Replace(item.Element("description").Value, "<[^>]*>",
                                           String.Empty, RegexOptions.IgnoreCase).Trim(),
               Url = item.Element("link").Value
            };
            items.Add(rit);
         }
         contextUI.Post(notUsed => {listBox.ItemsSource = items;}, null);
      } catch (Exception ex) {
         contextUI.Post(notUsed => {listBox.ItemsSource = null;
                                    MessageBox.Show(this, ex.ToString(), ex.Message);}, null);
      } finally {
         contextUI.Post(notUsed => {button.IsEnabled = true;}, null);
      }
   });
}
```

Weil die Methode im UI-Thread ausgeführt wird, resultiert aus der folgenden Anweisung ein **SynchronizationContext**-Objekt, das mit dem UI-Thread assoziiert ist:

```
SynchronizationContext contextUI = SynchronizationContext.Current;
```

Die **try**-Anweisung, welche einen Internet-Zugriff und die bei einem umfangreichen Feed eventuell zeitaufwändige Aufbereitung der Daten enthält, soll vom Threadpool ausgeführt werden. Daher erstellen wir eine anonyme, den Delegatentyp **Action** realisierende Methode per Lambda-Ausdruck mit der **try**-Anweisung im Rumpf und übergeben dieses Delegatenobjekt an die **Task.Factory** - Methode **StartNew()** als Aktualparameter:

```
Task.Factory.StartNew(() => {
  try {
     . . .
  } catch (Exception ex) {
     . . .
  }
});
```

Weil die **try**-Anweisung in einem Pool-Thread ausgeführt wird, darf hier kein (lesender oder schreibender) Zugriff auf den UI-Thread stattfinden. Daher wird die **Text**-Eigenschaft der **TextBox** noch im UI-Thread extrahiert.

Um das fertig aufbereitete **List<RssItem>** - Objekt als Wert der **ListBox**-Eigenschaft **ItemsSource** festzulegen, wird die **Post()** - Methode des mit dem UI-Thread assoziierten **SynchronizationContext**-Objekts aufgerufen mit einem per Lambda-Ausdruck realisierten **SendOrPostCallback**-Delegaten als Aktualparameter:

```
contextUI.Post(notUsed => {listBox.ItemsSource = items;}, null);
```

Der zweite **Post()** - Parameter dient zur Übergabe von Daten an den Delegaten, und im Beispiel genügt das Referenzliteral **null**. Der Parameter des Lambda-Ausdrucks wird vom Delegatentyp **SendOrPostCallback** verlangt und ist im Beispiel irrelevant.

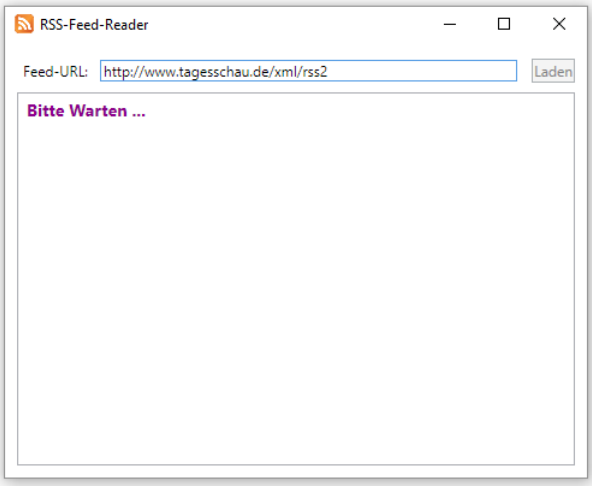
Auch die im **catch**- und im **finally**-Block befindlichen GUI-relevanten Anweisungen müssen etwas mühselig an den UI-Thread delegiert werden. U.a. wird der am Anfang der Ereignisbehandlung aus naheliegenden Gründen deaktivierte Befehlsschalter im **finally**-Block wieder nutzbar gemacht:

```
contextUI.Post(notUsed => {button.IsEnabled = true;}, null);
```

Damit der Benutzer während der Wartezeit darüber informiert ist, was das Programm gerade tut, erhält die **ListBox** einen provisorischen Inhalt

```
var waitInfo = new List<RssItem>()
         {new RssItem() {Title = "Feed wird geladen. Bitte Warten ..." }};
listBox.ItemsSource = waitInfo;
```

mit Erläuterung:



Den früher während des Ladens angezeigten Wait-Cursor brauchen wir nicht mehr.

15.3 Timer

Wenn eine Aktivität in festen Zeitabständen wiederholt werden muss (z.B. Aktualisieren der Zeitanzeige in einem Fenster eines WPF-Programms, Überprüfung der Funktionstüchtigkeit eines Servers), bieten die Timer-Klassen in der FCL eine bequeme und effiziente Lösung. Ihre Objekte können beauftragt werden, eine Methode regelmäßig aufzurufen. Anschließend werden zwei Klassen vorgestellt:

- Die Klasse **Timer** im Namensraum **System.Threading** kommt in Frage, wenn die regelmäßig auszuführende Aktion aufwändig ist und in einem Pool-Thread ablaufen sollte.
- Die Klasse **DispatcherTimer** im Namensraum **System.Windows.Threading** ist im Rahmen von WPF-Anwendungen bequemer einsetzbar. Allerdings laufen die regelmäßig auszuführenden Aktionen im UI-Thread ab, so dass bei hohem Zeitbedarf eine unergonomisch zäh reagierende Bedienoberfläche resultieren kann.

15.3.1 Regelmäßige Hintergrundaktivitäten

Mit Hilfe der Klasse **Timer** im Namensraum **System.Threading** kann man die CLR beauftragen, eine Methode regelmäßig in einem Pool-Thread auszuführen. Im folgenden Beispiel

```
tim = new Timer(new TimerCallback(HintergrundAktion), null, 0, 10000);
```

kommt eine Konstruktor-Überladung mit folgenden Parameter-Datentypen zum Einsatz:

- **TimerCallback** *callback*
  Dieser Delegatentyp verlangt für die regelmäßig auszuführende Methode folgende Bauart:
  void TimerCallback(**Object** *state)*
- **Object** *state*
  Die regelmäßig auszuführenden Methode erhält beim Aufruf das im zweiten Konstruktorparameter anzugebende Objekt, so dass ihr Verhalten gesteuert werden kann. Ist (wie im Beispiel) kein Parameter erforderlich, übergibt man eine **null**-Referenz.
- **long** *dueTime*
  Mit diesem Parameter wird die Zeit bis zum *ersten* Aufruf in Millisekunden festgelegt.
- **long** *period*
  Durch diesem Parameter wird die Zeit zwischen zwei Aufrufen in Millisekunden festgelegt. Soll es bei *einem* Aufruf bleiben, versort man den Parameter *period* mit dem Wert **Timeout.Infinite**.

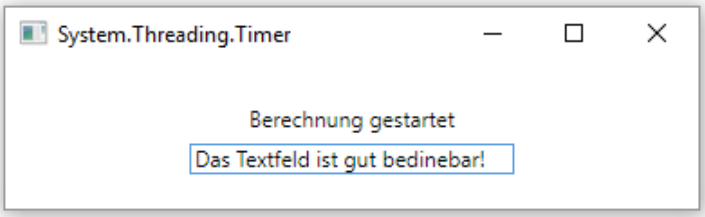
Bei aufwendigen Arbeiten kann es in Abhängigkeit von der gewählten Wartezeit zwischen zwei Aufrufen dazu kommen, dass die **TimerCallback**-Methode von mehreren Threads simultan ausgeführt wird. Wenn die Methode gemeinsame Daten (z.B. Instanz- oder Klassenvariablen) verändert, kommt eine Thread-Synchronisation in Frage (siehe Abschnitt 15.1.4). Eine ständig wachsende Zahl von simultanen Ausführungen der **TimerCallback**-Methode muss natürlich verhindert werden. Neben der Wahl eines passenden *period*-Konstruktorparameters besteht auch die Möglichkeit, für ein bereits aktives **Timer**-Objekt mit der Methode **Change()** die Intervalldauer zu verändern, z.B.:

```
tim.Change(0, 2000);
```

Wir stellen uns die Aufgabe, in einer GUI-Anwendung eine umfangreiche Aktivität regelmäßig per Threadpool im Hintergrund erledigen zu lassen, wobei die Benutzeroberfläche verzögerungsfrei bedienbar bleiben soll. Im folgenden Programm (eine handbestrickte WPF-Anwendung ohne XAML, vgl. Abschnitt 11.2.1) berechnet die Methode `HintergrundAktion` alle 10 Sekunden die Summe aus 200 Millionen Zufallszahlen:

```
using System;
using System.Windows;
using System.Threading;
using System.Windows.Controls;

class ThreadingTimer : Window {
   Label anzeige;
   Timer tim;
   Random zzg = new Random();

   ThreadingTimer() {
      Height = 120; Width = 400;
      Title = "System.Threading.Timer";

      StackPanel lm = new StackPanel();
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;

      anzeige = new Label();
      anzeige.HorizontalAlignment = System.Windows.HorizontalAlignment.Center;
      lm.Children.Add(anzeige);

      tim = new Timer(new TimerCallback(HintergrundAktion), null, 0, 10000);

      TextBox eingabe = new TextBox();
      eingabe.Width = 180;
      lm.Children.Add(eingabe);
   }

   void Melde (String s) {
      anzeige.Dispatcher.BeginInvoke((Action)(() => { anzeige.Content = s; }));
   }

   void HintergrundAktion(Object info) {
      double zs = 0;
      Melde("Berechnung gestartet");
      for (int i = 0; i < 200000000; i++)
         zs += zzg.Next(100);
      Melde("Aktuelle Zufallssumme: " + zs.ToString() +
            " berechnet um " + DateTime.Now.ToLongTimeString());
   }

   [STAThreadAttribute]
   static void Main() {
      new Application().Run(new ThreadingTimer());
   }
}
```

Trotz des erheblichen CPU-Zeitverbrauchs durch die Hintergrundaktivität reagiert die Benutzeroberfläche des Programms jederzeit verzögerungsfrei:



In einer WPF-Anwendung müssen Zugriffe auf Steuerelemente dem Thread vorbehalten bleiben, der sie erzeugt hat. Zugriffe durch fremde Threads werden von der CLR verhindert. In Abschnitt 15.2.3 wurde die **SynchronizationContext**-Methode **Post()** dazu verwendet, aus einem Hintergrund-Thread heraus die Bedienoberfläche zu aktualisieren. Im aktuellen Beispiel wird ein alterna-

tives Verfahren verwendet, weil an das **SynchronizationContext**-Objekt zum UI-Thread nicht ganz leicht heranzukommen ist:

- Der UI-Thread kann über die **Invoke()** - Methode oder die **BeginInvoke()** - Methode des zuständigen **Dispatcher**-Objekts veranlasst werden, eine Delegatenmethode auszuführen. An dieses Objekt kommt man über die **Dispatcher**-Eigenschaft eines Steuerelements heran.
- Das benötigte Parameterobjekt vom Typ **Delegate** kann z.B. per Lambda-Ausdruck erstellt werden. Weil der abstrakte Typ **Delegate** keinen Konstruktor besitzt, wird im Beispiel auf die **Delegate**-Ableitung **Action** zurückgegriffen:

```
anzeige.Dispatcher.BeginInvoke((Action)(() => { anzeige.Content = s; }));
```

Beim **Invoke()** - Aufruf wartet der Hintergrund-Thread, bis der UI-Thread die Aktualisierungsmethode beendet hat. Beim **BeginInvoke()** - Aufruf wartet der Hintergrund-Thread *nicht* auf das Ende der Aktualisierung durch den UI-Thread. In der Regel ist die asynchrone Variante (**BeginInvoke()**) zu bevorzugen. In unserem Beispiel wird so für das möglichst frühzeitige Ende der **Timer** - **Callback**-Methode `HintergrundAktion()` gesorgt, die einen Pool-Thread belegt.

Während zu jedem **BeginInvoke()** - Aufruf an ein *Delegatenobjekt* (siehe Abschnitt 15.6.1.1) der zugehörige **EndInvoke()** - Aufruf durchgeführt werden sollte, ist im vorliegenden Fall ausnahmsweise *kein* **EndInvoke()** - Aufruf erforderlich.

15.3.2 Regelmäßige Aktivitäten im UI-Thread

Im Namensraum **System.Windows.Threading** ist eine Klasse namens **DispatcherTimer** vorhanden, die ereignisorientiert arbeitet und bequem in eine UI-Anwendung einzubinden ist, z.B.:

```
using System;
using System.Windows;
using System.Windows.Threading;
using System.Windows.Controls;

class WPFTimer : Window {
   Label anzeige;
   DispatcherTimer tim;
   Random zzg = new Random();

   WPFTimer() {
      Height = 120; Width = 400;
      Title = "DispatcherTimer";

      StackPanel lm = new StackPanel();
      lm.VerticalAlignment = VerticalAlignment.Center;
      Content = lm;

      anzeige = new Label();
      anzeige.HorizontalAlignment = System.Windows.HorizontalAlignment.Center;
      lm.Children.Add(anzeige);

      tim = new DispatcherTimer();
      tim.Tick += new EventHandler(TimerAktion);
      tim.Interval = new TimeSpan(0,0,10);
      tim.Start();

      TextBox eingabe = new TextBox();
      eingabe.Width = 180;
      lm.Children.Add(eingabe);
   }
```

```
    void TimerAktion(Object myObject, EventArgs ea) {
       double zs = 0;
       anzeige.Content = "Berechnung gestartet"; // Bleibt ohne Effekt!
       for (int i = 0; i < 200000000; i++)
          zs += zzg.Next(100);
       anzeige.Content = "Aktuelle Zufallssumme: " + zs + " berechnet um " +
                    DateTime.Now.ToLongTimeString();
    }

    [STAThreadAttribute]
    static void Main() {
       new Application().Run(new WPFTimer());
    }
}
```

Diesmal laufen die vom Timer angestoßenen Methodenaufrufe jedoch im UI-Thread ab, und bei hoher Rechenlast resultiert eine unergonomisch zähe Bedienoberfläche.

Das aktuelle Beispiel ist dem Vorbild aus Abschnitt 15.3.1 nachempfunden und lässt ebenfalls alle 10 Sekunden die Summe aus 200 Millionen Zufallszahlen berechnen. Das hat aber nun zur Folge, dass nach jedem Tick-Ereignis einige Sekunden lang keine Eingaben in das Textfeld möglich sind.

Außerdem ist zu beachten, dass die Anzeige des Fensters nicht aktualisiert werden kann, solange eine **DispatcherTimer**-Ereignismethode läuft. Im Beispiel ist es daher *nicht* möglich, die **Label**-Anzeige „unterwegs“ zu aktualisieren:

```
anzeige.Content = "Berechnung gestartet"; // Bleibt ohne Effekt!
```

Bei der in Abschnitt 15.3.1 vorgestellten Lösung mit dem **Timer** aus dem Namensraum **System.Threading** war das möglich.

Für das regelmäßige Starten von Methoden mit geringem Rechenzeitbedarf ist die Klasse **DispatcherTimer** aber durchaus geeignet, weil beim Zugriff auf UI-Elemente keine Thread-Grenzen zu überwinden sind.

15.4 Aufgaben (Task Parallel Library)

Die mit .NET 4.0 eingeführte *Task Parallel Library* (TPL) bringt neue Klassen, die speziell zu optimalen Nutzung von Mehrkernsystemen konzipiert wurden. Für jede selbständig auszuführende Aufgabe entsteht ein Objekt der Klasse **Task**. Die Arbeitsaufgaben werden letztlich von Pool-Threads erledigt, wobei die ebenfalls mit .NET 4.0 eingeführten Optimierungen des Threadpools (siehe Abschnitt 15.2) für eine flotte Abwicklung sorgen.

Neben der Leistungsoptimierung bietet die TPL eine gute Kontrolle über die **Task**-Objekte (z.B. Warten auf Beendigung, Fortsetzungsaufgaben, zusammengesetzte Aufgeben, Abbrechen, Ausnahmebehandlung), so dass **Task**-Objekte in vielen Situationen gegenüber der direkten Verwendung von **Thread**-Objekten oder direkten Aufträgen an den Threadpool zu bevorzugen sind.

Daher empfiehlt Microsoft, in Anwendungen für eine Zielplattform ab .NET 4.0 die nebenläufige Programmierung möglichst per TPL zu realisieren.[1] Seit C# 5.0 ist bei dieser Empfehlung zu ergänzen, dass die TPL-Nutzung nach Möglichkeit über die neuen C# - Schlüsselwörter **async** und **await** geschehen sollte (siehe Abschnitt 15.5).

[1] Siehe http://msdn.microsoft.com/en-us/library/dd537609.aspx

15.4.1 Aufgaben ohne bzw. mit Rückgabe erstellen

Statt ein **Task**-Objekt per Konstruktor zu erzeugen und anschließend mit der **Start()** - Methode zu aktivieren,

```
Task task = new Task(SampleMean);
task.Start();
```

kann man die **StartNew()** - Methode der Klasse **TaskFactory** verwenden, um Kreation und Start in einem Aufruf zu erledigen. Ein Objekt der Klasse **TaskFactory** ist über die statische **Task**-Eigenschaft **Factory** ansprechbar, z.B.:

```
Task task = Task.Factory.StartNew(SampleMean);
```

Diese Technik wurde schon in Abschnitt 15.2.2 zur Vergabe von Arbeitsaufträgen an den Threadpool vorgeschlagen.

Im folgenden Programm entsteht aus der Methode `SampleMean()`, die den Mittelwert aus einer Anzahl von gleichverteilten Zufallszahlen aus dem Intervall [0; 1) ausgibt und keine Rückgabe liefert, ein Objekt der Klasse **Task**.[1] Die Aufgabe wird von einem Pool-Thread ausgeführt, der als Hintergrund-Thread die Anwendung nicht am Leben erhalten kann. In der **Main()** - Methode wird daher durch die später vorzustellende **Task**-Methode **Wait()** dafür gesorgt, dass der Haupt-Thread auf die Fertigstellung der Aufgabe wartet:

```
using System;
using System.Threading.Tasks;

class TaskDemo {
   const int SG = 10000;

   static void SampleMean() {
      Random zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < SG; j++)
         erg += zzg.NextDouble();
      Console.WriteLine("Stichprobenmittel " + (erg / SG));
   }

   static void Main() {
      Task task = Task.Factory.StartNew(SampleMean);
      // Parallele Arbeiten erledigen
      task.Wait();
   }
}
```

Damit ein Nutzen durch Parallelität entsteht, muss natürlich der Haupt-Thread während der Hintergrundbearbeitung selbst tätig werden, statt nur abzuwarten.

Seit .NET 4.5 ist die statische **Task**-Funktion **Run()** zum Starten einer Aufgabe verfügbar, z.B.:[2]

---

[1] Eine ähnliche Methode kann z.B. im Rahmen einer Statistik-Software sinnvoll sein, um die Verteilung des Stichprobenmittelwerts zu untersuchen.

[2] Im Beispiel muss durch die explizite Typumwandlung ein Fehler des Roslyn-Compilers kompensiert werden. Anderenfalls scheitert die Übersetzung mit der Meldung:

> Fehler CS0121 Der Aufruf unterscheidet nicht eindeutig zwischen den folgenden Methoden oder Eigenschaften: "Task.Run(Action)" und "Task.Run(Func<Task>).

Auf der Webseite

https://github.com/dotnet/roslyn/issues/250

des Roslyn-Projekts wird versprochen, im C# 7 eine Lösung des Problems in Erwägung zu ziehen:

> For a method group conversion, candidate methods whose return type doesn't match up with the delegate's return type are removed from the set.

```
Task task = Task.Run((Action)SampleMean);
```

Mit Hilfe der Klasse **Task** wurde eben eine nach dem Motto „Fire & Forget“ zu verwendende Hintergrundtätigkeit erstellt, die in der Praxis nicht allzu häufig benötigt wird. Soll eine Aufgabe bzw. die zugrunde liegende Methode einen **Rückgabewert** liefern, erzeugt man ein Objekt der generischen Klasse **Task<TResult>**, die von der Klasse **Task** abstammt und für den Rückgabetyp einen Typparameter besitzt. Im folgenden Programm

```
using System;
using System.Threading.Tasks;

class TaskResultDemo {
   const int SG = 10000;

   static double SampleMean() {
      Random zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < SG; j++)
         erg += zzg.NextDouble();
      return erg / SG;
   }

   static void Main() {
      Task<double> task = Task.Factory.StartNew<double>(SampleMean);
      // Parallele Arbeiten erledigen
      Console.WriteLine("Stichprobenmittel = " + task.Result);
   }
}
```

entsteht aus der Methode `SampleMean()`, die für eine Stichprobe mit gleichverteilten Zufallszahlen aus dem Intervall [0; 1) den Mittelwert bestimmt, mit Hilfe der Methode **StartNew<double>()** ein Objekt der Klasse **Task<double>**:

```
Task<double> task = Task.Factory.StartNew<double>(SampleMean);
```

Task<TResult> - Objekte machen ihr Resultat über die Eigenschaft **Result** verfügbar, z.B.:

```
Console.WriteLine("Stichprobenmittel = " + task.Result);
```

Ein Zugriff auf die **Result**-Eigenschaft einer noch nicht abgeschlossenen Aufgabe blockiert den aktuellen Thread. Er sollte sich so lange mit anderen Arbeiten beschäftigen, bis er das Ergebnis der Aufgabenbearbeitung benötigt.

Ist in einer Task eine unbehandelte Ausnahme aufgetreten, wird diese beim **Result**-Zugriff automatisch neu geworfen (siehe Abschnitt 15.4.5).

Ein Objekt der Klasse **Task<TResult>** wird in der englischsprachigen Literatur als *Future* bezeichnet, weil es ein Ergebnis repräsentiert, das im weiteren Verlauf eventuell verfügbar sein wird.

An ein **Task**- oder **Task<TResult>** - Objekt gelangt man auch durch den Aufruf von etlichen FCL-Methoden, z.B. aus der Klasse **Stream**: (vgl. Abschnitt 15.5.5)

- **public Task WriteAsync(byte[]** *buffer***, int** *offset***, int** *count***)**
  Vom angesprochenen **Stream**-Objekt wird ein **byte**-Array asynchron geschrieben. Die Methode kehrt sofort zurück, blockiert also nicht den aufrufenden Thread.
- **public Task<int> ReadAsync(byte[]** *buffer***, int** *offset***, int** *count***)**
  Das angesprochene **Stream**-Objekt liest asynchron Daten in einen **byte**-Array. Vom zurück gelieferten **Task<int>** - Objekt ist die Zahl der tatsächlich gelesenen Bytes über seine **Result**-Eigenschaft zu erfahren.

15.4.2 Zustände einer Aufgabe

In welchem Zustand sich ein **Task**-Objekt befindet, erfährt man über seine **Status**-Eigenschaft mit den folgenden möglichen Werten aus der Enumeration **TaskStatus**:

- **Created**
  Der Auftrag wurde initialisiert, aber noch nicht zur Bearbeitung eingeplant.
- **Cancelled**
  Die Aufgabe wurde abgebrochen (siehe Abschnitt 15.4.9).
- **Faulted**
  Die Bearbeitung wurde durch einen unbehandelten Ausnahmefehler abgebrochen (siehe Abschnitt 15.4.5).
- **RanToCompletion**
  Die Aufgabe wurde erfolgreich zu Ende geführt.
- **Running**
  Der Auftrag ist aktiv und noch nicht fertiggestellt.
- **WaitingForActivation**
  In diesem Zustand befindet sich z.B. einer Fortsetzungsaufgabe (siehe Abschnitt 15.4.7) vor dem Ende Ihres Vorgängers.
- **WaitingForChildrenToComplete**
  Die Aufgabe ist erledigt, wartet aber noch auf die Fertigstellung von Kindaufträgen (siehe Abschnitt 15.4.8).
- **WaitingToRun**
  Der Auftrag wartet darauf, von einem freien Thread ausgeführt zu werden.

Informationen über den Status einer Aufgabe lassen sich auch mit den folgenden booleschen **Task**-Eigenschaften ermitteln:

- **IsCanceled**
  Diese Eigenschaft hat den Wert **true**, wenn die Aufgabe abgebrochen worden ist (siehe Abschnitt 15.4.9).
- **IsCompleted**
  Diese Eigenschaft hat den Wert **true**, wenn die Aufgabe fehlerfrei beendet worden ist
- **IsFaulted**
  Diese Eigenschaft hat den Wert **true**, wenn die Bearbeitung durch einen Ausnahefehler abgebrochen worden ist.

15.4.3 Parameterabhängige Aufgaben

Soll eine Aufgabe bzw. die zugrunde liegende Methode in Abhängigkeit von einem Parameter tätig werden, ist ein alternativer Delegatentyp und eine entsprechende **StartNew<TResult>()** - Überladung zu verwenden.

Bei der oben als Beispiel verwendeten Methode `SampleMean()` zur Berechnung des Mittelwerts aus einer Serie von Zufallszahlen ist zu kritisieren, dass sie den voreingestellten, aus der Systemzeit abgeleiteten Startwert für den Pseudozufallszahlengenerator verwendet. So kann es passieren, dass mehrere, kurz nacheinander gestartete Aufgaben dieselben Pseudozufallszahlen produzieren. Dies wird bei der folgenden `SampleMean()` - Variante vermieden, die einen per Parameter übergebenen Startwert für den Pseudozufallszahlengenerator benutzt:

```
static double SampleMean(Object obj) {
   int start = (int) obj;
   Random zzg = new Random(start);
   double erg = 0.0;
   for (int j = 0; j < sg; j++)
      erg += zzg.NextDouble();
   return erg/sg;
}
```

Sie erfüllt den Delegatentyp **Func<in Object, out double>**

public delegate double Func<in Object, out double>(Object *arg***)**

und kann daher in der folgenden **StartNew<TResult>()** - Überladung

public Task<TResult> StartNew<TResult>(Func<Object, TResult> *function***, Object** *state***)**

dazu verwendet werden, **Task<double>** - Objekte mit Startparameter zu erstellen, z.B.:

```
Task.Factory.StartNew<double>(SampleMean, i);
```

15.4.4 Auf die Fertigstellung von Aufgaben warten

Bei vielen Algorithmen (Arbeitsabläufen) hängt das weitere Verhalten eines Threads vom Ergebnis einer Aufgabe ab, so dass deren Beendigung abgewartet werden muss. Um (den aktuellen Thread blockierend) auf die Fertigstellung eines **Task**-Objekts zu warten, ruft man dessen **Wait()** - Methode auf (analog zum Aufruf der **Thread**-Methode **Join()**, siehe Abschnitt 15.1.4.4.2).

Bei *mehreren* abzuwartenden Aufgaben kommen die **Task**-Methoden **WaitAll()** und **WaitAny()** zum Einsatz, die im Zusammenhang mit zusammengesetzten Aufgaben näher behandelt werden (siehe Abschnitt 15.4.8).

Die Methode **Wait()** haben wir bereits in Abschnitt 15.4.1 verwendet:

```
Task task = Task.Factory.StartNew(SampleMean);
// Parallele Arbeiten erledigen
task.Wait();
```

Zur potentiell unbegrenzt lange blockierenden **Wait()** - Überladung existiert eine Alternative mit Timeout-Parameter:

public bool Wait(TimeSpan *timeout***)**

Per Rückgabewert erfährt man, ob die Aufgabe in der festgelegten Zeitspanne fertig geworden ist.

Was im Fall von unbehandelten Ausnahmen geschieht, die während der Auftragserledigung aufgetreten sind, wird im nächsten Abschnitt beschrieben.

15.4.5 Unbehandelte Ausnahmen bei der Aufgabenbearbeitung

In diesem Abschnitt können nicht alle Details der Ausnahmebehandlung bei Verwendung der Task Parallel Library dargestellt werden. Für praktische Anwendungen sind weitere Informationen erforderlich (siehe z.B. http://msdn.microsoft.com/de-de/library/dd997415.aspx). Erfreulicherweise vereinfacht sich beim asynchronen Programmieren mit den seit C# 5.0 verfügbaren Schlüsselwörtern **async** und **await** speziell der Umgang mit unbehandelten Ausnahmen aus Hintergrundaufgaben (siehe Abschnitt 15.5.3).

15.4.5.1 Beobachtete Ausnahmen

Ist bei der Aufgabenbearbeitung (also im beteiligten Pool-Thread!) eine unbehandelte Ausnahme aufgetreten, wird diese ...

- beim Aufruf der **Task**-Methoden **Wait()**, **WaitAll()** und **WaitAny()**
- beim Zugriff auf eine **Task<TResult>** - Produktion per **Result**-Eigenschaft
- beim Zugriff auf die **Task**-Eigenschaft **Exception**

neu geworfen. Weil bei Beteiligung von mehreren Aufgaben auch mehrere Ausnahmen aufgetreten sein könnten, erfolgt generell die Verpackung in ein **AggregateException**-Objekt.

Erfolgt das Warten auf die Auftragsbearbeitung bzw. der Ergebniszugriff im Rahmen einer **try**-Anweisung kann die **AggregateException** abgefangen werden, damit sie nicht zum Beenden des Programms führt. Im folgenden Beispielprogramm beschränkt sich die Aufgabenmethode auf das Werfen einer Ausnahme, wobei zwei selbst definierte Ausnahmeklassen auftreten (vgl. Abschnitt 12.6).

```
using System;
using System.Threading.Tasks;

public sealed class HarmlessException : Exception {
   public HarmlessException(String message) : base(message) {}
}

public sealed class SeriousException : Exception {
   public SeriousException(String message) : base(message) {}
}

class WaitDemo {
   static void Werfer() {
      throw new HarmlessException("Harmlos");
      //throw new SeriousException("Ernstfall");
   }

   static bool InternalExceptionHandler(Exception e) {
      if (e is HarmlessException) {
         Console.WriteLine(e.Message);
         return true;
      } else
         return false;
   }

   static void Main() {
      Task t = Task.Factory.StartNew(Werfer);
      try {
         t.Wait();
      } catch (AggregateException ae) {
          ae.Handle(InternalExceptionHandler);
      }
      Console.WriteLine("Normales Ende");
   }
}
```

Zur Analyse der erhaltenen **AggregateException** wird deren Methode **Handle()** aufgerufen, die als Parameter ein Delegatenobjekt vom Typ **Func<Exception, bool>** erwartet. Dabei ist eine Methode verlangt, die einen Parameter vom Typ **Exception** entgegen nimmt und eine Rückgabe vom Typ **bool** abliefert. Diese Methode wird für jede im **AggregateException**-Objekt enthaltene Ausnahme aufgerufen und muss per Rückgabewert darüber informieren, ob eine Behandlung stattgefunden hat (**true**) oder nicht (**false**). Beim Rückgabewert **false** für mindestens eine Ausnahme wirft **Handle()** eine neue **AggregateException** mit den unbehandelten Ausnahmen. Kommt es bei einem Aufruf der **Func<Exception, bool>** - Methode zu einer Ausnahme, stellt **Handle()** seine Tätigkeit ein und leitet diese Ausnahme weiter.

Die Flexibilität und Komplexität eines **AggregateException**-Objekts sind nur bei einer zusammengesetzten Aufgabe relevant (siehe Abschnitt 15.4.8). Im aktuellen Beispiel kann dort nur eine ein-

zelne Ausnahme enthalten sein. Ist diese vom Typ `HarmlessException`, findet eine Behandlung statt, und das Programm wird normal weitergeführt:

```
Harmlos
Normales Ende
```

Im Falle einer `SeriousException` unterbleibt eine Behandlung, und das Programm wird von der CLR beendet.

Man kann sich über einen Ausnahmefehler bei der Auftragsbearbeitung informieren, ohne dass dabei die Ausnahme neu geworfen wird:

- Die **Task**-Eigenschaft **IsCanceled** informiert darüber, ob eine **OperationCanceledException** geworfen wurde.
- Die **Task**-Eigenschaft **IsFaulted** informiert darüber, ob eine beliebige andere Ausnahme geworfen wurde.

15.4.5.2 Unbeobachtete Ausnahmen

Tritt in einer Aufgabe eine unbehandelte Ausnahme auf, die nicht durch eines der zu Beginn des Abschnitts aufgelisteten **Task**-Mitglieder weitergeleitet wird, dann liegt eine so genannte *unbeobachtete* Ausnahme vor. Das passiert auch, wenn in einer per **Wait()** - Aufruf mit Timeout-Parameter untersuchten Aufgabe eine Ausnahme *nach* Ablauf der Wartezeit auftritt, z.B.:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class UnobservedException {
   static void Werfer() {
      Thread.Sleep(1000);
      throw new InvalidOperationException();
   }

   static void Main() {
      TaskScheduler.UnobservedTaskException +=
         (object sender, UnobservedTaskExceptionEventArgs eventArgs) => {
            Console.WriteLine("UnobservedTaskException-Handler");
            //eventArgs.SetObserved();
         };

      Task t = Task.Factory.StartNew(Werfer);

      try {
         t.Wait(50);
      } catch {
         Console.WriteLine("Diese Ausgabe wird nie erscheinen.");
      }
      Thread.Sleep(1000);
      t = null;
      GC.Collect();

      Console.WriteLine("\nDrücken Sie Enter zum Beenden.");
      Console.ReadLine();
   }
}
```

Eine unbeobachtete Ausnahme wird beim Abräumen des **Task**-Objekts per Garbage Collector von der CLR diagnostiziert. Nun wird das statische Ereignis **UnobservedTaskException** der Klasse **TaskScheduler** ausgelöst und bietet eine letzte Gelegenheit, sich um die Ausnahme zu kümmern (Griffiths 2013, S. 642f). Insbesondere kann die Ausnahme durch die Methode **SetObserved()** der

Klasse **UnobservedTaskExceptionEventArgs** als beobachtet deklariert und damit neutralisiert werden. Geschieht dies nicht, hängt laut MSDN-Dokumentation der weitere Ablauf von der ausführenden .NET-Version ab:[1]

- Die Version 4.0 beendet die Anwendung.
  Trotz eifrigen Bemühens ist es mir nicht gelungen, dieses Verhalten zu beobachten. Weil die .NET-Version 4.0 längst Geschichte ist, kann man mit dieser kleinen Unsicherheit leben.
- Die Version 4.5 ignoriert die unbeobachtete Ausnahme, wenn nicht diese Voreinstellung über eine Anwendungskonfiguration in der Datei **app.config** überschrieben wird:

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <runtime>
    <ThrowUnobservedTaskExceptions enabled="true"/>
  </runtime>
</configuration>
```

Das Ignorieren ist in der Regel *keine* sinnvolle Option. Also sollte ...

- ein **UnobservedTaskException**-Handler eingerichtet werden, der z.B. einen Log-Eintrag schreibt oder eine Mail verschickt,
- eventuell mit der beschriebenen Anwendungskonfiguration dafür gesorgt werden, dass die Anwendung mit der havarierten **Task** beendet wird.

Auch nach diesen Vorkehrungen unterscheidet sich die Reaktion auf eine unbeobachtete **Task**-Ausnahme noch deutlich von der in Abschnitt 15.1.8 beschriebenen Reaktion auf eine unbehandelte Ausnahme in einem Thread, weil die Reaktion erst im Rahmen der Finalisierung erfolgt, also mit erheblicher Verzögerung oder überhaupt nicht. Im nächsten Absatz wird daher vorgeschlagen, eine spezielle Fortsetzungsaufgabe zu definieren, um zeitnah auf unbeobachtete **Task**-Ausnahmen reagieren zu können.

Bei Verwendung der .NET 4.0 - Methoden für Fortsetzungsaufgaben (z.B. **ContinueWith()**, siehe Abschnitt 15.4.7.1) kann es leicht passieren, dass in einer Vorgängeraufgabe aufgetretene und unbehandelt gebliebene Ausnahmen übersehen werden. Eine Technik zur Vermeidung des Problems besteht darin, durch einen **ContinueWith()** - Aufruf mit dem **TaskContinuationOptions**-Wert **OnlyOnFaulted** eine spezielle Fortsetzungsaufgabe zu definieren, die ausschließlich nach dem Auftreten eines Ausnahmefehlers ausgeführt wird. Eine solche Fortsetzungsaufgabe, die sich um unbehandelte Ausnahmen kümmert, kann auch bei „Fire & Forget“ - Aufgaben sinnvoll sein, die ansonsten keinen Nachfolger brauchen.

15.4.6 Scheduler und Synchronisierungskontext

TPL-Aufgaben werden von einem **TaskSheduler** (dt.: Aufgabenplaner) verwaltet, und der voreingestellte Aufgabenplaner verwendet den Threadpool zur Durchführung der Aufgaben (Griffiths 2013, S. 641). Vor allem bei GUI-Anwendungen ist es oft erforderlich, dass eine Aufgabe im UI-Thread ausgeführt wird, weil sie Änderungen an Steuerelementen vornimmt. Über die statische **TaskSheduler**-Methode **FromCurrentSynchronization()** erhält man einen Aufgabenplaner mit dem Synchronisierungskontext des aufrufenden Threads. In der Regel kommen folgende Synchronisierungskontexte in Frage (Cleary 2014, S. 4)

- UI-Kontext
- Request-Kontext (bei einer ASP.NET - Anwendung)
- Threadpool-Kontext

---

[1] https://msdn.microsoft.com/de-de/library/jj160346(v=vs.110).aspx

Der UI- und der Request-Kontext sind Single-Thread - Kontexte, während der Pool-Kontext mit mehreren Threads arbeitet.

Aus der folgenden Anweisung mit der Deklaration und Initialisierung einer Instanzvariablen in einer Fensterklasse einer WPF-Anwendung resultiert ein **TaskSheduler**-Objekt, das mit dem UI-Thread assoziiert ist:

```
TaskScheduler uiTS = TaskScheduler.FromCurrentSynchronizationContext();
```

In der folgenden Ereignisbehandlungsmethode wird eine per Pool-Thread auszuführende Aufgabe gestartet. An das von **StartNew()** gelieferte **Task**-Objekt wird per **ContinueWith()** eine Fortsetzungsaufgabe angehängt (siehe Abschnitt 15.4.7.1), die im UI-Thread ausgeführt werden soll:

```
private void button_Click(object sender, RoutedEventArgs e) {
   label.Content = "";
   var task = Task.Factory.StartNew<double>(SampleMean);
   task.ContinueWith(t => { label.Content = t.Result.ToString(); }, uiTS);
}
```

15.4.7 Fortsetzungsaufgaben

15.4.7.1 Fortsetzung zu einer einzelnen Aufgabe

Über diverse Instanzmethoden der Klassen **Task** und **Task<TResult>** kann man eine Aufgabe erstellen, die gestartet wird, sobald eine andere Aufgabe abgeschlossen ist. In einem Beispiel verwenden wir die folgende Überladung der **Task**-Methode **ContinueWith()**,

public Task ContinueWith(Action<Task> *continuationAction***, TaskScheduler** *scheduler***)**

um auf die Erledigung einer Hintergrundberechnung durch ein **Task<double>** - Objekt eine Aufgabe folgen zu lassen, welche für die Ergebnisanzeige in einem WPF-Steuerelement sorgt. Wie über den zweiten **ContinueWith()** - Parameter vom Typ **TaskSheduler** dafür gesorgt wird, dass die Fortsetzungsaufgabe im UI-Thread abläuft, ist in Abschnitt 15.4.6 zu erfahren. Als erster **ContinueWith()** - Parameter wird ein Lambda-Ausdruck übergeben, welcher den Delegatentyp **Action<Task<double>>** realisiert (Rückgabetyp **void**, ein Parameter vom Typ **Task<double>**):

```
Task<double> task = Task.Factory.StartNew<double>(SampleMean);
task.ContinueWith(t => { label.Content = t.Result.ToString(); }, uiTS);
```

Ist der Vorgänger beim **ContinueWith()** - Aufruf bereits beendet, startet die Fortsetzungsaufgabe unmittelbar.

Eine Fortsetzungsaufgabe wird per Voreinstellung *auf jeden Fall* ausgeführt, also unabhängig davon, ob die vorherige Aufgabe erfolgreich beendet wurde, mit einem Ausnahmefehler gescheitert ist oder abgebrochen wurde. Bei einigen **ContinueWith()** - Überladungen kann durch einen Parameter vom Typ **TaskContinuationOptions** dafür gesorgt werden, dass die Fortsetzungsaufgabe nur dann ausgeführt wird, wenn der Vorgänger erfolgreich war (**OnlyOnRanToCompletion**), an einem Ausnahmefehler gescheitert ist (**OnlyOnFaulted**) bzw. abgebrochen wurde (**OnlyOnCanceled**). Neben den Fortsetzungsbedingungen enthält die Enumeration **TaskContinuationOptions** noch weitere Werte, u.a.:

- Alle Werte der in Abschnitt 15.2.2 vorgestellten Enumeration **TaskCreationOptions**
- **ExecuteSynchronously**
  Die Fortsetzungsaufgabe wird im selben Thread ausgeführt wie der Vorgänger. Das spart Verwaltungsaufwand, ist aber nur bei *kleinen* Fortsetzungsaufgaben empfehlenswert (Griffiths 2013, S. 640).

Um eine unbehandelte Ausnahme in einer Vorgängeraufgabe muss man sich explizit kümmern, damit sie nicht übersehen wird, was in obigem Beispiel geschehen würde. Zur Lösung des Problems

kann man durch einen **ContinueWith()** - Aufruf mit dem **TaskContinuationOptions**-Wert **OnlyOnFaulted** eine spezielle Fortsetzungsaufgabe definieren, die nur nach dem Auftreten eines Ausnahmefehlers ausgeführt wird, z.B.:

```
task.ContinueWith(t => { label.Content = t.Exception.InnerException.Message; },
                  new System.Threading.CancellationToken(),
                  TaskContinuationOptions.OnlyOnFaulted, uiTS);
```

Alternativ kann man in einer Fortsetzungsaufgabe den Status des Vorgängers abfragen, z.B.:

```
task.ContinueWith(t => {
   if (t.IsFaulted)
      label.Content = t.Exception.InnerException.Message;
   else
      label.Content = t.Result.ToString();
}, uiTS);
```

Seit .NET 4.5 bietet die **Task**-Methode **GetAwaiter()** eine Möglichkeit zur Vereinbarung einer Fortsetzungsaufgabe, die im Vergleich zu **ContinueWith()** zwei gravierende Vorteile bringt:

- Eine bei der Vorgängerbearbeitung aufgetretene und nicht behandelte Ausnahme wird beim Zugriff auf das Vorgängerergebnis mit **GetResult()** automatisch neu geworfen, kann also nicht übersehen werden. Freundlicherweise unterbleibt dabei die Verpackung in eine **AggregateException**, was die Behandlung erleichtert.
- Ein Synchronisierungskontext (vgl. Abschnitt 15.4.6) wird automatisch in die Fortsetzungsaufgabe übertragen, was z.B. die Aktualisierung von Steuerelementen nach Beendigung einer Hintergrundaufgabe vereinfacht.

Diese Vorteile bestehen übrigens auch beim asynchronen Programmieren mit den seit C# 5.0 verfügbaren Schlüsselwörtern **async** und **await** (siehe Abschnitt 15.5), weil dabei die Aufgabenverkettung aus .NET 4.5 zum Einsatz kommt (Albahari & Albahari 2015, S. 585). Wer die Methode **GetAwaiter()** in eigenem Quellcode benutzt, muss den folgenden Hinweis in der MSDN-Dokumentation ignorieren:

> This method is intended for compiler use rather than for use in application code.

Bevor die Sache zu abstrakt wird, stellen wir das obige Beispiel auf die neue Technik um. Ein Aufruf der **Task**-Methode **GetAwaiter()** liefert ein Objekt der Klasse **TaskAwaiter<TResult>** (aus dem Namensraum **System.Runtime.CompilerServices**):

```
var awaiter = task.GetAwaiter();
```

Ein solches Objekt kann über seine **OnCompleted()** - Methode beauftragt werden, nach Beendigung der Aufgabe die per Parameter übergebene Delegatenmethode auszuführen, z.B.:

```
awaiter.OnCompleted(() => {
   label.Content = awaiter.GetResult().ToString();
});
```

Im Beispiel wird die Delegatenmethode per Lambda-Ausdruck realisiert, und dort kann die Referenzvariable `awaiter` aus der umgebenden Methode dazu verwendet werden, per **GetResult()** das Ergebnis der gerade erledigten Aufgabe anzufordern. Weil **OnCompleted()** den Synchronisierungskontext von `button_Click()` verwendet, ist ein Zugriff auf UI-Steuerelemente erlaubt.

Die von **GetResult()** zu erwartenden Ausnahmen sollten behandelt werden, was (im Unterschied zu der oben beschriebenen **ContinueWith()** - Lösung) mit intuitiver Syntax gelingt:

```
awaiter.OnCompleted(() => {
   try {
      label.Content = awaiter.GetResult().ToString();
   } catch (Exception ex) {
      label.Content = ex.Message;
   }
});
```

15.4.7.2 Auf mehrere Aufgaben warten

Über die **TaskFactory**-Methode **ContinueWhenAll**() kann man eine Aufgabe erstellen, die gestartet wird, sobald aus einem Array von Aufgaben alle Elemente abgeschlossen sind. Um die Methode zu demonstrieren, setzen wir unser Beispiel zur Untersuchung von Zufallszahlen fort und erzeugen einen Array mit Elementen vom Typ **Task<double>**, die jeweils einen Stichprobenmittelwert berechnen:

```
var sampleMeanTaskArray = new Task<double>[ANZ];
for (int i = 0; i < ANZ; i++)
   sampleMeanTaskArray[i] = Task.Factory.StartNew<double>(SampleMean, i);
```

Dank moderner PC-Rechenleistung und TPL sollte es kein Problem darstellen, z.B. aus 10.000 Stichproben vom jeweiligen Umfang 10.000 parallel den Mittelwert zu berechnen. Wenn alle Stichprobenmittel vorliegen, sollen diese in einer Fortsetzungsaufgabe untersucht werden. Die zugrunde liegende Methode `Summary()` erfüllt den Delegatentyp **Action<Task<double>[]>**:

```
static void Summary(Task<double>[] sampleMeanTaskArray) {
   double x;
   int n = sampleMeanTaskArray.Length;
   double sum = 0.0;
   double sq = 0.0;
   for (int j = 0; j < n; j++) {
      x = sampleMeanTaskArray[j].Result;
      sum += x;
      sq += x*x;
   }
   double mean = sum / n;
   double variance = (sq - (sum * sum / n)) / (n - 1);
   double sd = Math.Sqrt(variance);
   Console.WriteLine("\nAnzahl = {0,25:d}\nUmfang = {1,25:d}\nMittel = {2,25:f7}"+
      "\nVarianz = {3,24:f7}\nStandardabweichung = {4,13:f7}", n, SG, mean, variance, sd);
}
```

Sie erhält per Parameter den Array mit den Aufgaben zum Berechnen der Stichprobenmittelwerte und ermittelt daraus das Gesamtmittel sowie die Varianz und die Standardabweichung der Stichprobenmittelwerte. So lässt sich z.B. beobachten, wie die Standardabweichung der Stichprobenmittelwerte (also der geschätzte Standardfehler) von der Anzahl und vom Umfang der Stichproben abhängt.

Aus dem folgenden Aufruf der **TaskFactory**-Methode **ContinueWhenAll<double>()** resultiert eine neue Aufgabe, welche asynchron die Methode `Summary()` ausführt, sobald alle Aufgaben im Array `sampleMeanTaskArray` erledigt sind:

```
Task sTsk = Task.Factory.ContinueWhenAll<double>(sampleMeanTaskArray,Summary);
// Parallele Arbeiten erledigen
sTsk.Wait();
```

Eine Analyse von 10.000 Stichproben vom Umfang 1000 zeigt, dass die Mittelwerte aus derartigen Stichproben nur noch sehr wenig variieren (Standardabweichung: 0,009):

```
Anzahl =                    10000
Umfang =                     1000
Mittel =                0,5000619
Varianz =               0,0000845
Standardabweichung =    0,0091901

Zeit im Sek.: 0,1226099
```

Über die **TaskFactory**-Methode **ContinueWhenAny**() kann man eine Aufgabe erstellen, die gestartet wird, sobald *irgendein* Element aus einem Array mit Aufgaben abgeschlossen ist.

15.4.8 Zusammengesetzte Aufgaben

Wird aus einem aktiven Auftrag heraus eine neue Aufgabe erstellt, so stehen diese in keiner besonderen Relation zueinander.[1] Mit einigen Überladungen der **TaskFactory**-Methode **StartNew()** lässt sich aber ein „Kindauftrag“ erstellen, indem der Parameter vom Enumerationstyp **TaskCreationOptions** einen Wert mit **AttachedToParent**-Flag erhält.[2] Das hat für den Elternauftrag folgende Konsequenzen (siehe Griffiths 2013, S. 645):

- Der Elternauftrag wird nur dann als abgeschlossen betrachtet, wenn alle Kindaufträge abgeschlossen sind.
- Tritt in mindestens einem Kindauftrag eine unbehandelte Ausnahme auf, dann wirft der Elternauftrag eine **AggrgateException** mit allen unbehandelten Ausnahmen aus Kindaufträgen, wenn z.B. auf den Elternauftrag gewartet oder auf sein Ergebnis zugegriffen wird (siehe Abschnitt 15.4.5).

Auch mit den statischen Methoden **WhenAll**() und **WhenAny**() der Klasse **Task** lässt sich eine Metaaufgabe erstellen, deren Staus von mehreren Detailaufgaben abhängt. Es sind u.a. die folgenden Überladungen vorhanden:[3]

- **public static Task WhenAll(params Task[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *alle* Elemente im Parameter-Array abgeschlossen sind.
- **public static Task<TResult[]> WhenAll<TResult>(params Task<TResult>[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *alle* Elemente im Parameter-Array abgeschlossen sind. Sie enthält einen Array mit den Ergebnissen der Elementaufgaben.
- **public static Task<Task> WhenAny(params Task[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *irgendein* Element im Parameter-Array abgeschlossen ist. Die abgeschlossene Elementaufgabe ist das Ergebnis der Metaaufgabe.
- **public static Task<Task<TResult>> WhenAny<TResult>(params Task<TResult>[]** *tasks***)**
  Die resultierende Metaaufgabe ist abgeschlossen, wenn *irgendein* Element im Parameter-Array abgeschlossen ist. Das Ergebnis der Metaaufgabe ist die abgeschlossene Elementaufgabe(, die wiederum ein Ergebnis vom Typ **TResult** enthält).

---

[1] Seit .NET 4.0 wird eine „sekundäre“ Aufgabe allerdings bevorzugt vom selben logischen Prozessorkern abgearbeitet wie die initiierende Aufgabe (siehe Abschnitt 15.2.2).

[2] Der Enumerationstyp **TaskCreationOptions** ist durch das **FlagsAttribute** dekoriert, so dass ein Wert für mehrere binäre Einzelattribute steht (siehe Abschnitt 13.5.1).

[3] Mit dem Schlüsselwort **params** wird ein Serienparameter deklariert (siehe Abschnitt 4.3.1.3.4), man kann also beim Aufruf der Methoden ...

- einen **Task**-Array als Aktualparameter übergeben,
- oder beliebig viele einzelne **Task**-Objekte auflisten.

Eine per **WhenAll()** erstellte Metaaufgabe wirft ggf. eine **AggrgateException**, die alle in Detailaufgaben aufgetretenen unbehandelten Ausnahmen enthält. Demgegenüber endet eine per **WhenAny()** erstellte Metaaufgabe stets mit dem Status **RanToCompletion**. Das gilt auch dann, wenn ihr Ergebnis, also die zuerst beendete Aufgabe den Status **Faulted** oder **Cancelled** besitzt. Man muss also den Status der Ergebnisaufgabe überprüfen.

Statt zu einer **WhenAll()** - oder **WhenAny()** - Rückgabe eine Fortsetzungsausgabe zu definieren, kann man die **TaskFactory**-Methoden **ContinueWhenAll**() und **ContinueWhenAny**() verwenden (siehe Abschnitt 15.4.7.2).

15.4.9 Aufgaben abbrechen

Mit Hilfe der TPL-Typen **CancellationTokenSource** und **CancellationToken** kann eine Aufgabe aufgefordert werden, ihre Tätigkeit einzustellen. Ein Objekt der Klasse **CancellationTokenSource**

```
CancellationTokenSource cts = new CancellationTokenSource();
```

stellt über seine **Token**-Eigenschaft eine Instanz vom Strukturtyp **CancellationToken** als „Teilnahmekärtchen“ am Terminierungsverfahren zur Verfügung:

```
CancellationToken ct = cts.Token;
```

Dieses Token kann einer neuen Task beim Starten zugeordnet werden, z.B.:

```
Task task = Task.Factory.StartNew(Punktieren, ct, ct);
```

Im Beispiel kommt die folgende **StartNew()** - Überladung zum Einsatz:

public Task StartNew(Action<Object> *action***, Object** *state***, CancellationToken** *token***)**

mit drei Parametern zum Einsatz:

- **Action<Object>** *action*
  Delegatenmethode mit Parameter vom Typ **Object**
- **Object** *state*
  Beim Aufruf der Delegatenmethode wird dieses Parameterobjekt übergeben.
- **CancellationToken** *token*
  Die Aufgabe kann über dieses Token unterbrochen werden.

Um die mit einem Token ausgestatteten Aufgaben zu unterbrechen, ruft man die **CancellationTokenSource** - Methode **Cancel()** auf, z.B.:

```
cts.Cancel();
```

Daraufhin wird die **IsCancellationRequested**-Eigenschaft der **CancellationToken**-Instanz auf den Wert **true** gesetzt.

Indem beim Abbrechen von Aufgaben *zwei* Instanzen beteiligt sind (vom Strukturtyp **CancellationToken** bzw. aus der Klasse **CancellationTokenSource**), kann man folgende Berechtigungen getrennt vergeben:

- Das Recht zum Erstellen einer Task, die vom Abbrechen betroffen ist
- Das Recht, das Abbrechen zu veranlassen

Seit .NET 4.5 besitzt die Klasse **CancellationTokenSource** eine Konstruktorüberladung mit Timeout-Parameter:

public CancellationTokenSource(int *millisecondsTimeout***)**

Mit dem zugehörigen Token gestartete Aufgaben erhalten automatisch ein Abbruchsignal, wenn seit dem Konstruktor-Aufruf die angegebene Wartezeit vergangen ist.

Um das **CancellationToken** für eine Aufgabe zu setzen, finden eine geeignete Überladungen ...

- in der **TaskFactory**-Methodengruppe **StartNew()**,
- in der statischen **Task**-Methodengruppe **Run()**,
- unter den **Task**-Konstruktoren.

Ist das Terminierungssignal schon vor Beginn der Bearbeitung gesetzt, wird die Aufgabe durch die TPL-Logik vom Zustand **WaitingToRun** sofort in den Zustand **Canceled** befördert, ohne je den Zustand **Running** erlebt zu haben.

Damit im Beispiel die Delegatenmethode vom Typ **Action<Object>** das Terminierungssignal kooperativ beachten kann, erhält sie per Parameter eine Kopie der **CancellationToken**-Instanz (verpackt per Autoboxing):

```
static void Punktieren(Object token) {
   CancellationToken ct = (CancellationToken) token;
   Console.WriteLine("Aufgabe in Bearbeitung");
   for (int i = 0; i < MAXIT; i++) {
      Thread.Sleep(500);
      Console.Write(".");
      if (ct.IsCancellationRequested) {
         Console.WriteLine("\nSignal zum Abbrechen erhalten\n");
         ct.ThrowIfCancellationRequested();
      }
   }
}
```

Die Methode prüft regelmäßig, ob die **IsCancellationRequested**-Eigenschaft der zuständigen **CancellationToken**-Instanz den Wert **true** besitzt. In diesem Fall stellt sie durch einen Aufruf der **CancellationToken**-Methode **ThrowIfCancellationRequested()** ihre Tätigkeit ein. Wie der Methodenname andeutet, wird eine Ausnahme geworfen, wobei die Klasse **OperationCanceledException** Verwendung findet. Dem Ausnahmeobjekt wird das **CancellationToken** der Aufgabe mit auf den Weg gegeben, so dass der Ausnahmefänger die Identität der Aufgabe feststellen kann, die kooperativ auf das Terminierungssignal reagiert hat. Diese Aufgabe befindet sich anschließend im Zustand **Cancelled**, so dass ihre Eigenschaft **IsCanceled** den Wert **true** besitzt.

In der folgenden **Main()** - Methode wird eine Aufgabe basierend auf der Methode `Punktieren()` gestartet, und der Benutzer kann den Zeitpunkt des Abbruchs bestimmen. Anschließend wird die **Wait()** - Methode des **Task**-Objekts aufgerufen, um das **AggregateException**-Ausnahmeobjekt mit Detailinformationen über das Terminierungsverfahren abzufangen:

```
static void Main() {
   CancellationTokenSource cts = new CancellationTokenSource();
   CancellationToken ct = cts.Token;

   Task task = Task.Factory.StartNew(Punktieren, ct, ct);
   Console.WriteLine("Aufgabe gestartet. Zustand:     " + task.Status +
                     ".\nStoppen mit Enter\n");

   Console.ReadLine();
   Console.WriteLine("\nZustand der Aufgabe vor Cancel: " + task.Status);
   cts.Cancel();
   Console.WriteLine("Signal zum Abbrechen gesetzt.\n");

   try {
      task.Wait();
   } catch (AggregateException ae) {
      foreach (Exception ie in ae.InnerExceptions)
         Console.WriteLine("Message der inneren Ausnahme:   " + ie.Message);
   }
   Console.WriteLine("Aufgabe abgebrochen. Zustand:   " + task.Status);
}
```

Beim folgenden Programmeinsatz hatte sich der Benutzer schon nach ca. einer Minute an den Punkten satt gesehen:

```
Aufgabe gestartet. Zustand:     WaitingToRun.
Stoppen mit Enter

Aufgabe in Bearbeitung
.....................................................

Zustand der Aufgabe vor Cancel: Running
Signal zum Abbrechen gesetzt.

.
Signal zum Abbrechen erhalten

Message der inneren Ausnahme:   Eine Aufgabe wurde abgebrochen.
Aufgabe abgebrochen. Zustand:   Canceled
```

Per **CancellationToken** lassen sich auch *synchrone* Methoden abbrechen, wobei der **Cancel()** - Aufruf natürlich aus einem anderen Thread stammen muss. Diese Technik wird auch bei einigen FCL-Methoden unterstützt. Verwendet man z.B. bei der **Task**-Methode **Wait()** eine Überladung mit **CancellationToken**-Parameter, dann endet die Methode ...

- regulär, wenn die abzuwartende Aufgabe abgeschlossen ist,
- mit einer Ausnahme vom Typ **OperationCanceledException**, wenn für das Token ein **Cancel()** - Aufruf erfolgt ist.

15.4.10 Parallele Aufgaben

Über die statische Methode **Invoke()**der Klasse **Parallel** aus dem Namensraum **System.Threading.Tasks** startet man eine Serie von Aktionen, wobei die CLR über das Ausmaß der Parallelisierung entscheidet. Man übergibt die Aufgaben per Serienparameter vom Typ **Action[]**, wobei **Action** ein Delegatentyp mit folgender Signatur ist:

public delegate void Action()

Im folgenden Programm erfüllt die Methode `RandomSum()`, die eine Summe von 10 Millionen Pseudozufallszahlen berechnet und zusammen mit dem ausführenden Thread auf der Konsole protokolliert, den Delegatentyp **Action**:

```
using System;
using System.Threading;
using System.Threading.Tasks;

class ParallelInvoke {
   const int ANZ = 16;
   const int SG = 10000000;

   static void SampleMean() {
      Random zzg = new Random();
      double erg = 0.0;
      for (int j = 0; j < SG; j++)
         erg += zzg.NextDouble();
      Console.WriteLine("Stichprobenmittel " + (erg / SG) + " berechnet von Thread " +
                        Thread.CurrentThread.ManagedThreadId);
   }
```

```
    static void Main() {
       Action[] av = new Action[ANZ];
       for (int i = 0; i < ANZ; i++)
          av[i] = new Action(SampleMean);
       long start = DateTime.Now.Ticks;
       Parallel.Invoke(av);
       Console.WriteLine("Benötigte Zeit in Sek.: " +
                         (DateTime.Now.Ticks - start) / 1.0e7);
    }
}
```

In der Methode **Main()** wird ein **Action**-Array mit Delegatenobjekten gefüllt, deren Aufgabe darin besteht, die Methode `SampleMean()` auszuführen. Per **Invoke()** - Aufruf werden **Task**-Objekte erstellt und durch eine geeignete Anzahl von Threads ausgeführt. Beim anschließend protokollierten Programmeinsatz auf einem Rechner mit vier virtuellen Kernen (Intel Core i3 mit Hyperthreading) sind 16 **Task**-Objekte auf fünf Threads verteilt worden:

```
Stichprobenmittel 0,500200598811819 berechnet von Thread 4
Stichprobenmittel 0,499824705870593 berechnet von Thread 1
Stichprobenmittel 0,500200598811819 berechnet von Thread 3
Stichprobenmittel 0,500200598811819 berechnet von Thread 6
Stichprobenmittel 0,500040970890621 berechnet von Thread 5
Stichprobenmittel 0,499832626971808 berechnet von Thread 1
Stichprobenmittel 0,499832626971808 berechnet von Thread 4
Stichprobenmittel 0,499916299050566 berechnet von Thread 6
Stichprobenmittel 0,499916299050566 berechnet von Thread 3
Stichprobenmittel 0,499930871129361 berechnet von Thread 5
Stichprobenmittel 0,49990912721051 berechnet von Thread 1
Stichprobenmittel 0,49990912721051 berechnet von Thread 4
Stichprobenmittel 0,500205799289236 berechnet von Thread 3
Stichprobenmittel 0,500205799289236 berechnet von Thread 6
Stichprobenmittel 0,500031607371117 berechnet von Thread 5
Stichprobenmittel 0,499968669174211 berechnet von Thread 1
Benötigte Zeit in Sek.: 1,0348991
```

Diese Zahl ergibt sich aus dem Vergleich der Kennungen, die man über die **Thread**-Eigenschaft **ManagedThreadId** in Erfahrung bringt.

Der **Invoke()** - Aufruf endet, wenn alle Aufgaben erledigt sind. Es ist keine Reihenfolge der Bearbeitung garantiert.

Muss eine Aufgabe für eine Folge von Indexwerten, aber nicht unbedingt in der natürlichen Reihenfolge, ausgeführt werden, erlaubt die statische **Parallel**-Methode **For()** eine Parallelisierung. Es ist eine Methode zu definieren, die den Delegatentyp **Action<int>** erfüllt, also von folgender Bauart sein muss:

void Action (**int** *index*)

Ein auf dieser Methode basierendes Delegatenobjekt wird zusammen mit einem Start- und einem Endwert für den Schleifenindex als Parameter an die Methode **For()** übergeben, z.B.:

```
Parallel.For(1, ANZ, SampleMean);
```

Die Delegatenmethode wird für jeden Indexwert aufgerufen und erhält diesen als Aktualparameter. Man darf sich aber keinesfalls darauf verlassen, dass die Aufrufe in der Reihenfolge der Indexwerte stattfinden.

Analog erstellt die **Parallel**-Methode **ForEach()** eine Serie von **Task**-Objekten aus einer Methode und passenden Argumenten in einer Kollektion.

Die bei der Aufgabenbearbeitung in den beteiligten Threads aufgetretenen unbehandelten Ausnahmen werden in einem Objekt der Klasse **AggregateException** gesammelt, das ggf. von **Invoke()**, **For()** bzw. **ForEach()** geworfen wird und im Rahmen einer **try**-Anweisung abgefangen werden

sollte. Zur Analyse ruft man die **AggregateException**- Methode **Handle()** auf (siehe Abschnitt 15.4.5).

15.5 C# - Sprachunterstützung für asynchrones Programmieren

Durch die in C# 5.0 eingeführten Schlüsselwörter **async** und **wait** wird die Nutzung asynchroner Programmiertechniken erleichtert, um die ihre Verbreitung zu fördern. Programmierer müssen beim Umstieg von synchronem auf asynchronen Code nur wenige Änderungen vornehmen, während der Compiler für die parallele Ausführung sorgt.

Ein Hauptanwendungsfeld sind die Ereignisbehandlungsmethoden von GUI-Anwendungen, wobei folgende Ziele zu realisieren sind:

- Die UI-Threads sollen wenig belastet werden.
- Die verfügbaren Multithreading-Ressourcen sollen gut ausgenutzt werden.
- Von Hintergrund-Threads ermittelte Ergebnisse sollen bequem in UI-Threads (also in Steuerelemente) kanalisiert werden.
- Die Fehlerbehandlung soll nicht über mehrere Methoden verteilt, sondern in einer Methode konzentriert werden.
- Die zu verwendende Syntax soll sich nur wenig von der synchronen Variante unterscheiden.

15.5.1 Die Schlüsselwörter async und await

Durch den Modifikator **async** wird eine Methode als *asynchron* deklariert. Sie kann dann den Operator **await** verwenden, um einen Suspendierungspunkt zu setzen. Der unäre **await**-Operator hat als Operanden typischerweise eine Aufgabe, also ein Objekt vom Typ **Task** oder **Task<TResult>**.[1] Per **await** wird der Compiler darüber informiert, dass die Methode nicht weiterarbeiten kann, bevor die Aufgabe erfolgreich beendet wurde. Ist die abzuwartende Aufgabe noch nicht erledigt, wird die Kontrolle an den Aufrufer der asynchronen Methode zurückgegeben. Die asynchrone Methode ist aber nicht beendet, sodass z.B. vorhandene **finally**-Blöcke jetzt noch nicht ausgeführt werden. Außerdem wird der aktuelle Thread nicht blockiert. Stellt der **await**-Operator fest, dass die abzuwartende Aufgabe bereits erledigt ist, macht die asynchrone Methode weiter, ohne die Kontrolle an den Aufrufer abzugeben.

Der auf den **await**-Operator folgende Rest der asynchronen Methode wird bei einem Kontrollwechsel zu einer Fortsetzungs-Callback-Methode, die nach Beendigung der abzuwartenden Aufgabe ausgeführt wird. Per Voreinstellung wird dabei derselbe Thread verwendet, in dem die synchrone Methode begonnen wurde.

In der Regel enthält eine asynchrone Methode einen oder mehrere **await**-Operatoren. Während die Verwendung das **await**-Operators in einer nicht mit **async** dekorierten Methode verboten ist und einen Compiler-Fehler nach sich zieht, hat eine **async**-Methode ohne **await**-Operator lediglich eine Compiler-Warnung zur Folge.

Erlaubte Rückgabetypen für eine **async**-Methode sind:

[1] Ein „awaitable“ ist in der Regel ein **Task**- bzw. **Task<T>** - Objekt, doch ist nur das Implementieren bestimmter Schnittstellen gefordert (siehe Griffiths 2013, S. 662ff).

- **Task**
  Dieser Typ eignet sich, wenn zu einer vormals synchronen Methode mit dem Rückgabetyp **void** eine asynchrone Variante erstellt wird. Damit kann die asynchrone Methode selbst einen **await**-Operanden liefern und zum abzuwartenden Auftrag werden.
- **Task<TResult>**
  Dieser Typ eignet sich, wenn zu einer vormals synchronen Methode mit dem Rückgabetyp **TResult** eine asynchrone Variante erstellt wird. Im Vergleich zum Rückgabetyp **Task** hat der Aufrufer die zusätzliche Option, ein Ergebnis entgegen zu nehmen.
- **void**
  Der Rückgabetyp **void** ist bei **async**-Methoden vor allem deshalb erlaubt, um Ereignisbehandlungsmethoden, für die meist der Rückgabetyp **void** vorgeschrieben ist, als **async** deklarieren zu können. Allerdings kann eine solche **async**-Methode keinen **await**-Operanden liefern und nicht zum abzuwartenden Auftrag werden.

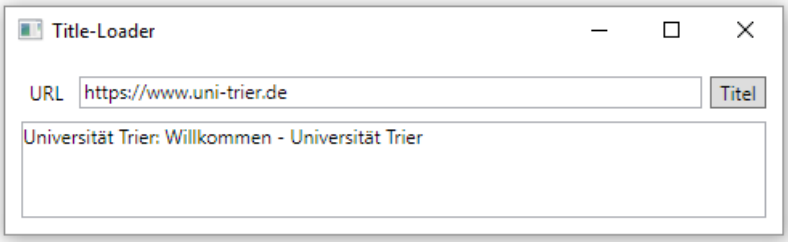
Die **Main()** - Methode eines Programms darf auch dann nicht als asynchron deklariert werden, wenn sie den Rückgabetyp **void** hat.

Für eine asynchrone Methode mit dem Rückgabetyp **Task** ist keine **return**-Anweisung mit Rückgabe zu verwenden. Von den **return**-Anweisungen einer asynchronen Methode mit dem Rückgabetyp **Task<TResult>** ist ein Ausdruck vom Typ **TResult** zu liefern. Die folgende Methode `GetHtmlTitleAsync()` ermittelt zu einer Webadresse das **title**-Element des dort anrufbaren HTML-Codes und liefert es als Zeichenfolge:[1]

```
private async Task<String> GetHtmlTitleAsync(String url) {
   using (var client = new HttpClient()) {
      String s = await client.GetStringAsync(url).ConfigureAwait(false);
      Match m = Regex.Match(s, @"<title>\s*(.+?)\s*</title>");
      if (m.Success)
         return m.Groups[1].Value;
      else
         return null;
   }
}
```

Im Beispiel wird ein **String**-Objekt (oder **null**) geliefert, obwohl **Task<String>** als Rückgabetyp deklariert ist.

Das Beispiel gehört zu einem Programm, das die Eingabe einer Webadresse erlaubt und auf Knopfdruck das zugehörige **title**-Element liefert:



Auch die Ereignisbehandlungsmethode zum Befehlsschalter des Programms ist asynchron:

[1] Der reguläre Ausdruck stammt von der Webseite https://www.dotnetperls.com/title-html.

```
private async void buttonTitle_Click(object sender, RoutedEventArgs e) {
   String temp = null;
   try {
      Task<String> ts = GetHtmlTitleAsync(textBox.Text);

      temp = textBlock.Text;
      textBlock.Text = "Bitte warten ....";
      buttonTitle.IsEnabled = false;

      String s = await ts;
      if (s != null && s.Length > 0)
         textBlock.Text = s;
   } catch (Exception ex) {
      textBlock.Text = temp;
      MessageBox.Show(this, ex.ToString(), ex.Message);
   } finally {
      buttonTitle.IsEnabled = true;
   }
}
```

Sie demonstriert eine typische Vorgehensweise:

- Die im UI-Thread ausgeführte Methode `buttonTitle_Click()` startet die asynchrone Methode `GetHtmlTitleAsync()` (siehe oben), deren Anfang ebenfalls im UI-Thread ausgeführt wird.
- `GetHtmlTitleAsync()` startet einen per Threadpool auszuführenden Webzugriff durch die **HttpClient**-Methode `GetStringAsync()` und wartet per **await**-Operator auf dessen Ergebnis, sodass die Kontrolle zur Methode `buttonTitle_Click()` zurückkehrt.
- Hier werden (im UI-Thread) einige anstehende Arbeiten erledigt:
  - Der bisherige Inhalt eines **TextBlock**-Steuerelements wird gesichert und durch eine Warteinstruktion ersetzt.
  - Der Befehlsschalter wird deaktiviert, damit der Benutzer während der Auftragsbearbeitung keinen neuen Auftrag erteilen kann.
- Wenn keine vom Auftragsergebnis unabhängigen Arbeiten anstehen, können die Auftragskreation und der **await**-Ausdruck auch in einer Anweisung stehen:
  ```
  String s = await GetHtmlTitleAsync(textBox.Text);
  ```
- Jetzt benötigt `buttonTitle_Click()` das Resultat von `GetHtmlTitleAsync()` und wartet daher auf die zugehörige Aufgabe.
- Der Kontrollfluss wird an den Aufrufer von `buttonTitle_Click()` abgegeben, so dass weitere Methoden aus der Nachrichtenwarteschlange des UI-Threads abgearbeitet werden können. Es ist z.B. möglich, das Fenster des Programms zu verschieben.
- Nachdem `GetStringAsync()` seine Aufgabe erledigt hat, wird `GetHtmlTitleAsync()` fortgesetzt. Weil für den **await**-Operanden mit `ConfigureAwait(false)` angeordnet worden war, dass der Synchronisierungskontext nicht konserviert werden soll (siehe Abschnitt 15.5.2),
  ```
  String s = await client.GetStringAsync(url).ConfigureAwait(false);
  ```
  läuft der Methodenrest *nicht* im UI-Thread ab, sondern in einem Pool-Thread.
- Nach Rückkehr von `GetHtmlTitleAsync()` wird der Rest von `buttonTitle_Click()` im UI-Thread ausgeführt und zeigt den ermittelten Webseiten-Titel an.
- Alle Ausnahmen können in `buttonTitle_Click()` gemeinsam behandelt werden, auch die von `GetHtmlTitleAsync()` geworfenen Ausnahmen (siehe Abschnitt 15.5.3). Die try-Anweisung in `buttonTitle_Click()` besitzt einen **finally**-Block, um den Befehlsschalter zu reaktivieren.

Weitere Details zu **async** und **await**:

- Per Konvention endet der Name einer asynchronen Methode mit **Async**, sofern es sich nicht um eine Ereignisbehandlungsmethode handelt.
- Der Modifikator **async** hat *keinen* Einfluss auf die Signatur einer Methode.
- In einer mit **async** dekorierten Methode darf **await** nicht als Bezeichner verwendet werden, anderenorts aber schon. Es ist (wie auch **async**) ein kontextbezogenes Schlüsselwort (vgl. Abschnitt 3.1.5).
- Auch anonyme Methoden (mit traditioneller oder Lambda-Syntax) lassen sich asynchron definieren (siehe Griffiths 2013, S. 662).
- In C# 5.0 durfte **await** nicht in **catch**- und **finally**-Blöcken aufgerufen werden; in C# 6.0 ist dies erlaubt.
- In asynchronen Methoden sind keine **ref**- oder **out**-Parameter erlaubt, denn die Methode kehrt ja in der Regel schon zum Aufrufer zurück, bevor sie vollständig ausgeführt worden ist.
- Wie bei jeder anderen Nutzung von Threadpools muss beachtet werden, dass hier Hintergrund-Threads tätig sind, die abgebrochen werden, wenn der letzte Vordergrund-Thread eines Programms endet (vgl. Abschnitt 15.2).

Bislang haben wir uns mit dem Warten auf *eine* asynchron ausgeführte Aufgabe beschäftigt. Über die statischen Methoden **WhenAll**() und **WhenAny**() der Klasse **Task** kann man eine Metaaufgabe erstellen und somit *mehrere* Aufgaben auf die Beobachtungsliste setzen (siehe Abschnitt 15.4.8).

In der folgenden Tabelle nach Cleary (2012) sind Techniken aus der Task Parallel Library aufgelistet, die in neuem Code durch **async** & **await** ersetzt werden sollten:

TPL	**async** & **await**	**Beschreibung**
task.Wait()	**await task**	Auf die Fertigstellung einer Aufgabe warten
task.Result	**await task**	Auf das Ergebnis einer Aufgabe zugreifen
Task.WaitAll()	**await Task.WhenAll()**	Warten, bis *alle* Aufgaben aus einem **Task**-Array abgeschlossen sind
Task.WaitAny()	**await Task.WhenAny()**	Warten, bis *irgendeine* Aufgabe aus einem **Task**-Array abgeschlossen ist

15.5.2 Thread-Affinität

Eine asynchrone Methode startet (wie jede andere Methode) synchron im aktuellen Thread. Wenn **await** auf ein **Task**- bzw. **Task<TResult>** - Objekt wartet, wird per Voreinstellung der beim Starten der asynchronen Methode aktuelle Synchronisierungskontext (der Wert von **SynchronizationContext.Current**) konserviert und (bei einem Wert ungleich **null**) vor der Ausführung einer Fortsetzungsmethode restauriert. Ist der Wert von **SynchronizationContext.Current** vor **await** ungleich **null** (z.B. bei einer asynchronen Ereignisbehandlungsmethode in einem beliebigen GUI-Framework), dann läuft die Fortsetzungsmethode im selben Synchronisierungskontext. Ist der Wert von **SynchronizationContext.Current** vor **await** hingegen gleich **null** (z.B. bei einer Konsolenanwendung), dann läuft die Fortsetzungsmethode in einem Pool-Thread, vermutlich meist im selben, der die abgewartete Aufgabe erledigt hat.

In einer WPF-Ereignisbehandlungsmethode sorgt die Thread-Restaurierung dafür, dass die nach einer asynchron erledigten Aufgabe fälligen Änderungen von Steuerelementen im UI-Thread ablaufen. Ein Vergleich mit dem Abschnitt 15.2.3 über die Verwendung von Worker-Threads in WPF-Anwendungen belegt, dass die asynchrone Programmierung mit **async** & **await** speziell bei der GUI-Aktualisierung erheblich bequemer ist.

Wenn allerdings *keine* Notwendigkeit dafür besteht, eine Fortsetzungsmethode im ursprünglichen Synchronisierungskontext auszuführen, ist der voreingestellte Kontexttransfer *nicht* sinnvoll. Es könnte eine Flaschenhalskonstruktion provoziert und speziell ein UI-Thread behindert werden. Soll

z.B. ein per Befehlsschalter initiierter HTTP-Transfer letztlich zu einer Dateiausgabe (statt zu einer GUI-Veränderung) führen, ist es sinnvoll, durch einen **ConfigureWait()** - Aufruf mit dem Aktualparameter **false** an das per **GetStreamAsync()** angeforderte **Task<Steam>** - Objekt die Verbindung mit dem UI-Thread zu kappen, z.B.:

```
System.IO.Stream str = await client.GetStreamAsync(url).ConfigureAwait(false);
```

Im Ergebnis wird die Fortsetzungsmethode in einem Pool-Thread ausgeführt.

Bis auf spezielle Ausnahmen (meist im Zusammenhang mit der GUI-Aktualisierung) sollte per **ConfigureWait(false)** die Wiederaufnahme im selben Synchronisierungskontext verhindert werden.

15.5.3 Unbehandelte Ausnahmen in asynchronen Methoden

Das bei der C# - Spracherweiterung um die Schlüsselwörter **async** und **await** angestrebte Ziel, asynchronen Code fast genauso bequem erstellen zu können wie synchronen Code, wird auch bei der Behandlung von Ausnahmen realisiert. Die vom Compiler zu lösende Herausforderung besteht darin, dass beim Auftreten einer Ausnahme die Methode, welche die Auftragsbearbeitung angefordert und sich sofort oder später per **await** zurückgezogen hat, vom Stack verschwunden ist. Genau in dieser Methode ist aber die Ausnahmebehandlung kodiert.

Ist eine abzuwartende Aufgabe beendet, wertet der **await**-Operator den Status aus und unterscheidet:

- Die Aufgabe wurde normal beendet.
- Während der Auftragsbearbeitung ist ein unbehandelter Ausnahmefehler aufgetreten.
  In diesem Fall wird die Ausnahme vom **await**-Operator erneut ausgelöst. Das geschieht *ohne* die lästige Verpackung in eine **AggregateException** (siehe Abschnitt 15.4.4).
- Der Auftrag wurde abgebrochen (siehe Abschnitt 15.4.9).
  In diesem Fall löst der **await**-Operator eine **OperationCanceledException** aus.

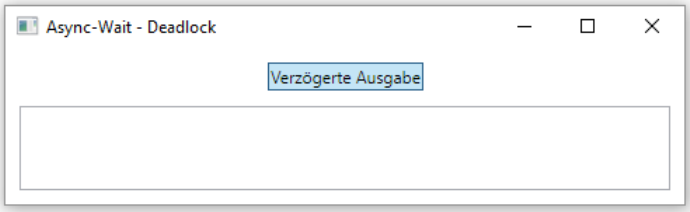
Insgesamt können die in einer asynchronen Methode geworfenen Ausnahmen von ihrem Aufrufer behandelt werden.

15.5.4 Tücken

Wird eine asynchrone Methode genutzt, sollte das zugehörige **Task**- oder **Task<TResult>** - Objekt per **await**-Ausdruck in den Programmablauf integriert werden. Die **Wait()** - Methode oder die **Result**-Eigenschaft einer **async**-Task anzusprechen, kann z.B. im UI-Synchronisierungskontext zum Deadlock führen, wie das folgende Beispiel nach Cleary (2014, S. 6) zeigt:

```
private async Task WaitAsync() {
   await Task.Delay(1000);
}

private void button_Click(object sender, RoutedEventArgs e) {
   Task task = WaitAsync();
   task.Wait();
   text.Text = "Ready";
}
```

Nach einem Klick auf den Befehlsschalter



startet `button_Click()` und ruft `WaitAsync()` auf. Dort wird mit **await** auf das Ablaufen einer durch die statische **Task**-Methode **Delay()** realisierten Verzögerung gewartet. Damit erhält `button_Click()` die Kontrolle zurück und wartet mit **Wait()** auf die Beendigung des **Task**-Objekts zu `WaitAsync()`. Allerdings blockiert `button_Click()` den UI-Thread. Nach Ablauf der **Delay()** - Zeit wartet die vom **await**-Operator erstellte Fortsetzungsmethode auf Ausführung, kommt aber im UI-Thread nicht zum Zug.

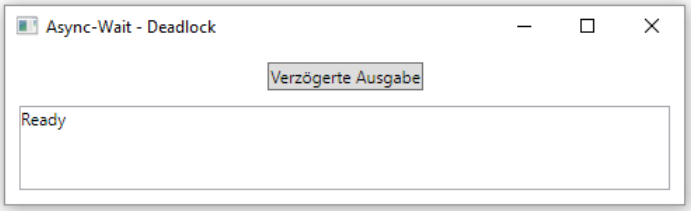
Zur Lösung des Problems bestehen zwei Optionen:

- In `WaitAsync()` wird veranlasst, dass die Fortsetzungsmethode vom Threadpool ausgeführt werden darf (vgl. Abschnitt 15.5.2):
  ```
  await Task.Delay(1000).ConfigureAwait(false);
  ```
- In `button_Click()` wird per **await** - Operator auf das Ende von `WaitAsync()` gewartet:
  ```
  await task;
  ```
  Dann gibt `button_Click()` im Wartezustand den UI-Thread frei, so dass die Fortsetzungsmethode zu `WaitAsync()` und danach die Fortsetzungsmethode zu `button_Click()` ausgeführt werden können.

Mit dieser Lösung

```
private async Task WaitAsync() {
   await Task.Delay(1000);
}

private async void button_Click(object sender, RoutedEventArgs e) {
   Task task = WaitAsync();
   await task;
   text.Text = "Ready";
}
```

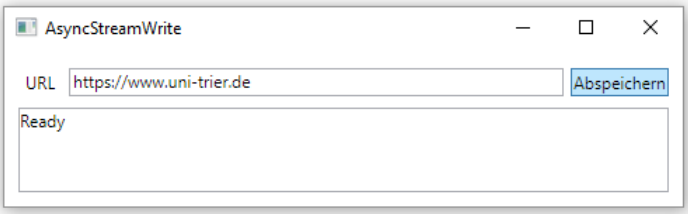
verhält sich das Programm wunschgemäß:



Die seit .NET 4.5 verfügbare statische **Task**-Methode **Delay()** ist übrigens in vielen Situationen eine überlegene Alternative zur **Thread**-Methode **Sleep()**. Als **await**-Operand macht sie es möglich, einen Algorithmus für eine Zeitspanne anzuhalten, ohne den Thread lahmzulegen.

15.5.5 Async-Methoden der Klasse Stream

Seit .NET 4.5 unterstützt die Klasse **Stream** die Task Parallel Library (TPL) durch die Methoden **ReadAsync()** und **WriteAsync()**. Das folgende Programm



befördert auf Knopfdruck den Inhalt einer Webseite in eine lokale Datei.

Die Ereignisbehandlungsmethode zum Befehlsschalter startet eine Aufgabe basierend auf der Methode `GetHtmlCodeAsync()`, führt einige vom Ergebnis des Auftrags unabhängige Arbeiten aus und wartet dann per **await**-Ausdruck auf die Fertigstellung der Aufgabe, ohne den UI-Thread zu blockieren:

```
private async void button_Click(object sender, RoutedEventArgs e) {
   String temp = null;
   try {
      Task ts = GetHtmlCodeAsync(textBox.Text);

      temp = result.Text;
      result.Text = "Bitte warten ....";
      button.IsEnabled = false;

      await ts;
      if (ts.Status == TaskStatus.RanToCompletion)
         result.Text = "Ready";
   } catch (Exception ex) {
      result.Text = temp;
      MessageBox.Show(this, ex.ToString(), ex.Message);
   } finally {
      button.IsEnabled = true;
   }
}
```

In der asynchronen Methode `GetHtmlCodeAsync()` wird zunächst mit der **HttpClient**-Methode **GetStringAsync()** der HTML-Code asynchron und (dank **await**) nicht-blockierend bezogen und in einer **String**-Variablen gespeichert. Anschließend wird aus der Zeichenfolge unter Verwendung der UTF8-Kodierung ein **byte**-Array erstellt, der schließlich mit der **Stream**-Methode **WriteAsync()** in eine Datei geschrieben wird:

```
private async Task GetHtmlCodeAsync(String url) {
   String s = null;
   using (var client = new HttpClient()) {
      s = await client.GetStringAsync(url).ConfigureAwait(false);
   }
   using (var stream = new FileStream("demo.txt", FileMode.Create, FileAccess.Write,
                                      FileShare.None, 4096, useAsync: true)) {
      var utf8enc = new System.Text.UTF8Encoding();
      byte[] sb = utf8enc.GetBytes(s);
      await stream.WriteAsync(sb, 0, sb.Length).ConfigureAwait(false);
   }
}
```

Für die beiden **await**-Operanden in `GetHtmlCodeAsync()` wird per `ConfigureAwait(false)` angeordnet, dass die zugehörige Fortsetzungsmethode *nicht* in dem beim Methodenstart gültigen Synchronisierungskontext (im Beispiel: UI-Thread) ausgeführt werden soll (siehe Abschnitt 15.5.2).

Beim Erstellen eines **Stream**-Objekts ist darauf zu achten, dass die Fähigkeiten des Betriebssystems zur Delegation von Ein-/Ausgabeoperationen an die Gerätetreiber genutzt werden. Dann wird *kein* Poolthread benötigt und eine erhebliche Beschleunigung (bis zum Faktor 10) erzielt, wenn ausschließlich eine asynchrone Nutzung stattfindet. Allerdings sollten in diesem Modus keine synchronen Zugriffe stattfinden, weil dann statt einer Beschleunigung eine Verzögerung in derselben Größenordnung zu erwarten ist.[1] Bei der Klasse **FileStream** wird die asynchrone Ein-/Ausgabe auf Betriebssystemebene im Konstruktor (je nach Überladung) durch die Parameter **useAsync** oder **options** aktiviert, z.B.:

```
FileStream(name, FileMode.Create, FileAccess.Write, FileShare.None, 4096, useAsync: true)
```

Ist diese Option nicht aktiviert, werden die TPL-Techniken (weniger performant) durch einen Pool-Thread realisiert.

15.6 Weitere Multithreading-Techniken

In diesem Abschnitt haben sich Techniken versammelt, ...

- die als veraltet gelten, aber wegen ihrer großen Tradition und Verbreitung erwähnt werden müssen (APM, EAP),
- die für neue Projekte relevant sind, aber aus Zeitgründen nur erwähnt werden können (PLINQ, TPL-Datenflussbibliothek).

15.6.1 Asynchronous Programming Model (APM)

Das APM-Entwurfsmuster (*Asynchronous Programming Model*) wurde schon in .NET 1.0 eingeführt und war lange die empfohlene Technik zur Verwendung von Pool-Threads. Im Vergleich zur direkten Verwendung der **Threadpool**-Methode **QueueUserWorkItem()** (siehe Abschnitt 15.2.1) bot das APM u.a. die folgenden Vorteile:

- Statt der Einschränkung auf den Delegatentyp **WaitCallback** können Methoden mit Rückgabe und Parametern per Pool-Thread ausgeführt werden.
- Der initiierende Thread kann sich über den Abschluss der Hintergrundtätigkeit informieren.

Mittlerweile erlaubt jedoch die in Abschnitt 15.4 vorgestellte Task Parallel Library (TPL) eine weitaus flexiblere Nutzung des Threadpools, und das APM sollte für neue Projekte *nicht mehr* verwendet werden (Albahari & Albahari 2015, S. 618). Weil in zu pflegenden Bestandsprojekten APM-Lösungen für asynchrone Operationen noch zahlreich anzutreffen sind, folgt eine kurze Beschreibung.

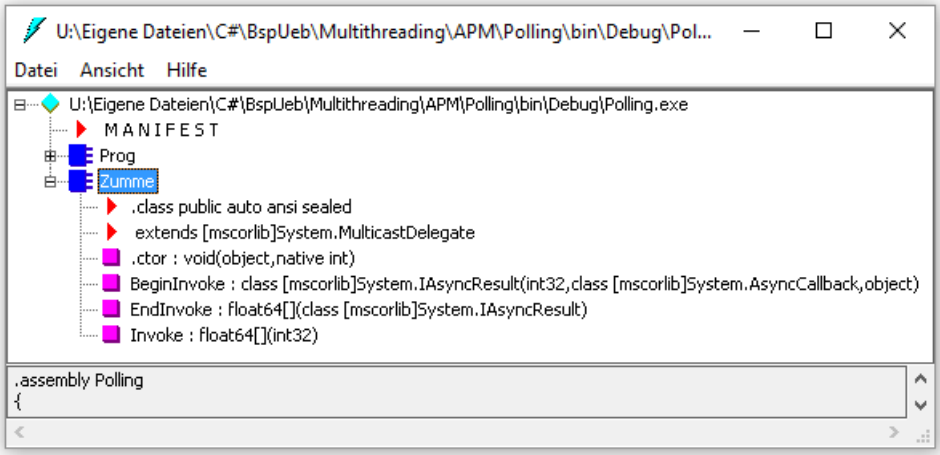
15.6.1.1 Asynchrone CPU-Nutzung

Zu *jeder* Delegatendefinition, z.B.:

```
public delegate double[] Zumme(int anz);
```

erstellt der Compiler eine Klasse mit den Methoden **BeginInvoke()** und **EndInvoke()**, wie die folgende **IlDasm**-Ausgabe für den Typ `Zumme` zeigt:

[1] https://msdn.microsoft.com/de-de/library/7db28s3c(v=vs.110).aspx



Um das recht komplexe Zusammenspiel verschiedener Methoden im APM anschaulich erläutern zu können, betrachten wir eine konkrete (nicht sonderlich sinnvolle) Methode, welche den Delegatentyp `Zumme` erfüllt:

```
static double[] RandomSums(int anz) {
   double[] erg = new double[anz];
   for (int i = 0; i < anz; i++) {
      for (int j = 0; j < 10000000; j++)
         erg[i] += zzg.Next(100);
   }
   return erg;
}
```

Sie liefert einen Array mit **double**-Elementen, die jeweils eine Summe von 10.000.000 Zufallszahlen enthalten. Über den Parameter wählt man die Anzahl der Array-Elemente. Wir bezeichnen `RandomSums()` später als *Arbeitsmethode*.

Wir erstellen aus `RandomSums()` ein Delegatenobjekt vom Typ `Zumme` (gleich als *Arbeitsdelegat* bezeichnet) und beauftragen dieses Objekt, seine Methode **BeginInvoke()** auszuführen:

```
Zumme rs = new Zumme(RandomSums);
IAsyncResult ar = rs.BeginInvoke(3, null, null);
```

Als Rückgabewert liefert **BeginInvoke()** ein Objekt vom Typ **IAsyncResult**, das die asynchron ausgeführte Operation identifiziert. Wir bezeichnen es gleich als *Operationsobjekt*. Es …

- muss der Arbeitsdelegatenmethode **EndInvoke()** (siehe unten) als Parameter übergeben werden, um das Ergebnis der abgeschlossenen Operation zu erhalten, z.B. die **double[]**-Rückgabe der Arbeitsmethode):
  ```
  double[] da = rs.EndInvoke(ar);
  ```
- kann mit seiner booleschen Eigenschaft **IsCompleted** über den Abschluss der Arbeiten informieren.
- enthält ein Signalisierungsobjekt vom Typ **WaitHandle** (vgl. Abschnitt 15.1.4.3), das über die Eigenschaft **AsyncWaitHandle** angesprochen werden kann.

BeginInvoke() besitzt alle Parameter der Delegatendefinition (im Beispiel: einen Parameter vom Typ **int**) und außerdem am Ende seiner Parameterliste:

- **AsyncCallback**
  Dieser Delegatentyp hat folgende Signatur:

  public delegate void AsyncCallback(IAsyncResult *ar***)**

  Beim **BeginInvoke()** - Aufruf kann eine Rückrufmethode angegeben werden, die bei Beendigung der Arbeitsmethode aufgerufen wird und das Operationsobjekt als Parameter erhält. In der ersten Variante des Beispielprogramms wird noch auf eine Rückrufmethode verzichtet und dem **BeginInvoke()** - Aufruf **null** an Stelle eines Rückruf-Delegatenobjekts übergeben.
- **Object**
  Dieses Objekt ist in der **IAsyncResult**-Rückgabe über die Eigenschaft **AsyncState** ansprechbar. Wir benutzen diesen Parameter später, um der Rückrufmethode den Arbeitsdelegaten bekannt zu machen. In der ersten Variante des Beispielprogramms wird auf diese Option verzichtet und dem **BeginInvoke()** - Aufruf **null** als dritter Parameter übergeben.

Durch den **BeginInvoke()** - Aufruf an den Arbeitsdelegaten gelangt über die **ThreadPool**-Methode **QueueUserWorkItem()** ein Arbeitsauftrag in die Warteschlange des Threadpools.

In der ersten Variante des Beispielprogramms stellen wir über die Eigenschaft **IsCompleted** des Operationsobjekts fest, ob der bearbeitende Hintergrund-Thread fertig ist:

```
int sek = 0;
while (!ar.IsCompleted) {
 Console.WriteLine("Warte seit {0} Sekunden auf den Hintergrund-Thread",sek++);
 Thread.Sleep(1000);
}
```

Alternativ lässt sich derselbe Zweck auch über einen **WaitOne()** - Aufruf an das im Operationsobjekt enthaltenen **WaitHandle** erreichen:

```
int sek = 0;
while (!ar.AsyncWaitHandle.WaitOne(1000))
   Console.WriteLine("Warte seit {0} Sekunden auf den Hintergrund-Thread", ++sek);
```

Hat der Pool-Thread seine Arbeit abgeschlossen, kann man beim Arbeitsdelegaten per **EndInvoke()** - Aufruf die Ergebnisse anfordern, wobei das Operationsobjekt als Parameter zu übergeben ist:

```
double[] da = rs.EndInvoke(ar);
```

Die **EndInvoke()** - Methode hat denselben Rückgabewert wie die die Arbeitsmethode (im Beispiel: die **double[]**).

Neben der Übergabe der Arbeitsergebnisse des Pool-Threads hat die **EndInvoke()** - Methode noch weitere Aufgaben und sollte daher auf jeden Fall aufgerufen werden:

- Ist bei der Hintergrundaktivität eine Ausnahme aufgetreten, wird diese von **EndInvoke()** erneut geworfen, so dass sie vom aufrufenden Thread zu Kenntnis genommen und bearbeitet werden kann.
- Die von der CLR zur Verwaltung der asynchronen Operation benutzten Ressourcen werden freigegeben. Ohne Aufruf von **EndInvoke()** entsteht ein Ressourcenleck (Richter 2006, S. 621).

Statt in einer **while**-Schleife zwischen zwei **IsCompleted**-Abfragen den primären Thread schlafen zu legen (siehe oben), könnten wir die Wartezeit für nützliche Arbeiten verwenden.

Eine wenig empfehlenswerte Alternative besteht darin, ohne Prüfung des Bearbeitungszustands die **EndInvoke()** - Methode des Arbeitsdelegaten aufzurufen, weil diese Methode auf das Auftragsende wartet und damit den aktuellen Thread blockiert.

Mit der **Rückruftechnik** kommen wir ohne wiederholtes Nachfragen (*Polling*) und ohne Warten an die Ergebnisse einer asynchronen Auftragsabwicklung heran. Dazu definieren wir eine Rückrufmethode, die den Delegatentyp **AsyncCallBack** (siehe oben) erfüllt, z.B.:

```
static void Report(IAsyncResult ar) {
   Zumme z = (Zumme)ar.AsyncState;
   Console.WriteLine("Report der ermittelten Summen:");
   double[] da = z.EndInvoke(ar);
   foreach (double zs in da)
      Console.WriteLine(" "+zs);
}
```

Sobald die Arbeitsmethode beendet ist, wird (immer noch im Pool-Thread) die Rückrufmethode aufgerufen und dabei per Aktualparameter mit einer Referenz auf das Operationsobjekt versorgt. Im Beispiel soll die Rückrufmethode `Report()` das Operationsobjekt in einem **EndInvoke()** - Aufruf an den Arbeitsdelegaten verwenden, um die Ergebnisse anzufordern. Die nötige Referenz auf den Arbeitsdelegaten wird folgendermaßen in die Rückrufmethode transportiert:

- Im **BeginInvoke()** - Aufruf an den Arbeitsdelegaten wird seine eigene Adresse als dritter Parameter übergeben:
  ```
  rs.BeginInvoke(3, new AsyncCallback(Report), rs);
  ```
- Wie oben erläutert, ist dieser Parameter im Operationsobjekt enthalten und über die Eigenschaft **AsyncState** ansprechbar.

Das folgende Programm

```
using System;
using System.Threading;

public delegate double[] Zumme(int anz);

class Prog {
   static Random zzg = new Random();

   static double[] RandomSums(int anz) {
      . . .
   }
   static void Report(IAsyncResult ar) {
      . . .
   }

   static void Main() {
      Zumme rs = new Zumme(RandomSums);
      rs.BeginInvoke(3, new AsyncCallback(Report), rs);
      Console.WriteLine("Der Arbeitsdelegat soll per Pool-Thread 3 " +
         "Summen von Zufallszahlen ermitteln.\n");
      Console.WriteLine("Der primäre Thread schläft 5 Sekunden.\n");
      Thread.Sleep(5000);
      Console.WriteLine("\nDer primäre Thread ist aufgewacht.\n");
   }
}
```

demonstriert die APM-Rückruftechnik, verzichtet aber der Übersichtlichkeit halber auf eine sinnvolle Aktivität im Vordergrund-Thread.

```
Der Arbeitsdelegat soll per Pool-Thread 3 Summen von Zufallszahlen ermitteln.

Der primäre Thread schläft 5 Sekunden.

Report der ermittelten Summen:
 495128107
 495006556
 495034464

Der primäre Thread ist aufgewacht.
```

Über die Rückruftechnik gewinnen wir an Flexibilität bei der Verwertung von Ergebnissen einer asynchronen Auftragsabwicklung, weil nach Beendigung der Arbeitsmethode automatisch die Rückrufmethode gestartet wird. Doch ist zu betonen, dass die Rückrufmethode im Hintergrund-Thread abläuft. Wenn der initiierende Thread auf den erfolgreichen Abschluss eines Hintergrund-Threads angewiesen ist, muss irgendeine Synchronisation stattfinden.

Bei Richter (2006, S. 621) finden sich wichtige Empfehlungen zum APM-Einsatz:

- Auf einen **BeginInvoke()** - Aufruf sollte unbedingt ein **EndInvoke()** - Aufruf mit dem zugehörigen Operationsobjekt als Parameter folgen, weil ansonsten von der asynchronen Operation belegte CLR-Ressourcen nicht mehr frei gegeben werden. Wie das obige Beispiel zeigt, kann der **EndInvoke()** - Aufruf in der Rückrufmethode und somit im Hintergrund-Thread erfolgen, so dass der aufrufende Thread auf keinen Fall warten muss.[1]
- Für jedes Operationsobjekt sollte **EndInvoke()** nur *einmal* aufgerufen werden, weil diese Methode eventuell Aufräumungsarbeiten ausführt, so dass ein erneuter Aufruf scheitern könnte.

15.6.1.2 Asynchrone Schreib- und Leseoperationen

Als asynchrone Alternative zu den in Abschnitt 14.1.3 vorgestellten synchronen Methoden **Read()** und **Write()** unterstützt die Klasse **Stream** das APM-Muster durch die Methodenpaare

- **BeginRead()** und **EndRead()**
- **BeginWrite()** und **EndWrite()**

Allerdings ist mir auf diesem Weg *keine* asynchrone Dateiausgabe mit Hilfe der Klasse **FileStream** gelungen. Unter Windows 10 (64 Bit) mit .NET 4.6 zeigte **BeginWrite()** ein synchrones (blockierendes) Verhalten.[2]

Seit .NET 4.5 unterstützt **Stream** auch die modernere Task Parallel Library (TPL) durch die Methoden **ReadAsync()** und **WriteAsync()**, denen bei neuen Projekten der Vorzug gegeben werden sollte (siehe Beispiel in Abschnitt 15.5.5).

15.6.2 Event-based asynchronous Pattern (EAP)

Klassen, die das in .NET 2.0 eingeführte *Event-based asynchronous Pattern* (EAP) implementieren, bieten Methoden mit dem Namensausklang **Async** an, welche die asynchrone Ausführung von zeitaufwändigen Operationen erlauben. Ein Beispiel ist die Klasse **SmtpClient** mit der Methode **SendAsync()** zum Versenden einer Mail. Die Beendigung einer **Async**-Methode meldet ein Ereignis mit dem Namensausklang **Completed**, das denselben Namensanfang besitzt wie die Methode.

---

[1] Eine Ausnahme von der Pflicht zum **EndInvoke()** - Aufruf besteht beim **BeginInvoke()** - Aufruf an das **Dispatcher**-Objekt des UI-Threads (siehe Abschnitt 15.3.1).

[2] Hier findet sich ein Analyse des Problems:
https://www.codeproject.com/tips/575618/avoiding-deadlocks-with-system-io-stream-beginread

Das zu **SendAsync()** gehörige Ereignis heißt also **SendCompleted**. Mittlerweile wird das EAP als veraltete und überflüssige Technik angesehen (siehe z.B. Albahari & Albahari 2015, S. 619).

15.6.3 PLINQ und TPL-Datenflussbibliothek

Im Unterschied zu den bisher im Abschnitt 15.6 erwähnten Techniken sind die folgenden Verfahren aktuell, können aber leider aus Zeitgründen nicht behandelt werden:

- PLINQ
  Das mit .NET 4.0 eingeführte PLINQ (paralleles LINQ) kann die LINQ-Standardabfrageoperationen für aufzählbare Kollektionen (LINQ to Objects) durch Parallelität beschleunigen.[1]
- TPL-Datenflussbibliothek
  Die komplexe, mit .NET 4.5 eingeführte TPL-Datenflussbibliothek erweitert die TPL um die Möglichkeit zur Modellierung von Arbeitsabläufen bei einem möglichst hohen Grad an Parallelität.[2]

15.7 Übungsaufgaben zu Kapitel 15

1) Welche von den folgenden Aussagen sind richtig?

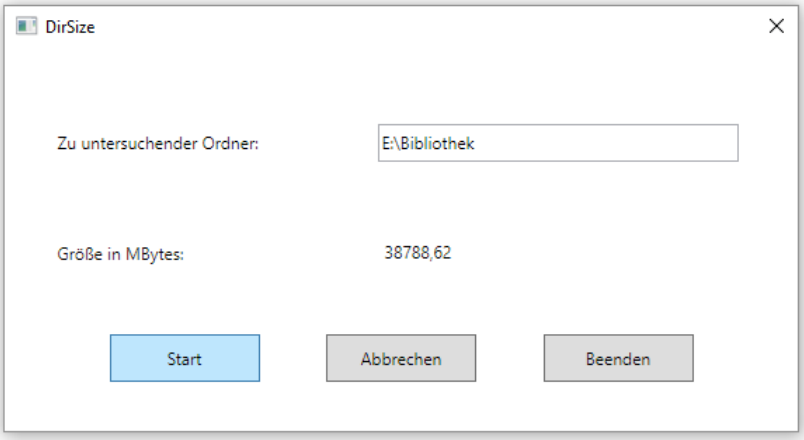
1. Ein Thread im Zustand **Stopped** lässt sich mit der Methode **Start()** reaktivieren
2. WPF-Steuerelemente können nur in dem Thread angesprochen werden, der sie erstellt hat.
3. Bei der **lock**-Anweisung
   ```
   lock(sperre) {
   ...
   }
   ```
   ist sichergestellt, dass der kritische Block auch beim Auftreten einer Ausnahme verlassen wird.
4. Ist für einen Thread die **Abort()** - Methode aufgerufen worden, ist sein Ende nicht mehr zu verhindern.

2) Objekte der Signalisierungsklasse **CountdownSignal** enthalten einen Zähler und starten mit einem positiven Wert, der sich bei jedem Aufruf der Instanzmethode **Signal()** um Eins verringert. Hat sich ein Thread per **Wait()** - Aufruf an das Signalisierungsobjekt in Wartestellung begeben, wird er beim Zählerstand Null reaktiviert. Verwenden Sie die Klasse im Rahmen eines Konsolenprogramms für einen simulierten Raketenstart, der in einem eigenen Thread abläuft, sobald ein **CountdownSignal** 10mal gesetzt worden ist. Die Kontrollausgaben des Programms könnten ungefähr so aussehen:

```
Thread-Rakete bereit, wartet auf CountdownEvent.

Countdown läuft:
10 9 8 7 6 5 4 3 2 1

Thread-Rakete startet:
.. . .  .  .   .   .    .    .     .     .      .      .      .
```

[1] https://msdn.microsoft.com/de-de/library/dd460688(v=vs.110).aspx
[2] https://msdn.microsoft.com/de-de/library/hh228603(v=vs.110).aspx

3) Erstellen Sie eine WPF-Anwendung, welche die Größe eines Ordners (inklusive aller Unterordner) in einer unterbrechbaren Aufgabe berechnet, so dass die Bedienelemente des Formulars stets verzögerungsfrei reagieren, z.B.:



Anhang

A. Operatorentabelle

In der folgenden Tabelle sind alle im Kurs behandelten Operatoren in absteigender Bindungskraft (von oben nach unten) aufgelistet. Gruppen von Operatoren mit gleicher Bindungskraft (Priorität) sind durch horizontale Linien abgegrenzt.

Operator	**Bedeutung**
x.*y*	Member-Zugriff
Methode(*Parameter*)	Methoden- oder Delegatenaufruf
[]	Index-Zugriff
x++, *x* --	Postinkrement bzw. -dekrement
new	Objekterzeugung
typeof(*Type*)	Typ eines Bezeichners ermitteln
nameof(*Var*)	Name eines Bezeichners ermitteln
checked, unchecked	Ganzzahl-Überlaufdiagnose
default(*Type*)	Standardwert eines Typs Wert ermitteln
?., ?[	Null-bedingter Operator
-*x*	Vorzeichenumkehr
!	Negation
~	Bitweise Negation
++*x*, --*x*	Präinkrement bzw. -dekrement
(*Type*)*x*	Typumwandlung
await	Auf die Beendigung einer Task warten
*, /	Multiplikation, Division
%	Modulo (Divisionsrest)
+, -	Addition, Subtraktion
+	String-Verkettung

Operator	Bedeutung
<<, >>	Links- bzw. Rechts-Shift
>, <, >=, <=, is, as	Vergleichsoperatoren
==, !=	Gleichheit, Ungleichheit
&	Bitweises UND
&	Logisches UND (mit unbedingter Auswertung)
^	Exklusives bitweises ODER
^	Exklusives logisches ODER
|	Bitweises ODER
|	Logisches ODER (mit unbedingter Auswertung)
&&	Logisches UND (mit bedingter Auswertung)
||	Logisches ODER (mit bedingter Auswertung)
??	Null-Sammeloperator (Null-Koaleszenz)
? :	Konditionaloperator
=	Wertzuweisung
+=, -=, *=, /=, %=	Wertzuweisung mit Aktualisierung
=>	Lambda-Operator

Mit Ausnahme der Zuweisungsoperatoren sind alle binären Operatoren *links*-assoziativ. Die Zuweisungsoperatoren und der Konditionaloperator sind *rechts*-assoziativ.

Eine wertvolle Referenz zur Klärung von Zweifelsfällen bzgl. der Bindungskraft (Priorität) von Operatoren ist das Buch von Albahari & Albahari (2015, S. 53ff).

B. Lösungsvorschläge zu den Übungsaufgaben

Kapitel 1 (Einleitung)

Aufgabe 1

Das Prinzip der Datenkapselung reduziert die Fehlerquote und damit den Aufwand zur Fehlersuche und -bereinigung. Die perfektionierte Modularisierung durch die Koppelung von Merkmalen und zugehörigen Handlungskompetenzen in einer Klassendefinition erleichtert die …

- Kooperation von mehreren Programmierern bei großen Projekten,
- die Wiederverwendung von Software.

Aufgabe 2

1. **Falsch**
   In C# werden Programme für die .NET - Plattform entwickelt, die grundsätzlich auf jedem Betriebssystem aufsetzen kann. Dank der Bemühungen des Mono-Projekt und der Firma Xamarin (mittlerweile von Microsoft übernommen) lässt sich mit C# auch Software für Linux, MacOS-X, Android und iOS entwickeln.
2. **Richtig**
3. **Falsch**
   Die Standardbibliothek (FCL) ist für alle .NET - Sprachen identisch.
4. **Richtig**
   Allerdings müssen die Regeln der CLS (Common Language Specification) eingehalten werden, damit die Interoperabilität garantiert ist.

Aufgabe 3

Bisher wurden als Aufgabe der CLR erwähnt:

- Verifikation des CIL-Codes beim Laden
  So werden technische Defekte abgefangen.
- Der JIT-Compiler in der CLR übersetzt den CIL-Code der Assemblies in Maschinencode.
- Speicherverwaltung (GC, Garbage Collection)
- Überwachung von Code mit beschränkten Rechten (via Internet bezogen)

Aufgabe 4

Namensräume und Assemblies sind zwei voneinander *unabhängige* Organisationsstrukturen:

- Klassen, die zum selben Namensraum gehören, können in *verschiedenen* Assemblies implementiert sein.
- In einem Assembly können Klassen aus verschiedenen Namensräumen implementiert werden, was aber üblicherweise nicht geschieht.

Z.B. befindet sich die Klasse **Uri** aus dem Namensraum **System**, die zur Modellierung von Netzwerk-Ressourcen dient, im DLL-Assembly **System.dll**. Die zum selben Namensraum gehörende Klasse **Console**, die in unseren Übungsprogrammen häufig zur Ein-/Ausgabe per Konsolen verwendet wird, steckt jedoch im DLL-Assembly **mscorlib.dll**, das auch ohne Referenz vom Compiler stets durchsucht wird.

Aufgabe 5

CIL	Ein .NET - Compiler übersetzt den Quellcode nicht in Maschinensprache, sondern in die *Common Intermediate Language* (kurz: CIL). Diesen Zwischencode übersetzt der JIT-Compiler in der CLR in Maschinencode.
FCL	Die Standardklassenbibliothek der .NET - Plattform wird als *Framework Class Library* bezeichnet. Sie enthält ausgereifte Lösungen (Klassen) für praktisch alle Routineaufgaben der Programmierung (z.B. grafische Bedienoberflächen, Dateiverarbeitung, Netzwerkprogrammierung).
CLS	Microsoft hat unter dem Namen *Common Language Specification* einen Sprachumfang definiert, den *jede* .NET - Programmiersprache erfüllen muss. Beschränkt man sich bei der Klassendefinition auf diesen kleinsten gemeinsamen Nenner, ist die Interoperabilität mit anderen CLS-kompatiblen Klassen sichergestellt.
COM	Diese traditionelle Windows-Komponententechnologie (*Component Object Model*) soll vom .NET - Framework abgelöst werden. Aktuell (2016) spielt COM-Software in der Windows-Welt aber noch eine große Rolle.

Kapitel 2 (Werkzeuge zum Entwickeln von C# - Programmen)

Aufgabe 2

Beim Hallo-Programm lohnt sich die **using**-Direktive für den Namensraum **System** ausnahmsweise nicht, weil das Namensraumpräfix im Quellcode nur einmal auftritt:

```
class Hallo {
   static void Main() {
      System.Console.WriteLine("Hallo Allerseits!");
   }
}
```

Aufgabe 3

- Das Schlüsselwort **using** wird klein geschrieben.
- Der Methodenname **WriteLine()** ist falsch geschrieben.
- Die Zeichenfolge im Parameter des **WriteLine()** - Aufrufs muss mit dem "-Zeichen abgeschlossen werden.
- Die schließende Klammer zum Rumpf der Klassendefinition fehlt.

Kapitel 3 (Elementare Sprachelemente)

Abschnitt 3.1 (Einstieg)

Aufgabe 1

Der 1. Aufruf scheitert:	Der Methodenname **Main** muss groß geschrieben werden.
Der 2. Aufruf klappt:	Der Modifikator **public** ist allerdings nicht erforderlich.
Der 3. Aufruf klappt:	Statt **void** ist auch der Rückgabetyp **int** erlaubt. Dann muss **Main()** aber einen **int**-Wert an die CLR zurückliefern. Wie das per **return**-Anweisung bewerkstelligt wird, erfahren Sie später.
Der 4. Aufruf scheitert:	Der Rückgabetyp **double** ist verboten.
Der 5. Aufruf klappt:	Diese Variante haben wir bisher meistens benutzt.

Aufgabe 2

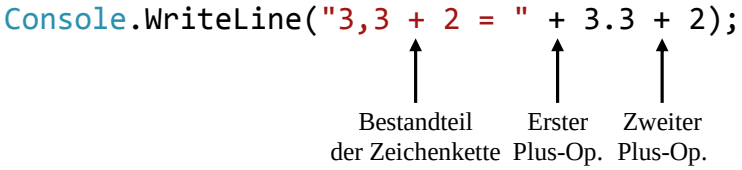
Unzulässig sind:

- `4you`
  Namen müssen mit einem Buchstaben beginnen.
- `else`
  Schlüsselwörter wie **else** sind als Namen verboten.

Abschnitt 3.2 (Ausgabe bei Konsolenanwendungen)

Aufgabe 1

Von den beiden im **WriteLine()** - Parameter auftretenden Plus-Operatoren



wird der linke (unmittelbar auf die Zeichenkette folgende) zuerst ausgeführt und bewirkt eine Verkettung von Zeichenfolgen, wobei die Zahl 3.3 automatisch in eine Zeichenfolge konvertiert wird. Anschließend wirkt der zweite Plus-Operator analog und erweitert die Zeichenfolge um die Ziffer „2“.

Mit runden Klammern kann man dafür sorgen, dass der *zweite* Plus-Operator zuerst ausgeführt wird. Er steht zwischen zwei numerischen Argumenten und addiert diese. Das Ergebnis wird dann vom ersten Plus-Operator in eine Zeichenfolgenverkettung einbezogen:

Quellcode	Ausgabe
`using System;` `class Prog {` `  static void Main() {` `  Console.WriteLine("3,3 + 2 = " + (3.3 + 2));` `  }` `}`	`3,3 + 2 = 5,3`

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FormAus

Abschnitt 3.3 (Variablen und Datentypen)

Aufgabe 1

Die Variable `i` ist nur im innersten Block gültig.

Aufgabe 2

So lässt sich das Programm übersetzen:

```
class Prog {
  static void Main() {
    float pi = 3.141593f;
    double radius = 2.0;
    System.Console.WriteLine("Der Flächeninhalt beträgt: {0:f3}",
        pi * radius * radius);
  }
}
```

Aufgabe 3

Lösungsvorschlag:

```
using System;
class Prog {
  static void Main() {
    Console.WriteLine("Dies ist ein Zeichenketten-Literal:\n\t\"Hallo\"");
  }
}
```

Aufgabe 4

char gehört zu den integralen (ganzzahligen) Datentypen. Jedes Zeichen wird über seine Nummer im Unicode-Zeichensatz gespeichert, das Zeichen 'c' offenbar durch die Zahl 99 (im Dezimalsystem). Der dezimalen Zahl 99 entspricht die hexadezimale Zahl 0x63 (= 6 · 16 + 3). In der folgenden Anweisung wird der **char**-Variablen `zeichen` die Unicode-Escape-Sequenz für das Zeichen 'c' zugewiesen:

```
char zeichen = '\u0063';
```

Abschnitt 3.5 (Operatoren und Ausdrücke)

Aufgabe 1

Die Ausdrücke haben folgende Typen und Werte:

Ausdruck	Typ	Wert	Anmerkungen
`6/4*2.0`	**double**	`2.0`	Abarbeitung mit Zwischenergebnissen: **`6/4`**`*2.0` **`1*2.0`**
`(int)6/4.0*3`	**double**	`4.5`	Der Typumwandlungsoperator hat die höchste Priorität und bezieht sich daher (ohne Wirkung) auf die 6. Abarbeitung mit Zwischenergebnissen: **`(int)6`**`/4.0*3` **`6/4.0`**`*3` **`1.5*3`**
`(int)(6/4.0*3)`	**int**	`4`	Abarbeitung mit Zwischenergebnissen: `(int)(`**`6/4.0`**`*3)` `(int)(`**`1.5*3`**`)` **`(int)4.5`**
`3*5+8/3%4*5`	**int**	`25`	Abarbeitung mit Zwischenergebnissen: **`3*5`**`+8/3%4*5` `15 + `**`8/3%4`**`*5` `15 + `**`2%4`**`*5` `15 + `**`2*5`** **`15 + 10`**

Aufgabe 2

`erg1` erhält den Wert 2, denn:

- `(i++ == j ? 7 : 8)` hat den Wert 8, weil 2 ≠ 3 ist.
- 8 `%` 3 ergibt 2.

`erg2` erhält den Wert 0, denn:

- Der Präinkrementoperator trifft auf die bereits vom Postinkrementoperator in der vorangehenden Zeile auf den Wert 3 erhöhte Variable `i` und setzt sie auf den Wert 4.
- Dies ist auch der Wert des Ausdrucks `++i`, so dass die Bedingung im Konditionaloperator erneut den Wert **false** hat.
- `(++i == j ? 7 : 8)` hat also den Wert 8, und 8 `%` 2 ergibt 0.

Aufgabe 3

Die Vergleichsoperatoren (>, ==) haben eine höhere Priorität als die logischen Operatoren und der Zuweisungsoperator, so dass z.B. in der folgenden Anweisung

```
la1 = 2 > 3 && 2 == 2 ^ 1 == 1;
```

auf runde Klammern verzichtet werden konnte. Besser lesbar ist aber wohl die äquivalente Variante:

```
la1 = (2 > 3) && (2 == 2) ^ (1 == 1);
```

`la1` erhält den Wert **false**, denn der Operator ^ wird aufgrund seiner höheren Bindungskraft vor dem Operator **&&** ausgewertet.

`la2` erhält den Wert **true**, weil die runden Klammern dafür sorgen, dass der Operator ^ zuletzt ausgewertet wird.

`la3` erhält den Wert **false**.

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\Exp

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2Euro

Abschnitt 3.7 (*Anweisungen****)***

Aufgabe 1

Weil die **else**-Klausel der *zweiten* (nach oben nächstgelegenen) **if**-Anweisung zugeordnet wird, ergibt sich folgender „Gewinnplan“:

`losNr`	Gewinn
durch 13 teilbar	0 €
nicht durch 13, aber durch 7 teilbar	1 €
weder durch 13, noch durch 7 teilbar	100 €

Aufgabe 2

Im logischen Ausdruck der **if**-Anweisung findet an Stelle eines Vergleichs eine *Zuweisung* statt. Der Zuweisungsausdruck `(b = false)` besitzt den bei einer **if**-Anweisung erforderlichen Typ **bool**, weil die Variable `b` und der zugewiesene Ausdruck (Literal **false**) diesen Typ besitzen.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\PrimitivOBC

Die Lösung ohne **break** und **continue** ist komplizierter, also in der Regel nicht sinnvoll.

Aufgabe 4

Das Semikolon am Ende der Zeile

```
while (i < 100) ;
```

wird vom Compiler als die zur **while**-Schleife gehörige (leere) Anweisung interpretiert, so dass mangels `i`-Inkrementierung eine *Endlosschleife* vorliegt. Die restlichen Zeilen werden als selbständiger Anweisungsblock gedeutet, aber nie ausgeführt.

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\DM2EuroS

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Elementare Sprachelemente\FloP

Bei $i = 52$ erhält man letztmals das mathematisch korrekte Vergleichsergebnis. Dies ist gerade die Anzahl der Bits in der **double**-Mantisse.

Die Zahl 1,0 hat nach der Norm IEE 754 die folgende normalisierte **double**-Darstellung (vgl. Abschnitt 3.3.5.1)

$$(-1)^0 \cdot 2^{(1023-1023)} \cdot (1+0{,}0)$$

mit den Bits:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000000

Das Bitmuster zum Vergleichswert:

$$1{,}0 + 2^{-i} = (-1)^0 \cdot 2^{(1023-1023)} \cdot (1 + 2^{-i}),\ i = 1, \ldots, 52$$

hat eine zusätzliche Eins beim i-ten Mantissen-Bit. Bei $i = 52$ ist diese Eins am rechten Rand angekommen und letztmals vorhanden:

Vorz.	Exponent	Mantisse
0	01111111111	0000000000000000000000000000000000000000000000000001

Während bei binärer Gleitkommatechnik mit **double**-Präzision die Zahl

$$1{,}0 + 2^{-53} \approx 1{,}0 + 1{,}1102230246251565 \cdot 10^{-16} \approx 1{,}0000000000000001110223024625157$$

nicht mehr von der Zahl 1,0 zu unterscheiden ist, kann die Zahl

$$2^{-53} \approx 1{,}1102230246251565 \cdot 10^{-16}$$

mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	01111001010	0000000000000000000000000000000000000000000000000000

„mühelos“ von der 0,0 mit dem Bitmuster

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000000

unterschieden werden.

Erst ab 2^{-1023} ($\approx 1{,}1125369292536007 \cdot 10^{-308}$) wird erstmals die denormalisierte Darstellung benötigt

Vorz.	Exponent	Mantisse
0	00000000000	1000000000000000000000000000000000000000000000000000

und bei 2^{-1074} ($\approx 4{,}9406564584124654 \cdot 10^{-324}$)

Vorz.	Exponent	Mantisse
0	00000000000	0000000000000000000000000000000000000000000000000001

ist letztmals eine Unterscheidung von der Zahl 0,0 möglich.

Aufgabe 7

Lösungsvorschläge mit den beiden Algorithmus-Varianten befinden sich in den Ordnern:

...\BspUeb\Elementare Sprachelemente\GGT.Diff
...\BspUeb\Elementare Sprachelemente\GGT.Mod

Kapitel 4 (Klassen und Objekte)

Aufgabe 1

1. **Falsch**
2. **Stimmt**
   Die Methodenüberladung ist erlaubt und nützlich.
3. **Falsch**
   Es darf überhaupt kein Rückgabetyp angegeben werden.
4. **Falsch**
   Mit der Datenkapselung wird verhindert, dass die *Methoden fremder Klassen* auf Instanzvariablen zugreifen. Es geht *nicht* darum, Objekte einer Klasse voreinander zu schützen. Die von einem Objekt ausgeführten Methoden haben stets vollen Zugriff auf die Instanzvariablen anderer Objekte derselben Klassen, sofern eine entsprechende Referenz vorhanden ist. Der Klassendesigner ist für das sinnvolle Verhalten der Methoden verantwortlich.
5. **Stimmt**
6. **Falsch**

Aufgabe 2

Ein **readonly** - Feld kann nach der Initialisierung nicht mehr verändert werden, auch nicht von klasseneigenen Methoden. Eine *getonly* - Eigenschaft erlaubt nur den lesenden Zugriff. Ein zugrunde liegendes Feld kann jedoch von klasseneigenen Methoden jederzeit geändert werden. Methoden fremder Klassen sollte in der Regel der direkte Zugriff auf das Feld versperrt sein (Datenkapselung).

Aufgabe 3

In der Anweisung

```
return Anzahl;
```

wurde `Anzahl` versehentlich groß geschrieben, so dass sich die **get**-Methode der Eigenschaft `Anzahl` selbst aufruft. Diese ungeplante Rekursion führt zu einem Stack-Überlauf (vgl. Abschnitt 4.7.1).

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\R2Vek

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Klassen und Objekte\FakulRek

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Einleitung\Bruch\GUI

Aufgabe 7

Die korrekte Lösung:

Begriff	Pos.
Definition einer Instanzmethode mit Referenzrückgabe	7
Deklaration lokale Variable	8
Definition einer Instanzmethode mit Wertparameter vom Typ einer Klasse	5
Deklaration von Instanzvariablen	1
Methodenaufruf	6
Deklaration einer statischen Eigenschaft	12

Begriff	Pos.
Konstruktordefinition	3
Deklaration einer Klassenvariablen	2
Objekterzeugung	11
Definition einer Klassenmethode	10
Definition einer Instanzeigenschaft	4
Operatorüberladung	9

Kapitel 5 (Weitere .NETte Typen)

Aufgabe 1

Dank Autoboxing kann man einer **object**-Variablen auch einen **int**-Wert zuweisen, wobei ein neues Objekt auf dem Heap erstellt wird, das den **int**-Wert als Kopie erhält. Im Ausdruck

```
o1 == o2
```

werden die Inhalte der beiden Referenzvariablen, also die Adressen der beiden referenzierten Objekte, verglichen, die im Beispielprogramm verschieden sind.

Beim Vergleich von zwei Referenzvariablen mit Datentyp **String** orientiert sich der Identitätsoperator allerdings *nicht* an den enthaltenen Adressen, sondern an den Inhalten der referenzierten **String**-Objekte (siehe Abschnitt 5.4.1.2.2).

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Lotto

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\Eratosthenes

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Arrays\FloatMatrix

Aufgabe 5

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\PerZuf

Aufgabe 6

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Weitere .NETte Typen\Zeichenfolgen\StringUtil

Kapitel 6 (Vererbung und Polymorphie)

Aufgabe 1

In der Basisklasse fehlt ein parameterfreier Konstruktor. Weil die abgeleitete Klasse keinen expliziten Konstruktor besitzt, kommt dort der Standardkonstruktor zum Einsatz, der den parameterfreien Konstruktor der Basisklasse aufruft (vgl. Abschnitt 6.3).

Aufgabe 2

In der Klasse `Figur` haben `xpos` und `ypos` den voreingestellten Zugriffsschutz **private**. Damit hat die `Kreis`-Klasse keinen direkten Zugriff. Soll dieser Zugriff möglich sein, müssen `xpos` und `ypos` in der `Figur`-Definition die Schutzstufe **protected** (oder **public**) erhalten.

Aufgabe 3

Beim *Überladen* existieren in einer Klasse mehrere Methoden mit demselben Namen, aber verschiedenen Parameterlisten. Eventuell sind einige von den überladenen Methoden in der Klasse selbst definiert und andere geerbt.

Beim *Verdecken* und beim *Überschreiben* findet eine Ersetzung der Basisklassenmethode durch eine Unterklassenmethode mit gleichem Namen und identischer Parameterliste statt. Der wesentliche Unterschied zwischen den beiden Ersetzungstechniken zeigt sich dann, wenn ein Unterklassenobjekt über eine Referenzvariable vom *Basisklassentyp* angesprochen wird:

- Bei verdeckenden Methoden kommt die *Basisklassenvariante* zum Einsatz (frühe Bindung).
- Bei überschreibenden Methoden wird die *Unterklassenvariante* benutzt (späte bzw. dynamische Bindung).

Bei klassenbezogenen Methoden kommt das späte Binden bzw. Überschreiben nicht in Frage.

Kapitel 7 (Typgenerisches Programmieren und Kollektionen)

Aufgabe 1

Sie finden einen Lösungsvorschlag in:

...\BspUeb\Typgenerizität und Kollektionen\PersonenListe

Über die Aufgabenstellung hinausgehend enthält der Lösungsvorschlag eine Erweiterung der Klasse `Person` um die Methode **CompareTo(**`Person p`**)** und die somit gerechtfertigte Zusicherung, das Interface **IComparable**<`Person`> zu erfüllen (siehe Kopf der Klassendefinition):

```
using System;
class Person : IComparable<Person> {
   public string Vorname;
   public string Name;
   public Person(string vorname, string nachname) {
      Vorname = vorname;
      Name = nachname;
   }
   public int CompareTo(Person p) {
      int vergl = (this.Name+this.Vorname).CompareTo(p.Name+p.Vorname);
      if (vergl < 0)
         return -1;
      else
         if (vergl == 0)
            return 0;
         else
            return 1;
   }
}
```

Infolgedessen können die Elemente des **List**<`Person`> - Objekts sogar sortiert werden. Weitere Informationen über Schnittstellen (*Interfaces*) folgen in Kapitel 8).

Aufgabe 2

Ein typisches Ergebnis (gemessen auf einem Rechner unter Windows 10 mit Intel-CPU Core i3 550):

```
Zeit in Millisek. für List<int>: 10,0138
Zeit in Millisek. für ArrayList: 98,089
```

Das zugehörige Programm finden Sie in:

...\BspUeb\Typgenerizität und Kollektionen\Listenwerte

Kapitel 8 (Interfaces)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Interfaces\Bruch

Aufgabe 2

Leicht vereinfachend kann man die wesentlichen Unterschiede so beschreiben:

- **Bestandteile**
  Eine abstrakte Klasse enthält mindestens *eine* abstrakte Methode und ansonsten beliebige Klassen-Member. Demgegenüber enthält ein Interface ausschließlich abstrakte Methoden, Eigenschaften, Indexer und Ereignisse, wobei das Schlüsselwort **abstract** im Kopf der Methodendefinitionen ebenso überflüssig wie verboten ist. Außerdem sind bei einem Interface Konstruktoren, Felder und statische Member verboten.
- **Abstammungsverhältnisse (Typkompatibilitäten)**
  Eine Klasse kann nur *eine* abstrakte Basisklasse besitzen, aber beliebig viele Interfaces implementieren.

Aufgabe 3

Weil Interfaces auch als Datentypen taugen und eine Klasse mehrere Interfaces implementieren darf, sind ihre Objekte zu mehreren Datentypen kompatibel. Eine Klasse „erbt“ allerdings nichts von den Schnittstellen, sondern sie gibt Verpflichtungserklärungen ab und muss die entsprechenden Implementierungs-Leistungen erbringen.

Kapitel 9 (Delegaten und Ereignisse)

Aufgabe 1

1. **Falsch**
   Ein Ereignis ist ein (statisches) Feld mit einem Delegatentyp, weist aber Besonderheiten im Vergleich zu einer gewöhnlichen Delegatenvariablen auf.
2. **Stimmt**
   Damit die Variablen der ehemaligen Umgebung noch nach dem Ende der generierenden Methode noch verfügbar sind, erzeugt der Compiler ein Objekt zur Aufbewahrung.
3. **Stimmt**
4. **Falsch**
   Wie vorzugehen ist, wird in Abschnitt 9.2.3 erläutert.
5. **Falsch**
   Das ist Unsinn. Am Ende von Abschnitt 9.2.2 wird eine strikt zu vermeidende Speicherplatzverschwendung durch die unterlassene Aufhebung einer Registrierung beschrieben.

Aufgabe 2

Lösungsvorschlag:

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

class FindAll {
   static void Main() {
      var list = new List<String> {"Doro","Liselotte","Friedrich","Theo","Walter"};
      Console.WriteLine("Liste der Kurznamen mit max. 4 Zeichen:");
      var k4 = list.FindAll(s => s.Length <= 4 ? true : false);
      foreach (string s in k4)
         Console.WriteLine(s);
   }
}
```

Aufgabe 3

Im folgenden Lösungsvorschlag ist die Metamethode statisch realisiert:

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

class FindAll {
   static Predicate<String> ListLE(int k) {
      return (s => s.Length <= k ? true : false);
   }

   static void Main() {
      var list = new List<String> {"Doro","Liselotte","Friedrich","Theo","Walter"};
      for (int i = 4; i < 10; i++) {
         Console.WriteLine($"\nListe der Kurznamen mit max. {i} Zeichen:");
         var k = list.FindAll(ListLE(i));
         foreach (string s in k)
            Console.WriteLine(s);
      }
   }
}
```

Kapitel 10 (Sonstige C# - Sprachbestandteile)

Aufgabe 1

Wie bei jedem Ausdruck mit binärem Operator wird im folgenden Beispiel

```
ass?[1]?.Length == ++res
```

zunächst der linke Operand des Identitätsoperators ausgewertet. Wenn die Variable `ass` ins Leere zeigt, oder das Element 1 des **String**-Arrays fehlt, kann die linke Seite dank der Null-bedingten Operatoren trotzdem fehlerfrei ausgewertet werden (resultierender Wert: **null**). Folglich wird anschließend auch die rechte Seite des Identitätsoperators ausgewertet.

Kapitel 11 (Einstieg in die GUI-Programmierung mit WPF)

Aufgabe 1

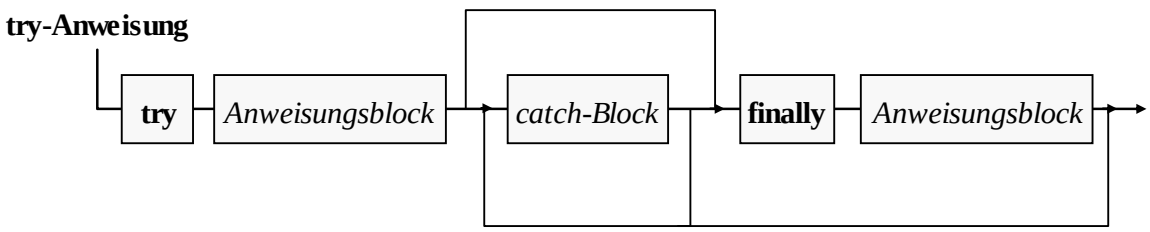
Sie finden einen Lösungsvorschlag im Verzeichnis:

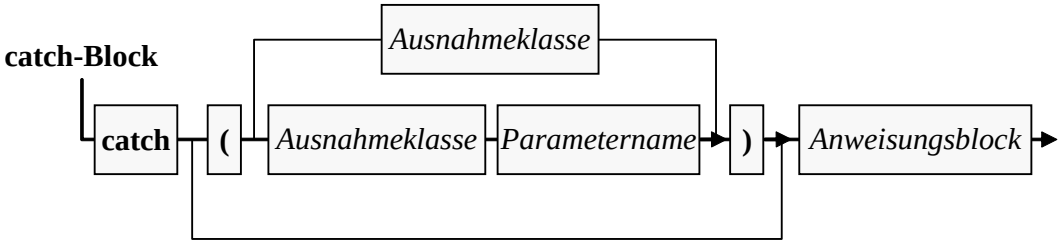
...\BspUeb\WPF\Eurokonverter

Kapitel 12 (Ausnahmebehandlung)

Aufgabe 1

Lösungsvorschlag:





Aufgabe 2

Die Klasse **OverflowException** stammt von der Klasse **ArithmeticException** ab. Weil die **Main()** - Methode der Klasse `Sequenzen` einen **ArithmeticException**-Handler besitzt, wird dort auch die **OverflowException** „behandelt“.

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmebehandlung\DuaLog\ArgumentOutOfRangeException

Aufgabe 4

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ausnahmebehandlung\EinfachStapelEx

Kapitel 13 (Attribute)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Attribute\NonsenseAttribute

Kapitel 14 (Ein- und Ausgabe über Datenströme)

Aufgabe 1

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Ein- und Ausgabe über Datenströme\Mittelwerte

Aufgabe 2

Das Einschalten der **AutoFlush**-Funktion ist bei einer Dateiausgabe in der Regel überflüssig und sehr zeitintensiv. Also sollte die folgende Zeile entfernt werden:

```
sw.AutoFlush = true;
```

Kapitel 15 (Multithreading)

Aufgabe 1

- Falsch
- Richtig
- Richtig
- Falsch

Aufgabe 2

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\Thread-Koordination\Signalisierungsobjekte\CountdownEvent

Aufgabe 3

Sie finden einen Lösungsvorschlag im Verzeichnis:

...\BspUeb\Multithreading\DirSize

Literatur

Agafonov, E. & Koryavchenko, A. (2015). *Mastering C# Concurrency*. Birmingham: Packt Publishing.

Albahari, J. (2010). *Threading in C#*. Online-Dokument: http://www.albahari.com/threading/.

Albahari, J. & Albahari, B. (2015). *C# 6.0 in a Nutshell* (6th ed.). Beijing: O'Reilly.

Baltes-Götz, B. (2003). *Einführung in das Programmieren mit Visual C++ 6.0*. Online-Dokument: http://www.uni-trier.de/index.php?id=22785

Baltes-Götz, B. & Götz, J. (2016). *Einführung in das Programmieren mit Java 8*. Online-Dokument: http://www.uni-trier.de/index.php?id=22787

Baltes-Götz, B. (2010). *Einführung in das Programmieren mit C# 3.0*. Online-Dokument: https://www.uni-trier.de/index.php?id=30454

Balzert, H. (2011). *Lehrbuch der Objektmodellierung: Analyse und Entwurf mit der UML 2*. Heidelberg: Spektrum.

Bayer, J. (2006). *Das C# 2005 Codebook*. München: Addison-Wesley.

Booch, G. et al. (2007). *Object-Oriented Analysis and Design with Applications* (3rd ed.). Boston, MA: Addison-Wesley.

Cleary, S. (2012). *Async and Await*. Online-Dokument: https://blog.stephencleary.com/2012/02/async-and-await.html

Cleary, S. (2014). *Concurrency in C#. Cookbook*. Sebastopol: CA O'Reilly.

Doberenz, W. & Gewinnus, T. (2010). *Datenbankprogrammierung mit Visual C# 2010*. Unterschleißheim: Microsoft Press.

Doberenz, W. & Gewinnus, T. (2015). *Visual C# 2015. Grundlagen, Profiwissen, Rezepte*. München: Hanser Verlag.

Ebner, M. (2000). *Delphi 5 Datenbankprogrammierung*. München: Addison-Wesley.

ECMA (2006). *C# Language Specification* (4th ed.). Online-Dokument: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-334.pdf

ECMA (2009). *Open XML Paper Specification*. Online-Dokument: http://www.ecma-international.org/publications/standards/Ecma-388.htm

Eller, F. & Kofler, M. (2005). *Visual C#*. München: Addison-Wesley.

Frischalowski, D. (2007). *Windows Presentation Foundation*. München: Addison Wesley.

Goll, J., Weiß, C. & Rothländer, P. (2000). *Java als erste Programmiersprache*. Stuttgart: Teubner

Griffiths, I. (2013). *Programming C# 5.0*. Beijing: O'Reilly.

Gunnerson, E. (2002). *C#* (2. Aufl.). Bonn: Galileo

Krüger, M. (2002). *C# Coding Style Guide*. (Version 0.3). Online-Dokument: http://www.icsharpcode.net/TechNotes/SharpDevelopCodingStyle03.pdf

Kühnel, A. (2010). *Visual C# 2010*. Bonn: Galileo Press. Auch als OpenBook verfügbar: http://www.galileocomputing.de/openbook/visual_csharp/

Lau, O. (2009). Faites vos jeux! Zufallszahlen erzeugen, erkennen und anwenden. *c't Magazin für Computertechnik*. 2009, Heft 2, 172-178.

Lammarsch, J. & Lammarsch, M. (2004). *C#*. Hannover: RRZN.

Lahres, B. & Rayman, G. (2009). *Praxisbuch Objektorientierung. Professionelle Entwurfsverfahren* (2. Aufl.). Bonn: Galileo

Lerman, J. (2010). *Programming Entity Framework* (2nd ed). Sebastopol, CA: O'Reilly Media.

Liberty, J. (2002). *Programming C#* (2nd ed.). Sebastopol, CA: O'Reilly.

Liskov, B. H. & Wing, J. M. (1999). *Behavioral Subtyping Using Invariants and Constraints*. Online-Dokument: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps

Louis, D. & Strasser, S. (2002). *C# in 21 Tagen*. München: Markt + Technik.

Louis, D. & Strasser, S. (2008). *Microsoft Visual C# 2008 - Das Entwicklerbuch*. Unterschleißheim: Microsoft Press Deutschland.

MacBeth, G. S. (2004). *C#. Programmer's Handbook*. New York: Springer.

MacDonald, M. (2012). *Pro WPF 4.5 in C#* (4rd ed.). New York, NY: Apress.

Microsoft Inc. (2012). *C# Language Specification 5.0*. Online-Dokument: https://www.microsoft.com/en-us/download/details.aspx?id=7029

Misner, S. (2007). *Microsoft SQL Server 2005 Express Edition. Start Now!* Redmond, WA: Microsoft Press.

Morrison, V. (2005a). *Concurrency: What Every Dev Must Know About Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Morrison, V. (2005b). *Memory Models: Understand the Impact of Low-Lock Techniques in Multithreaded Apps*. MSDN Magazine. August 2005. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Mössenböck, H. (2005). *Sprechen Sie Java? Einführung in das systematische Programmieren*. Heidelberg (3. Aufl.). Heidelberg, dpunkt.

Mössenböck, H. (2003). *Softwareentwicklung mit C#. Eine kompakter Lehrgang*. Heidelberg: dpunkt.

Mössenböck, H. (2016). *Kompaktkurs C# 6.0.* Heidelberg: dpunkt.

Müller, N. (2004). *Rechner-Arithmetik*. Online-Dokument: http://www.informatik.uni-trier.de/~mueller/Lehre/2005-arith/arith-2003-folien.pdf

Nathan, A. (2010). *WPF 4. Unleashed*. Indianapolis, IN: SAMS.

Petzold, C. (2008). *Foundations: Vector Graphics and the WPF Shape Class*. MSDN-Magazine. März 2008. Online-Dokument: http://msdn.microsoft.com/de-de/magazine/cc337899.aspx.

Richter, J. (2006). *Microsoft .NET Framework Programmierung in C#* (2. Aufl.). Unterschleißheim: Microsoft Press.

Richter, J. (2012). *CLR via C#*. Redmond, WA: Microsoft Press.

Sceppa, D. (2003). *Microsoft ADO.NET - Das Entwicklerhandbuch*. Unterschleißheim: Microsoft Press.

Schacherl, R. (2014). *Windows-8-Apps für C#-Entwickler*. Frankfurt a.M.: entwickler.press.

Sells, C & Griffiths, I. (2007). *Programming WPF* (2nd ed.). Sebastopol, CA: O'Reilly.

Spurgeon, C. E. (2000). *Ethernet. The Definitive Guide*. Sebastopol, CA: O'Reilly.

Strey, A. (2003). Computer-Arithmetik. Online-Dokument: http://www.informatik.uni-ulm.de/ni/Lehre/WS02/CA/CompArith.html

Troelsen, A. & Japikse, P. (2015). *C# 6.0 and the .NET 4.6 Framework*. New York, NY: Apress.

Stichwortregister

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z